# Parallel Computing project assignment 2019-2020

## Project Title: C++ / CUDA kernel image processing

**Student: Francesco Areoluci**

**Credits: 9 CFUs**

**Abstract**

*The purpose of the proposed project is to develop an application that can be used to perform image processing using the kernel convolution. Various kernel types have been developed in order to accomplish various purposes: from the image enhancing to the edge detection. The application is developed using two multithread approaches: C++ and CUDA.*

## 1 Introduction

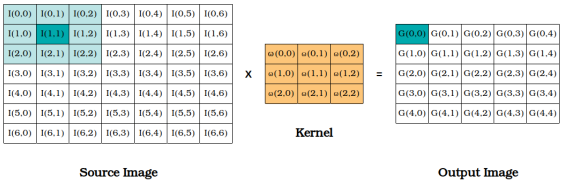
The developed application can be used to perform kernel convolution on a requested image.
The convolution is a mathematical operation which can be used to process two signals. In particular, its result express how the shape of a signal $f$ is modified by another signal $g$. The continuos, one dimensional, convolution operation between the signals $f(t)$ and $g(t)$ can be expressed as following:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

The extension to the two dimensional discreet values operation that can be used in image processing, is the following:

$$g(x,y) = \omega * f(x,y) = \\ = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy)f(x+dx,y+dy) \quad (1)$$

The term $f$ in eq. (1) is the source image, $\omega$ is the mask (or kernel) and $g$ is the output image.




The operation of kernel convolution will output a matrix smaller than the original one. To deal with this problem the original image has to be padded. There are many techniques that can be used to deal with padding, such as:

- **zero-padding**: pixels with zero values are used as padding
- **pixel replication**: the nearest border pixel is used as padding
- **pixel mirroring**: border pixels are mirrored in the padding

The number of operations of the convolution increases with the size of the kernel: the operation becomes onerous when large filters and/or higher resolution images are used.
A multithread approach can help in reducing the execution time of the operation by delegating threads in performing the convolution on subsets of the image. In the following chapters two multithread approaches will be presented: C++ and CUDA.

## 2 Sequential C++ application

The image kernel processing application has been firstly developed in C++ using a sequential approach.
The application executes the following steps:

1. Requested image is loaded in memory in grayscale, using the png++ library.
2. The kernel can be specified using the command line arguments. Various kernel types have been implemented, such as Gaussian, Laplacian and edge detector.
3. The loaded image is padded with respect to kernel size. The padding can be done with zero-padding and border pixels replication methods.
4. The convolution is then performed between the kernel matrix and the image matrix. For each source image pixel the convolution is performed by multiplying each pixel in the sliding window with the kernel coefficients. The sum of these values is the output pixel's value.

## 3 Multithread C++ application

The application performance can be improved by using a multithread approach.
The convolution algorithm depends on the source image pixel's values and the kernel coefficients. These values are only readed and not modified. Moreover, the output pixel values are written in the output matrix only once and only by a thread. Thus, this operation, is an embarassingly parallel problem: each thread can read, perform operation and write the output in memory without the need of synchronization with the other working threads.
The C++ language has introduced higher level threads abstraction in its version C++11 and offers the thread management via the STL library. Using this approach, threads can be created and joined by the forker process.
Each thread performs the convolution operation on a sequence of pixels of the source image. These pixels are loaded sequentially by each thread from the source image, which is loaded in memory as a linearized matrix.

### 3.1 Performance analysis

The sequential and multithread version of the application have been tested in order to evaluate the performances using a set of images in different resolutions and various kernel sizes, from 3x3 to 15x15. The comparison is done over the convolution operation, the image loading and padding are excluded. This is done to evaluate the filtering speedup.

**Table 1:** 3x3 kernel processing time (ms)

| Res | Seq | 2 ths | 4 ths | 8 ths |
|---|---|---|---|---|
| 1280x720 | 10.678 | 5.785 | 4.599 | 3.741 |
| 1920x1080 | 14.696 | 8.159 | 7.447 | 4.719 |
| 2560x1440 | 37.537 | 20.729 | 14.486 | 9.494 |
| 8120x4568 | 273.593 | 149.006 | 88.586 | 69.426 |

**Table 2:** 5x5 kernel processing time (ms)

| Res | Seq | 2 ths | 4 ths | 8 ths |
|---|---|---|---|---|
| 1280x720 | 22.791 | 12.127 | 8.423 | 7.355 |
| 1920x1080 | 38.767 | 21.326 | 15.835 | 8.866 |
| 2560x1440 | 80.746 | 42.924 | 29.915 | 17.904 |
| 8120x4568 | 702.248 | 367.885 | 203.287 | 148.137 |

**Table 3:** 7x7 kernel processing time (ms)

| Res | Seq | 2 ths | 4 ths | 8 ths |
|---|---|---|---|---|
| 1280x720 | 34.391 | 17.783 | 12.724 | 9.221 |
| 1920x1080 | 74.237 | 40.250 | 27.658 | 15.340 |
| 2560x1440 | 132.652 | 69.663 | 42.495 | 26.612 |
| 8120x4568 | 1337.175 | 688.930 | 371.958 | 260.692 |

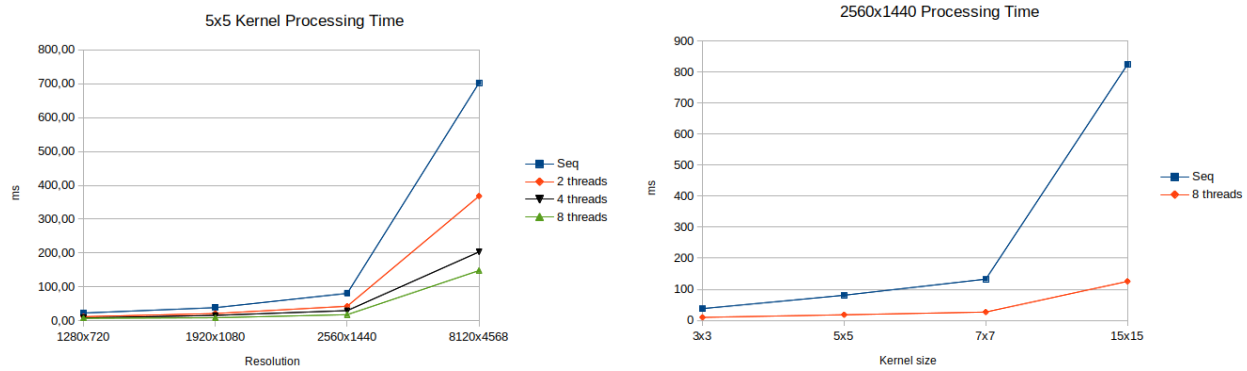



**Figure 1:** Convolution execution times on a 5x5 kernel with various resolutions (left) - Convolution execution times on a 2560x1440 image with various kernel sizes (right)

The tests have been executed on a Intel I7-7700HQ processor.
As we can see, the performances of the convolution operation increase with the number of threads and the performance improvements are more noticeable for higher kernel sizes and image resolutions. The following tables show the obtained speedup for the executed tests.

**Table 4:** 3x3 kernel processing time speedup

| Res | 2 threads | 4 threads | 8 threads |
|---|---|---|---|
| 1280x720 | 1.85 | 2.34 | 2.80 |
| 1920x1080 | 1.80 | 2.08 | 3.23 |
| 2560x1440 | 1.81 | 2.59 | 3.89 |
| 8120x4568 | 1.84 | 3.10 | 3.99 |

**Table 5:** 5x5 kernel processing time speedup

| Res | 2 threads | 4 threads | 8 threads |
|---|---|---|---|
| 1280x720 | 1.88 | 2.64 | 3.15 |
| 1920x1080 | 1.82 | 2.42 | 4.41 |
| 2560x1440 | 1.88 | 2.67 | 4.52 |
| 8120x4568 | 1.91 | 3.40 | 4.68 |

**Table 6:** 7x7 kernel processing time speedup

| Res | 2 threads | 4 threads | 8 threads |
|---|---|---|---|
| 1280x720 | 1.93 | 2.67 | 3.70 |
| 1920x1080 | 1.84 | 2.70 | 4.94 |
| 2560x1440 | 1.90 | 3.09 | 4.95 |
| 8120x4568 | 1.94 | 3.57 | 5.02 |

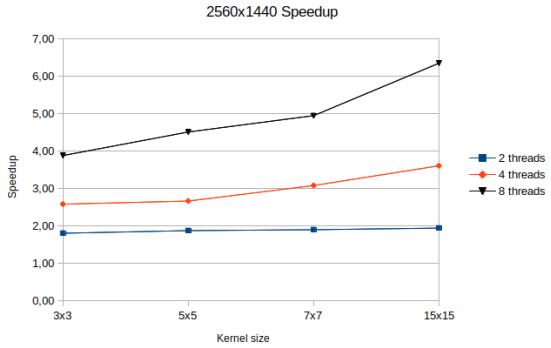



**Figure 2:** Total speed up using a 2560x1440 image

## 4 CUDA application

CUDA is a parallel computing platform that can be used in order to develop applications that can be executed on graphical processing units (GPUs). This platform enables the developer to speed up the parallelizable part of the computation of an application by using the high number of cores of a GPU.
A CUDA application is made of two parts:

- **Host code**: code that will be executed by the CPU
- **Device code**: code that will be executed by the GPU

The CUDA platform allows the developer to manage the memory allocation on the

GPU. In particular, there are three types of memory that can be used:

- **Global memory**: this is the main (and slower) memory of the device. The global memory is shared among all blocks of the grid. Threads can read/write from/to it.
- **Constant memory**: this type of memory is cached and shared among all blocks of the grid. The threads of the grid can only read from this memory. It is a small size memory and can be used to store variables that will not be modified during the processing.
- **Shared memory**: small size, low latency memory shared among all threads in a block. This memory can be used to store data that will be reused by the threads of the same block.

The device memory access is the bottleneck of a CUDA application. Thus, the memory management has to be correctly implemented in order to improve the application performance.

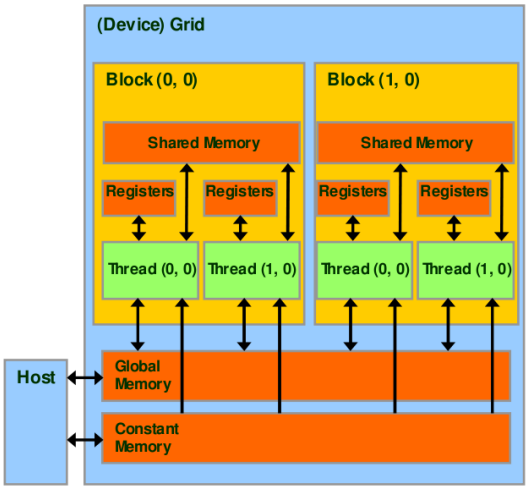



**Figure 3:** CUDA device memory model

The CUDA application performs the following operations:

1. Memory is allocated in the device to reserve the space for the source image, the filter and the output image
2. The source image and the filter are transferred from the host to the device
3. The CUDA kernel is executed to perform the convolution
4. Once the processing is executed, the output image is transferred back from the device to the host.

Three CUDA kernel have been developed and will be now described.

**Global memory kernel**
This kernel version uses only the global memory. The source image and the mask are transferred to the device using this memory. Once this process has been completed, each thread will evaluate the convolution for a pixel. The pixel's coordinates of the source image that will be accessed are evaluated using the thread and block index inside the grid.

**Constant memory kernel**
This kernel version uses the global memory to transfer the source image to the device, while the constant memory is used to store the mask coefficients, which will be only readed by the threads.
Each thread will access the constant memory, which is faster than the global memory, to get the filter cofficients.

**Shared + constant memory kernel**
This kernel version uses the constant memory as the previously described ones, but uses also the shared memory. Each thread of a block shares this memory. In order to do this, once the source image is loaded in the global memory, each block will load the corresponding source image tile of pixels into the shared memory.
Once this process is done, each thread of

the block will start the convolution operation on the loaded tile. The improvements will be noticeable as the mask size grows, because more pixels will be reused by the threads of the same block.

### 4.1 Performance analysis

The three version of the CUDA application have been tested to evaluate the performance improvement with respect to the C++ sequential version. Various image resolutions and kernel sizes have been used to test the application.
The following tables show the execution time of the filter processing, without the data transfer, for the three types of developed CUDA kernels: global memory, constant memory and shared + constant memory. All the tests have been executed on a NVIDIA 950m laptop GPU.

**Table 7:** 3x3 CUDA kernel processing time (ms), without data transfer

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 0.591 | 0.590 | 0.589 |
| 1920x1080 | 1.272 | 1.270 | 1.268 |
| 2560x1440 | 2.244 | 2.236 | 2.226 |
| 8120x4568 | 22.426 | 22.400 | 22.315 |

**Table 8:** 5x5 CUDA kernel processing time (ms), without data transfer

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 1.422 | 1.139 | 1.075 |
| 1920x1080 | 3.134 | 2.538 | 2.386 |
| 2560x1440 | 5.556 | 4.462 | 4.209 |
| 8120x4568 | 56.030 | 44.263 | 41.912 |

**Table 9:** 7x7 CUDA kernel processing time (ms), without data transfer

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 2.741 | 1.966 | 1.807 |
| 1920x1080 | 6.120 | 4.348 | 4.003 |
| 2560x1440 | 10.849 | 7.675 | 7.090 |
| 8120x4568 | 109.223 | 76.73 | 70.695 |

The following tables show the execution time of the filter processing, including the data transfer, for the three developed kernels.

**Table 10:** 3x3 CUDA kernel processing time (ms), with data transfer

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 3.068 | 2.466 | 2.519 |
| 1920x1080 | 6.124 | 4.983 | 4.645 |
| 2560x1440 | 10.068 | 8.293 | 8.257 |
| 8120x4568 | 83.356 | 75.143 | 75.058 |

**Table 11:** 5x5 CUDA kernel processing time (ms), with data transfer

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 3.798 | 3.102 | 2.933 |
| 1920x1080 | 7.998 | 6.341 | 5.989 |
| 2560x1440 | 12.436 | 10.228 | 9.868 |
| 8120x4568 | 115.413 | 98.251 | 94.647 |

**Table 12:** 7x7 CUDA kernel processing time (ms), with data transfer

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 4.678 | 3.868 | 3.587 |
| 1920x1080 | 10.001 | 7.765 | 7.365 |
| 2560x1440 | 17.377 | 13.614 | 12.660 |
| 8120x4568 | 162.648 | 129.092 | 124.205 |

The following tables show the speedup of the executed tests for each CUDA kernel version. The first term is the speedup without the data transfer, while the second is the speedup with the data transfer.

**Table 13:** 3x3 CUDA kernel speedup

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 18.07/3.48 | 18.11/4.33 | 18.13/4.24 |
| 1920x1080 | 11.56/2.40 | 11.57/2.95 | 11.59/3.16 |
| 2560x1440 | 16.73/3.73 | 16.79/4.53 | 16.86/4.55 |
| 8120x4568 | 12.20/3.28 | 12.21/3.64 | 12.26/3.65 |

**Table 14:** 5x5 CUDA kernel speedup

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 16.03/6.00 | 20.02/7.35 | 21.21/7.77 |
| 1920x1080 | 12.37/4.85 | 15.28/6.11 | 16.25/6.47 |
| 2560x1440 | 14.53/6.49 | 18.10/7.89 | 19.18/8.18 |
| 8120x4568 | 12.53/6.08 | 15.87/7.13 | 16.76/7.42 |

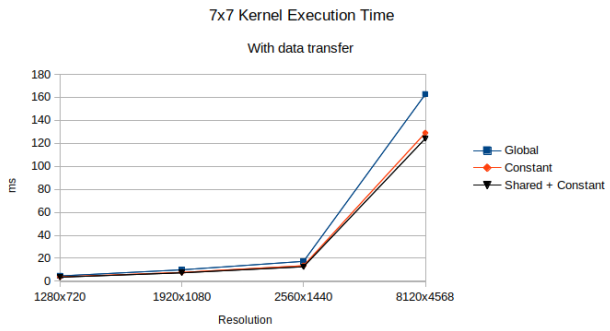



**Figure 4:** CUDA Convolution execution times on a 7x7 kernel with various resolutions without data transfer (left) and with data transfer (right)

**Table 15:** 7x7 CUDA kernel speedup

| Res | Global | Constant | Shared + Const |
|---|---|---|---|
| 1280x720 | 12.55/7.35 | 17.49/8.89 | 19.04/9.59 |
| 1920x1080 | 12.13/7.42 | 17.07/9.56 | 18.55/10.08 |
| 2560x1440 | 12.23/7.63 | 17.28/9.74 | 18.71/10.48 |
| 8120x4568 | 12.24/8.22 | 17.43/10.36 | 18.91/10.77 |

The speedup with respect to the C++ sequential version is noticeable and increases with higher mask sizes and image resolutions. The shared + constant version provides the higher speedup, while the global version provides the lower speedup. The execution time differences between the three version increase with higher mask size, as more pixels are reused and more mask coefficients are stored in constant memory. As we can see from the results, the data transfer from/to the device/host is very onerous, especially for higher resolution images.

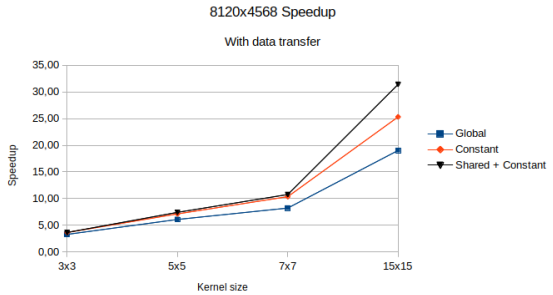



**Figure 5:** Total speed up using a 8120x4568 image, without data transfer

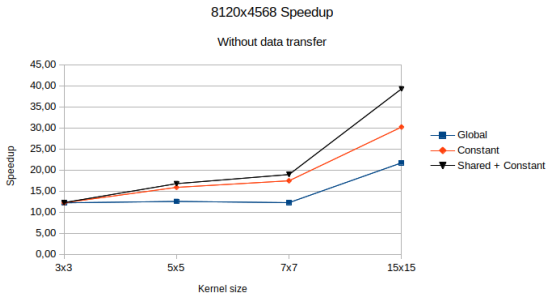



**Figure 6:** Total speed up using a 8120x4568 image, with data transfer

## Grid organization

The CUDA grid organization plays an important role in determining the performance of the executed kernel. In this case, the block dimensions (i.e. the number of threads per block) determine the number of blocks that will compose the grid in order to execute the convolution on each pixel of the source image. If each block has smaller dimension, an higher number of blocks has to be used in order to fill up the source image.
The following results show how the performance of the three kernel is affected by varying the block size.

**Table 16:** Global memory - 7x7 convolution (ms), various block sizes

| Image Resolution | 8x8 | 16x16 | 32x32 |
|---|---|---|---|
| 1280x720 | 3.31 | 2.90 | 2.74 |
| 1920x1080 | 7.42 | 6.51 | 6.12 |
| 2560x1440 | 13.35 | 11.38 | 10.85 |
| 8120x4568 | 134.21 | 117.14 | 109.22 |

**Table 17:** Constant memory - 7x7 convolution (ms), various block sizes

| Image Resolution | 8x8 | 16x16 | 32x32 |
|---|---|---|---|
| 1280x720 | 2.08 | 1.97 | 1.96 |
| 1920x1080 | 4.55 | 4.38 | 4.34 |
| 2560x1440 | 7.89 | 7.78 | 7.68 |
| 8120x4568 | 87.18 | 78.42 | 76.73 |

**Table 18:** Shared memory - 7x7 convolution (ms), various block sizes

| Image Resolution | 16x16 | 32x32 | 32x8 | 64x8 |
|---|---|---|---|---|
| 1280x720 | 2.26 | 2.28 | 2.06 | 1.81 |
| 1920x1080 | 5.04 | 5.02 | 4.60 | 4.00 |
| 2560x1440 | 8.94 | 8.93 | 7.98 | 7.09 |
| 8120x4568 | 91.17 | 90.64 | 80.54 | 70.69 |

As we can see from the results, the performance impact of the block size is more noticeable for higher resolution images.
The global memory kernel can benefit from the usage of larger blocks, while the constant memory kernel shows less peformance improvements.
The shared memory kernel uses a different approach in order to load in shared memory each tile. The padding size for each shared tile depends on the filter size. Thus, in order to be able to use various filter size, sub-blocks have been introduced: a thread in a block can load in shared memory and perform the convolution on more than one pixels. Using this approach the best performance can be obtained by using rectangular thread blocks.

## 5 Conclusions

The developed application can be used to perform image convolution with various types of kernels. The application has been built using the C++ language and the CUDA platform. The C++ multithread application shows performance improvements with respect to the sequential version, especially for larger kernel and higher resolution images. The CUDA application can be run on GPUs and provides higher performance on the convolution processing.
The application can be further improved by optimizing the padding process of the source image, which can be slow for higher mask sizes and image resolutions. Moreover, the separable kernel and Fast Fourier Transform-based convolution features can be added in order to provide faster image processing.