Kernel Processing

Parallel Computing

Luigi Ariano

luigi.ariano@stud.unifi.it

Edoardo Bonanni

edoardo.bonanni@stud.unifi.it

Master's Degree in Computer Engineering

Università degli Studi di Firenze

2022-2023

Abstract

The purpose of this project is to apply a Gaussian filter, or low pass, trying to parallelize this type of operation with various techniques. We propose **two sequential** and two **parallel implementations**, in order to measure how the parallel versions can improve the performances.

1 Introduction

Given an RGB image as input, in this project, we implemented two sequential programs and two parallel ones in order to calculate the corresponding modified image: the first parallel version we propose uses **PThread**, while the second one uses **CUDA**, as regards the two sequential versions, we developed two different types of implementation that more closely mirrored the parallel versions, both from the point of view of the structures used and from the implementation itself. In this way, we can estimate how the parallel implementations can improve the performances, measured in terms of mean execution time and speedup.

2 Kernel Image Processing

In this project, we have implemented the application of a Gaussian filter to an RGB image. In image processing, a Gaussian smoothing is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics context, typically to reduce image noise and reduce detail.

Basically, the visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen.

The Gaussian blur is a type of image-blurring filter that uses a Gaussian function for calculating the transformation to apply to each pixel in the image. The formula of a Gaussian function in two dimensions is:

$$w(x, y) = \frac{1}{2\pi\sigma^2}\mathbf{e}^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution.

Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function, in fact, values from this distribution are used to build a convolution matrix which is applied to the original image.

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

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2}$$

Considering the field of image processing, in which images are used as signals to be processed, the previous formula is extended by considering discrete and two-dimensional functions. Therefore, the processed image g in output is obtained through the convolution between a filter ω and the image f in input:

$$G(x, y) = \sum_{\alpha=-a}^{a}\sum_{\beta=-b}^{b} \omega(\alpha, \beta)f(x+\alpha, y+\beta) \tag{3}$$

Each pixel's new value is obtained from a weighted average of that pixel's neighborhood, where the central pixel's value receives the heaviest weight (having the highest Gaussian value) and neighboring pixels receive smaller weights as their distance to the original pixel increases.

So, since the Fourier transform of a Gaussian is another Gaussian, applying a Gaussian blur has the effect of reducing the image's high-frequency components; a Gaussian filter is thus a low-pass filter.

Since for each pixel of the image the surrounding pixels are also considered, a problem arises on the edges of the image. Therefore, there are several techniques to overcome this type of problem and obtain the correct output result:

- **omit missing pixels**: those that do not exist are not considered when averaging in a filter. The problem with this solution is that an ad hoc case study must be implemented for the edge pixels by choosing different weights on the basis of the number of existing image pixels;
- **zero-padding**: pixels with zero value are used as padding so that the filter is always contained in the image;
- **wrap around**: going to fill the missing values by making a circular transaction but introduces artifacts in the output image;
- **truncation**: the filter is applied only to the pixels that are not on the edges obtaining an output image smaller than the input image;
- **pixel replication**: the nearest edge pixel is used as padding, obtaining a good result and also avoiding artifacts in the output image.

For the above reasons, we have chosen to use the **pixel replication** technique in order to have an excellent result for the output image.

3 Implementation

In this section, we describe the sequential and parallel implementations of kernel image processing. First, we will show the implementations in C++ (sequential and PThread) where we have used a 3D matrix as image representation structure and then another (sequential and CUDA) in which the image to be processed is represented as a simple flat array. In figure 1, the input image used for this task is shown.

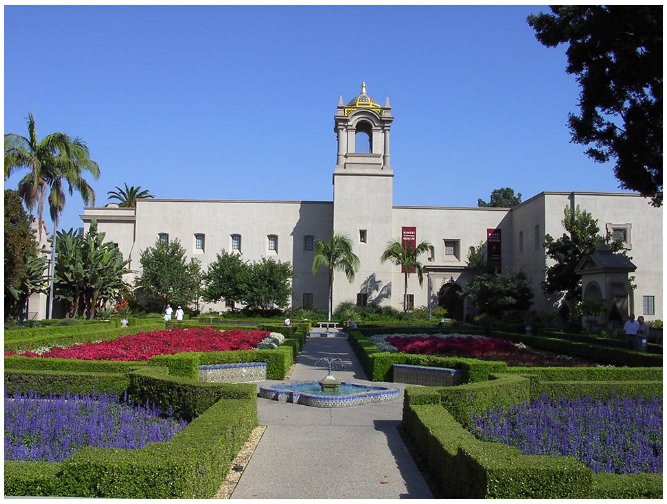

Figure 1: Input image.

3.1 AbstractKernel

```
class AbstractKernel {
protected:
    int kernelDimension = 0;
    int kernelSize = 0;
    float scalarValue = 0;
    float** kernel{};

public:
    virtual float** getKernel() = 0;
    virtual float* getFlatKernel() = 0;
    virtual int getKernelSize() = 0;
    virtual float getScalarValue() = 0;
    virtual int getKernelDimension() = 0;
};
```

A first fundamental aspect to consider concerns the definition of the kernel to be applied to the input image. Trying to obtain as much generality and structure as possible, we have chosen to create the abstract class **AbstractKernel** which contains the main features to be implemented in a kernel:

- the structure of the filter itself;
- dimension (width or height);
- size;
- scalar value to be multiplied by the filter coefficients.

From this class, it is possible to derive any type of filter with the relative features and methods necessary to use this tool correctly.

In particular, in our case, we have chosen to use a Gaussian-type filter of two dimensions 3x3 and 5x5, creating the relative coefficient matrices and setting the necessary features by deriving the previous abstract class.

In this way, these two Gaussian filters can be used correctly for the implementations made, having also created a method which flattens the matrix of the coefficients that is necessary for the second approach implemented. In fact, $getFlatKernel()$ method, which returns a flat version of the filter with respect to the classic 2D matrix used for the version with PThreads, is used in order to have congruence with the operations and structures implemented for the version in CUDA where the images are also stored in a flat array.

3.2 PaddedImage

```
vector<vector<vector<float>>> PaddedImage::createPaddedImage() {
    vector<vector<vector<float>>> imageWithPadding;
    imageWithPadding.resize(height);
    for (int i = 0; i < height; i++) {
        imageWithPadding[i].resize(width);
        for (int j = 0; j < width; j++) {
            imageWithPadding[i][j].resize(numChannels);
        }
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int index_i = i - padding;
            if (index_i < 0) {
                index_i = 0;
            } else if(index_i >= originalImage.rows) {
                index_i = originalImage.rows - 1;
            }
            int index_j = j - padding;
            if (index_j < 0){
                index_j = 0;
            } else if (index_j >= originalImage.cols) {
                index_j = originalImage.cols - 1;
            }
            Vec3b pixel = originalImage.at<Vec3b>(index_i,
                  index_j);
            imageWithPadding[i][j][0] = (float)pixel[2];
            imageWithPadding[i][j][1] = (float)pixel[1];
            imageWithPadding[i][j][2] = (float)pixel[0];
        }
    }
    return imageWithPadding;
}
```

The PaddedImage class was essential to implement for the first sequential and parallel version (PThread), as it allows to efficiently manage the structure of an image.

The OpenCV library is used to read the image by providing the path as input, to obtain its width, height, number of channels and size. Once read, a 3D matrix of vectors is created, one for each channel, containing the RGB values of the pixels. Being that the image needs padding to be processed correctly through a filter, we have also replicated, through the **pixel replication** technique, the rows and columns of the original image a number of times equal to the chosen padding. The padding, obviously, will depend on the size of the kernel used in order to have congruence between the size of the filter and the image itself.

3.3 Sequential implementation - PaddedImage

```
void sequential(const PaddedImage& image, AbstractKernel& kernel){
    float** kernelMatrix = kernel.getKernel();
    float scalarValue = kernel.getScalarValue();
    int kernelDimension = kernel.getKernelDimension();
    int padding = image.getPadding();
    int paddedWidth = image.getWidth();
    int paddedHeight = image.getHeight();
    vector<vector<vector<float>>> paddedImage =
          image.getPaddedImage();
    vector<vector<vector<float>>> blurredImage = paddedImage;

    // start convolution
    for (int i = padding; i < paddedHeight - padding; i++) {
        for (int j = padding; j < paddedWidth - padding; j++) {
            float newValueR = 0;
            float newValueG = 0;
            float newValueB = 0;
            for (int k = -padding; k < kernelDimension - padding;
                  k++) {
                for (int l = -padding; l < kernelDimension -
                      padding; l++) {
                    newValueR += paddedImage[i + k][j + l][0] *
                          kernelMatrix[k + padding][l + padding];
                    newValueG += paddedImage[i + k][j + l][1] *
                          kernelMatrix[k + padding][l + padding];
                    newValueB += paddedImage[i + k][j + l][2] *
                          kernelMatrix[k + padding][l + padding];
                }
            }
            blurredImage[i][j][0] = newValueR / scalarValue;
            blurredImage[i][j][1] = newValueG / scalarValue;
            blurredImage[i][j][2] = newValueB / scalarValue;
        }
    }
    blurredImage.clear();
}
```

This first sequential version is actually very simple as the convolution is literally performed between the coefficients of the chosen Gaussian filter and the corresponding RGB values of the pixels. Therefore, through the *for loop* present, it is possible to access directly both each pixel of the image and the relative surrounding pixels and the corresponding coefficients of the filter. The new calculated value for the central pixel, obtained via the weighted sum of the image values and the kernel coefficients, is assigned to the corresponding position of the output image: **blurredImage**. It is necessary to use the blurredImage as an output image, having the same structure as the input one, in order to be able to assign the new calculated values and not modify the pixel values of the original image which would lead to wrong calculations for the values of output, given that for each pixel its surroundings are also considered.

Obviously, we don't iterate on the padding pixels that we added to the original image but only on the real pixels of the input image, as they are used only as an contour to correctly calculate the border values and have no artifacts.

3.4 PThread implementation

```
vector<double> parallelPThread(int numThreads, const PaddedImage&
      image, AbstractKernel& kernel){
    float** kernelMatrix = kernel.getKernel();
    float scalarValue = kernel.getScalarValue();
    int kernelDimension = kernel.getKernelDimension();
    int padding = image.getPadding();
    int paddedWidth = image.getWidth();
    int paddedHeight = image.getHeight();
    vector<vector<vector<float>>> paddedImage =
          image.getPaddedImage();
    for(int nThread = 2; nThread <= numThreads; nThread+=2) {
        vector<vector<vector<float>>> blurredImage =
              image.getPaddedImage();
        vector<pthread_t> threads(nThread);
        vector<kernelProcessing_args> arguments(nThread);

        ... //implemented in 3 different versions

        blurredImage.clear();
        threads.clear();
        arguments.clear();
    }
}
```

POSIX Threads, commonly known as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

To concretely test this type of parallel approach, we decided to implement three types of code that allow to divide the image to be processed in different ways. Splitting the image in different ways, together with the number of threads, affects the execution time of the process as one approach can parallelize better than another for the chosen task.

Consider the three types of approaches chosen for splitting the original image into smaller sub-images based on the number of threads:

- **parallelPThreadRowsDivision**: the subdivision of the image takes place by rows based on the number of threads chosen. Another trivial case would have been choosing the subdivision by columns, but it would not have contributed to obtaining interesting results as the functioning would have been the same;
- **parallelPThreadRowsColumnsDivision**: the image is divided into both rows and columns, creating a grid with blocks of variable size based on the number of threads and the size of the image itself;
- **parallelPThreadBlocks**: in this case, the original image is divided into blocks with a size fixed a priori and distributed according to the number of threads chosen.

3.4.1 parallelPThreadRowsDivision

```
vector<double> parallelPThread(int numThreads, const PaddedImage&
      image, AbstractKernel& kernel){

    ...

    // start first implementation
    int chunkSizeHeight =
          floor((paddedHeight-(padding*2))/nThread);
    for (int t = 0; t < nThread - 1; t++) {
            arguments[t].paddedImage = &paddedImage;
            arguments[t].blurredImage = &blurredImage;
            arguments[t].kernelMatrix = kernelMatrix;
            arguments[t].width = paddedWidth;
            arguments[t].padding = padding;
            arguments[t].kernelDimension = kernelDimension;
            arguments[t].scalarValue = scalarValue;
            arguments[t].startIndex_i = chunkSizeHeight * t +
                  padding;
            arguments[t].endIndex_i = chunkSizeHeight * (t + 1) -
                  1 + padding;
            if (pthread_create(&threads[t], NULL, applyKernelRows,
                  (void *) &arguments[t]) != 0)
                cout << "Error" << endl;
        }
        arguments[nThread - 1].paddedImage = &paddedImage;
        arguments[nThread - 1].blurredImage = &blurredImage;
        arguments[nThread - 1].kernelMatrix = kernelMatrix;
        arguments[nThread - 1].width = paddedWidth;
        arguments[nThread - 1].padding = padding;
        arguments[nThread - 1].kernelDimension = kernelDimension;
        arguments[nThread - 1].scalarValue = scalarValue;
        arguments[nThread - 1].startIndex_i = chunkSizeHeight *
              (nThread - 1) + padding;
        arguments[nThread - 1].endIndex_i = paddedHeight - padding
              - 1;
        if (pthread_create(&threads[nThread - 1], NULL,
              applyKernelRows, (void *) &arguments[nThread - 1]) !=
              0)
            cout << "Error" << endl;

        for (auto thread: threads) {
            pthread_join(thread, NULL);
        }
        // end first implementation

        ...
}

void* applyKernelRows(void *args) {
    // apply filtering
    auto *arguments = (kernelProcessingRows_args*) args;
    for (int i = arguments->startIndex_i; i <=
          arguments->endIndex_i; i++) {
        for (int j = arguments->padding; j <
              arguments->width-arguments->padding; j++) {
            float newValueR = 0;
            float newValueG = 0;
            float newValueB = 0;
            for (int k = -arguments->padding; k <
                  arguments->kernelDimension-arguments->padding;
                  k++) {
                for (int l = -arguments->padding; l <
                      arguments->kernelDimension-arguments->padding;
                      l++) {
                    newValueR += (*arguments->paddedImage)[i + k][j
                          + l][0] * arguments->kernelMatrix[k +
                          arguments->padding][l +
                          arguments->padding];
                    newValueG += (*arguments->paddedImage)[i + k][j
                          + l][1] * arguments->kernelMatrix[k +
                          arguments->padding][l +
                          arguments->padding];
                    newValueB += (*arguments->paddedImage)[i + k][j
                          + l][2] * arguments->kernelMatrix[k +
                          arguments->padding][l +
                          arguments->padding];
                }
            }
            (*arguments->blurredImage)[i][j][0] = newValueR /
                  arguments->scalarValue;
            (*arguments->blurredImage)[i][j][1] = newValueG /
                  arguments->scalarValue;
            (*arguments->blurredImage)[i][j][2] = newValueB /
                  arguments->scalarValue;
        }
    }
    return (void*)("Done!");
}
```

In the first implementation developed, we thought of dividing the image by rows based on the height of the original image and the number of threads, creating a **chunk**. Hence, the size of a chunk is defined as follows: $chunkSizeHeight = floor((paddedHeight - (padding * 2))/nThread)$.

In practice, each thread must have the information necessary to apply the filtering, such as the reference of the original image and the new one in output, the Gaussian filter and the portion of the image that must be processed by that thread and defined by the start and end indices.

Everything is then partitioned for all (threads - 1) chosen, assigning the remaining portion of the image to the last thread since the chunks may not be sufficient to cover the entire size of the image.

Obviously, for each thread, the convolution operation between the pixel values assigned to it and the coefficients of the filter used will take place.

3.4.2 parallelPThreadRowsColumnsDivision

```
vector<double> parallelPThread(int numThreads, const PaddedImage&
      image, AbstractKernel& kernel){

        ...

        // start second implementation
        int threadRows;
        int threadColumns;
        if (nThread == 2 || nThread == 4 || nThread % 4 != 0) {
            threadRows = nThread / 2;
            threadColumns = nThread / threadRows;
        }
        else {
            threadColumns = nThread / 4;
            threadRows = nThread / threadColumns;
        }

        int chuckSizeHeight = floor((paddedHeight - (padding * 2))
              / threadRows);
        int chuckSizeWidth = floor((paddedWidth - (padding * 2)) /
              threadColumns);
        StartEndRowColIndices rowsColsIndices =
              createStartEndIndicesRowColForChunk(paddedWidth,
              paddedHeight, padding, chuckSizeWidth,
              chuckSizeHeight, threadRows, threadColumns);
        for (int t = 0; t < nThread; t++) {
            arguments[t].paddedImage = &paddedImage;
            arguments[t].blurredImage = &blurredImage;
            arguments[t].kernelMatrix = kernelMatrix;
            arguments[t].padding = padding;
            arguments[t].kernelDimension = kernelDimension;
            arguments[t].scalarValue = scalarValue;
            arguments[t].startIndex_i =
                  rowsColsIndices.startIndex_i[t];
            arguments[t].endIndex_i =
                  rowsColsIndices.endIndex_i[t];
            arguments[t].startIndex_j =
                  rowsColsIndices.startIndex_j[t];
            arguments[t].endIndex_j =
                  rowsColsIndices.endIndex_j[t];
            if (pthread_create(&threads[t], NULL,
                  applyKernelRowsColumns, (void *) &arguments[t])
                  != 0)
                cout << "Error" << endl;
        }

        for (auto thread: threads) {
            pthread_join(thread, NULL);
        }
        // end second implementation

        ...
}

void* applyKernelRowsColumns(void *args) {
    // apply filtering
    auto *arguments = (kernelProcessing_args*) args;
    for (int i = arguments->startIndex_i; i <=
         arguments->endIndex_i; i++) {
        for (int j = arguments->startIndex_j; j <=
             arguments->endIndex_j; j++) {

            ... // same as applyKernelRows

        }
    }
    return (void*)("Done!");
}
```

In the second implementation, the splitting of the image is done by both rows and columns, creating a grid of pixels into which the image is divided. In this case, chunks with a width, height based on the dimensions of the original image and the number of threads are considered: chunk dimensions are defined as follows: $chuckSizeHeight = floor((paddedHeight - (padding * 2))/threadRows)$, $chuckSizeWidth = floor((paddedWidth - (padding * 2))/threadColumns)$.

The *threadRows* and *threadColumns* variables represent how many threads are distributed between the height and width of the image, trying to divide these as much as possible between the two dimensions, in order to obtain fairly compact blocks and a better subdivision of the image.

Also in this case, the convolution operation is performed between the pixel values assigned to the executing thread and the coefficients of the used filter.

3.4.3 parallelPThreadBlocks

```
vector<double> parallelPThread(int numThreads, const PaddedImage&
      image, AbstractKernel& kernel){

        ...

        // start third implementation
        int blockRows;
        int blockColumns;
        if(paddedHeight % block_dim == 0)
            blockRows = paddedHeight / block_dim;
        else
            blockRows = (int)(trunc(paddedHeight / block_dim)) + 1;
        if(paddedWidth % block_dim == 0)
            blockColumns = paddedWidth / block_dim;
        else
            blockColumns = (int)(trunc(paddedWidth / block_dim)) +
                  1;
        int numBlocks = blockRows * blockColumns;
        if(numBlocks >= nThread) {
            vector<StartEndBlockIndices> threadBlockIndices =
                  createStartEndIndicesBlockForChunk(nThread,
                  block_dim, paddedWidth, paddedHeight, padding);
            for (int t = 0; t < nThread; t++) {
                arguments[t].paddedImage = &paddedImage;
                arguments[t].blurredImage = &blurredImage;
                arguments[t].kernelMatrix = kernelMatrix;
                arguments[t].padding = padding;
                arguments[t].kernelDimension = kernelDimension;
                arguments[t].scalarValue = scalarValue;
                arguments[t].threadBlocksIndices =
                      threadBlocksindices[t];
                if (pthread_create(&threads[t], NULL,
                      applyKernelBlocks, (void *) &arguments[t]) !=
                      0)
                    cout << "Error" << endl;
            }

            for (auto thread: threads) {
                pthread_join(thread, NULL);
            }
        }
        // end third implementation

        ...
}

void* applyKernelBlocks(void *args) {
    // apply filtering
    auto *arguments = (kernelProcessingBlocks_args*) args;
    int numbBlocks =
          (int)arguments->threadBlocksIndices.startIndex_i.size();
    StartEndBlockIndices threadBlocks =
          arguments->threadBlocksIndices;
    for (int b = 0; b < numbBlocks; b++){
        for (int i =
              arguments->threadBlocksIndices.startIndex_i[b]; i <=
              arguments->threadBlocksIndices.endIndex_i[b]; i++) {
            for (int j =
                  arguments->threadBlocksIndices.startIndex_j[b]; j
                  <= arguments->threadBlocksIndices.endIndex_j[b];
                  j++) {

                ... // same as applyKernelRows and
                    applyKernelRowsColumns

            }
        }
    }
    return (void*)("Done!");
}
```

In the last type of implementation, we thought of dividing the image into blocks of pixels with a fixed size and distributing them among the chosen number of threads.

Initially, the number of blocks present between the rows and columns is considered through a simple division between the height/width of the image and the size of the chosen block, obviously checking that the size of the block is congruous with that of the image.

Once the number of blocks present in which the image is divided has been calculated, they are distributed among the number of threads defined in order to correctly parallelize the entire process and, to do that, it's calculated the number of blocks that cover the entire image by using this formula: $numBlocks = blockRows * blockColumns$.

Once this value has been calculated, the number of blocks is uniformly distributed among the selected threads to avoid that one thread takes more blocks than the others and the cases in which the number of threads is greater than the number of blocks found are not considered because we would have unused threads.

Naturally, for the last few blocks that constitute the final pixels of the image, it's necessary to check that you are not processing the padding pixels or exceeding the image size.

In the end, we have only considered the cases in which the number of threads is greater than or equal to the number of blocks into which the image to be processed is divided since, vice versa, it would be useless to have threads that do not process any part of the input image.

3.5 FlatPaddedImage

The FlataddedImage class was essential to implement for the second sequential and parallel version (CUDA), as it allows to efficiently manage the structure of an image.

After reading the image through the OpenCV library and creating a 3D matrix related to the RGB channels also containing the padding pixels (same as section 3.2), it is necessary to build the flat array keeping the added padding pixels. Therefore, by iterating the matrix by rows, the values of the image pixels for the three RGB channels are accessed, creating an array in which these values are consecutively stored for each picture element of the image.

Based on the position of the pixel in the array, it is still possible to understand whether the pixel being considered is part of the original image or the padding.

```
float* FlatPaddedImage::createFlatPaddedImage() {
    int paddedWidth = originalWidth + (padding * 2);
    int paddedHeight = originalHeight + (padding * 2);
    vector<vector<vector<float>>> imageWithPadding;
    imageWithPadding.resize(paddedHeight);
    for (int i = 0; i < paddedHeight; i++) {
        imageWithPadding[i].resize(paddedWidth);
        for (int j = 0; j < paddedWidth; j++) {
            imageWithPadding[i][j].resize(numChannels);
        }
    }
    for (int i = 0; i < paddedHeight; i++) {
        for (int j = 0; j < paddedWidth; j++) {
            int index_i = i - padding;
            if (index_i < 0) {
                index_i = 0;
            } else if(index_i >= originalImage.rows) {
                index_i = originalImage.rows - 1;
            }
            int index_j = j - padding;
            if (index_j < 0){
                index_j = 0;
            } else if (index_j >= originalImage.cols) {
                index_j = originalImage.cols - 1;
            }

            Vec3b pixel = originalImage.at<Vec3b>(index_i,
                  index_j);
            imageWithPadding[i][j][0] = (float)pixel[2];
            imageWithPadding[i][j][1] = (float)pixel[1];
            imageWithPadding[i][j][2] = (float)pixel[0];
        }
    }
    int size = paddedWidth * paddedHeight * numChannels;
    float* flatImage = new float[size];
    for(int i=0; i < paddedHeight; i++) {
        for (int j = 0; j < paddedWidth; j++) {
            flatImage[(i * paddedWidth * numChannels) + (j *
                  numChannels)] = imageWithPadding[i][j][0];
            flatImage[(i * paddedWidth * numChannels) + (j *
                  numChannels) + 1] = imageWithPadding[i][j][1];
            flatImage[(i * paddedWidth * numChannels) + (j *
                  numChannels) + 2] = imageWithPadding[i][j][2];
        }
    }
    return flatImage;
}
```

3.6 Sequential implementation - FlatPaddedImage

In this sequential version, the pixels of the original image are iterated and the filter is applied with the kernel centered on the considered pixel (as in the sequential version described in the section 3.3), but using a flat array instead of a matrix both for the image and filter.

The new values of the 3 RGB channels calculated through the filter coefficients applied on neighboring pixels will be assigned to the corresponding and correct position in the output image, i.e. **flatBlurredImage**. The main difference compared to the previous sequential version lies in the access to the RGB channels of a pixel which in this case are consecutive with respect to being represented in 3 different vectors relating to the same position of the pixel in the image.

In this case, as in the several CUDA versions that we have implemented, the kernel also has a flat structure to simply have congruence with the structures used with the images.

```
void sequential(float* flatPaddedImage, int originalWidth, int
    originalHeight, int numChannels, int padding, float*
    flatBlurredImage, float* gaussianKernel, int kernelDim,
    float scalarValue) {
  unsigned int maskIndex;
  unsigned int pixelPos;
  unsigned int outputPixelPos;

  // start convolution
  for (int i = 0; i < (originalHeight + 2*padding); i++){
      for (int j = 0; j < (originalWidth + 2 * padding); j++){
          if (j > 1 && j < (originalWidth + padding) && i > 1 &&
               i < (originalHeight + padding)) {
              float pixValR = 0;
              float pixValG = 0;
              float pixValB = 0;
              for(int k = -padding; k < kernelDim - padding; k++)
                  {
                  for(int l = -padding; l < kernelDim - padding;
                      l++) {
                      pixelPos = ((i + k) * (originalWidth +
                          2*padding) * numChannels) + ((j + l) *
                          numChannels);
                      maskIndex = (k + padding) * kernelDim + (l +
                          padding);
                      pixValR += flatPaddedImage[pixelPos] *
                          gaussianKernel[maskIndex];
                      pixValG += flatPaddedImage[pixelPos + 1] *
                          gaussianKernel[maskIndex];
                      pixValB += flatPaddedImage[pixelPos + 2] *
                          gaussianKernel[maskIndex];
                  }
              }
              outputPixelPos = (i * (originalWidth + 2*padding) *
                  numChannels) + (j * numChannels);
              flatBlurredImage[outputPixelPos] = pixValR /
                  scalarValue;
              flatBlurredImage[outputPixelPos + 1] = pixValG /
                  scalarValue;
              flatBlurredImage[outputPixelPos + 2] = pixValB /
                  scalarValue;
          }
      }
  }
}
```

3.7 Cuda Implementation

CUDA (or Compute Unified Device Architecture), created by NVidia, 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 typical CUDA program structure consists of five main steps:

1. allocate CPU and GPU memories;

2. copy data (in this case input image, output image and filter) from CPU memory (host) to GPU mem-

ory (device)

3. invoke the CUDA functions, called *kernel*, to perform program-specific computation;
4. copy data (output image) back from GPU memory to CPU memory;
5. destroy GPU memories.

The CUDA platform allows the developer to manage the memory allocation on the GPU. In particular, there are various types of memory that can be used:

- local **registers** per thread;
- a parallel data cache or **shared**;
- a read-only **constant** cache that is shared by all the threads;
- a read-only texture cache (**global**) that is shared by all the processors;
- a **local** cached memory like registers:

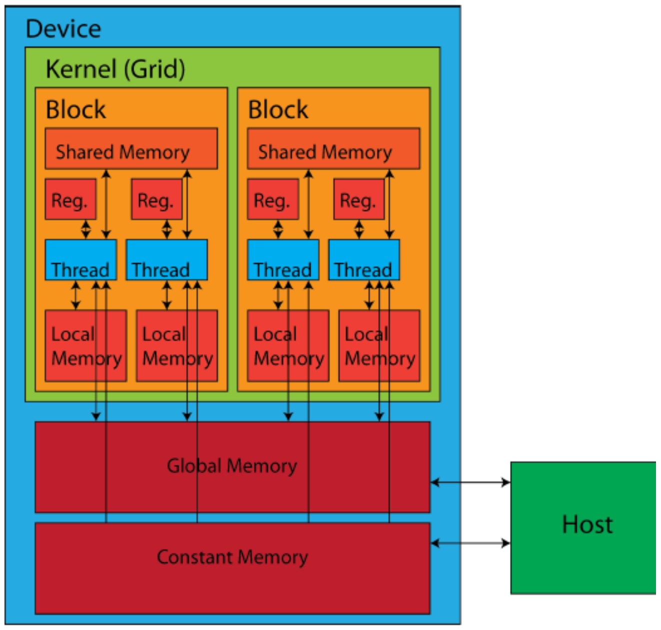

Figure 2: Cuda Memories.

Variable Declaration	Memory	Scope	Lifetime	Performance Penalty
int localVar	register	thread	thread	1x
int localArray[10]	local	thread	thread	100x
__shared__ int sharedVar	shared	block	block	1x
__device__ int globalVar	global	grid	application	100x
__constant__ int constantVar	constant	grid	application	1x

Table 1: Table shows the variables that can be instantiated through CUDA, the memory in which they are stored, their scope, lifetime and the performance reduction due to local/global memory usage.

To represent arrays not local in CUDA, three types of memories can be used:

- **Global**: 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**: 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**: 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.

Therefore, three different implementations have been developed in CUDA which differ from the type of memory where the input/output image and the filter are stored. In particular, in the first version we have the images and the filter both in global memory, the second differs from the previous one in storing the filter in constant memory, and the third keeps the filter in constant memory but uses the shared memory to save a portion of input image data which will be reused by threads in the same block.

3.7.1 Image and Filter Global

```
void CUDAPreparationGlobal(int numBlocks, const FlatPaddedImage&
    paddedImage, AbstractKernel& kernel) {
   for (int blockDimension = 2; blockDimension <= numBlocks;
        blockDimension *= 2) {
      float *flatPaddedImage;
      float *flatPaddedImage_device;
      float *flatBlurredImage;
      float *flatBlurredImage_device;
      float *gaussianKernel;
      float *gaussianKernel_device;

      // get originalWidth. originalHeight, numChannels,
           padding, paddedSize, kernelDim, kernelSize,
           scalarValue

      // allocate host memory
      checkCudaGlobal(cudaMallocHost((void **) &flatPaddedImage,
           sizeof(float) * paddedSize));
      checkCudaGlobal(cudaMallocHost((void **)
           &flatBlurredImage, sizeof(float) * paddedSize));
      checkCudaGlobal(cudaMallocHost((void **) &gaussianKernel,
           sizeof(float) * kernelSize));

      flatPaddedImage = paddedImage.getFlatPaddedImage();
      gaussianKernel = kernel.getFlatKernel();

      // allocate device memory
      checkCudaGlobal(cudaMalloc((void **)
           &flatPaddedImage_device, sizeof(float) * paddedSize));
      checkCudaGlobal(cudaMalloc((void **)
           &flatBlurredImage_device, sizeof(float) *
           paddedSize));
      checkCudaGlobal(cudaMalloc((void **)
           &gaussianKernel_device, sizeof(float) * kernelSize));

      // transfer data from host to device memory
      checkCudaGlobal(cudaMemcpy(flatPaddedImage_device,
           flatPaddedImage, sizeof(float) * paddedSize,
           cudaMemcpyHostToDevice));
      checkCudaGlobal(cudaMemcpy(flatBlurredImage_device,
           flatBlurredImage, sizeof(float) * paddedSize,
           cudaMemcpyHostToDevice));
      checkCudaGlobal(cudaMemcpy(gaussianKernel_device,
           gaussianKernel, sizeof(float) * kernelSize,
           cudaMemcpyHostToDevice));
        // define DimGrid and DimBlock
        dim3 DimGrid((int) ceil((float) (originalWidth + (padding
             * 2)) / (float) blockDimension), (int) ceil((float)
             (originalHeight + (padding * 2)) / (float)
             blockDimension), 1);
        dim3 DimBlock(blockDimension, blockDimension, 1);

        // start global convolution
        global_kernel_convolution_3D<<<DimGrid,
             DimBlock>>>(flatPaddedImage_device, originalWidth,
             originalHeight, numChannels, padding,
             flatBlurredImage_device, gaussianKernel_device,
             kernelDim, scalarValue);
        cudaDeviceSynchronize();

        // transfer data back to host memory
        checkCudaGlobal(cudaMemcpy(flatBlurredImage,
             flatBlurredImage_device, sizeof(float) * paddedSize,
             cudaMemcpyDeviceToHost));

        // deallocate device memory
        checkCudaGlobal(cudaFree(flatPaddedImage_device));
        checkCudaGlobal(cudaFree(flatBlurredImage_device));
        checkCudaGlobal(cudaFree(gaussianKernel_device));

        // free host memory
        cudaFreeHost(flatPaddedImage);
        cudaFreeHost(flatBlurredImage);
        cudaFreeHost(gaussianKernel);
        cudaDeviceReset();
    }
}

__global__ void global_kernel_convolution_3D(float*
     flatPaddedImage, int originalWidth, int originalHeight, int
     numChannels, int padding, float* flatBlurredImage, float*
     gaussianKernel, int kernelDim, float scalarValue) {
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int maskIndex;
    unsigned int pixelPos;
    unsigned int outputPixelPos;

    // start convolution
    if (x > 1 && x < (originalWidth + padding) && y > 1 && y <
         (originalHeight + padding)) {
        float pixValR = 0;
        float pixValG = 0;
        float pixValB = 0;
        for(int k = -padding; k < kernelDim - padding; k++) {
            for(int l = -padding; l < kernelDim - padding; l++) {
                pixelPos = ((y + k) * (originalWidth + 2*padding) *
                      numChannels) + ((x + l) * numChannels);
                maskIndex = (k + padding) * kernelDim + (l +
                      padding);
                pixValR += flatPaddedImage[pixelPos] *
                      gaussianKernel[maskIndex];
                pixValG += flatPaddedImage[pixelPos + 1] *
                      gaussianKernel[maskIndex];
                pixValB += flatPaddedImage[pixelPos + 2] *
                      gaussianKernel[maskIndex];
            }
        }
        outputPixelPos = (y * (originalWidth + 2*padding) *
             numChannels) + (x * numChannels);
        flatBlurredImage[outputPixelPos] = pixValR / scalarValue;
        flatBlurredImage[outputPixelPos + 1] = pixValG /
             scalarValue;
        flatBlurredImage[outputPixelPos + 2] = pixValB /
             scalarValue;
    }
}
```

Initially, it is essential to have a preparation phase where you do the preliminary operations before applying the convolution operation in CUDA.

First of all, it is necessary to allocate a certain space in the host memory for the three fundamental structures used for our task: flatPaddedImage, flatBlurredImage and gaussianKernel, by multiplying the size of these structures by the size of a float having these vectors stored float values.

After, it is also necessary to allocate the same space in the global (device) memory since the code will have to be executed through CUDA and the structures must be stored in the appropriate memory.

Once the necessary space has been allocated in both the mentioned memories, the data transfer of the three vectors, i.e. flatPaddedImage, flatBlurredImage and gaussianKernel, is applied from the host to the device memory in order to have all the data in the correct memory and perform the convolution from the code in CUDA.

Another fundamental aspect to consider concerns the size of the grid and of the blocks: *DimGrid*, *DimBlock*. The organization of these CUDA structures have an important role in determining the performance of the running kernel.

In this case, it's necessary to define the size of the *DimGrid* grid, i.e. the number of blocks, as the ratio (rounded upwards) between the width/height of the padded image and the size of the chosen block. Subsequently, *DimBlock* is defined, i.e. the number of threads per block, by setting the size of the block itself in advance (we have chosen several dimensions to test the effectiveness of the various methods implemented) and in order to have the number of threads/blocks that will form the grid to convolute on each pixel of the source image. Obviously, if each block has smaller dimension, a greater number of blocks must be used to fill the source image.

Moreover, as blocks are defined in CUDA, it's not possible to have more than 1024 threads per block which is why we chose a maximum size of 32 for our tests.

Finally, the CUDA convolution operation will be performed using the **__global__ void global_kernel_convolution_3D** function between the image pixels and the Gaussian filter coefficients chosen by the CUDA threads.

3.7.2 Image Global and Filter Constant

```
const unsigned int MAX_KERNEL_SIZE = 25;
__device__ __constant__ float
     gaussianKernel_device[MAX_KERNEL_SIZE];

void CUDAPreparationConstant(int numBlocks, const
     FlatPaddedImage& paddedImage, AbstractKernel& kernel) {
    for (int blockDimension = 2; blockDimension <= numBlocks;
         blockDimension *= 2) {

        // initialization same as global implementation

        float gaussianKernel[MAX_KERNEL_SIZE];

        // cudaMallocHost for flatPaddedImage and flatBlurredImage
             same as global implementation
        checkCudaConstant(cudaMallocHost((void **)
             &gaussianKernel, sizeof(float) * MAX_KERNEL_SIZE));

        for (int i = 0; i < MAX_KERNEL_SIZE; i++){
            gaussianKernel[i] = gaussianKernel_temp[i];
        }

        // allocate device memory same for flatPaddedImage and
             flatBlurredImage as global implementation
        // no device allocation for kernel because it's constant

        // transfer data from host to device memory same as global
              implementation

        // DimGrid and DimBlock same as global implementation

        // start constant convolution
        constant_kernel_convolution_3D<<<DimGrid,
              DimBlock>>>(flatPaddedImage_device, originalWidth,
              originalHeight, numChannels, padding,
              flatBlurredImage_device, kernelDim, scalarValue);
        cudaDeviceSynchronize();

        // transfer data back to host memory same as global
              implementation

        // deallocate device memory for flatPaddedImage and
              flatBlurredImage same as global implementation

        // no deallocation for gaussianKernel_device because it's
              constant

        // deallocate host memory and cuda reset same as global
              implementation
    }
}

__global__ void constant_kernel_convolution_3D(float*
      flatPaddedImage, int originalWidth, int originalHeight, int
      numChannels, int padding, float* flatBlurredImage, int
      kernelDim, float scalarValue) {
    ... // same as global_kernel_convolution_3D
        for(int k = -padding; k < kernelDim - padding; k++) {
            for(int l = -padding; l < kernelDim - padding; l++) {
                pixelPos = ((y + k) * (originalWidth + 2*padding) *
                      numChannels) + ((x + l) * numChannels);
                maskIndex = (k + padding) * kernelDim + (l +
                      padding);
                pixValR += flatPaddedImage[pixelPos] *
                      gaussianKernel_device[maskIndex];
                pixValG += flatPaddedImage[pixelPos + 1] *
                      gaussianKernel_device[maskIndex];
                pixValB += flatPaddedImage[pixelPos + 2] *
                      gaussianKernel_device[maskIndex];
            }
        }
    ... // same as global_kernel_convolution_3D
}
```

In the second implementation, the same memory allocation and data transfer operations are performed between the host and the device memory for the flatPaddedImage and flatBlurredImage images substantially replicating the operations described before. The difference with respect to the previous case concerns the management of the gaussianKernel, which is not transferred to the global memory but rather to the constant. This type of memory is cached and shared among all blocks of the grid, it has small size and can be used to store variables that will not be modified during the processing.

Therefore, it is used to store the mask coefficients that will be only read by the threads accessing the constant memory, which is faster than the global one to get the filter coefficients.

The *DimGrid*, *DimBlock* structures are defined in the same way as in the previous case, as the operation and access to the pixels is the same and, for how blocks are defined in CUDA, we chose a maximum size of 32 for blocks in our tests.

Finally, through the **__global__ void constant_kernel_convolution_3D** function, the convolution operation is performed with the pixel values stored in the global memory while the filter coefficients in the constant memory for faster access.

3.7.3 Image Shared and Filter Constant

```
const unsigned int max_kernel_size = 25;
__device__ __constant__ float
      gaussianKernel_device[max_kernel_size];

void CUDAPreparationShared(int numBlocks, const FlatPaddedImage&
      paddedImage, AbstractKernel& kernel) {
    for (int blockDimension = 2; blockDimension <= numBlocks;
          blockDimension *= 2) {
        if (blockDimension > 2 * paddedImage.getPadding()) {
            // initialization same as global/constant
                  implementation
            // kernel initialization same as constant
                  implementation

            const int tileWidth = blockDimension;
            const int tileHeight = blockDimension;
            const int blockWidth = tileWidth - 2 * padding;
            const int blockHeight = tileHeight - 2 * padding;

            // cudaMallocHost for flatPaddedImage,
                  flatBlurredImage, gaussianKernel same as constant
                  implementation (no device allocation for kernel)

            // allocate device memory same as global/constant
                  implementation

            // transfer data from host to device memory same as
                  global/constant implementation

            // define DimGrid and DimBlock
            dim3 DimGrid((int) ceil((float) (originalWidth +
                  (padding * 2)) / (float) blockWidth), (int)
                  ceil((float) (originalHeight + (padding * 2)) /
                  (float) blockHeight));
            dim3 DimBlock(tileWidth, tileHeight);

            // define shared memory size
            int sharedMemorySize = tileWidth * tileHeight *
                  numChannels * sizeof(float);

            // start shared convolution
            shared_kernel_convolution_3D<<<DimGrid, DimBlock,
                  sharedMemorySize>>>(flatPaddedImage_device,
                  originalWidth, originalHeight, numChannels,
                  padding, flatBlurredImage_device, kernelDim,
                  scalarValue);
            cudaDeviceSynchronize();

            // transfer data back to host memory same as
                  global/constant implementation

            // deallocate device memory same as constant
                  implementation (no deallocation for
                  gaussianKernel_device)

            // deallocate host memory and cuda reset same as
                  global/constant implementation
        }
    }
}

__global__ void shared_kernel_convolution_3D(float*
      flatPaddedImage, int originalWidth, int originalHeight, int
      numChannels, int padding, float* flatBlurredImage, int
      kernelDim, float scalarValue) {
    extern __shared__ float shared_data[];

    // define width, height for block and tile
    unsigned int blockWidth = blockDim.x - (2*padding);
    unsigned int blockHeight = blockDim.y - (2*padding);
    unsigned int tileWidth = blockDim.x;
    unsigned int tileHeight = blockDim.y;

    // define rows, columns start/end indices for block and tile
    unsigned int blockStartCol = blockIdx.x * blockWidth + padding;
    unsigned int blockEndCol = blockStartCol + blockWidth;
    unsigned int blockStartRow = blockIdx.y * blockHeight +
         padding;
    unsigned int blockEndRow = blockStartRow + blockHeight;
    unsigned int tileStartCol = blockStartCol - padding;
    unsigned int tileEndCol = blockEndCol + padding;
    unsigned int tileStartRow = blockStartRow - padding;
    unsigned int tileEndRow = blockEndRow + padding;
    unsigned int tilePixelPosCol = threadIdx.x;
    unsigned int pixelPosCol = tileStartCol + tilePixelPosCol;

    unsigned int tilePixelPosRow;
    unsigned int pixelPosRow;
    unsigned int pixelPos;
    unsigned int tilePixelPos;
    unsigned int maskIndex;
    unsigned int outputPixelPos;

    // number of blocks in tile
    unsigned int blocksInTile = (int)(ceil((float)tileHeight /
         (float)blockDim.y));

    // load pixel values in shared memory
    for (int b = 0; b < blocksInTile; b++) {
        tilePixelPosRow = threadIdx.y + b * blockDim.y;
        pixelPosRow = tileStartRow + tilePixelPosRow;
        // check if the pixel is in the image
        if (pixelPosCol < tileEndCol && pixelPosCol <
             (originalWidth + 2*padding) &&
            pixelPosRow < tileEndRow && pixelPosRow <
                 (originalHeight + 2*padding)) {
            pixelPos = (pixelPosRow * (originalWidth + 2*padding)
                 * numChannels) + (pixelPosCol * numChannels);
            tilePixelPos = (tilePixelPosRow * tileWidth *
                 numChannels) + (tilePixelPosCol * numChannels);
            shared_data[tilePixelPos] = flatPaddedImage[pixelPos];
            shared_data[tilePixelPos + 1] =
                 flatPaddedImage[pixelPos + 1];
            shared_data[tilePixelPos + 2] =
                 flatPaddedImage[pixelPos + 2];
        }
    }
    __syncthreads();

    // start convolution
    for (int b = 0; b < blocksInTile; b++) {
        tilePixelPosRow = threadIdx.y + b * blockDim.y;
        pixelPosRow = tileStartRow + tilePixelPosRow;
        if (pixelPosCol >= tileStartCol + padding && pixelPosCol <
             tileEndCol - padding && pixelPosCol < (originalWidth
             + 2*padding) &&
            pixelPosRow >= tileStartRow + padding && pixelPosRow <
                 tileEndRow - padding && pixelPosRow <
                 (originalHeight + 2*padding)) {
            float pixValR = 0;
            float pixValG = 0;
            float pixValB = 0;
            for (int k = -padding; k < kernelDim - padding; k++) {
                for (int l = -padding; l < kernelDim - padding;
                     l++) {
                    tilePixelPos = ((tilePixelPosRow + k) *
                         tileWidth * numChannels) +
                         ((tilePixelPosCol + l) * numChannels);
                    maskIndex = (k + padding) * kernelDim + (l +
                         padding);
                    pixValR += shared_data[tilePixelPos] *
                         gaussianKernel_device[maskIndex];
                    pixValG += shared_data[tilePixelPos + 1] *
                         gaussianKernel_device[maskIndex];
                    pixValB += shared_data[tilePixelPos + 2] *
                         gaussianKernel_device[maskIndex];
                }
            }
            outputPixelPos = (pixelPosRow * (originalWidth +
                 2*padding) * numChannels) + (pixelPosCol *
                 numChannels);
            flatBlurredImage[outputPixelPos] = pixValR /
                 scalarValue;
            flatBlurredImage[outputPixelPos + 1] = pixValG /
                 scalarValue;
            flatBlurredImage[outputPixelPos + 2] = pixValB /
                 scalarValue;
        }
    }
}
```

The latter implementation is different from the other two previously described. Initially, the same memory allocation and data transfer operations are performed for the input and output images, also allocating the gaussianKernel in the constant memory as described above.

In this version, pixel blocks of the original image are stored in shared memory, where each thread of a block shares this memory. To do this, once the source image has been loaded into the global memory, each block will load the corresponding pixel tile of the source image into the shared memory.

In this case, it is essential to define $DimGrid$, $DimBlock$ in a slightly different way given the type of approach: $DimGrid$ is defined in the same way as in the previous cases, but the variables $blockWidth$ and $blockHeight$ correspond to the pre-chosen size of the block subtracted the $padding*2$ value, instead, $DimBlock$ is defined exactly by the a priori dimensions of the blocks represented by the variables $tileWidth$, $tileHeight$.

This decision to differentiate the blocks between $blockWidth$, $blockHeight$ and $tileWidth$, $tileHeight$ is necessary in order to manage the image pixels in the simplest and most correct way possible. In this way, we divide the image into a suitable number of blocks, not considering the padding, but the number of threads per block must also take into account these border pixels which will be processed in the convolution operation.

This leads to a rather limited choice of block size given how they are defined $blockWidth$ and $blockHeight$, it is necessary to choose a size greater than the value of $padding*2$ in order not to have blocks of zero size. For the reasons above, the maximum block size is 32 since the number of threads per block cannot be greater than 1024.

Furthermore, it is necessary to define the size of the shared memory where the pixel values will be stored in the corresponding tiles. The size is defined by the following formula: $sharedMemorySize = tileWidth * tileHeight * numChannels * sizeof(float)$. This information needs to be passed to the convolution function in CUDA in order to know this size and dynamically allocate the space needed for the tiles.

Before performing the convolution operation, it is necessary to fill the tiles with the corresponding pixel portion of the input image, also considering the padding pixels and synchronizing the threads in order to complete this operation before starting the filtering.

At this point, by using **__global__ void shared_kernel_convolution_3D** function, the convolution operation is performed between the filter coefficients, stored in the constant memory, and the pixel values of the image in the corresponding tile stored in the shared memory.

4 Tests

All of the following tests are executed on a computer with the following specifications:

- OS: Microsoft Windows 10 Pro;
- CPU: 11th Gen Intel(R) Core(TM) i9-11900KF@3.5GHz (TBoost 5.1GHz) Octa-Core;
- GPU: GeForce RTX 3070Ti;
- RAM: 32GB DDR4 @3600 MHz.

Tests consider mean execution times which are calculated by repeating the computation of resulting image 100 times for each test image for every implementation proposed, using only a 5x5 Gaussian filter. In particular, the test images differs in size: we consider the same image scaled at different resolution 480p, 720p, 1080p, 2K, 4K where "p" means that the image has a vertical resolution of 480, 720 and so on. Regarding the parallel approaches, we repeat these test also changing the threads' number and the blocks dimension.

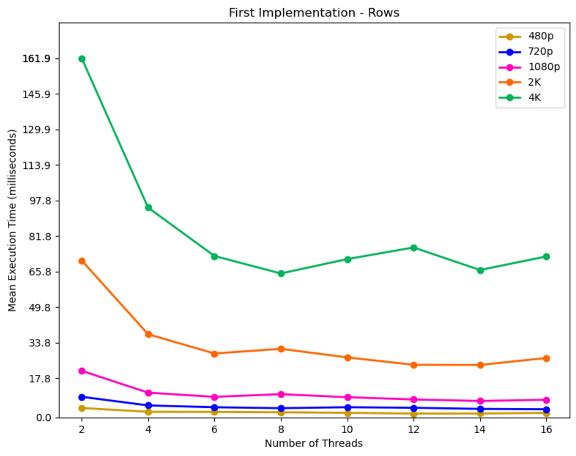

Figure 4: The mean execution times for the developed algorithm in PThread, first implementation, based on the number of threads and the size of the images are shown.

5 Results

In this section we discuss about the results achieved by the tests. As expected, in both cases, the results achieved following the parallel approaches are significantly better than the sequential ones. Regardless of the type of implementation used, sequential or parallel, in output we get a correctly blurred image shown in figure 3 using a 5x5 Gaussian filter for each test performed.

Figure 3: This figure shows the output image after gaussian filtering.

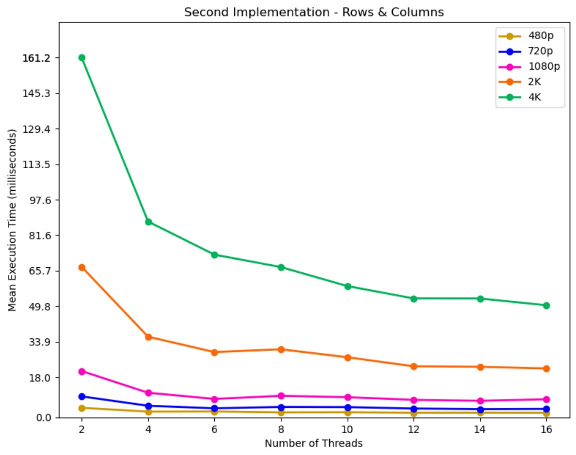

Figure 5: The mean execution times for the developed algorithm in PThread, second implementation, based on the number of threads and the size of the images are shown.

5.1 PThread - Implementation Results

In figure 4 is shown the mean execution times obtained with PThread first implementation, where the image is divided by rows, repeating tests on images with different size and varying the threads' number. As we can notice, obviously, the bigger the image the more the execution time increases with a decrease of this time as the number of threads increases and obtaining the best result with 14 threads.

Also as regards the second implementation in PThread (figure 5), where the image is divided into rows and columns, the behavior is similar to the previous case with an optimal result for 14/16 threads for the various input image sizes.

For the third implementation, instead, it is necessary to consider not only the mean execution time as the number of threads and the resolution of the input images vary, but also the effectiveness of this approach for blocks of different chosen sizes.

As also shown in the tables 2, 3, 4, 5 and 6, regardless of image resolution in input, the subdivision of the images into blocks of **16** and the number of threads equal to **14** turns out to be the best possible configuration, obtaining a lower execution time (figure 6).

Furthermore, we have seen that, among the three parallel PThread implementations, the best is the last one where the image is divided into blocks with a size fixed at 16, obtaining a clear improvement for input images with a big resolution as shown in figure 7.

480p	*Thread*							
Block Size	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
4	4.44	2.79	2.60	2.35	2.47	2.01	2.03	2.05
8	4.29	2.55	2.22	2.27	2.34	2.09	2.00	2.14
16	4.68	2.72	2.17	2.35	2.31	2.03	2.02	2.00
32	4.64	2.87	2.14	2.19	2.44	2.26	2.35	2.11
64	4.67	2.90	2.49	2.58	2.63	2.50	2.46	2.49
128	4.67	2.90	2.71	2.72	2.77	2.58	2.68	2.45
256	4.74	3.83	2.47	-	-	-	-	-

Table 2: Third PThread implementation: Gaussian kernel execution time (ms), for 480p resolution of the input image.

720p	*Thread*							
Block Size	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
4	10.11	5.53	4.30	4.60	4.68	4.35	4.22	4.25
8	9.61	5.38	4.68	4.70	4.64	4.41	4.19	3.92
16	9.88	5.59	4.20	4.66	4.73	4.25	3.96	4.02
32	9.86	5.42	4.30	4.89	4.66	4.27	4.22	4.04
64	10.28	5.77	5.12	5.08	5.26	4.71	4.89	4.69
128	10.45	5.97	4.86	5.44	5.31	5.32	4.80	4.47
256	10.66	5.91	4.81	5.21	5.30	4.30	-	-

Table 3: Third PThread implementation: Gaussian kernel execution time (ms), for 720p resolution of the input image.

1080p	*Thread*							
Block Size	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
4	22.45	11.91	10.11	10.72	10.39	9.10	8.59	8.96
8	21.10	11.24	8.67	9.89	9.31	8.37	8.26	8.12
16	21.22	11.33	8.97	9.62	9.34	8.31	7.64	8.46
32	21.76	11.72	9.01	9.40	9.99	8.52	8.45	9.09
64	21.93	11.86	9.12	10.61	10.71	10.10	9.90	9.75
128	23.25	13.15	10.42	11.95	10.72	9.72	9.48	9.12
256	25.95	15.36	10.14	9.16	10.32	10.24	9.76	8.64

Table 4: Third PThread implementation: Gaussian kernel execution time (ms), for 1080p resolution of the input image.

Subsequently, in figure 8, we also compare the execution times obtained with the different image size, using both the sequential and the parallel approach. In particular, regarding the third parallel implementation with PThread, are represented the minimum execution times obtained from the previous test, varying the number of threads. We can see that we achieved with the parallel approach good improvements mostly with the biggest image and for a higher number of threads, as we expected.

2K	*Thread*							
Block Size	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
4	69.53	39.05	28.15	30.77	29.81	27.21	26.82	26.78
8	65.91	34.85	32.22	32.34	27.83	26.02	24.48	24.67
16	68.91	36.48	26.45	29.28	27.41	24.43	23.29	23.80
32	69.02	36.54	31.89	31.21	28.07	25.99	24.80	25.71
64	68.82	36.56	29.91	33.42	32.82	31.56	31.17	33.53
128	74.12	44.44	33.72	33.80	31.18	29.31	28.69	28.91
256	75.90	41.54	32.39	32.54	30.00	27.05	26.33	26.78

Table 5: Third PThread implementation: Gaussian kernel execution time (ms), for 2K resolution of the input image.

4K	*Thread*							
Block Size	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
4	152.99	89.03	73.95	72.34	66.99	62.63	64.13	69.03
8	146.50	85.47	66.07	69.12	64.41	57.26	55.24	55.23
16	151.94	82.58	66.80	64.87	58.32	54.88	49.39	52.52
32	154.31	85.94	68.64	65.38	59.98	57.00	57.12	59.62
64	153.78	88.89	70.58	71.54	70.46	75.70	78.22	77.83
128	177.69	99.95	82.73	76.06	70.52	70.75	63.82	65.31
256	166.21	90.58	78.74	69.05	64.36	57.94	56.25	57.95

Table 6: Third PThread implementation: Gaussian kernel execution time (ms), for 4K resolution of the input image.

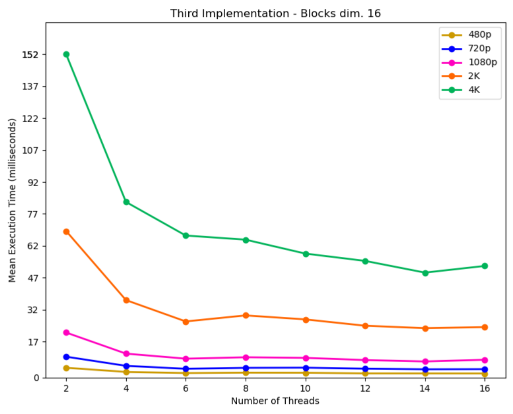

Figure 6: The mean execution times for the third PThread implementation, based on the number of threads and the size of the images are shown.

Finally, figure 9 compares the different speedup obtained for each image size, varying the number of threads and considering only the third parallel implementation with block size at 16, being the one with the best results.

The speedup is defined as $S_P = t_S/t_P$, where P is the number of processors, t_S is the completion time of the sequential algorithm and t_P is the completion time of the parallel algorithm. Basically, the speedup is perfect to ideal if $S_P = P$. We achieved the best results with the 4K image, using 14 threads: in this case we obtained a very high value (6.13) greatly speeding up execution compared to the sequential version.

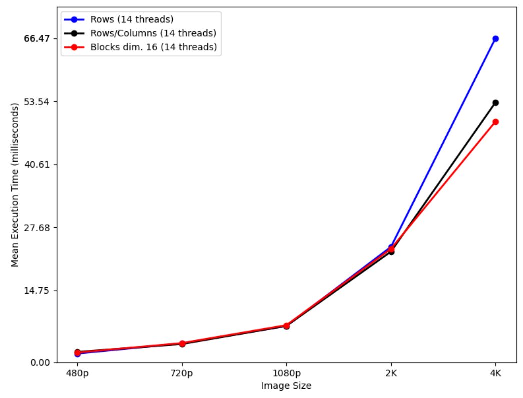

Figure 7: The mean execution times for the three PThread implementations based on image size.

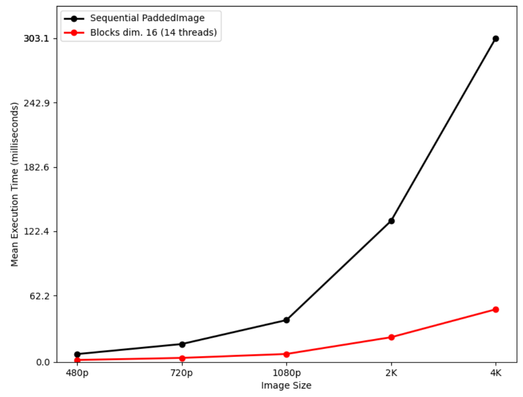

Figure 8: Comparison of execution times for each image size between the sequential and third parallel approach.

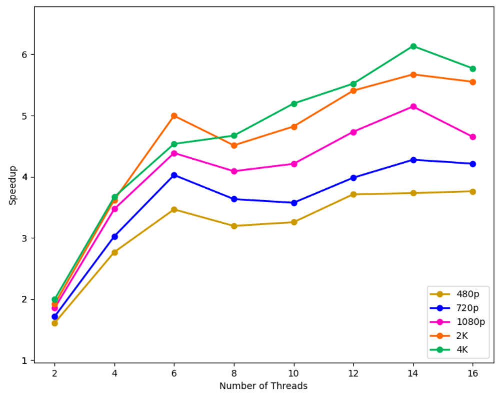

Figure 9: Comparison of speedup with PThread third implementation for different image size and varying threads' number.

5.2 CUDA - Implementation Results

The tables 7, 8 and 9 show the mean execution times as the number of threads and resolution of the input image vary for the three types of CUDA implementation. In particular, in the cited tables only the effective execution time of the filtering is shown without considering the memory allocation time and the data transfer between the host and the device memory.

As we can notice, obviously, the bigger the image the more the execution time increases with a decrease of this time as the number of threads increases and obtaining the best result with 16 threads for the three implementations.

Global Memory	*Resolution*				
Block Dim.	**480p**	**720p**	**1080p**	**2K**	**4K**
2	0.31	0.70	1.49	4.70	10.46
4	0.09	0.22	0.44	1.41	3.02
8	0.07	0.10	0.21	0.59	1.26
16	0.06	0.10	0.19	0.51	1.04
32	0.07	0.13	0.24	0.66	1.41

Table 7: CUDA Gaussian kernel execution time (ms), with global memory and without data transfer.

Constant Memory	*Resolution*				
Block Dim.	**480p**	**720p**	**1080p**	**2K**	**4K**
2	0.34	0.65	1.42	4.40	9.92
4	0.11	0.20	0.41	1.33	2.87
8	0.07	0.11	0.20	0.56	1.18
16	0.06	0.09	0.17	0.47	0.98
32	0.07	0.12	0.22	0.64	1.33

Table 8: CUDA Gaussian kernel execution time (ms), with constant memory and without data transfer.

Shared Memory	*Resolution*				
Block Dim.	**480p**	**720p**	**1080p**	**2K**	**4K**
2	-	-	-	-	-
4	-	-	-	-	-
8	0.12	0.21	0.46	1.33	3.06
16	0.06	0.10	0.19	0.54	1.11
32	0.07	0.12	0.23	0.67	1.40

Table 9: CUDA Gaussian kernel execution time (ms), with shared memory and without data transfer. For a few threads, it is not possible to divide the image adequately due to the implementation with shared memory.

As also shown in figure 10, the best performance is obtained by using the constant memory where the filter coefficients are initially stored and from which the threads can read their values. The improvement becomes evident as the size of the input images increases, obtaining similar execution times for small images. Basically, we expected to see this approach perform better than using global memory, but an interesting observation to make concerns the implementation with shared memory. For our type of task, the use of shared memory turns out to be worse in terms of execution probably due to the transfer time of the data in the tiles. Most likely, for our

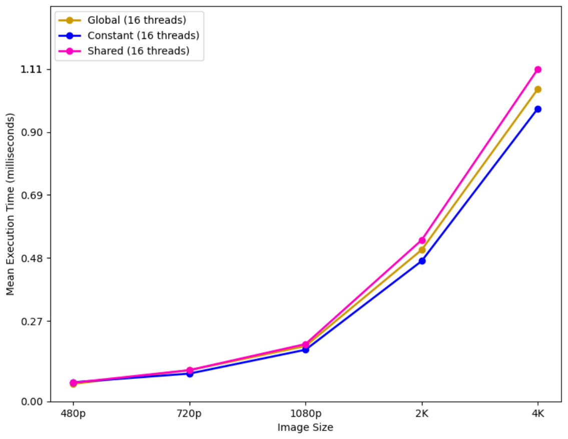

Figure 10: The mean execution times for the three CUDA implementations based on the size of the images are shown.

project, it is not possible to obtain a substantial improvement using this type of memory and the transfer cost is too expensive for computational performance purposes.

In the tables 10, 11 and 12, however, the mean execution times are shown as the number of threads and the resolution of the input image vary, considering also the data transfer and allocation time between host and device memories. The behavior is similar to the previous case where the execution time increases for larger images and decreases as the number of threads increases, obtaining the best result with 16 threads. The difference with respect to the first is given by a similarity of the results obtained in terms of mean execution times, as none of the three approaches is evidently better than the other due to copy time limitation in the device memory. Figure 11 shows exactly this similarity of results obtained for the three CUDA implementations due to copy time.

Global Memory	*Resolution*				
Block Dim.	**480p**	**720p**	**1080p**	**2K**	**4K**
2	1.27	2.62	5.49	17.65	39.39
4	1.05	2.12	4.44	14.36	31.95
8	1.03	2.02	4.21	13.54	30.19
16	1.02	2.02	4.19	13.46	29.97
32	1.03	2.05	4.24	13.61	30.34

Table 10: CUDA Gaussian kernel execution time (ms), with global memory and data transfer.

We consider the speedup obtained with the use of the constant memory, being the best approach for our task, both for the case with only the effective execution time and also the one with the copy time.

For the first case, as shown in the figure 12, we can see that we have obtained extraordinary results, in fact, all the speedups are very high with a significant improvement in performance, especially for the 4K image and 16 threads which has a value of 393.30.

Instead, if we also consider the copy and allocation time between the host and the device memory, this enormous speedup obtained is considerably limited obtaining a peak at 13.19 for 16 threads with the input image at 1080p resolution as shown in figure 13.

Constant Memory	*Resolution*				
Block Dim.	**480p**	**720p**	**1080p**	**2K**	**4K**
2	1.25	2.47	5.36	17.33	38.85
4	1.02	2.02	4.35	14.26	31.80
8	0.98	1.93	4.14	13.49	30.11
16	0.97	1.91	4.11	13.40	29.91
32	0.98	1.94	4.16	13.57	30.26

Table 11: CUDA Gaussian kernel execution time (ms), with constant memory and data transfer.

Shared Memory	*Resolution*				
Block Dim.	**480p**	**720p**	**1080p**	**2K**	**4K**
2	-	-	-	-	-
4	-	-	-	-	-
8	1.01	2.03	4.40	14.27	31.98
16	0.95	1.92	4.13	13.48	30.03
32	0.96	1.94	4.17	13.61	30.32

Table 12: CUDA Gaussian kernel execution time (ms), with shared memory and data transfer. For a few threads, it is not possible to divide the image adequately due to the implementation with shared memory.

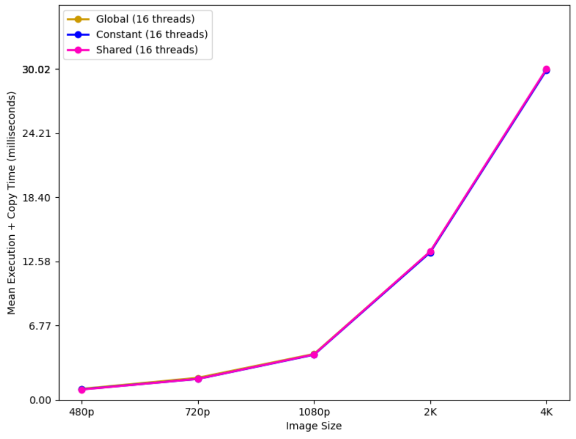

Figure 11: The mean execution + copy times for the three CUDA implementations based on the size of the images are shown.

5.3 Comparison between PThread and CUDA implementations

In the end, we also compare the sequential and parallel approaches proposed in this project.

As shown in figure 14, one of the two sequential approaches turns out to be more efficient than the other with an evident improvement in terms of execution as the res-

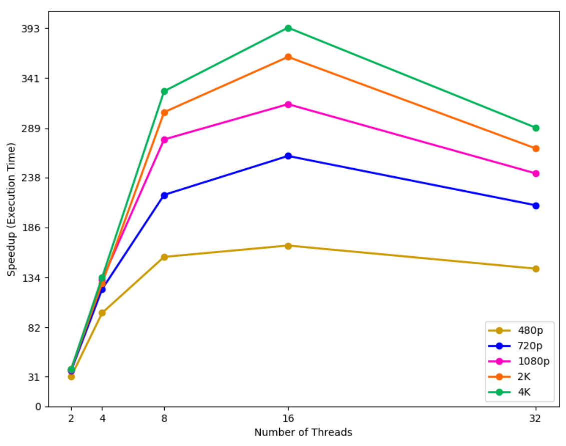

Figure 12: Comparison of speedup with CUDA constant memory approach for different image size and varying threads' number, without copy time.

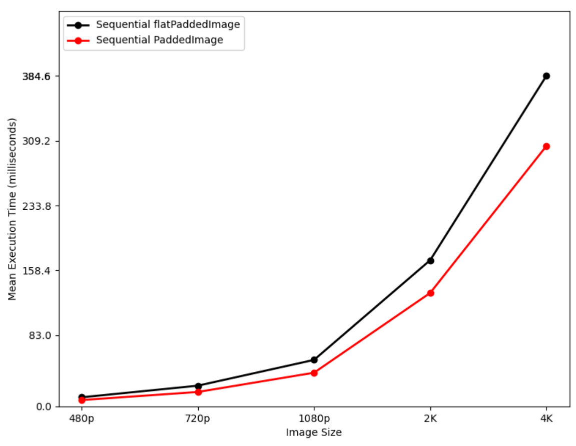

Figure 14: Comparison of execution times for each image size between the two sequential approaches.

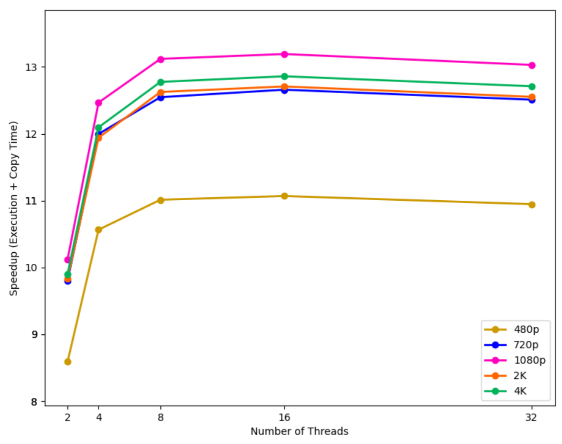

Figure 13: Comparison of speedup with CUDA constant memory approach for different image size and varying threads' number, with copy time.

olution of the input image increases. The approach with the **paddedImage** turns out to be better than the one with the **flatPaddedImage**, probably the data structure used to represent the input and output images is more efficient for our task.

On the other hand, considering the two parallel approaches, it can be seen that the implementation in CUDA is more efficient than the one with PThreads in terms of execution time, obviously considering the best implementation for the respective approaches.

As shown in figure 15, the difference in terms of efficiency is much better in the case of CUDA if only the execution times are considered otherwise the difference between the two approaches is less evident when the time

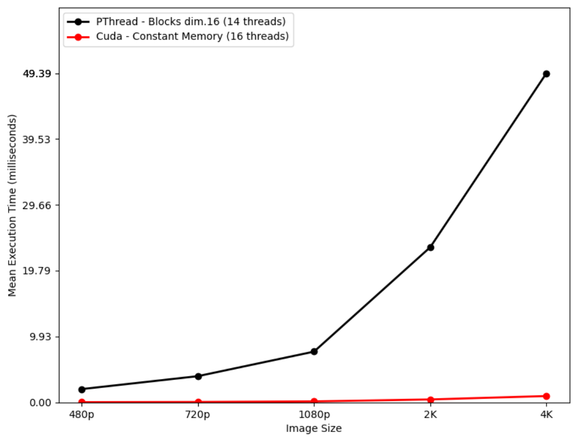

Figure 15: Comparison of execution times for each image size between the two parallel approaches, by considering only the execution time for CUDA.

is also considered of copy, figure 16. In both cases, the efficiency of the CUDA approach is more noticeable as the input size of the image to be processed increases.

Also in terms of speedup, the CUDA approach turns out to be better than the one with PThreads since the improvement obtained is extremely greater than the respective sequential version.

6 Conclusions

From the results obtained, the best approach for the sequential versions turned out to be the one with the **paddedImage**, probably due to the structures used to represent the images and the kernel which are more efficient for our task.

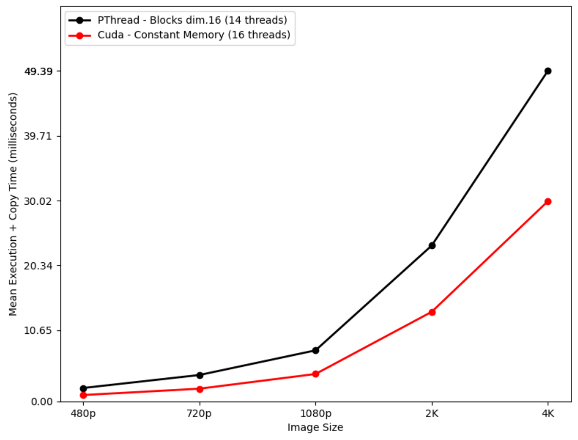

Figure 16: Comparison of execution times for each image size between the two parallel approaches, by considering the execution + copy time for CUDA.

As for the parallel versions, it was interesting to find out that the best approach for PThread is the one in which we have divided the image into blocks of fixed size, while for CUDA the use of constant memory allows to obtain the best results in terms of execution time.

We expected that the allocation and transfer of data between host and device memory would greatly limit the efficiency of the CUDA code, obtaining substantially similar results for the three approaches. However, the CUDA version was found to be vastly better than the PThread version regardless of copy time considerations.

Obviously, this evidence of efficiency obtained in the results is more noticeable among the methods implemented as the number of threads and the size of the input image increase.