Kernel image processing: sequential, multithread and CUDA implementation

Riccardo Fantechi
E-mail address
riccardo.fantechi1@edu.unifi.it

Abstract

In this work we will show the implementation and optimization of image convolution through sequential, parallel and CUDA programming techniques. Initially, a sequential version of the algorithm is developed and analyzed followed by a multithreading approach to exploit CPU parallelism. Subsequently, three CUDA-based implementations are introduced: a global memory based kernel, a kernel utilizing constant memory for the filter values, and an advanced version that combines constant memory for the kernel with shared memory for image blocks. Performance comparison are conducted across all version, focusing on execution times and computational efficiency. The study highlights the significant performance improvements achieved through multithreading and CUDA, demonstrating the potential of GPU based optimization in image processing application.

1. Introduction

Image processing is a crucial case of study within computer vision and digital image analysis that focuses on manipulating images to improve their quality, transform or extract information. A critical concept in image processing is the use of kernels, which are small, predefined matrices applied to images through a mathematical operation known as convolution. Kernels enable various image transformations by emphasizing, reducing, or detecting specific features in the image. A kernel, also referred to as filter, is a small matrix, often of size 3x3, 5x5 or 7x7, containing numerical values that determine how an image is transformed. The convolution operation let kernels slide over the image, pixel by pixel, performing element-wise multiplication with the corresponding pixel values in the image and summing results. This operation creates a new pixel value that forms part of the output image. Different types of kernels serve unique purpose in image processing, this are the one that we used:

- **Edge detection**

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

  This matrix highlights the boundaries or edges within an image. It works by detecting areas with a high intensity gradient, which correspond to sudden changes in brightness. The output is a gray-scale image where edges are represented as bright lines on a dark background.

- **Gaussian blur**

$$\begin{bmatrix} 0.0050 & 0.0090 & 0.0139 & 0.0164 & 0.0139 & 0.0090 & 0.0050 \\ 0.0090 & 0.0164 & 0.0253 & 0.0298 & 0.0253 & 0.0164 & 0.0090 \\ 0.0139 & 0.0253 & 0.0390 & 0.0460 & 0.0390 & 0.0253 & 0.0139 \\ 0.0164 & 0.0298 & 0.0460 & 0.0543 & 0.0460 & 0.0298 & 0.0164 \\ 0.0139 & 0.0253 & 0.0390 & 0.0460 & 0.0390 & 0.0253 & 0.0139 \\ 0.0090 & 0.0164 & 0.0253 & 0.0298 & 0.0253 & 0.0164 & 0.0090 \\ 0.0050 & 0.0090 & 0.0139 & 0.0164 & 0.0139 & 0.0090 & 0.0050 \end{bmatrix}$$

  This filter is used for image smoothing by reducing noise ad detail. It applies a Gaussian function eq.(1), which blurs the image in a way that preserves the overall structure while removing sharp features. The function that creates this kernel can take the dimension and the standard deviation as parameters.

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (1)$$

- **Laplacian**

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The Laplacian filter enhances regions with rapid intensity variations, such as edges or fine details. It use the second derivative of the image to detect changes in it, producing a edge-enhanced version of the image.

- **Gaussian-Laplacian (LoG)**

$$\begin{bmatrix} 0 & 0 & -2 & 0 & 0 \\ 0 & -2 & -1 & -2 & 0 \\ -2 & -1 & 16 & -1 & -2 \\ 0 & -2 & -1 & -2 & 0 \\ 0 & 0 & -2 & 0 & 0 \end{bmatrix}$$

This filter combines Gaussian smoothing with the Laplacian operator. First it reduces noise using the Gaussian blur and then highlights areas with rapid intensity changes by applying the Laplacian. The result is an image with a clearer edge map compered to the traditional edge detection method.

- **Sharpen**

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The sharpen filter increases the contrast of adjacent pixels to enhance edges and details in an image. It works by emphasizing the high frequency components of the image, making the output appear crisper and more detailed than the original

1.1. Setup

The development was conducted on a MacBook Air M1 (2020), with an Apple M1 chip, 8 GB of RAM and running on macOS Sonoma 14.2, for the sequential and multithreading implementation. Unfortunately there is no CUDA support for MacOS so we had to use a remote server that the University of Florence makes available to the students. This machine has an Intel(R) Xeon(R) Silver 4314 CPU, with 64 GB of RAM and a NVIDIA RTX A2000 with 12 GB as graphic device, running on Ubuntu 22.04.4 LTS.

1.2. Image pre-processing and control

Before performing the convolution operation between image and kernel it is necessary to add a padding to the image based on the size of the filter. Padding is the amount of space between the image and the outer edge of it, we use this to make sure that we can slide the kernel perfectly over the image and prevent the kernel from being multiplied by a non-existent values. Two type of padding were implemented:

- **Zero Padding**: Adds zeros around the edges of the image, useful for applications where edge details are not critical.
- **Replicate Padding**: Duplicates the border pixels around the image to reduce distortions caused by the lack of information at the edges.

To choose the right padding dimension we have to consider the kernel size used, for a kernel size $n \times n$, a padding of $(n-1)/2$ pixels was added to each side. This ensures that every pixel in the original image is processed without any data loss. Regarding image management and control we use a library from GitHub, named Writer, to handle image-to-numeric data transformation. This library allowed us to convert an image pixel values into arrays of numerical data suitable for processing. The library was selected for its efficiency and simplicity in handling various image formats, making it ideal for our purpose. Let's see the load and save functions in detail:

```
1 void image::loadImage(const std::string
  ↪ pathImage) {
2     int width ;
3     int height;
4     int channels;
5     unsigned char* data =
      ↪ stbi_load(pathImage.c_str(),
      ↪ &width, &height, &channels, 1);
6
```

```
    //Make sure the dimension are right
    imageMatrix.resize(width*height);
    for (size_t i = 0; i < width*height;
    ↪ ++i) {
        imageMatrix[i] =
        ↪ static_cast<float>(data[i]) /
        ↪ 255.0f; //Normalize values
        ↪ [0,1]
    }

    //Save state
    imageHeight = height;
    imageWidth = width;
    //Free memory
    stbi_image_free(data);
}
```

as we can see the library provides the *stbi_load* function which transforms all the pixel values into char, after which we normalized the values and saved the image state. The same approach was used for the image saving function:

```
void image::saveImage(const std::string
↪ pathImage) {
    std::vector<unsigned char>
    ↪ imageData(imageWidth*imageHeight);

    //Convert normalized float from [0.0,
    ↪ 1.0] to unsigned char [0, 255]
    std::transform(imageMatrix.begin(),
    ↪ imageMatrix.end(),
    ↪ imageData.begin(), [](float
    ↪ value) {
        return static_cast<unsigned
        ↪ char>(std::clamp(value *
        ↪ 255.0f, 0.0f, 255.0f));
    });

    //Create the image.png in the path
    stbi_write_png(pathImage.c_str(),
    ↪ imageWidth, imageHeight, 1,
    ↪ imageData.data(), imageWidth);
}
```

note how the *stbi_write_png* function takes as input the saving path, the dimension and the values to be saved in order to recreate the png image.

1.3. CUDA overview

CUDA (Compute Unified Device Architectures) is a parallel computing platform and programming model developed by NVIDIA, providing high computational power for application like deep learning, image processing, and for scientific simulations. CUDA extends the C programming languages and includes a rich library to manage and optimize GPU-based computations. The key features of CUDA are:

- Parallelism: CUDA allows task to be executed in parallel by distributing computations across all the GPU cores.
- Threads and Blocks: CUDA uses a hierarchy of threads organized into blocks, which are further grouped into grids, than each thread can operate on a specific data element.
- Scalability: The same code can run on GPUs of different size and capabilities

The memory system in CUDA is hierarchical and includes sereval types of memory, each with unique characteristics. Efficiente memory usage is crucial for optimizing CUDA programs, as we will see in the implementation. This are the main types of memory used in CUDA:

- Global Memory: This is the largest and most versatile type, resides in the GPU's DRAM and is accessible to all threads across all blocks. It support vast amount of data storage, however, its high latency makes it inefficient for frequent memory operations.
- Shared Memory: On the other hand, this type of memory offers a much faster alternative for intra-block communication. It is a small memory shared among threads within a block and has significantly lower latency compared to the global memory. Shared memory is especially effective when threads in a block need to collaborate, such as in convolution techniques. However, its limited size requires careful allocation and management.
- Constant Memory: This is another specialized type that is particularly well suited for read-only data shared by all threads, as a filter that do not change during the execution. Stored in a cache optimized for uniform

access, constant memory is highly efficient when the same data is read by many threads simultaneously, as it avoids the high latency associated with global memory.

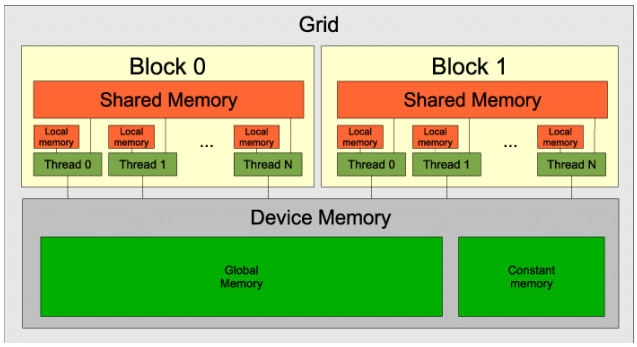

Figure 1. CUDA memory model

Each of these memory types was used in the project, especially the combination of some of these led to excellent results in terms of execution time.

2. Code walk-through

In this section we will see all the C++ code implementations. First we will see how the convolution operation between the image and the kernel is implemented sequentially, after which we will use a multi-threading approach via OpenMP to improve the execution time with CPU threads, and then move on to the CUDA implementation.

2.1. Sequential implementation

In the sequential approach, the convolution operation processes the image pixel by pixel, iterating through the entire input image to compute the output. This method, while straightforward and easy to understand, is computationally intensive due to its nested loops over both the image dimensions and the kernel. The convolution operation is expressed in eq.(2), in short we are sliding the kernel over the image, while accumulating a multiplication sum.

$$I_{\text{output}}(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} K(i,j) \cdot I_{\text{input}}(x+i, y+j) \tag{2}$$

Where:

- $K(i,j)$ is the filter value at position (i,j)
- $I_{\text{input}}(x+i, y+j)$ is the image pixel at position $(x+i, y+j)$
- k is half the size of the filter

We start by applying padding with the function we discussed previously and then we initialize the output image, creating a blank output image of the same dimensions as the input image to store the results of the convolution.

```
void image::applyConvolution(const kernel& kernel, bool padding) {

    addPadding(kernel.getKernelHeight(),padding);
    //Allocate output vector
    std::vector<float> output (imageHeight*imageWidth , 0.0f);
```

Than we iterate over the image looping through each pixel in the input image and applying the convolution operation at every pixel. The filter is overlaid on the image, centered on the current pixel, the filter values are multiplied by the corresponding pixel values in the overlapping region of the image, and the products are summed to compute the output value.

```
for (int i = 0; i < imageHeight; i++) {
    for (int j = 0; j < imageWidth; j++) {
        float sum = 0.0f; //To store the local sum
        //Iterate through all the kernel values
        for (int ki = 0; ki < kernel.getKernelHeight(); ki++) {
            for (int kj = 0; kj < kernel.getKernelWidth(); kj++) {
                //Compute the correspondent index
                int row = i + ki;
                int col = j + kj;
                if (row >= 0 && row < imageHeight && col >= 0 && col < imageWidth) {
                    int imageIndex = row * imageWidth + col;
```

```
            int kernelIndex = ki *
            ↪ kernel.getKernelWidth() + kj;
            sum += imageMatrix[imageIndex] *
            ↪ kernel.getKernel()[kernelIndex];
                }
            }
        }
        output[i*imageWidth + j] = sum;
    }
```

The computed value is stored in the corresponding position in the output image, if the convolution result exceeds the valid pixel range [0, 255] it is clipped to stay within bounds.

```
if(sum < 0){
    sum = 0;
    }
else if (sum > 255){
    sum = 255;
    }
```

2.2. Multi-thread implementation

To improve the performance of the convolution operation, a multi-thread approach was implemented using OpenMP, which provides an easy way to exploit multiple threads on shared-memory systems. The outermost loop, which iterates over the rows of the image, was parallelized, each thread processes a different row of the image independently, reducing the overall execution time. The OpenMP directive used was *#pragma omp parallel for* which assigns iterations of the loop to multiple threads. This approach ensures that each thread computes the convolution for one or more rows of the image without interference from other threads.

```
#pragma omp parallel for
for (int i = 0; i < imageHeight; i++) {
    for (int j = 0; j < imageWidth; j++)
    ↪ {
        float sum = 0.0f; //To store the
        ↪ local sum

        //Iterate through the kernel
        for (int ki = 0; ki <
        ↪ kernel.getKernelHeight();
        ↪ ki++) {
            for (int kj = 0; kj <
            ↪ kernel.getKernelWidth();
            ↪ kj++) {
                int row = i + ki;
                int col = j + kj;
                if (row >= 0 && row <
                ↪ imageHeight && col >=
                ↪ 0 && col <
                ↪ imageWidth) {
                    sum +=
                    ↪ imageMatrix[row *
                    ↪ imageWidth + col]
                    ↪ *
                    ↪ kernel.getKernel()[ki
                    ↪ *
                    ↪ kernel.getKernelWidth()
                    ↪ + kj];
                }
            }
        }
        //Save the sum in the output
        ↪ matrix
        output[i*imageWidth + j] = sum;
    }
}
```

Since the computation of each pixel in the output image is independent, it can be performed in parallel without synchronization issues. By distributing the workload among multiple threads, the overall computation time is significantly reduced compared to the sequential implementation, as we will see in the analysis. Unfortunately, while the outermost loop was parallelized, the inner loops were left sequential to minimize thread synchronization overhead, this limits the effectiveness of the method and does not make it entirely optimal.

2.3. CUDA implementation

The CUDA implementation focuses on accelerating the convolution operation by leveraging the parallel computing capabilities of GPUs. The convolution algorithm was implemented using three different approaches, each utilizing a specific type of GPU memory. The first approach utilizes global memory, this version demonstrates the simplest implementation of CUDA, where all data resides in global memory. The second approach takes advantage of constant memory, this method is particularly efficient for applying fixed convolution kernels, as the kernel values are stored in constant memory, reducing global

memory access which has high-latency. The final approach utilizes shared memory, a fast, low-latency memory shared among threads within a block. By dividing the image into blocks and loading relevant portions into shared memory, this version minimizes redundant global memory access, furthermore, by combining it with the constant method we managed to obtain excellent results. This approach is particularly suited for larger kernel sizes and high resolution images. Each implementation explores the trade offs between memory latency, bandwidth and computation to achieve optimal performance for the convolution operation.

- **Global Memory Version** The global memory implementation uses a CUDA kernel to perform convolution on an image matrix by assigning each thread to compute the convolution result for a specific pixel. The global thread index is calculated using $blockIdx$ and $threadIdx$, enabling parallel processing across the image. Each thread retrieves image and kernel values from global memory, performs the convolution calculation, and writes the result back to global memory. Boundary checks ensure threads operate within valid image regions, while thresholding normalizes pixel values to the [0, 255] range. Data is transferred between host and device memory using $cudaMemcpy$, and grid-block dimension are configured to match the image size. This is how the memory has been managed:

```
//Allocate device memory
float *d_image, *d_kernel, *d_output;
cudaMalloc(reinterpret_cast<void**>
  ↪  (&d_image),
  ↪  imageMatrix.size()*sizeof(float));
cudaMalloc(reinterpret_cast<void**>
  ↪  (&d_kernel),
kernel.getKernel().size()*sizeof(float));
cudaMalloc(reinterpret_cast<void**>
  ↪  (&d_output),
  ↪  output.size()*sizeof(float));

//Transfer data from host to device
  ↪  memory
cudaMemcpy(d_image,
  ↪  imageMatrix.data(),imageMatrix.size()
  ↪  *
  ↪  sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_kernel,
  ↪  kernel.getKernel().data(),
kernel.getKernel().size() *
  ↪  sizeof(float),
cudaMemcpyHostToDevice);
```

First we allocate the memory on the device with the right size, then we transfer the data from the host to the device, as we will see this is the part that takes the most execution time due to the high-latency of the global memory. Than we call the CUDA kernel function with the block and grid size and we wait until each thread has finished executing.

```
applyGlobalCUDAConvolutionKernel<<<gridDim,
  ↪  blockDim>>>(d_kernel, d_image,
  ↪  d_output, imageWidth,
  ↪  imageHeight,kernel.getKernelWidth());

//Waits for threads to finish
cudaDeviceSynchronize();
```

After the convolution we copy back the result from the device to the host, and than we free the allocated memory.

```
//Copy back the result from the GPU
cudaMemcpy(output.data(),d_output,
output.size()* sizeof(float),
cudaMemcpyDeviceToHost);

//Free the memory
cudaFree(d_image);
cudaFree(d_kernel);
cudaFree(d_output);
```

- **Constant memory Version** For the constant version we use the same logic seen previously, but passing the filter through constant memory.

```
//Allocate memory for storing constant
  ↪  kernel vector
__device__ __constant__ float
  ↪  d_constantKernel[25*25];
```

We simply allocate space for the input and

output image and then copy the kernel with the *cudaMemcpyToSymbol* directive.

```
cudaMemcpyToSymbol(d_constantKernel,
kernel.getKernel().data(),
kernel.getKernel().size()*sizeof(float),
0, cudaMemcpyHostToDevice);
```

The CUDA kernel function remains the same with the only difference that to access the kernel values we will go to the constant memory space that we have allocated.

- **Shared Memory Version** The shared memory version is the most complicated at a code level, in addition to managing the kernel with constant memory, we must use the shared memory to pass blocks of image on which to perform the convolution operation. In order to do this each block will load the corresponding input 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 file. The improvements will be noticeable as the filter size grows, because more pixels will be reused by the threads of the same block. We allocate the shared memory with the directive *extern __shared__ float sharedMemory[]*. Than we load the block of pixels on which we want to work via the thread index.

```
//Load of needed near pixel, with
    ↪ bounds
if( x < width && y < height){
    sharedMemory[sharedIndex] = mat[y
        ↪ * width + x];
}else{
    sharedMemory[sharedIndex] = 0.0f;
}

//Load of needed extra bounds
if(tx < midKernel){
    //Left bound
    sharedMemory[sharedIndex -
        ↪ midKernel] = (x >= midKernel)
        ↪ ? mat[y * width + x -
        ↪ midKernel] : 0.0f;
    //Right bound
    sharedMemory[sharedIndex +
        ↪ blockDim.x] = (x + blockDim.x
        ↪ < width) ? mat[y * width + x +
        ↪ blockDim.x] : 0.0f;
}
if (ty < midKernel) {
    //Upper bound
    sharedMemory[sharedIndex -
        ↪ midKernel * sharedWidth] = (y
        ↪ >= midKernel) ? mat[(y -
        ↪ midKernel) * width + x] :
        ↪ 0.0f;
    //Lower bound
    sharedMemory[sharedIndex +
        ↪ blockDim.y * sharedWidth] = (y
        ↪ + blockDim.y < height) ?
        ↪ mat[(y + blockDim.y) * width +
        ↪ x] : 0.0f;
}
//Synchronize all thread to make sure
    ↪ that the shared memory is loaded
__syncthreads();
```

When we call the CUDA kernel function we have also to calculate the shared memory size using

```
unsigned int sharedMemSize =
    ↪ (blockDim.x +
    ↪ kernel.getKernelWidth() - 1)
    ↪ *(blockDim.y +
    ↪ kernel.getKernelHeight() - 1) *
    ↪ sizeof(float);
```

Than we can call the CUDA kernel specifying the size of the shared memory

```
applySharedCUDAConvolutionKernel<<<gridDim,
    ↪ blockDim,
    ↪ sharedMemSize>>>(d_image,
    ↪ d_output, imageWidth,
    ↪ imageHeight,kernel.getKernelHeight());
```

3. Test

To evaluate the CUDA implementation over the multi-threading one, three distinct test were conducted to analyze the performance.

- **Thread Scalability Evaluation**, aimed to asses the impact of increasing the number of threads and resolution of the image.
- **Memory Type Evaluation**, the focus shifted to evaluating performance variations using different type of CUDA memory.

- **Grid and Block Organization Evaluation**, we concentrated on evaluating the different performances by varying the size of threads per block and blocks per grid.

3.1. Speedup

To measure the performance of the parallel implementation, we calculated the speedup by comparing the execution time of the sequential, parallel and CUDA version of the code. In our case we focused both on the convolution execution time, and for the CUDA code we also take in consideration the time to transfer the needed data. These measurements were collected in separate runs for all the version of the code. To calculate the speedup we used the eq.(3).

$$\text{Speedup} = \frac{\text{Time(Sequential)}}{\text{Time(Parallel)}} \tag{3}$$

By recording times for both implementations over multiple iterations, we obtained a dataset that allowed us to compute the average speedup.

3.2. Thread Scalability Evaluation

This test is conducted with the multi-threading version of the code. We evaluate the execution times for three different kernel sizes:

- Filter size 3×3
  As expected, as the number of threads increases, the time taken by the operation decreases, for example considering the resolution *1920x1080* we notice that the execution time drops from 3.54(s), of the sequential version, to 0.17(s) with 64 threads.

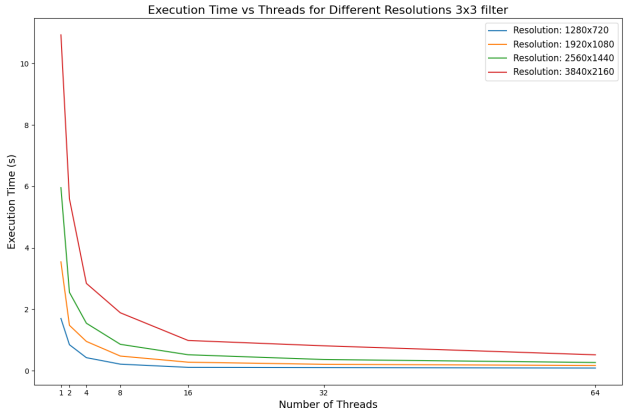

Figure 2. 3x3 filter

Table 1. 3×3 kernel processing time(s)

	Risoluzioni			
Thread	*1280x720*	*1920x1080*	*2560x1440*	*3840x2160*
1	1.70144	3.54398	5.9586	10.9299
2	0.847324	1.47878	2.55174	5.5976
4	0.425263	0.954184	1.54633	2.84254
8	0.214304	0.478256	0.860375	1.88654
16	0.10154	0.277737	0.52027	0.984308
32	0.102423	0.209604	0.364756	0.811517
64	0.0878304	0.170771	0.268073	0.51841

- Filter size 5×5
  By increasing the size of the kernel we notice a worsening of performance, in fact always considering the resolution *1920x1080*, we notice that in the sequential version we take 8.64(s) to execute the code improving the result to 0.46(s) with 64 threads. Although these two graphs have the same trend, they have different scales, in fact the latter, in the worst case, reaches three times the precious one.

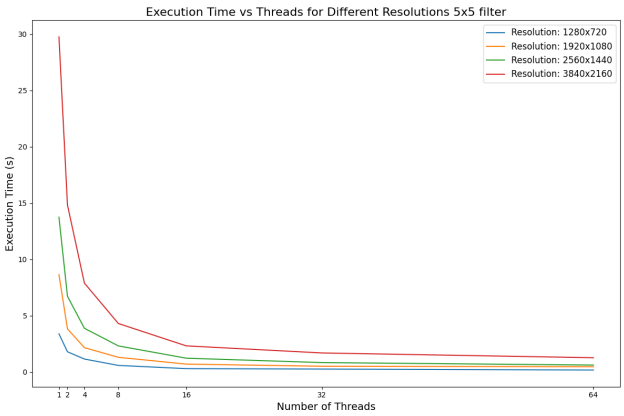

Figure 3. 5x5 filter

Table 2. 5×5 kernel processing time(s)

	Risoluzioni			
Thread	*1280x720*	*1920x1080*	*2560x1440*	*3840x2160*
1	3.38214	8.64237	13.7455	29.7608
2	1.79107	3.8293	6.72035	14.8343
4	1.14609	2.15591	3.87746	7.8904
8	0.579377	1.30084	2.31263	4.30352
16	0.295625	0.695285	1.2223	2.31712
32	0.256095	0.506785	0.829696	1.68542
64	0.168905	0.463903	0.611535	1.26343

- Filter size 7×7
  We can find the same effect with this filter, we notice how the curves have the same trend but considering the resolution *1920x1080* we notice that in the sequential version we have 15.01(s) execution times that drops down to 0.73(s) with 64 threads.

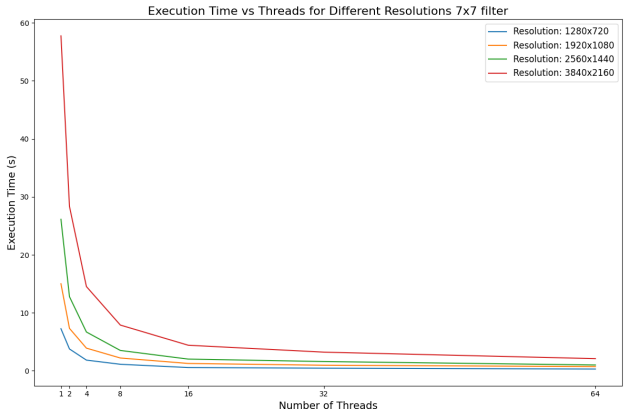

Figure 4. 7x7 filter

Table 3. 7×7 kernel processing time(s)

	Risoluzioni			
Thread	*1280x720*	*1920x1080*	*2560x1440*	*3840x2160*
1	7.2573	15.0141	26.1269	57.7501
2	3.76987	7.32145	12.7917	28.3655
4	1.84453	3.91083	6.69166	14.5266
8	1.12864	2.21041	3.5106	7.88962
16	0.567581	1.26958	2.02306	4.41374
32	0.456342	0.963561	1.59754	3.21812
64	0.307193	0.733279	1.00899	2.10232

As we expect as the number of threads increases we notice a great improvement in execution times. Obviously we notice a similar trend in the different graphs, paying attention to the values, we note however that the execution times with few threads have very high values and then converge around the 32 threads to values under 1.5(s). In the next graphs we can see the speedup with two different kernel size for all the image resolution Figure 5.

3.3. Memory Type Evaluation

This test is conducted with the CUDA implementation, we compared the execution times of the three versions we talked about before. The following table show the execution times of the filter processing, without the data transfer:

- Filter size 3×3
  We can already see, from the smaller kernel size, that there is a great improvement in the execution times of the convolution operation.

Table 4. CUDA kernel processing time (ms), without data transfer

Resolution	**Global (ms)**	**Constant (ms)**	**Shared (ms)**
1280x720	0.084	0.078	0.095
1920x1080	0.151	0.148	0.162
2560x1440	0.241	0.254	0.254
3840x2160	0.484	0.501	0.501

- Filter size 5×5
  Here we can look at the execution times for a 5×5 kernel, as expected we have great improvement with the use of shared memory due to its low-latency.

Table 5. CUDA kernel processing time (ms), without data transfer

Resolution	**Global (ms)**	**Constant (ms)**	**Shared (ms)**
1280x720	0.499	0.275	0.167
1920x1080	0.317	0.270	0.179
2560x1440	0.516	0.458	0.287
3840x2160	1.131	0.982	0.586

- Filter size 7×7
  One thing we want to point out is that for the resolution *1920x1080* we found shorter execution times compared to the minimum resolution *1280x720*, we can notice this fact even better in the largest filter size we analyzed.

Table 6. CUDA kernel processing time (ms), without data transfer

Resolution	**Global (ms)**	**Constant (ms)**	**Shared (ms)**
1280x720	0.5258	0.2520	0.1331
1920x1080	0.4264	0.3590	0.0820
2560x1440	0.7412	0.6326	0.1239
3840x2160	1.5778	1.3926	0.2536

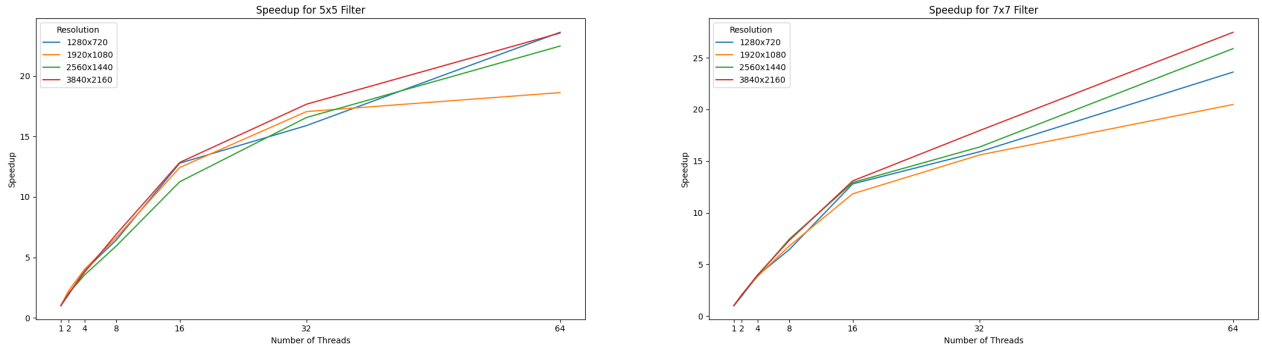

Figure 5. Multi-thread Speedup for 5x5 and 7x7 kernel for all the image resolution

The following table show the execution time of the filter processing, including the data transfer for the three version of the CUDA code.

- Filter size 3×3

Table 7. CUDA kernel processing time (ms), with data transfer 3x3

Resolution	Global (ms)	Constant (ms)	Shared (ms)
1280x720	13.44	268.992	1.678
1920x1080	294.44	183.122	141.780
2560x1440	155.03	291.292	4.888
3840x2160	7.44	10.667	12.339

- Filter size 5×5

Table 8. CUDA kernel processing time (ms), with data transfer 5x5

Resolution	Global (ms)	Constant (ms)	Shared (ms)
1280x720	660.6	4.12	241.5
1920x1080	14.35	50.6	227.43
2560x1440	38.63	5.27	4.22
3840x2160	8.638	9.52	7.956

- Filter size 7×7

Table 9. CUDA kernel processing time (ms), with data transfer 7x7

Resolution	Global (ms)	Constant (ms)	Shared (ms)
1280x720	747.9930	5.5235	1.7575
1920x1080	10.3162	58.9005	142.4415
2560x1440	55.9286	6.2800	5.1288
3840x2160	23.2367	10.2890	8.6959

Evaluating the total execution times we notice how there is a greater use of time in the operations carried out by the global memory, while on average for the operations in shared memory we find big improvements except for same cases where bottlenecks have occurred which make the code less efficient.

3.4. Grid and Block Organization Evaluaiton

The organization of the CUDA grid is crucial for the performance of the kernel being executed. Specifically, the size of the blocks (defined by the number of threads per block) influences how many blocks will make up the grid to process every pixel in the input image. If each block has smaller dimension, an higher number of blocks has to be used in order to fill up the source image. In the next table we can see how the performances of the three memory type implementations are affected by the block size.

Table 10. Global memory convolution (ms) for a 5x5 filter

Resolution	8x8	16x16	32x32
1280x720	551.64	535.06	882.96
1920x1080	5.60	4.16	58.98
2560x1440	13.61	7.79	29.70
3840x2160	11.86	13.77	17.82

Table 11. Constant memory convolution (ms) for a 5x5 filter

Resolution	8x8	16x16	32x32
1280x720	774.73	842.03	581.96
1920x1080	29.92	21.24	31.63
2560x1440	48.93	51.02	46.06
3840x2160	23.28	12.86	10.88

Table 12. Shared memory convolution (ms) for a 5x5 filter

Resolution	**16x16**	**32x32**	**32x8**	**64x8**
1280x720	559.96	530.49	534.68	560.97
1920x1080	21.55	3.41	6.14	22.52
2560x1440	28.45	27.41	10.65	26.89
3840x2160	10.94	11.35	59.00	8.24

The results indicate that the effect of block size on performance becomes more significant with higher-resolution images.

4. Conclusion

This project demonstrated how significant performance improvements can be achieved through parallel processing, starting with CPU multithreading and extending to CUDA-based GPU acceleration.

We observed that even with CPU multithreading, it is possible to achieve substantial speedup compared to sequential execution. However, leveraging CUDA allowed us to further optimize performance by efficiently managing memory types global, constant and shared on the specific requirements of the task.

This progression highlights the value of tailoring memory usage and thread organization to fully exploit the capabilities of the GPU, providing a flexible and powerful approach for high-performance computing.