Kernel Image Processing

Davide Del Bimbo
davide.delbimbo@edu.unifi.it

Abstract

Kernel Image Processing (KIP) is a technique wherein a convolution matrix is applied to an image. This report introduces a parallelized implementation of the Kernel Image Processing technique, developed both in C++ and CUDA. The application is designed to support multichannel image processing.
This report conducts a comprehensive performance analysis, with a specific focus on speed-up metrics conducted on an Intel i7-10750H CPU and an NVIDIA GeForce RTX 3060 GPU. The study compares the sequential and parallel versions of the Kernel Image Processing technique, showing the efficacy and inherent advantages that parallelization brings to this image processing paradigm.

1. Introduction

This report provides an examination of the Kernel Image Processing technique, offering insights into its theoretical foundation and implementation principles. Subsequently, it introduces a parallelization strategy with CUDA, a parallel programming framework specifically designed for optimizing GPU utilization.

2. Kernel Image Processing

The use of *kernels* - also known as *convolution matrices* or *masks* - is invaluable to image processing. Some techniques, like blurring, edge detection and sharpening, rely on the application of kernels across an image.
In image processing, a kernel is a small matrix used to perform various operations on the image [1]. These operations are accomplished by performing a convolution between the kernel and an image. In other words, each pixel in the processed output image represents a function of nearby pixels in the input image. The kernel is that function.

Each image is composed of pixels and each pixel consists of a combination of primary colors. Specifically, any pixel can be represented as a multidimensional array of numerical elements whose value indicates the intensity of a specific pixel color. The value we can assign to a pixel may vary depending on the image format (i.e. in an 8-bit image, each pixel is assigned a value ranging from 0 to 255). The size of a pixel array is defined by the number of **channels** in the image. A channel represents a gray-scale image, made of just one of primary colors [2] (i.e. an image from a standard digital camera will have a red, green and blue channel, while a gray-scale image has just one channel).

Mathematically, an image processing is described by the **convolution** operation between an original input image and a kernel function. Specifically, the general expression of a discrete convolution is:

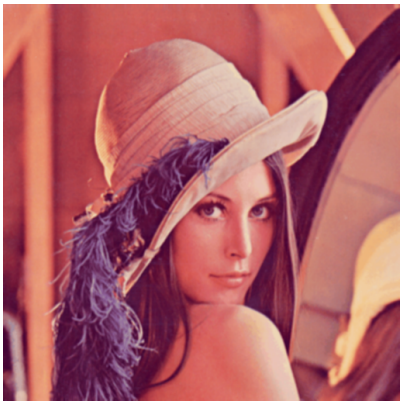
$$g(x,y) = \omega \otimes f(x,y) = \sum_{i=-1}^{a} \sum_{j=-b}^{b} \omega(i,j) f(x-i, y-j)$$

where $g(x,y)$ represents the processed output image, $f(x,y)$ represents the original input image and ω represents the filter kernel. Each element of the filter kernel is considered in the interval $-a \leq i \leq a$ and $-b \leq j \leq b$.

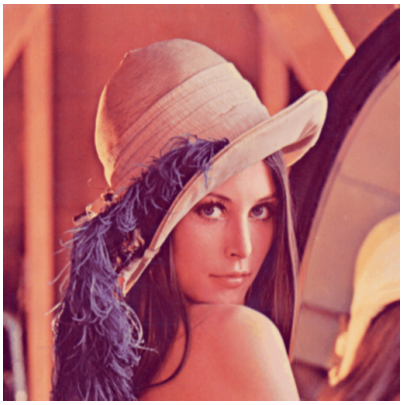
Depending on the values of the kernel elements, we can apply different effects to an image (fig. 1a):

- **Box Blur**: each pixel in the output image has a value equal to the average value of its neighboring pixels in the input image (fig. 1b).
- **Gaussian Blur**: is similar to the Box Blur kernel, but it is dependent upon the Gaussian function (i.e. a function which creates a distribution of values around the center point). This results in a kernel in which pixels near the center contribute more towards the new pixel value than those further away (fig. 1c).
- **Edge Detection**: detects edges within an image (i.e. captures abrupt transitions in image values). Notice that if all pixel values are comparable, the resultant pixel value will be close to 0. However, edges (i.e. locations with extreme differences in pixel values) will result in values far from zero (fig. 1d).
  Observe that if the sum of the kernel elements is 1, then the brightness of each pixel in the output image remains unchanged from those in the original input image. In contrast, for example, if the sum is 0, then the brightness decreases (at similar values, the convolution of pixels takes the value 0, thus resulting in black).

(a) Original input image.



(b) Box Blur effect.



(c) Gaussian Blur effect.

(d) Edge Detection effect.

(e) Sharpen effect.

(f) Unsharp Mask effect.

Figure 1: Kernel image processing with different filters.

- **Sharpen**: accentuates the edges of the image. This operation is similar to Edge Detection, but the sum of the kernel elements is kept at 1, ensuring that the brightness of each pixels remains unchanged from the original image (fig. 1e).

- **Unsharp Mask**: makes the image sharper. This technique is based on creating a Gaussian blurred copy of the original input image and subtracting that copy from it. In this way, pixels above a certain threshold are sharpened, enhancing light and dark pixels (fig. 1f).

2.1. Convolution algorithm

Convolution is the process of adding each element of an input image to its local neighbors, weighted by the kernel. Convolution is a mathematical operation, denoted by the symbol $\otimes$, which describes a rule of how to combine two functions to form a third function. Specifically, an input image, or *feature map*, and a kernel are combined to form a processed output image (fig. 2). The convolution algorithm is often interpreted as a filter, since the kernel filters the feature map for certain information.

Typically, the kernel is a square matrix of odd dimensions and its central element is called *origin*. That element is overlaid on the current pixel of the input image we are processing and whose convolution with nearby elements we want to calculate.

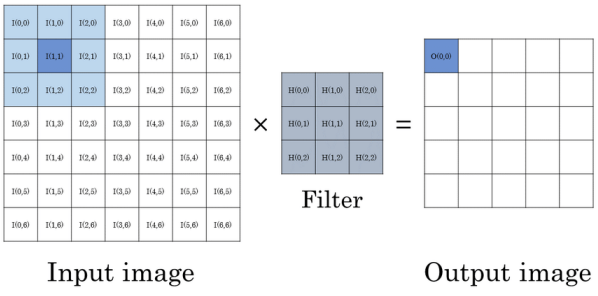


Figure 2: Image convolution with an input image of size 7×7 and a filter kernel of size 3×3.

In 1 is shown the pseudo-code of the convolution operation between an input image and a kernel. Note that the corresponding pixels of the input image are determined with respect to the origin of the kernel.

Algorithm 1 Convolution Operation.

Input:
- Original image f.
- Kernel ω.

Output:
- Processed image g.

```
for pixel in f do
  accumulator ← 0
  for element in ω do
    if element position corresponds to pixel position then
      accumulator += pixel * element
  Set output pixel in g to accumulator.
```

The convolution operation between an input image f of size $m \times n$ and a kernel ω of size $k \times h$ requires a complexity of $O(n \times m \times k \times h)$. Specifically, for each pixel of the input image f, it is necessary to perform a convolution with all neighboring pixels contained in the sliding window of filter kernel ω size.

During the convolution operation, it is necessary to handle boundary cases in which the convolution is performed on pixels located at the edges of the input image. In these cases, convolution requires values from pixels located outside the image boundaries. Several strategies, known as **padding**, are available to handle the problem of image edges:

- **Constant**: pixels outside the image are assumed to be a constant value. Typically we consider a value 0, so the contours of the output image will be bounded by a black frame.
- **Replicate**: pixel values along the edges are replicated (i.e. the nearest edge pixels are extended). This implies that pixels along the edge of the image get the same value as the nearest pixels within the image.
- **Mirror**: pixels along the edges are mirrored. That is, pixels on the outside of the image are mirrored to the nearest pixels inside the image.

In addition, to maintain the same brightness as the input image, we can implement a process of **normalization**. Normalization is defined as the division of each kernel element by the sum of all kernel elements, such that the sum of elements of a normalized kernel is 1. This ensures that the average pixel of the processed image has the same brightness as the average pixel of the original image.

3. Sequential implementation

The sequential implementation of the Kernel Image Processing in C++ is shown in 1.

To achieve this goal, we designed an `Image` class to encapsulate multi-channel image data. Each image is defined by a vector of unsigned char elements, representing the pixel values, and the three sizes of the image (width, height and channels).
To load image data, we used the public library `stb` [3], which allows to read data from an existing image in various formats. The same library was also used to save the processed image.
In addition, the `Image` class provides methods for converting the data structure from an *Array of Structures* (AoS) architecture, where pixels of each channel are alternately organized, to an *Structure of Arrays* (SoA) architecture, where pixels of each channel are sequentially organized (fig. 3). In this way, we can evaluate whether the spatial and temporal locality data access optimization provided by the SoA architecture is beneficial for a Kernel Image Processing problem. To determine whether the image is in the AoS or SoA format, we use a flag `is_SoA`. This simplifies data access based on the specified architecture.

Similarly, we developed a `Kernel` class that represents a kernel filter. Each kernel is defined by a vector of floating-point values representing its characteristics and the two sizes of the kernel (width and height).

The `Convolution` class is designed to execute the convolution operation sequentially. The `convolution` method applies the specified kernel to the original input image. For each pixel in the input image (previously padded by one of the selected methods), the convolution is performed by multiplying each pixel in the sliding window with the coefficients of the kernel. The result of this convolution is assigned to the corresponding pixel in the output image (the value is clamped between 0 and 255).

Listing 1 Convolution sequential implementation.

```
Image convolution(Image& image, Kernel& kernel,
↪  Image& padded_image) {
  // Initialize the output image.
  Image output_image = Image(width, height,
  ↪  channels);

  // Iterate over the image.
  for(int c=0; c<channels; c++) {
    for(int y=0; y<height; y++) {
      for(int x=0; x<width; x++) {
        // Output value for the current pixel.
        float output_value = 0;

        // Iterate over the kernel.
        for(int ky=0; ky<kernel_height; ky++) {
          for(int kx=0; kx<kernel_width; kx++) {
            // Get the pixel index to convolve.
            int col = x + kx - kernel_width / 2 +
            ↪  padding_width; // Column index.
            int row = y + ky - kernel_width / 2 +
            ↪  padding_height; // Row index.
```

Figure 3: Array of Structures (AoS) and Structure of Arrays (SoA) architectures.

```
            // Convolve the pixel.
            output_value += padded_image(col,
            ↪  row, c) * kernel(kx, ky);
          }
        }

        // Set the output value (clamped between
        ↪  0 and 255).
        output_image(x, y, c) = clamp(0.0f,
        ↪  output_value, 255.0f);
      }
    }
  }

  // Return the convolved image.
  return output_image;
}
```

To facilitate pixel access based on the image's AoS or SoA architecture, the `Image` class provides an overload of the call function operator illustrated in 2.

Listing 2 Image data access.

```
uint8_t &Image::operator()(int col, int row, int
↪  channel) {
    // Get the 1D pixel index.
    int pixel_index = is_SoA ? ((channel * width
    ↪  * height) + (row * width) + col) : ((row
    ↪  * width + col) * channels + channel);

    return data[pixel_index];
}
```

Similarly, in order to facilitate access to linearized organized kernel elements, the `Kernel` class provides an overload of the call function operator, as shown in 3.

Listing 3 Kernel data access.

```
float &Kernel::operator()(int col, int row) {
   // Get the 1D pixel index.
    const int kernel_index = (row * width) + col;

    return data[kernel_index];
}
```

The `Convolution` class also provides a `convolve` method to perform the whole operation. In detail, this method creates the padded image from the input image using a specified technique. Then, it performs sequential convolution for a fixed number of iterations and calculates the average time required for convolution. Therefore, it stores the relevant information in a text file for reference purposes. Finally, the processed image is saved to a specified path.

4. Parallel implementation

In order to optimize the convolution operation, were implemented different levels of parallelism. First, was implemented a low-level of parallelism exploiting the hardware features, by transforming the AoS (Array of Structures) architecture to SoA (Structure of Arrays). This is because the SoA architecture turns out to be more cache-friendly. In fact, during a read of an element from memory, some following values are also loaded into the cache due to the **memory burst**. Therefore, with the AoS architecture, an element may not be completely within a cache line, requiring an additional read to obtain all the values. This optimization is particularly critical for CUDA to allow consecutive threads to access consecutive memory addresses, ensuring **coalesced memory access**. This means that memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth.

The convolution operation represents an embarrassingly parallel problem. In fact, this operation depends exclusively on the pixel values in the original input image and the kernel coefficients. However, these values are only read and never changed during the convolution operation. As a result, each thread can independently access the data needed to perform the convolution operation. In addition, the pixel values in the output image are written only once and by only one thread. Therefore, each thread can read, perform the convolution operation and write the output to memory without the need of synchronization with other threads.

CUDA [4] is a parallel computing framework that can be used to develop applications that can be executed on Graphics Processing Units (GPUs). Using this platform, we can speed-up the parallel section of the code using the high number of cores provided by a streaming multiprocessor contained in a GPU. A CUDA application consists of two parts:

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

In addition, CUDA allows the developer to manage memory allocation on the GPU. Specifically, three different types of memory can be used (fig. 4):

- **Global memory**: is the main and slowest memory in the device. Global memory is shared among all blocks in the grid.
- **Constant memory**: is a small and fast memory that can be used to store variables that will not be changed during processing. This type of memory is shared among all blocks in the grid. However, a thread can only read from this memory.
- **Shared memory**: is a small and low-latency memory, shared among all threads in a block. This memory can be used to store data that will be reused by threads in the block.

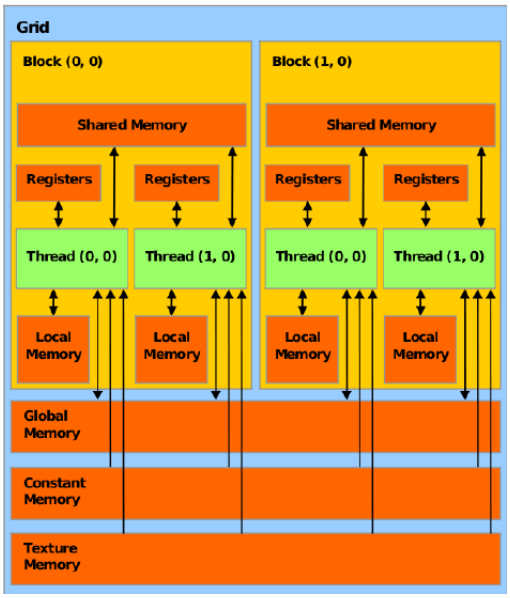


Figure 4: CUDA memory model.

To implement the parallel solution in CUDA, the first step is to allocate memory on the device for reserve space needed to store the original input image, the kernel filter and the processed output image. This step can be completed using the function `cudaMalloc` provided by CUDA.
Next, it is necessary to perform the transfer of input image data and kernel filter coefficients from host to device, in order to enable parallel processing on the GPU. This is accomplished by using CUDA's `cudaMemcpy` function in `cudaMemcpyHostToDevice` mode, which facilitates the transfer of data from the host memory to the device's global memory. In case we want to take advantage of constant memory to store the kernel as a constant variable that doesn't change, we can use CUDA's `cudaMemcpyToSymbol` function.
After that, we run the CUDA kernel to perform the convolution operation on the data transferred to the GPU. This involves instantiating an appropriate number of grids and blocks to make the best use of GPU resources. Specifically, we use 256 threads for each block and any grid consists of enough blocks to process the entire image (rounded up).
When the convolution operation is complete, it is necessary to transfer the processed output image from the device's global memory to the host memory. This can be achieved again through the `cudaMemcpy` function, but this time in `cudaMemcpyDeviceToHost` mode.

In order to observe the different behaviors in device memory utilization, three different CUDA kernels were implemented, which we briefly describe below.

4.1. Global memory

In this kernel implementation, we use only global memory. Therefore, the original input image and the filter kernel are transferred to the device's global memory. Once this transfer phase is completed, each thread is assigned to evaluate the convolution for a single pixel. Since each thread is dedicated to computing a single pixel, it is necessary to have a number of threads at least equal to the total number of pixels in the image. The coordinates of the pixel in the original input image which thread accesses, are determined using the thread and block index within the grid, as follows:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

4.2. Constant memory

In this kernel implementation, we transfer the original input image data to device's global memory and the the filter kernel coefficients to constant memory, which will only be read by the threads. This solution allows each thread to access kernel elements directly from constant memory instead of global memory, taking advantage of the higher speed access. The kernel implementation remains approximately the same as the previous one.

4.3. Shared memory

In this kernel implementation, in addition to using constant memory for fast access to the kernel filter, we also use shared memory among threads in a block to optimize access to shared data. In fact, threads in a block may need common pixels to perform convolution computation. Therefore, storing these pixels in a shared memory could reduce the overall number of accesses to global memory, which is characterized by a higher access cost. However, the shared memory has a smaller size than the global memory. Therefore, in order to best exploit this memory resource, we adopt the *tiling* technique, which consists of splitting data into tiles of smaller size. For this purpose, after loading the original input image into the global memory, each block provides for loading the corresponding pixel tile into the shared memory.

After this loading phase is completed, each thread in a block performs the convolution computation for the pixel it is associated on the output image. The CUDA function `__syncthreads()` is used to ensure synchronization of all threads within the block, avoiding race conditions while accessing shared memory.

We observe that, in the described implementation, in addition to loading the elements belonging to the tile (which has the same size as a block) into the shared memory, also the elements outside the edge of the tile are loaded. This solution allows to compute the convolution of pixels at the edges of the tile. Consequently, for each tile, $(\texttt{tile\_width} + \texttt{kernel\_width} - 1)^2$ elements are loaded into the shared memory.

In this implementation, each block has the same size as the *tile* on which it is to operate. Consequently, to also include the neighborhood of pixels at the edge of the *tile*, it is necessary for some threads to perform a second batch load operation. In fact, due to the edge conditions, the number of pixels to be loaded into the shared memory is greater than the *tile* size. Therefore, it is greater than the number of threads in the block instantiated to load the data. Thus, a two-batch loading process is implemented (fig. 5).
However, this solution involves each thread being mapped to the computation of a pixel in the output image, thus ensuring that each thread performs the convolution operation.

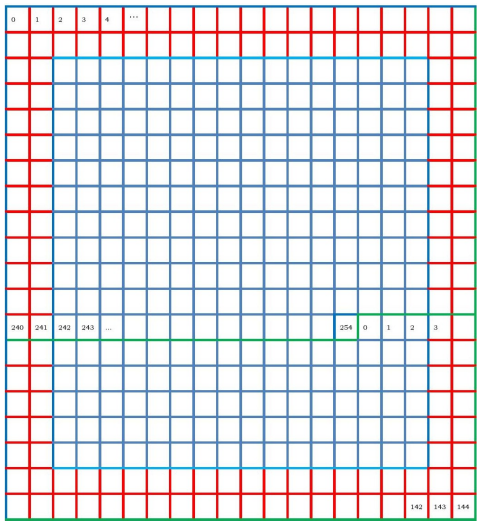


Figure 5: Tiling process for loading data into the shared memory. The blue boxes represent the elements of the tile, while the red boxes correspond to the elements of the neighboring tile, which are needed to handle edge conditions. The union of the blue and red boxes represents the total shared memory locations. Since the size of the tile is smaller than the actual number of elements to be loaded, some threads highlighted in green have to perform a second loading step.

The use of shared memory offers the benefit of reducing the overall number of global memory accesses, which is particularly useful in situations where shared data must be read repeatedly by threads in a block, as in the case of convolution.

4.4. Asynchronous loading

Pinned memory refers to a section of memory that is physically locked in RAM, enabling direct access by the device without CPU involvement. This direct access is allowed through the Direct Memory Access (DMA), bypassing the virtual memory paging process. In particular, the operating system does not have access to the pinned memory area.

During a synchronous transfer, when pageable memory is used, the data access can trigger a page fault, resulting in increased overhead. During the data transfer process, the driver must access each page of pageable memory, copy it to the pinned buffer and pass it to the DMA.

In contrast, an asynchronous transfer involves the allocation of pinned memory on the host. This allows for a preliminary host-to-host transfer from a pageable memory space to the allocated pinned memory space. Then the asynchronous transfer to the device can be performed (fig. 7). This approach reduces the overhead of data transfer between host and device. Consequently, pinned memory reduces the data copy time between the CPU and the GPU.

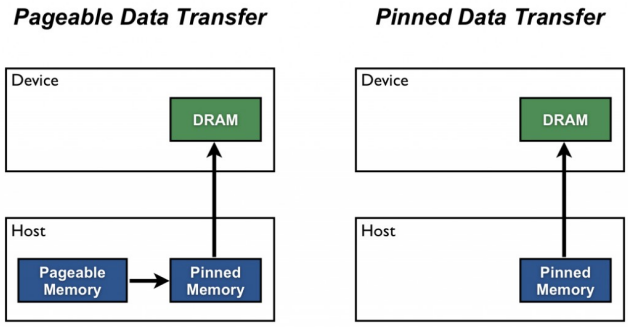


Figure 6: Differences between using pageable memory and pinned memory.

CUDA streams further improve the efficiency of data transfers by enabling concurrent execution of multiple operations (e.g. while one kernel is actively executing, we can perform data transfer within device memory). This concurrent processing approach optimizes the overall speedup of GPU-accelerated applications, reducing idle times.
Kernel executions, host-to-device memory copy and device-to-host memory copy that do not specify a stream parameter or, equivalently, that set the stream parameter to zero are executed on the default stream [5].

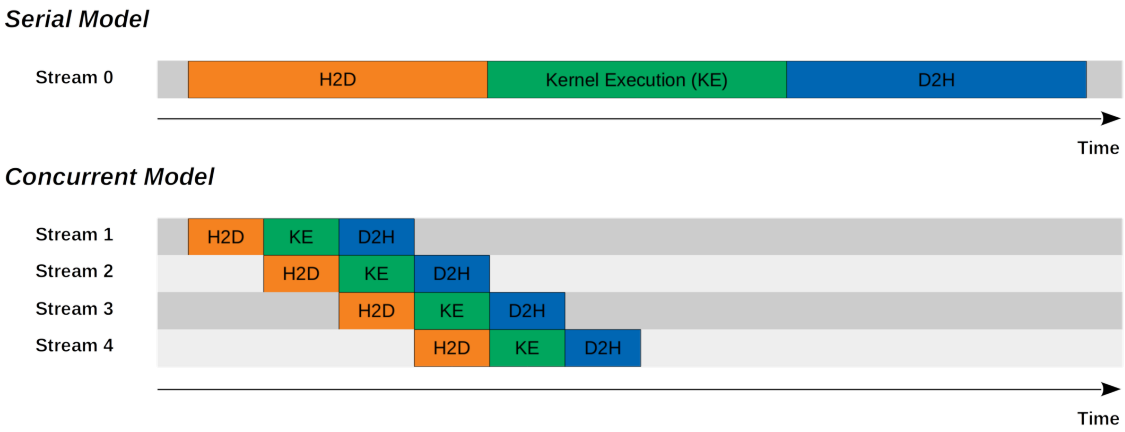


Figure 7: Serial and concurrent execution models.

5. Performance

We now perform a detailed performance analysis of the convolution operation on an Intel i7-10750H processor and an NVIDIA GeForce RTX 3060 GPU. The performance analysis includes a study of speedup as input image resolution and filter size change. In addition, we make a comparison between the Array of Structures (AoS) and the Structure of Arrays (SoA) architectures. Finally, let's see how it improves the speedup compared to asynchronous loading implementation.

5.1. Speedup

A detailed speedup analysis of the implementation was conducted. For each analysis, the algorithm was run repeatedly for 15 iterations and considering the average execution time.

The first evaluation aims to measure the speedup of the parallel code in relation to image resolution. For this purpose, we examined five different image resolutions: 853×480px (referred to as **480p**), 1280×720px (referred to as **720p**), 1920×1080px (referred to as **HD**), 3840×2160px (referred to as **4K**) and 7680×4320px (referred to as **8K**). In this analysis, we used a 3×3 Gaussian blur filter as a benchmark to evaluate the performance of the parallel code. The results of this experiment are shown in Table 1.

Memory / Resolution	**Global**	**Constant**	**Shared**
480p	3645.86	7709.04	6682.36
720p	4929.97	9681.91	8251.72
HD	8534.28	11207.86	9084.96
4K	10602.63	12181.03	9990.22
8K	13780.7	**15118.17**	12433

Table 1: Speedup for different image resolutions.

Figure 8 provides a visual representation of the speedup, considering the three different memory levels described above.
The maximum speedup recorded was 15118.17, achieved by using constant memory for storing the kernel filter. The results are consistent with expectations and show an increase in speed with the maximum value recorded in the context of the largest image. However, we note that shared memory didn't prove to be the optimal solution. This could be attributed to inefficient tiling management or the limited number of multiprocessor streams in the device on which the analysis was performed.

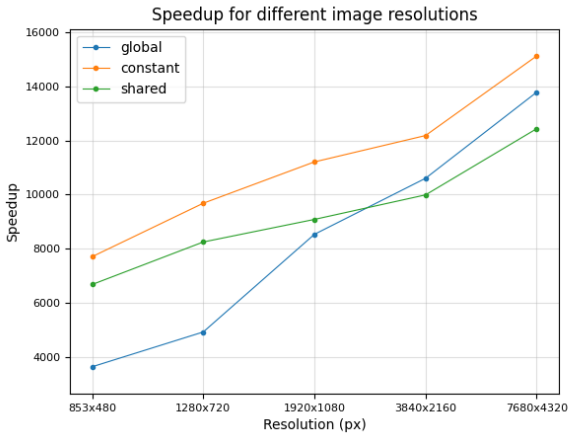


Figure 8: Speedup graph for different image resolutions.

We proceeded to analyze the speedup of the parallel code concerning different kernel filter sizes. To conduct this analysis, we employed the HD image as a benchmark and tested the code with four distinct filter sizes: 3×3, 5×5, 7×7 and 9×9.
The results of this experiment are shown in Table 2.

Filter \ Memory	Global	Constant	Shared
3×3	8554.41	11232.12	9133.13
5×5	11578.48	13738.88	14922.6
7×7	13550.8	15232.74	20230.6
9×9	14236.82	15679.22	**24850.18**

Table 2: Speedup for different filter sizes.

Figure 9 provides a visual representation of the speedup. The maximum speedup recorded is 24850.18. As expected, the speedup shows an increase corresponding to the increase in filter size. Specifically, the results illustrates a proportional increase in speedup with the expansion of filter size, with the maximum value observed in the context of the largest filter. Interestingly, optimal performance of shared memory is observed in this scenario, showcasing an increased speedup as the filter size grows. This result may be due to the fact that the tile size increases when more neighboring data are considered, thus facilitating data locality. With larger filter sizes, the tile becomes a more significant working unit and the shared memory is more efficient in loading and reusing the portions of data needed for convolutional operations.

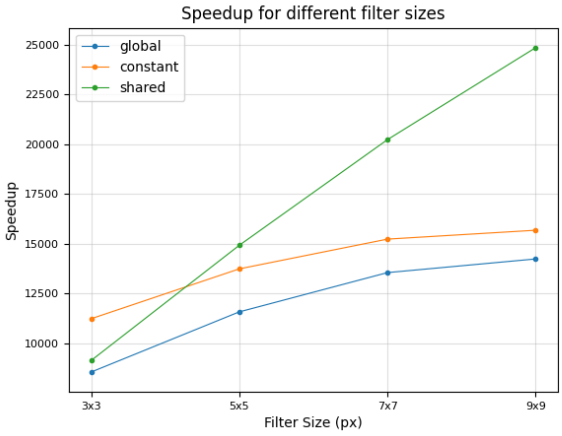


Figure 9: Speedup graph for different number of clusters.

5.2. AoS vs SoA

Now, we proceed with a comparison of execution times between the implementation based on the AoS (Array of Structures) architecture and the implementation based on the SoA (Structure of Arrays) architecture. AoS and SoA architectures represent different approaches in storing and organizing data in a program. At the hardware level, the use of SoA offers significant advantages in terms of parallelism and efficient memory access. The SoA architecture allows temporal and spatial locality of data in memory, facilitating more efficient access to contiguous data. This makes the data structure more cache-friendly and allows coalesced memory access.

To conduct this comparison, we executed the code across 5 different image resolutions as described earlier (480p, 720p, HD, 4K, and 8K).
Figure 10 shows the comparison of execution times. The results illustrates that the SoA architecture in this application leads to a little more efficient solution, with a speedup of about 1.03.

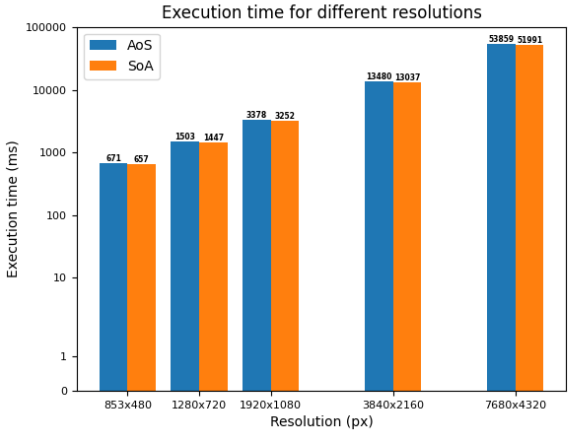


Figure 10: Comparison of AoS and SoA execution times for different image resolutions.

5.3. Asynchronous loading

Finally, we examine the execution time comparison between the synchronous implementation using pageable memory and the asynchronous implementation using pinned memory and CUDA streams. In this context, we adopted three CUDA streams to split the computation of input data into three different kernel executions, thus reducing the idle time when one kernel executes while loading data for the next. This implementation results in a significant reduction in parallel program execution time.

To conduct this comparison, we ran the code again on 5 different image resolutions (480p, 720p, HD, 4K and 8K).
Figure 11 shows the comparison of execution times. The results illustrates that the asynchronous model provides a significantly better solution, as expected, increasing speedup as the input size increases. However, it is important to note that for small input sizes, this solution may not lead to significant improvements, as overhead costs may emerge that overcome the benefits obtained.

6. Conclusions

We have seen that the convolution operation has been implemented in various ways, using a sequential and parallel approach. Adopting a parallel approach with CUDA has

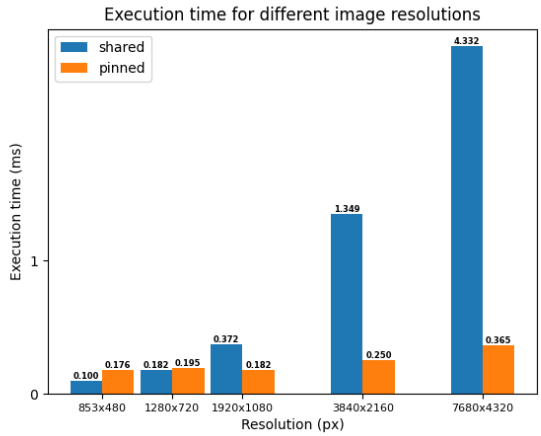


Figure 11: Comparison of synchronous and asynchronous models execution times for different image resolutions.

proven to be particularly advantageous, enabling a significant improvement in computational performance. CUDA streaming multiprocessors can can exploit the increased number of threads to efficiently perform the convolution operation on large images, with benefits that increase with the number of pixels and kernel size.
In addition, the application of optimization techniques, such as asynchronous loading and organizing the data in SoA architecture, helped to achieve some improvements over the sequential implementation. These arrangements helped to optimize the overall speedup of the process.

References

[1] Wikipedia contributors, "Kernel (image processing)." https://en.wikipedia.org/w/index.php?title=Kernel_(image_processing)&oldid=1180652863, 2023. 1

[2] Wikipedia contributors, "Channel (digital image)." https://en.wikipedia.org/w/index.php?title=Channel_(digital_image)&oldid=1191674988, 2023. 1

[3] nothings, "stb." https://github.com/nothings/stb, 2023. 3

[4] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89." https://developer.nvidia.com/cuda-toolkit, 2023. 4

[5] Lei Mao, "Cuda streams." https://leimao.github.io/blog/CUDA-Stream/, 2023. 6