Kernel Image Processing with CUDA

Tommaso Botarelli
tommaso.botarelli@edu.unifi.it
7136210

Abstract

Kernel-based image processing is a fundamental technique used in computer vision, medical imaging, and artificial intelligence. However, as image resolutions increase, applying kernel-based filters sequentially on CPUs becomes a computational bottleneck, especially for real-time applications. This paper explores the acceleration of kernel-based image processing using CUDA (Compute Unified Device Architecture) to leverage the parallel computing power of GPUs. We compare a sequential CPU implementation with multiple CUDA-based parallel implementations, evaluating their computational complexity, execution time, and efficiency. Experimental results demonstrate a significant speedup achieved with CUDA, highlighting the benefits of parallelization in reducing processing time.

1. Introduction

Image processing plays a crucial role in various fields, including computer vision, medical imaging, and artificial intelligence. One of the fundamental techniques in image processing is kernel-based filtering, which involves applying a small matrix (kernel) to modify pixel values in an image. This operation is used for tasks such as edge detection, blurring, and sharpening. However, as image resolutions continue to increase, the computational cost of applying kernel-based filters becomes a significant bottleneck, particularly when processing large datasets in real-time applications.

Traditional sequential implementations of kernel filtering process each pixel individually, leading to high execution times, especially for large images and complex filters. To address this performance challenge, parallel computing techniques can be leveraged to accelerate processing. Among these techniques, CUDA (Compute Unified Device Architecture) provides a powerful framework for harnessing the computational capabilities of GPUs, which excel in handling parallel workloads.

In this paper, we explore the problem of kernel-based image processing and compare two implementations: a sequential approach executed on a CPU and a parallel implementation using CUDA to run on a GPU. We analyze the computational complexity of both approaches and evaluate their performance in terms of execution time and efficiency. The results highlight the significant speedup achieved with CUDA, demonstrating the importance of GPU acceleration for real-time image processing applications.

By leveraging CUDA, we can process images orders of magnitude faster than traditional CPU-based methods, making it an essential tool for high-performance image processing tasks. This study underscores the necessity of parallel computing in modern image processing and provides insights into the benefits and challenges of implementing CUDA-based solutions.

2. Problem Definition

Given a **grayscale** or **color image** represented as a two-dimensional matrix I of size $H \times W$ (for a grayscale image) or a three-dimensional matrix I of size $H \times W \times C$ (for a color image with C channels), and a **convolution kernel** (or filter) K of size $k \times k$, the goal is to compute a new image I', where each pixel value $I'(x, y)$ is determined by applying the kernel K over a local neighborhood of $I(x, y)$.

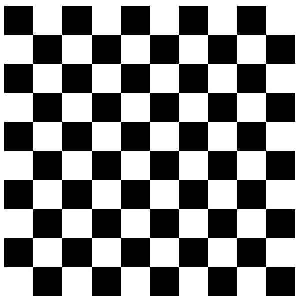
2.1. Convolution Operation

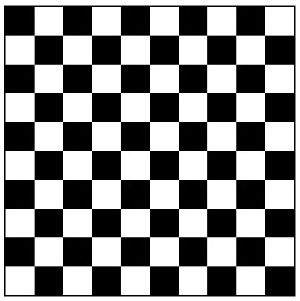
Formally, the output pixel value at position (x, y) is computed as:

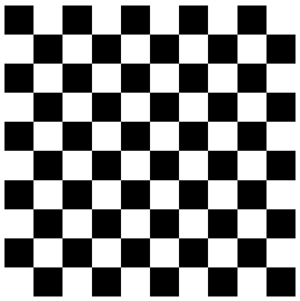
$$I'(x,y) = \sum_{i=-\frac{k}{2}}^{\frac{k}{2}} \sum_{j=-\frac{k}{2}}^{\frac{k}{2}} K(i,j) \cdot I(x+i, y+j)$$

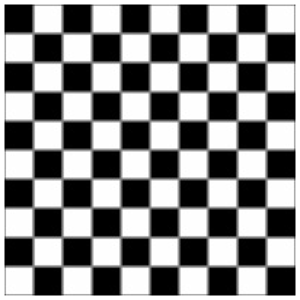
where:

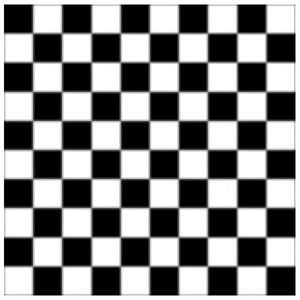
- $K(i, j)$ is the kernel weight at position (i, j).
- $I(x + i, y + j)$ is the pixel value from the input image at the corresponding location.


(a) Initial Image


(b) Image with zero padding


(c) Identity kernel


(d) Blur kernel


(e) Gaussian kernel

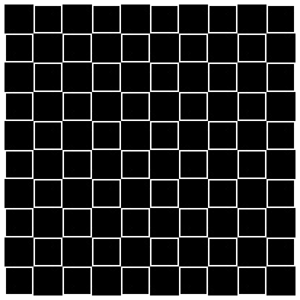

(f) Edge kernel

Figure 1: Image and different types of kernel applied to it. The kernel size is 3 and the zero padding is set to add black pixels.

- The summation iterates over all elements of the kernel, applying the weighted sum over the local neighborhood.

2.2. Boundary Handling

Since the kernel accesses neighboring pixels, boundary conditions must be addressed. Common strategies include:

- **Zero padding**: Setting out-of-bounds pixels to zero.
- **Replication padding**: Extending the edge pixels.
- **Reflection padding**: Mirroring the pixel values at the boundary.

2.3. Computational Complexity

For an image of size $H \times W$ and a kernel of size $k \times k$, the computational complexity of applying the kernel to the entire image is:

$$O(H \cdot W \cdot k^2)$$

which becomes computationally expensive for large images and large kernels, especially in real-time applications.

2.4. Types of Kernel Filters

Different kernel filters are used in image processing for various purposes. Below are some common types:

2.4.1 Identity Kernel

The identity kernel leaves the image unchanged:

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

2.4.2 Edge Detection Kernels

Edge detection kernels highlight the edges in an image by detecting changes in intensity.

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

2.4.3 Gaussian Blur Kernel

A Gaussian blur smooths an image by averaging pixel intensities using a Gaussian function:

$$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

2.4.4 Box Blur Kernel

A box blur (or average filter) replaces each pixel with the average of its neighbors:

$$K = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

3. Sequential Implementation

In this section we provide a simple sequential implementation to address the problem of Kernel Image Processing. The computational complexity exposed in 2.3 can be viewed in 1 as a 4 level nested loop.

In the following sections we assume that the number of channel of the image is 3 (we assume that the image are RGB).

```
for each channel in input image:
 for each image-row in input image:
   for each pixel in image row:

       set accumulator to zero

       for each kernel-row in kernel:
          for each element in kernel-row:

             if element position
                 corresponding* to pixel
                 position then
                multiply element value
                    corresponding* to pixel
                     value
                add result to accumulator
             endif

       set output image pixel to
           accumulator
```

Algorithm 1: Pseudo code for the sequential implementation [5]

4. Parallelization

In this section we provide an explanation of why it's useful and important to create a parallelized implementation for the problem of Kernel Image Processing and we introduce CUDA. Then we provide different CUDA implementation of the algorithm that will be used in the experimentation.

4.1. Necessity of Parallelization in Kernel Image Processing

Kernel-based image processing is inherently computationally intensive as we've already talked in 2.3.

As image resolutions increase, the number of pixel operations grows significantly, making real-time processing infeasible on traditional CPU architectures. Since each pixel in the output image can be computed independently from the others, the problem is highly **parallelizable**. This makes kernel-based filtering an ideal candidate for GPU acceleration.

Modern GPUs contain thousands of cores that can execute computations simultaneously, making them well-suited for parallel workloads like image filtering. By mapping individual pixel computations to separate GPU threads, we can achieve significant speedups compared to a sequential CPU implementation.

4.2. CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform developed by NVIDIA that allows developers to leverage GPUs for general-purpose computations. It enables fine-grained control over thread execution, shared memory usage, and data transfers.

```cpp
struct Image {
   long long int rows;
   long long int cols;

   short int* channels[3];
}
```

Algorithm 2:]C++ code: Image struct used for the sequential algorithm implementation. Using structure of arrays (SoA) allows to reach better performances for the sequential code. `short int` type for pixel values is chosen to optimize both the sequential (due to vectorization) and the parallel code (it reduces the size of memory moved between device and host). The type is ok because we know that the value of a pixel can be a number in [0, 256]

4.3. GPU Data Transfer Overhead

One of the main challenges when using CUDA is the overhead associated with moving data between the **host** (CPU) and the **device** (GPU). The steps for executing a CUDA kernel typically involve:

- **Transferring the input image from CPU to GPU (host to device).**
- **Executing the kernel (parallel computation on GPU).**
- **Retrieving the processed image from GPU to CPU (device to host).**

The data transfer between the host and device occurs over the PCIe (Peripheral Component Interconnect Express) bus, which has a much lower bandwidth compared to the GPU's internal memory. The cost of these memory transfers can significantly impact overall performance.

To mitigate this issue, one of the best optimization is **Minimizing data transfers**. Indeed reducing unnecessary memory copies by keeping data in GPU memory as long as possible can lead to better overall performances.

By carefully managing data transfers and optimizing memory access patterns, the performance bottleneck can be minimized, making CUDA-based parallelization an effective approach for high-performance kernel image processing.

4.4. Algorithms Implementations

In this work we provide several algorithm implementations. The implementations are provided to give two different parallelization perspective. In the following experimentation we will observe the advantages/disadvantages of each solution in the practice, here we provide an overview of each method.

```
...
int pixelsPerChannel = resultImageRows *
    resultImageCols;
int threadsPerBlock = 1024;
int numBlocks = (pixelsPerChannel +
    threadsPerBlock - 1) / threadsPerBlock;
...
for (int channelIndex = 0; channelIndex <
    3; channelIndex++){
      cudaMemcpy(d_image, h_image->channels
          [channelIndex], rows*cols*sizeof(
          short int), cudaMemcpyHostToDevice
          );
      kernelProcessing<<<numBlocks,
          threadsPerBlock>>>(d_image,
          d_kernel, d_resultImage,
          resultImageCols, paddingSize, cols
          , kernelCols, pixelsPerChannel);
      cudaMemcpy(h_resultImage->channels[
          channelIndex], d_resultImage,
          h_resultImage->cols*h_resultImage
          ->rows*sizeof(short int),
          cudaMemcpyDeviceToHost);
   }
...
```

Algorithm 3: C++ code: code snippet for 1-channel based implementation. Note that we need to pass tot he kernel function all the arguments used in it and we don't use `__constant__`.

4.4.1 1 Channel Based Parallelization

The first implementation has the main idea of parallelize the execution in a single channel. Then we can reconstruct the image by loop over the three channel applying the kernel code (code executed in parallel by the GPU) three times in a row.

The main advantage of this technique is that it uses the standard data structure of the sequential algorithm. Indeed with CUDA it is not advantageous to move entire data structures (`struct` in c++, see Algorithm 2) but it is convenient to move simple arrays. Move simple arrays can be done naturally with this first implementation.

Despite this, however, the main disadvantage is due to the repeated moving of data between host and device. For each channel it will be necessary to move the array of values for that channel into GPU memory and after computation of convolutions it will be necessary to move it back into host memory. Additionally we need to pass as argument several values used in the kernel function (as the kernel values, it's size, ...). This lead to have each threads with his personal copy of the variable.

```
short int* linearizeImage(Image* image) {
  int channelSize = image->rows * image->
      cols;
  int N = channelSize * 3;
  short int* pixels = new short int[N];

  for (channelIndex = 0; channelIndex < 3;
      channelIndex++) {
     for (i = 0; i < image->rows; i++) {
        for (j = 0; j < image->cols; j++) {
           pixels[channelIndex*channelSize +
                i*image->cols + j] = image->
               channels[channelIndex][i*
               image->cols + j];
        }
     }
  }

  return pixels;
}
```

Algorithm 4: C++ code: linearize function. It creates a big array from the three channels image. The array is built by linearize channel by channel the image.

4.4.2 1 channel Based Parallelization with `__constant__`

In CUDA code we can use `__constant__` to improve the memory management [4]. The key advantages are:

- The constant memory is optimized for read operations.

It resides in a special part of global memory that benefits from a dedicated constant cache

- When multiple threads access the same constant value, the access is broadcast efficiently, leading to lower memory latency compared to global memory

So in this implementation the main idea is to follow 4.4.1 but using `__constant__` for all the variables used in a read only access in the kernel function. Even the kernel can be stored in constant memory. This is ok because usually the kernel isn't a big matrix (at max in the order of thousands of values in kernel image processing) so we can store it in constant memory without go beyond the limit of 64 Kb.

```
// host code
// 2D grid for numBlocks with 3 channels
dim3 threadsPerBlock(n, n);
dim3 numBlocks((rows + threadsPerBlock.x -
    1) / threadsPerBlock.x, (cols +
    threadsPerBlock.y - 1) / threadsPerBlock
    .y, 3);
...
kernelFunction<<<numBlocks, threadsPerBlock
    >>>(...);

// kernel function
// i and j are constructed from the thread
    coordinates
int i = blockIdx.x * blockDim.x + threadIdx
    .x;
int j = blockIdx.y * blockDim.y + threadIdx
    .y;
int channelIndex = blockIdx.z;
```

Algorithm 5: C++ code: first implementation for the memory coalescing problem. The indices of the pixel to process are taken from the thread and block coordinates

4.4.3 3 Channel Based Parallelization

One big disadvantage of using the previous techniques is that the parallelization doesn't reach the 100%. Indeed, as we can see in the snippet of code of Algorithm 3, before and after each call to the kernel function we need to move data in and out the GPU.

For this reason in this implementation the main idea is to transform the `struct Image` values of the three channels into a single big array of pixels representing all the channel linearized (see 4). Then after the computation a reconstruction of the image is requested.

The main disadvantages of this method is the time cost for the linearization of the array. This mechanism can introduce delay even bigger than the memory one but we can reduce the impact of it with parallelization on CPU side (for example using OpenMP).

```
// host code
// 1D grid for numBlocks with 3 channels
int threadsPerBlock = n;
dim3 numBlocks((cols * rows +
    threadsPerBlock - 1) / threadsPerBlock,
    1, 3);
...
kernelFunction<<<numBlocks, threadsPerBlock
    >>>(...);

// kernel function
// first calculate the global index in the
    image
// then use blockIdx.z as the channel index
int index = blockIdx.x * blockDim.x +
    threadIdx.x;
int i = index / const_resultCols;
int j = index % const_resultCols;
int channelIndex = blockIdx.z;
```

Algorithm 6: C++ code: second implementation for the memory coalescing problem. The indices of the pixel to process are built from the global thread index

Memory Coalescing. When we use this method we have to organize the thread access to the data and to distribute the workload. CUDA provides a a hierarchy of threads and blocks to organize the computation. Indeed these construct can be organized in a grid (see Figure 2). The cartesian grid of both threads and blocks can be defined using `Dim3` type. Then starting from these coordinates each thread can calculate the index that has to process in the image. How the data is accessed inside each block can impact the performances, so in this work we presents two different implementations to address this problem (see Algorithm 5 and Algorithm 6).

4.4.4 3 Channel with 3 Different Arrays

The last method implemented in this work has the main idea of passing as argument all the different three channel in three different arrays structure. This method allows each thread to calculate each index in the image in each channel. So we add some computation to each thread but `Image` isn't linearized (see 7).

5. Experimentation

In this section we provide an overview on the experiments performed. We first show the system used for experimentation and then we explain in the detail the experimentation setup.

Average Times [ms]						
Kernel Size	$1channel_{noConst}$	$3channel_{grid}$	$3channel$	$3channel_3$	$1channel$	Seq
3	1.89	8.73	2.83	2.48	2.13	28.88
9	2.48	11.22	4.27	2.66	2.38	232.30
17	4.87	8.27	4.86	3.65	3.67	1004.94
33	11.76	23.23	10.19	9.42	9.36	3940.20
63	39.07	90.89	30.55	30.00	29.71	15379.80
121	142.25	467.18	164.62	104.56	163.21	60764.00
Speedup						
3	15.25	3.31	10.21	11.63	13.57	-
9	93.86	20.71	54.45	87.21	97.48	-
17	206.46	121.53	206.63	275.17	273.48	-
33	334.99	169.63	386.60	418.24	420.88	-
63	393.62	169.21	503.37	512.63	517.71	-
121	427.17	130.07	369.11	581.16	372.30	-

Table 1: Fixed image size experimentation results

5.1. System Overview

The entire experimentation has been conducted in a dualsocket Intel Xeon Silver 4314 system with 32 physical cores. Then the GPU used for the CUDA section is a **NVIDIA RTX A2000 12GB**. All the infos about it are shown in Table 2.

```
__global__ void
    kernelProcessing_threeChannelTogether(
    short int* inputImage0, short int*
    inputImage1, short int* inputImage2,
    short int* resultImage0, short int*
    resultImage1, short int* resultImage2)
    {...}

// host code for calling the function
kernelFunction<<<numBlocks, threadsPerBlock
    >>>(d_imagePixels_channel0, ...,
    d_resultImagePixels_channel0, ...);
// then the results are copied back
// to memory directly in the struct
cudaMemcpy(h_resultImage->channels[0],
    d_resultImagePixels_channel0,
    pixelsPerChannel*sizeof(short int),
    cudaMemcpyDeviceToHost);
```

Algorithm 7: C++ code: Kernel function signature and how the host call it for the last implementation with 3 different arrays for each channel (input and output)

5.2. Experimentation's Setup

Each implementation has been compared to the sequential code performance in two tests:

- *Fixed image size and kernel size increasing*. With this configuration the goal is to compare the performance of the parallel code with the sequential code with increasing size of the kernel.
- *Fixed kernel size and image size increasing*. With this configuration the goal is to compare the performance of the parallel code with the sequential code with increasing size of the kernel.

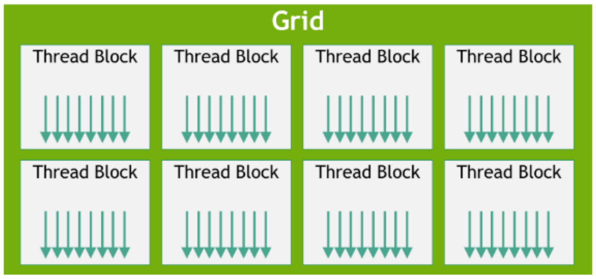


Figure 2: Threads and Blocks organization in CUDA [2]

The two tests are a sort of **weak scaling**, a standard experiment to test the robustness of a parallel code to scale with increasing data size.

For each test the execution time of the algorithm has been collected from an average of multiple execution to have a robust measure. The execution time doesn't include the image creation (indeed an image generator to create image like 1 has been implemented) and the data initialization. For the algorithms that make use of linearization and delinearization of the image the execution time includes the time occurred for those operations.

The code provide the options to select multiple kernel filter. For all the experimentation the `BLUR` filter has been used.

Specification	Details
Memory Interface	192-bit
Memory Bandwidth	288 GB/s
Error-Correcting Code (ECC)	Yes
CUDA Cores (Ampere)	3,328
Tensor Cores (3rd Gen)	104
RT Cores (2nd Gen)	26
Single-Precision Performance	8.0 TFLOPS
RT Core Performance	15.6 TFLOPS
Tensor Performance	63.9 TFLOPS
System Interface	PCIe 4.0 x16
Power Consumption	70 W (Total Board Power)
Thermal Solution	Active

Table 2: Some infos about NVIDIA RTX A2000 12GB GPU used for the experimentation. For a full list of the specs see [3]

5.3. Nomenclature

In this section a nomenclature used for the results representation is shown. Indeed we provide a simple and concise nomenclature for each implementation:

- Seq, sequential implementation;
- $1channel_{noConst}$, CUDA implementation of Section 4.4.1;
- $1channel$, CUDA implementation of Section 4.4.2;
- $3channel_{grid}$, CUDA implementation of Section 4.4.3 with the implementation shown in Algorithm 5;
- $3channel$, CUDA implementation of Section 4.4.3 with the implementation shown in Algorithm 6;
- $3channel_3$, CUDA implementation of section 4.4.4;

For all the implementations from $1channel$ down to $3channel_3$ the `__constant__` variables are used for the kernel values, number of columns of the result and input image, number of rows of the result and input image, etc.

6. Results

In this section the results of the experimentation are shown. We first provide an overview over the Fixed Image Size and then we move on the Fixed Kernel Size weak scaling.

6.1. Fixed Image Size

In this experiment the size of the image is fixed to a resolution of $1000x1000$ ($1'000'000$ values). Then the kernel size is increased as we can see in Table [N]. All the CUDA implementations times are the result of an average over 1000 executions. The average time for Seq is given from an average of $1000, 100, 50, 10, 5, 2$ executions respectively. All the results are shown in Table 1.

As can be seen in Figure 3, the overall trend appears to be of higher speedup as the kernel size increases. This is exactly what we could expect since the complexity of the algorithm is quadratic to the size of the kernel. By going to parallelize the code the complexity savings allows the algorithm to run with increasing speedup.

Furthermore, from the table and figure it can be seen that the addition of the `__constant__` variables had a positive impact in performance. In fact, a general improvement in the speedup performance of $1channel_{noConst}$ and $1channel$ can be seen. This improvement in the performance is robust to the kernel size scaling.

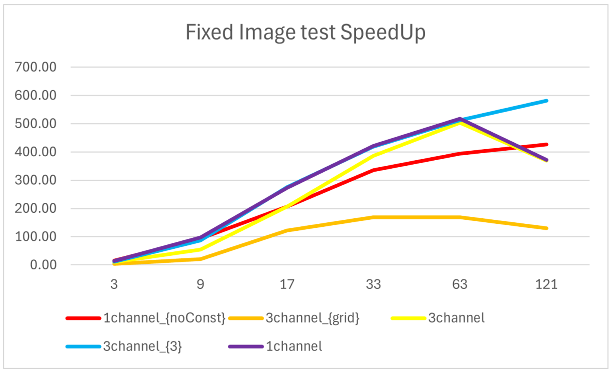


Figure 3: Fixed Image speedup

The advantages of using constant memory for various parameters and kernel values turn out to be robust to increasing kernel size. It must be said, however, that such memory has a maximum capacity and that such methods in the case of exceeding this capacity cannot be executed. In general however, in the context of kernel image processing, even relatively large kernels (> 63) can be handled using constant memory taking advantage of the benefits that this experimentation confirms.

Another point that can be observed is the performance of the $3channel_{grid}$ method. This unlike $3channel$ exhibits a marked deterioration in speedup. This is caused to an optimized organization of data in memory in the second method. Indeed thread inside the same block access neighbour data values.

As a final observation we can see that despite the sequential image linearization performed by $3channel$ the impact on performance in the case of small image sizes is rather small. The application of this method, assuming CPU-side parallelization of the linearization and delinearization methods could be in most cases optimal.

So in general from this test we can say that $3channel$

Average Times [ms]						
Image Size	$1channel_{noConst}$	$3channel_{grid}$	$3channel$	$3channel_3$	$1channel$	Seq
100x100	0.21	0.25	0.21	0.25	0.20	4.00
100x1000	0.47	0.80	0.60	0.46	0.45	44.45
1000x1000	2.91	11.10	4.34	2.88	2.60	462.52
1000x10000	26.88	106.37	90.43	23.71	23.56	4317.20
10000x10000	521.08	1225.91	1027.72	481.76	489.77	42243.00
10000x100000	5828.72	7413.29	5904.32	5323.00	5354.60	438437.00
Speedup						
100x100	19.41	16.13	19.12	16.13	20.07	-
100x1000	95.11	55.32	73.53	95.69	99.65	-
1000x1000	158.81	41.68	106.67	160.50	178.11	-
1000x10000	160.60	40.59	47.74	182.11	183.26	-
10000x10000	81.07	34.46	41.10	87.68	86.25	-
10000x100000	75.22	59.14	74.26	82.37	81.88	-

Table 3: Fixed kernel size experimentation results

despite the linearization of the image into arrays (and subsequent reconstruction) allows us to achieve similar performance as with $3channel_3$ with, however, greater generalization of the algorithm. In fact, this method could also be adapted to "images" with more than three channels (one can thus handle even multidimensional arrays efficiently) while $3channel_3$ is adaptable with more difficulty to an extension of the input array size. Experimentation has also confirmed the advantages that good use of constant memory brings to the execution of this type of algorithm.

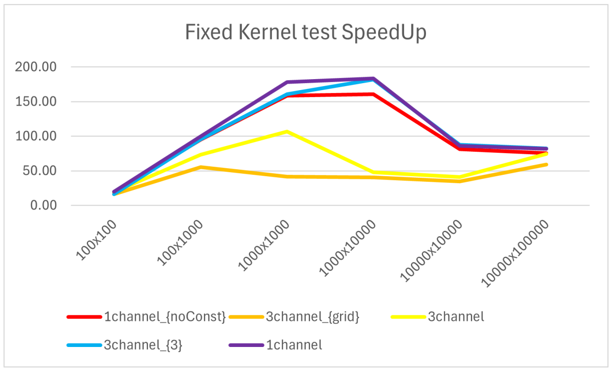


Figure 4: Fixed Image speedup

6.2. Fixed Kernel Size

In this experiment the size of the kernel is fixed to a resolution of 11×11 (121 values). Then the image size is increased as we can see in Table 3. All the CUDA implementations times are the result of an average over $1000, 1000, 1000, 1000, 1000, 100$ executions respectively. The average time for Seq is given from an average of $1000, 100, 50, 10, 5, 2$ executions respectively. All the results are shown in Table 3.

It can be seen from Figure 4 that the $3channel$ method in this test performs much less well than the previous one. This is largely due to the image linearization procedure. In fact, in this case this procedure being sequential suffers more from the magnitude of the increase in image size.

In contrast, the methods that do not perform linearization have a similar trend, initially the speedup increases and then see a deterioration in the transition from 1000×10000 image to 10000×10000 image. This is largely due to the delay in passing data from CPU to GPU. This is also confirmed by the fact that the times for the last processed image are similar among the various methods under investigation, thus highlighting a delay introduced mainly by the data handoff and thus common to the different techniques.

Despite the similar results the speedup of $1channel$ always turns out to be better than its counterpart $1channel_{noConst}$ confirming once again when accurate memory utilization is a crucial factor in achieving high speedups.

7. Conclusion

In conclusion, it can be confirmed that CUDA turns out to be an excellent tool to use for kernel image processing. As seen from the experimentation depending on the size of the images and the kernel used various techniques can be chosen that allow a high performance speedup compared to the sequential counterpart. However, the implementation of algorithms using CUDA is not trivial; in fact, the size of the data involved must be evaluated to achieve optimal performance. In addition, memory usage must be taken into account; the use of $\_\_constant\_\_$ variables was indeed crucial.

In addition, the image linearization steps have a decisive impact on the performance of the various methods that make use of them. For this reason in case of actual use of these methods it would be necessary to take into consideration the need for CPU-side parallelization using, for example, OpenMP.

7.1. Additional Materials

The code and the experimentation results are available in GitHub (see [1]).

References

[1] Tommaso Botarelli. *Parallel Computing Project*. URL: https://github.com/TommasoBotarelli/ParallelComputing_UNIFI_Project/tree/master/kernel_image_processing.

[2] NVIDIA. *Documentation*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[3] NVIDIA. *NVIDIA RTX A2000 12 GB datasheet*. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/rtx-a2000/nvidia-rtx-a2000-datasheet-1987439-r5.pdf.

[4] NVIDIA. *Variable memory space specifiers*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#variable-memory-space-specifiers.

[5] Wikipedia. *Kernel Image Processing*. URL: https://en.wikipedia.org/wiki/Kernel_(image_processing).