Kernel - Image Processing

Filippo Taiti
filippo.taiti@edu.unifi.it

February 18, 2026

Abstract

In image processing, a convolution kernel (or filter) is a small array of numbers that is slid over an image in order to modify or analyze it. Of course, running such an algorithm on computing units designed for massive computation leads to a significant speed-up.

1 Introduction

An image can be represented as a large matrix of pixels. A kernel, on the other hand, is a much smaller matrix that is applied to each pixel in the image using a mathematical operation known as **convolution**. Depending on the values of the kernel, different effects can be achieved, such as blurring, edge detection, and sharpening.
There are several computational challenges, including the high number of operations to be performed, which results in long execution times if dedicated hardware tools are not used. Another problem is at the hardware level: since each output pixel requires reading many neighboring pixels, these pixels may not always be contiguous in memory, which can lead to cache misses.
The first problem can be solved by using hardware tools designed for massive computation, such as GPUs. In this case, an implementation in the CUDA language will be provided. The second problem, on the other hand, can be mitigated by representing the data in special structures to maximize contiguity in memory.
This project will examine Gaussian kernels (of various sizes), which are filters that transform the original image into a blurred version.

2 Algorithm Description

The algorithm works as follows: For each pixel in the original image:

1. The kernel overlaps an area of the image.
2. The kernel values are multiplied element by element with the underlying pixels.
3. The results are added together.
4. The resulting value becomes the new value of the central pixel.

3 Implementation

The algorithm was implemented in two versions: a sequential version written in "classic" C++ and several parallel versions using CUDA directives.
As for the parallel version, several comparisons were made:

- Given an image, apply the algorithm directly to all 3 channels and to one channel at a time using CUDA streams. In this scenario, the time taken

to transfer data to and from memory was also included, as the aim was to determine the impact of CUDA streams on the entire image processing. In both cases, the version using tiling was compared with the more "naive" version that does not use it.

- In the second type of test, the two image storage approaches were compared: interval and planar formats. In this scenario, the impact of tiling was also evaluated.
- In the last scenario, using images in planar format and tiling, the tile width (i.e., the block size) was varied between the following values: 8, 16, 32.

The images are represented in planar format to be more memory-friendly.

4 Experimental Setup

- CPU: AMD Ryzen 7 9700x: 8 core (16 thread), 640 KB L1 cache, 8 MB L2 cache, 32 MB L3 cache.
- GPU: NVIDIA RTX 5070 12 GB DDR7 VRAM 4.608 KB L1 cache, 32.768 KB L2 cache.
- RAM: 48 GB DDR5
- Compiler: `nvcc` for CUDA and `clang++` for C++, compiled with optimization flags`-O3 -march=native -mtune=native -funroll-loops`.
- OS: Ubuntu 24.04

As a dataset, three images with increasing resolution were used: 640x800, 2560x1440, and 3840x2160.
Three Gaussian kernels, measuring 3x3, 7x7, and 11x11, all generated manually, were tested on these images. The accuracy of the results was verified by printing the processed images using the *stb_image* library to see if the desired effect was achieved.
In each test case, 100 iterations + 2 warm-up iterations were performed.

4.1 Test 1: Classical approach vs CUDA streams

This first test case serves to analyze the benefits of CUDA streams. It is a way to answer the question: "Is it really worth it?"

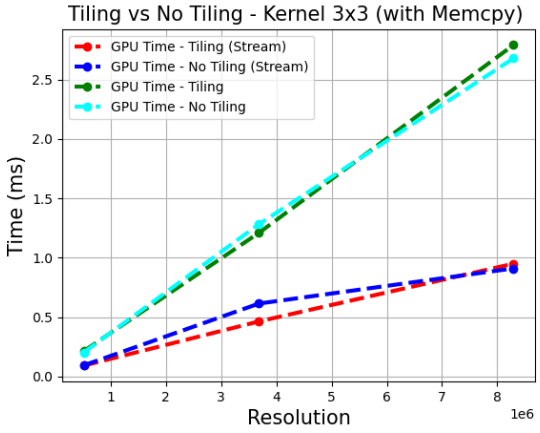

Figure 1: GPU performance - Kernel 3x3

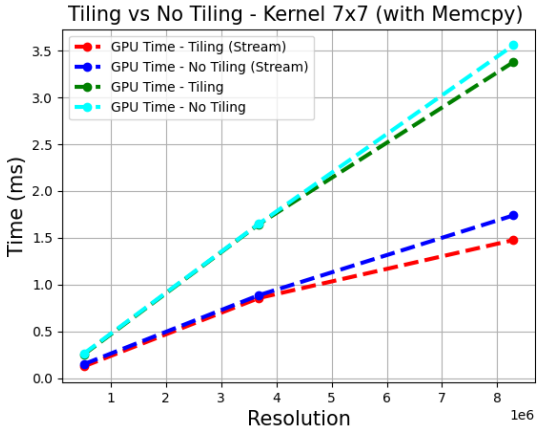

Figure 2: GPU performance - Kernel 7x7

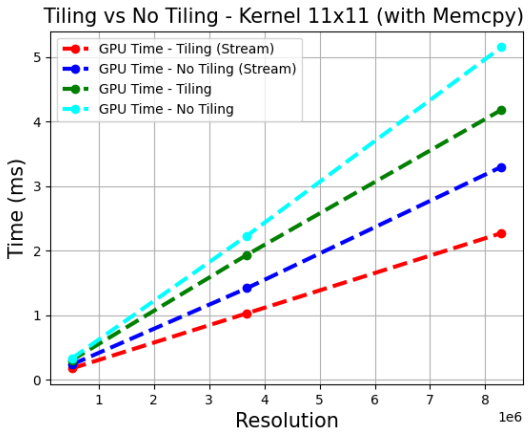

Figure 3: GPU performance - Kernel 11x11

The results show that using CUDA streams significantly improves performance.
Furthermore, note that with the 3x3 kernel, the problem is memory-bound, so tiling generally tends to cause minimal benefits, if not worsening performance. Moving to larger kernels (7x7 and 11x11), the problem becomes compute-bound, and therefore technical optimizations at the memory level lead to better performance.
In this scenario, it is interesting to observe the standard deviation of the two cases. Only the 4K image is analyzed to avoid redundancy, since the other two cases lead to the same conclusions.

	3x3	7x7	11x11
Tiling-Stream	0.2157	0.2672	0.2887
No Tiling-Stream	0.1462	0.0450	0.3937
Tiling	0.0037	0.0069	0.0120
No Tiling	0.0012	0.0079	0.3061

Table 1: Standard deviations of 4K image in Test 1

In the case where CUDA streams are used, the standard deviation is higher since the hardware has to handle both data transfer and the execution of the CUDA kernel "simultaneously", while in the "standard" case, the two things happen separately.

4.2 Test 2: Interleaved format vs Planar format

Kernel 3x3

Figure 4: CPU performance for 3x3 kernel

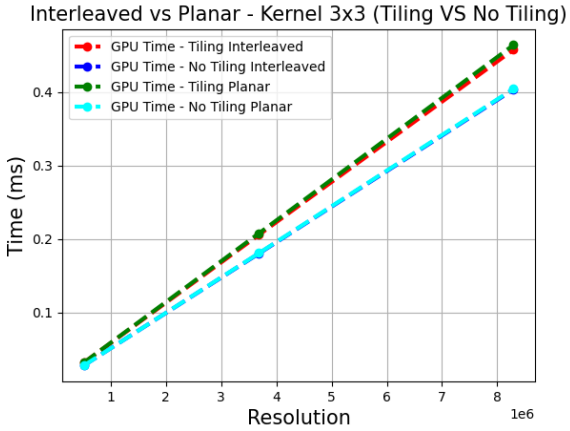

Figure 5: GPU performance for 3x3 kernel

Kernel 7x7

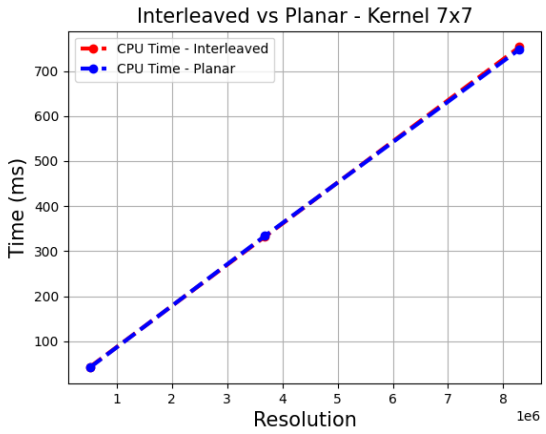

Figure 6: CPU performance for 7x7 kernel

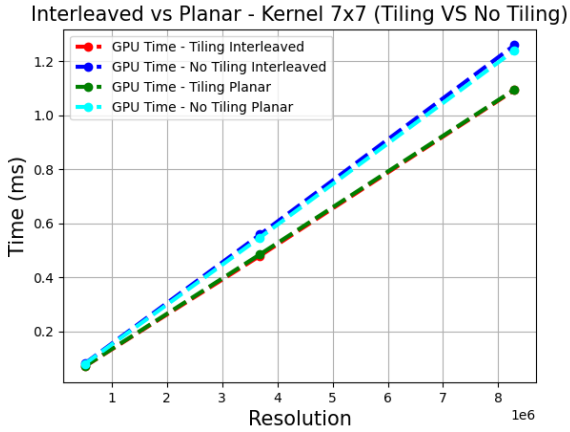

Figure 7: GPU performance for 7x7 kernel

Kernel 11x11

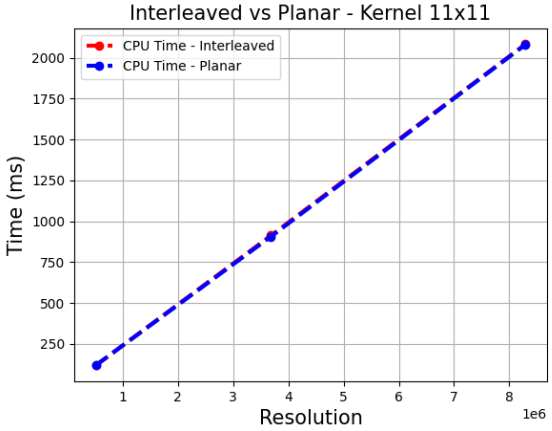

Figure 8: CPU performance for 11x11 kernel

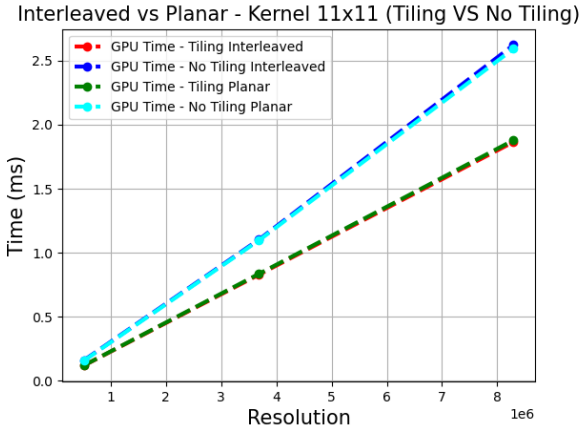

Figure 9: GPU performance for 11x11 kernel

The results show that using shared memory for small kernels (3x3 in this case) leads to worse results than using global memory directly. However, when the kernel size starts to increase (7x7 and 11x11), using shared memory leads to a significant improvement in performance. It should also be noted that the planar format is better than the interleaved format, as it is more memory-friendly.
Furthermore, we see that no significant differences are observed on the CPU. This behavior is mainly

due to the ability of modern CPUs to hide memory latency, along with compiler-level optimizations.

4.3 Test 3: Various block size

In this last scenario, the execution times will be shown as the block size varies. Since the tiling variant was chosen for the tests, the block size coincides with the tile width. The images are represented in planar format.

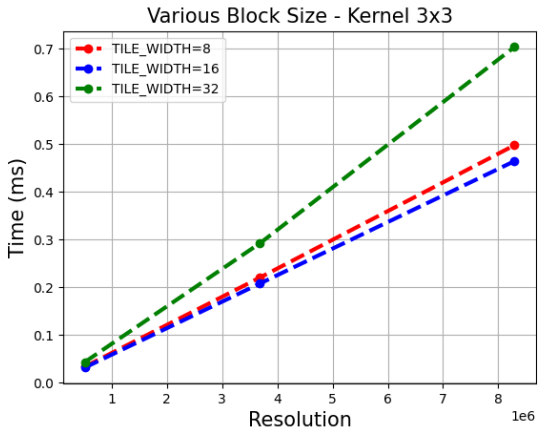

Figure 10: GPU performance - Kernel 3x3

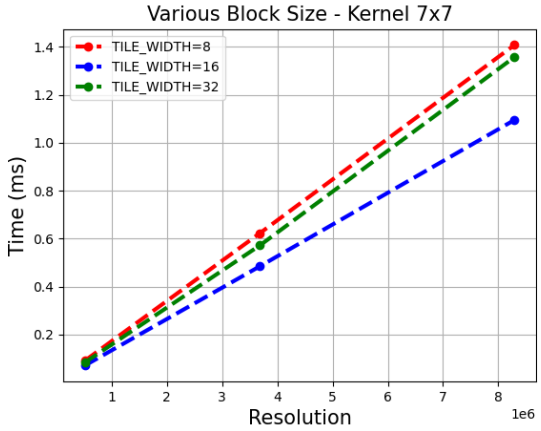

Figure 11: GPU performance - Kernel 7x7

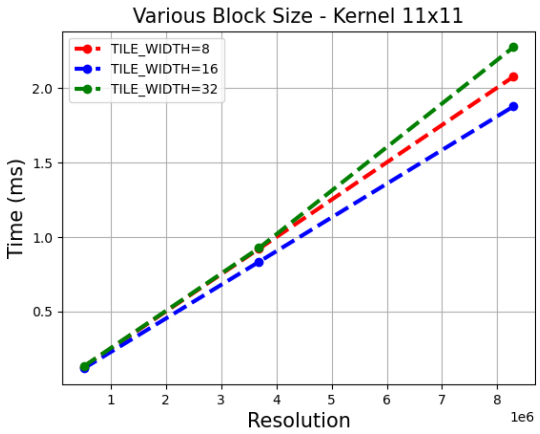

Figure 12: GPU performance - Kernel 11x11

The results show that using a moderate size leads to the best results. In fact, if blocks that are too small are used, the GPU is less able to hide latency, and in general the GPU performs less "computational work" than it could, while if blocks that are too large are used, there is less shared memory for each block and therefore performance worsens.

5 Limitations & Future work suggestions

As discussed above, this algorithm is computationally very demanding as it requires a very high amount of computation. In particular, if the source image has a very high resolution and/or the convolutional kernel is very large, execution on the CPU can become unsustainable.
A possible solution to the problem, as well as a possible subject for future implementations, is to exploit the separability property of Gaussian kernels: an NxN Gaussian kernel can be rewritten as the product of two vectors Nx1 and 1xN. Therefore, one can consider applying these sub-kernels in sequence and achieve a huge reduction in the number of operations. Despite this, in massive computing scenarios such as this, it is still preferable to use a GPU.