Kernel - Image Processing

Filippo Taiti
filippo.taiti@edu.unifi.it

February 6, 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 in C++ and a version written in CUDA language. In both cases, the algorithm was expressed as a generic convolutional kernel, and depending on the type of kernel taken as input by the algorithm, a different result will be obtained. In terms of implementation choices, the C++ version is written as a kernel that is applied directly to all image channels, while the basic CUDA version is applied to

only one channel at a time, but through the use of CUDA streams, the application is performed on all image channels. In particular, a mechanism known as **tiling** is essential in this context, i.e., avoiding continuous reading from global memory, which is slower than other smaller memories, such as shared memory, by loading image elements into shared memory as they are needed and then reading from the latter. As will be shown later, this mechanism significantly reduces execution times. The CUDA version was therefore implemented both using this mechanism and without it.
In accordance with best practices, convolutional filters have been allocated in constant memory.
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 graphical analysis of the times will show three graphs, one for each kernel. Each of these graphs shows the CPU (sequential) and GPU times with and without tiling, measured in milliseconds. The x-axis represents the resolutions, while the y-axis represents the times. In particular, to avoid representation problems, times have been represented on a logarithmic scale.
The tile width was set to 16, while the blocks are 32x32 in size.
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.

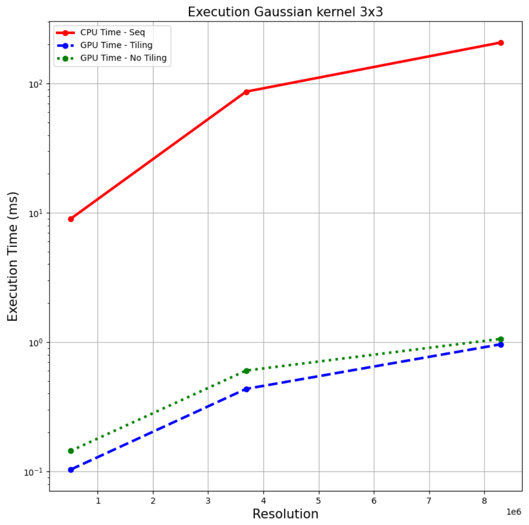

Figure 1: Gaussian kernel 3x3 performance

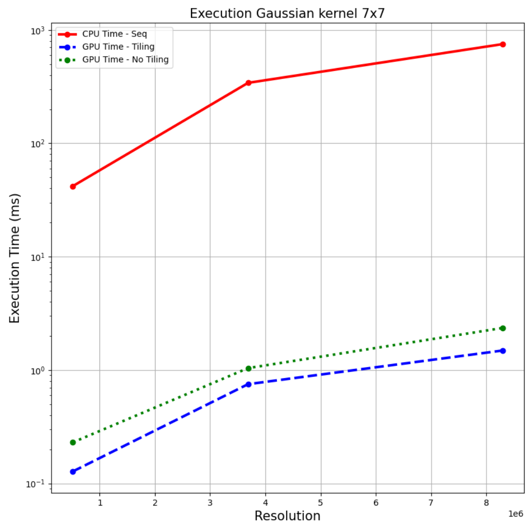

Figure 2: Gaussian kernel 7x7 performance

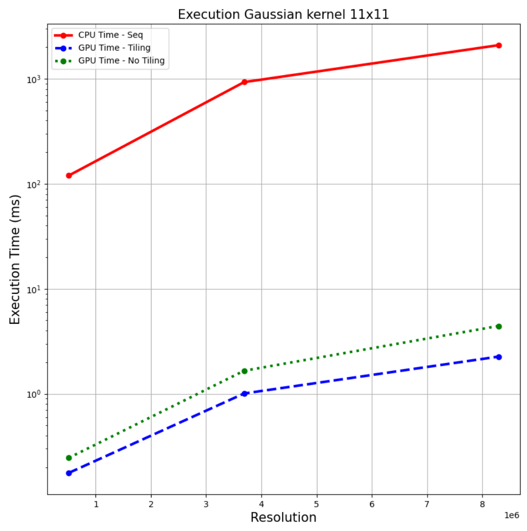

Figure 3: Gaussian kernel 11x11 performance

The results are in line with expectations: as can be seen from the graphs, GPU usage significantly reduces processing times compared to sequential execution on the CPU.
Another thing to note is that the use of tiling leads to better performance.
Below is an example of application.

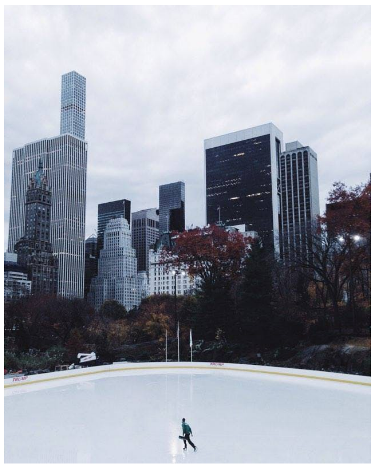

Figure 4: Original image

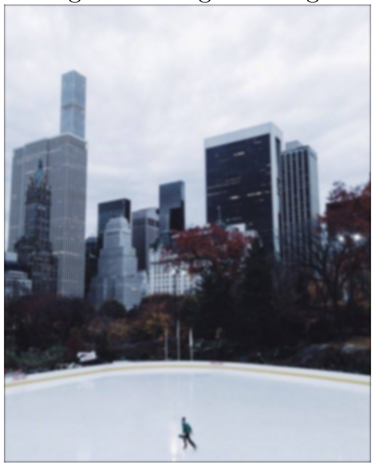

Figure 5: Blurred image with 11x11 Gaussian Kernel

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.