A Comparison of Sequential and Parallel Implementations in Kernel Image Processing with CUDA

Marco Billi
Università degli studi di Firenze
marco.billi@edu.unifi.it

Abstract

This work presents a performance evaluation of sequential and parallel implementations of kernel-based image processing. The parallel version is implemented using CUDA to leverage the SIMT architecture of GPUs. Benchmarking was performed on a RGB images of varying dimensions dataset using different kernel size to assess speedup. The results demonstrate that the CUDA-based implementation achieves a significant reduction in execution time compared to the sequential baseline, particularly with larger kernels. These findings underscore the effectiveness of parallelization in enhancing the computational efficiency of image processing operations.

1. Introduction

Kernel-based image processing is a fundamental framework widely employed for analyzing and manipulating the structure of RGB images. Core operations such as blurring and sharpening are essential in tasks like noise reduction and edge detection. These operations are primarily implemented through convolution and its optimized variants, such as separable convolution. The latter is only applicable to kernels that can be expressed as the outer product of two one-dimensional filters.

In many academic and experimental contexts, RGB image datasets are publicly available and easily accessible for research purposes. To evaluate the robustness of the proposed implementations, a dataset containing images of varying resolutions is used. This choice allows for testing the generalization of the approach beyond fixed-size inputs, reflecting more realistic application scenarios.

This study aims to evaluate and compare the performance of sequential and parallel implementations of both standard and separable convolution, with a focus on the impact of different kernel sizes on computational efficiency. The sequential implementation serves as a baseline, while the parallel version is developed using CUDA to leverage GPU-based SIMT architecture. A series of benchmarks is conducted to analyze the scalability and efficiency of each approach, highlighting the benefits of parallelization in kernel-based image processing.

2. Implementations

The implementation of kernel operations was developed in C++, following a modular architecture that ensures a clear separation of concerns among image handling, kernel configuration, and algorithmic processing. The codebase is organized into section components in main.cu file [1]:

- **Image and Kernel section:** Handles creation, shape rendering, and saving of RGB images and kernels.
- **Convolution section:** Contains both sequential and CUDA-parallelized implementations of convolution and separable convolution.
- **Benchmark section:** Executes timed tests on images, collecting performance data.

2.1. Image and Kernel Representation

RGB images are represented using a custom `struct` that encapsulates pixel data as a one-dimensional array of `uint8_t`, along with metadata such as width and height. All images are loaded from a dataset containing samples with varying resolutions. The convolution kernel is implemented as a conventional two-dimensional mask, represented using `std::vector<std::vector<float>>`. To ensure effective image transformations through convolution, kernels are constrained to have odd dimensions and must maintain internal coherence among their values (in particular I'm going to use gaussian blur for my experiments). Additionally, the kernel implementation includes support for normalization, as well as decomposition into separable

[1] All source code is available at https://github.com/billimarco/Kernel_Image_Processing

one-dimensional horizontal and vertical components using Singular Value Decomposition (SVD).

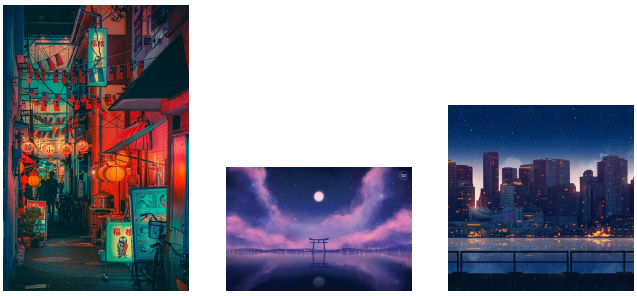

Figure 1. Sample of dataset anime images

2.2. Sequential Implementations

The sequential implementation of the **RGB convolution** is realized through a naive, yet straightforward, nested loop approach. The function `convolveRGB` applies a given square kernel to an input RGB image, represented by the `STBImage` structure. The kernel is assumed to have odd dimensions, and its center is computed as `size / 2`, allowing symmetric application over the image grid. The algorithm iterates over each pixel of the input image, and for each color channel, it performs a 2D convolution by summing the product of neighboring pixels and the corresponding kernel weights. To handle border conditions, zero-padding is applied: pixels that fall outside the image boundaries are treated as having a value of zero. Each output pixel is computed independently by accumulating the weighted sum of neighboring values, followed by clamping the result to the valid 8-bit range [0, 255]. The result is stored in a newly initialized image of the same dimensions as the input. This implementation provides a clear baseline for correctness and facilitates comparison with optimized or parallelized versions.

In addition to the standard 2D convolution, a **separable RGB convolution** version is implemented to exploit the mathematical property of certain kernels that allows them to be decomposed into the outer product of two 1D vectors (like gaussian blur). This method reduces computational complexity while preserving the effect of the full 2D convolution. The function `separableConvolutionRGB` takes an input RGB image and a kernel, and first attempts to decompose the kernel into two 1D vectors (one horizontal and one vertical) using Singular Value Decomposition (SVD). If the decomposition fails (i.e., the kernel is not separable like sharpening), the function throws an exception. The convolution is executed in two sequential passes:

- **First Pass – Horizontal Convolution:** Each row of the input image is convolved with the horizontal 1D kernel. The result is stored in a temporary image buffer. Zero-padding is applied at the borders, treating out-of-bounds pixel values as zero.
- **Second Pass – Vertical Convolution:** The temporary image is then convolved column-wise with the vertical 1D kernel, again applying zero-padding at the image boundaries. The result is stored in the final output image.

For both passes, the convolution is applied independently to each color channel (R, G, B). As with the standard version, values are clamped to the `[0, 255]` range to ensure valid 8-bit output.

This separable implementation significantly improves performance by reducing the number of required operations from $\mathcal{O}(k^2)$ per pixel to $\mathcal{O}(2k)$, where k is the kernel size. The result is functionally equivalent to the full 2D convolution when the kernel is separable, and serves as an important optimization baseline for more advanced or parallelized implementations.

2.3. Parallel Implementations with CUDA

To exploit data-level parallelism and improve performance over sequential convolution, we implemented a **CUDA-based version of the RGB convolution** routine. This implementation processes RGB images on the GPU, with each pixel handled independently by a separate CUDA thread. The approach leverages the inherently parallel nature of the convolution operation, where each output pixel is computed as a weighted sum of neighboring input pixels using a given convolution kernel. The CUDA kernel function `convolveKernelRGB` performs a 2D convolution over the input image. Each thread computes the result for a single pixel and all color channels of that pixel. To correctly apply the kernel, each thread calculates its global image coordinates `(x, y)` and iterates over the kernel window centered at that pixel. Border handling is achieved through zero-padding. The host function `convolveRGB_CUDA` prepares the data for GPU execution by:

- Flattening the 2D kernel into a 1D array;
- Allocating device memory for the input image, output image, and kernel;
- Transferring data from host to device;
- Launching the CUDA kernel using a configurable thread block size;
- Synchronizing the device and copying results back to host memory.

The kernel grid is configured based on the image resolution and chosen thread block dimensions, ensuring full

coverage of the image domain. As in the sequential implementation, the convolution result is clamped to the valid 8-bit range [0,255] before being stored in the output image.

To optimize further, a **CUDA-based version of the separable RGB convolution** is implemented. The kernel is decomposed into two 1D vectors (horizontal and vertical) using Singular Value Decomposition (SVD). The rest is similar to `convolveRGB_CUDA`, but in two passes:

- **Horizontal pass in CUDA:** Each thread convolves the image rows with the 1D horizontal kernel, storing intermediate results in a temporary buffer on the device.
- **Vertical pass in CUDA:** The intermediate result is then convolved column-wise with the vertical 1D kernel by another kernel launch.

The host functions for both approaches manage GPU memory allocations, data transfers, kernel launches, synchronization, and resource deallocation.

3. Methodology

To assess and compare the performance of the two convolution implementation, I decided to take a small dataset with 779 images with different resolutions (figure 1). Using images of different sizes allows us to evaluate how each implementation scales with resolution and how it handles computational load under heterogeneous conditions. To evaluate performance, the mean execution time and speedup were computed for the processing of a single image. [2]

My evaluation is structured around **kernel size**:

- **3×3** – small kernel
- **7×7** – medium kernel
- **11×11** – large kernel

In particular, **Gaussian blur** was chosen as the reference operation for evaluating execution times and speedup. This decision is based on the assumption that the type of image transformation does not significantly impact performance, but rather the computational structure of the convolution itself. Gaussian blur is especially suitable for this analysis because its kernel is separable, unlike other operations such as sharpening. This allows for a fair comparison between the standard and separable convolution implementations.

In my implementation, the CUDA grid size is computed dynamically based on the image dimensions and the chosen `dim3 blockDim` configuration. Specifically, the block size can be adjusted at runtime to adapt to different image resolutions or GPU architectures. This flexibility is essential for optimizing performance in CUDA programming for several reasons, like maximizing the number of active warps per streaming multiprocessor and improving memory coalescing. The following block configurations were tested to evaluate their impact on performance:

- **(4, 4)** – Very small block size; useful for verifying correctness but typically underutilizes GPU resources.
- **(8, 4)** – Slightly more threads per block; may offer better occupancy while still limited in performance.
- **(8, 8)** – Balanced square configuration; allows moderate parallelism with manageable resource usage.
- **(16, 8)** – Rectangular block; suitable for images with different aspect ratios, improves memory access patterns.
- **(16, 16)** – Common configuration; aligns well with warp size (32 threads) and enables good memory coalescing.
- **(32, 8)** – Fully occupies a warp per row; often used to optimize horizontal memory access.
- **(32, 16)** – Larger rectangular block; maximizes warp utilization and can benefit from coalesced memory accesses if the layout is favorable.
- **(32, 32)** – Very large block; maximizes parallelism per block but may reduce occupancy if shared resources are limited.

4. Experiments and Results

The experiments were conducted on a system equipped with an **NVIDIA GeForce GTX 1050 Ti GPU**. The 1050 Ti is a graphics card based on the Pascal architecture, featuring 768 CUDA cores, a base clock of 1290 MHz, and 4 GB of GDDR5 memory. It supports full CUDA capabilities and is suitable for evaluating GPU performance in small parallel workloads. The system was configured with the nvcc compiler, utilizing optimization level 03 to enable vectorization.

The analysis begins with the sequential execution times for both standard and separable convolution as the kernel size increases (Figure 2). The runtime for standard convolution exhibits a quadratic growth trend, while the separable version grows linearly. These results are consistent with the theoretical computational complexities of the two approaches.

[2] Anime dataset is available at https://huggingface.co/datasets/sulpha/anime-sceneries

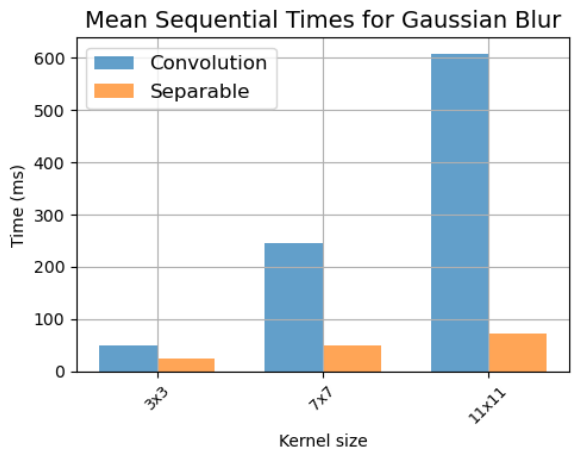

Figure 2. Comparison of sequential times as the kernel size increases

As shown in Figure 3, this trend does not exactly hold under parallel execution. The following observations can be made:

- **Larger kernels yield higher speedup:** The speedup obtained through parallelization increases with the kernel size. This effect is particularly evident in the case of standard (non-separable) convolution, where the computational load grows quadratically with kernel dimensions. In contrast, separable convolution, which has inherently lower computational complexity, shows a more moderate improvement, though still significant, when the kernel size increases.
- **Effect of block size on performance:** Larger CUDA block sizes generally lead to better speedup, as they increase the number of threads per block and improve resource utilization. However, block sizes exceeding the number of available CUDA cores (768 in the case of the GTX 1050 Ti) can introduce overhead due to scheduling and resource contention. This behavior is particularly visible in the performance for the 11×11 kernel, where excessive block sizes result in diminishing returns. Configurations such as $(16, 16)$ and $(32, 8)$ offer a good balance between parallelism and memory efficiency, making them practical default choices.
- **Performance characteristics of separable convolution:** While separable convolution offers reduced computational complexity, its performance gains under parallelization are less dramatic than standard convolution. For small kernels 3×3, the overhead of performing two passes (horizontal and vertical) may offset the theoretical benefits, and in some configurations may even result in equal or slightly worse performance than standard convolution. However, as kernel size increases, the advantages of separable convolution become evident, making it the preferred method when applicable.

5. Conclusion

In this work, I investigated the performance of RGB image convolution using both standard and separable kernels, comparing a sequential CPU implementation with GPU-accelerated versions developed in CUDA. The results show that while separable convolution provides a theoretical advantage due to its reduced computational complexity, this benefit becomes practically significant only for larger kernel sizes.

The CUDA implementations consistently outperformed the sequential version, especially as kernel size increased. I observed that the choice of block size has a direct impact on performance: configurations such as $(16, 16)$ and $(32, 8)$ offered a good balance between parallel efficiency and hardware utilization, particularly on devices with limited CUDA cores like the GTX 1050 Ti.

Separable convolution on the GPU exhibited more stable performance across different block sizes, but showed limited advantages for small kernels due to the two-pass structure. However, for larger kernels, it clearly outperformed the standard convolution in terms of execution time.

In conclusion, combining kernel separability with GPU parallelism allows for significant acceleration of convolution-based image processing, provided that implementation details, such as memory access patterns and kernel launch configurations, are properly optimized.

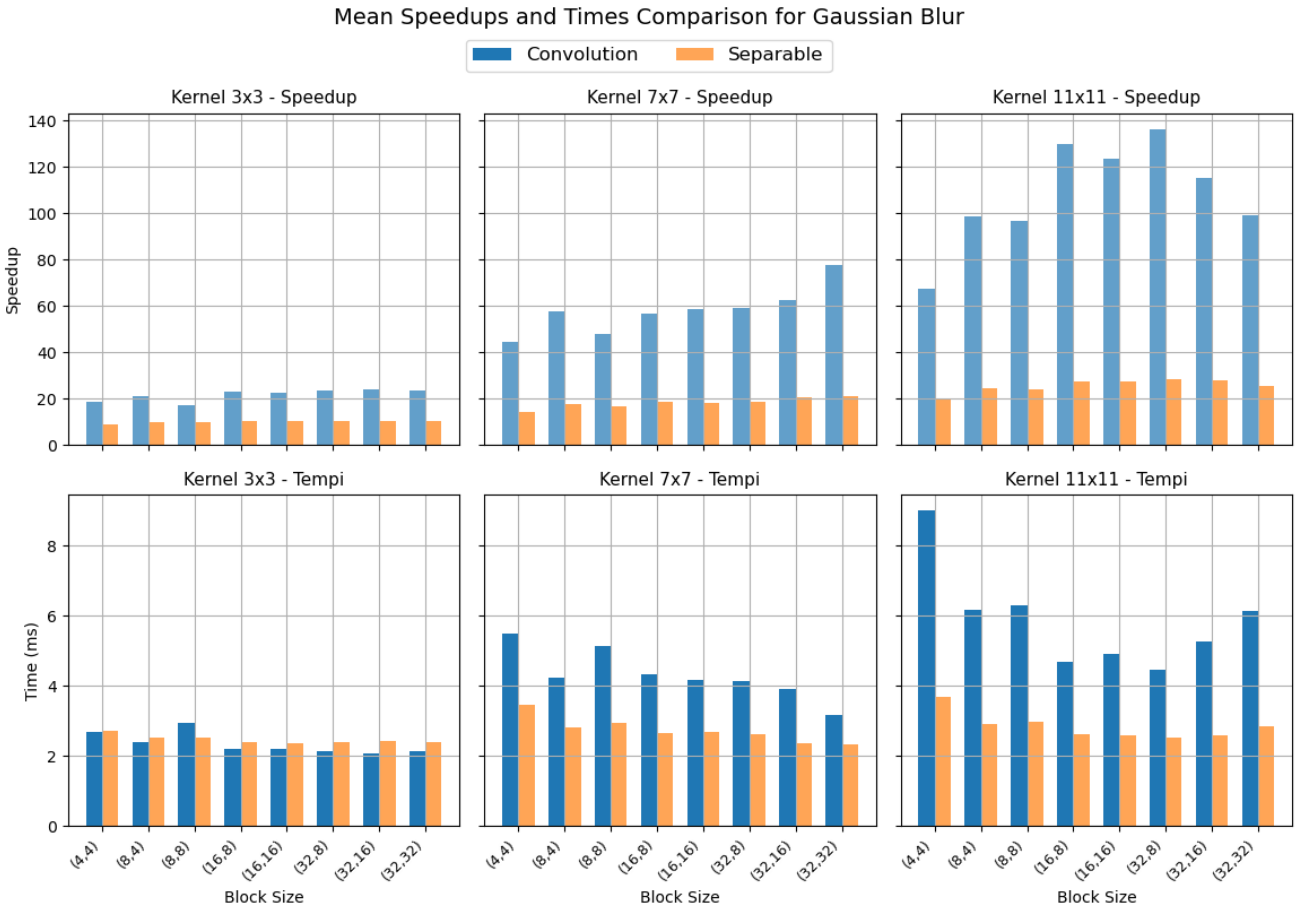

Figure 3. Comparison of speedup and times results across all CUDA configurations.

A. Visualization of a image with gaussian blur

(a) Basis (b) 3x3

(c) 7x7 (d) 11x11

Figure 4. Comparisons of different gaussian blur kernel convolution on an image