Image Convolution with Filters: Performance Optimization through Parallelization

Lapo Chiostrini
Department of Information Engineering
University of Florence
lapo.chiostrini1@edu.unifi.it

Lorenzo Cappetti
Department of Information Engineering
University of Florence
lorenzo.cappetti@edu.unifi.it

Abstract

This work analyzes and optimizes 2D convolution on RGB images by comparing three implementations: (i) a sequential CPU baseline, (ii) a CPU-parallel version based on processes with shared memory implemented in Python, and (iii) a GPU version using CUDA with shared-memory tiling and constant-memory kernels. Tests are conducted on a synthetic dataset of images up to 8000×8000 *pixels and Gaussian kernels of size* 3×3*,* 5×5*, and* 7×7*, following a benchmarking protocol with a warm-up phase and an average over 10 runs. On the experimental platform (Intel i9-13900H and NVIDIA RTX 4060 Laptop), CPU parallelization achieves speedups of up to about* $6\times$*, showing sublinear scalability and diminishing returns beyond 12–16 processes due to overhead and memory-bandwidth saturation. The CUDA implementation reaches speedups on the order of* 350–$440\times$ *relative to the baseline, with runtimes on the order of* $\sim$*1 s for* 8000×8000 *images including Host–Device transfers. The analysis confirms the predominantly memory-bound nature of the workload for small kernels, highlighting that data locality, reuse (tiling(ii) a CPU-parallel version based on processes with shared memory, and (), and memory management are as critical as parallelism.*

1. Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

2. Introduction

Image convolution is one of the fundamental operations in computer vision and digital signal processing. Applied through kernels (filters) of different sizes, it enables effects such as blur, sharpening, edge detection, and Gaussian smoothing, forming the basis of many advanced techniques in image analysis and deep learning.

For high-resolution images, convolution can become computationally demanding: an 8000×8000 RGB image with a 7×7 kernel requires about 19 billion floating-point operations. This computational complexity makes both algorithmic and architectural optimization techniques necessary.

The goal of this project is to conduct a quantitative performance analysis of different implementations of the convolution algorithm, exploring:

- **CPU parallelization**: Block partitioning and the use of shared memory to reduce overhead.
- **GPU acceleration**: Exploiting CUDA massive parallelism.
- **Scalability**: Behavior as image size, kernel size, and computational resources vary.

2.1. Contributions

This work presents:

1. Three complete and optimized implementations (sequential, CPU-parallel, GPU).
2. A systematic benchmark on synthetic datasets of different sizes (800×800 up to 8000×8000 pixels) and across varying kernel sizes.
3. A comparative analysis of speedups and the identification of bottlenecks.
4. Guidelines for choosing the most suitable implementation depending on the application context.

3. Convolution

3.1. Mathematical Definition

Let $I \in \mathbb{R}^{h_{in} \times w_{in}}$ be a two-dimensional image and $K \in \mathbb{R}^{k_h \times k_w}$ a kernel. The operation commonly referred to as 2D convolution (more precisely, a correlation) produces an output image O defined as:

$$O(i,j) = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} \tilde{I}(i+m,\, j+n) \cdot K(m,n) \qquad (1)$$

where $\tilde{I}$ denotes the image extended with *zero-padding* at the borders:

$$\tilde{I}(x,y) = \begin{cases} I(x,y) & \text{if } 0 \le x < h_{in} \ \wedge \ 0 \le y < w_{in}, \\ 0 & \text{otherwise.} \end{cases}$$

In the benchmarks, the *same* convention (zero padding) is adopted, hence:

$$0 \le i < h_{in}, \quad 0 \le j < w_{in},$$

and the output dimensions match the input ones:

$$h_{out} = h_{in}, \qquad w_{out} = w_{in}.$$

Note (valid). In the case of *valid convolution* (no padding), the output dimensions are:

$$h_{out} = h_{in} - k_h + 1, \qquad w_{out} = w_{in} - k_w + 1. \quad (2)$$

For RGB images, the operation is applied independently to each channel $c \in \{R, G, B\}$, producing three output maps that are subsequently recombined.

3.2. Computational Complexity

For an image of size $H \times W$ and a square kernel of size $K \times K$, each output pixel requires computing the sum of products between the kernel values and the corresponding image pixels. In particular, for each output pixel the following are needed:

- K^2 floating-point multiplications;
- $K^2 - 1$ floating-point additions.

The total number of operations per pixel is therefore approximately $2K^2$ FLOPs. With the *same* convention adopted in the benchmarks, the number of pixels in the output image is:

$$H \cdot W$$

The overall computational complexity of the 2D convolution operation is thus:

$$O(H \cdot W \cdot K^2) \qquad (3)$$

for a single channel, and:

$$O(3 \cdot H \cdot W \cdot K^2) \qquad (4)$$

in the case of RGB images, where the operation is applied independently to each channel.

Note (valid). In the case of *valid convolution*, i.e., without padding, the number of pixels in the output image is:

$$(H - K + 1)(W - K + 1) \approx H \cdot W$$

for values of H and W significantly larger than K.

3.3. Numerical Example

Considering an RGB image of size 8000×8000 pixels and a kernel of size 7×7, the output image, in the *valid* case, has dimensions 7994×7994 pixels. The total number of floating-point operations required to compute the convolution over all three channels is:

$$3 \times 7994^2 \times (49 + 48) \approx 1.86 \times 10^{10} \text{ FLOP}. \qquad (5)$$

This value highlights the high computational cost of direct convolution on high-resolution images.

3.4. Kernel Types

In this project, several standard kernels are used, each designed to achieve specific image-processing effects:

- **Blur (3×3)**: a uniform smoothing operator, where all kernel coefficients take the same value;
- **Sharpen**: a detail-enhancement kernel, characterized by a positive central coefficient and negative surrounding coefficients;
- **Edge Detection**: an edge-detection kernel based on the Laplacian operator;
- **Emboss**: a kernel that produces a 3D relief effect by emphasizing intensity variations;
- **Gaussian (3×3, 5×5, 7×7)**: a weighted smoothing kernel, whose coefficients follow a normalized binomial Gaussian distribution.

4. Data Generation

4.1. Synthetic Dataset

To ensure **reproducibility** and **control over the input sizes**, a synthetic dataset of random RGB images is generated:

- **Tested sizes**: 800×800, 1600×1600, 3200×3200, 6400×6400, 8000×8000 pixels;
- **Format**: **8-bit RGB** (`uint8`), **3 channels**;
- **Content**: **uniform random values** in $[0, 255]$ for each pixel;
- **Generation**: NumPy `random.randint()`.

4.2. Test Kernels

Three **Gaussian kernels** of increasing size are evaluated:

- **Gaussian 3×3**: **9 coefficients**;
- **Gaussian 5×5**: **25 coefficients**, **binomial distribution**;
- **Gaussian 7×7**: **49 coefficients**, **stronger smoothing**.

All kernels are **normalized** to preserve the **average image brightness**.

5. Sequential Implementation

The sequential version serves as the **baseline reference** for evaluating the performance of the subsequent parallel implementations. The algorithm performs the entire convolution operation on the CPU in **single-thread** mode, without exploiting any form of parallelism.

5.1. Processing Pipeline

The procedure followed by the sequential implementation can be summarized in the following steps:

- **Image loading**: the input RGB image is read from disk and converted into a three-dimensional array ($H \times W \times 3$);
- **Numeric conversion**: pixel values are converted to **32-bit floating-point** format to make arithmetic operations efficient.
- **Kernel preparation**:
  - **coefficient normalization** (if required), to preserve the average brightness;
  - **2D kernel flipping**, as prescribed by the formal definition of convolution;
  - conversion to **floating-point** format.
- **Output allocation**: the output image array is preallocated with dimensions $(H-K+1) \times (W-K+1) \times 3$.
- **Convolution Computation**: we compute the convolutions for each channels
- **Save Results**: the nre image is saved on disk

5.2. Convolution Computation

The convolution is applied independently to each RGB channel. For each channel, the algorithm:

- sequentially scans all valid positions of the output image;
- for each position, extracts the local image window corresponding to the kernel size;
- computes the **sum of element-wise products** between the window and the kernel;
- assigns the resulting value to the output pixel.

This scheme directly implements the mathematical definition of two-dimensional convolution through **nested loops**, representing the most intuitive and general approach.

5.3. Post-processing and Measurement

After the convolution:

- the output values are clamped to the range $[0, 255]$;
- data are converted back to `uint8`;
- the three channels are recombined into the final RGB image.

A CPU profiler is used to:

- identify the most expensive functions;
- pinpoint the main computational bottlenecks;
- provide a quantitative reference for comparison with the parallel versions.

6. CPU Parallel Implementation

The second implementation introduces **CPU parallelism**, leveraging `multiprocessing` to distribute the workload across multiple cores. The main goal is to reduce convolution runtime while avoiding the typical overhead of transferring large arrays between processes, through the explicit use of **shared memory**.

6.1. Parallelization Strategy

Before describing the adopted strategy in detail, it is important to highlight that the relationship between the number of physical CPU cores and the degree of parallelism achievable via Python `multiprocessing` is not perfectly linear.

In theory, assigning one process per physical core would yield a speedup that is almost proportional to the number of cores. In practice, however, several factors make this relationship **weak**: the overhead of process creation and management, contention for shared-memory resources, and synchronization latency reduce efficiency as the number of processes increases.

As a consequence, beyond a certain threshold, further increasing the number of processes does not provide proportional benefits in terms of runtime, and it may even degrade performance due to higher scheduling and communication overhead.

Parallelism is applied simultaneously along two dimensions:

- **Channel parallelism**: the three RGB channels are processed independently;
- **Spatial parallelism**: the output image is partitioned into rectangular blocks, each assigned to a distinct process.

Each work unit therefore corresponds to the convolution of a single spatial block of a single channel. This approach enables effective utilization of all available cores while balancing the computational load.

6.2. Block Partitioning

The output image of size $(H - K + 1) \times (W - K + 1)$ is divided into blocks of size $B \times B$.

- The block size can be set manually;
- Alternatively, it can be computed automatically as a function of the number of available cores, in order to obtain a sufficient number of blocks to saturate computational resources.

For each block, a region of the input image extended by $(K - 1)$ pixels along the borders is considered, so that the convolution computation is independent across adjacent blocks.

6.3. Shared Memory Usage

To avoid the cost of serializing (pickling) large NumPy arrays across processes, both the input image and the output image are allocated in **shared memory**.

- The **input** image is copied only once into a shared buffer in `float32` format;
- The **output** image is preallocated in shared memory as a `float32` array;
- Each process directly accesses the shared buffers without duplicating the data.

This mechanism **drastically reduces communication overhead** between processes and enables true process-based parallelism, bypassing the limitations imposed by Python's **Global Interpreter Lock (GIL)**.

6.4. Convolution Computation in Worker Processes

Each process performs convolution on:

- a specific RGB channel;
- an assigned spatial block.

For each block:

- the required input region is extracted, including the overlap required by the kernel;
- direct convolution is applied using **nested loops**, computing the **sum of products** between the local window and the **flipped kernel**;
- the result is written directly into the corresponding portion of the output image in **shared memory**.

At the end of the computation, all processes collectively fill the final output image.

6.5. Post-processing and Synchronization

After the parallel activities complete:

- the content of the shared output image is copied into a local array;
- values are clamped to the range $[0, 255]$ and converted to `uint8`;
- shared memory is explicitly released.

7. GPU Parallel Implementation

The third implementation is the **core of this work** and performs the convolution operation entirely on the GPU using CUDA. The computation exploits the **massive parallelism** of the SIMT architecture, assigning each thread the computation of a single output pixel.

Particular attention is devoted to **memory-access optimization**, through **spatial tiling**, **shared memory** usage, and **constant memory** for the kernel coefficients. The goal is to maximize computational throughput and reduce repeated accesses to global memory.

7.1. Overall Implementation Structure

RGB convolution on the GPU follows these steps:

- loading the RGB image on the host side and separating the R, G, and B channels;
- allocating **pinned memory** on the host for input and output;
- copying the convolution kernel coefficients into **constant memory** on the device;
- allocating **global memory** on the device for input and output data;
- copying data from the host pinned buffers to the device global memory;
- launching the CUDA kernel for each color channel;
- copying results from device global memory back to the host pinned buffers;
- final copy from pinned buffers to standard host memory.

7.2. CUDA Kernel Organization

GPU convolution is organized through a **spatial tiling** strategy. The output image is partitioned into square tiles of size $X \times X$ pixels. Each CUDA block is responsible for computing one output tile and uses additional threads to load into shared memory the pixels required to handle kernel borders (the *halo*).

As a result, the effective CUDA block size is $(X + K - 1) \times (X + K - 1)$, where K denotes the convolution kernel size. This configuration allows each block to operate independently.

7.3. Shared Memory Usage

At the beginning of each block execution, threads cooperate to load from global memory into shared memory an extended tile of the input image, including the *halo* region. Pixels falling outside the image boundaries are initialized to zero.

After an explicit thread synchronization, each thread computes a single output pixel by applying the convolution kernel to the data resident in shared memory. This strategy enables multiple reuse of the same data within the block and significantly reduces the number of global memory accesses.

7.4. Constant Memory and Kernel Specialization

The convolution kernel coefficients are stored in the GPU **constant memory**, enabling efficient, cache-friendly accesses by all threads within a warp.

The CUDA kernel is also specialized at compile time for $K = 3, 5, 7$ using templates. This choice eliminates runtime checks on the kernel size and enables **loop unrolling** of the inner loops, improving computational efficiency.

7.5. RGB Channel Handling

The three color channels R, G, and B are processed separately. For each channel, a distinct CUDA kernel launch is performed, reusing the same grid, block, and shared-memory configuration. This approach simplifies kernel structure and effectively exploits the GPU's internal parallelism.

7.6. Host–Device Transfers

To reduce communication overhead between CPU and GPU, the input and output images on the host side are allocated in **pinned memory**. Using pinned memory enables more efficient transfers over the PCIe bus than standard host memory.

7.7. Summary of Optimizations

The main optimizations implemented are:

- **spatial tiling** with **shared memory** to maximize data reuse;
- use of **constant memory** for the convolution kernel coefficients;
- **compile-time kernel specialization** for fixed sizes;
- use of **pinned memory** to accelerate Host–Device transfers.

Overall, these design choices allow the CUDA version to effectively leverage the GPU's **SIMT architecture**, achieving a significant performance improvement over serial CPU implementations, especially for large images.

8. Experimental Conditions

8.1. Experimental Setup

Hardware All benchmarks were executed on a single machine with the following specifications:

- **CPU**: 13th Gen Intel(R) Core(TM) i9-13900H (14 cores / 20 threads)
- **RAM**: 32 GB DDR5-6000
- **GPU**: NVIDIA GeForce RTX 4060 (Laptop GPU), 8 GB VRAM

Software

- **OS**: Linux
- **Python**: 3.11.5
- **NumPy**: 1.24.3 (Intel MKL)
- **Joblib**: 1.3.2
- **CUDA Toolkit**: 13.0

Note on the operational definition of convolution In the benchmarks, the output is produced on an $H \times W$ grid and pixels outside the image boundaries are treated as zero (implicit zero-padding). This choice is consistent with the CUDA implementation based on shared-memory tiling and avoids shape differences across implementations during runtime comparison.

8.2. Explored Parameters

Experiments were conducted by systematically varying the parameters that determine: (i) the problem size, (ii) the computational intensity, and (iii) the degree of CPU and GPU parallelism. Table 1 summarizes the explored parameter space.

Table 1. Experimental parameters explored in the benchmarks.

Parameter	Tested values
Kernel size ($K \times K$)	3×3, 5×5, 7×7
Image size (pixels)	800^2, 1600^2, 3200^2, 6400^2, 8000^2
Number of CPU processes	4, 8, 12, 16, 20
CUDA tile size	8×8, 16×16, 24×24, 32×32

8.3. Benchmark Protocol

For each parameter combination:

1. the synthetic image is generated once and saved;
2. a **warm-up run** is executed to initialize caches and drivers;
3. **10 runs** are performed and the **average time** is reported;
4. timings are measured **internally by the implementations**.

To reduce variance, runs were executed under the most stable conditions possible (no significant concurrent workloads).

8.4. Experimental Goals

The experiments are designed to answer the following questions:

- **CPU scalability:** how does speedup change as the number of processes increases, and where does saturation occur?
- **GPU acceleration:** what is the gain of the CUDA version relative to the sequential baseline?
- **Impact of CUDA tile size:** which tile sizes maximize performance, and how sensitive is the kernel to this choice?
- **Fair comparison:** when normalizing by the number of operations (time per MAC), how do conclusions change relative to absolute runtimes?
- **Overall comparison:** which implementation is preferable as the problem scale varies?

9. Results and Analysis

This section presents the benchmark results obtained for the three developed implementations (**sequential**, **CPU-parallel**, and **CUDA on GPU**). For each configuration, we report **absolute runtimes** and **speedups** relative to the sequential baseline; moreover, where appropriate, we discuss **CPU parallel efficiency** and the impact of GPU launch parameters (**tile size**).

9.1. Sequential Implementation Performance

The sequential version provides the **reference baseline** for all subsequent comparisons. Table 2 reports the absolute execution times for different combinations of image size and kernel size.

Runtime increases regularly with image size, in accordance with the expected complexity $O(H \cdot W \cdot K^2)$. Within the considered range ($K = 3, 5, 7$), the runtime increase from 3×3 to 7×7 is relatively modest compared to the growth in the number of pixels; this suggests that, for small kernels, the overall cost is significantly influenced by **data movement** and **memory accesses**, in addition to the number of arithmetic operations.

Table 2. Execution times of the sequential version (ConvSeq-Dumb) as a function of image size and kernel size.

Image Size	K=3 (s)	K=5 (s)	K=7 (s)
800×800	3.82	3.95	3.83
1600×1600	15.25	15.25	15.86
3200×3200	59.69	60.81	59.99
6400×6400	234.03	238.79	244.43
8000×8000	369.30	371.00	374.04

9.2. CPU Parallelization Scalability

The CPU-parallel implementation was tested by varying the number of worker processes from 4 to 20. Tables 3, 5, and 7 report the speedups achieved relative to the sequential version for different image and kernel sizes. Tables 4, 6, and 8 report the parallel efficiency $E = S/P$.

Table 3. Speedup of the CPU-parallel version for a 3×3 kernel.

Image Size	4 jobs	8 jobs	12 jobs	16 jobs	20 jobs
800×800	2.86×	4.52×	5.70×	5.76×	6.09×
1600×1600	2.81×	4.27×	4.75×	4.78×	4.99×
3200×3200	2.60×	3.78×	4.67×	4.74×	4.84×
6400×6400	2.55×	3.72×	4.52×	4.51×	4.78×
8000×8000	2.55×	3.69×	4.50×	4.62×	4.83×

Table 4. Parallel efficiency of the CPU version for a 3×3 kernel ($E = S/P$).

Image Size	4 jobs	8 jobs	12 jobs	16 jobs	20 jobs
800×800	71.6%	56.5%	47.5%	36.0%	30.4%
1600×1600	70.2%	53.4%	39.6%	29.9%	24.9%
3200×3200	65.0%	47.2%	38.9%	29.6%	24.2%
6400×6400	63.7%	46.5%	37.7%	28.2%	23.9%
8000×8000	63.8%	46.1%	37.5%	28.9%	24.2%

Table 5. Speedup of the CPU-parallel version for a 5×5 kernel.

Image Size	4 jobs	8 jobs	12 jobs	16 jobs	20 jobs
800×800	2.85×	4.46×	5.75×	5.76×	6.05×
1600×1600	2.78×	4.28×	4.71×	4.66×	4.89×
3200×3200	2.76×	3.82×	4.56×	4.58×	4.89×
6400×6400	2.54×	3.78×	4.56×	4.69×	4.74×
8000×8000	2.54×	3.70×	4.45×	4.59×	4.81×

Table 6. Parallel efficiency of the CPU version for a 5×5 kernel ($E = S/P$).

Image Size	4 jobs	8 jobs	12 jobs	16 jobs	20 jobs
800×800	71.1%	55.7%	47.9%	36.0%	30.3%
1600×1600	69.5%	53.5%	39.3%	29.1%	24.5%
3200×3200	69.1%	47.8%	38.0%	28.6%	24.4%
6400×6400	63.6%	47.2%	38.0%	29.3%	23.7%
8000×8000	63.5%	46.3%	37.1%	28.7%	24.1%

Table 7. Speedup of the CPU-parallel version for a 7×7 kernel.

Image Size	4 jobs	8 jobs	12 jobs	16 jobs	20 jobs
800×800	2.59×	4.29×	5.48×	5.21×	5.69×
1600×1600	2.82×	4.45×	4.85×	4.77×	5.00×
3200×3200	2.67×	3.70×	4.57×	4.56×	4.80×
6400×6400	2.60×	3.75×	4.53×	4.51×	4.83×
8000×8000	2.54×	3.75×	4.42×	4.67×	4.77×

Table 8. Parallel efficiency of the CPU version for a 7×7 kernel ($E = S/P$).

Image Size	4 jobs	8 jobs	12 jobs	16 jobs	20 jobs
800×800	64.9%	53.7%	45.7%	32.6%	28.4%
1600×1600	70.5%	55.6%	40.4%	29.8%	25.0%
3200×3200	66.9%	46.3%	38.0%	28.5%	24.0%
6400×6400	65.0%	46.9%	37.8%	28.2%	24.2%
8000×8000	63.4%	46.8%	36.9%	29.2%	23.9%

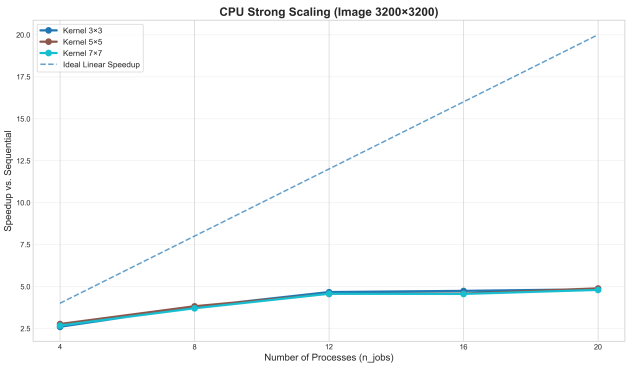

Figure 1. Strong scaling of the CPU-parallel version: speedup relative to the baseline as a function of the number of processes.

Observations on CPU Scalability

- **Sublinear scalability.** Speedup increases as the number of processes grows, but remains significantly below the ideal behavior. The maximum observed speedup is on the order of $6\times$ using 20 processes, despite the presence of 14 physical cores. This result is consistent with process-management overhead and contention for shared resources, in particular caches and memory bandwidth.

- **Reduced marginal gains beyond 12–16 processes.** Beyond 12 processes, the speedup increase becomes progressively limited: moving from 12 to 16 or 20 processes yields gains of only a few tenths. This indicates the onset of a diminishing-returns regime, where performance is primarily constrained by memory bandwidth and parallelization overhead rather than compute capacity.

- **Dependence on image size.** For large images, speedup tends to stabilize in the range $4.5\times$ to $5.0\times$. For smaller images, slightly higher speedups are observed, partly due to better data locality and partly due to the larger relative weight of fixed overheads in the sequential version.

- **Parallel efficiency.** Defining efficiency as $E = S/P$, a systematic reduction is observed as the number of processes increases: from about 60–70% with 4 processes

down to about 24–30% with 20 processes. This behavior highlights that increasing parallelism does not translate proportionally into useful work.

9.3. CUDA Implementation Performance

The GPU implementation was tested using different tile sizes, with the goal of evaluating the effect of **tiling** and **shared memory** usage on performance. Tables 9, 11, and 13 report the speedup of the CUDA implementation relative to the sequential version for kernel sizes 3×3, 5×5, and 7×7, respectively. Absolute execution times, expressed in milliseconds, are reported in Tables 10, 12, and 14. Figure 2 provides a compact graphical representation of the same data, visually highlighting the speedup trend as a function of image size and tile size.

Table 9. CUDA speedup for a 3×3 kernel

Image Size	tile=8	tile=16	tile=24	tile=32
800×800	424.9×	424.9×	424.9×	382.4×
1600×1600	381.1×	390.9×	390.9×	381.1×
3200×3200	370.8×	373.1×	373.1×	357.4×
6400×6400	362.3×	363.4×	348.3×	336.2×
8000×8000	366.7×	367.8×	351.0×	350.0×

Table 10. CUDA execution time for a 3×3 kernel (ms)

Image Size	tile=8	tile=16	tile=24	tile=32
800×800	9.00	9.00	9.00	10.00
1600×1600	40.00	39.00	39.00	40.00
3200×3200	161.0	160.0	160.0	167.0
6400×6400	646.0	644.0	672.0	696.0
8000×8000	1007.0	1004.0	1052.0	1055.0

Table 11. CUDA speedup for a 5×5 kernel

Image Size	tile=8	tile=16	tile=24	tile=32
800×800	438.4×	394.6×	438.4×	394.6×
1600×1600	390.9×	390.9×	390.9×	381.1×
3200×3200	377.7×	362.0×	377.7×	359.8×
6400×6400	368.5×	369.6×	354.8×	352.2×
8000×8000	367.7×	350.7×	369.1×	348.4×

Table 12. CUDA execution time for a 5×5 kernel (ms)

Image Size	tile=8	tile=16	tile=24	tile=32
800×800	9.00	10.00	9.00	10.00
1600×1600	39.00	39.00	39.00	40.00
3200×3200	161.0	168.0	161.0	169.0
6400×6400	648.0	646.0	673.0	678.0
8000×8000	1009.0	1058.0	1005.0	1065.0

Table 13. CUDA speedup for a 7×7 kernel

Image Size	tile=8	tile=16	tile=24	tile=32
800×800	383.1×	383.1×	425.7×	383.1×
1600×1600	396.4×	396.4×	396.4×	386.8×
3200×3200	368.0×	368.0×	370.3×	355.0×
6400×6400	374.3×	374.9×	376.6×	354.2×
8000×8000	368.5×	369.6×	370.0×	348.6×

Table 14. CUDA execution time for a 7×7 kernel (ms)

Image Size	tile=8	tile=16	tile=24	tile=32
800×800	10.00	10.00	9.00	10.00
1600×1600	40.00	40.00	40.00	41.00
3200×3200	163.0	163.0	162.0	169.0
6400×6400	653.0	652.0	649.0	690.0
8000×8000	1015.0	1012.0	1011.0	1073.0

9.4. CUDA Speedup Relative to the Sequential Version

From the data reported in Tables 9–13 and in Figure 2, it emerges that the CUDA implementation delivers an **extremely high speedup** over the sequential version across all analyzed configurations.

For 800×800 images, speedup reaches values above 400×, while for larger images, up to 8000×8000, it typically remains between 350× and 380×. Speedup is largely stable as image size increases, indicating good **GPU scalability**.

9.5. Observations on the CUDA Implementation

The experimental results highlight several relevant aspects:

- **High speedup.** The CUDA implementation achieves speedups on the order of several hundred relative to the sequential version, confirming the effectiveness of the GPU's **massive parallelism**.
- **Scaling with image size.** Execution time grows proportionally with image size, while speedup remains nearly constant.
- **Limited overhead.** Even for relatively small images, speedup remains high, indicating that data-transfer and kernel-launch costs are limited.
- **Limited impact of kernel size.** Differences among 3×3, 5×5, and 7×7 kernels are modest, suggesting a predominantly **memory-bound** behavior.
- **Benefits of tiling and shared memory.** Using tiling and shared memory reduces global-memory accesses, contributing to the observed performance stability.

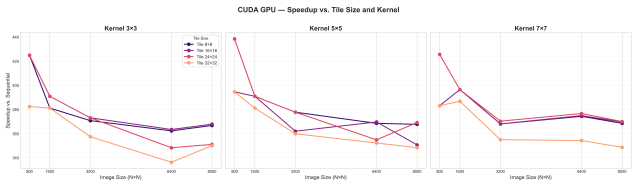

Figure 2. Speedup of the CUDA implementation relative to the sequential version.

9.6. Impact of CUDA Tile Size

Table 15 analyzes the percentage variation in performance as a function of tile size for each kernel.

Table 15. Tile Size Sensitivity – Performance variation (%)

Kernel Size	Avg. Speedup	Tile Sensitivity
3×3	376.1×	26.4%
5×5	378.9×	25.9%
7×7	377.5×	22.1%

Trade-off analysis As shown in Figure 3, configurations with tile sizes from 8×8 to 24×24 exhibit nearly identical execution times, indicating that the kernel operates in a predominantly **memory-bound** regime and that occupancy is not the primary limiting factor.

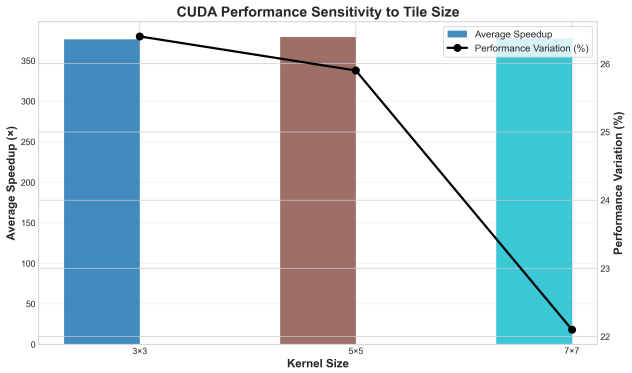

Figure 3. Sensitivity of CUDA performance to tile size: average speedup and percentage variation for different kernel sizes.

The 32×32 tile instead introduces a slight performance degradation, attributable to higher register pressure and reduced effective occupancy. As a result, a 16×16 or 24×24 tile represents the best compromise between **data reuse in shared memory** and efficient utilization of hardware resources.

9.7. Normalized Analysis: Time per MAC

The **time-per-MAC** (multiply–accumulate) metric normalizes performance with respect to the effective computational workload, enabling a fairer comparison across different configurations. Tables 16 and 17 report the results.

Table 16. Time per MAC (ns) – CPU Parallel (best n_jobs)

Image Size	K=3	K=5	K=7
800×800	36.35	13.58	7.16
1600×1600	44.22	16.23	8.42
3200×3200	44.63	16.20	8.31
6400×6400	44.30	16.39	8.40
8000×8000	44.24	16.06	8.33

Table 17. Time per MAC (ns) – CUDA (best tile size)

Image Size	K=3	K=5	K=7
800×800	0.521	0.188	0.096
1600×1600	0.564	0.203	0.106
3200×3200	0.579	0.210	0.108
6400×6400	0.582	0.210	0.108
8000×8000	0.581	0.209	0.108

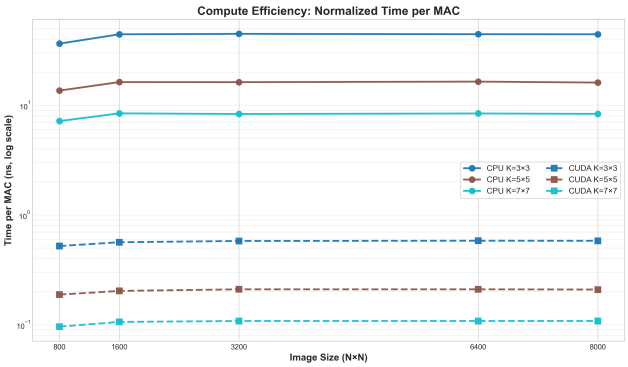

Figure 4. Time per MAC: CPU vs CUDA

Interpretation The time-per-MAC analysis reveals several key aspects:

- **CUDA efficiency**: the GPU is about **70–80**× more efficient than the CPU per MAC, highlighting the advantage of **massive parallelism**.
- **Scaling with** K^2: time per MAC decreases as K increases, confirming that fixed overheads (memory accesses, setup) are amortized more effectively for larger kernels.
- **Stability**: for large images, time per MAC stabilizes, indicating the onset of a steady-state regime where per-operation time is dominated by platform throughput (in particular **memory bandwidth** and **data-access efficiency**), while fixed overheads become less relevant as image size grows.

9.8. Overall Comparison of Implementations

Table 18 summarizes the absolute performance of the three implementations for selected configurations.

Table 18. Runtime Comparison (s)

Resolution	Seq.	CPU Par.	CUDA
800, K3	3.82	0.63	0.01
3200, K5	60.80	12.43	0.16
8000, K7	369.10	76.40	1.00

Considerations on the Computational Regime

- **Memory-bandwidth dominance.** Across all implementations, the main limiting factor is **data access**: the sequential CPU is constrained by main-memory access latency; the parallel CPU by saturation of shared memory bandwidth; and the GPU by global-memory throughput despite the use of shared memory.
- **Architectural scaling.** The GPU is about **75–80**$\times$ faster than the best CPU-parallel configuration, thanks to higher memory bandwidth and the available massive parallelism.
- **Memory-bound regime.** For small kernels (3×3–7×7), all implementations operate in a **memory-bound** regime. For larger kernels, one would expect a transition toward a **compute-bound** regime, where the GPU advantage could become even more pronounced.

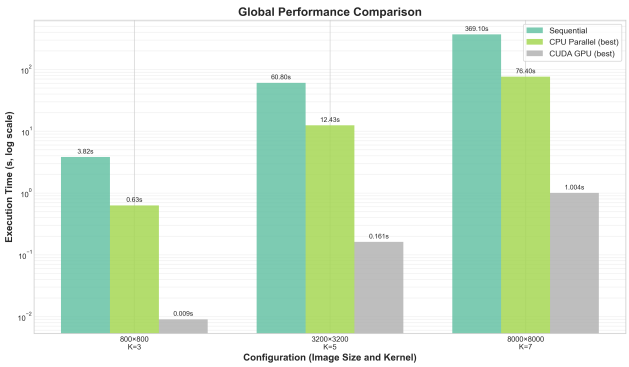

Figure 5. Overall performance: sequential, CPU, CUDA

10. Conclusions

In this work, we studied 2D convolution on RGB images as a case study for optimizing **compute-/memory-intensive workloads**, comparing three implementation approaches: a sequential CPU baseline, a CPU-parallel version based on processes with shared memory, and a CUDA-based GPU version. The goal was not only to measure absolute runtimes, but to understand *which* design choices (domain partitioning, data reuse, memory management) determine the observed performance gains.

10.1. Main Results

The benchmarks confirm that parallelization is highly effective, but with very different characteristics across the two architectures:

1. **Multi-core CPU: tangible gains but early saturation.** The CPU-parallel version achieves speedups up to about $6\times$, with benefits that grow rapidly for the first cores/processes and then tend to plateau. The drop in efficiency as parallelism increases is consistent with a limitation dominated by the memory hierarchy: bandwidth contention, cache pressure, and process-coordination overhead reduce marginal gains beyond the saturation threshold.
2. **CUDA GPU: two orders of magnitude acceleration.** The CUDA version reaches speedups on the order of 10^2–10^3 relative to the baseline, maintaining stable and scalable runtimes even for very large images. The shared-memory *tiling* strategy increases data locality and reuse within each block, mitigating the cost of repeated global-memory accesses and making execution robust to problem size.
3. **Memory-bound nature of the workload.** Normalization via time per MAC shows that, for small kernels (3×3–7×7), the main bottleneck remains **data access** rather than raw compute capability. In this regime, the GPU is markedly more efficient per elementary operation thanks to available bandwidth and parallelism, while the CPU hits the limits imposed by the memory subsystem earlier.

10.2. Contributions

This work provides four main contributions:

- **Structured experimental comparison**: quantitative evaluation over a controlled set of image sizes, kernel sizes, and parallelization parameters (CPU processes and CUDA tile sizes), in order to isolate the effect of configuration choices.
- **Architectural interpretation of results**: explanation of trends in terms of memory-bandwidth saturation, data locality, and synchronization/overhead costs, linking measurements to performance causes.
- **Normalized metric**: use of **time per MAC** to compare intrinsic efficiency across implementations independently of problem scale.

- **Reproducible implementations**: availability of three complete versions (sequential CPU, parallel CPU, CUDA) usable as a baseline for extensions and as a methodological reference for similar benchmarks.

10.3. Limitations and Future Work

The results are encouraging, but should be interpreted in light of some limitations:

- **Kernel scope**: the analysis focuses on 3×3–7×7 kernels, for which behavior is largely memory-bound; larger kernels (or operators with higher arithmetic intensity) could alter the compute/memory balance.
- **Hardware dependence**: benchmarks were run on a single platform; differences in cache, RAM bandwidth, and GPU architecture can affect both absolute runtimes and optimal parameters (number of processes, tile size).
- **Non-exhaustive CUDA optimizations and transfers**: the CUDA version adopts tiling and shared memory but does not explore additional techniques; moreover, including Host–Device transfers penalizes pipeline scenarios where data remain resident on the device.

Natural future directions include:

1. **Separable convolutions**: for Gaussian/derivative filters, reducing complexity from $O(K^2)$ to $O(2K)$.
2. **Mixed precision**: FP16/INT8 (when admissible) to increase throughput and reduce memory traffic.
3. **End-to-end evaluation and tuning**: measuring performance in pipelines (data resident on the device) and introducing auto-tuning for CUDA tile size and CPU parallelism.

10.4. Final Remarks

Overall, the analysis shows that performance differences across implementations do not stem solely from parallelism, but primarily from the ability to control **data access**. The parallel CPU approach provides a significant improvement with limited development effort, but quickly reaches a diminishing-returns regime imposed by the memory subsystem. The GPU, thanks to massive parallelism and higher bandwidth, enables two-order-of-magnitude accelerations, provided that computation is structured to maximize locality and reuse (tiling, shared memory, grid configuration).

In practical applications, the platform choice should therefore depend on the trade-off between throughput/latency requirements and development complexity: a multi-core CPU is often sufficient for offline processing or moderate workloads, whereas the GPU becomes critical when problem scale or real-time constraints demand high throughput. Understanding these architectural trade-offs—bandwidth, memory hierarchies, synchronization, and execution model—is essential to design truly scalable solutions for modern image processing.

References

[1] Harris, C.R., et al. *Array programming with NumPy*. Nature 585, 357–362 (2020).

[2] Joblib Development Team. *Joblib: running Python functions as pipeline jobs*. `https://joblib.readthedocs.io/`

[3] Gonzalez, R.C., Woods, R.E. *Digital Image Processing*. Pearson, 4th Edition, 2018.

[4] NVIDIA Corporation. *CUDA C Programming Guide*. `https://docs.nvidia.com/cuda/`

[5] Intel Corporation. *Intel Math Kernel Library*. `https://software.intel.com/mkl`

[6] Python Software Foundation. *multiprocessing.shared_memory*. Python 3.11 Documentation, 2023.

[7] Rocklin, M. *Einsum: More Than Meets the Eye*. NumPy Enhancement Proposal, 2018.

[8] Amdahl, G.M. *Validity of the single processor approach to achieving large scale computing capabilities*. AFIPS Conference Proceedings, 1967.

[9] Lapemaya, Cappetti99. `https://github.com/Cappetti99/Game-of-Life` *Parallel Image Filters*. GitHub repository, 2026.