

Image Convolution with Filters: Performance Optimization through Parallelization

Lorenzo Cappetti, Lapo Chiostrini

January 2026



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Table of Contents

1 Introduction

► Introduction

► Approach

► Results

► Conclusions

The Problem: 2D Image Convolution

1 Introduction

- 2D convolution is a fundamental operation in **image processing**
- Applications: blurring, sharpening, edge detection, custom filters
- **Computationally expensive:** for each output pixel:
 - Read a $K \times K$ neighborhood from the input image
 - Multiply values by the kernel coefficients
 - Accumulate (sum) the partial results
- For large images ($>4K$) and larger kernels (7×7), runtime becomes critical

Computational Complexity

For an RGB image of size $N \times N$ and a $K \times K$ kernel:

- **Output:**
 $(N - K + 1) \times (N - K + 1) \times 3$
- **MACs:** $\approx N^2 \cdot K^2 \cdot 3$
- **Memory accesses:**
 $\approx N^2 \cdot K^2 \cdot 3 \cdot 2$ (read + write)

Project Goals

1 Introduction

Main Objectives

1. Implement a **sequential** version (Python baseline)
2. Parallelize on a **multi-core CPU** (Python + joblib)
3. Accelerate on **GPU** (CUDA)
4. Compare performance and analyze speedups

Technical Challenges

- **Memory-bound**: many reads, relatively little computation
- Optimize **data locality** and cache reuse
- Minimize synchronization and scheduling **overhead**
- Handle different image sizes (800px – 8000px)
- Support multiple kernel sizes (3x3, 5x5, 7x7)

Key question: How much can we improve performance through effective parallelization?

Table of Contents

2 Approach

► Introduction

► Approach

► Results

► Conclusions

Sequential Algorithm

2 Approach

Strategy: simple and readable baseline

- Two nested loops over the output pixels
- For each pixel: weighted sum over a $K \times K$ neighborhood
- Process 3 channels (RGB)
- **Reference baseline:** used to validate correctness and compute speedups
- **Low data reuse:** nearby outputs repeatedly load overlapping input regions

Trade-offs

- **Pros:** clarity, easy debugging, correct baseline
- **Cons:** high interpreter overhead (nested loops)
- **Bottleneck:** memory traffic dominates; $O(H \cdot W \cdot K^2)$ scaling

CPU Parallelization

2 Approach

Idea: split the workload into independent tasks

- Spatial partitioning: split the image into blocks
- Each block produces a portion of the output
- Shared-memory access to the input data (when possible)
- Scheduling with `joblib.Parallel`
- Configurable number of processes
- Kernel reused across all tasks

Key sources of overhead

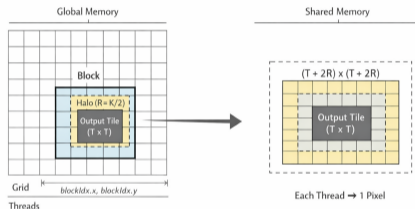
- Pickling / IPC between processes (if data are not shared)
- Task granularity: too small \Rightarrow high overhead
- Memory pressure: tasks may copy portions of the image

GPU Approach (CUDA)

2 Approach

Idea: one thread computes one output pixel

- Shared-memory tiling: overlapping neighborhoods are loaded once from global memory and reused within the block
- Coalesced global memory loads so adjacent threads access adjacent addresses
- Constant memory for kernel coefficients
- Loop unrolling to reduce loop overhead
- Pinned memory to accelerate Host-Device transfers



Shared-memory tiling (tile+halo): load once, reuse across threads.

Table of Contents

3 Results

► Introduction

► Approach

► Results

► Conclusions

Experimental Setup

3 Results

Benchmark methodology

- Vary one parameter at a time (kernel size, image size, CPU processes, CUDA tile size)
- Report **runtime** and **speedup** relative to the sequential baseline

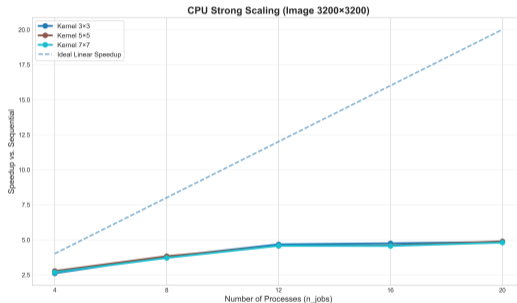
Parameter	Tested values
Kernel size ($K \times K$)	3×3 , 5×5 , 7×7
Image size (pixels)	800^2 , 1600^2 , 3200^2 , 6400^2 , 8000^2
Number of CPU processes	4, 8, 12, 16, 20
CUDA tile size	8×8 , 16×16 , 24×24 , 32×32

All speedups are computed with respect to the sequential Python implementation.

Results: CPU Speedup (Parallel)

3 Results

- **Sublinear strong scaling:** speedup increases with the number of processes but remains far from ideal.
- **Maximum observed:** $\approx 5\times$ with 20 processes.
- **Saturation** beyond 12–16 processes: diminishing marginal gains.
- Main causes: **cache/memory bandwidth contention** + process coordination overhead.

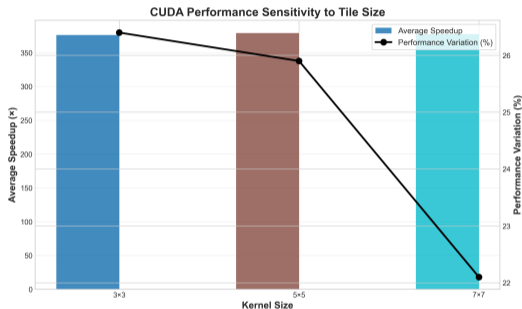


Strong scaling of the CPU-parallel version: speedup relative to the baseline vs. number of processes.

Results: GPU Speedup (CUDA)

3 Results

- **Two orders of magnitude speedup:** typically $350\text{--}380\times$ (up to $\sim 400\times$ on smaller images).
- **Stable scaling with image size:** runtime grows with input size, but **speedup stays nearly constant**.
- **Limited overhead:** performance gains remain high even for moderately sized images.
- **Limited kernel-size impact** for $3 \times 3\text{--}7 \times 7 \Rightarrow$ predominantly **memory-bound** behavior.

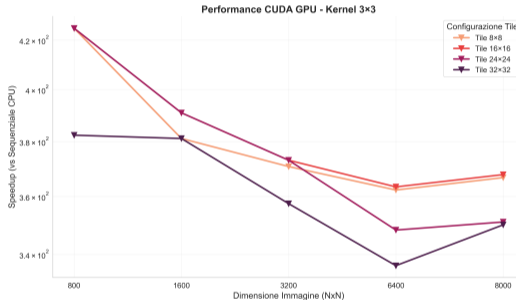


Average CUDA speedup and performance variability (%) across tile sizes for 3×3 , 5×5 , and 7×7 kernels.

CUDA: Impact of Tile Size

3 Results

- Tiles 8×8 – 24×24 : very similar runtimes \Rightarrow **memory-bound regime**.
- Tile 32×32 : slight slowdown (higher register pressure, reduced effective occupancy).
- Typical trade-off: 16×16 or 24×24 .
- 16×16 is the best one.

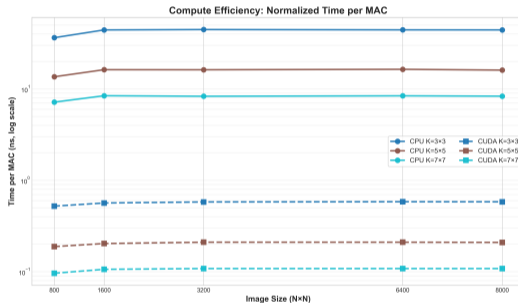


Sensitivity of CUDA performance to tile size: average speedup and percentage variation for different kernel sizes.

Normalized Analysis: Time per MAC

3 Results

- The **time-per-MAC (multiply-accumulate)** metric compares “intrinsic” efficiency by normalizing for the workload.
- The GPU is about **70–80×** more efficient per MAC than the CPU.
- As K increases, time/MAC decreases (overheads are amortized).
- For large images, time/MAC stabilizes \Rightarrow bandwidth-dominated throughput.

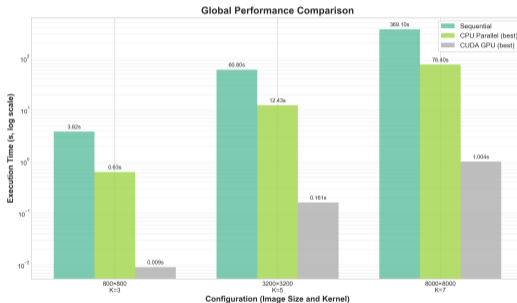


Time per MAC: CPU vs. CUDA.

Overall Comparison: CPU vs. CUDA

3 Results

- Across all versions, the main limiting factor is **data access** (memory-bandwidth dominated).
- The GPU is **75–80×** faster than the best CPU-parallel configuration.
- For 3×3 – 7×7 kernels: all implementations operate in a **memory-bound** regime.



Overall performance: sequential vs. CPU-parallel vs. CUDA.

Table of Contents

4 Conclusions

► Introduction

► Approach

► Results

► Conclusions

Conclusions: Key Takeaways

4 Conclusions

- **CPU parallelization:** maximum speedup of $\sim 6\times$ (20 processes), with **saturation** beyond 12–16 processes.
 - Parallel efficiency drops due to **memory bandwidth/cache contention** + overhead (process management / data movement).
- **CUDA acceleration:** speedups on the order of **350–400 \times** , remaining high even for moderately sized images.
 - **Shared-memory tiling** improves locality and data reuse, reducing the cost of repeated global memory accesses.
- **Dominant regime:** for small kernels (3×3 – 7×7), all implementations are **predominantly memory-bound**.
 - Confirmed by the **time-per-MAC** metric: the GPU is **70–80 \times** more efficient per MAC than the CPU.

Lessons Learned

4 Conclusions

CPU parallelism

- **More processes \neq more performance:** beyond a threshold, **bandwidth** dominates and speedup flattens.
- Task **granularity** matters: too fine \Rightarrow overhead; too coarse \Rightarrow poor load balancing.
- Avoid unnecessary copies: **data sharing** and memory layout have a real impact.

CUDA optimization

- Performance comes from **coalescing + tiling + shared-memory reuse**.
- **Tile size:** bigger is not always better (32×32 can hurt due to register pressure / occupancy).
- Metrics matter: **time-per-MAC** helps distinguish compute-bound vs. memory-bound behavior.

Recommended practice: start with a correct baseline, profile, then optimize memory before compute.

Image Convolution with Filters: Performance Optimization through Parallelization

Thank you for listening!