# Parallelization of kernel image processing

Samuele Marrani
samuele.marrani@edu.unifi.it

Mathilde Patrissi
mathilde.patrissi@edu.unifi.it

## Abstract

*Kernel Image Processing focuses on digital image filtering techniques based on **kernel convolution**, a fundamental operation in image processing. A kernel is a small matrix applied to each pixel of an image, where the new pixel value is computed as a weighted sum of its neighboring pixels. This approach enables common filtering operations such as blurring, sharpening, and edge detection.*
*The main goal of this project is to compare the computational performance of three different implementations of kernel-based image processing: a **sequential C++** implementation, a parallel CPU version using **OpenMP**, and a GPU-based implementation using **CUDA**. The comparison highlights the impact of parallelism on execution time and efficiency across different hardware architectures.*



## 1 Introduction

In image processing, a *kernel* (also referred to as a *convolution matrix* or *mask*) is a small matrix of numerical values used to apply spatial filters to digital images. The application of a kernel to an image is performed through a convolution operation, in which each output pixel is computed as a weighted combination of neighboring input pixels according to the kernel coefficients. Kernel-based filtering represents a fundamental technique in digital image processing and is widely used for operations such as image smoothing, sharpening, and edge detection. These methods operate directly in the spatial domain and form the basis of many classical computer vision algorithms as well as modern approaches such as convolutional neural networks. Kernel Image Processing is particularly well suited for parallel computation, since the convolution operation is applied independently to each pixel. This makes kernel image processing an ideal case study for evaluating different computational paradigms, ranging from sequential execution on a single CPU core to parallel implementations on multi-core CPUs and GPUs.

### 1.1 Mathematical formulation

From an algorithmic perspective, kernel image processing is based on the discrete two-dimensional convolution operation. Let $I(x, y)$ denote the input image and $K(i, j)$ a convolution kernel of size $(2m + 1) \times (2n + 1)$, centered at position $(0, 0)$. The output image $O(x, y)$ is obtained by sliding the kernel across the image and computing, for each pixel, the weighted sum of the corresponding neighborhood:

$$O(x, y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} I(x + i,\, y + j) \cdot K(i,\, j)$$

This operation consists of an element-wise multiplication between the kernel coefficients and the underlying pixel values, followed by a summation that produces the new pixel value. The convolution can be interpreted as a linear filtering operation in the spatial domain. In practical implementations, special care must be taken when processing image boundaries, since the kernel can partially extend beyond the image limits. Common strategies include zero padding, border replication, or mirroring. Due to its regular memory access pattern and high degree of data parallelism,

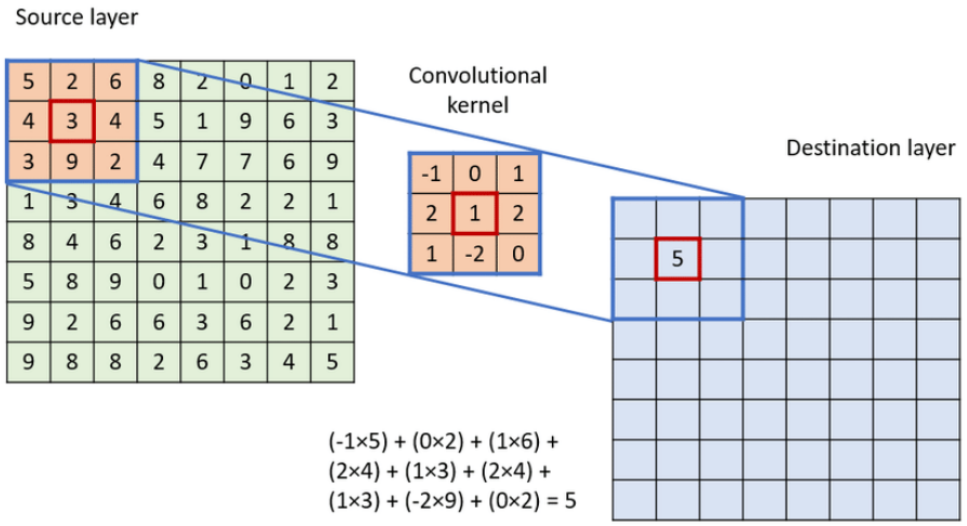



Figure 1: Kernel image processing graphical explanation.

the convolution operation is particularly suitable for optimization on parallel architectures such as CPUs with OpenMP and GPUs with CUDA.

### 1.2 Presented filters

In this section we present a set of discrete spatial filters commonly employed for edge detection, image sharpening, second-order differentiation, and noise reduction. Each filter is defined by a finite convolution kernel $K \in \mathbb{R}^{n\times n}$, with odd spatial support $n \in \{3, 5\}$, applied to an input image $I$ through standard discrete convolution.

**Edge Detection Filters** Edge detection kernels are designed to emphasize high-frequency components of the image, corresponding to abrupt intensity variations. The proposed $3 \times 3$ and $5 \times 5$ edge filters are isotropic and zero-sum, with a strong positive central coefficient surrounded by negative weights. This structure approximates a discrete second-order derivative operator, enhancing regions with large local contrast while suppressing uniform areas. These kernels can be interpreted as high-pass filters whose frequency response attenuates low spatial frequencies and amplifies high-frequency components associated with edges. An example is shown in figure 2.

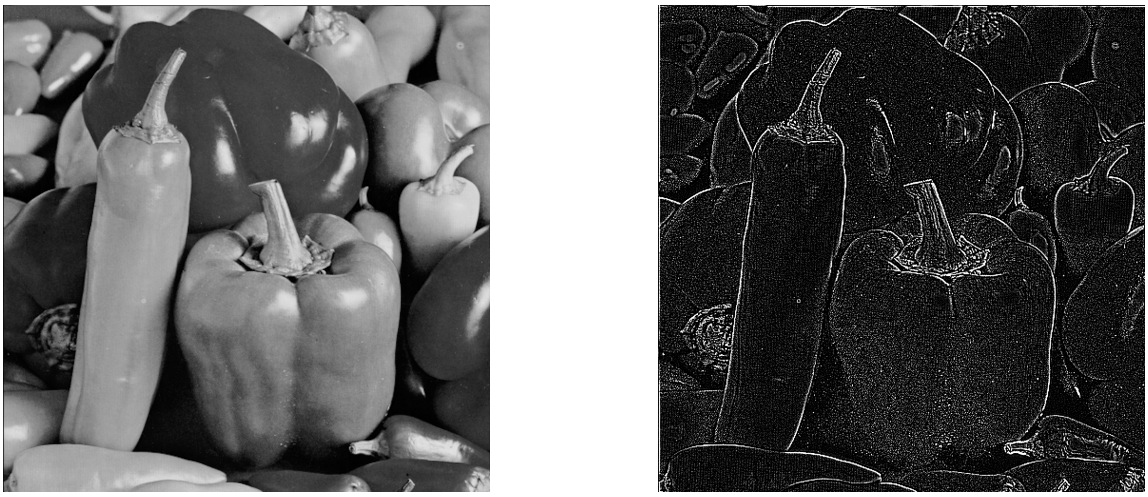
**Sharpening Filters** Sharpening kernels follow a similar structure to edge detection filters but are scaled such that the sum of all coefficients equals one. This guarantees preservation of the global image intensity while reinforcing high-frequency details. From a signal processing perspective, sharpening can be seen as the superposition of the original image and a scaled high-pass component, yielding enhanced local contrast without altering the mean luminance.

**Laplacian Filters** The Laplacian kernels provide a discrete approximation of the continuous Laplace operator

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}.$$

These filters are rotationally symmetric and emphasize regions where the second derivative of intensity is large, such as edges and fine structures. Unlike first-order gradient operators, the Laplacian responds equally to edges of all orientations but is more sensitive to noise, which motivates its frequent combination with smoothing operators.

**Gaussian Smoothing Filters** Gaussian filters implement a low-pass filtering operation by convolving the image with a discrete approximation of a two-



(a) Original image

(b) Filtered image

Figure 2: Edge filtering result (1024x1024 with 5x5 kernel).

dimensional Gaussian function

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right).$$

The presented kernels correspond to normalized discrete samples of this function for different kernel sizes. Their coefficients are strictly positive and sum to one, ensuring energy preservation and effective attenuation of high-frequency noise while maintaining spatial coherence. Two variants are considered: analytically normalized Gaussian kernels and empirically precomputed approximations. While both achieve similar smoothing behavior, the latter allow greater flexibility in tuning the effective standard deviation without recomputing normalization factors at runtime.

**Kernel Size Considerations** Increasing the kernel size from $3 \times 3$ to $5 \times 5$ generally improves frequency selectivity and robustness to noise at the cost of higher computational complexity and reduced sensitivity to fine details. The choice of kernel size thus represents a trade-off between spatial precision and noise suppression, depending on the target application.

### 1.3 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface for parallel programming on shared-memory architectures, widely adopted on modern multi-core systems. It enables an incremental parallelization approach, allowing developers to extend sequential programs written in C, C++, or Fortran through compiler directives, runtime library routines, and environment variables. OpenMP follows a *fork–join* execution model, where a master thread spawns a team of worker threads to execute parallel regions and synchronizes with them at the end of each region. The programmer can explicitly control work sharing, synchronization, and data scoping, achieving a balance between ease of use and expressive power.

### 1.4 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on graphics processing units (GPUs). CUDA adopts a heterogeneous execution model in which the CPU (host) manages the execution of massively parallel kernels on the GPU (device).
CUDA exposes a hierarchical execution model:

kernels are executed by a large number of lightweight threads organized into thread blocks and grids. This hierarchy maps directly onto the underlying GPU hardware and enables fine-grained data parallelism at scale. The CUDA *memory model* is explicit and hierarchical, reflecting the diverse memory spaces available on the GPU. These include *global memory*, which is accessible by all threads but has relatively high latency; *shared memory*, which is shared among threads within the same block and offers low-latency access; *constant memory*, a specialized, read-only memory space; and per-thread private memory such as registers and local memory. Efficient utilization of these memory spaces is critical for achieving high performance and requires careful management by the programmer.

# 2 Proposed solutions

In this project, several sequential and parallel solutions are proposed to address the *Kernel image processing (KIP)* problem. All of them are written in the C++ programming language and some of them also use CUDA C dialect. These approaches consist of the following versions:

1. Sequential;
2. Parallel OMP;
3. Parallel CUDA.

## 2.1 Common Implementation Aspects

All approaches share a common design in terms of data representation and processing workflow, ensuring a fair comparison across different execution models. The processing pipeline consists of three main stages:

1. image acquisition and preprocessing, where input images are converted to grayscale and padded to handle boundary conditions;
2. kernel convolution, during which filter coefficients are applied to the image data;
3. post-processing, where output values are prepared for storage.

Images are represented as one-dimensional arrays of floating-point values in row-major order to optimize memory access locality. Pixel intensities are normalized within the range [0, 255], and explicit padding is applied to manage border effects during convolution. Convolution kernels are defined as square matrices with odd dimensions (e.g., 3×3, 5×5). Kernel coefficients are normalized to preserve overall image brightness, and their memory layout is adapted to the access patterns of each execution model.

## 2.2 Sequential Implementation

The sequential implementation performs the convolution by iterating over each output pixel and computing a weighted sum of neighboring pixels from a padded version of the input image. The computation is organized using nested loops and follows a straightforward row-major memory access pattern. Each output pixel is processed independently and written to a separate buffer, ensuring correctness and simplicity.

```
    std::vector<float> result(width *
        height);
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float sum = 0.0f;
            for (int ky = 0; ky <
                kernelSize; ky++) {
                int py = (y + ky)*
                    paddedwith +x;
                int pky=ky * kernelSize;
                for (int kx = 0; kx <
                    kernelSize; kx++) {
                    float pixelValue =
                        padded[py + kx];
                    float kernelValue =
                        kernel[pky + kx];
                    sum += pixelValue *
                        kernelValue;
                }
            }
            result[y * width + x] =
                std::min(255.0f,
                std::max(0.0f, sum));
        }
    }
```

### 2.3 Parallel Implementation

The OpenMP-based parallel implementation extends the sequential approach by exploiting shared-memory parallelism. The outer loops of the convolution process are parallelized using OpenMP directives, enabling multiple threads to process distinct image regions concurrently.

```
#pragma omp parallel for collapse(2)
    schedule(dynamic)
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float sum = 0.0f;
            for (int ky = 0; ky <
                kernelSize; ky++) {
                int py = (y + ky)*
                    paddedwith +x;
                int pky=ky * kernelSize;
                for (int kx = 0; kx <
                    kernelSize; kx++) {
                    float pixelValue =
                        padded[py + kx];
                    float kernelValue =
                        kernel[pky + kx];
                    sum += pixelValue *
                        kernelValue;
                }
            }
            result[y * width + x] =
                std::min(255.0f,
                std::max(0.0f, sum));
        }
    }

    imgProc.setImageData(result, width,
        height);
    return true;
}
```

Dynamic scheduling is employed to improve load balancing because, first of all, in a real operating system, other processes can consume CPU resources unpredictably. This is a case of external interference: if a core is temporarily occupied by another system task, the threads assigned to that core would be suspended, while the others would continue their work. Furthermore, there is the situation where threads share the same core when their number exceeds the number of physical cores. This would result in different execution times for the threads. Additionally, the rows containing edge pixels will have a different processing time due to the introduction of conditional jumps that are not present for central pixels. Lastly, thread safety is ensured by assigning independent output regions to each thread.

### 2.4 CUDA Implementation

The CUDA implementation takes advantage of GPU parallelism by applying three separate memory optimization strategies, each realized as an individual kernel. Fixed block dimensions are used and kernels up to 25×25 in size are supported (`MAX_KERNEL_SIZE = 25`). Thread blocks are arranged in a 16×16 layout, while grid dimensions are calculated dynamically based on the image size. The implementation handles padded images to properly manage border conditions, and the output values are clamped between 0 and 255 using `fmaxf` and `fminf`.

**Global Memory Implementation** The baseline implementation allocates and manages both the input image data and the convolution kernel in global memory using `cudaMalloc` and `cudaMemcpy`. This straightforward approach incurs higher memory access latency compared to shared or constant memory optimizations, as all threads must directly access global memory for both the image and kernel data during convolution operations.

```
    cudaMemcpy(d_input,
        padded.data(),paddedWidth *
        paddedHeight * sizeof(float),
        cudaMemcpyHost ToDevice);
    cudaMalloc(&d_kernel, kernelSize *
        kernelSize * sizeof(float));
    cudaMemcpy(d_kernel,
        filter.getKernelData().data(),
        kernelSize * kernelSize *
        sizeof(float),cudaMemcpyHost
        ToDevice);

    processGlobalMem <<<gridSize,
        blockSize >>> (
        d_input, d_output, d_kernel,
        width, height, paddedWidth,
            paddedHeight,
        kernelSize );
```

Each thread computes its pixel coordinates using `blockIdx` and `threadIdx`, directly accessing global memory for both the input image and kernel coefficients.

```
    const int x = blockIdx.x * blockDim.x
        + threadIdx.x;
    const int y = blockIdx.y * blockDim.y
        + threadIdx.y;
    if (x >= width || y >= height) return;
    float sum = 0.0f;
    for (int ky = 0; ky < kernelSize;
        ky++) {
        int py = (y + ky)* paddedWidth +x;
        int pky=ky * kernelSize;
        for (int kx = 0; kx < kernelSize;
            kx++) {
            float pixelValue = padded[py +
                kx];
            float kernelValue = kernel[pky
                + kx];
            sum += pixelValue*kernelValue;
        }
    }
    sum = fmaxf(0.0f, fminf(sum, 255.0f));
    d_output[y * width + x] = sum;
```

**Constant Memory Implementation** This implementation optimizes kernel coefficient access by storing the convolution kernel in constant memory using `cudaMemcpyToSymbol`.

```
__device__ __constant__ float
    d_filterKernel[MAX_KERNEL_SIZE *
    MAX_KERNEL_SIZE];

[...]

    cudaMemcpy(d_input,
        padded.data(),paddedWidth*
        paddedHeight*sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(d_filterKernel,
        filter.getKernelData().data(),
        kernelSize*kernelSize*sizeof(float));
    processConstantMem <<<gridSize,
        blockSize >>> ( d_input, d_output,
        width, height, paddedWidth,
            paddedHeight, kernelSize);
```

By exploiting the low-latency, read-only characteristics of constant memory, this approach reduces access overhead and improves overall kernel performance. While the image data remains in global memory, caching the kernel coefficients in constant memory enhances access patterns when multiple threads concurrently read identical kernel values.

```
    const int x = blockIdx.x * blockDim.x
        + threadIdx.x;
    const int y = blockIdx.y * blockDim.y
        + threadIdx.y;
    if (x >= width || y >= height) return;
    float sum = 0.0f;
    for (int ky = 0; ky < kernelSize;
        ky++) {
        int py = (y + ky)* paddedWidth +x;
        int pky=ky * kernelSize;
        for (int kx = 0; kx < kernelSize;
            kx++) {
            float pixelValue = padded[py +
                kx];
            float kernelValue =
                d_filterKernel[pky + kx];
            sum += pixelValue *
                kernelValue;
        }
    }
    sum = fmaxf(0.0f, fminf(sum, 255.0f));
    d_output[y * width + x] = sum;
```

**Shared Memory Implementation** The most advanced implementation leverages shared memory to construct a cached tile of the input image within each thread block.

This approach adopts a two-phase strategy: initially, threads cooperatively load a block of image data, including the necessary external pixels, into shared memory arrays and synchronize using `__syncthreads()`.

Subsequently, the convolution operation is executed utilizing the cached data. This methodology necessitates meticulous handling of boundary conditions and explicit thread synchronization, yet it substantially reduces the overhead associ-

ated with global memory accesses.

```
const int radius = kernelSize / 2;
__shared__ float
    sharedMem[BLOCK_SIZE+MAX_KERNEL_SIZE
    /2][BLOCK_SIZE+MAX_KERNEL_SIZE/2];
const int tx = threadIdx.x;
const int ty = threadIdx.y;
const int x = blockIdx.x *
    (blockDim.x-2*radius) + tx;
const int y = blockIdx.y *
    (blockDim.y-2*radius) + ty;
const int xx = blockIdx.x *
    (blockDim.x-2*radius) + tx-radius;
const int yy = blockIdx.y *
    (blockDim.y-2*radius) + ty-radius;

if (y >= 0 && y < paddedHeight && x >=
    0 && x < paddedWidth)
        sharedMem[ty][tx] =  d_input[y
            * paddedWidth + x];

__syncthreads();

if (x>=0 && x < (width+radius) && y>=0
    && y < (height+radius))
{
  if (tx>=radius && tx <
      (blockDim.x-radius) && ty>=radius
      && ty < (blockDim.y-radius))
  {
    float sum = 0.0f;
    for (int ky = 0; ky < kernelSize;
        ky++) {
        for (int kx = 0; kx <
            kernelSize; kx++) {
            float pixelValue =
                sharedMem[ky+ty-radius]
                [kx+tx-radius];
            float kernelValue =
                d_filterKernel[ky *
                kernelSize +kx ];
            sum += pixelValue *
                kernelValue;
        }
    }
    sum = fmaxf(0.0f, fminf(sum,
        255.0f));
    d_output[yy * width + xx] = sum;
  }
}
```

# 3 Experimental setup and results

This section describes the experimental setup and the experimental campaign conducted to evaluate the performance of the image filtering application implemented using different computational paradigms. The application applies a convolution filter to an input image and measures the execution time of each implementation. To ensure a fair comparison, all implementations are evaluated under identical conditions.
The experimental analysis focuses exclusively on the *gaussian filter*, as it represents the most computationally relevant and widely used convolution filter among those implemented. The gaussian filter is particularly suitable for performance evaluation due to its regular access pattern and its sensitivity to kernel size and memory access strategies.

The tests are performed by varying the following parameters:

- convolution **kernel size**: $3 \times 3$ and $5 \times 5$;
- input **image resolution**: $512 \times 512$, $1024 \times 1024$, and $2048 \times 2048$ pixels;
- number of **OpenMP threads**: 2, 4, and 8;
- **CUDA memory model**: global memory, shared memory, and constant memory.

For each test configuration, the input image is loaded from disk and the gaussian convolution kernel is generated according to the selected kernel size. After the filtering operation, the resulting image is saved to disk and the execution time is measured using high-resolution timers.
All timings are collected in microseconds and stored for subsequent analysis.

All experiments were conducted on a system with the following hardware and software specifications:

- CPU: Intel Core i7 CPU 870 (4 cores - 8 threads, 2.93GHz)

- GPU: NVIDIA GeForce GT 710 (192 cores, 2GB)
- RAM: 32GB (4x8GB) DDR3-1600MHz PC3-12800
- Storage sda1: SSD 240GB KINGSTON A400
- Storage sdb1: SSD 240GB KINGSTON A400
- Storage sdc1: SSD 240GB KINGSTON A400
- Operating System: Linux Mint 20.3 MATE 1.26.0

### 3.1 Sequential CPU implementation

The sequential CPU implementation serves as the baseline for performance comparison. The gaussian filtering operation is executed on a single CPU core by iterating over all image pixels and directly applying the convolution kernel, without any form of parallelism. For each image resolution and kernel size, the execution time is measured by recording timestamps immediately before and after the convolution process. Upon completion, the filtered image is saved to disk. The collected execution times provide a reference point for evaluating the performance gains achieved by parallel CPU and GPU implementations.

### 3.2 OpenMP parallel implementation

The OpenMP implementation exploits shared-memory parallelism on the CPU by distributing the gaussian filtering workload across multiple threads. Multiple experimental runs are performed by varying the number of threads, specifically using 2, 4, and 8 threads, in order to analyze scalability and parallel efficiency. For each combination of image resolution, kernel size, and thread count, the convolution operation is executed in parallel. Execution time is measured using the same methodology adopted for the sequential implementation, ensuring consistency across tests. Each run produces an output image and a corresponding timing record that includes the number of threads used, enabling a direct comparison of performance improvements relative to the baseline. For each thread configuration, the filtering operation is executed in parallel, and the execution time is measured in the same manner as in the sequential case. This approach allows the evaluation of scalability with respect to the available CPU resources.

### 3.3 CUDA GPU implementation

The CUDA implementation offloads the gaussian convolution operation to the GPU, exploiting data-level parallelism. To evaluate the impact of different memory access strategies on performance, multiple CUDA memory configurations are tested. Specifically, three memory models are considered: global memory, shared memory, and constant memory. For each memory configuration, the gaussian filter is applied to images of varying resolutions and kernel sizes. Before each execution, the availability of GPU resources is verified. The execution time is measured using high-resolution timers, and the resulting image is saved to disk. Each test is recorded together with metadata describing the memory model employed and the kernel size. This experimental setup allows a detailed analysis of how different CUDA memory strategies influence performance when applying a computationally intensive convolution filter.

| Gaussian filter | | |
|---|---|---|
| Kernel | $3 \times 3$ | |
| Image size | $512 \times 512$ | |
| Sequential | 34.78 ms | |
| | Configuration | Execution time [ms] |
| OpenMP | 2 threads | 29.56 |
| | 4 threads | 17.22 |
| | 8 threads | 15.47 |
| CUDA | Global memory | 9.81 |
| | Constant memory | 10.77 |
| | Shared memory | 8.92 |

Table 1: Execution times for the gaussian filter using a $3 \times 3$ kernel on a $512 \times 512$ image. Comparison between sequential CPU, OpenMP, and CUDA implementations.

| Gaussian filter | | |
|---|---|---|
| Kernel | $3 \times 3$ | |
| Image size | $1024 \times 1024$ | |
| Sequential | 148.97 ms | |
| | Configuration | Execution time [ms] |
| OpenMP | 2 threads | 126.59 |
| | 4 threads | 69.55 |
| | 8 threads | 69.40 |
| CUDA | Global memory | 37.45 |
| | Constant memory | 37.75 |
| | Shared memory | 32.58 |

Table 2: Execution times for the gaussian filter using a $3 \times 3$ kernel on a $1024 \times 1024$ image. Comparison between sequential CPU, OpenMP, and CUDA implementations.

| Gaussian filter | | |
|---|---|---|
| Kernel | $3 \times 3$ | |
| Image size | $2048 \times 2048$ | |
| Sequential | 549.19 ms | |
| | Configuration | Execution time [ms] |
| OpenMP | 2 threads | 486.57 |
| | 4 threads | 299.24 |
| | 8 threads | 249.09 |
| CUDA | Global memory | 137.24 |
| | Constant memory | 147.29 |
| | Shared memory | 128.95 |

Table 3: Execution times for the gaussian filter using a $3 \times 3$ kernel on a $2048 \times 2048$ image. Comparison between sequential CPU, OpenMP, and CUDA implementations.

| Gaussian filter | | |
|---|---|---|
| Kernel | $5 \times 5$ | |
| Image size | $512 \times 512$ | |
| Sequential | 72.90 ms | |
| | Configuration | Execution time [ms] |
| OpenMP | 2 threads | 57.23 |
| | 4 threads | 31.41 |
| | 8 threads | 27.23 |
| CUDA | Global memory | 11.97 |
| | Constant memory | 10.64 |
| | Shared memory | 9.57 |

Table 4: Execution times for the gaussian filter using a $5 \times 5$ kernel on a $512 \times 512$ image. Comparison between sequential CPU, OpenMP, and CUDA implementations.

| Gaussian filter | | |
|---|---|---|
| Kernel | $5 \times 5$ | |
| Image size | $1024 \times 1024$ | |
| Sequential | 291.84 ms | |
| | Configuration | Execution time [ms] |
| OpenMP | 2 threads | 227.32 |
| | 4 threads | 145.96 |
| | 8 threads | 107.43 |
| CUDA | Global memory | 40.99 |
| | Constant memory | 40.47 |
| | Shared memory | 33.54 |

Table 5: Execution times for the gaussian filter using a $5 \times 5$ kernel on a $1024 \times 1024$ image. Comparison between sequential CPU, OpenMP, and CUDA implementations.

| Gaussian filter | | |
|---|---|---|
| Kernel | $5 \times 5$ | |
| Image size | $2048 \times 2048$ | |
| Sequential | 1127.53 ms | |
| | Configuration | Execution time [ms] |
| OpenMP | 2 threads | 892.88 |
| | 4 threads | 508.34 |
| | 8 threads | 423.92 |
| CUDA | Global memory | 153.30 |
| | Constant memory | 152.39 |
| | Shared memory | 132.64 |

Table 6: Execution times for the gaussian filter using a $5 \times 5$ kernel on a $2048 \times 2048$ image. Comparison between sequential CPU, OpenMP, and CUDA implementations.

### 3.4 Results summary

In this section we present a global summary of the execution times for the gaussian filter, considering sequential and parallel implementations. Tables n.7 and n.8 show the sequential time, the best parallel time and the relative speedup.
Several interesting observations emerge from these data:

- **Scaling with image size and threads:** As expected, the OpenMP implementation shows an improvement in execution time when increasing the number of threads. However, the benefit is limited compared to the GPU implementations. The speedup from OpenMP is modest, since the overhead of thread management becomes a significant

fraction of the total execution time.
OpenMP is consistently slower than the best CUDA variant.

- **GPU memory impact:** Among the CUDA memory models, shared memory consistently provides the best performance across all image sizes and kernel sizes. This confirms the importance of optimizing memory access patterns on the GPU.
  Global and constant memory implementations are slightly slower; interestingly, the performance difference between global and constant memory is minimal, likely due to effective caching mechanisms, but shared memory consistently outperforms them, demonstrating its ability to reduce memory latency and maximize parallel throughput.

- **Speedup trends:** The best observed parallel speedups, calculated as sequential time divided by the fastest parallel time, range from roughly 3.9x for $3 \times 3$ kernel on $512 \times 512$ images up to 8.7x for the $5 \times 5$ kernel on $1024 \times 1024$ images. This indicates that larger workloads and more compute-intensive filters benefit more from GPU acceleration.
  The speedup does not increase indefinitely; it is constrained by algorithmic overhead and memory bandwidth, especially for smaller images where the parallel execution cannot fully hide these costs.

- **Algorithmic vs hardware limits:** The sequential execution time grows rapidly with both kernel size and image size, confirming the quadratic complexity of the convolution algorithm.
  Parallelization on the CPU provides moderate improvement, but true performance gains require GPU offloading. CUDA with shared memory consistently dominates, highlighting that hardware-aware optimization is crucial to achieve high performance.

- **Critical insight:** For small images, the overhead of parallel execution (both CPU and GPU) reduces the relative benefit.
  Conversely, as the image size increases, GPU acceleration shows clear advantages, making OpenMP a suboptimal choice for large-scale image filtering. Therefore, while CPU parallelism can be useful for medium workloads, GPU memory optimization and massive parallelism are essential for high-resolution or computationally heavy filtering tasks.

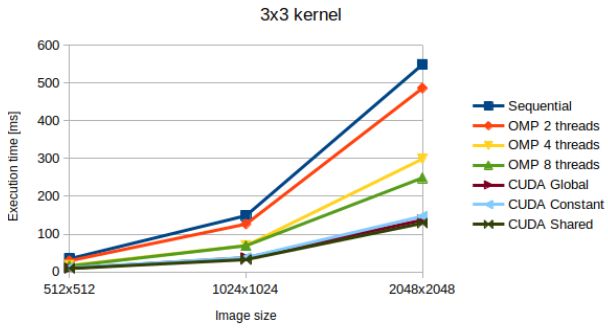



Figure 3: Execution times - 3x3 kernel.

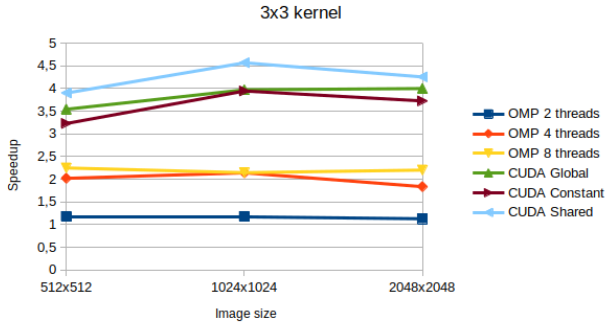



Figure 4: Speedups- 3x3 kernel.

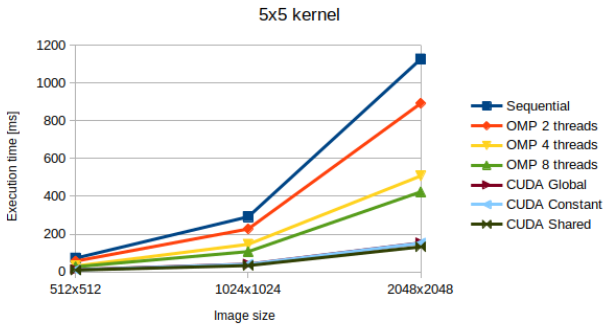



Figure 5: Execution times - 5x5 kernel.

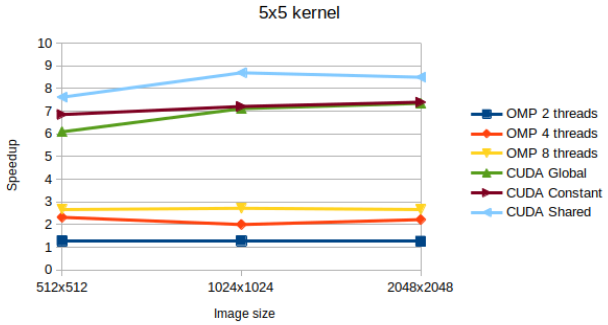



Figure 6: Speedups - 5x5 kernel.

The following tables show the best execution times and the relative speedups for the gaussian filter with various kernel size. CUDA shared memory was chosen as the parallel variant because achieves the best results.

| Img size | Sequential | Parallel | Best speedup |
|---|---|---|---|
| 512x512 | 34.78ms | 8.92ms | 3.90x |
| 1024x1024 | 148.97ms | 32.58ms | 4.57x |
| 2048x2048 | 549.19ms | 128.95ms | 4.26x |

Table 7: Gaussian filter with 3x3 kernel.

| Img size | Sequential | Parallel | Best speedup |
|---|---|---|---|
| 512x512 | 72.90ms | 9.57ms | 7.62x |
| 1024x1024 | 291.84ms | 33.54ms | 8.70x |
| 2048x2048 | 1127.53ms | 132.64ms | 8.50x |

Table 8: Gaussian filter with 5x5 kernel.

# 4 Conclusions

In this work, a Kernel Image Processing (KIP) application was analyzed with the objective of assessing its suitability for different execution paradigms. The local and data-parallel nature of convolution-based image filtering makes KIP a representative and relevant case study for evaluating both CPU and GPU computing models.
The implemented solutions demonstrate how the same processing workflow can be expressed using a sequential approach, shared-memory parallelism with OpenMP, and GPU acceleration through CUDA, with limited changes to the overall application structure with the exception of CUDA - shared memory. This unified design enables a consistent testing methodology and facilitates a fair comparison among the different implementations.
Overall, this work highlights the suitability of Kernel Image Processing as a benchmark problem for exploring parallel and heterogeneous computing techniques, providing a solid foundation for further performance analysis and optimization.

# References

[1] Wikipedia, *Kernel (image processing)*. https://en.wikipedia.org/wiki/Kernel_(image_processing)

[2] EITCA Academy, *What is the mathematical formula of the convolution operation on a 2D image?* https://eitca.org/artificial-intelligence/eitc-ai-adl-advanced-deep-learning/advanced-computer-vision/convolutional-neural-networks-for-image-recognition/what-is-the-mathematical-formula-of-the-convolution-operation-on-a-2d-image/

[3] CV Explained, *Kernels*. https://cvexplained.wordpress.com/2020/04/30/kernels/

[4] CUDA Documentation, *Get started with CUDA*.
https://developer.nvidia.com/cuda

[5] OpenMP Documentation, *OpenMP reference guides*.
https://www.openmp.org/resources/refguides/

[6] Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing*, 3rd Edition, Prentice Hall, 2008.

[7] S. Umbaugh, *Computer Vision and Image Processing: A Practical Approach Using CVIPtools*, Prentice Hall, 1998.

[8] Shared memory:
https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

[9] CUDA Programming Guide:
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory

[10] Victor Podlozhnyuk, *Image Convolution with CUDA*, July 2012

[11] NVIDIA Corporation, *CUDA SHARED MEMORY*

[12] William K. Pratt, *Digital Image Processing*, Second Edition, 1991, Sun Microsystems, California