Kernel Image Processing in C++/CUDA

Parallel Computing-2024/2025 Course Project

Leonardo Biondi
leonardo.biondi@edu.unifi.it

Abstract

In this paper we present a comparative analysis of three different implementations of kernel image processing: a sequential version in C++, a parallel version using OpenMP for multi-core CPUs, and a highly parallel version using CUDA for GPUs. We compare the performance and efficiency of these implementations on different kernel filters.

1. Introduction

Image processing through kernel filters represents one of the fundamental operations in the field of computer vision and digital image processing. This technique, known as Kernel Image Processing, involves applying a convolution matrix (kernel) to the image to achieve various effects such as blurring, sharpening or edge detection.

In this paper, we will focus on the implementation and analysis of different kernel image processing techniques, with a focus on the performance obtained through three distinct approaches:

- A sequential implementation in C++, used as a baseline for performance comparisons
- A parallel version that leverages OpenMP for parallelism on multi-core CPUs
- A highly parallel implementation using CUDA to harness the computational power of GPUs

The choice of these three approaches is motivated by the inherently parallelizable nature of kernel processing, where each pixel of the output image can be computed independently of the others. This feature makes the algorithm particularly suitable for both thread-level parallelism on CPU (OpenMP) and massively parallel processing on GPU (CUDA).

Major contributions of this work include:

- Implementation of different types of kernel filters such as Gaussian blur, sharpening, edge detection, laplacian and difference of gaussian
- A detailed analysis of the performance of the three implementations under varying image and kernel sizes
- A comparison of the efficiency of different memory optimization strategies in CUDA, including the use of shared and constant memory
- A quantitative assessment of the trade-off between implementation complexity and performance gain

The rest of the paper is organized as follows: Section 2 provides the necessary theoretical background on kernel image processing and implemented filters. Section 3 describes in detail the three implemented implementations. Section 4 presents the experimental results and performance analysis. Finally, Section 5 summarizes the conclusions of the work and discusses possible future developments.

2. Theoretical Background

2.1. Kernel Image Processing

Kernel Image Processing is a fundamental technique in digital image processing that is based on the mathematical operation of convolution. In this process, a small matrix, called a kernel or convolution mask, is systematically applied to the input image to produce a new image.

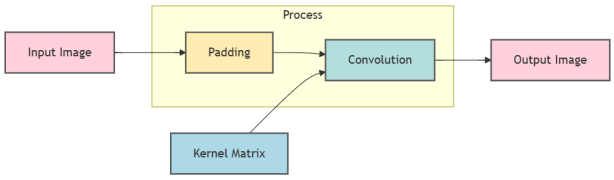

Figure 1. Kernel convolution process: the input image is first subjected to padding, then convolution with the kernel matrix is applied to produce the output image

Kernel image processing is based on the 2D discrete convolution operation. Given an input image I and a kernel

matrix K of dimension $d \times d$ (with d odd), the value of each pixel of the output image P at coordinates (x, y) is calculated as:

$$P(x,y) = \sum_{dx=-r}^{r} \sum_{dy=-r}^{r} K(dx,dy) I(x+dx, y+dy) \quad (1)$$

where:

- $P(x, y)$ is the output pixel at coordinates (x, y)
- $K(i, j)$ is the kernel value at position (i, j)
- $I(x, y)$ is the input image pixel
- r is the radius of the kernel, with total size $(2r + 1) \times (2r + 1)$

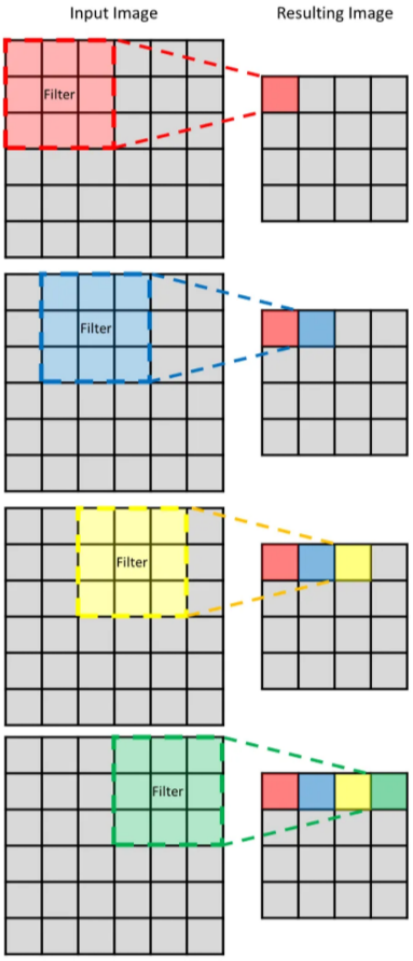

Figure 2. Sequential convolution process: the kernel is applied progressively to the input image (left) to produce the corresponding pixels in the resulting image (right). The different colors show the successive positions of the kernel and the corresponding output pixels generated.

To preserve the average brightness of the image, the kernel must be normalized. Normalization consists of dividing all kernel elements by their total sum:

$$K_{normalized}(i,j) = \frac{K(i,j)}{\sum_{i=-r}^{r} \sum_{j=-r}^{r} K(i,j)} \quad (2)$$

For convenience of representation, we often keep the coefficients un-normalized and multiply the entire matrix by the inverse of the sum of the elements. For example, for a 5x5 Gaussian kernel, we will have:

$$K_5 = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \quad (3)$$

- The highest value (36/256) in the center gives more weight to the current pixel
- Values decrease radially to create uniform blurring
- The sum of all coefficients is 1 to preserve brightness

A crucial aspect in the implementation of kernel image processing is the management of image edges. In our work, we implemented the "extend" padding method, where pixels at the edges are extended to provide the values needed for convolution. This approach can be expressed as:

$$I_{padded}(x,y) = \begin{cases} I(0,y) & \text{se } x < 0 \\ I(width-1,y) & \text{se } x \geq width \\ I(x,0) & \text{se } y < 0 \\ I(x,height-1) & \text{se } y \geq height \\ I(x,y) & \text{altrimenti} \end{cases} \quad (4)$$

The padding equation can be interpreted as follows:

- $I(0, y)$ when $x < 0$:
  - Applies when attempting to access a negative x position (left of the image)
  - Uses the value of the first pixel of row y (left edge)
- $I(width - 1, y)$ when $x \geq width$:
  - Applies when attempting to access beyond the width of the image (right)
  - Uses the value of the last pixel of row y (right edge)
- $I(x, 0)$ when $y < 0$:
  - Applies when trying to access a negative y position (above the image)

- Uses the value of the first pixel of the x column (top edge)

- $I(x, height - 1)$ when $y \geq height$:
  - Applies when attempting to access beyond the height of the image (below)
  - Uses the value of the last pixel of the x column (bottom border)

- $I(x, y)$ otherwise:
  - Applies for all positions within the image
  - Uses the normal value of the pixel

2.2. Filters Implemented

In this work we have implemented several types of kernel filters, each with specific characteristics and applications:

2.2.1 Gaussian Blur

The Gaussian blur filter is based on the two-dimensional Gaussian function:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5}$$

dove σ controlla il grado di sfocatura. Nel nostro caso, il kernel viene discretizzato in una matrice di dimensione specificata, con coefficienti normalizzati per preservare la luminosità dell'immagine.

2.2.2 Sharpening Filter

The implemented sharpening filter uses a 3x3 kernel to increase the local contrast of the image:

$$K_{sharp} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{6}$$

2.2.3 Edge Detection

Edge detection filter highlights abrupt changes in intensity in the image:

$$K_{edge} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{7}$$

2.2.4 Laplacian Filter

The Laplacian operator is used to detect areas of rapid change in intensity:

$$K_{laplace} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{8}$$

2.2.5 Difference of Gaussian (DoG)

The DoG filter is implemented as a 5x5 matrix that approximates the difference of two Gaussians:

$$K_{dog} = \begin{bmatrix} 0 & -1 & -1 & -1 & 0 \\ -1 & -2 & -2 & -2 & -1 \\ -1 & -2 & 16 & -2 & -1 \\ -1 & -2 & -2 & -2 & -1 \\ 0 & -1 & -1 & -1 & 0 \end{bmatrix} \tag{9}$$

Figure 3. Comparison between the original image (Lena) and the results of applying the different implemented filters: Gaussian blur, Sharpening, Edge Detection, Laplacian and Difference of Gaussian (DoG).

3. Implementation

In this section, we present three different implementations of kernel image processing, each targeting different computational paradigms: a sequential CPU implementation, a

parallel CPU implementation using OpenMP, and a GPU-accelerated version using CUDA.

3.1. Common Implementation Aspects

All implementations share a common foundation in terms of data structures and processing pipeline:

Image Processing Pipeline
The pipeline consists of three main stages:

- Image acquisition and preprocessing: converting input images to grayscale format and preparing data structures
- Kernel convolution: applying the filter coefficients to process the image
- Post-processing: normalizing results and preparing output

Data Structures
Images are represented as linear arrays of floating-point values, optimized for memory access patterns:

- Pixel intensities are normalized to the range [0, 255]
- Row-major ordering is used for efficient memory access
- Edge padding is implemented to handle boundary conditions

Kernel Management
The convolution kernels are designed with specific characteristics:

- Square matrices of odd dimensions (3×3, 5×5, 7×7)
- Normalized coefficients to maintain image intensity balance
- Optimized storage format for each implementation's memory access patterns

3.2. Sequential Implementation

The sequential implementation performs a convolution operation by first creating a padded version of the input image to handle border conditions effectively. The core processing adopts a four-nested loop structure: the two outer loops iterate over each pixel of the output image, while the two inner loops handle the convolution operation by applying the kernel coefficients. For each pixel, the algorithm computes a weighted sum using the kernel values and the corresponding input image pixels from the padded image. The result is then clamped between 0 and 255 to maintain valid pixel intensity values. The implementation uses a straightforward row-major memory access pattern and processes each pixel independently, storing results in a separate output buffer to avoid interference with the input data.

3.3. OpenMP Parallel Implementation

The OpenMP implementation enhances the sequential version by introducing parallel processing capabilities while maintaining the same core convolution operation. The implementation first prepares a padded image and sets the desired number of threads through `omp_set_num_threads()`. The parallelization strategy employs OpenMP's parallel for directive with two key optimizations: a `collapse(2)` clause that combines the two outer loops to increase the parallel workload granularity, and a `schedule(dynamic)` clause for balanced work distribution. Each thread independently processes its assigned pixels, computing the convolution sum and applying the intensity clamping operations. The implementation maintains thread safety naturally as each thread writes to separate portions of the output buffer, eliminating the need for explicit synchronization mechanisms. The memory access pattern remains similar to the sequential version, but with multiple threads concurrently processing different regions of the image.
Key parallel processing features include:

- Configurable thread count for performance tuning
- Dynamic work scheduling for load balancing
- Automatic parallel execution of the two outer loops
- Thread-local sum computation without shared variables

3.4. CUDA Implementation

The CUDA implementation exploits GPU parallelism through three different memory optimization approaches, each implemented as a separate kernel. The implementation defines fixed block dimensions (BLOCK_DIM_X = 16, BLOCK_DIM_Y = 16) and supports kernel sizes up to 25×25 (MAX_KERNEL_SIZE = 25).

Common Implementation Features

- Thread blocks organized in a 16×16 pattern
- Grid dimensions calculated dynamically based on image size
- Padded image handling for border conditions
- Result clamping between 0 and 255 using fmaxf and fminf

Memory Management Strategies
The implementation provides three distinct approaches to memory utilization:

- *Global Memory Version:* The baseline implementation allocates and manages both image data and kernel coefficients in global memory using cudaMalloc and cudaMemcpy. Each thread computes its pixel coordinates using blockIdx and threadIdx, accessing global memory directly for both input image and kernel coefficient data.
- *Constant Memory Version:* This version optimizes kernel coefficient access by storing the convolution kernel in constant memory (__constant__ memory space) using cudaMemcpyToSymbol. While the image data remains in global memory, the constant caching of kernel coefficients improves access patterns when threads simultaneously read the same kernel values.
- *Shared Memory Version:* The most sophisticated implementation uses shared memory to create a cached tile of the input image within each thread block. It implements a two-phase approach: first, threads cooperatively load a block of image data (including padding regions) into shared memory arrays, synchronize using __syncthreads(), and then perform the convolution computation using the cached data. This implementation requires careful management of boundary conditions and explicit thread synchronization but significantly reduces global memory access overhead.

4. Results and Performance Analysis

4.1. Test Environment

All tests were conducted on a system with the following specifications:

- CPU: Intel i7-7700K
- GPU: NVIDIA GeForce RTX 3060
- RAM: 16GB
- Operating System: Windows 10
- CUDA Version: 12.7

4.2. Testing Setup

We conducted a comprehensive testing campaign evaluating three parallel approaches against a sequential baseline implementation across six image resolutions (512×512, 768×768, 1024×1024, 1536×1536, 2048×2048, and 8192×8192 pixels) and three kernel sizes (3×3, 5×5, and 7×7). Our testing framework included a sequential C++ implementation as baseline, three CUDA implementations with different memory architectures (global, constant, and shared memory), and OpenMP implementations with 2, 4, and 8 threads. For each configuration, we measured total execution time (including data transfer for CUDA implementations), pure computation time, and data transfer overhead between CPU and GPU memory. This comprehensive setup resulted in 144 distinct test configurations, providing detailed insights into both computational performance and memory transfer characteristics. Speedup calculations were performed relative to the sequential version, considering both total execution time and pure computation time for a thorough performance analysis.

4.3. Sequential vs OpenMP Performance

The comparison between sequential and OpenMP implementations reveals the benefits and limitations of CPU-based parallelization for image processing tasks. The sequential implementation serves as our baseline, while the OpenMP version exploits multi-threading with 2, 4, and 8 threads. Tables 1, 2, and 3 show the execution times and speedups achieved with different configurations. The sequential implementation shows expected behavior with execution time increasing quadratically with image resolution. Larger kernels demonstrate a more pronounced impact on processing time, with the 7x7 kernel taking approximately 4-5 times longer than the 3x3 kernel. For the largest image (8192x8192), processing time ranges from 6.7 seconds (3x3) to 27.4 seconds (7x7), highlighting the computational intensity of the operation.

Table 1. OpenMP performance with 3×3 kernel (execution time in ms)

Res	Seq	2 thr	4 thr	8 thr
512^2	23.86	23.42	13.77	11.91
768^2	62.72	33.83	23.55	22.26
1024^2	100.55	63.01	47.85	41.57
1536^2	220.29	160.92	124.89	130.84
2048^2	389.71	282.26	195.12	166.66
8192^2	6705.15	4254.24	2828.40	2444.93

Table 2. OpenMP performance with 5×5 kernel (execution time in ms)

Res	Seq	2 thr	4 thr	8 thr
512^2	74.79	37.36	23.48	19.47
768^2	146.20	79.83	54.23	45.35
1024^24	247.30	146.22	95.63	76.67
1536^2	755.76	362.77	257.94	243.24
2048^2	953.18	646.86	418.44	321.25
8192^2	15559.22	9681.22	6726.44	4862.87

Table 3. OpenMP performance with 7×7 kernel (execution time in ms)

Res	Seq	2 thr	4 thr	8 thr
512²	121.63	68.20	42.28	35.84
768²	266.40	148.90	90.18	80.77
1024²	437.52	266.93	180.48	137.32
1536²	1096.63	702.79	424.44	347.62
2048²	1720.53	1124.78	702.96	635.99
8192²	27483.04	18210.98	11048.12	8571.05

Image Resolution Impact:
The speedup generally improves with larger image resolutions, particularly evident in the 8192x8192 case, where the parallel implementation achieves its highest efficiency. This is expected as larger images provide more opportunities for parallel processing and better amortize the overhead of thread management. However, for small images (512x512), the overhead of thread management can sometimes outweigh the benefits of parallelization.

Table 4. OpenMP speedup with 3×3 kernel

Res	2 thr	4 thr	8 thr
512²	1.02	1.73	2.00
768²	1.85	2.66	2.82
1024²	1.60	2.10	2.42
1536²	1.37	1.76	1.68
2048²	1.38	2.00	2.34
8192²	1.58	2.37	2.74

Table 5. OpenMP speedup with 5×5 kernel

Res	2 thr	4 thr	8 thr
512²	2.00	3.19	3.84
768²	1.83	2.70	3.22
1024²	1.69	2.59	3.23
1536²	2.08	2.93	3.11
2048²	1.47	2.28	2.97
8192²	1.61	2.31	3.20

Scaling with Thread Count:
The results show consistent performance improvements as the number of threads increases from 2 to 8. The best speedup is achieved with 8 threads, reaching up to 3.84x for 5x5 kernels and 3.39x for 7x7 kernels. However, the scaling is not linear due to memory bandwidth limitations and thread management overhead. We observe diminishing returns when moving from 4 to 8 threads, suggesting memory bandwidth saturation.

Table 6. OpenMP speedup with 7×7 kernel

Res	2 thr	4 thr	8 thr
512²	1.78	2.88	3.39
768²	1.79	2.95	3.30
1024²	1.64	2.42	3.19
1536²	1.56	2.58	3.16
2048²	1.53	2.45	2.71
8192²	1.51	2.49	3.21

Kernel Size Effect:
Larger kernel sizes (5x5 and 7x7) show better speedup compared to the 3x3 kernel. This is because larger kernels involve more computational work per pixel, improving the computation-to-memory-access ratio in the parallel implementation. The 5x5 kernel shows the best balance between computational intensity and memory access patterns, while the 7x7 kernel, despite being more computationally intensive, starts to show some memory access pattern inefficiencies.

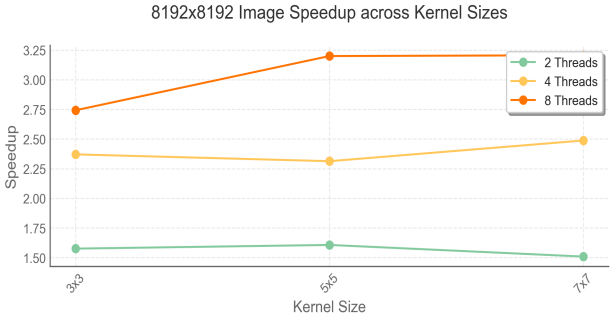

Figure 4. 8192x8192 Image Speedup across Kernel Sizes

Efficiency Analysis:
The parallel efficiency (speedup/number of threads) ranges from 25% to 48% with 8 threads, indicating some overhead in the parallelization. This is typical for memory-bound operations like image processing. Cache utilization becomes increasingly important with larger kernels, and thread synchronization overhead increases with both image size and thread count.

4.4. Analysis of CUDA Implementations

The performance evaluation focused on three key aspects: comparison between different memory implementations,

Figure 5. 5x5 Kernel Processing Time across Image Resolutions (left) - 8192x8192 Image Processing Time across Kernel Sizes (right)

block size configuration, and the impact of memory transfer overhead on the overall performance.

Memory Implementation Comparison

The three kernel implementations (global, constant, and shared memory) were evaluated using various image resolutions (from 512×512 to 8192×8192) and kernel sizes (3×3, 5×5, and 7×7). For the 7×7 kernel on an 8192×8192 image, examining pure computation times (excluding data transfer):

- Global Memory: 151.854 ms
- Constant Memory: 69.371 ms
- Shared Memory: 110.776 ms

The constant memory implementation shows the best computational performance, being approximately 54% faster than the global memory version. This efficiency can be attributed to the caching mechanism of constant memory, which is particularly effective for the filter coefficients that are accessed repeatedly by all threads.

Table 7. 7×7 Kernel Processing Time Breakdown (ms) - 8192×8192 Image

Implementation	Computation	Transfer	Total
Global	151.854	247.046	398.900
Constant	69.371	126.100	195.471
Shared	110.776	138.196	248.972

The graph in Figure 6 illustrates the scaling behavior of the three CUDA implementations across different image resolutions. Several key observations can be made:

Figure 6. CUDA Execution Time across Image Resolutions (7x7 kernel). The graph shows execution times on logarithmic scale for the three memory implementations as image resolution increases.

- At lower resolutions (512x512 to 1024x1024), the performance difference between implementations is minimal, suggesting that for smaller images the choice of memory type has less impact.
- The execution time increases logarithmically with image resolution for all implementations, demonstrating good scaling behavior.
- The constant memory implementation maintains a consistent performance advantage over global memory at higher resolutions, particularly evident in the 2048x2048 to 8192x8192 range.
- The shared memory implementation shows intermediate performance, closer to constant memory at lower resolutions but diverging slightly at higher resolutions.

Block Size Configuration

The implementation uses fixed block dimensions of 16×16 threads (BLOCK_DIM_X = 16, BLOCK_DIM_Y = 16). This configuration was chosen as it provides a good balance between several factors:

- **Warp Efficiency:** A block size of 16×16 = 256 threads ensures efficient warp utilization, as it is a multiple of the warp size (32 threads)
- **Occupancy:** This block size allows multiple blocks to be active simultaneously on each Streaming Multiprocessor (SM), promoting better hardware utilization
- **Shared Memory Usage:** For the shared memory implementation, 16×16 blocks provide a good trade-off between shared memory size requirements and data reuse, particularly when considering the maximum kernel size (MAX_KERNEL_SIZE = 25)
- **Memory Access Patterns:** The 16×16 configuration promotes coalesced memory accesses, as threads within a warp access consecutive memory locations

Data Transfer Overhead and Speedup Analysis

For a comprehensive performance analysis, we calculate two different speedup metrics (*ST=Sequential Time, CT=Computation Time, TT=Transfer Time*):

$$\text{Speedup without transfer} = \frac{\text{ST}}{\text{CUDA CT}} \tag{10}$$

$$\text{Speedup with transfer} = \frac{\text{ST}}{\text{CUDA CT} + \text{TT}} \tag{11}$$

To analyze in detail the performance of different CUDA implementations, we consider two key aspects: speedup as kernel size changes and the impact of data transfer. Figure

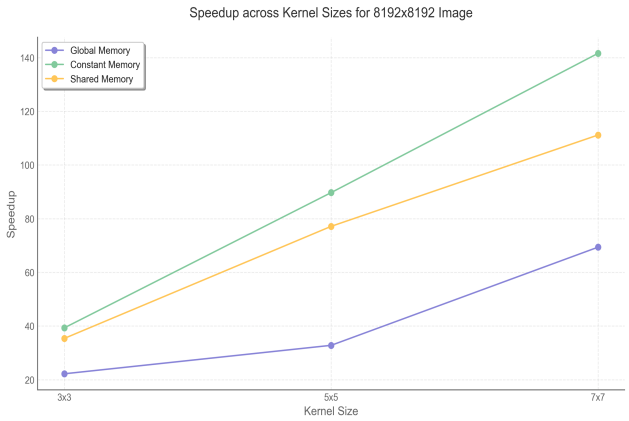

Figure 7. Speedup of CUDA implementations versus sequential execution for different kernel sizes on an 8192x8192 image. The graph shows how speedup increases with kernel size, with Constant Memory achieving the best performance.

7 shows how the speedup varies as the kernel size increases. The implementation with Constant Memory shows a significant advantage, achieving a speedup of about 140x with 7x7 kernels. This is due to the efficient caching of kernel coefficients, which are accessed frequently by all threads. Shared Memory, while showing better performance than Global Memory, does not reach the levels of Constant Memory due to the additional overhead required for data loading and synchronization. Figure 8 illustrates the impact of data

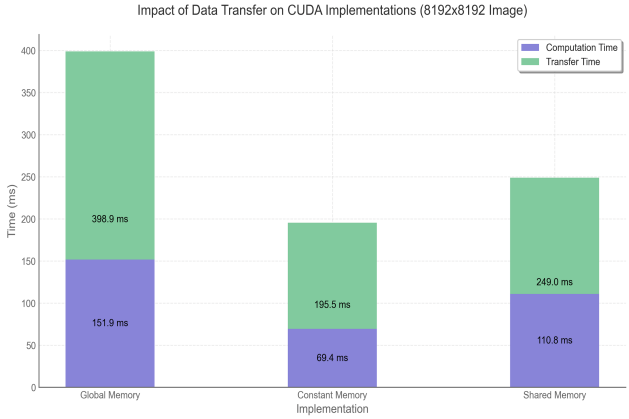

Figure 8. Breakdown of execution times for the three CUDA implementations on an 8192x8192 image with 7x7 kernel. The bars show computation time (blue) and data transfer time (green).

transfer on overall performance. For the implementation with Constant Memory, which shows the best performance, the computation time is only 69,371 ms, but data transfer takes an additional 126,100 ms, bringing the total time to 195,471 ms. This highlights how data transfer represents a significant bottleneck, constituting about 64% of the total execution time. The implementation with Global Memory shows the highest transfer overhead, while Shared Memory ranks somewhere in between, with a computation time of 110,776 ms and a transfer time of 138,196 ms.

Comparing these metrics for different kernel sizes on the 8192×8192 image reveals the significant impact of memory operations. The results for the 3×3, 5×5 and 7x7 kernels are shown in Tables 8, 9 and 10 respectively.

Table 8. Speedup Analysis - 3×3 Kernel on 8192×8192 Image

Implementation	Without Transfer	With Transfer
Global Memory	170.26	22.22
Constant Memory	352.61	39.33
Shared Memory	194.74	35.40

Table 9. Speedup Analysis - 5×5 Kernel on 8192×8192 Image

Implementation	Without Transfer	With Transfer
Global Memory	71.23	32.82
Constant Memory	360.78	89.71
Shared Memory	77.14	33.09

Table 10. Speedup Analysis - 7×7 Kernel on 8192×8192 Image

Implementation	Without Transfer	With Transfer
Global Memory	182.23	69.37
Constant Memory	398.90	141.57
Shared Memory	249.86	111.15

These results highlight several key findings:

- **Transfer Overhead:** Data transfer represents a significant portion of the total execution time, reducing the effective speedup by approximately 60-90% across all implementations. This is particularly evident in the 3×3 kernel case, where the speedup with transfer is only 22.22x for the global memory implementation, compared to 170.26x without transfer.

- **Kernel Size Impact:** The impact of data transfer is more pronounced for smaller kernel sizes (e.g., 3×3) compared to larger ones (e.g., 5×5 and 7×7). This is because smaller kernels involve less computational work per pixel, making the transfer overhead relatively more significant.

- **Memory Type Impact:** Constant memory consistently shows the best overall performance both with and without transfer times, achieving a remarkable 398.90x speedup in pure computation for the 7×7 kernel. Even with transfer overhead, it maintains a significant advantage over other implementations.

- **Scalability:** The benefits of GPU acceleration are most evident with larger images and kernel sizes, where the computational intensity justifies the data transfer overhead. For example, the 7×7 kernel shows a smaller relative reduction in speedup when including transfer times compared to the 3×3 kernel.

- **Comparison Across Kernels:** For the 7×7 kernel, the constant memory implementation achieves a speedup of 398.90x without transfer and 141.57x with transfer, demonstrating its efficiency once again. The shared memory implementation, while less performant than constant memory, still outperforms global memory, confirming the importance of memory optimization.

The analysis shows that while CUDA implementations provide substantial performance improvements over sequential processing, the efficiency is significantly impacted by data transfer overhead. This suggests that in real-world applications, strategies to minimize data transfers (such as batch processing or streaming) could further improve overall performance.

5. Conclusions

This work presents a comprehensive analysis of three different implementations of kernel image processing: sequential C++, OpenMP-based CPU parallelization, and CUDA-based GPU acceleration. Through extensive testing and analysis, we have demonstrated the effectiveness and limitations of each approach, providing valuable insights for choosing the appropriate implementation based on specific use cases and requirements. Key findings from our research include:

Implementation Performance

- The CUDA implementation with constant memory achieved the highest performance, showing up to 398.90× speedup over sequential processing for 7×7 kernels (without transfer overhead)

- OpenMP implementation demonstrated moderate speedup (up to 3.84× with 8 threads), making it a viable option for systems without GPU acceleration

- Memory transfer overhead significantly impacts CUDA performance, reducing effective speedup by 60-90%

Scaling Characteristics

- All implementations showed improved efficiency with larger image sizes

- Larger kernel sizes (5×5, 7×7) demonstrated better parallelization benefits compared to smaller ones (3×3)

- OpenMP scaling showed diminishing returns beyond 4 threads, indicating memory bandwidth limitations

Memory Optimization Impact

- Constant memory implementation in CUDA proved most effective, being approximately 54% faster than global memory

- Shared memory implementation showed intermediate performance, balancing computation efficiency with data management overhead

- Block size configuration (16×16) provided optimal balance between warp efficiency and hardware utilization

Practical Implications

- For small images or simple kernels, the overhead of GPU data transfer may outweigh the benefits of parallel processing

- OpenMP provides a good compromise between performance and implementation complexity for moderate workloads
- CUDA implementations are most beneficial for large images and computationally intensive kernels

In conclusion, this project demonstrates the significant potential of parallel processing for image processing applications while highlighting the importance of choosing the appropriate implementation based on specific use cases and hardware availability.

References

[1] Kernel Image Processing Documentation.
https://en.wikipedia.org/wiki/Kernel_(image_processing)

[2] OpenMP Programming Guide.
https://www.openmp.org/resources/refguides/

[3] CUDA Programming Guide.
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[4] Video Tutorial: Image Processing with C++
https://www.youtube.com/watch?v=twu-cRlCbG4

[5] Stack Overflow
https://stackoverflow.com