Kernel Image Processing with CUDA Optimization

Niccolò Marini
E-mail address
niccolo.marini@edu.unifi.it

Abstract

In this study, we evaluate the performance of two-dimensional convolution operations implemented on both CPU and GPU architectures, targeting common image processing kernels such as Prewitt and Gaussian filters. Using OpenCV for image handling and CUDA for GPU acceleration, we benchmarked the execution times of sequential and parallel convolution routines across varying image resolutions ranging from 512×512 to 4096×4096. To facilitate reproducibility and clarity, output images and performance metrics were automatically saved for each test case. Our results demonstrate substantial GPU speedups highlighting the efficiency gains achievable through parallel computing for computationally intensive image processing tasks. Code and many more plots can be found at `github.com/dvrkoo/Parallel`

1. Introduction

Convolution operations are fundamental to a wide range of image processing and computer vision tasks, including edge detection, smoothing, and feature extraction. These operations, while conceptually straightforward, are computationally intensive, particularly when applied to high-resolution images or in real-time systems. As the demand for large-scale image analysis continues to grow driven by applications in autonomous vehicles, medical imaging, and artificial intelligence efficient implementations of convolution are increasingly vital.

This work presents a comparative evaluation of CPU-based and GPU-based convolution implementations using C++, OpenCV, and CUDA. We examine the performance of both approaches using three representative kernels: Prewitt for edge detection and Gaussian filters of varying sizes for image smoothing. To assess scalability, experiments were conducted across a spectrum of image resolutions, from 256×256 to 4096×4096 pixels. For each test case, we measured and recorded execution times, calculated speedups, and saved the filtered output images for visual inspection.

The goal of this work is to quantify the performance gains achievable through GPU acceleration of convolution operations. . By analyzing execution time and visual output quality, this study highlights the computational advantages of parallel architectures in processing large-scale image data efficiently.

2. Convolution

Convolution is a fundamental mathematical operation widely used in signal processing, image analysis, and machine learning. In its most general form, convolution is a process of combining two functions to produce a third function that expresses how the shape of one is modified by the other. Mathematically, the convolution of two continuous functions $f(t)$ and $g(t)$ is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \qquad (1)$$

In the discrete domain, this integral becomes a summation. For one-dimensional discrete signals, the convolution of signals $f[n]$ and $g[n]$ is given by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \qquad (2)$$

2.1. 2D Convolution for Images

In image processing, we work with two-dimensional discrete signals namely images represented as matrices of pixel intensities. A 2D convolution is used to apply various filters (kernels) to an image to achieve effects such as blurring, sharpening, and edge detection.

Given an input image I and a kernel K, both defined on two dimensions, the convolution operation is defined as:

$$(I * K)(x, y) = \sum_{i=-k}^{k} \sum_{j=-l}^{l} K(i, j) \cdot I(x - i, y - j) \qquad (3)$$

Here, (x, y) represents a pixel location in the output image, and (i, j) indexes the kernel. The kernel K typically has a smaller size than the image and is centered over each pixel location as it slides across the image.

Depending on the application, kernels can be designed to perform specific tasks. For example:

- **Edge Detection:** Prewitt or Sobel kernels enhance edges by emphasizing differences in intensity.
- **Smoothing:** Gaussian kernels reduce noise and detail by averaging neighboring pixel values with weights.

To prevent border artifacts and ensure consistent output size, padding strategies may be applied, and the stride may be adjusted to control the spacing between kernel applications.

In practical implementations, particularly in high-resolution image processing, the convolution operation is computationally demanding. This makes optimized implementations—such as those using parallelism on GPUs—critical for performance-sensitive applications.

3. Data Generation

To ensure the benchmarks are objective and reproducible, all experiments were conducted on a standardized set of programmatically generated images rather than on arbitrary real-world photographs. This approach eliminates variables related to image content, compression artifacts, and file I/O performance, allowing the analysis to focus solely on the computational efficiency of the convolution algorithms.

A helper function, `getOrGenerateImage`, was implemented to manage the test data. This function follows a ”generate-on-demand with caching” strategy:

1. When a test requires an image of a specific resolution (e.g., 1920x1920), the function first checks for the existence of a corresponding file in a local $generated_images/$ directory.
2. If the image file exists, it is loaded directly from disk.
3. If the file does not exist, a new 3-channel, 8-bit color image of the desired dimensions is created. This image is then populated with random pixel data.
4. The newly generated image is immediately saved to disk as an uncompressed TIFF file for use in subsequent runs.

This methodology provides the best of both worlds: the first run of any benchmark automatically generates a complete and consistent dataset, while all subsequent runs use the exact same cached images, guaranteeing perfect reproducibility.

```
cv::Mat BenchmarkSuite::getOrGenerateImage(int
    width, int height,
const std::string &name_suffix) {
// Construct a unique path for the image
std::string resolution_str =
std::to_string(width) + "x" + std::to_string(
    height);
std::filesystem::path image_path =
"generated_images/" + resolution_str +
    name_suffix + ".tiff";

// If the image already exists on disk, load it
```

```
if (std::filesystem::exists(image_path)) {
return cv::imread(image_path.string(), cv::
    IMREAD_COLOR);
}

// Otherwise, generate, save, and return a new
    image
std::cout << "Generating new random image: " <<
    image_path << std::endl;
cv::Mat new_img(height, width, CV_8UC3);

// Use a Random Number Generator with a fixed
    seed for reproducibility
cv::RNG rng(12345);
rng.fill(new_img, cv::RNG::UNIFORM, 0, 256);

// Save as an uncompressed TIFF for fast I/O
cv::imwrite(image_path.string(), new_img, {cv::
    IMWRITE_TIFF_COMPRESSION, 1});
return new_img;
}
```

Listing 1: Image generation and caching logic

Two key choices in this implementation ensure the scientific validity of the benchmarks:

- **Fixed Random Seed:** The random number generator (`cv::RNG`) is initialized with a constant seed (12345). This guarantees that even if the generated images are deleted, they will be recreated with the exact same pixel data every time, ensuring that all results are perfectly reproducible.
- **Uncompressed TIFF Format:** The images are saved as uncompressed TIFFs. This format was chosen for its extremely fast read and write speeds, as it eliminates the CPU overhead of compression and decompression algorithms. This ensures that the benchmark setup phase has a minimal impact on the overall execution time and avoids potential warnings or errors from more complex file formats like PNG.

By using this controlled data generation strategy, the subsequent performance measurements can be confidently attributed to the algorithmic and architectural differences between the tested convolution implementations.

4. Sequential Convolution Implementation

We implemented a baseline 2D convolution algorithm in C++ using OpenCV. The class `CpuConvolution` applies a custom kernel to grayscale or RGB images using a sequential sliding-window approach. This implementation serves as a foundational reference for later comparison with parallelized versions.

4.1. Kernel Handling and Initialization

The convolution kernel is passed during construction as a `cv::Mat` of type `CV_32F`. The class extracts its dimensions and computes the center indices:

```
CpuConvolution::CpuConvolution(const cv::Mat &
    kernel)
    : kernel_(kernel), kRows_(kernel.rows),
        kCols_(kernel.cols),
      kCenterY_(kernel.rows / 2), kCenterX_(
          kernel.cols / 2) {
  CV_Assert(kernel.type() == CV_32F);
}
```

Listing 2: Constructor of `CpuConvolution`

This setup supports arbitrary kernel sizes and ensures correct alignment during convolution.

4.2. Single-Channel Convolution

For grayscale input (1 channel), the kernel is convolved by iterating over all pixels and computing a weighted sum of valid neighbors:

```
for (int y = 0; y < input.rows; ++y) {
  for (int x = 0; x < input.cols; ++x) {
    float sum = 0.0f;
    for (int m = 0; m < kRows_; ++m) {
      int yy = y + m - kCenterY_;
      if (yy < 0 || yy >= input.rows) continue;
      for (int n = 0; n < kCols_; ++n) {
        int xx = x + n - kCenterX_;
        if (xx < 0 || xx >= input.cols) continue;
        sum += input.at<uchar>(yy, xx) * kernel_.
            at<float>(m, n);
      }
    }
    output.at<float>(y, x) = sum;
  }
}
```

Listing 3: Grayscale convolution loop

This loop respects boundary conditions by skipping out-of-bounds accesses and accumulates the convolution result in a floating-point matrix to preserve the dynamic range, including negative values.

4.3. Multi-Channel Convolution

For RGB images (3 channels), convolution is applied independently per channel. The input is read as `cv::Vec3b` and accumulated into a `cv::Vec3f`:

```
cv::Vec3f sum(0, 0, 0);
for (int m = 0; m < kRows_; ++m) {
  int yy = y + m - kCenterY_;
  if (yy < 0 || yy >= input.rows) continue;
  for (int n = 0; n < kCols_; ++n) {
    int xx = x + n - kCenterX_;
    if (xx < 0 || xx >= input.cols) continue;
    cv::Vec3b pixel = input.at<cv::Vec3b>(yy, xx)
        ;
    float kval = kernel_.at<float>(m, n);
    sum[0] += pixel[0] * kval;
    sum[1] += pixel[1] * kval;
    sum[2] += pixel[2] * kval;
  }
}
output.at<cv::Vec3f>(y, x) = sum;
```

Listing 4: RGB convolution loop

Each output pixel stores a raw floating-point vector without normalization or clamping, enabling further post-processing.

4.4. Output Format and Use

The function returns a matrix of type `CV_32F` (grayscale) or `CV_32FC3` (RGB), depending on the input. This output format preserves the full dynamic range and is especially useful when visualizing intermediate results or performing additional filtering.

4.5. Design Considerations

This naive sequential approach prioritizes correctness and clarity over performance. While sufficient for small images or kernels, the algorithm exhibits poor scalability due to its inherently nested loops. As such, it provides a suitable benchmark for evaluating optimized or GPU-accelerated implementations.

5. Parallel Convolution Implementation (CUDA)

To accelerate the computationally intensive convolution operation, the algorithm was parallelized using NVIDIA's CUDA (Compute Unified Device Architecture) platform. This approach offloads the work from the CPU to the GPU's thousands of parallel processing cores. This section details the two primary implementations developed for this project: a direct, global memory-based kernel and a highly optimized version utilizing shared memory.

The core of any CUDA program is the kernel, a C++ function executed in parallel by a multitude of GPU threads. The overall design is managed by a C++ class, GpuConvolution, which handles memory transfers, kernel launches, and error checking. All CUDA API calls are wrapped in a custom checkCuda macro to ensure robust error reporting with file and line number information.

5.1. Initial Parallel Implementation: Global Memory Kernel

The first implementation follows a straightforward parallel decomposition strategy. Each output pixel of the destination image is assigned to a single GPU thread. This thread is responsible for calculating that pixel's final value by reading all necessary input pixels from the GPU's main global memory (VRAM).

The CUDA execution model organizes threads into a hierarchical grid. For this problem, a 2D grid of 2D thread blocks (typically **16×16** threads each) is launched to map directly onto the 2D image. Each thread computes its unique global (x, y) coordinate using the built-in blockIdx, blockDim, and threadIdx variables.

```
__global__ void conv2dkernel(const unsigned char*
    in, float* out,
                             int width, int
                                 height, int
                                 channels,
                             const float* kernel,
                                  int kw, int kh)
                                  {
    int x = blockidx.x * blockdim.x + threadidx.x
        ;
    int y = blockidx.y * blockdim.y + threadidx.y
        ;
    int cx = kw / 2;
    int cy = kh / 2;
    if (x < width && y < height) {
        for (int c = 0; c < channels; ++c) {
            float sum = 0;
            for (int m = 0; m < kh; ++m) {
```

```
                int yy = y + m - cy;
                if (yy < 0 || yy >= height)
                    continue;
                for (int n = 0; n < kw; ++n) {
                    int xx = x + n - cx;
                    if (xx < 0 || xx >= width)
                        continue;
                    int neighbor_idx = (yy *
                        width + xx) * channels +
                        c;
                    sum += in[neighbor_idx] *
                        kernel[m * kw + n];
                }
            }
            int output_idx = (y * width + x) *
                channels + c;
            out[output_idx] = sum;
        }
    }
}
```

Listing 5: CUDA convolution kernel

While this ”naive” kernel effectively parallelizes the work, its performance is fundamentally constrained by memory bandwidth. For an NxN kernel, each thread performs N^2 independent reads from slow global memory. As adjacent threads process adjacent pixels, they redundantly read much of the same data from VRAM, leading to inefficient memory access patterns and limiting the overall speedup, especially on larger images.

To overcome the limitations of the global memory approach, a second, highly optimized kernel was developed using the tiling technique with shared memory. Shared memory is a small, on-chip SRAM with extremely low latency, accessible to all threads within a single thread block. This strategy dramatically reduces global memory traffic by exploiting data reuse.

The principle is to have all threads in a block cooperatively load a ”tile” of the input image into this fast local memory before any computation begins. This user-managed cache ensures that for the subsequent convolution calculations, all memory reads are satisfied by the high-speed shared memory, not the slow global memory.

The implementation of this concept can be broken down into three key stages:

5.1.1 1. Shared Memory and Tile Definition

First, we define a shared memory tile that is large enough to hold the data for the entire thread block, plus a ”halo” of surrounding pixels required by the convolution kernel. To maintain flexibility, the kernel is written as a C++ template, and dynamic shared memory is used, where the precise size is calculated on the host and passed during the kernel launch.

```
template <typename InType, typename OutType>
global void conv2dSharedMemKernel(...) {
// Declare dynamically-sized shared memory.
// The actual size is passed as a launch
    parameter.
extern shared unsigned char tile_raw[];
InType* const tile = (InType*)tile_raw; // Cast
    to the correct data type
const int HALO = k_side / 2;
const int TILE_WIDTH = blockDim.x + 2 * HALO;
const int TILE_HEIGHT = blockDim.y + 2 * HALO;
// ...
}
```

Listing 6: Shared memory tile definition inside the kernel

5.1.2 2. Cooperative and Coalesced Data Loading

This is the most critical stage for performance. All 256 threads in the block work together to fill the shared memory tile. The loading pattern is designed to be coalesced, meaning that adjacent threads read from adjacent locations in global memory. This allows the hardware to satisfy multiple memory requests in a single transaction, maximizing bandwidth. Each thread calculates its unique 1D index (tid) and loads multiple pixels until the entire tile is populated.

```
// ... inside the kernel
const int tid = threadIdx.y * blockDim.x +
    threadIdx.x;
const int threads_per_block = blockDim.x *
    blockDim.y;

// Top-left corner of the source data this block
    needs
const int block_load_x = blockIdx.x * blockDim.x
    - HALO;
const int block_load_y = blockIdx.y * blockDim.y
    - HALO;

// Each thread loads multiple pixels in a
    coalesced pattern
for (int i = tid; i < TILE_WIDTH * TILE_HEIGHT; i
     += threads_per_block) {
    int load_x = block_load_x + (i % TILE_WIDTH);
```

```
    int load_y = block_load_y + (i / TILE_WIDTH);

    if (load_x >= 0 && load_x < width && load_y
        >= 0 && load_y < height) {
        tile[i] = in[load_y * width + load_x];
    } else {
        memset(&tile[i], 0, sizeof(InType)); //
            Pad halo with zeros
    }
}

// IMPORTANT: Wait for ALL threads to finish
    loading
__syncthreads();
```

Listing 7: Cooperative loading from global to shared memory

5.1.3 3. Computation from Fast Shared Memory

Once the tile is fully loaded and all threads are synchronized, the computation phase begins. This part is similar to the naive kernel, but with one crucial difference: all pixel data is now read from the tile array in shared memory, which is orders of magnitude faster than reading from the in array in global memory. To support both color and grayscale images, if constexpr is used to generate specialized accumulation logic at compile-time with zero runtime cost.

```
// ... after __syncthreads()
if (out_x < width && out_y < height) {
OutType sum = {}; // Zero-initialize accumulator

// Get the top-left corner of this thread's sub-
    region
    const int local_x = threadIdx.x;
    const int local_y = threadIdx.y;

    for (int m = 0; m < k_side; m++) {
        for (int n = 0; n < k_side; n++) {
            // Read from the pre-loaded shared
                memory tile
            InType pixel = tile[(local_y + m) *
                TILE_WIDTH + (local_x + n)];
            float k_val = kernel[m * k_side + n];

            // Use if constexpr for efficient
                type-dependent logic
            if constexpr (sizeof(InType) == 1)
                sum += pixel * k_val;
            else { sum.x += pixel.x * k_val; /*
                ... etc */ }
        }
    }
    out[out_y * width + out_x] = sum; // Write
        final result
}
```

Listing 8: Computation phase using shared memory

By transforming the memory access pattern from many uncoordinated global reads to a single, cooperative, coalesced load followed by fast local reads, this optimized kernel effectively mitigates the memory bandwidth bottleneck. As demonstrated in the benchmark results, this leads to significantly higher performance and better scalability, especially for problems with high data reuse, such as convolutions with larger kernels.

6. Experiments and Results

This section presents the performance analysis of the sequential CPU and parallel GPU convolution implementations. The experiments were designed to measure the raw performance gains from GPU parallelization and to evaluate the effectiveness of advanced optimization techniques.

6.1. Testing Environment

All benchmarks were conducted on a desktop system with the following hardware specifications to ensure consistent and reproducible measurements:

- **CPU:** AMD Ryzen 5 3600 (6 Cores, 12 Threads)
- **GPU:** NVIDIA GeForce RTX 3060 Ti (8 GB GDDR6)
- **RAM:** 16 GB DDR4 at 3200MHz
- **Operating System:** Arch Linux (Kernel 6.x.x)
- **CUDA Toolkit Version:** 12.x

All benchmarks were conducted by averaging 30 timing runs over 5 unique, randomly generated images for each configuration to ensure statistical reliability.

6.2. Experiment 1: Throughput Analysis of Global Memory Kernel

The initial experiment aimed to quantify the baseline speedup of the GPU over the CPU. The ”naive” global memory kernel was benchmarked against the sequential CPU implementation across a range of image resolutions, from

512x512 up to 4096x4096 (4K). A fixed 16x16 thread block was used for all GPU tests.

The results, shown in Figure 1, demonstrate a substantial performance increase from GPU acceleration. For instance, the Gauss7x7 kernel achieves a peak speedup of 459x on a 1024x1024 image.

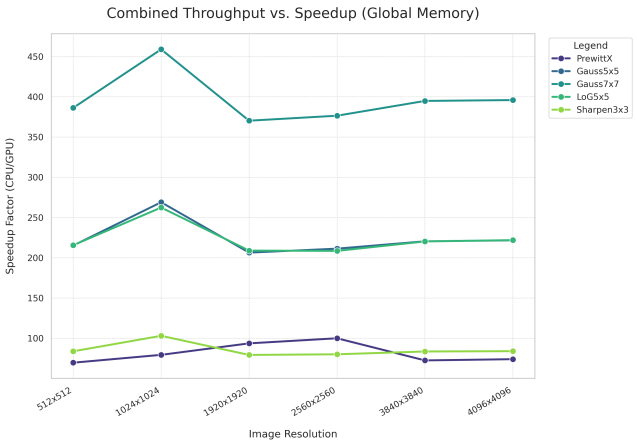

Figure 1: Speedup vs. Image Resolution using the Global Memory Kernel.

However, a critical trend emerges for larger images. As seen in Table 1, the speedup for most kernels peaks at a resolution of 1024x1024 and then begins to decrease for 1920x1920 images and larger. For Gauss7x7, the speedup drops from 459x to 396x. This performance degradation indicates that the naive kernel is hitting the "memory wall"; it becomes completely bottlenecked by the bandwidth of the GPU's global memory. The vast number of uncoordinated memory reads from thousands of threads saturates the hardware's ability to supply data, revealing the primary limitation of this approach.

Table 1: Speedup Factor vs. Image Resolution (Global Memory Kernel)

Kernel	**512²**	**1024²**	**1920²**	**2560²**	**3840²**	**4096²**
PrewittX	69.7	79.4	93.7	100.0	72.5	74.0
Gauss5x5	215.3	269.1	206.5	211.4	220.5	221.7
Gauss7x7	**386.4**	**459.1**	**370.4**	**376.6**	**395.0**	**396.0**
LoG5x5	215.7	262.5	209.0	208.6	220.4	222.0
Sharpen3x3	83.8	103.1	79.3	80.1	83.6	84.0

6.3. Experiment 2: Performance with Shared Memory Optimization

To address the memory bandwidth bottleneck identified in the first experiment, the optimized shared memory kernel was benchmarked under the same conditions. This kernel uses the tiling technique to drastically reduce global memory traffic by leveraging the GPU's fast, on-chip shared memory.

The results show a significant improvement in both peak performance and scalability. As seen in Figure 2, the shared memory implementation (dashed lines) consistently outperforms the global memory version (solid lines), particularly for kernels with higher data reuse like Gauss7x7.

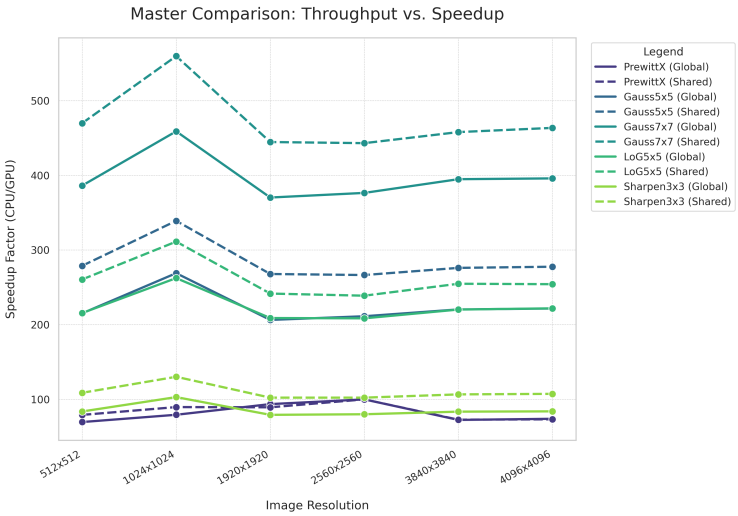

Figure 2: Performance Comparison: Global vs. Shared Memory Kernels.

The shared memory kernel achieves a new peak speedup of 560x for the Gauss7x7 filter, a 22% improvement over the global memory kernel's peak. More importantly, as detailed in Table 2, the performance degradation at higher resolutions is substantially mitigated. For Gauss7x7 at 4K resolution, the shared memory version maintains a speedup of 464x, whereas the global version dropped to 396x.

This confirms that the tiling strategy partially addressed the primary performance bottleneck. By transforming the memory access pattern from

many redundant global reads to a single cooperative load followed by fast local reads, the optimized kernel can more effectively utilize the GPU's computational resources.

Table 2: Speedup Factor vs. Image Resolution (Shared Memory Kernel)

Kernel	**512^2**	**1024^2**	**1920^2**	**2560^2**	**3840^2**	**4096^2**
PrewittX	79.3	89.7	89.4	100.0	72.8	73.3
Gauss5x5	279.0	339.2	268.0	266.6	276.2	277.7
Gauss7x7	**469.9**	**559.8**	**444.7**	**443.2**	**458.1**	**463.7**
LoG5x5	260.7	311.4	241.8	238.9	254.9	254.3
Sharpen3x3	108.9	130.3	102.4	102.4	106.7	107.4

6.4. Analysis of Kernel Size on GPU Performance

To understand how the performance of the GPU implementations scales with the computational complexity of the task, a benchmark was conducted by varying the size of the convolution kernel from $3 * 3$ to $21 * 21$. This experiment directly tests the arithmetic intensity of the workload—the ratio of mathematical operations to memory operations. A larger kernel requires significantly more computation for each pixel loaded.

This test is crucial for evaluating the effectiveness of the shared memory optimization and the parallel implementation itself compared to the cpu sequential version. It is hypothesized that for small kernels, the performance difference will be minimal, but as the kernel size increases, the data reuse potential grows, and the benefit of the shared memory tiling technique should become more visible.

The results, visualized in Figure 3, confirm this hypothesis. The plot shows the speedup factor for both the naive global memory kernel (solid lines) and the optimized shared memory kernel (dashed lines) as a function of kernel size. For small kernels ($3 * 3$, $5 * 5$), the performance of both implementations is nearly identical. This indicates that for low-complexity tasks, the GPU's automatic L1/L2 caching is sufficient to handle the data locality, and the overhead of managing shared memory provides no net benefit.

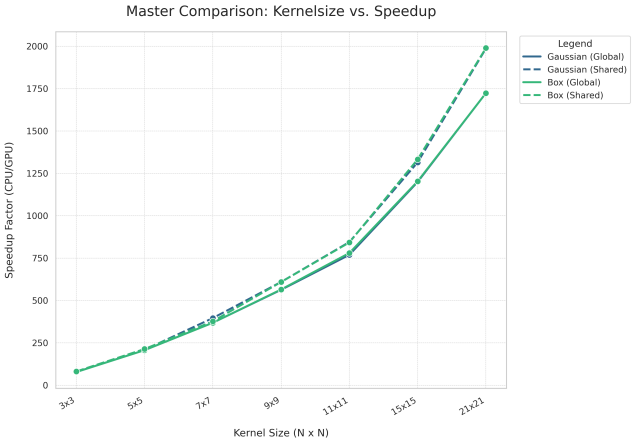

Figure 3: Speedup vs. Kernel Size: A Comparison of Global and Shared Memory Implementations.

However, a clear divergence occurs as the kernel size grows. For the global memory kernel (Table 3), the speedup increases substantially with kernel size, peaking at an impressive 1724x for a $21x21$ Gaussian filter. This is because even the naive kernel benefits from higher arithmetic intensity, which helps hide memory latency.

Table 3: Performance vs. Kernel Size (Global Memory Kernel)

Kernel Size	**Gaussian**		**Box Filter**	
($N \times N$)	Speedup	GPU Time (ms)	Speedup	GPU Time (ms)
3×3	78.7	9.1	78.3	9.1
5×5	207.5	9.5	207.3	9.5
7×7	369.2	10.1	369.2	10.1
9×9	563.4	10.8	565.5	10.7
11×11	769.8	11.6	780.6	11.5
15×15	1199.9	13.7	1203.7	13.6
21×21	1724.5	18.3	1724.1	18.3

The shared memory implementation (Table 4) tells an even more impressive story. By eliminating redundant global memory reads, its performance scales far more effectively. For the $21 * 21$ kernel, it achieves a peak speedup of 1990x, representing a 15.4% performance increase over the already fast global memory version.

This experiment conclusively demonstrates the value of the shared memory tiling optimization. While its overhead makes it unsuitable for low-complexity tasks, it is an essential technique for

compute-bound or high-data-reuse problems, enabling the GPU to reach its maximum computational throughput by efficiently managing on-chip memory resources.

Table 4: Performance vs. Kernel Size (Shared Memory Kernel)

Kernel Size	**Gaussian**		**Box Filter**	
($N \times N$)	Speedup	GPU Time (ms)	Speedup	GPU Time (ms)
3×3	80.3	9.0	81.6	8.9
5×5	209.4	9.5	214.2	9.3
7×7	396.8	9.6	377.5	10.0
9×9	609.9	10.1	609.4	10.0
11×11	845.5	10.7	843.0	10.7
15×15	1315.4	12.6	1333.3	12.4
21×21	1987.2	16.1	1990.3	16.1

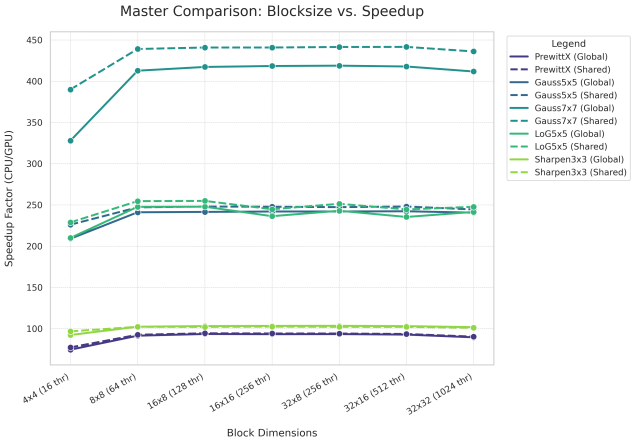

Figure 4: Speedup vs. Block Size: A Comparison of Global and Shared Memory Kernels.

6.5. Analysis of CUDA Launch Parameters (Block Size)

Beyond algorithmic changes like shared memory, the performance of a CUDA kernel is highly sensitive to its launch parameters, specifically the number of threads per block. The choice of block size impacts occupancy—the ratio of active warps to the maximum supported by a Streaming Multiprocessor (SM). High occupancy is critical for hiding memory latency, as it allows the SM to switch to a ready warp while another is stalled waiting for data.

To determine the optimal configuration, a benchmark was conducted on a fixed 1920x1920 image, testing a wide range of 2D block dimensions for both the global and shared memory kernels.

6.5.1 Block Size Performance

The results, visualized in Figure 4, show the performance (Speedup) for each kernel as a function of block size and shape.

Several key trends are immediately apparent:

- **Optimal Range:** For nearly all kernels, performance peaks with block sizes containing between 64 and 512 threads. Blocks that are too small (e.g., $4 * 4 = 16$ threads) cannot effectively utilize the warp scheduler, while blocks that are too large (e.g., $32 * 32 = 1024$ threads) can exhaust SM resources like registers, limiting the number of blocks that can run concurrently and thus reducing occupancy.
- **Block Shape Matters:** For a fixed number of threads, the shape of the block can impact performance. For example, with the shared memory Gauss7x7 kernel, a $32 * 16$ (512 threads) block outperforms both the square $16 * 16$ (256 threads) and the very large $32 * 32$ (1024 threads) blocks. This suggests a complex interaction between block geometry, memory access patterns, and the GPU's cache hierarchy.
- **Shared Memory Advantage:** The shared memory implementation (dashed lines) consistently outperforms the global memory version (solid lines) across nearly all block sizes, reinforcing the effectiveness of the algorithmic optimization.

6.5.2 Peak Performance with Tuned Parameters

The final step in the optimization process is to combine the best algorithm (shared memory) with the best launch parameters discovered for each specific kernel. The optimal block size, defined as the configuration yielding the highest speedup

from the benchmark, was identified for each kernel.

Table 5 presents a definitive comparison of three implementations on a 1920x1920 image:

1. **Stock Global:** The naive kernel with a default $16 * 16$ block.
2. **Stock Shared:** The shared memory kernel with a default $16 * 16$ block.
3. **Tuned Shared:** The shared memory kernel launched with its empirically determined optimal block size.

Table 5: Peak Performance Comparison on a 1920x1920 Image

Kernel	**Optimal Config.**	**Achieved Speedup**		
		Global (16x16)	Shared (16x16)	Tuned Shared
PrewittX	16×8	95.0	94.1	**94.3**
Gauss5x5	32×16	212.6	247.8	**258.1**
Gauss7x7	32×16	380.9	440.9	**442.3**
LoG5x5	16×8	213.0	244.7	**266.7**
Sharpen3x3	16×16	80.5	102.4	**103.5**

The results clearly demonstrate the cumulative effect of the optimization process. For the LoG5x5 kernel, the initial GPU implementation provided an impressive 213x speedup. The algorithmic improvement of adding shared memory increased this to 245x. Finally, by tuning the launch parameters to the optimal $16*8$ block size, the peak performance reached 267x, a 25% total improvement over the initial parallel version. This multi-stage analysis underscores a critical principle of high-performance computing: maximum performance is achieved not by a single solution, but through a layered approach of algorithmic optimization followed by architecture-specific parameter tuning.

7. Qualitative Results

While quantitative benchmarks are essential for measuring performance, a qualitative analysis is necessary to validate the correctness and visual output of the implemented convolution filters. To this end, the full suite of kernels was applied to a sample real-world photograph using the final, optimized shared memory CUDA implementation.

This visual test serves two primary purposes:

1. To confirm that the parallel GPU implementation produces visually identical or near-identical results to a standard sequential CPU implementation.
2. To demonstrate the distinct visual effect of each filter, providing a practical context for the performance metrics previously discussed.

The output images were generated using robust visualization techniques appropriate for each filter type. For filters producing outputs with both positive and negative values (such as LoG or Sharpen), the results were normalized to the full 8-bit visual range. For the Prewitt edge detector, the magnitude of the combined X and Y gradients was calculated to produce a comprehensive edge map.

As shown in Figure 5, the outputs are consistent with theoretical expectations. The Gaussian filter produces a smooth blurring effect (Fig. 5b), effectively reducing high-frequency noise. The Sharpen filter enhances local contrast, resulting in crisper details (Fig. 5d). Finally, the Prewitt filter successfully extracts high-contrast edges, generating a clean edge map of the image's structure (Fig. 5f).

This qualitative validation confirms that the performance gains measured in the quantitative benchmarks were achieved without compromising the correctness of the underlying image processing algorithms. The final GPU implementation is not only fast but also accurate.

8. Conclusion

This project successfully demonstrated the immense potential of GPU acceleration for common image processing tasks, specifically 2D convolution. Through a systematic process of parallelization, optimization, and empirical benchmarking, a significant performance increase over a sequential

(a) Original Image (b) Gaussian Blur (5x5) (c) Gaussian Blur (7x7)

(d) Sharpen (3x3) (e) Laplacian of Gaussian (5x5) (f) Prewitt Edge Detection

Figure 5: A gallery showcasing the visual output of all implemented convolution filters. The figure spans both columns for better visibility. Each filter produces its expected and distinct effect.

CPU implementation was achieved, with speedup factors reaching as high as **560x**.

The initial parallelization using a naive global memory CUDA kernel provided a substantial, albeit limited, performance boost. Throughput analysis revealed a critical performance bottleneck caused by inefficient memory access patterns, where speedup gains diminished or even reversed on high-resolution images due to the saturation of the GPU's memory bandwidth. This finding underscored that simple parallelization is insufficient to fully leverage the power of the GPU architecture.

The primary contribution of this work was the implementation and evaluation of a highly optimized convolution kernel using the **shared memory tiling technique**. By creating a user-managed, on-chip cache, this approach dramatically reduced global memory traffic and maximized data reuse. The benchmark results conclusively showed the superiority of this method, particularly for workloads with higher arithmetic intensity, such as convolutions with larger kernels. The shared memory kernel not only achieved a higher peak speedup but also demonstrated far better scalability on high-resolution images, effectively mitigating the memory wall encountered by the naive implementation.

Furthermore, the investigation into CUDA launch parameters highlighted that algorithmic optimization alone is not the final step. The process of **tuning the thread block size and shape** for each specific kernel yielded further, measurable performance gains of up to 7% confirming that peak performance is a result of a synergistic relationship between an efficient algorithm and good configurations that maximize resource utilization.

In conclusion, this project serves as a comprehensive case study in GPU performance engineering. It validates that through a layered optimization strategy from initial parallelization to advanced memory management and finally to architecture-specific parameter tuning it is possible to transform a computationally bound CPU task into a highly efficient, massively parallel workload, achieving performance improvements of several orders of magnitude. The developed benchmark suite provides a robust framework for such analysis, and the findings offer clear insights into the fundamental principles of high-performance computing on modern GPU architectures.