Parallelizing Morphological Image Operations: A Comparative Study of CPU and GPU Implementations

Jay Senoner
E-mail address
jay.senoner@edu.unifi.it

Abstract

This report presents a comprehensive performance comparison of three implementations for fundamental morphological image processing operations: a sequential Python baseline, a multi-core CPU parallel version using Python's 'multiprocessing' module, and a parallel GPU version implemented in CUDA. The objective is to analyze and quantify the performance gains and scalability characteristics of different parallel programming models when applied to a classic, data-parallel computer vision task. Key operations including erosion, dilation, opening, and closing are benchmarked across a wide range of image sizes and structuring element sizes. Results demonstrate that while CPU multiprocessing provides a modest and predictable speedup limited by the number of physical cores, the GPU implementation delivers orders-of-magnitude performance improvement that scales dramatically with problem size. The findings highlight the architectural advantages of GPUs for image processing and explore the trade-offs between different parallelization strategies. Code is available at github.com/jaysenoner99/PP_2.

1. Introduction

Mathematical morphology is a traditional technique in digital image processing, providing a powerful theoretical framework for analyzing and manipulating the geometrical structures within an image. Based on set theory, it probes an image with a small shape or template called a "structuring element" (SE) to extract useful information and modify the image's form. Its applications are widespread, ranging from noise reduction, edge detection, and object segmentation in computer vision to feature extraction in medical imaging and optical character recognition. The fundamental operations, erosion and dilation, form the basis for more complex operators like opening and closing.

(a) Original Grayscale Image (b) Image after Erosion

Figure 1: Visual comparison of an image before and after a morphological closing operation.

1.1. Mathematical Morphology Operations

Formally, given a grayscale image I and a structuring element SE, the two basic morphological operations are defined as follows for each pixel (x, y):

- **Erosion**: Shrinks or thins the bright regions of an image. The value of the output pixel is the minimum value of the input image in the neighborhood defined by the SE.

$$(I \ominus SE)(x, y) = \min_{(i,j) \in SE} \{I(x + i, y + j)\}$$

- **Dilation**: Expands or thickens the bright re-

gions. The value of the output pixel is the maximum value of the input image in the neighborhood.

$$(I \oplus SE)(x, y) = \max_{(i,j) \in SE} \{I(x + i, y + j)\}$$

From these, two compound operations are derived:

- **Opening**: An erosion followed by a dilation. It smooths contours, breaks narrow connections, and removes small bright spots.

$$I \circ SE = (I \ominus SE) \oplus SE$$

- **Closing**: A dilation followed by an erosion. It fuses narrow breaks, fills small holes, and smooths contours of dark regions.

$$I \bullet SE = (I \oplus SE) \ominus SE$$

The computational complexity for a naive implementation of a basic morphological operation on an image of size $W \times H$ with a structuring element of size $w \times h$ is:

$$\mathcal{O}(W \cdot H \cdot w \cdot h)$$

1.2. The Case for Parallelization

The structure of morphological operations is inherently data-parallel. The calculation for each output pixel depends only on a small neighborhood of input pixels and is completely independent of all other output pixels. This "embarrassingly parallel" nature makes it an ideal candidate for parallelization. We explore three distinct approaches:

1. **Sequential**: A baseline implementation using nested loops in Python, serving as the reference for performance measurement.
2. **CPU Parallel**: A parallel approach using Python's `multiprocessing` module. The image is divided into horizontal strips, and each strip is processed by a separate CPU core.
3. **GPU Parallel**: A parallel approach using CUDA. Each pixel of the output image is assigned to a unique GPU thread, allowing thousands of pixels to be computed simultaneously.

The goal is to analyze the trade-offs in terms of speedup, scalability, and implementation complexity between these different parallel models.

1.3. System Specification

All benchmarks were executed on a system with the following specifications:

- **CPU**: Intel(R) Core(TM) i7- 10700K CPU @ 3.80GHz.
- **GPU**: NVIDIA GeForce RTX 2060 SUPER with 8 GB VRAM
- **Software**: Python 3.12.2, CUDA Toolkit 12.9.1

2. Implementation Details

This section describes the implementation of the three distinct versions of the morphological operations. In this study, we constrain our analysis to square images and square structuring elements. Specifically, all structuring elements are designed with an odd number of cells per side. This simplification ensures that each structuring element possesses a unique central pixel, which is essential for anchoring the morphological operation symmetrically around each image pixel.

2.1. Sequential Implementation

The sequential baseline was implemented in Python using NumPy for data representation. The core logic relies on a set of nested loops. The outer loops iterate over every pixel (x, y) of the input image, while the inner loops iterate over the structuring element to find the minimum (for erosion) or maximum (for dilation) value in the corresponding neighborhood. Boundary checks are performed to ensure that pixel lookups do not go out of bounds. As shown in Listing 1, the sequential implementation nests 4 `for` loops. Due to the

interpreted nature of Python, such deeply nested loops are known to incur significant overhead, and we therefore anticipate this implementation to be computationally slow, forming a conservative but realistic lower bound for performance comparison.

```python
def erode_cpu(image, se):
    h, w = image.shape
    se_h, se_w = se.shape
    se_ch, se_cw = se_h // 2, se_w // 2
    output = np.zeros_like(image)

    # Iterate over each pixel of the input image
    for y in range(h):
        for x in range(w):
            min_val = 255
            # Apply the structuring element
            for j in range(se_h):
                for i in range(se_w):
                    if se[j, i] == 1:
                        img_y = y + j - se_ch
                        img_x = x + i - se_cw
                        # Boundary check
                        if 0 <= img_y < h and 0
                            <= img_x < w:
                            min_val =
                                min(min_val,
                                image[img_y,
                                img_x])
            output[y, x] = min_val
    return output
```

Listing 1: Simplified Python logic for sequential erosion.

2.2. CPU Multi-Processing Implementation

The CPU parallel version was implemented using Python's `multiprocessing.Pool`. This approach divides the image processing task into several independent sub-problems. The core idea is **data parallelism**. The input image is split into horizontal strips. The number of strips is equal to the number of available CPU cores, as reported by `multiprocessing.cpu_count()`. This works perfectly for morphological operations because the calculation for any given output pixel only depends on a small neighborhood of input pixels. A worker processing row N does not need to know the result from a worker processing row $N - 1$, making the tasks highly independent. To avoid the overhead of serializing and sending large image data to each worker, an `initializer` function is used to load the shared image and SE data into the global memory space of each child process upon creation. Each worker process is then assigned with a single strip (defined by a y-range) and executes a modified version of the sequential logic on its assigned portion. After all workers complete their tasks, the main process collects the processed strips and stitches them back together into the final output image using `np.vstack`. The core logic for distributing work is encapsulated in the `_run_in_parallel` function, as shown in Listing 2.

```python
def _run_in_parallel(image, se, worker_func):
    img_h, _ = image.shape

    num_processes = multiprocessing.cpu_count()

    # Divide the image into chunks (list of
        y-ranges)
    chunk_size = (img_h + num_processes - 1) //
        num_processes
    chunks = [(i, min(i + chunk_size, img_h))
              for i in range(0, img_h,
                  chunk_size)]

    # Create a pool of worker processes,
        initializing
    # each with shared data
    with multiprocessing.Pool(
        processes=num_processes,
        initializer=_init_worker,
        initargs=(image, se)
    ) as pool:
        # Map the worker function to the chunks
        results = pool.map(worker_func, chunks)

    # Stitch the resulting chunks back together
    return np.vstack(results)
```

Listing 2: The generic parallel runner function for CPU multiprocessing. It divides the image into horizontal chunks and distributes them to a pool of worker processes.

It is important to note that the decomposition of the image into horizontal strips for CPU multiprocessing is inherently safe from race conditions. Although a worker processing a boundary row must read pixels from a "halo" region that falls within another worker's assigned output strip, all such reads are performed on the original, unmodified input image. This input image is treated as a shared, read-only resource by all processes. Since each worker writes its results to a private, independent memory buffer, and no interprocess communication occurs during the computation phase, data consistency is guaranteed without the need for locks or other synchronization

mechanisms.

2.3. GPU (CUDA) Implementation

The GPU version leverages the massive parallelism of the CUDA architecture by assigning each pixel of the input image to an unique GPU thread. In this work, PyCUDA is used to manage the GPU. This includes compiling the CUDA C++ kernel (`SourceModule()`), allocating memory on the device (`cuda.mem_alloc()`), and transferring data between the host (CPU memory) and the device (GPU memory) using `memcpy_htod()` and `memcpy_dtoh()`. A simple, generic CUDA kernel was written to perform both erosion and dilation, controlled by a flag. The core principle is to launch a 2D grid of threads where each thread is responsible for computing a single output pixel. The structure of the grid is determined by the `BLOCK_SIZE` parameter, which defines the number of threads collaborating within a single group. Given the block size, the grid size is then calculated to ensure that enough blocks are launched to cover the entire image. This is accomplished using a standard integer arithmetic trick for ceiling division. This calculation guarantees that even for image dimensions that are not perfect multiples of the block dimensions, a thread will be mapped to every pixel. This strategy may launch more threads than there are pixels, a detail handled by the kernel. Inside the kernel, each thread calculates its unique global (x, y) coordinate using the built-in `blockIdx`, `threadIdx`, and `threadIdx` variables. This coordinate directly maps to the output pixel it must compute. The logic within each thread mirrors the sequential implementation: it iterates over the structuring element, calculates neighbor coordinates, performs boundary checks, and computes the local minimum or maximum. All memory reads for neighbor pixels are from the GPU's global memory.

```
__global__ void morph_kernel(
    const unsigned char* in_data,
    unsigned char* out_data,
    const int* se_data,
    int width, int height,
    int se_width, int se_height,
    int is_erosion)
{
    // Calculate the global x and y coordinates
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    int se_cx = se_width / 2;
    int se_cy = se_height / 2;

    unsigned char result_val = is_erosion ? 255 : 0;

    for (int j = 0; j < se_height; ++j) {
        for (int i = 0; i < se_width; ++i) {
            if (se_data[j * se_width + i] == 1) {
                int nx = x + i - se_cx;
                int ny = y + j - se_cy;

                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    unsigned char n_val = in_data[ny * width + nx];
                    if (is_erosion) {
                        result_val = min(result_val, n_val);
                    } else {
                        result_val = max(result_val, n_val);
                    }
                }
            }
        }
    }
    out_data[y * width + x] = result_val;
}
```

Listing 3: The generic CUDA kernel for morphological operations.

The compound operations, opening and closing, are implemented as sequences of the basic erosion and dilation kernels. A naive approach would involve two separate calls to our kernel, with the intermediate result being transferred back to the host (CPU) and then back to the device (GPU) for the second stage. This would incur significant data transfer overhead, as PCIe bandwidth is often a bottleneck. To mitigate this, our implementation employs a more efficient "*ping-pong*" buffer strategy on the GPU. As shown in the `_chained_op_template` function (Listing 4), this approach works as follows:

1. Two device memory buffers (`d_buffer_A` and `d_buffer_B`) are allocated for the image data, in addition to the buffer for the

structuring element.

2. The initial image is copied from the host to `d_buffer_A`.
3. The first kernel (e.g., erosion) is executed, reading from `d_buffer_A` and writing its output to `d_buffer_B`.
4. The second kernel (e.g., dilation) is then immediately executed, using the output of the first stage (`d_buffer_B`) as its input and writing the final result back into `d_buffer_A`.
5. Only the final result in `d_buffer_A` is copied back to the host.

This method completely eliminates the intermediate host-device data transfer, keeping the entire two-stage pipeline on the GPU and significantly improving the performance of the opening and closing operations.

```
def _chained_op_template(
    image: np.ndarray, se: np.ndarray,
        is_erosion_first: bool, block_size=(16,
        16, 1)
) -> np.ndarray:
    img_h, img_w = image.shape
    se_h, se_w = se.shape

    image = image.astype(np.uint8)
    se = se.astype(np.int32)


    d_se = cuda.mem_alloc(se.nbytes)
    d_buffer_A = cuda.mem_alloc(image.nbytes)
    d_buffer_B = cuda.mem_alloc(image.nbytes)

    cuda.memcpy_htod(d_se, se)
    cuda.memcpy_htod(d_buffer_A, image)

    # Input: d_buffer_A, Output: d_buffer_B
    _execute_gpu_kernel(
        d_buffer_A,
        d_buffer_B,
        d_se,
        img_w,
        img_h,
        se_w,
        se_h,
        int(is_erosion_first),
        block_size,
    )

    # Input: d_buffer_B, Output: d_buffer_A
    _execute_gpu_kernel(
        d_buffer_B,
        d_buffer_A,
        d_se,
        img_w,
        img_h,
        se_w,
        se_h,
        int(not is_erosion_first),
        block_size,
    )

    # The final result is now in d_buffer_A.
        Copy it back to the host.
    output_image = np.empty_like(image)
    cuda.memcpy_dtoh(output_image, d_buffer_A)

    # Free all GPU memory
    d_se.free()
    d_buffer_A.free()
    d_buffer_B.free()

    return output_image
```

Listing 4: The ”ping-pong” buffer strategy for efficient chained GPU operations, avoiding intermediate data transfers.

3. Performance Evaluation and Analysis

A rigorous benchmarking framework was developed to measure performances and compare the three implementations. Performance was measured across two primary axes: image size and structuring element (SE) size.

3.1. Experimental Setup

The key performance metric is **speedup**, calculated relative to the sequential baseline:

$$Speedup = \frac{t_s}{t_P}$$

where, in our setup, t_P can be either the execution time of the CPU-parallel or the GPU-parallel implementation, while t_s is always the execution time of the sequential implementation. Execution times for the CPU-based implementations were measured using `time.time()`. For the GPU implementation, `pycuda.driver.Event` was used to measure the kernel execution and data transfer time accurately, excluding host-side Python overhead.

The following parameters were varied:

- **Image Size**: Ranging from 256x256 up to 4096x4096 pixels with $\times 2$ increments.

- **Structuring Element Size**: 3x3, 7x7, 9x9, and 15x15.

A separate benchmark was conducted for the GPU implementation to analyze the impact of CUDA block size on performance, testing configurations from 4x4 to 32x32 threads per block.

3.2. Benchmarking Methodology

The benchmarking script, `benchmark.py`, automates the testing process. For each combination of image size and SE size, the following steps are performed:

1. **Data Generation**: A square test image of the specified size is generated. Each pixel is assigned a pseudo-random 8-bit integer value. A square structuring element is also created. To ensure that results are perfectly reproducible across runs, the random number generator is seeded with a fixed value (`np.random.seed(0)`) at the beginning of each configuration test.

2. **GPU Warm-up**: Before any timed measurements are taken, a small, untimed "warm-up" routine is executed on the GPU. This is a critical step that accounts for one-time overheads such as CUDA context creation and kernel just-in-time (JIT) compilation. By performing this warm-up, we ensure that subsequent timed runs reflect the true, steady-state performance of the GPU implementation.

3. **Execution and Timing**: The script then executes and times the CPU-Sequential, CPU-Multi-Processing, and GPU implementations on the identical generated input image.

4. **Correctness Verification**: After each run, the output of the parallel implementations is compared against the sequential baseline output using `np.allclose()` to verify correctness.

5. **Data Logging**: The configuration parameters, execution times, and calculated speedups for each test are systematically logged to a `.csv` file for subsequent analysis and plotting.

4. Results

This section presents the results from the comprehensive benchmarks. The analysis focuses on execution time and speedup to highlight the strengths and weaknesses of each parallelization strategy.

4.1. Execution Time Analysis

As shown in Figures 2 - 10, the execution times for all three implementations increase with both image size and SE size, as predicted by the computational complexity. The performance gap between the CPU implementations and the GPU implementation widens dramatically as the problem size (either image dimensions or SE dimensions) increases. This indicates that the fixed overhead of GPU data transfer becomes increasingly insignificant compared to the massive computational savings. The multi-processing version consistently outperforms the sequential version by a stable factor, demonstrating the benefit of utilizing all available CPU cores.

4.2. GPU Speedup Analysis

The speedup plots shown in Figures 11 - 14 provide the clearest picture of the GPU parallel implementation efficiency as the problem size increases. The GPU exhibits massive speedups, ranging from 1000x on smaller problems to well over 10000x on the largest image and SE sizes. Crucially, the GPU speedup is not constant; it *increases* with problem size. This demonstrates exceptional scalability, as larger problems can more effectively make use of the thousands GPU cores and hide data transfer latency.

4.3. GPU Block Size Analysis

The benchmark testing different thread block sizes (Figures 15 and 16) reveals a classic performance tuning curve. For the tested problem, performance is poor for very small block sizes (e.g., 4x4 or 8x8) because there are not enough threads

per block for the GPU's scheduler to effectively hide memory latency. Performance improves significantly and peaks around a block size of 16x16 (256 threads per block), which represents the optimal balance between parallelism and resource usage for this kernel on the test hardware. Increasing the block size further to the maximum value of 32x32 (1024 threads per block) shows either flat or slightly degraded performance (Figure 16), suggesting that the kernel may become limited by other resources at maximum block occupancy.

4.4. GPU vs. CPU-MP Speedup Comparison

While both parallel implementations offer a performance benefit over the sequential baseline, their scalability characteristics are fundamentally different due to their underlying architectures. A direct comparison of their speedups reveals the superiority of the parallel GPU approach.

The CPU Multi-Processing implementation delivers a consistent and predictable performance improvement.

As shown in Tables 1 through 4, its speedup quickly saturates to 7.2x - 7.5x when the input image size ranges from 1024x1024 to 4096x4096. The limited scalability of the CPU-MP implementation is evident. For the Erosion operation (Table 1) with a 15x15 SE, the speedup increment is only 0.3x when going from an input image of dimension 1024x1024 to 4096x4096.

When considering compound operations like Opening and Closing (Tables 3 and 4), the CPU-MP implementation even exhibits a slight drop in speedup at the largest image size. This indicates that further increases in problem size do not yield additional parallel efficiency for the CPU-MP implementation.

In contrast, the GPU implementation showcases greater scalability. Its speedup grows dramatically with the problem size. For a small 256x256 image, the GPU is already hundreds of times faster than the sequential code. For a large 4096x4096 image with a 15x15 structuring element, this speedup exceeds 60,000x (Tables 3 and 4). This is because larger problems can more effectively utilize the thousands of available GPU cores, fully amortizing the overhead of data transfer between the host and device. In summary, the results align with theoretical expectations. While the CPU-MP version provides a valuable speedup over the sequential version, the results demonstrates that the GPU's massively parallel architecture is fundamentally more effective for exploiting the data-parallel nature of morphological image processing tasks.

5. Conclusions

This project successfully implemented and compared three distinct implementations of fundamental morphological image processing algorithms. Through rigorous benchmarking, we have drawn clear conclusions about the efficacy of different parallel programming models for this embarrassingly parallel task. The sequential Python implementation served as a baseline to compute the speedups of the parallel implementations. The CPU multi-processing version demonstrated that CPU data-parallelism on a multi-core CPU provides a valuable performance improvement. The CUDA-based GPU implementation, however, proved to be the vastly superior approach. By leveraging a data-parallel model with thousands of threads, it achieved speedups of several orders of magnitude. More importantly, its performance advantage scaled with the problem size, making it exceptionally well-suited for the high-resolution images and large structuring elements common in computer vision applications. In conclusion, this study empirically confirms that for image processing tasks like basic morphological operations, the massively parallel architecture of a GPU offers performance that is unattainable with traditional CPU-based parallelism, highlighting the importance of choosing the right parallel model and hardware for the computational structure of the problem at hand.

6. Results Visualizations

This section contains all the relevant plots and tables referenced in the report.

6.1. Exectution time

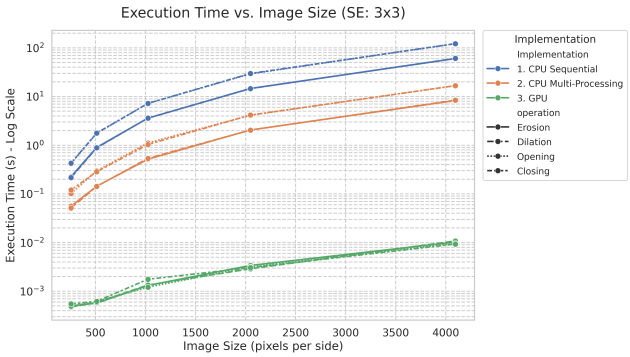

Figure 2: Execution time versus image size for a 3x3 structuring element.

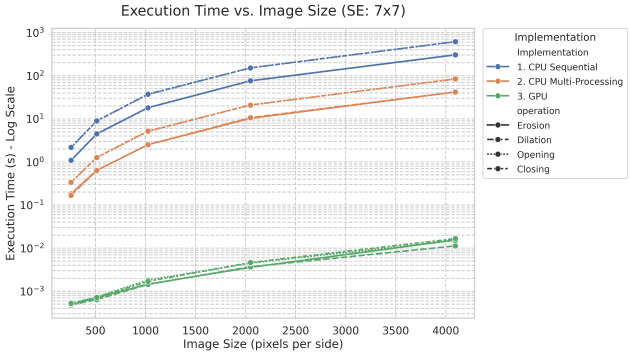

Figure 3: Execution time versus image size for a 7x7 structuring element.

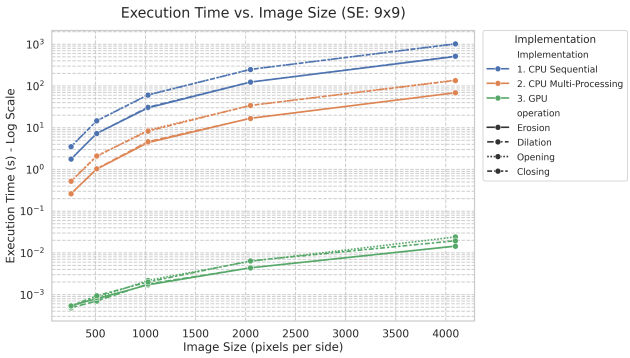

Figure 4: Execution time versus image size for a 9x9 structuring element.

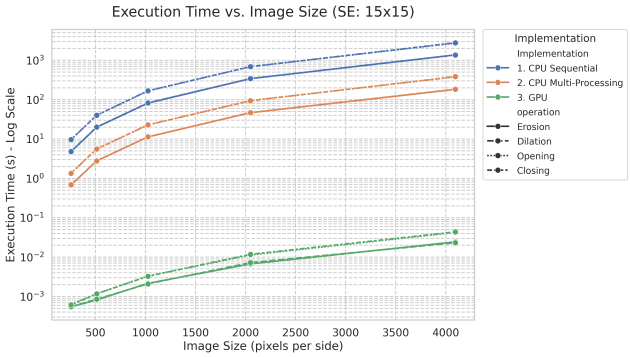

Figure 5: Execution time versus image size for a 15x15 structuring element.

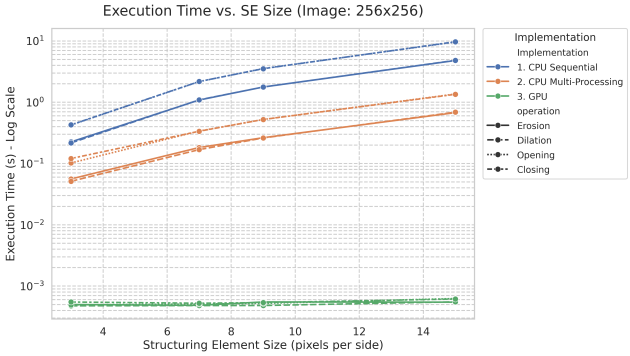

Figure 6: Execution time versus SE size for a 256x256 input image.

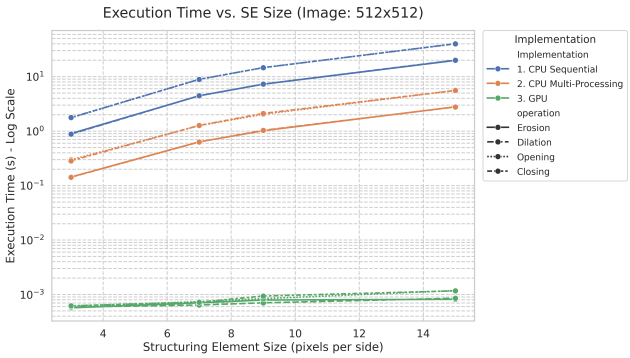

Figure 7: Execution time versus SE size for a 512x512 input image.

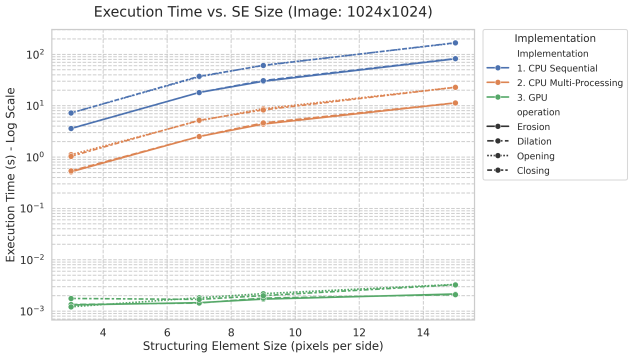

Figure 8: Execution time versus SE size for a 1024x1024 input image.

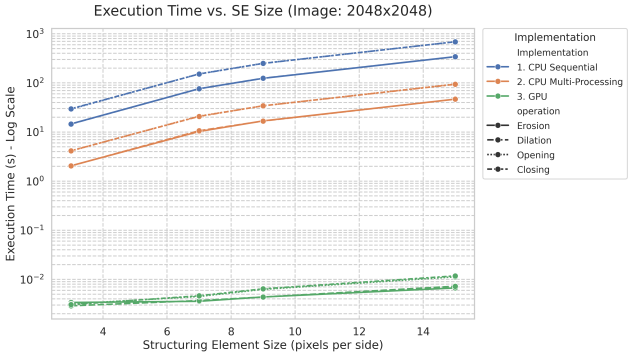

Figure 9: Execution time versus SE size for a 2048x2048 input image.

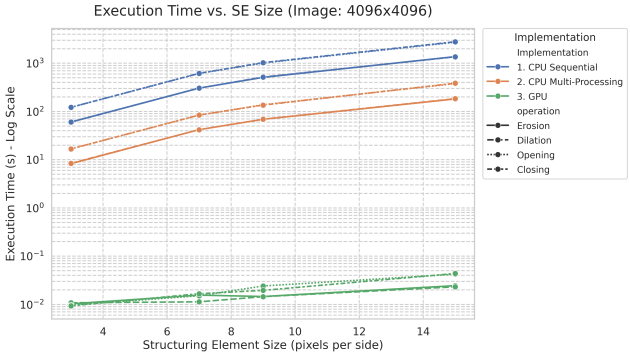

Figure 10: Execution time versus SE size for a 4096x4096 input image.

6.2. Speedups

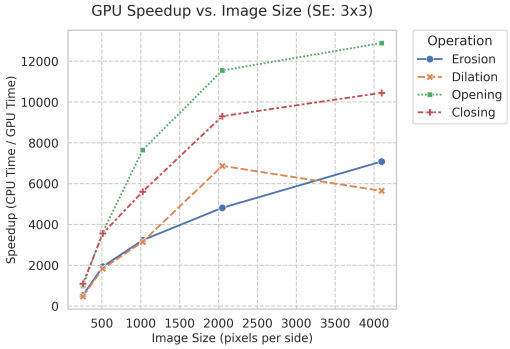

Figure 11: GPU speedup versus image size for a 3x3 SE

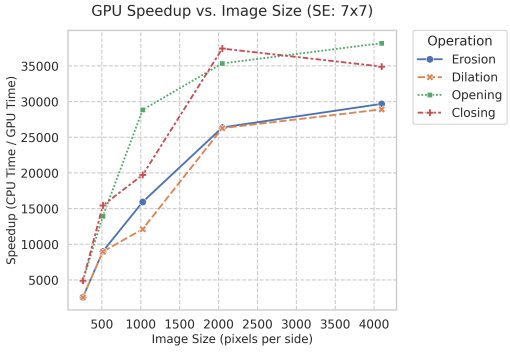

Figure 12: GPU speedup versus image size for a 7x7 SE

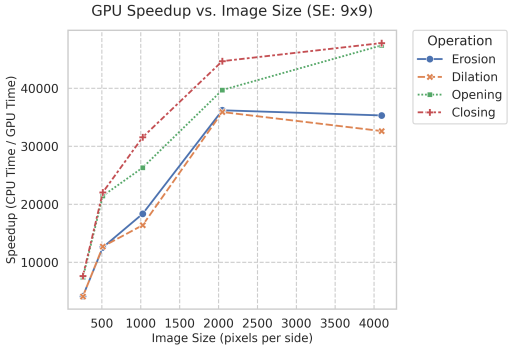

Figure 13: GPU speedup versus image size for a 9x9 SE

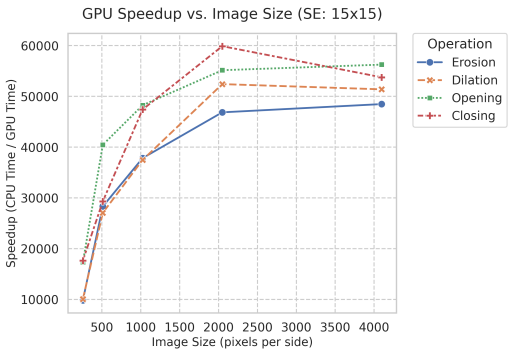

Figure 14: GPU speedup versus image size for a 15x15 SE

6.3. Blocksize benchmark

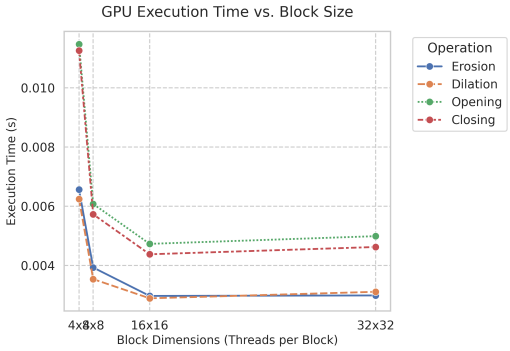

Figure 15: GPU execution time vs. block size for a 2048x2048 input image and a 9x9 SE

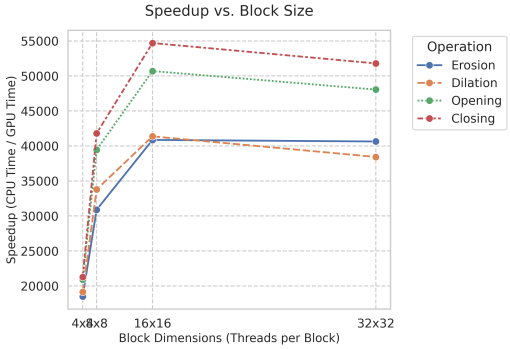

Figure 16: GPU speedup vs. block size for a 2048x2048 input image and a 9x9 SE

6.4. GPU vs. CPU-MP Speedup Comparison

Table 1: Speedup comparison for the **Erosion** operation.

Image	SE	CPU-MP	GPU
256x256	3x3	4.02x	451x
	15x15	7.09x	8,786x
1024x1024	3x3	6.96x	2,728x
	15x15	7.24x	37,895x
4096x4096	3x3	7.28x	5,863x
	15x15	7.54x	56,120x

Table 2: Speedup comparison for the **Dilation** operation.

Image	SE	CPU-MP	GPU
256x256	3x3	4.23x	455x
	15x15	7.01x	8,731x
1024x1024	3x3	6.72x	2,683x
	15x15	7.29x	39,530x
4096x4096	3x3	7.21x	5,616x
	15x15	7.39x	58,832x

Table 3: Speedup comparison for the **Opening** operation.

Image	SE	CPU-MP	GPU
256x256	3x3	4.16x	860x
	15x15	7.06x	15,365x
1024x1024	3x3	6.49x	5,957x
	15x15	7.25x	50,106x
4096x4096	3x3	7.27x	12,608x
	15x15	7.22x	64,707x

Table 4: Speedup comparison for the **Closing** operation.

Image	SE	CPU-MP	GPU
256x256	3x3	3.55x	780x
	15x15	7.21x	15,806x
1024x1024	3x3	6.97x	4,094x
	15x15	7.38x	51,657x
4096x4096	3x3	7.30x	13,083x
	15x15	7.24x	63,841x