Final Project: Parallel Image Processing

Johann Zhang, Mohammad Khan, Tung Pham

I. Introduction

Processing images and videos is a computationally intensive task. Operations such as flipping, rotating, and applying transformations on large datasets demand significant computational resources and are very relevant everyday content creation and manipulation. High resolution videos and large scale movies require hours in waiting time for video processing to occur, wasting essential time and money investing in better hardware each year. Photos also vary in scale; smaller photos are easier and faster to process but often done in large batches in the context of social media companies like Instagram or Facebook which employ compression techniques, once again highlighting the importance of parallelization to save seconds into years. Image processing on large images is also of great importance as is the case for histopathology imaging. This project aims to mitigate these performance issues by employing parallelization techniques, specifically leveraging OpenMP and MPI, to accelerate image processing tasks. The focus is on reducing processing time while maintaining accuracy, with an emphasis on flipping, rotating images, color transformation, and gaussian blur.

Image processing serves as an appropriate base for parallelization techniques due to the make up of an image. An image is a 2D array of pixels where each pixel is characterized by an x and y coordinate and its value, where the number of columns multiplied by the number of rows determines the resolution. The larger the resolution the greater the area and opportunity of parallelization since the image can be broken down in more parts for optimal parallelization which is ideal in extremely large images like in histopathology.

II. Background

Sequential image transformations involve iterating through each pixel in an image matrix along with each color channel that image was read in. While straightforward, this approach is prone to performance issues as it does not leverage modern multi-core processors for tasks that can be divided out and processed in parallel. This resulted in significant overhead and delays for high-resolution images or large datasets. By parallelizing these operations, the workload can be divided amongst multiple threads or processes, significantly reducing runtime and improving scalability.

Existing popular packages for image processing, like NumPy, do provide means for parallel implementation [3]. However, they do not provide an out of the box implementation for parallel. In addition, there are already packages that leverage these parallel techniques for performance boosting. One popular package specialized for image and video processing, OpenCV, is leverage OCL under the hood for optimization through optimization of their algorithm [1]. As an attempt at parallelizing the existing NumPy vector calculation, NumExpr provides an out of the box optimization on matrix acceleration through utilization of multi-thread approach [2].

In this project, we would like to reimplement some of the techniques with OpenMPI or OpenMP and attempt to reproduce the result of parallelism and discuss the performance benefit gained through different scalability tests and runtime comparison.

III. Implementation

In our project, we attempted to reimplement some of the approach with sequential as our baseline and attempt at a parallelization implementation using either OpenMPI or OpenMP. These techniques include Image Flipping, Image Rotation, Color transformation, Fourier Transform and finally kernel-based transformation with Gaussian Blur [1]. In all of our approaches, OpenCV only acts as a kernel for I/O operations and is stored in OpenCV Mat representation during processing. This is due to the nature of our data which will be discussed further in section IV.

3.1 Image Flipping

The sequential approach iterates through rows and columns of each channel, swapping pixel (for vertical) or whole row (for horizontal) data at index i with index $n - i$ where n is the total row (for horizontal) or column (for vertical) of the image. Horizontal flipping involved performing swap for rows from top to bottom, while vertical flipping swapped pixels along each row. Using OpenCV, the program loads the input image into memory that is represented using OpenCV's Mat structure, and then performs the necessary transformation. When using OpenCV to read the input image in, it represents the image as a 3D matrix with where the third dimension is the channels of the image, consisting of red, green and blue, each channel will have the corresponding value of that channel for each pixel in the image. This naive solution provided the foundational algorithms for parallel implementations and a baseline for comparison with our parallel implementation.

The parallel approach utilized OpenMPI for inter-process communication. The root process will handle reading the image file and store it in a MPI window which is shared amongst the processes. Then the input is divided amongst the processes by $\frac{n}{p}$ where n is the number of rows and p is the number of processes currently available on the machine. For horizontal flip, the algorithm is more straightforward as we only need to swap the memory pointer of index i with index $n - rank$. However, for vertical flip, a more manual approach where each pixel of a row at index i is flipped with $row\_length - i$. The algorithm for each flip direction is illustra[1]ted in algorithm 1 and 2.

[1] The implementation of this can be found in https://github.com/joseph280996/PHPC-Project

Algorithm 1 Parallel Horizontal Image Flip

```
1: procedure FlipHorizontalParallel(shared_data, rows, cols, channels, rank, num_processes)
2:     half_rows ← rows / 2
3:     block_size ← half_rows / num_processes
4:     start_row ← rank $\times$ block_size
5:     end_row ← (rank + 1) $\times$ block_size
6:     for i = start_row to end_row - 1 do
7:         corresponding_row ← rows - 1 - i
8:         Swap(row, corresponding_row) for all color channels
9:     end for
10: end procedure
```

Algorithm 2 Parallel Vertical Image Flip

```
1: procedure FlipVerticalParallel(shared_data, rows, cols, channels, rank, num_processes)
2:     block_size ← rows / num_processes
3:     start_row ← rank $\times$ block_size
4:     end_row ← (rank + 1) $\times$ block_size
5:     for i = start_row to end_row - 1 do
6:         for j = 0 to (cols / 2) - 1 do
7:             left_idx ← (i $\times$ cols + j)
8:             right_idx ← (i $\times$ cols + (cols - 1 - j))
9:             swap(left_idx, right_idx) for all color channel
10:         end for
11:     end for
12: end procedure
```

3.2 Image rotation

For parallel rotation, shared memory across all threads was allocated for both input image and output image. By storing the image in a shared memory location, each process can directly modify the image block that was assigned. This in turns reduces any overhead of data transmission and thus boosting the performance of the algorithm. With this in mind, each thread will write each (row, column) in its block on the input Mat to (column, row) on the output Mat.

Algorithm 3 90-Degree Image Rotation (Parallel with MPI)

```
1: procedure RotateImage90Parallel(input_data, output_data, rows, cols, channels, rank, num_processes)
2:     block_size ← rows / num_processes
3:     start_row ← rank × block_size
4:     end_row ← (rank+1) × block_size
5:     if direction = CLOCKWISE then
6:         for r = start_row to end_row-1 do
7:             for c = 0 to cols-1 do
8:                 out_r ← c
9:                 out_c ← rows - 1 - r
10:                Assign at $(out\_r, out\_c)$ with value at $(in\_c, in\_r)$
11:            end for
12:        end for
13:    else                                        ▷ COUNTERCLOCKWISE
14:        for r = start_row to end_row-1 do
15:            for c = 0 to cols-1 do
16:                out_r ← cols - 1 - c
17:                out_c ← r
18:                Assign at $(out\_r, out\_c)$ with value at $(in\_c, in\_r)$
19:            end for
20:        end for
21:    end if
22: end procedure
```

3.3 Color transformation

In color transformation, work was distributed in accordance with the number of rows of the input image where each process handles $\frac{r}{p}$, where r is the number of rows of the image and p is the number of processes available on the machine. The image is again being stored in a shared memory location to reduce any data communication overhead.

Algorithm 4 Parallel Image Color Channel Transformation

```
Require: Image of dimensions $rows \times cols$, number of processes $P$
Require: Channel increments: $red\_inc$, $green\_inc$, $blue\_inc$
 1: $block\_size \leftarrow \lfloor rows/P \rfloor$
 2: $start\_row \leftarrow rank \times block\_size$
 3: $end\_row \leftarrow \begin{cases} rows & \text{if } rank = P - 1 \\ (rank + 1) \times block\_size & \text{otherwise} \end{cases}$
 4: for $r \leftarrow start\_row$ to $end\_row - 1$ do
 5:     for $c \leftarrow 0$ to $cols - 1$ do
 6:         $new\_color \leftarrow$ Calculate the new magnitude of each channels
 7:         $row[r, c] \leftarrow new\_color$
 8:     end for
 9: end for
10: Synchronize all processes
```

3.4 Fourier Transform

In the implementation of Fast Fourier transform, image size was assumed to be the power of 2. The baseline sequential approach used Cooley-Tukey algorithm for 1D fast fourier transform [4]. In order to adapt this to a 2D image input, a pipeline was developed which took inspiration from the curriculum approach as described in Figure 1.

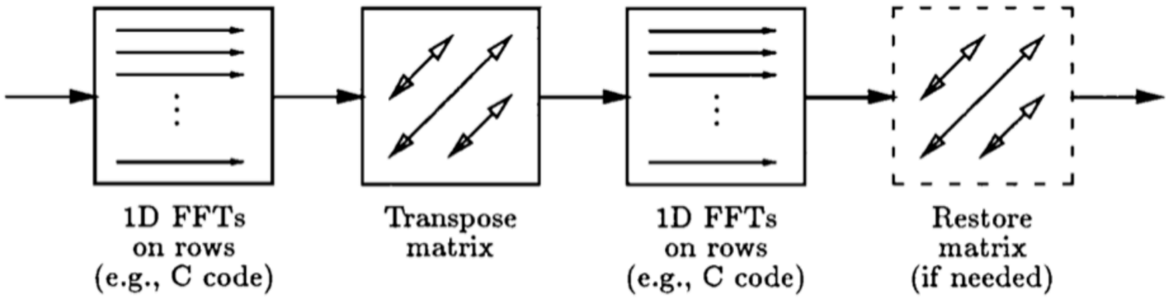

Figure 1. 2D Fast Fourier Transform pipeline.

In the parallel approach, due to complication in implementation, OpenMP was utilized instead of OpenMPI for parallelization. This allows parallelism within a single computer rather than across a cluster of computers but was still able to accomplish the work and demonstrate the potential of parallelism for this task.

To enhance the sequential approach, the butterfly operation was initially selected for optimization. As each block performs independent operations on its allocated portion of the data, it was safe to parallelize this without concerns on data race or conflict access across threads.

Then each phase of the pipeline can also be parallelized as each operates on a separate row and therefore is not prone for data access conflicts.

Algorithm 5 1D Fast Fourier Transform

```
1: procedure FFT(x)
2:     n ← length(x)
3:     bits ← $\log_2(n)$
4:     result ← new array of size n
                                                  ▷ Bit reversal stage
5:     for i = 0 to n-1 do
6:         result[BitReverse(i, bits)] ← x[i]
7:     end for
                                                  ▷ Butterfly operations
8:     for stage = 1 to bits do
9:         m ← $2^{\text{stage}}$
10:        half_m ← m/2
11:        w_m ← $e^{-2\pi i/m}$
12:        for k = 0 to n-1 step m do in parallel
13:            w ← 1
14:            for j = 0 to half_m-1 do
15:                t ← w × result[k + j + half_m]
16:                u ← result[k + j]
17:                result[k + j] ← u + t
18:                result[k + j + half_m] ← u - t
19:                w ← w × w_m
20:            end for
21:        end for
22:    end for
        return result
23: end procedure
```

Algorithm 6 2D Fast Fourier Transform

```
1: procedure FFT2D(channel)
2:     rows ← channel.rows
3:     cols ← channel.cols
4:     padded_rows ← NextPowerOf2(rows)
5:     padded_cols ← NextPowerOf2(cols)
6:     complex_image ← new 2D array[padded_rows][padded_cols]
7:     Convert image to complex numbers and pad
                                                  ▷ Apply FFT to rows
8:     for i = 0 to padded_rows-1 do in parallel
9:         row ← complex_image[i]
10:        row ← FFT(row)
11:    end for
                                                  ▷ Apply FFT to columns
12:    for j = 0 to padded_cols-1 do in parallel
13:        col ← new array[padded_rows]
14:        col ← FFT(col)
15:    end for
        return complex_image
16: end procedure
```

3.5 Gaussian Blur

Gaussian Blur is a smoothing operation that reduces image noise and detail by applying a mathematical function called the Gaussian function (bell curve) to each pixel and its neighbors. In this process, each pixel's new value is calculated as a weighted average of itself and neighboring pixels and the closer the neighbors the more influence (higher weights) it has on the single pixel than distant ones. In the implementation, a kernel of size 5 was used as the weight matrix for each pixel, the value of each weight follows Gaussian distribution.

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

A 2 pass implementation was used where the kernel updates follow the first pass in the horizontal direction and the seconds pass follows the vertical direction.

Similar to the Fast Fourier Transform approach, due to implementation complications, OpenMP was again utilized in place of OpenMPI for a shared memory implementation.

Algorithm 7 Parallel Gaussian Blur using OpenMP

```
Require: Image $I$ of size $w \times h$, radius $r$, sigma $\sigma$
 1: $k \leftarrow$ CreateGaussianKernel($r$, $\sigma$)
 2: $T \leftarrow$ temporary buffer of size $w \times h \times channels$
 3: for each pixel in each channel of a row in parallel do
 4:     $sum \leftarrow 0$
 5:     for $i \leftarrow -r$ to $r$ do
 6:         $sum \leftarrow sum + I[y, srcX, c] \times k[i + r]$
 7:     end for
 8:     $T[y, x, c] \leftarrow sum$
 9: end for
10:
11: Implicit barrier synchronization
12: for each pixel in each channel of a column in parallel do
13:     $sum \leftarrow 0$
14:     for $i \leftarrow -r$ to $r$ do
15:         $sum \leftarrow sum + T[srcY, x, c] \times k[i + r]$
16:     end for
17:     $I[y, x, c] \leftarrow sum$
18: end for
19:
```

IV. Experiment

In this project, execution time comparison was conducted by recording the execution time of an approach on a single image. The execution time is defined as the time taken to purely perform the transformation. This is without taking into consideration any I/O operations or communication overheads to create a window or shared memory location across the processes or threads. For OpenMPI implementations, the test was conducted using 8 parallel processes while for OpenMP implementation was tested using 10 concurrent threads.

The machine used for testing was an M2 Macbook with 10 cores. The test image for the comparison test was sampled from the WIDER [5] dataset, a popular dataset for face detection, that has a size of 1024x759.

For a strong scalability test, the same image for comparison test was used while the number of processes or threads increased from 1 to 10 for implementations with OpenMPI (as it was impossible to increase the number of processes beyond the maximum available cores of the machine). For OpenMP implementations, the number of threads increases from 1 to 40.

For the weak scalability test, the image sampled from WIDER was resized to meet the need of increasing input. The $\frac{p}{n}$ ratio was kept fixed at approximately $\frac{1}{62500}$. Since each image has rows and columns, n was defined as a total number of pixels ($rows * columns$).

V. Results

The execution time of each method was recorded for comparison as well as weak and strong scalability tests were conducted to better understand the performance benefit of each method.

5.1 Parallel and Sequential Comparison

Method	Parallel Execution Time (ms)	Sequential Execution Time (ms)	Percent reduction (%)
Flipping Vertical	2.462	3.112	20.887
Flipping Horizontal	4.972	16.307	69.51
Rotation	6.441	10.010	35.654
Color Transformation	1.284	8.540	84.965
Fourier Transform	900.653	2539.706	64.537
Gaussian Blur	112.109	542.606	79.339

Table 1: Execution time on the test image using sequential approach and parallel approach. Percent reduction was calculated using $\frac{parallel\_time - sequential\_time}{sequential\_time} * 100\%$.

5.3 Strong Scalability Test Results

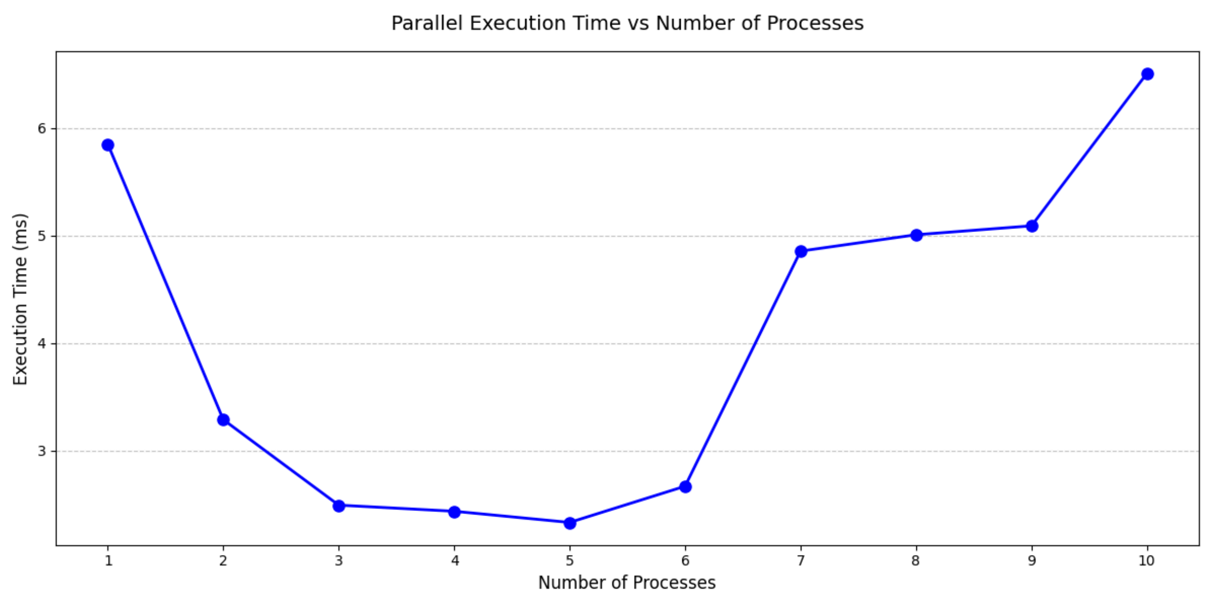

Figure 2: **Strong** Scalability Test result for **Image Rotation**. The graph shows clear signs of reduction until 5 processes and clear signs of increase in execution time as the number of processes increases beyond this point. One potential cause is there are other programs that are running at the time of the test causing less available cores leading to overhead as it needs to wait. This however, indicates that the implementation did not scale well as the number of processes increased.

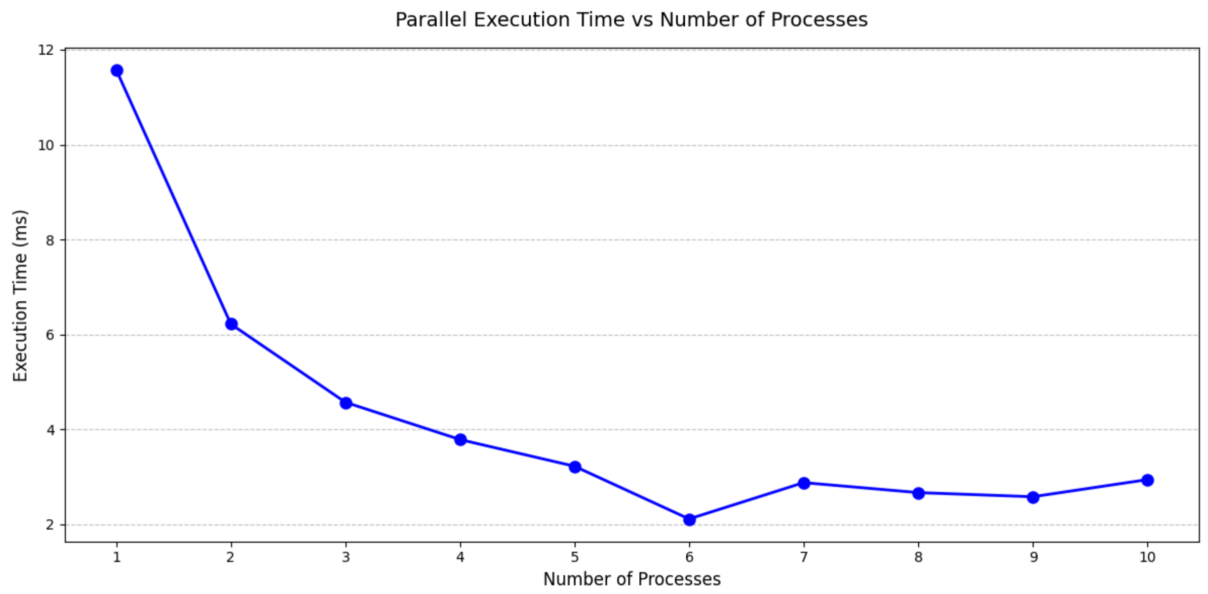

Figure 3: **Strong** scalability Test result for **Horizontal Flipping**. The graph illustrates a smooth reduction as more processes were added and no real sign of increase in performance degradation until the number of processes reach 10.

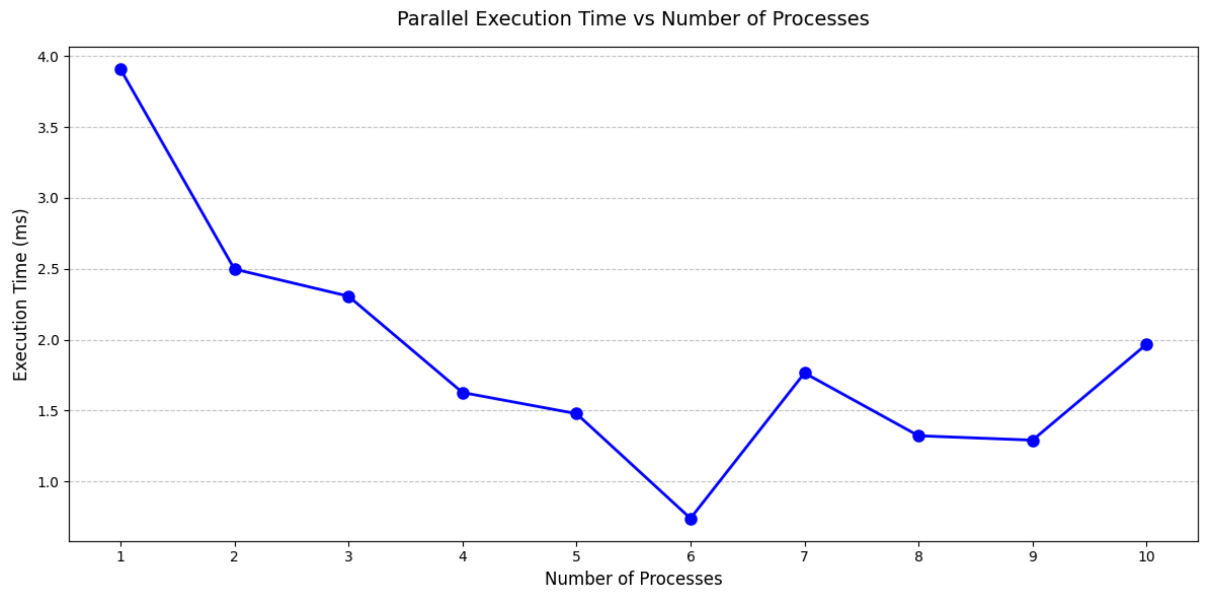

Figure 4: **Strong** Scalability Test result for **Vertical Flipping**. The graph shows a clear reduction in execution time until 6 processes were utilized, beyond this point, there is a clear degradation in performances which indicates poor scalability test result.

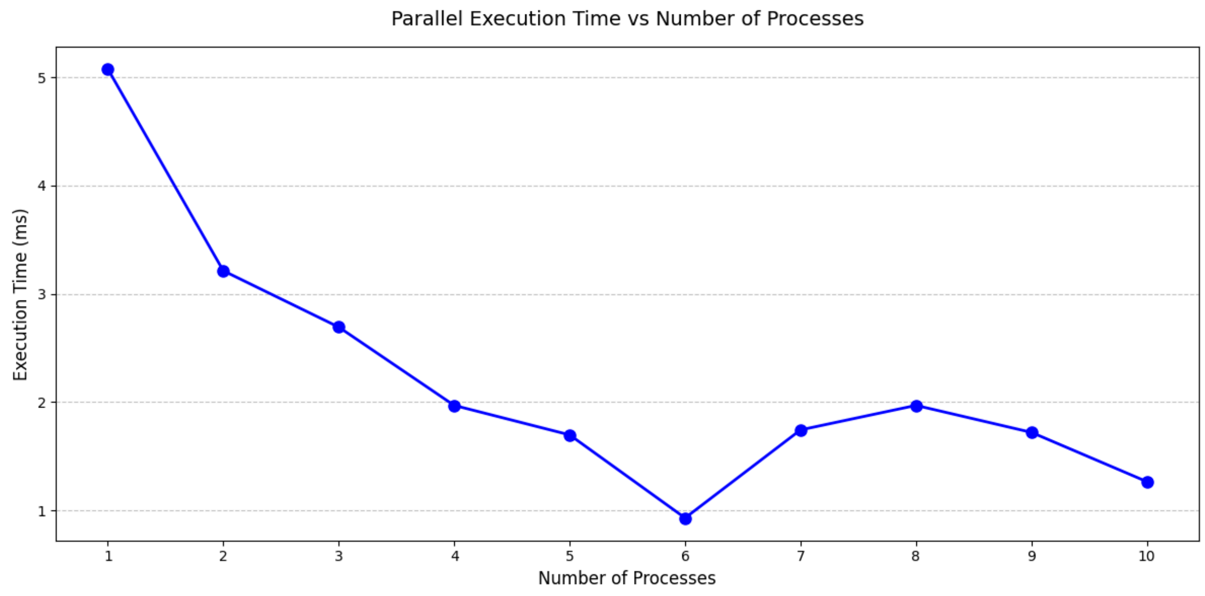

Figure 5: **Strong** Scalability Test result for **Color Transformation**. The graph shows a similar distribution as the previous flipping. However, near 10 processes count, it has a downward trend while for flipping it has an upward trend.

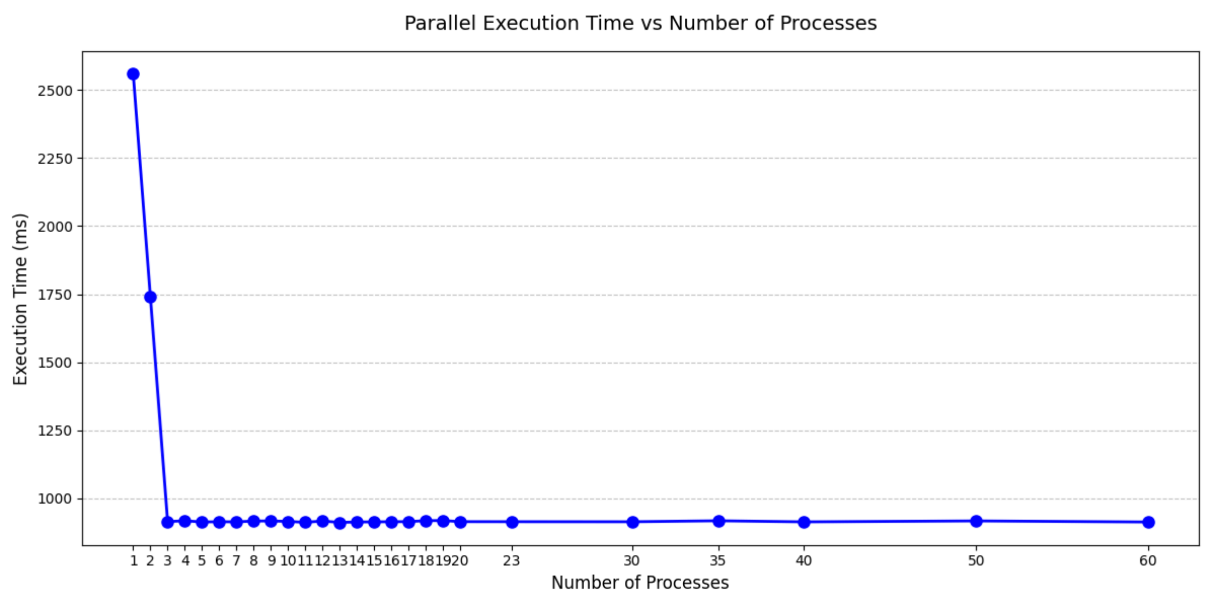

Figure 6: **Strong** Scalability Test result for **FFT**. The graph shows a linear drop from 1 thread used to 3 threads used. Beyond this point, the graph plateau and has minimal changes. This indicates a good strong scalability where as we add more threads, the execution time is near linear. In addition, as the number of threads increased, there was no sign of performance degradation and no performance benefit as additional parallelism didn't yield better results.

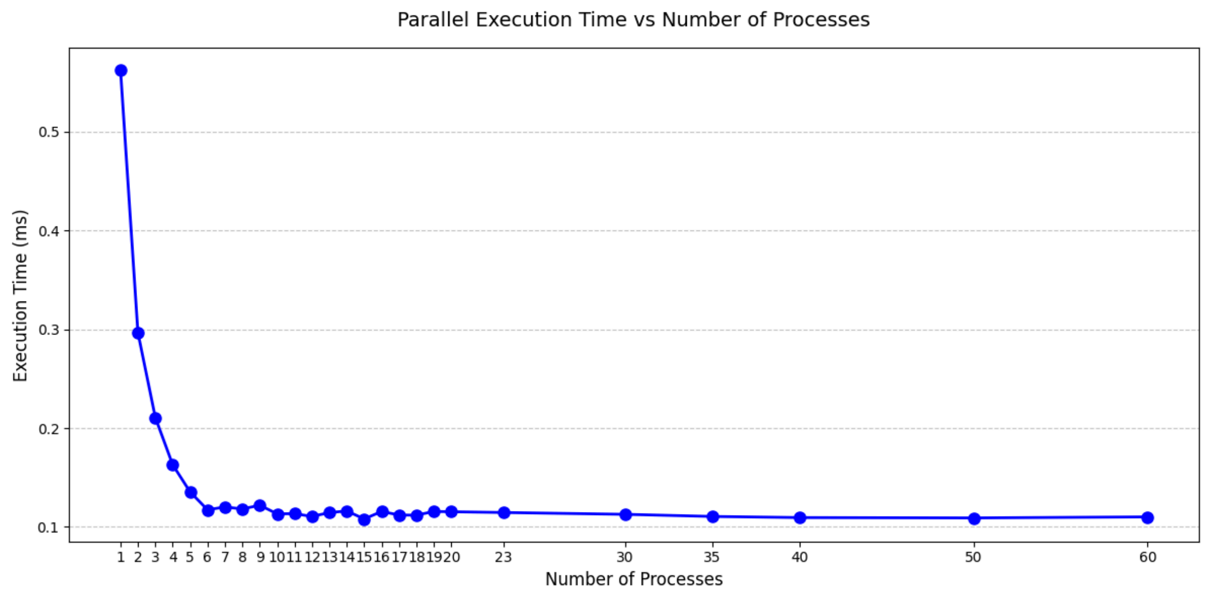

Figure 7: Strong Scalability Test result for **Gaussian Blur**. The graph shows a similar distribution as the FFT implementation. However, it has a smooth curve and a gradually downward trend in comparison. It also plateaus beyond 20 threads used and doesn't exhibit any upward or downward trends which indicates that as more threads are added, it reaches the point of convergence but doesn't go up in execution time which indicates that the implementation scales well as more threads are added.

5.2 Weak Scalability Test Results

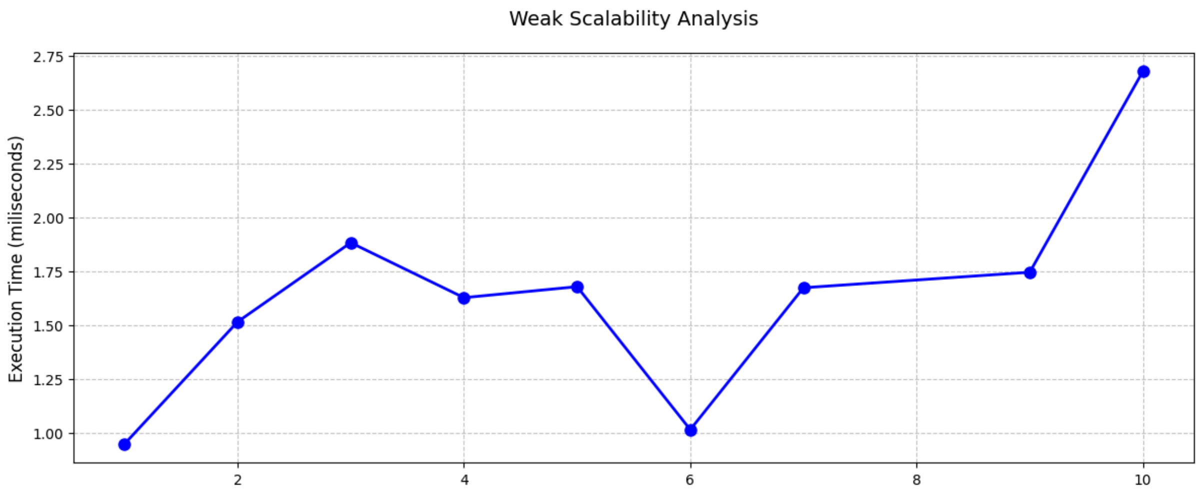

Figure 8: Weak Scalability Test result for **Flipping Horizontally**. As the number of parallelism added, the execution time increases initially when number of processes increases from 1 to 3. However, from 4 to 9 processes used, the plot exhibits a more constant execution time achieved. A significant drop at 6 processes used was recorded. This first part indicates a fairly good scalability test. However, beyond that, the plot exhibit an upward trends indicating a poor weak scalability test result.

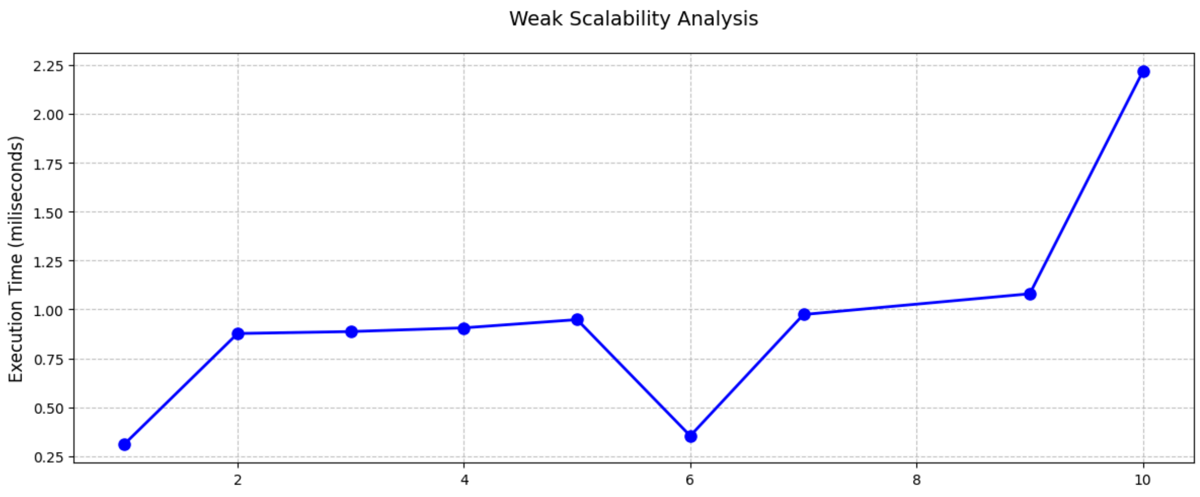

Figure 9: Weak Scalability Test result for **Flipping Vertically.** Similar to horizontal implementation, a sudden raise was recorded when the number of parallelism increased initially. However, the execution began to stay constant with a slight upward trend and had a sudden drop at 6 processes. This indicates a good weak scalability. Once reaching 10 processes, a sudden raise in execution time was recorded indicating that once reaching this number of processes, the implementation has very poor weak scalability.

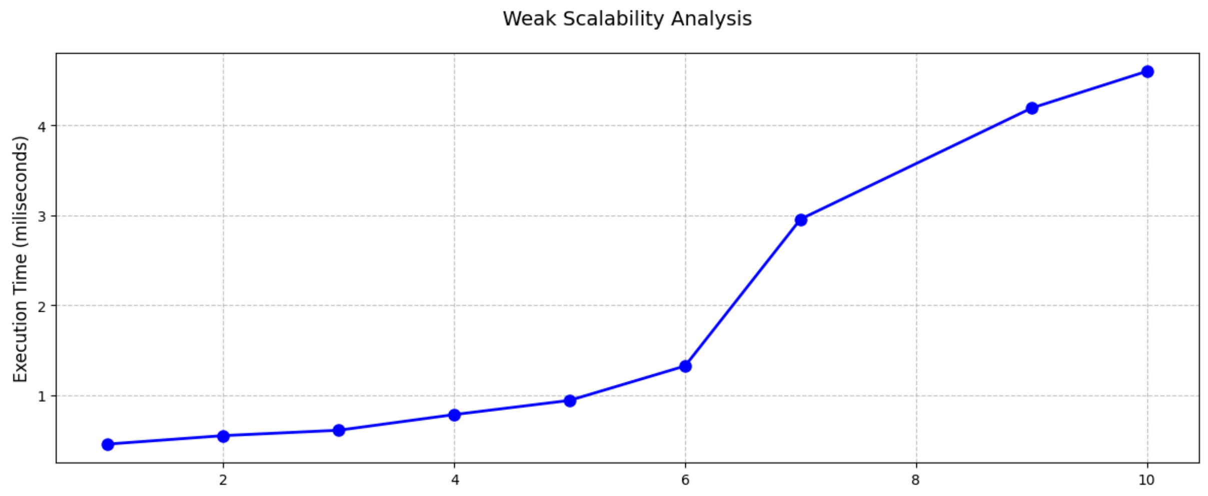

Figure 10: Weak Scalability Test result for **Rotation.** Unlike flipping implementations, rotation implementation graphs have a cubic-root like shape. The graph appears to have a slightly upward trend with 1 process until 6 processes utilized which indicates a good weak scalability. The execution time then has a sudden rise from 6 processes to 7 processes and appears to start converging post 8 processes which indicate a poor scalability test result.

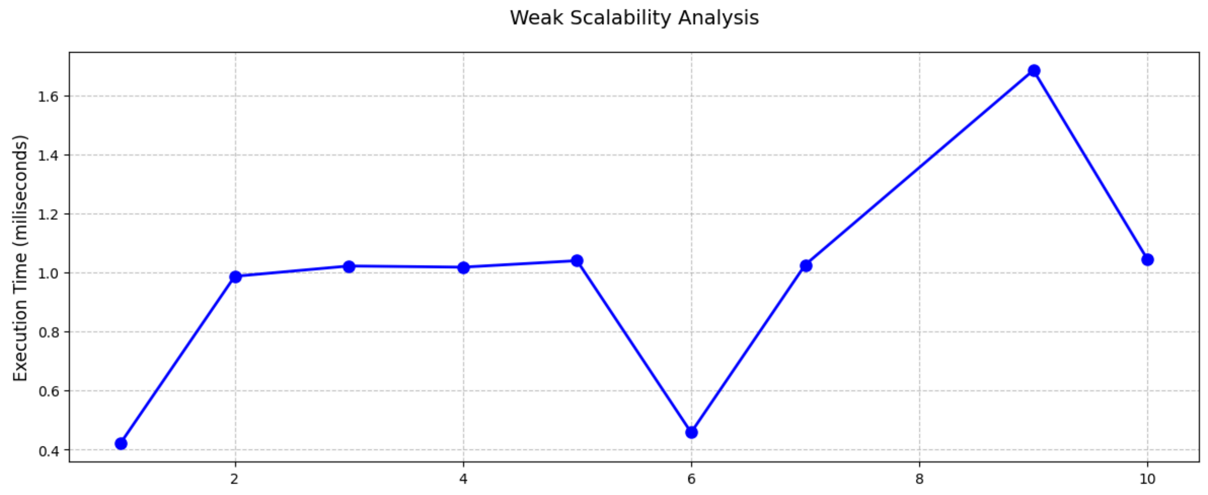

Figure 11: Weak Scalability Test result for **Color Transformation.** The plot's appearance is similar to that of flipping results where a sudden rise in execution time was observed when an additional parallelism was added on top of increased input size. However, the plots remain constants for 2 to 5 processes used and a sudden drop at 6 processes and rise back up at 7. A sudden rise again at 9 processes utilized and drop at 10 which indicates that the execution time stabilized around 1 ms of execution time with a fluctuation of 0.6. This is a sign of a good weak scalability result as the amount of time converges or remains constant as the input size and more parallelism is added.

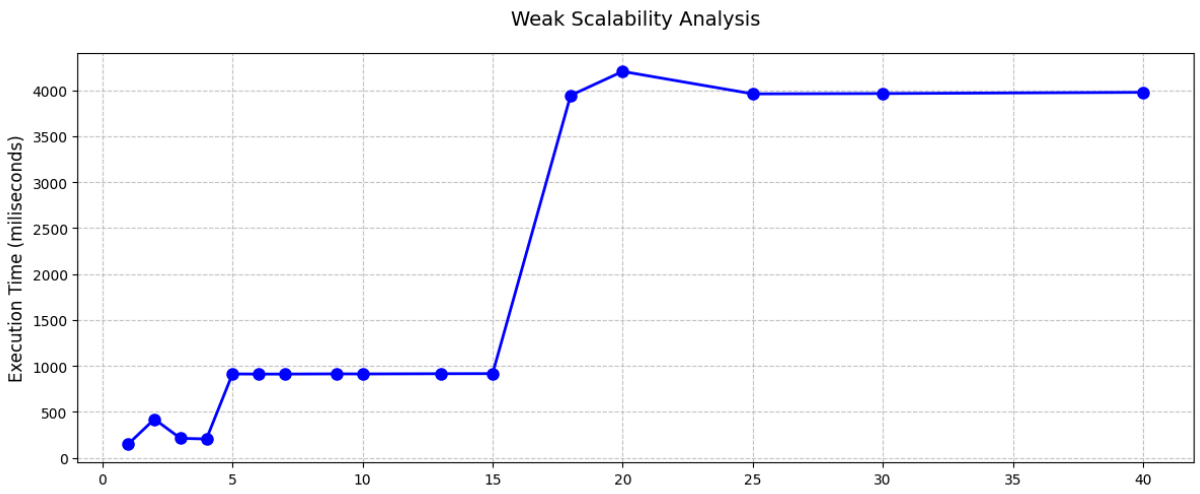

Figure 12: Weak Scalability Test result for **FFT.** The plot exhibits a good scalability initially from 1 threads to 4 threads utilized. A sudden rise was observed when spawning 5 threads for processing and the plot remained constant until 15 threads were utilized which indicates signs of good scalability. When utilizing beyond 15 threads, the plot exhibits a rise in execution time again. However, the time taken for completion remains constant afterwards. This shows signs of poor scalability post 15 threads.

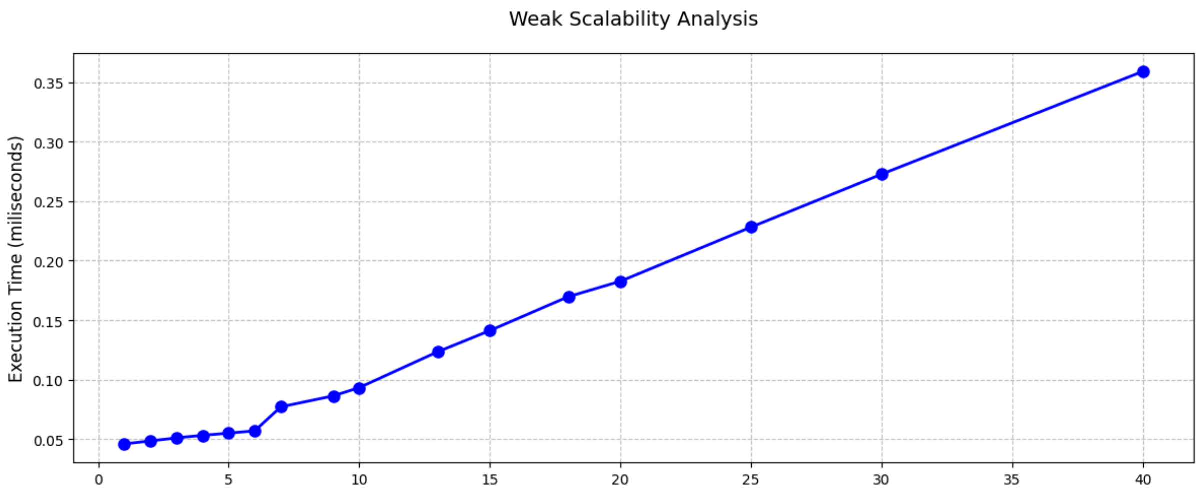

Figure 13: Weak Scalability Test result for **Gaussian Blur.** The plot initially appears to have a more constant result when utilizing 1 to 6 threads. However, when reaching 7 threads, a sudden rise was observed and the plot began to have a linear behavior upward which indicates a poor scalability as the number of input and parallelism increases. The later part could be the result of exceeding the maximum parallelism of the tested machine.

VI. Discussion

The project achieved significant successes in terms of performance. Parallel implementations demonstrated notable time reductions when compared to sequential time. The use of shared memory and efficient data partitioning ensured minimal communication overhead between processes.

In the strong scalability tests, most implementations produce observable good scalability results. Only rotation results pale in comparison to the others implementations. However, with a limited number of processes utilized, rotation implementation produced a very capable result which was able to achieve better results when compared to sequential implementation run time.

In weak scalability tests, due to inability to increase the number of processes over the limitation of the tested machine, it is hard to make a clear comparison for OpenMPI implementations in situations where number of processes kept being added beyond the parallel capability of the machine while increasing the input size like tests done with OpenMP implementations. However, with its current number of parallelism available, it was able to produce very promising results where execution time for flipping, rotation, and color transformation remains constant on some configurations. For OpenMP implementation, we can see that the test result was good while the number of resources utilized remains in the parallel capability of the machine. When going beyond that, the algorithm suffers from delays as more threads added must wait for available processors to pick up causing overhead.

However, the project faced challenges, particularly in managing data synchronization. Initial attempts to implement parallel rotations resulted in incorrect outputs due to improper memory handling. These issues were resolved by refining the use of MPI shared memory and ensuring proper synchronization through barriers. Another limitation was the focus on basic transformations, which restricted the exploration of more complex image processing techniques such as Gaussian blurring and Fast Fourier Transform. This resulted in limited understanding of the techniques leading to improper utilization of MPI to parallelizing tasks across processes and resulted in segmentation fault issues.

VII. Future Work

Future work will focus on extending the functionality of the current implementation. Advanced transformations like compression techniques and more, including convolution operations like Gaussian blur and edge detection will be implemented. Another possible area of expansion could be the parallelization of IO operations that will be explored to enhance performances when loading large videos or images which is crucial in the case of histopathology images spanning over 10GB. Current implementations rely on sequential loading, which can be a bottleneck and cause for a high initial start time even before processing begins.

In addition, a combination of OpenMP and OpenMPI will also be implemented to scale the operation on multiple images at once and test on a high performance cluster of computers. This allows better incorporation of the process nodes and threads within each node, potentially, providing even better performance boosting.

Additionally, dynamic load balancing techniques will be explored to improve workload distribution, inspired by load balance techniques that can be seen in operating system process distribution like round robin and more. Exploring GPU acceleration through CUDA may be another major enhancement, allowing even faster processing times for large images and videos, however, this may be far too complex as CPU parallelization is already challenging enough.

VIII. Conclusion

This project successfully demonstrated the advantages of parallelization in image processing tasks. By leveraging MPI, significant performance improvements were made, making the approach valuable for resource intensive applications that range from small scale to large scale image processing scenarios. While some challenges arose, they provided valuable insights into the complexities of parallel computation and algorithm design and highlighted the need for parallelization. The results of this project suggest future enhancements and efficient image processing solutions that are yet to be made in industry. Future work like IO parallelization will be implemented in order to make further advancements.

References

1. Khronos Group. (n.d.). OpenCL acceleration with OpenCV. OpenCV. Retrieved December 21, 2024, from https://opencv.org/opencl/
2. PyData Development Team. (n.d.). PyData/numexpr: Fast numerical array expression evaluator for Python, NumPy, PyTables, pandas, and more. GitHub. Retrieved December 21, 2024, from https://github.com/pydata/numexpr
3. NumPy Development Team. (n.d.). SIMD optimizations in NumPy. NumPy Documentation. Retrieved December 21, 2024, from https://numpy.org/doc/stable/reference/simd/index.html
4. Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 19(90), 297-301. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf
5. Yang, S., Luo, P., Loy, C. C., & Tang, X. (n.d.). WIDER FACE: A face detection benchmark. Retrieved December 21, 2024, from http://shuoyang1213.me/WIDERFACE/