PIPL - Parallel Image Processing Library

Agastya Sampath
Carnegie Mellon University
Pittsburgh, USA
agastyas@andrew.cmu.edu

Shijie Bian
Carnegie Mellon University
Pittsburgh, USA
sbian@andrew.cmu.edu

Abstract

In this project, we aimed to explore the benefits of parallel programming in the context of image processing. We implemented several image processing techniques, including image denoising, color enhancement, and color conversion, and created a comprehensive image processing library (PIPL - Parallel Image Processing Library) using these techniques. To improve the performance of the library, we utilized parallelizing tools such as OpenMP and CUDA on GPU to distribute the workload and increase processing speed. We conducted detailed profiling of the library's performance under various tasks and parallelization scenarios with different computing resources, hoping to provide further insight into how parallel programming can greatly benefit tasks related to image processing.

The detailed documentation of our project and the source code can be accessed from our project webpage[1].

Reference Format:
Agastya Sampath and Shijie Bian. 2023. PIPL - Parallel Image Processing Library. Carnegie Mellon University, Pittsburgh, PA, USA, 14 pages.

1 Background

1.1 Motivation

As a team, we believe that image processing is a critical area of study due to the ubiquity of images in our daily lives. From social media to medical imaging, images are essential in a variety of fields, and ensuring their quality is of utmost importance. However, image processing can be a computationally intensive task, especially when dealing with large images. As such, we recognize the importance of leveraging parallelism to accelerate image processing techniques. This project provides us with a unique opportunity to contribute to the development of a parallel image processing library, where we can apply and analyze various parallel programming techniques, such as OpenMP and CUDA on GPU. The non-trivial nature of this project lies in the fact that different processing techniques may benefit from parallelism differently, and thus require careful profiling and analysis to ensure optimal performance.

[1] https://agastya-sampath.github.io/618-project-s23/

1.2 Image Denoising

Image denoising is the process of removing unwanted noise or disturbances from an image. Noise in an image can be introduced due to various reasons such as low-light conditions during image capture, transmission errors, or limitations of the image sensor. The presence of noise in an image can reduce the quality of the image and make it difficult to interpret or analyze. As demonstrated in Figure 2, image denoising techniques remove the noise and restore the original image as accurately as possible with minimal impact on the resolution. There are various image-denoising techniques available that can be used depending on the type of noise and the level of noise reduction required. In our project, we explored two main algorithms: **Median Filtering** [4] and **Non-Local Means Filtering (NLM)** [1].

Specifically, the **Median Filtering** algorithm performs noise removal and smoothing by replacing each pixel value in the image with the median value of its neighboring pixels. Specifically in our algorithm which is detailed in the **Approach Section**, we start by finding all the neighboring pixels within the filter range for the current pixel. We then sort the RGB values of these neighboring pixels and select the median pixel. We average the RGB values of the chosen median pixel and assign the result to the output image at the corresponding position. This process is repeated for every pixel in the image, resulting in a denoised output image.

1. **Key Data Structures**: Two-dimensional arrays of unsigned char pixels representing the original image and resulting denoised image constructed using the CImg library [5], a vector of RGB tuples representing the neighbor pixels within the filter range for each pixel in the original image, and an RGB tuple representing the accumulated values for each color channel of the median pixel
2. **Key Operations**: A standard sorting performed on the vector of RGB values.
3. **Inputs**: The original image, the filter size, and the percentage scaling of filter size.
4. **Outputs**: The resulting denoised image (note that it will be smaller and of less resolution due to the median filtering of neighborhood pixels).
5. **Computationally Expensive Portions**: the nested loop that iterates over each pixel in the original image and finds the neighbors within the filter range, which can benefit from parallelism.

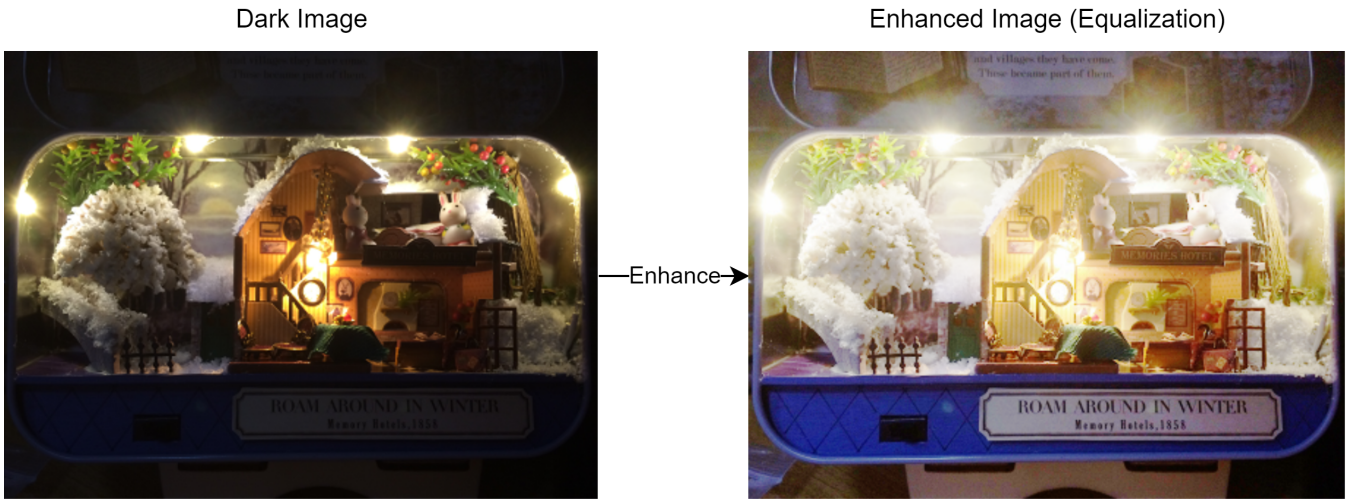

Figure 1. An example demonstrating the color enhancement tool, using Histogram Equalization

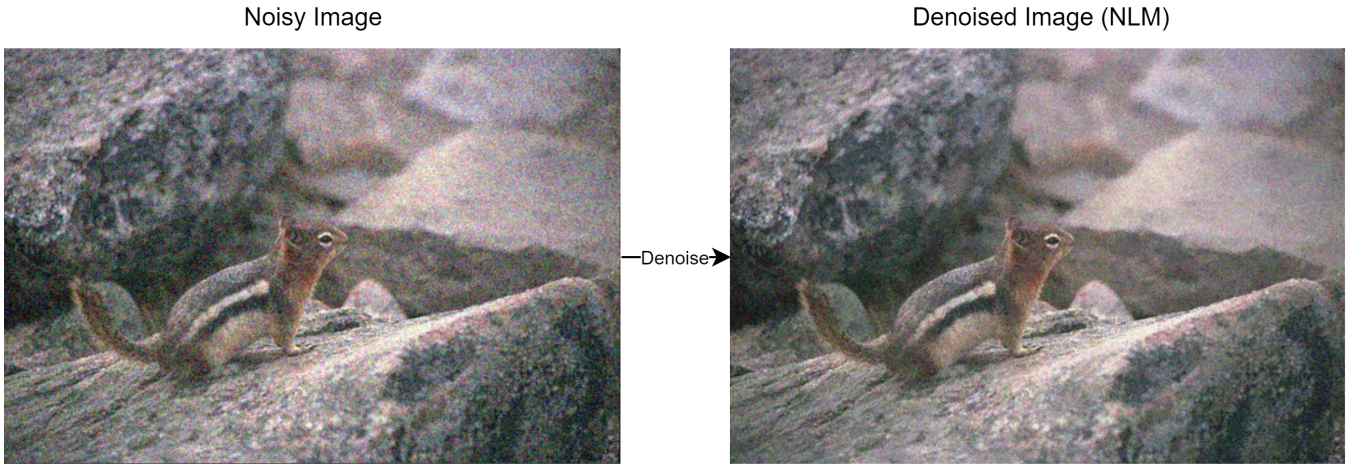

Figure 2. An example demonstrating the image denoising tool, using Non-Local Means Filtering

The second algorithm that we implemented, the **Non-local Means** algorithm, performs denoising by computing a weighted average of the pixel values in similar patches around that location, where the weights are determined by a similarity metric that compares the patch at that location with the patches in the neighborhood. Specifically in our algorithm which is detailed in the **Approach Section**, we iterate over each pixel in the input image and, for each pixel, search for similar patches in a larger search window around the pixel, where the patch similarity is defined as the Euclidean distance between the pixel RGB values. Once similar patches are found, we calculate a weight for each patch as $exp(-patchDistance/(h * h))$ where h represents the smoothing factor. The weights are then used to average the pixel values of similar patches, which gives a denoised estimate of the original patch.

1. **Key Data Structures**: Two-dimensional arrays of unsigned char pixels representing the original image and resulting denoised image constructed using the CImg library [5].
2. **Key Operations**: Calculating Euclidean distances between neighboring pixels within windows and updating weights during iterations over the pixels.
3. **Inputs**: The original image, the denoising smoothing factor, the size of the patch window, and the size of the search window.
4. **Outputs**: The resulting denoised image with the same dimension as the original image.

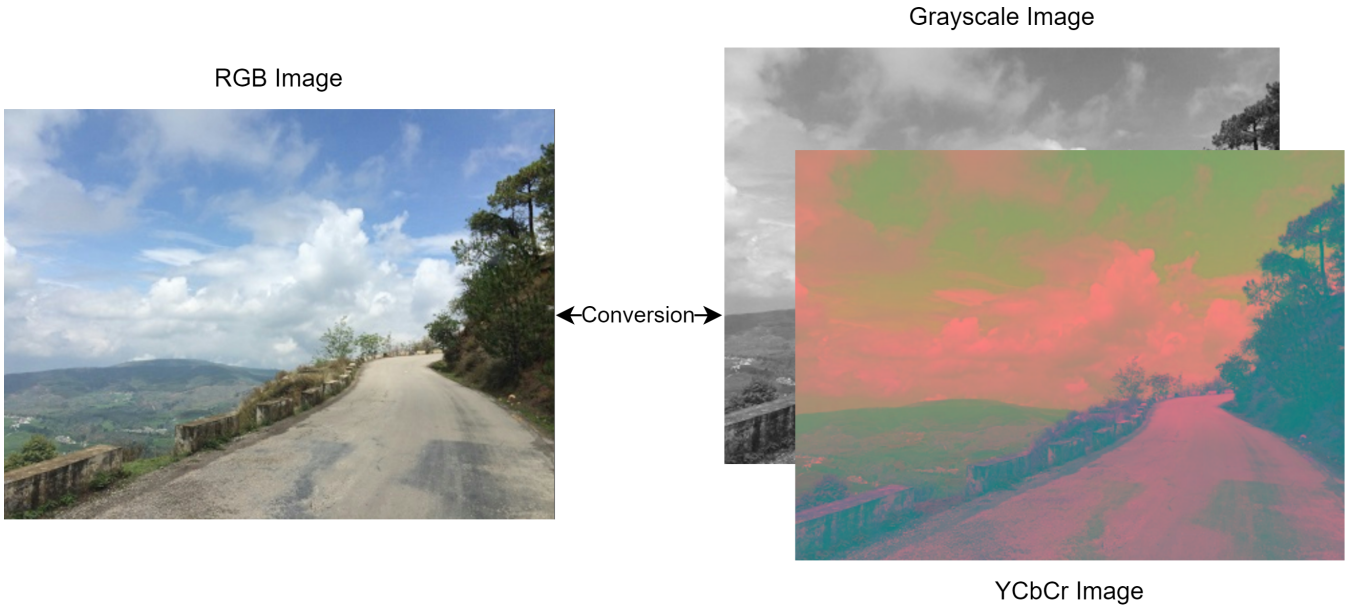

Figure 3. An example demonstrating the color conversion tool, using YCbCR encoding representations

5. **Computationally Expensive Portions**: the double nested loop that iterates over each pixel in the search window and patch window.

1.3 Color Enhancement

Color enhancement is an image processing technique that focuses on improving the visual quality of an image by enhancing its color attributes, in order to increase the image's visual appeal and make it more visually engaging. As illustrated in Figure 1, popular color enhancement techniques involve adjusting various attributes such as brightness, contrast, saturation, and hue, to bring out the image's desired color tones. In our project, we consider two color enhancement algorithms: **Histogram Equalization** [3] and **Contrast Limited Adaptive Histogram Equalization (CLAHE)** [2] and their corresponding parallelized versions for enhancing the contrast of colors and making darker images brighter and with better saturation.

The first algorithm, **Histogram Equalization**, enhances the visual quality of an image by redistributing the pixel intensities. In our implementation, we first creat histograms for each color channel of the input image using 256 bins (for 256 color units). We then compute the average histogram by taking the average of the histograms for the three color channels, calculate the cumulative histogram using the average histogram, and finally apply histogram equalization to each color channel of the input image by mapping the original intensity values to the corresponding values in the cumulative histogram.

1. **Key Data Structures**: Two-dimensional arrays of unsigned char pixels representing the original image and resulting denoised image constructed using the CImg library [5]. A 1D array of unsigned integer to store histogram values for each color channel. A 1D array of floats to store the cumulative histogram values.
2. **Key Operations**: Calculating color histogram for each channel (increment the histogram count for the corresponding color channel) per channel. Calculate the average histogram by summing the histogram counts of each color channel and divide by the total number of pixels. Compute cumulative histogram values, normalize, and apply to color channels of each pixel.
3. **Inputs**: The original image.
4. **Outputs**: The resulting enhanced image with the same dimension as the original image.
5. **Computationally Expensive Portions**: The algorithm is highly data-parallel, as each pixel can be processed independently. There are no explicit dependencies between pixels, except for the histogram and cumulative/normalization calculation stages.

The second algorithm, **CLAHE**, aims to improve the color contrast of images by transforming the intensity distribution of the image's pixels, similar to the previous histogram equalization method, but dividing image into regions and applying equalization individually. In our implementation of the algorithm, we expanded the previous histogram equalization code and segmented the image into grids based on the input hyperparameter of *gridSize*. For each small grid region,

Denoising - Median Filtering

```
Input: orig (original image), res (resulting image), k (kernel size), perc (percentage scaling)
Output: res (resulting image) -> by reference

1. Set mid = (k - 1) / 2
2. Set res size to: (orig.width() - 2 * mid, orig.height() - 2 * mid), since we loose resolution
3. For each pixel in image (excluding padding):
4.      Create empty vector of rgb values called rgbNeighbors
5.      Create rgbAccumulate initialized to 0
6.      For each pixel within the filter range of the current pixel:
7.          Append the RGB value to rgbNeighbors
8.      Find the median pixel value within rgbNeighbors (main sorting happens here)
9.      Set n_neighbors to k^2 / 2 times the percentage scaling (filter size * the percentage scaling)
10.     For each color (R, G, B):
11.         Sort rgbNeighbors based on current color
12.         Sum the n_neighbors pixel values around the median value for current color
13.     Divide each accumulated color value by 2*n_neighbors+1 to get the average
14.     Assign the averaged RGB value to the corresponding pixel in res
```

Figure 4. Pseudocode for the serial implementation of Median Filtering algorithm.

we first calculate a local histogram for each color channel (the same way as the previous algorithm). We then compute a cumulative histogram for each channel and average them to get the final cumulative histogram. The equalized value for each pixel in each color channel is calculated by first finding its corresponding value in the cumulative histogram, and then applying a linear transformation that maps to the range [0,255].

1. **Key Data Structures**: Two-dimensional arrays of unsigned char pixels representing the original image and resulting denoised image constructed using the CImg library [5]. A 1D array of unsigned integers to store histogram values for each color channel. A 1D array of floats to store the cumulative histogram values.
2. **Key Operations**: Calculating color histogram for each channel (increment the histogram count for the corresponding color channel) per channel. Calculate the average histogram by summing the histogram counts of each color channel and dividing by the total number of pixels. Compute cumulative histogram values, normalize, and apply them to color channels of each pixel. The mentioned operations are done for each grid cell within the segmented image.
3. **Inputs**: The original image. A hyperparameter of *clipLimit* to limit the contrast enhancement factor and tune the resulting image to prevent over-exposure. Grid size parameter *gridSize*.
4. **Outputs**: The resulting enhanced image with the same dimension as the original image.
5. **Computationally Expensive Portions**: Similar to the prior algorithm, there is a significant amount of parallelism in this program since each pixel can be processed independently. Additionally, the computation for each pixel is data-parallel, meaning that the same operations are performed on each pixel in parallel.

1.4 Color Conversion

Finally, we also included a color conversion tool inside our library. This serves as an addition to our previous project proposal since the algorithm is less convoluted than the previous functionalities, is easier to decouple during parallelisation, and can demonstrate certain limitations of abusing parallelism. More specifically in our algorithm, we implemented the functionality of converting an RGB image to a YCbCR image. YCbCR is a special type of image encoding method used predominantly in video compression, where Y represents the brightness of the image, and Cb and Cr represent the color information of the image. During implementation, we strictly followed the official definition of RGB to YCbCR conversion formulae (shown in the pseudocode of the **Approach Section**) and performed pixel-level scaling of RGB color channels. The algorithm itself runs extremely fast even in serial mode due to the simplicity and arithmetic intensity and a lack of loop and iterations. In this case, we performed both CUDA on GPU and OpenMP parallelism to explore how much parallel computing can improve or hurt the performance of these relatively trivial image-processing algorithms.

Denoising - NLM Filtering

```
Input: image (original image), res (resulting image) -> by reference, h (parameter for Non-Local Means denoising), patchSize (size
of the patch window), searchWindowSize (size of the search window)
Output: res (resulting image) -> by reference

1. Calculate halfPatchSize = patchSize / 2 and halfSearchWindowSize = searchWindowSize / 2
2. Set res = image
3. For each pixel (x,y) in the image:
4.      Initialize rgbAccumulate to (0,0,0) and weightSum to 0
5.      For each pixel (i,j) in the search window around (x,y):
6.          If (i,j) is within the bounds of the image:
7.              Initialize rgbPatch to (0,0,0)
8.              For each pixel (k,l) in the patch window around (i,j):
9.                  If (k,l) is within the bounds of the image:
10.                     Calculate rDistance, gDistance, and bDistance between the patch window pixel and the search window pixel
11.                     Calculate patchDistance = rDistance + gDistance + bDistance
12.                     Calculate weight = exp(-patchDistance / (h * h))
13.                     Update rgbPatch += (image(k,l,0), image(k,l,1), image(k,l,2)) * weight
14.                     Update weightSum += weight
15.             Update rgbAccumulate += rgbPatch
16.     Calculate the final pixel values for (x,y) by averaging the accumulated RGB values
```

Figure 5. Pseudocode for the serial implementation of NLM Filtering algorithm.

Enhancement - Histogram Equalization

```
Input: img (original image), res (resulting image) -> by reference
Output: res (resulting image) -> by reference

1. Set res equal to img
2. Set width equal to img.width()
3. Set height equal to img.height()
4. Set bins equal to 256 (for color channels)
5. Create empty histogram for each color channel: hist_r, hist_g, hist_b
6. Set each element of hist_r, hist_g, and hist_b to 0
7. For each pixel (x, y) in img:
8.      Increment the corresponding bin in hist_r, hist_g, and hist_b by 1
9.      Compute the average histogram, hist_avg, by averaging the corresponding values in hist_r, hist_g, and hist_b
10.     Create an empty cumulative histogram, cum_hist
11.     Set the first element of cum_hist to the first element of hist_avg
12. For each element in hist_avg (starting at index 1):
13.     Set the corresponding element in cum_hist to the sum of the current element in hist_avg and the previous element in cum_hist
14. For each pixel (x, y) in img:
15.     Get the R, G, and B values of the pixel
16.     Set the R, G, and B values of the corresponding pixel in res to the corresponding value in cum_hist scaled to the range [0, 255]
```

Figure 6. Pseudocode for the serial implementation of Histogram Equalization color enhancing algorithm.

2 Approach

2.1 Technologies

Our choice of **C++** as our primary programming language is well-suited to the parallel programming methodologies that we used. Additionally, C++ is a low-level language that allows us to optimize our code for memory access and cache locality, which are important considerations in image processing. We have run our code on both CPU and GPU architectures on Linux, which allowed us to explore different levels of parallelism depending on the task at hand. To aid in our development and testing, we have also used the GHC machines from CMU, which provide a convenient environment for debugging and running sanity checks. Finally, we also had access to Bridges PSC supercomputer, which allowed us to perform extensive profiling and optimization of our code.

During development, we decided to leverage the parallel programming knowledge gained throughout the course to dissect the different problems of the 6 algorithms for the 3 image-processing functionalities of our library. Specifically, we utilized **OpenMP** and **CUDA on GPU** as our major parallelism tools to boost the speed and performance of our algorithms, where the prior tool leverages shared memory programming paradigm and the latter one uses the distributed memory programming paradigm.

Enhancement - CLAHE

```
Input: img (original image), res (resulting image) -> by reference, clipLimit (integer), gridSize (integer)
Output: res (resulting image) -> by reference

1. Set res equal to img
2. Set width equal to img.width()
3. Set height equal to img.height()
4. Set bins equal to 256 (for color channels)
5. Calculate max_slope as clipLimit divided by gridSize squared
6. Calculate half_win as gridSize divided by 2
7. For each pixel (x, y) in img:
8.      Set x_min, x_max, y_min, y_max
9.      Create empty histograms for each color channel: local_hist_r, local_hist_g, local_hist_b
10.     For each pixel (i, j) within the range of (x_min, y_min) and (x_max, y_max):
11.          Increment the corresponding bin in local_hist_r, local_hist_g, and local_hist_b by 1
12.     For each element in local_hist_r, local_hist_g, and local_hist_b (starting at index 1):
13.         Set the corresponding element in cum_hist_r/g/b_local
14.     For each color channel c:
15.         Set cum_hist_local to cum_hist_avg_local
16.         For each pixel (x_res, y_res) within the range of (x_min, y_min) and (x_max, y_max):
17.             Set x_orig to the maximum between 0 and the minimum between width - 1 and x_res
18.             Obtain original pixel from the coordinates calculated above
19.             Calculate equalized value, and set the corresponding coordinate pixel of res
```

Figure 7. Pseudocode for the serial implementation of CLAHE color enhancing algorithm.

During implementation, we experimented with both **OpenCV** and **CImg**, and eventually decided to use the latter one. The reason is two-fold: (1) OpenCV provides too powerful functionalities and underlying automatic optimizations that we think negatively impact our own implementation of paralleled optimizations and will undermine the purpose of our project, (2) CImg has a very basic underlying image encoding methodology, where each image can be transformed as an unsigned character pointer, where each pixel has RGB channels so that we can easily manipulate them using OpenMP and CUDA since these are basic pointer and byte manipulations in C++.

The following sections will describe the different algorithms and our serial and parallel implementations for them using the frameworks we have chosen.

2.2 Image Denoising - Median Filtering

2.2.1 Serial Implementation. We first introduce our serial implementation of the **Median Filtering** image denoising tool of the proposed library in the following pseudocode of Figure 4. As shown in the pseudocode, there are a total of three *for* iterative loops, where the first one on the outer layer is pixel-level, while the latter two are of the same level, and are iterating within a much more restricted space (i.e., pixels within a neighboring filter range or of three color channels). The most straightforward optimization would be to parallelize over the pixels since the pixel-level operations are relatively independent of each other. However, we could also utilize parallelism over blocks of pixels and explore the sorting and collocation of neighbouring pixel RGB values to boost the performance.

2.2.2 Parallelization approach - OpenMP. To parallelize the given median filtering algorithm in OpenMP, we first analyzed the sections of the code that took up the most time. Through extensive profiling, we found that the neighbor pixel information collection and sorting steps are the biggest bottlenecks for this algorithm. However, since these steps are not very computationally expensive for small filter sizes, which are the target application use-case for denoising, we decided to parallelize the algorithm using a simple approach of parallelizing over small blocks of pixels per OpenMP thread. We implemented this approach by using *#pragma omp parallel for* directives over the pixel iteration loops. Specifically, we parallelized over the outer loop that iterates over the rows of pixels and the inner loop that iterates over the columns of pixels. To avoid race conditions, we used the *schedule* clause to assign each OpenMP thread a small block of pixels to process dynamically. The block size was set to eight pixels, which worked well in our tests, but this value can be adjusted depending on the target hardware and image size. Within each thread, we processed each pixel independently by collecting the neighbor pixel information and sorting the pixel values to find the median. We then

calculated the average value of the pixels within the filter range and assigned the result to the corresponding pixel in the output image.

2.2.3 Parallelization approach - CUDA. For the CUDA implementation, we used a kernel that performs the median filtering operation for a single pixel. The kernel is defined as *MedianFilterKernel* and takes as input the original image, the resulting image, the width and height of the image, and the percentage of neighbors to consider in the filtering operation. To take advantage of the large number of threads available on a GPU, we parallelize the kernel across all pixels in the image. We calculate the index of the current pixel using the *blockIdx* and *threadIdx* variables and then apply the median filter operation to the pixel. Within the kernel, we first compute the median pixel index for the current pixel's neighborhood, based on the kernel size k (which is defined as 5 in this implementation). We then create a local array to store the R, G, and B values of all neighboring pixels, and sort the R, G, and B values independently. We then compute the average RGB values of neighbors within a specified range, based on the input percentage. Finally, we assign the resulting RGB values to the corresponding pixel in the resulting image. To launch the kernel, we allocate memory on the device for both the original and resulting image, and copy the original image to device memory using *cudaMemcpy*, as we learned from the previous projects. We then set the grid and block sizes based on the image size and kernel size, and call the kernel using *MedianFilterKernel«<gridSize, blockSize»>*. Once the kernel has finished executing, we copy the resulting image from device memory to host memory using *cudaMemcpy*, so that we can save the output.

2.3 Image Denoising - NLM Filtering

2.3.1 Serial Implementation. We first introduce our serial implementation of the **NLM Filtering** image denoising tool of the proposed library in the following pseudocode of Figure 5. As we can see in the pseudocode, the main operations are performed pixel-level, and we need to iterate not only through the search windows neighboring the target pixel to find the means, but also the patch windows to ensure the resolution is not undermined. The calculations and updates are relatively trivial in this case, which motivated our subsequent parallelization approaches.

2.3.2 Parallelization approach - OpenMP. To parallelize the given NLM enhancement algorithm in OpenMP, we first analyzed the code and identified the computationally intensive sections. We determined that the most time-consuming section of the algorithm is the nested loops that iterate over each pixel in the search and patch windows to calculate the weighted average. We decided to parallelize this section by using OpenMP directives to parallelize over the outer loop that iterates over the image rows and the inner loop that iterates over the image columns. We used the *collapse* clause to combine the two loops, and the *schedule* clause to assign small blocks of pixels dynamically to each OpenMP thread. The block size was set to eight pixels, which we determined to be an optimal size for balancing workload and minimizing overhead. Within each thread, we processed each pixel independently by iterating over the pixels in the search and patch windows and calculating the weighted average using the Euclidean distance metric. We used a nested loop structure to iterate over the pixels in the windows (as we did in the serial version), and we checked each pixel's position relative to the image boundaries to ensure that the windows did not extend beyond the image boundaries. We updated the pixel values in the output image with the calculated average values. More specifically, we used the *#pragma omp parallel for* directive to parallelize the outer loop that iterates over the image rows, and we used the *#pragma omp parallel for* directive to parallelize the inner loop that iterates over the image columns. We also used the *schedule* clause to assign small blocks of pixels dynamically to each OpenMP thread. We set the block size to eight pixels for both loops in our base implementation, and treat it as a hyperparameter that can be tuned subsequently to profile performance.

2.3.3 Parallelization approach - CUDA. To parallelize the given NLM enhancement algorithm in CUDA, we first analyzed the code and identified the most computationally intensive sections. We determined that the nested loops that iterate over each pixel in the search and patch windows to calculate the weighted average are the most time-consuming section of the algorithm. We decided to parallelize this section by launching multiple threads to process different pixels simultaneously. We first allocated memory on the device using *cudaMalloc* to store the input image and the output result image. Then, we copied the input image from the host memory to the device memory using the *cudaMemcpy*. Next, we defined the block and grid sizes to launch the kernel function *«<gridSize, blockSize»>*. We used a two-dimensional block size of 4x4 threads (hyperparameter) to process each pixel and calculated the grid size such that each block would process a portion of the image that does not overlap with the other blocks. Each thread processes a single pixel in the search window and calculates the weighted average of the pixels in the patch window. The resulting pixel values are stored in a shared memory array, and the weighted average is computed in parallel using atomic operations to update the final pixel value.

In summary, the outermost loop in the kernel function is executed by the GPU grid, where each grid block corresponds to a block of threads; the image pixels are mapped to threads, the thread blocks are mapped to GPU multiprocessors, and the GPU grid is mapped to the entire GPU device. The block sizes and corresponding grid sizes are therefore important as they define the amount of pixel-level parallelism along with GPU resources.

2.4 Color Enhancement - Histogram Equalization

2.4.1 Serial Implementation. We first introduce our serial implementation of the **Histogram Equalization** color enhancement tool of the proposed library in the following pseudocode of Figure 6. As shown in the pseudocode, the algorithm itself has the characteristic where all the iterative loops are on the same level (outermost level) without any nested ones. This means that we can greatly leverage OpenMP's parallelism to optimize each of these loops individually. Furthermore, as detailed in the subsequent sections, we can also create separate kernel functions for each iterative loop and parallelize them individually by launching individual kernel modules with CUDA on GPU.

2.4.2 Parallelization approach - OpenMP. To parallelize the histogram equalization algorithm in OpenMP, we analyzed the code and identified the computationally intensive sections. We determined that the most time-consuming section is the nested loops that iterate over each pixel in the image to calculate the histograms and apply the histogram equalization. As mentioned earlier, the for loops are not nested and are all on the outer level so we can optimize them individually without undermining each other. We decided to parallelize these sections by using OpenMP directives to parallelize the outer loops that iterate over the image rows and columns. We used the *#pragma omp parallel for* directive to parallelize the outer loops that iterate over the image rows and columns separately. We used the *schedule* clause to assign small blocks of pixels dynamically to each OpenMP thread. We set the block size to 8 pixels for both loops, which we determined to be an optimal size for balancing workload and minimizing overhead, as mentioned above. We also used the *SIMD* clause to vectorize the histogram calculation loop for further performance optimization. Finally, we used the *dynamic* schedule to distribute the workload dynamically among threads based on the number of available threads and the amount of work remaining. This allowed us to strive for achieving good load balancing and efficient use of the resources.

For highlighting, we utilized a SIMD (Single Instruction Multiple Data) approach, with multiple elements of the array processed in parallel with a single instruction (*pragma omp parallel for simd*), resulting in better performance during the histogram average calculations. We also used *schedule(dynamic, 64)* clause to specify that loop iterations should be dynamically assigned to threads in chunks of 64 iterations. Each thread executing in parallel would process a chunk of data (i.e., a subset of image rows or columns in our scenario) independently of other threads. As we configured the scheduling method to be *dynamic*, the OpenMP runtime system automatically divides the work among threads and maps them to cores or threads available on the machine. Therefore, the data structures used in this implementation, such as CImg represented as an unsigned char array pointer, were accessed by multiple threads concurrently.

2.4.3 Parallelization approach - CUDA. As mentioned above, the for loops are not nested and are all on the outer level. Therefore, we find it helpful to define individual kernel functions for each of the loops so that we have better modularity and more flexibility during the tuning of grid sizes. Specifically, the histogram equalization algorithm is divided into three parts: computing the histogram, computing the cumulative histogram, and applying the equalization. These three parts are implemented as three separate CUDA kernels. The *histogramKernel()* function calculates the histogram for each color channel of the input image. Since the histogram calculation involves multiple pixels, the computation is parallelized across threads in the GPU. Each thread is responsible for processing a subset of the image pixels, and the results are combined using atomic operations to ensure correctness. Similarly, the *equalizationKernel()* function applies histogram equalization to each color channel of the input image. The computation is also parallelized across threads in the GPU, with each thread processing a subset of the image pixels. The result of the histogram equalization is then written back to the device memory. Overall, the data structures used in the implementation (unsigned char arrays for the input and output images, and int and float arrays for the histograms) are mapped to the GPU architecture, where the parallelism of the problem is exploited using the thread-level parallelism of the GPU, with each thread computing a single histogram bin or pixel value.

2.5 Color Enhancement - CLAHE

2.5.1 Serial Implementation. We first introduce our serial implementation of the **CLAHE** color enhancement tool of the proposed library in the following pseudocode of Figure 7. As shown in the pseudocode, the outermost loop is iterating through all the pixels of the images, with a total of four loops nested below, three of which are of the second level and one within the third level. Due to such intertwined looping nature, we explored the parallelism of the outermost pixel-level iterations, as detailed below.

2.5.2 Parallelization approach - OpenMP. To parallelize the CLAHE algorithm, we first analyzed the code to identify the most time-consuming sections. We found that the nested loops that iterate over each pixel in the image to calculate the histograms and apply the histogram equalization are the most computationally intensive sections. We parallelized these sections by using OpenMP directives to parallelize the outer loops that iterate over the image rows and columns. Specifically, we used the *#pragma omp parallel for* directive to parallelize the outer loops that iterate over the image rows and columns separately. We also used the *schedule* clause to assign small blocks of pixels dynamically to each OpenMP thread. We set the block size to 8 pixels for both loops, which

we determined to be an optimal size for balancing workload. To further optimize the code, we declared the image dimensions, the number of channels, and the number of histogram bins as constant variables outside the parallelized sections. We also precomputed some variables that remain constant for each window, such as the maximum allowed slope and the half-window size for the local histograms, outside of the OpenMP code. For the mapping of parallelism, each thread operates on a different set of pixels within the image. Within each thread, the algorithm performs local histogram equalization on small windows of the image in parallel. The size of the windows is determined by the *gridSize* parameter, which we treat as the hyperparameter. Each thread computes the histogram and cumulative histogram of the pixel intensities in its assigned window for each color channel separately. The cumulative histograms are then averaged across the three color channels to create a single histogram for the window. The equalized value for each pixel in each color channel is then calculated using the cumulative histogram and the maximum allowed slope.

2.5.3 Parallelization approach - CUDA.

The kernel function that we defined over the pixel-level iterative loop of the serial implementation uses two nested loops to iterate over each tile in the image. For each tile, the function computes the local histogram of pixel values and then computes the cumulative histogram. The cumulative histogram is used to calculate an equalized value for each pixel in the tile. The function then copies the equalized values into the output image. The kernel uses CUDA's thread blocks and grids to distribute the work among the GPU threads. Each thread is responsible for processing one pixel in the image. The *blockDims* and *gridDims* variables determine the number of threads per block and the number of blocks in the grid, which we treat as hyperparameters.

2.6 Color Conversion

We present the pseudocode for the **Color Conversion** tool of our library in Figure 8. This algorithm performs RGB to YCbCr color conversion (and vice versa, which we omit since it is just a reversed scaling of the color channels). Although this algorithm is relatively simple compared to the other functionalities in our library, it serves as a valuable benchmark for evaluating the performance of our parallelization approaches with a less computationally intensive scenario. To parallelize this algorithm, we use the *#pragma omp parallel for* directive with OpenMP to parallelize the outer loops that iterate over the image rows and columns separately. For GPU acceleration, we parallelize each pixel's RGB channel transformation using CUDA by mapping the kernel threads to each pixel's execution.

Conversion

```
1. Set res equal to img
2. For each pixel (x, y) in img:
3.      res R = 0.229 * orig R + 0.587 * orig G + 0.114 * orig B
4.      res G = -0.168736 * orig R - 0.331264 * orig G + 0.5 * orig B + 128
5.      res B = 0.5 * orig R - 0.418688 * orig G - 0.081312 * orig B + 128
```

Figure 8. Pseudocode for the serial implementation of RGB to YCbCr color conversion algorithm.

2.7 Iterative Development Process

During the development stage, we encountered a number of issues that required us to make key decisions in order to facilitate our progress. These decisions are documented in this section. Our development process began with an analysis of which image representation tool to use, and we considered both the **CImg** and **OpenCV** libraries, both of which are widely used for image processing in C++. After exploring the **OpenCV** library, we found that it was too powerful for our project, as it provided not only image encoding for all image types but also sophisticated optimization and image processing functionalities that were well-documented. Choosing **OpenCV** would have undermined our purpose and made analysis and performance profiling more complicated. Instead, we decided to use the **CImg** library, which encodes images in a raw and crude way, with colored images represented as unsigned character sets and RGB values arranged for each pixel in a fixed pattern. This raw representation allowed us to explore pixel-level parallelism by directly casting the images into character array pointers, passing them into our functions, and accessing and modifying the values within the arrays in a parallelized manner to achieve the desired processing effect on the resulting image.

During the development of the CUDA implementation, we encountered a major issue where the output of the image only had RGB values separately, resulting in cells with monotonic colors and no diversity. This issue was difficult to debug, as we were implementing parallelism over the character array representing the images. Eventually, we discovered that the issue was due to the alignment of the RGB values for each pixel, as indicated in Figure 9. We had assumed that they were aligned in an interleaving way, but they were actually aligned in a block fashion, with all the R values for all pixels in the same region of the array, followed by the G and B values. This had not been an issue during the serial version, but it hindered our progress in developing the CUDA implementation. Despite this setback, we were able to solve the issue and revise our parallelism over the RGB channels of pixels accordingly, resulting in correct image results. Through this process, we learned the importance of gaining a better understanding of underlying data structure alignments and memory ordering to facilitate successful parallel programming.

```
// CImg Alignment Reference: https://cimg.eu/reference/storage.html
__global__ void kernel(){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height)
        return;

    // What we thought the access is:
    int offset = (y * width + x) * 3
    int r = img[offset];
    int g = img[offset + 1];
    int b = img[offset + 2];

    // What the access should be:
    int offset = y * width + x;
    int r = img[offset];
    int g = img[offset + width * height];
    int b = img[offset + 2 * width * height];
}
```

Figure 9. CImg RGB channel alignment of pixels, on CUDA parallel implementation.

3 Results

We have evaluated the implemented algorithms using various approaches to provide a comprehensive analysis. Initially, we present the results of the base comparison of the algorithms over their implementations on the serial C++ framework, the OpenMP framework, and the CUDA framework. Next, we investigate the scaling behavior of the OpenMP implementations by varying the number of threads launched for each algorithm. Furthermore, we perform an in-depth analysis of how the algorithms' parameters affect their performance, specifically for three of our parametric algorithms. Finally, we analyze the impact of the problem size on our implementations' behavior by presenting results for the algorithms on SD, HD, UHD, and 4K images with fixed parameters.

3.1 Comparison of Frameworks

We compare the performance of the algorithm implementations across the frameworks normalized in Figure 10.

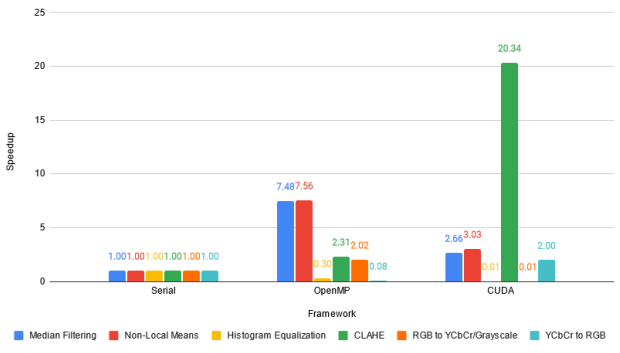

Figure 10. Relative speedup of algorithms across frameworks (SD images).

Based on the plot above, we observed several trends. Note that the execution here is shown for a Standard Definition (SD) image, and the effects of scaling are analyzed separately.

- The algorithm with the lowest amount of computation, i.e., histogram equalization, is slowed down by both parallel frameworks, indicating that the serial implementation is optimal.
- Color conversion algorithms, which also have low amounts of computation, only see slight benefits from parallelization. The conversion to YCbCr performs better on OpenMP compared to CUDA, while the opposite applies to the conversion to RGB. It's difficult to determine the reason for this discrepancy (we assume it to be the difference in communication cost of transferring the image on the GPU from the CImg format for RGB and YCbCr, and the thread overhead for the same in the OpenMP case), but they only speed up the execution by a very low amount (2x) compared to the serial version due to the low amount of compute required.
- Algorithms with significant amounts of computation heavily benefit from parallelization, with large speedups observed. The denoising algorithms show better performance on the OpenMP implementation than the CUDA implementation, while CLAHE does better on the CUDA implementation. The reason for this is the amount of work we had chosen for the denoising algorithm through parameter scaling is much lower compared to CLAHE, and the additional work required in CLAHE is enough to amortize the communication overhead on the GPU.

3.2 Thread Scaling for OpenMP Implementations

Figure 11 shows a plot depicting how the OpenMP implementations of the six algorithms scale with the number of threads launched. The values are normalized to their single-thread performance, and the plot can be interpreted as the inverse of the speedup obtained by scaling the number of threads.

After analyzing the performance of all the algorithms in Figure 11, we observe that parallel execution using OpenMP provides a speedup for all the algorithms, with the denoising algorithms and CLAHE obtaining the highest speedup due to their high computational complexity. However, we notice two anomalies in the results. First, the performance scaling from 8 to 16 threads does not provide the best speedup, which we attribute to the extra threads being hyperthreads rather than true threads running on individual cores in our system. Second, the CLAHE algorithm experiences a slowdown when moving from one to two threads, possibly due to bad cache access patterns resulting in frequent evictions, making the performance memory-bound. It is noteworthy that these observations are based on normalized values with respect

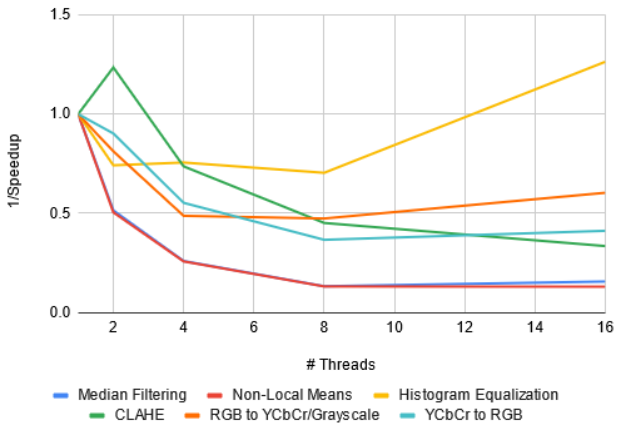

Figure 11. Scaling for OpenMP implementations with respect to the number of launched threads.

to single-thread performance, gauged as the inverse of the speedup obtained through scaling the number of threads, as presented in Figure 11.

3.3 Parameter Scaling

We present the results of how the parametric algorithms we implemented scale with their parameters for the various implementations. This could help us determine the best implementation for different values of the parameters.

3.3.1 Median Filtering. Median Filtering has two parameters that could affect the execution time of the algorithm - the filter/patch size considered for taking the median, and the percentage of neighbor pixels to be considered in the median calculation.

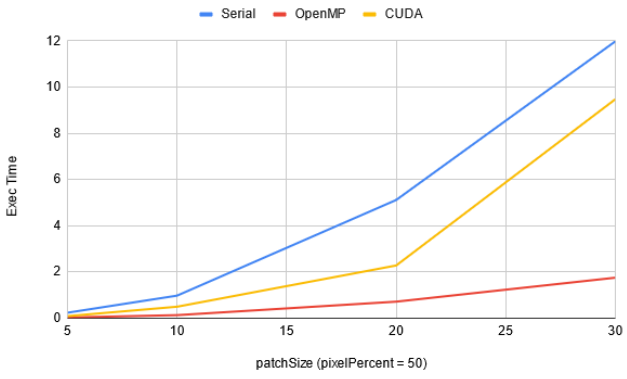

Figure 12. Execution Time Scaling for Median Filtering with Patch/Filter Size.

The plot in Figure 12 illustrates how the execution time of the algorithm changes as the filter size increases. It is observed that for most algorithms, the execution time increases superlinearly as the patch size increases. Interestingly, the OpenMP implementation continues to perform the best, with the best scaling observed with respect to the patch size. This is because the naive implementation of the OpenMP algorithm does not encounter any issues with pixel-level parallelization, and the tested filter sizes follow a good cache access pattern with interleaved thread execution. On the other hand, the CUDA implementation does not scale as well because the number of global memory accesses increases with the number of neighbors considered, which leads to increased execution time.

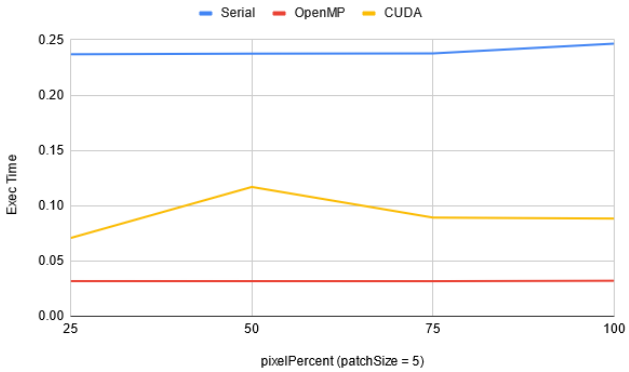

Figure 13. Execution Time Scaling for Median Filtering with Percentage of Neighbor Pixels considered.

Figure 13 displays the algorithm's performance with respect to the percentage of pixels considered. Since small filter sizes are optimal for our image, the percentage of pixels considered has a negligible effect on execution time across all frameworks. As a result, the OpenMP implementation exhibits the best scaling overall, making it the preferred choice for Median Filtering when considering parameter scaling.

3.3.2 Non-Local Means. Non-local Means algorithm has three parameters, but we focus our analysis on the scaling of two parameters and ignore the smoothing parameter as it only affects the mean scaling and would not affect execution time heavily.

Figure 14 depicts the scaling of the NLM algorithm with respect to the filter size considered for calculating the mean. Similar to Median Filtering, the execution time increases for all frameworks as the filter size increases. However, the OpenMP implementation shows the best performance, with the best scaling with respect to the filter size. This is because our implementation benefits from pixel-level parallelization, and the filter sizes tested conform to a good cache access pattern with interleaved thread execution. Notably, the scaling of the CUDA implementation is better than in the case of Median Filtering due to the more involved work in this algorithm, which amortizes the cost of communication to the

Figure 14. Execution Time Scaling for Non-Local Means with Patch Size.

GPU. Therefore, for denoising tasks with parameter scaling, OpenMP is considered by us the optimal choice.

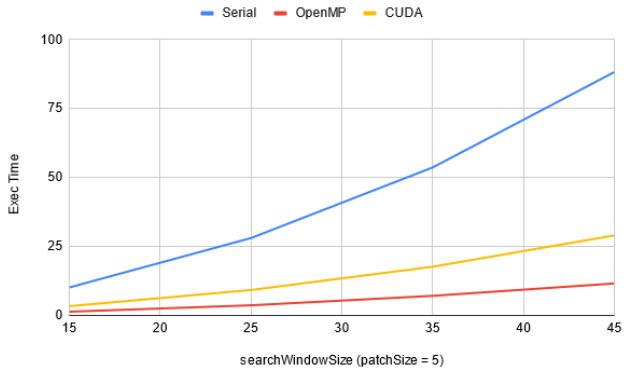

Figure 15. Execution Time Scaling for Non-Local Means with Search Window Size.

Figure 15 illustrates the algorithm's scalability in relation to the size of the search window. Our analysis reveals that the algorithm's scalability is comparable to its scalability concerning the *patchSize* but with a less pronounced impact. Furthermore, the observed trends remain consistent with those described earlier.

3.3.3 CLAHE. The performance of the CLAHE algorithm is influenced by two parameters, namely, the clip limit for the histogram values and the grid size for dividing the image into blocks. In this study, we investigate the impact of both these parameters on the algorithm's execution time and present the scaling results for each parameter.

In Figure 16, we present the scalability results for the CLAHE algorithm in relation to the grid size, with a fixed clipLimit value, for the three frameworks used in this study. The results show a slightly superlinear increase in execution time with grid size, similar to the trend observed for

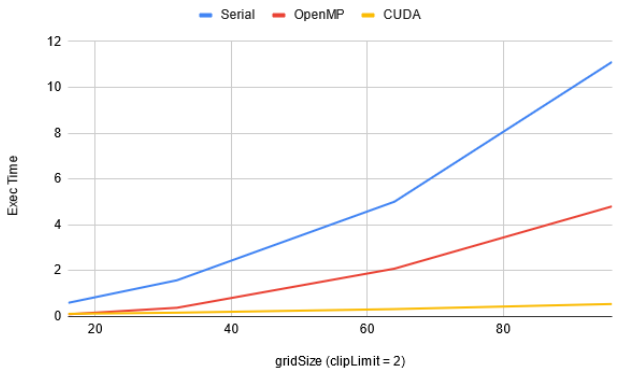

Figure 16. Execution Time Scaling for CLAHE with Grid Size.

the *patchSize* in denoising algorithms. Notably, the CUDA implementation outperforms the other frameworks for all grid sizes, with almost constant execution time as the grid size increases. This can be attributed to the efficient cache access pattern within the CLAHE algorithm and the ample availability of CUDA cores for performing the blocked execution. In contrast, the blocked pixel division in OpenMP results in more context switches due to the larger amount of compute per pixel block that cannot be parallelized with the limited number of threads running in parallel.

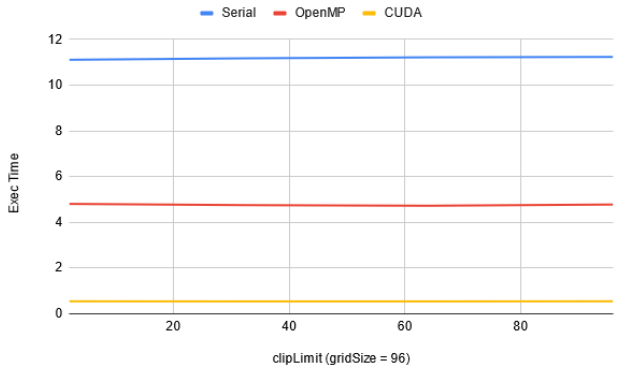

Figure 17. Execution Time Scaling for CLAHE with Clip Limit.

Figure 17 illustrates the scalability results of the CLAHE algorithm with respect to the clip limit, with a fixed grid size. Surprisingly, we observed no discernible impact of the clip limit on the algorithm's execution time, owing to our implementation's unique optimization method. Specifically, the value clipping limit is optimized via a slope calculation deferred until the end, rendering it independent of the clip limit and increasing its dependence on the grid size. Once again, the CUDA implementation outperformed the other

frameworks and showed similar scalability with the clip limit parameter.

3.4 Problem Size Scaling

This section showcases the Relative Speedup plots for the three algorithms on HD images, Ultra HD images, and 4K images. We aim to identify the frameworks that exhibit the best scalability as the problem size increases.

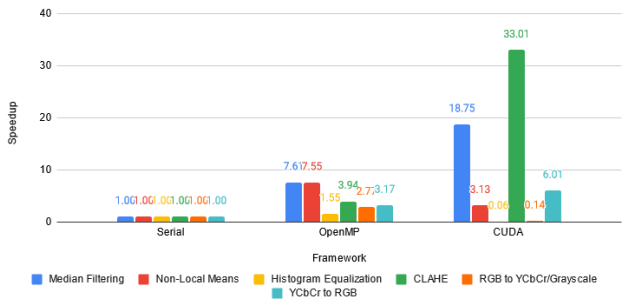

Figure 18. Relative speedup of algorithms across frameworks (HD images).

Figure 18 displays the speedup plot for the six algorithms on HD images, showcasing the performance of different frameworks. While the general trend remains consistent, the difference in speedup between the two best-performing frameworks widens. Notably, two observations stand out in this plot. Firstly, the median filtering algorithm benefits from GPU usage, as the CUDA implementation surpasses the OpenMP implementation for HD images. Secondly, the algorithms with low compute requirements, such as histogram equalization and color conversion, exhibit noticeable speedup gains through parallelization, with one of the parallel implementations surpassing the serial implementation.

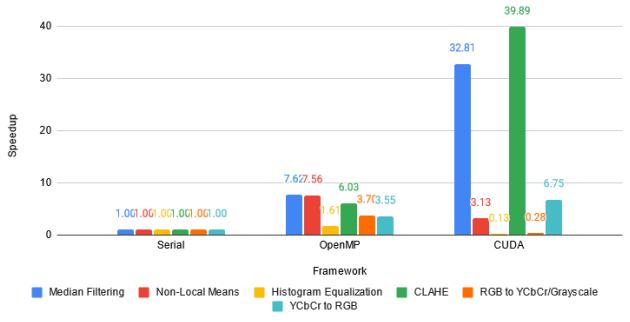

Figure 19. Relative speedup of algorithms across frameworks (UHD images).

Figure 19 presents the speedup plot for the six algorithms on Ultra HD images, demonstrating the performance of various frameworks. The trend remains consistent, with only slight differences. As noted earlier, the gaps between speedups are more significant, and the algorithms that typically execute faster benefit more from parallelization.

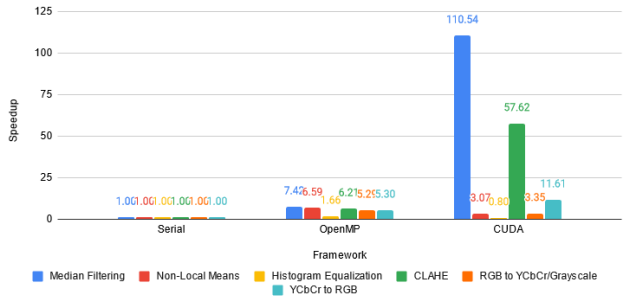

Figure 20. Relative speedup of algorithms across frameworks (4K images).

Figure 20 presents the speedup plot for the six algorithms on 4K images, demonstrating the performance of various frameworks. We observe the same trends as in the previous plots, with the same exceptions.

An analysis of the scaling across the plots reveals that the CUDA implementations become increasingly desirable for every algorithm. It appears that at some point, the CUDA implementation will surpass the OpenMP implementations as image resolution increases. However, for the current image sizes (4K), the OpenMP implementations for the *Non-Local Means*, *Histogram Equalization*, and *RGB to YCbCr/Grayscale* algorithms still perform better than the CUDA implementations.

4 Conclusion

In this study, we performed a comprehensive evaluation of the performance of OpenMP and CUDA frameworks for six commonly used image processing algorithms across various image sizes and parameter settings. Our results showed that the choice of parallelization framework has a significant impact on algorithm execution time, with CUDA implementations performing better for larger image sizes due to better cache access patterns and a higher degree of parallelism. However, we also observed that the OpenMP framework outperforms CUDA for certain algorithms, specifically Non-Local Means.

Moreover, we investigated the scaling of algorithms with respect to the number of threads deployed in the OpenMP framework. Our findings revealed that all algorithms, except for the CLAHE implementation, demonstrated fair speedups. The CLAHE implementation showed poor performance with two-thread execution, which could be attributed to a bad cache access pattern resulting in frequent memory evictions.

We also found that the performance scaling of algorithms varies with image sizes and algorithm parameters. Median Filtering, Non-Local Means, and CLAHE algorithms were found to be sensitive to their patch/grid size/search window

size parameters, while the choice of the parallel framework remains unaffected by such scaling. We also discovered that parallelization provides benefits even for less computationally intensive algorithms like histogram equalization and color conversion for larger image sizes.

In summary, our study highlights the importance of carefully selecting the parallelization framework and algorithm parameters for image processing applications. Consequently, developers can achieve significant performance improvements by selecting the appropriate framework and fine-tuning the algorithm parameters.

5 Future Work

Our study has opened up several directions for future research in the field of parallelization for image processing algorithms. Firstly, it would be useful to investigate the impact of optimizing the distribution of work on the GPU by tuning the grid and block sizes for the algorithms we have implemented. This could provide insights into achieving better performance with GPU parallelization.

Secondly, while our study has identified the best-performing parallelization framework for each algorithm, it is possible that more sophisticated parallelization techniques could yield even better performance. We invite researchers to explore such techniques using our library across different frameworks to identify even better-performing parallel code.

Finally, we believe that extending the analysis of parallelization techniques to other image-processing algorithms that operate on two-dimensional structures could yield interesting results. This could include other image-processing tasks such as object detection and segmentation. We anticipate that such research would benefit both image-processing tasks and parallelization techniques.

Acknowledgments

To Professor Zhihao Jia and Professor Brian Railing for conducting the course, guiding us and addressing our concerns throughout the project.

References

[1] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. 2011. Non-Local Means Denoising. *Image Processing On Line* 1 (2011), 208–212. https://doi.org/10.5201/ipol.2011.bcm_nlm.

[2] Akshansh Mishra. 2021. Contrast Limited Adaptive Histogram Equalization (CLAHE) Approach for Enhancement of the Microstructures of Friction Stir Welded Joints. arXiv:2109.00886 [cs.CV]

[3] Wan Azani Mustafa and Mohamed Mydin M. Abdul Kader. 2018. A Review of Histogram Equalization Techniques in Image Enhancement Application. *Journal of Physics: Conference Series* 1019, 1 (jun 2018), 012026. https://doi.org/10.1088/1742-6596/1019/1/012026

[4] David C. Stone. 1995. Application of median filtering to noisy data. *Canadian Journal of Chemistry* 73, 10 (1995), 1573–1581. https://doi.org/10.1139/v95-195 arXiv:https://doi.org/10.1139/v95-195

[5] David Tschumperlé. 2012. The CImg Library. In *IPOL 2012 Meeting on Image Processing Libraries*. Cachan, France, 4 pp. https://hal.science/hal-00927458

A Contributions and Credit Distribution

The division of work was as follows:

- **Agastya Sampath** - Implementation of the image processing algorithms, setup for the OpenMP framework, naive OpenMP parallelization, profiling the algorithms for extensive parallelization, CUDA implementation of Median Filtering algorithm, contributing to CUDA implementation of Histogram Equalization, recording results and analysis for parallelization, contributing to the Report (Sections 3, 4, 5), revising report (50%)
- **Shijie Bian** - Revising OpenMP implementations for algorithms, profiling the algorithms for parallelization, set up for the CUDA framework, CUDA implementations of all algorithms except Median Filtering, main code for library implementation calling the algorithms, recording image results for algorithm outputs, writing pseudocodes for each algorithm, contributing to the Report (Sections Abstract, 1, 2), revising report (50%)