Parallel Optimization Analysis for Image Processing

Min Yub Song

Abstract

Image processing optimization has become critical with the rise of AI and computer vision applications. This project presents a comparative analysis of image processing operations (grayscale conversion and Gaussian blur) implemented across CPU processing, CUDA GPU implementation, and OpenCV's optimization functions. Using Google Colab's GPU environment, execution times and processing consistency were evaluated. The results demonstrate that CUDA GPU implementation processes images approximately 1000x faster than CPU for blur operations and outperforms OpenCV-GPU by 10x while maintaining superior consistency in processing times across multiple runs.

1 Introduction

Parallel optimization for image processing is one of the algorithms that goes unnoticed most of the time due to its subtle yet readily accessible feature on modern-day devices. Image processing optimization allows the device's hardware to process images and output images in a way that solves everyday tasks ranging from editing photos to enabling computer vision applications in artificial intelligence and machine learning. With the rise of deep learning models and image recognition, object detection, and image generation, efficient image processing has become increasingly critical. Previously, editing images required professional software to achieve desired outputs, but now these capabilities are readily available on mobile devices such as Magic Eraser on Google Photos and Apple's AI. The rapid advancement of GPU computing has created new opportunities for optimizing image processing operations. Current approaches often rely on traditional CPU processing or standard computer vision libraries, which may not fully utilize available hardware capabilities. The key contributions include a comprehensive performance comparison between CPU, GPU, and OpenCV implementations, analyzing processing consistency across multiple runs, and demonstrating significant performance gains through CUDA implementation. Through this direct comparison, practical performance gains that demonstrate the real-world benefits of parallel processing were observed.

2 Background

Parallel optimization is the responsibility of the GPU to compute the arithmetic in parallel as opposed to the serial calculations in the CPU. Modern-day CPUs have multiple cores responsible for carrying out multiple calculations, but they are incomparable to the computing power of GPUs. Because of the hardware architecture difference between a CPU and GPU, the image processing task is well suited for mainly GPU in terms of time and cost. The large number of cores in a GPU with a cache of a limited size allows simultaneous calculations on each pixel. At the same time, the serial CPU is required to perform the calculation first and can only proceed to the next once previous calculations are complete. While it may not significantly matter for more minor calculations requiring a short period of time, the two image processing for this project focuses on two different methods of image processing. First is the grayscale conversion, which uses a trivial equation to convert colored pixels to gray pixels. In contrast, the second, Gaussian Blur, uses convolution to process each pixel. As shown in Figure 1, Convolution uses kernels, also known as filters, to compute the arithmetic by applying a matrix operation that combines the neighboring pixel values according to predefined weights, making it significantly more computationally intensive than grayscale conversion [2].

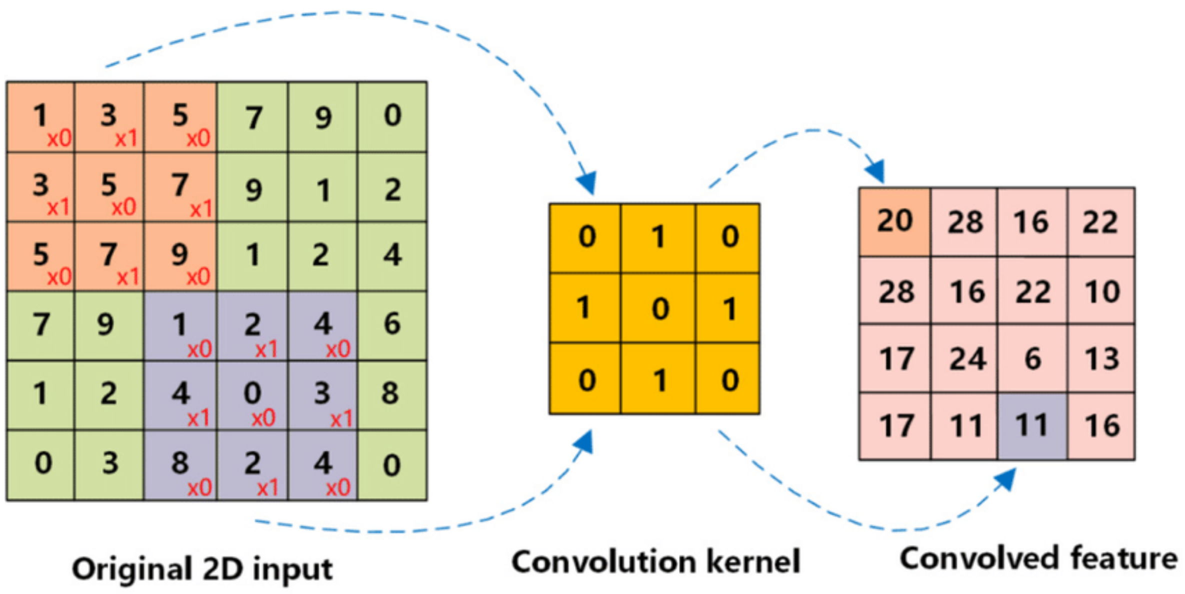


Figure 1: Convolution

2.1 Related Work

Early research in image processing focused primarily on CPU-based implementations due to hardware limitations. However, with advancing technology, images can now be processed by harnessing the computational power of GPUs. This transition allowed new optimization algorithms to be used, particularly in handling memory transfers and kernel operations. Recent studies have demonstrated various approaches to optimizing basic image operations like filter and color adjustment conversions.

With the adoption of GPU computing, image processing capabilities have been applied in many different fields, including computer vision, with a particular focus on optimizing convolution operations for GPU architecture. Studies show how GPU implementations can significantly impact processing speed and efficiency.

2.1.1 OpenCV

OpenCV (Open Source Computer Vision Library) is considered a standard tool in computer vision applications, with its readily optimized implementations for various image processing operations such as face recognition, surveillance, detection and counting of vehicles on highways, object recognition and more. The library's functionality extents to video processing, medical images and robotics applications as well[1]. Studies have evaluated OpenCV's performance measure against GPU implementations for automatic memory management, auto-allocation of the output data, and multi-threading. It also has OpenCV CUDA module to utilize CUDA's computational capabilities. Some limitation is when it comes to having complex algorithms leading to increase in compilation time and code size. With OpenCV, it is possible to use its features to implement CUDA kernels by using image file reading and writing functions directly. These insights provide the trade-offs between the ease-of-use approach and the optimization potential[3].

2.1.2 Comparative Analysis

Beyond OpenCV and direct CUDA implementations, several alternative are available. Intel's Integrated Performance Primitive which offers highly optimized implementations for CPU-based image processing, to compete with the cost of rising CPU and GPU provides as an alternative to optimization and ArrayFire, a high performance library for GPU computing that provides optimized implementations for common image processing operations While theses are some alternatives. Building from these works, my study compares performance across the three approaches. While previous research has often focused on the individual implementations of the three, this work offers a direct yet easy-to-understand comparison between CPU, GPU, and OpenCV implementations, providing practical insights for choosing the most appropriate approach for image processing applications.

3 Design

The experimental set up was designed to compare three different implementations of image processing operations. Using various images of size and color to measure the processing time it took for grayscale conversion and Gaussian blur application. The set ups were CPU implementation, CUDA GPU implementation, and OpenCV implementation. For OpenCV implementation, the code was ran in two different environments, one using the standard CPU within the executing device, and other with Google Colab's T4 GPU environment along with CUDA GPU implementation to offer benchmark for both.

3.1 Algorithm

Two fundamental image processing operations were selected for this project: grayscale conversion and Gaussian blur. While the grayscale offered a relatively straightforward operation, where each pixel's RGB values were converted using a weighted equation:

$$Grayscale = 0.299 * R + 0.587 * G + 0.114 * B$$

(R, G, B is the number for the color components to human perception of brightness. if a pixel has R = 100, G = 150, B = 200, then $Grayscale = 0.299 * 100 + 0.587 * 150 + 0.114 * 200 = 140.75$)

the Gaussian blue involved a more complex kernel convolution varying in kernel size from [3,3] to [41,41]. These operations were chosen to demonstrate the performance difference between parallel and serial image processing. The implementation shown on the code is of [5,5] matrix kernel with applied Gaussian distribution (where the weight is focused more heavily on the center of the matrix, with smaller weights on the outer radius of the matrix) weight normalized by 2048 to prevent altering of the brightness of the resulting image.

$$\frac{1}{2048}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 128 & 192 & 128 & 4 \\ 6 & 192 & 512 & 128 & 6 \\ 4 & 128 & 192 & 128 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

New Pixel Value:

$$\sum_{i=1}^{5} \sum_{j=1}^{5} kernel[i, j] * image[i, j]$$

The grayscale weight calculation would be done on a single pixel and moves on to the next pixel in the image. In contrast, the Gaussian blur will compute the sum of all the products in the kernel to the original image to determine the new output image center pixel. As the kernel size increases, the blurring effect on the image would increase due to the Gaussian distribution. Another method of increasing the blurring effect is to pass the blur multiple times or use equally distributed weights for the kernel, as it will not have the smoothening effect per the Gaussian blur.

3.2 Implementation

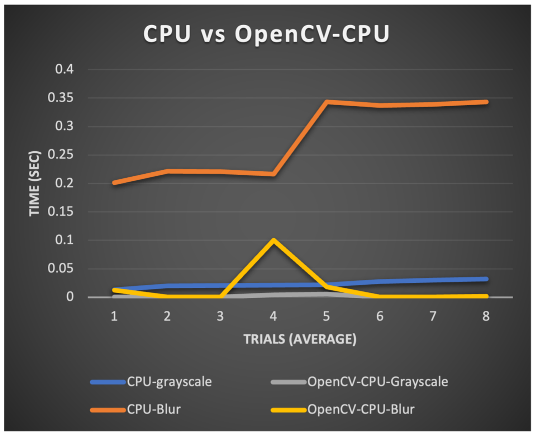
The implementation approach for each method was structured to maximize its respective strengths. The CPU implementation processed pixels sequentially. The GPU implementation utilized the CUDA threads configuration of 16x16 blocks, enabling parallel processing of pixels while optimizing memory access patterns. OpenCV used the built-in functions cv2.cvtColor and cv2.GaussianBlur.

3.3 Methodology

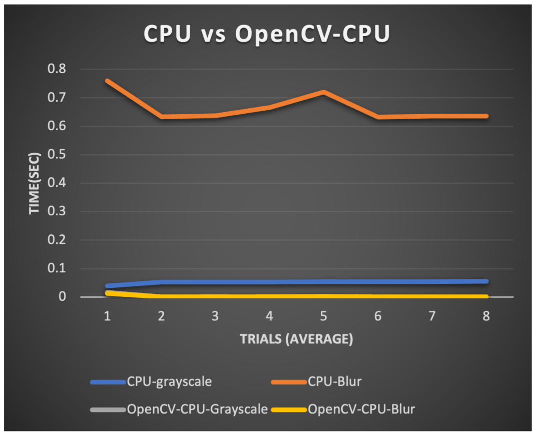
The testing methodology was designed to ensure fair and comprehensive comparison. Different image sizes were used to evaluate how each implementation scales with input size. Each operation was run through 10 trials to establish consistency and reliability of the results. Then, the trimmed mean was used to drop the lowest and highest times as well as outliers, especially for the implementations using CPU for processing due to the inconsistencies of the time it recorded for the given number of trials. For the accuracy of the timing for each processing, the GPU implementation adopted the image reading and writing functions from OpenCV packages, which are not included in the processing time. The performance metrics focused on processing time, result consistency, and memory usage patterns.

4 Evaluation

From the tests, the results from the CUDA GPU implementation was proven to be the fastest and accurate throughout numerous trials. Not only were each time consistent, it showed a great difference compared to the CPU image processing time but also proved that it is the fastest out of the three. From Figure 2, the fluctuations of the CPU implementation resulted from the way modern CPU's handle multitasking, resource allocation, and thermal management. Due to this effect, even by taking the trimmed mean to prevent the inconsistency of the graphs, shows signs of time fluctuating for both the CPU's and OpenCV-CPU's time when processing the sample image. However the contrast between the two is significantly high that the CPU processing time fluctuation can be attributed to the external factors as mentioned previously, whereas the OpenCV-CPU implementation shows relatively lower variability due to its optimized functions and efficient use of CPU resources.



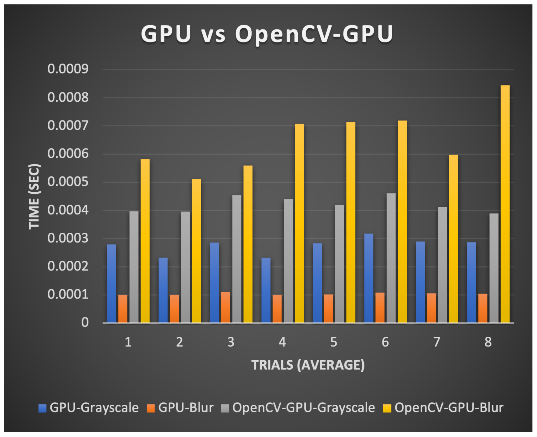
(a) CPU vs OpenCV-CPU for small image



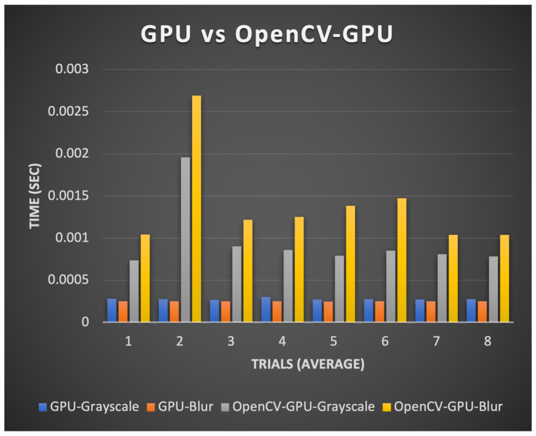
(b) CPU vs OpenCV-CPU for large image

Figure 2: Comparison of CPU vs OpenCV-CPU performance.

As for the CUDA GPU versus OpenCV-GPU, while there were consistency in both tests, CUDA GPU's consistency in processing time were far greater than the processing time for OpenCV-GPU. However the greatest difference between the two processing implementations were from the Gaussian Blur image processing application. In Figure 3, the small image shows the consistency of GPU-Blur where the OpenCV-GPU-Blur shows less consistency, and this trend follows for the large image as well. Not only the processing time was consistent for both GPU-Grayscale and GPU-Blur but it was observed to be at least 4x faster than the OpenCV-GPU with giving 0.000249 average for GPU-Blur, while OpenCV-GPU-Blur observed average was 0.001393, noting both consistency and efficiency of the GPU processing power.



(a) CPU vs OpenCV-CPU for small image



(b) CPU vs OpenCV-CPU for large image

Figure 3: Comparison of CPU vs OpenCV-CPU performance.

4.1 Results

From the tests, it was demonstrated that the CUDA GPU implementation consistently outperformed the CPU and OpenCV-CPU/GPU implementations in terms of both accuracy and speed. Through numerous trials, CUDA GPU showed significant consistency: CUDA GPU showed improvements of approximately 810x faster than CPU implementation, 33x faster than OpenCV-CPU implementation, and 2.75x faster than OpenCV-GPU implementation showing processing power fastest with least amount of fluctuation in time compare to others. This performance superiority is due to GPU's ability to handle parallel computations efficiently, by using the large number of cores to process multiple pixels simultaneously. In contrast, the CPU implementation showed fluctuations in processing time due to the way the CPU handles multitasking, resource allocation, and thermal management on the device. The CPU handles a wide range of tasks concurrently, so the processing power availability is reduced for operations such as image processing to prevent overheating. Even when applying a trimmed mean to smooth out the extreme values, the variability in CPU performance showed that it was not able to compete with the performance of the GPU: CPU showed high variability with standard deviation of ± 0.0047 seconds for grayscale and ± 0.0512 seconds for blur operations. GPU implementation with standard deviations of ± 0.000021 seconds for gray scale and ± 0.000003 seconds for blur operations, indication very high consistency compared to moderate standard deviation from OpenCV-GPU of $\pm 0.0000032, \pm 0.000128$ seconds for grayscale and blur operations respectively. OpenCV implementation showed moderate level of variability in processing time due to the higher optimization compared to the CPU, but the contrast between the GPU and OpenCV was significant. CUDA GPU outperformed OpenCV due to its optimized memory access patten and its ability to efficiently distribute tasks across numerous processing units. Lastly, the notable time difference between GPU's 0.00011 seconds compared to CPU's 12.68 seconds for the Gaussian Blur operation showed significant difference between the two. These attributes makes GPU image processing ideal for handling large scale parallelization, such as Gaussian blurring, where by convolution each pixel is processed independently and simultaneously.

5 Conclusion

Even though the limitation of CPU exists when compared to GPU it is still sufficient to process images, only when the calculation becomes complex such involving convolution into the algorithm will CPU will be significantly slower yet producing the same output image as shown on Figure 4. As advancements continue in GPU technology over time, including the implementation of more optimization algorithms, the performance gap between CPUs and GPUs continues will continue to get wider. While CPUs are also advancing with multi-core architectures and improved instruction sets, GPUs maintain their advantage in parallel processing tasks like image processing due to their fundamental architecture designed for simultaneous parallel computations. The increased performance advantages of GPUs, combined with greater accessibility and affordability, make parallel optimization capabilities more widely available to developers and researchers. This widespread availability of GPU technology not only allows more people to experience high-performance computing but also will drive innovation in parallel processing applications

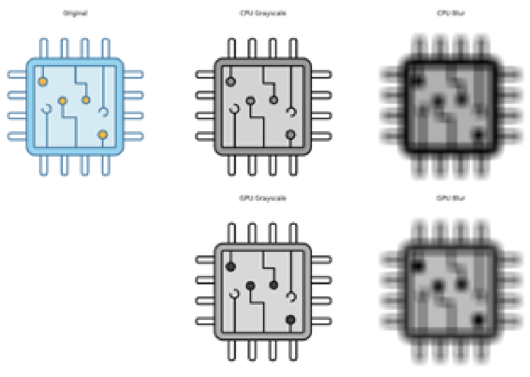


Figure 4: CPU and GPU Gaussian Blur

5.1 Future Work

I plan to explore more ways of image optimization and convolution from this project. I have used image processing software before and feel that as I learn more about the GPU. With the lessons learned from this class and project, I understand more about how my software and applications work. I plan to expand this knowledge of parallel optimization to understand more complex optimization implementations and algorithms. Also, I plan on learning more about memory optimization and scalability so that I can use the GPU to its fullest extent. I hope to learn more about GPUs and how different GPU architectures result in different image processing capabilities and performance trade-offs.

6 Contribution

Conducted research on Convolution and different types of image processing processes. Picked two to show the comparison without delving too deep into the subject since it was intended for someone like me who's new to parallel optimization and to show the real world application of parallel optimization. Implemented codes for CPU/GPU/OpenCV, conducted tests and completed the report.

7 Code

First, I used for the project was named sample.png in my Google Drive, which requires mounting of the drive for Colab Notebook, so it requires the tester to upload a sample.png image on their Google Drive before running the code.

Run second code cell, and third code cell is the CPU grayscaling/blur, so it requires the third, fourth, and fifth for compile and executing the code.

Then running the ninth code cell which is the OpenCV implementation to be done after CPU so that change in runtime isn't necessary later

After the CPU portion has been completed, change the runtime to Python 3, T4 GPU and remount the drive, and run the required cells skipping CPU code cells. Then run sixth, seventh, eighth code cells for the CUDA GPU implementation.

Run ninth code cell again for GPU comparison this time

Final tenth cell to output the images back on to the drive as well as on the notebook itself

References

[1] Introduction to opencv, 2024.

[2] NVIDIA Corporation. Separable convolution. Technical documentation, NVIDIA, 2007.

[3] Gloria Bueno Garcia and Oscar Deniz Suarez. *Learning Image Processing with OpenCV.* Packt Publishing, 2015.