Accelerating Image Convolution: A Comparative Study of CPU and GPU Performance with Python/Numba

Faiyaz Bin Yousuf[1*]

[1*]Department of Computer Science and Engineering, BRAC University, Kha 224 Pragati Sarani, Merul Badda, Dhaka, 1212, Bangladesh.

Corresponding author(s). E-mail(s): faiyaz.bin.yousuf@g.bracu.ac.bd;

Abstract

Two-dimensional convolution is an essential but computationally expensive operation in image processing. As image resolutions increase, traditional sequential CPU processing becomes a significant bottleneck, requiring parallel computing solutions. This paper presents a comprehensive performance and scalability analysis of a Gaussian blur filter to evaluate modern parallel architectures. We implement the algorithm in Python using the Numba Just-In-Time compiler and benchmark it on a sequential single-core CPU, a parallel 16-thread multi-core CPU, and a massively parallel NVIDIA RTX 3070 GPU. The experiment was conducted on four image sizes, from 0.3 to 3.7 megapixels. Our results demonstrate the profound impact of parallelism. The GPU implementation was the fastest in all cases, achieving a maximum speed of approximately 130 times that of the single-core CPU and 24 times that of the fully utilized multi-core CPU. Furthermore, the GPU's performance exhibited superior scalability, with execution time remaining almost constant regardless of image size. We conclude that for data-parallel tasks like image convolution, the GPU's specialized architecture is the most critical factor for achieving high performance, and that frameworks like Python/Numba are effective tools for leveraging this power.

Keywords: High-Performance Computing, GPU Computing, CUDA, Python, Numba, Image Processing, Convolution, Performance Analysis

1 Introduction

The field of image processing is a vital sub-discipline of digital signal processing, containing a wide variety of mathematical and algebraic operations used in countless modern applications. Many of these core operations, such as two-dimensional (2D) convolution, are both memory-intensive and computationally complex. As image resolutions increase and algorithms become more sophisticated, traditional single-threaded solutions on Central Processing Units (CPUs) are often unable to deliver the performance required for real-time or efficient processing. This computational bottleneck has driven a shift in the field from sequential to parallel programming approaches, creating a demand for high-performance computing solutions to handle the increasing complexity and data size inherent in modern computer vision tasks.

The primary solution to this performance challenge is the adoption of parallel computing architectures. Research in this area has consistently shown that Graphics Processing Units (GPUs), with their massively parallel architecture, are particularly well-suited for image processing tasks [1]. Studies have demonstrated that the performance advantage of GPUs over CPUs grows significantly as the input image size increases [2]. For specific algorithms like the Gaussian blur, researchers have compared multi-core CPU and GPU implementations, confirming the GPU's superior performance and speedup [3]. While the performance benefits of GPUs are clear, the choice of programming framework also plays a crucial role. The traditional approach uses C/C++ with CUDA, but modern-day high-level languages like Python, paired with Just-In-Time (JIT) compilers like Numba, offer a trade-off between the difficulty of development and raw performance. The performance characteristics and potential bottlenecks of the Python-Numba toolchain are themselves an important area of study [4].

Building upon this context, this paper presents a comprehensive performance and scalability analysis of the Gaussian blur algorithm implemented using Python with the Numba JIT compiler. Our study provides a direct comparison of three distinct computational architectures: a sequential single-core Central Processing Unit (CPU), a parallel multi-core CPU utilizing all available threads, and a massively parallel Graphics Processing Unit (GPU). We systematically measure the execution time for applying a 5x5 convolution kernel to images of four different resolutions to analyze how each architecture's performance scales with increasing computational load. The remainder of this paper is organized as follows: Section 2 details our experimental setup, Section 3 presents the quantitative results of our benchmarks, and Section 4 discusses the implications of our findings and concludes the paper.

2 Methodology

Our performance analysis was conducted on a custom-built desktop computer. This section details the hardware and software environment, the implementation of the Gaussian blur algorithm across the three architectures, and the performance evaluation procedure. This experimental workflow is visually summarized in Figure 1.

2.1 Experimental Setup

The hardware testbed consists of a central processing unit and a graphics processing unit with the following specifications:

- **CPU:** AMD Ryzen 7 3700X, featuring 8 cores and 16 threads.
- **GPU:** NVIDIA GeForce RTX 3070, featuring 5,888 CUDA cores.

The software environment was built on a Windows operating system. All code was written in Python (version 3.11). The core libraries used for the implementation and analysis include: **Numba** for Just-In-Time (JIT) compilation of CUDA kernels, **NumPy** for efficient numerical array manipulation, **Pillow** for image loading and saving, and **Matplotlib** for data visualization.

2.2 Algorithm Implementation

To evaluate the performance across different architectures, the 5x5 Gaussian blur convolution was implemented using three distinct methods, each tailored to a specific execution model.

2.2.1 Single-Core CPU Implementation

The baseline for our comparison is a sequential implementation designed for a single CPU core. This approach is straightforward but computationally naive. It consists of a series of nested `for` loops that iterate through every pixel's y-coordinate, x-coordinate, and each color channel (R, G, B) of the input image. Inside the innermost loop, another pair of nested loops iterates over the 5x5 kernel, multiplying the kernel weights with the corresponding neighboring pixel values and summing the results to produce the final value for the output pixel. This method processes the entire image pixel by pixel in a strictly sequential order.

2.2.2 Multi-Core CPU Implementation

To leverage the power of modern multi-core processors, a task-parallel version was implemented using Python's `multiprocessing` library. The core strategy involved dividing the input image into a number of horizontal chunks equal to the number of available CPU threads (16 in this case). A pool of worker processes was then created. Each worker process was assigned one chunk of the image and executed a function identical to the single-core blur algorithm on its smaller portion of the data. The processes ran in parallel, one on each available core. After all chunks were processed, their results were collected and reassembled in the correct order to form the final, complete blurred image.

2.2.3 GPU Implementation

For the massively parallel architecture, a data-parallel approach was implemented using the Numba library to compile a Python function into an NVIDIA CUDA kernel. In this model, a GPU grid is launched with thousands of threads, where each thread is uniquely mapped to a single (x, y) coordinate of the output image. Each thread is

responsible for calculating only its assigned pixel's final value. It does this by executing the 5x5 convolution logic independently, looping through the kernel weights and reading the required 25 neighboring pixel values directly from the input image stored in the GPU's global memory. This allows all pixels to be processed conceptually at the same time, taking full advantage of the GPU's thousands of cores.

2.3 Performance Evaluation

The primary metric for our analysis was execution time. To conduct a thorough scalability analysis, the performance of all three implementations was benchmarked across four distinct image resolutions: Standard Definition (640x480), High Definition (1280x720), Full High Definition (1920x1080), and Quad High Definition (2560x1440).

Execution times were measured in seconds using Python's native `time` library. For each test, timing was initiated immediately before the core computational function was called and concluded immediately after it returned. This approach ensures that the measurements reflect only the processing time and exclude the overhead of file I/O operations such as loading the image from disk and saving the final result. To ensure a stable execution environment and prevent driver-level conflicts between Python's `multiprocessing` library and the CUDA context, the CPU-based tests and GPU-based tests were run in separate, isolated script executions.

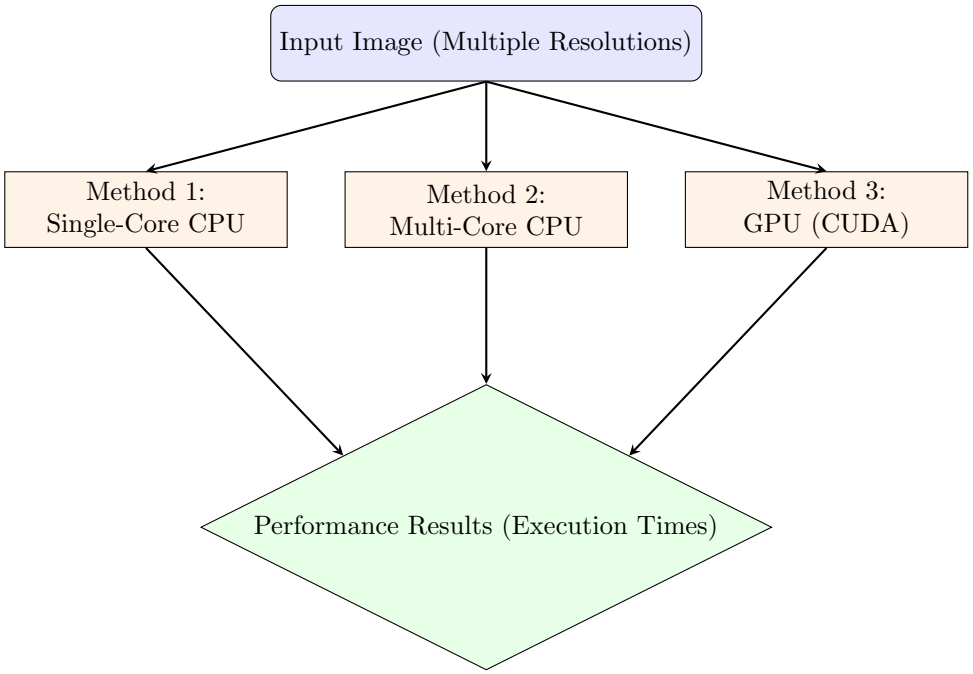

Fig. 1 The experimental workflow. An input image is processed at various resolutions by three distinct implementations, and the execution time for each is recorded for a comparative performance analysis.

3 Results

The performance of the three implemented methods, that is, the single-core CPU, multi-core CPU, and GPU, was benchmarked across the four specified image resolutions. The raw execution times, measured in seconds, were recorded for the core convolution task and are summarized in Table 1.

Table 1 Execution times (in seconds) for 5x5 Gaussian Blur across different architectures and image resolutions.

Resolution	Dimensions	Total Pixels	Single-Core CPU (s)	Multi-Core CPU (s)	GPU (s)
SD (480p)	640x480	307,200	5.3052	1.5501	0.2811
HD (720p)	1280x720	921,600	15.7494	2.8214	0.2834
FHD (1080p)	1920x1080	2,073,600	36.5290	5.7986	0.2772
QHD (1440p)	2560x1440	3,686,400	68.9172	9.6404	0.2832

To better visualize the performance trends and scalability of each architecture, the execution times were plotted against the total number of pixels on a graph with a logarithmic y-axis, as shown in Figure 2.

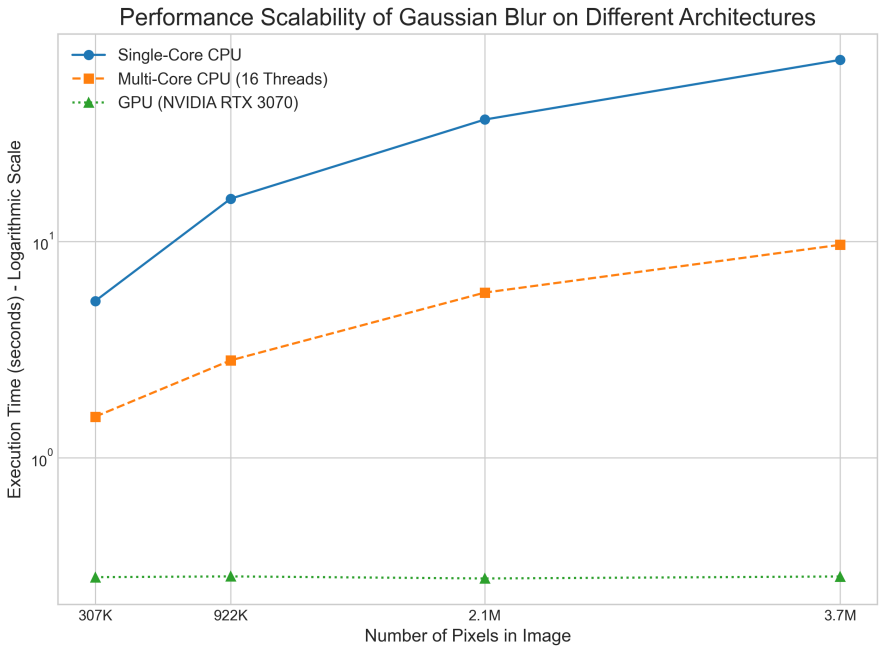

Fig. 2 Performance scalability of the three implementations. Note that the y-axis (Execution Time) is on a logarithmic scale to clearly display the vast differences in performance.

The results demonstrate a consistent and clear evaluation of performance. As illustrated in the figure, the execution times for both the single-core and multi-core CPU implementations increase substantially as the number of pixels grows. The single-core CPU shows the steepest increase, indicating poor scalability. The multi-core CPU implementation shows a more moderate, near-linear increase in execution time. In stark contrast, the GPU execution time remains relatively flat and consistently low across all tested resolutions, showcasing its superior scalability for this data-parallel task.

4 Discussion

The results of our experiment present a clear and compelling narrative about the suitability of different parallel architectures for computationally intensive image processing tasks. The most significant finding is the profound performance and scalability advantage of the GPU-based implementation over both the single-core and multi-core CPU methods. The root cause of this disparity lies in the fundamental architectural differences between CPUs and GPUs [1]. A CPU is designed with a few powerful cores optimized for low-latency access to large caches, making it ideal for sequential or complex task-based workloads. In contrast, a GPU is a throughput-oriented device composed of thousands of simpler arithmetic logic units (ALUs) designed to execute the same instruction on a massive number of data elements concurrently [1]. Our Gaussian blur task, being a classic data-parallel problem, maps perfectly to this architecture, allowing the GPU to mitigate the performance bottleneck seen on the CPU.

A deeper analysis of the speedup factors reveals further insights. The transition from a single-core to a multi-core CPU implementation yielded a consistent speedup of approximately 5.4x. While significant, this is substantially less than the theoretical maximum of 16x for the 16-thread processor. This sub-linear scaling is expected and can be attributed to various overheads, including the cost of creating and managing parallel processes, splitting the data into chunks, and potential memory bandwidth limitations. In contrast, the GPU's speedup of nearly 130x over the single-core baseline demonstrates the transformative efficiency of its data-parallel model for this workload. Perhaps the most compelling result is the GPU's approximately 24x speedup over the fully-utilized, 16-thread multi-core CPU. This highlights that for a task as parallelizable as image convolution, the architectural specialization of the GPU provides an order-of-magnitude performance improvement that cannot be matched by simply adding more general-purpose CPU cores.

A notable observation from the results is the relatively flat performance curve of the GPU, as seen in Figure 2. While the computational load increased by over 12 times from the smallest to the largest image size, the GPU's execution time remained nearly constant. This suggests that for this specific algorithm on the test hardware, the runtime is dominated by fixed overheads—such as kernel launch latency and data transfer between the host and device—rather than the core computation. The convolution itself is executed so efficiently that its duration is a negligible fraction of the total measured time. Furthermore, it is important to consider these results within the context of our

chosen toolchain. As noted by Oden (2020), while Python with Numba offers significant development convenience, its performance does not always match that of native C-CUDA due to factors like Just-In-Time (JIT) compilation overhead and the performance of the surrounding Python code that orchestrates the GPU calls [4]. Therefore, the substantial speedups observed in this study are representative of the gains within a high-level framework, and a lower-level implementation could potentially yield even greater acceleration.

5 Conclusion

Our paper presented an extensive performance and scalability analysis of the 5x5 Gaussian blur algorithm, a computationally intensive image processing task. We implemented the algorithm in Python using the Numba library and benchmarked its performance across three distinct architectures: a sequential single-core CPU, a parallel 16-thread multi-core CPU, and a highly parallel NVIDIA RTX 3070 GPU. Our findings from the scalability analysis were quite definitive. The GPU implementation demonstrated a performance advantage, achieving a speedup of approximately 130x over the single-core CPU and 24x over the fully utilized multi-core CPU for a high-resolution image. Furthermore, the GPU's execution time remained nearly constant regardless of image size, highlighting its superior scalability. This study confirms that for data-parallel tasks like image convolution, the specialized architecture of the GPU is the most crucial factor for achieving high performance. It significantly outpaces even parallel general-purpose CPUs.

6 Future Work

While this study provides a performance comparison, several avenues for future research exist.

- **Kernel Optimization:** The GPU kernel used in this study was a direct implementation of the algorithm. Future work could explore advanced optimization techniques, such as the use of GPU shared memory, to reduce global memory latency and potentially achieve even greater speeds and efficiency.
- **Data Transfer Overhead Analysis:** Our analysis intentionally excluded the time required to transfer data between the host (CPU) and the device (GPU). A future study could explicitly measure this overhead, as it is a critical performance factor in many real-world applications [3].
- **Broader Algorithmic Scope:** This research could be expanded to include other complex, convolution-based filters (e.g., Sobel, bilateral filters) or different types of image processing algorithms to determine if the observed performance trends hold true across a wider range of tasks.
- **Comparative Framework Analysis:** A direct performance comparison between the Python/Numba implementation and a traditional C/C++ with CUDA implementation on the same hardware could provide quantitative data on the performance trade-offs associated with this high-level programming framework, building on the findings of Oden (2020) [4].

References

[1] Afif, M., Said, Y., Atri, M.: Computer vision algorithms acceleration using graphic processors nvidia cuda. Cluster Computing (2020)

[2] Hangün, B., Eyecioğlu, : Performance comparison between opencv built in cpu and gpu functions on image processing operations. INTERNATIONAL JOURNAL of ENGINEERING SCIENCE AND APPLICATION **1**(2), 34–41 (2017)

[3] Ibrahim, N.M., ElFarag, A.A., Kadry, R.: Gaussian blur through parallel computing. In: Proceedings of the International Conference on Image Processing and Vision Engineering (IMPROVE), pp. 175–179. SCITEPRESS - Science and Technology Publications, ??? (2021). https://doi.org/10.5220/0010513301750179

[4] Oden, L.: Lessons learned from comparing c-cuda and python-numba for gpu-computing. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 216–223. IEEE, ??? (2020). https://doi.org/10.1109/PDP50117.2020.00041

Acknowledgements. I would like to express my sincere gratitude to my course instructor, Annajit Alim Rasel sir, for his invaluable guidance and support throughout this research project.