© Rzeszów University Press
ISSN 2719-6550
e-ISSN 2719-7417

"Journal of Education, Technology and Computer Science"
No. 6(36)/2025
https://journals.ur.edu.pl/jetacomps

Received: 3.08.2025
Accepted for printing: 24.11.2025
Published: 31.12.2025
License: CC BY-NC-ND 4.0

DOI: 10.15584/jetacomps.2025.6.17
Scientific

PAWEŁ DYMORA[1], **MIROSŁAW MAZUREK**[2]

General Computing Using CUDA Technology on NVIDIA GPU

[1] ORCID: 0000-0002-4473-823X, PhD Eng., University of Technology, Faculty of Electrical and Computer Engineering, Poland; email: Pawel.Dymora@prz.edu.pl
[2] ORCID: 0000-0002-4366-1701, PhD Eng., University of Technology, Faculty of Electrical and Computer Engineering, Poland; email: mirekmaz@prz.edu.pl

Abstract

The article presents a detailed analysis of the computing capabilities of the GPU (Graphics Processing Unit) using NVIDIA Compute Unified Device Architecture (NVIDIA CUDA) compared to traditional sequential computing methods. For this purpose, an application implementing the Gaussian blur algorithm was developed. Then, an implementation of the problem was created in the form of a program. The next step presented the methodology of conducting a study comparing the efficiency of solving the problem with several test configurations. Then, research was carried out during which the data obtained in the form of program implementation times were collected. This paper aims to evaluate the computational capabilities of the GPU using NVIDIA CUDA compared to traditional sequential computing methods. The comparison was made through a developed application that implements the Gaussian fuzzy algorithm. The article can serve as a valuable educational resource for teaching parallel programming and algorithm optimization using GPU and CUDA technologies. The conducted analysis also provides a strong example of an educational project that combines algorithm theory with practical application in the context of improving computational performance.

Keywords: CUDA, NVIDIA, GPU, Technology, Gaussian Blur, Parallel Compute

Introduction

Central Processing Units (CPUs) are the foundation of computers designed for general computing tasks, having a broad instruction set. This allows CPUs to handle a wide variety of computing tasks. Early CPUs had only one core responsible for executing arithmetic instructions (Kirk, Hwu 2009). A single-core processor can perform calculations sequentially, meaning that each instruction must be executed in turn. The core cannot move on to execute the next instruction before the current instruction has finished. Despite the fact that they are less advanced than modern multi-core counterparts, single-core processors are still effective at efficiently managing lightweight, unparalleled tasks (Andersch et al, 2002; Bakyo, 2003). Among other things, they offer a simpler memory hierarchy, which makes their design cost lower compared to multicore processors. Furthermore, software designed specifically for single-threaded environments typically shows higher performance on single-threaded applications due to limited context switching and minimal interference from other competing processes.

As technology has advanced, processor manufacturers have begun to add multiple cores to a single processor, giving rise to multi-core processors. Modern consumer desktops typically feature quad-core or six-core configurations, while high-end servers and HPC platforms have dozens of cores per socket. Multi-core processors allow multiple threads to run simultaneously, leading to increased performance for parallel workloads (Polsson, 2012).

The advent of multi-core processors has brought more opportunities to improve system-level performance with parallel processing techniques such as symmetric multi-core processing (SMP), asymmetric multi-core processing (AMP), and NUMA architectures. SMP involves evenly distributing computational tasks between identical cores that have equal access rights to memory and I/O resources. AMP, on the other hand, assigns unique functions to individual cores, creating dedicated channels for specific activities (e.g., video encoding and decoding, network traffic management). NUMA architectures involve grouping cores around localised memory banks, minimising latency associated with memory requests (Gwizdała, 2016; Intel's First Microprocessor).

Optimising multi-core processor computing requires consideration of key elements such as cache hierarchy and shared resource allocation strategy. Ensuring that tasks are appropriately allocated between available cores ensures optimal resource utilisation and alleviates potential bottlenecks resulting from insufficient memory or I/O device bandwidth. GPUs were originally designed to process images and video on screens, but were not as efficient as CPUs in terms of processing power. Nevertheless, they were more efficient at certain tasks due to their parallel processing architecture, which allowed multiple allocated tasks to be processed simultaneously. This parallel processing capability of GPUs was used by developers to increase the performance of an entire computer or server. GPUs began to be used for more general computing tasks, which is now commonly referred to as GPU computing (Choquette, Lee, Krashinsky, Balan, Khailany, 2021).

The article can serve as a valuable educational resource for teaching key concepts in computer science, particularly within courses focused on computer architecture, operating systems, and parallel programming. It offers a clear introduction to the evolution of CPUs from single-core to multi-core processors, helping students understand the motivations and benefits of parallel processing. Concepts such as SMP, AMP, and NUMA architectures can enrich discussions on system--level optimization and task scheduling. The comparison between CPU and GPU architectures provides a foundation for exploring the differences between sequential and parallel computing, while the transition of GPUs into general-purpose computing devices introduces students to modern hardware acceleration. Especially noteworthy is the innovative perspective on GPU computing, with a focus on how NVIDIA's parallel architecture has revolutionized data processing beyond traditional graphics tasks. By highlighting the repurposing of GPUs for general-purpose computation (GPGPU), the text demonstrates the originality of leveraging massively parallel architectures to achieve performance gains across various computing domains. The discussion on memory hierarchy and resource allocation strategies supports practical lessons in software engineering and system design. This material can be effectively used for both theoretical understanding and practical lab exercises, such as benchmarking different hardware configurations. It encourages critical thinking about how technological advances influence software development and system performance. By presenting these topics coherently and innovatively, the text supports the development of a well-rounded understanding of modern computing systems, which is essential for future IT professionals.

Building a sample CUDA program

The CUDA example application presented shows an implementation of a simple program whose task is to perform the sum of two input matrices, A and B of size N x N, and write the result to matrix C. The primary function responsible for performing this operation is VecAdd, which is executed on the GPU as a kernel using CUDA (Ghorpade, Parande, Kulkarni, Bawaskar; Dehal, Munjal, Ansari, Kushwaha, 2018).

Initially, memory allocation is done on both the host and the CUDA device to store the input matrices A and B, along with the resultant matrix C. Dynamic allocation was invoked via malloc on the host side to reserve space for each matrix. Once memory is allocated, the input arrays are initialised on the host before being copied to device memory using cudaMalloc to allocate memory on the CUDA device and cudaMemcpy to transfer data between host and device memory.

The main piece of code resides within the VecAdd kernel, prefixed with _global_, where each thread computes one element from the final matrix C by adding the corresponding elements of matrices A and B. Each thread is identified to compute a specific subset of indexes through the following expression, which defines the variable i:

```
i = blockDim.x * blockIdx.x + threadIdx.x
```

Where blockDim.x indicates the number of threads per block, while blockIdx.x represents the index of the current block in execution relative to all running blocks. Finally, ThreadIdx.x indicates the position of a thread inside its corresponding block. The mapping presented here helps to distribute work evenly across multiple threads, ensuring efficient use of the computational resources available on the GPU. Before performing the operation, we check that the index i does not exceed the size of the matrices being computed. If everything is correct, the kernel continues. At the very end, after executing the GPU kernel with VecAdd<<blocksPerGrid, threadsPerBlock>>, the results stored in the C array must be moved back to the system RAM so that they become available to the CPU again. The cuda-Memcpy command is executed, where the data is now moved from the CUDA device to the host. Once completed, the allocated device memory is freed by calling cudaFree for all GPU-side variables. We also free the memory on the host using the built-in free() function, after which the application is terminated (Fatica, 2008; Tullsen, Eggers, Levy, 1995).

Gaussian blur

The Gaussian function, also known as the Gaussian curve or bell curve, was developed by German mathematician Carl Friedrich Gauss in the early 19th century. Gauss introduced the concept of a normal distribution, which is a continuous probability distribution characterised by a symmetrical bell-shaped graph. This distribution is often used to model real-world phenomena in which there is a tendency towards a central value, with decreasing probability of extreme deviations from this value (Ibrahim, ElFarag,Kadry, 2021).

The Gaussian blurring technique involves calculating a weighted average of the pixel intensities around each target pixel in the input image. These weights are determined according to their position along the Gaussian curve, meaning that more weight is given to pixels closer to the centre than those at the edges. As a result, the output image appears softer and less noisy compared to the input, making Gaussian blur a popular choice for postprocessing tasks such as noise reduction, edge smoothing, and antialiasing. The formula for the two-dimensional Gaussian blur function has the following form:

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

Gaussian blurring is a widely used technique in various fields due to its effectiveness in reducing high-frequency noise. Thus, the algorithm reduces the noise output, which is the reason for its variety of applications. One of these is data animation, where Gaussian blurring gets rid of elements that can be subjected to an identification process. For example, a blurred face, together with other covered data on an ID card, will make it significantly more difficult to trace a person.

Another application is the simulation of motion blur. Animations created with computer programmes require the closest possible reproduction of reality in order to accurately reproduce the situations occurring in it. Gaussian blur allows the simulation of motion blur, so that animators are able to convincingly reflect the movement of an object over time.

Performance testing of sequential and parallel processing

The performance test to be carried out was the application of a Gaussian blur to images. The Gaussian blur will be applied through an application that can run in CPU calculation mode and using CUDA technology. The application also measures the time during which the calculations will be performed. It allows a time comparison to be made between the two calculation methods.

Measurements were made on a dataset of selected images with different resolutions. These files have the following resolutions: 512 x 512, 1280 x 720, 1920 x 1080, 2560 x 1440, 3840 x 2160, and 7680 x 4320, respectively (Figure 1). Several images with different dimensions were selected to see if there was an effect on the execution time of the programme from the number of pixels processed. For each case, the test was performed 10 times, with their output value being their average. The data was automatically collected by an automation script, which at the end of the run saves the results in CSV format for analysis.

The Gaussian blur overlay programme was run in a minimum system load situation to allow it to use as many system resources as possible to ensure consistent and maximum performance.

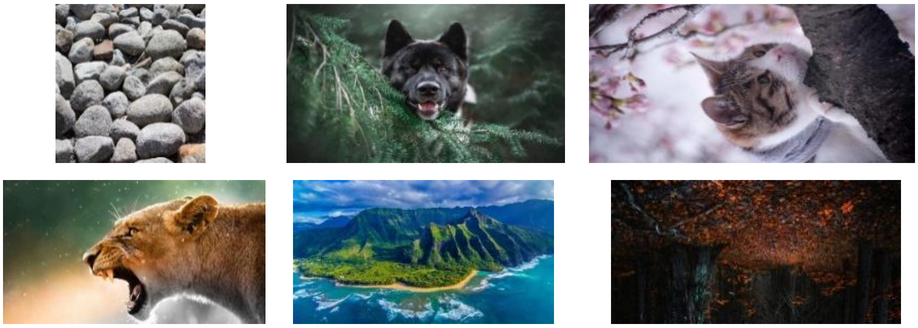
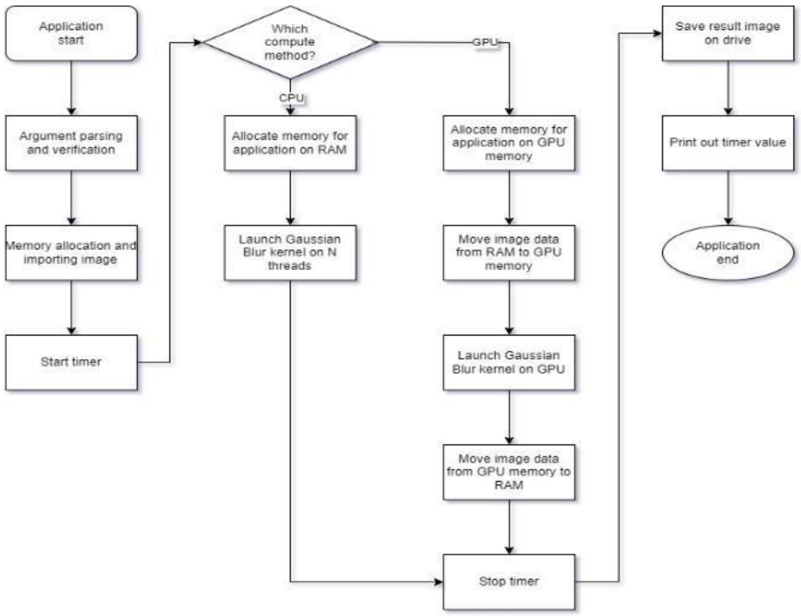


Figure 1. Images on which tests were conducted

Implementation of the Gaussian blur algorithm

The entire programme fits into approximately 420 lines of code, written in C++, specifically in the C++17 standard (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html; https://en.cppreference.com/w/cpp/thread/thread; https://en.cppreference.com/w/cpp/chrono). The programme is divided into an initialisation part, a computation part, and a finalisation part (Figure 2). It implements the Gaussian fuzzy algorithm in two ways: sequential computation and parallel computation. The supported image format is PNG only.



Application start
Argument parsing and verification
Memory allocation and importing image
Start timer
Which compute method?
CPU
GPU
Allocate memory for application on RAM
Launch Gaussian Blur kernel on N threads
Allocate memory for application on GPU memory
Move image data from RAM to GPU memory
Launch Gaussian Blur kernel on GPU
Move image data from GPU memory to RAM
Stop timer
Save result image on drive
Print out timer value
Application end

Figure 2. Block diagram of the designed Gaussian blur application

The first task of the program after it has started (the initialisation part) is to parse the arguments that determine its operation. The path to the input image is then checked for correctness. Once the programme has started, a timer is started to measure the length of the segment. The application allocates the appropriate amount of memory for storing the image data to RAM or in the memory of the graphics card, depending on the device performing the calculation. The next step is to allocate sub-tasks to the number of threads specified in the arguments to the program, and to call the function that will start the calculation. A subtask is a fragment of the whole task – applying a Gaussian blur. When the calculation is complete, we copy the results to the output variable and stop the timer. We then save the processed image with the superimposed Gaussian blur in the specified path, after which we write out the status of the timer, which will show us how long it took to complete the task. Finally, we release the reserved memory to avoid situations where the memory would not release automatically. At this point, the programme is terminated (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html; https://docs.nvidia.com/cuda/cuda-runtime-api/index.html).

Analysis of measurement results

The first test carried out was to check the effect of the Gaussian blur radius on execution time. The comparison was done on a "Leaves" image with a resolution of 7680 x 4320 with different numbers of allocated CPU threads, and blur radii with the following values: r = 2, r = 4, r = 6, r = 8, and r = 10. The figure shows six distinctive trends, each allocated to one of the test cases. The X-axis shows the number of threads allocated to the task, from 1 thread to the maximum number of threads available on the processor – 12, while the Y-axis shows the Gaussian blur processing time in milliseconds.

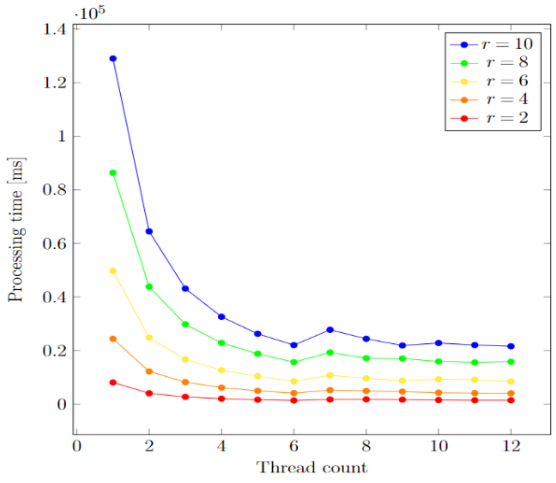


Figure 3. Gaussian blur processing time based on tested blur radius for a 7680 x 4320 image using 12 CPU threads

With a larger value of the Gaussian blur radius, the processing time of the programme increases. The reason for this phenomenon is the increase in the number of pixels required to determine the blur factor for a single point in a given region, which translates into a greater number of operations needed to be performed. The second highlighted element is the correlation of processing time to the number of threads dedicated to the task. As the number of threads increases, the computation time decreases. The rationale for this relationship is the process of allocating subtasks to the processor. The programme allocates an equal number of pixels to be processed for each thread, so that the computational performance will increase with the number of allocated threads. Upon closer observation, it can be seen that there is an anomaly. When a computation is allocated to several threads greater than the number of physical CPU cores, the task execution time does not decrease. When a task is allocated to 7 threads, the process takes noticeably longer to execute than with 6 threads (Figure 3).

Another element investigated was the effect of the number of threads allocated to apply Gaussian blur and image resolution, i.e., the number of pixels computed, on processing time.

Figure 4 shows six distinctive trends, analogous to the previous graph, each assigned to one of the test images. The X-axis shows the number of threads allocated to the task, from 1 thread to 12, while the Y-axis represents the Gaussian blur processing time in milliseconds. Analysing the graph, we can see a relationship. The resolution of the image has a proportional effect on the processing time of the Gaussian blur algorithm. The trend for the 'Leaves' image significantly diverges from the rest, where, for it, the time spent on calculation was 15.8 seconds at best. This is due to the fact that as the pixels required for processing increase, we will see the running time of the algorithm increase.

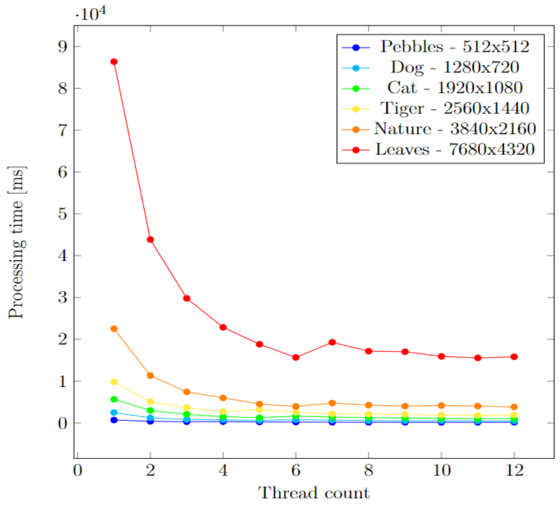


Figure 4. Gaussian blur processing time based on image resolution and allocated thread count

Figure 5 shows six different cases of CPU thread configuration and one GPU (CUDA) configuration, where each case has a trend. The X-axis represents the number of pixels of the processed image on a logarithmic scale, and the Y-axis represents the time taken by the programme to complete the task, also placed on a logarithmic scale. The logarithmic scale has been used for data clarity. CPU trends are tested in 1, 2, 4, 6, 8, and 12 thread situations, where CUDA was allocated the maximum number of cores available – 4864. After examining the graph shown, it becomes clear that there is a direct proportional relationship between the number of pixels computed and processing time in each configuration tested. This relationship is somewhat less pronounced in the case of CUDA.

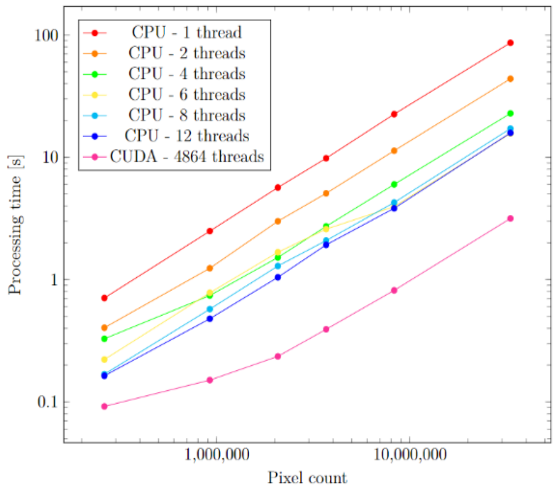


Figure 5. Gaussian blur processing time based on image resolution and allocated thread count on different devices

Table 1. Collection of results from Gaussian blur application on CPU and CUDA device

Pixel count	Processing time for 12 CPU threads [ms]	Processing time for 4864 CUDA threads [ms]	% Difference
262 144	163.46	91.88	77.91
921 600	478.80	150.71	217.70
2 073 600	1044.23	235.68	343.07
3 686 400	1923.91	392.26	390.47
8 294 400	3827.68	816.49	368.80
33 177 600	15830.40	3161.32	400.75

Optimization

Optimisation methods were used in the development and compilation of the application. As a result, the programme is able to perform the task faster while maintaining the structure of the algorithm. Two methods were used that allowed a significant reduction in the execution time of the calculations. The first optimization method used is the so-called loop boring. This involves replicating the contents of the for loop to create N copies together with the original, thus reducing the number of iterations needed. The second optimization method used lies in the NVCC compiler (NVIDIA CUDA Compiler, NVCC). The compiler has a number of flags that optimise the programme at the compilation stage. One of these is the -O3 flag, which imposes more than 80 different optimisation methods. With this flag, the compiler improves the performance of the application at the expense of compile time and debuggability. However, this method has one disadvantage: The functioning of the application may be subtly altered. Taking this into account, the hashes of the resulting images were checked in both situations. It turned out that the hashes are identical, which means that the resulting images are identical.

The consistency of the programme's performance is maintained, which means that this optimisation method can be applied to the rest of the tests performed.

The test was performed with different numbers of allocated CPU threads on the 'Leaves' image, assuming a Gaussian blur radius of r = 8. The highest resolution image was chosen to make the possible discrepancy as large as possible.

Table 2. Collection of results from Gaussian blur application on CPU with and without applying optimizations during compilation

Pixel count	Processing time without optimizations [ms]	Processing time with optimizations [ms]	% Difference
262 144	917.75	163.46	461.45
921 600	2897.84	478.80	505.22
2 073 600	6663.05	1044.23	538.08
3 686 400	10368.00	1923.91	438.90
8 294 400	23891.70	3827.68	524.18
33 177 600	97657.00	15830.40	516.89

Figure 6 shows two situations: a program compiled with optimisations and without optimisations. By analysing the graph, we are able to conclude that the execution time of the program compiled with optimisations is significantly less than without optimisations, regardless of the number of pixels. The difference in performance is clear and significant. The processing time of the programme compiled with the aforementioned optimisations is faster by an average of 497%, and by 538% in the best case. The optimisations applied had a very positive effect on the execution speed.

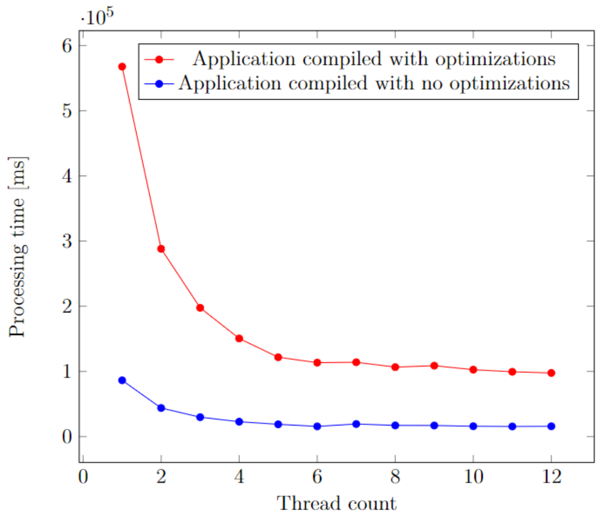


Figure 6. Performance comparison on the CPU without optimizations and with optimizations on an image with a resolution of 7680 x 4320 and a Gaussian blur radius of r = 8

Conclusion

This paper analyses and compares the computational performance between central processing units (CPUs) and graphics processing units (GPUs) using NVIDIA Compute Unified Device Architecture (CUDA) technology. In this case, a Gaussian blur overlay operation was applied to the images. A wide range of resolutions, up to 8K (7680x4320), were used for the test to highlight the impact of task size on execution time. Optimisation strategies such as loop boring and the use of built-in optimisation methods in the NVCC compiler were applied.

Analysis of the resulting data shows a significant speed-up in the execution of Gaussian fuzzing operations on the GPU compared to the CPU. Specifically, the performance difference ranged from 77% up to 400% advantage for the GPU, depending on the resolution of the image under test. The conclusions obtained from the practical part of the work indicate the performance potential of the GPU to accelerate complex tasks typically performed on traditional CPUs.

Studies have confirmed the performance advantage of GPUs using CUDA technology over conventional CPU-based computing. More precisely, this advantage exists for several problems, such as the manipulation of large arrays and intensive numerical calculations. Further developments in the field of algorithms will make it possible to use them on the GPU, allowing an increase in computing power for more problems.

The article serves as a highly effective educational resource for teaching modern computing concepts, particularly in the fields of parallel programming, high-performance computing, and system optimization. By comparing CPU and GPU performance through a practical implementation of the Gaussian blur algorithm using NVIDIA CUDA technology, students gain hands-on experience in understanding the impact of data size, task distribution, and hardware architecture on computational efficiency. The use of high-resolution image processing, up to 8K, allows learners to grasp scalability challenges and appreciate the performance potential of GPUs. Innovative techniques such as loop unrolling and NVCC compiler optimizations demonstrate real-world strategies for improving code execution, fostering critical skills in performance tuning. The originality of applying traditional algorithms to cutting-edge GPU architectures highlights how established methods can evolve through hardware advancements. This educational approach not only deepens students' understanding of architecture-level differences but also equips them with key competencies in parallel and heterogeneous programming–skills that are increasingly essential in AI, data science, and modern software engineering. Moreover, analyzing real performance data enhances students' analytical thinking and problem-solving abilities, while exposing them to the innovations of GPU computing and the transformative role of NVIDIA technologies in accelerating complex computations.

Acknowledgments

We are grateful to Michał Kluz, a graduate student at Rzeszów University of Technology, for supporting us in collecting useful information.

References

Andersch, M., Palmer, G., Krashinsky, R., Stam, N., Mehta, V., Brito, G., Ramaswamy, S. (2022). *NVIDIA Hopper Architecture In-Depth*. Retrieved from: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth (1.07.2025).

Bakyo, J. (2003). *Great Microprocessors of the Past and Present*. Retrieved from: https://web.archive.org/web/20120415121639/http://jbayko.sasktelwebsite.net/cpu.html (1.07.2025).

Choquette, J., Lee, E., Krashinsky, R., Balan, V., Khailany, B. (2021). 3.2 The A100 Datacenter GPU and Ampere Architecture. *IEEE International Solid-State Circuits Conference (ISSCC)*. 2021. DOI: 10.1109/ISSCC42613.2021.9365803.

Dehal, R.S., Munjal, C., Ansari, A.A., Kushwaha, A.S. (2018). *GPU Computing Revolution: CUDA*. 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN), Greater Noida, India. DOI: 10.1109/ICACCCN.2018. 8748495.

Fatica, M. (2008). *CUDA toolkit and libraries*. 2008 IEEE Hot Chips 20 Symposium (HCS), Stanford, CA, USA. DOI: 1109/HOTCHIPS.2008.7476520.

Ghorpade, J., Parande, J., Kulkarni, M., Bawaskar, A. *GPGPU Processing in CUDA Architecture*. arXiv:1202.434.

Gwizdała, P. (2016). *Generations of the computer processors*. Retrieved from: https://www.slideshare.net/ArshadQureshi5/generation-of-computer-processors-52195241 (8.07.2025).

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (11.07.2025).

https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (11.07.2025).

https://en.cppreference.com/w/cpp/chrono (11.07.2025).

https://en.cppreference.com/w/cpp/thread/thread (11.07.2025).

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (11.07.2025).

Ibrahim, N.M., ElFarag, A.A., Kadry, R. (2021). Gaussian Blur through Parallel Computing. *Proceedings of the International Conference on Image Processing and Vision Engineering IMPROVE*, *1*, 175–179. DOI: 10.5220/0010513301750179.

Intel's First Microprocessor. Retrieved from: https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html (11.07.2025).

Kirk, D., Hwu, W. (2009). *A Simple Example, Tools, and CUDA Threads*. Retrieved from: online: https://nanohub.org/resources/7234/download/lecture3_cuda_threads_tools_examples.pdf (8.07.2025).

Polsson, K. (2012). *Chronology of Personal Computers*. Retrieved from: https://web.archive.org/web/20120415044730/http://www.islandnet.com/%7Ekpolsson/comphist/ (8.07.2025).

Tullsen, D.M., Eggers, S.J., Levy, H.M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings 22nd Annual International Symposium on Computer Architecture* (pp. 392–403). Santa Margherita Ligure, Italy.