# Investigation of CUDA Graphs performance for selected parallel applications

Oksana Diakun[0009−0005−5055−9245] and Paweł Czarnul[0000−0002−4918−9196]

Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology, Narutowicza 11/12, 80-233 Poland
oksdiaku@pg.edu.pl, pczarnul@eti.pg.edu.pl

**Abstract.** In this paper, we contribute by providing a direct comparison of performance of NAS Parallel Benchmarks implemented with standard CUDA against our extended implementation with CUDA Graphs. The evaluation was conducted for four hardware platforms, using two desktop GPUs—NVIDIA GeForce RTX 2080 and NVIDIA GeForce RTX 4070 Ti—and two server-class GPUs—NVIDIA Quadro 8000 and NVIDIA A100 80GB. The primary focus of the comparison was the execution time of the benchmarks, analyzed for various problem sizes (classes S, A, B, C and D). Two applications exhibited noticeable performance gains from the implementation of CUDA Graphs. The CG code demonstrated the most consistent improvements across all cases, achieving an average relative speedup of 3.3%. The highest result of 4.13% (Class C) for this algorithm was achieved for the NVIDIA GeForce RTX 4070 Ti card. The LU code showed gains primarily on newer generation GPUs, with an average speedup of 2% on the RTX 4070 Ti and 7%, on the A100, with a maximum gain of 11.87% (Class B). In contrast, visible negative performance was observed for MG and some instances of IS, but we attribute that to the relatively small absolute running time in which case additional overheads cannot be mitigated by the new mechanism.

**Keywords:** GPGPU · CUDA Graphs · Parallel Computing · High-performance computing · GPU Acceleration · Performance evaluation

## 1 Introduction

NVIDIA CUDA (Compute Unified Device Architecture) technology, for many years already, has allowed programmers to parallelize code on GPU devices [5]. CUDA versions have progressively introduced new features and optimizations. CUDA Graphs [7], investigated in this paper, were designed to tackle the overhead associated with CPU-GPU interactions, especially when handling a large number of small and/or interconnected tasks. CUDA Graphs allows to bundle a series of CUDA operations into a single graph entity that can be executed all at once. This reduces the frequency and cost of CPU-GPU communication, including kernel launch overhead, as visualized in [4].

This paper aims to assess the performance of CUDA Graphs across a diverse set of parallel applications. The main goal of this research is to compare the

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

performance of applications that use CUDA Graphs with traditional CUDA versions without CUDA Graphs. This comparison will help identify situations where CUDA Graphs offer performance improvements and where their use may be less beneficial. Section 2 presents existing examples of applying CUDA Graphs to various applications. Section 2 describes motivations and contribution of this work. Section 3 specifies the set of selected benchmarks and its updating with CUDA Graphs. Section 4 describes the conducted experiments, presents and discusses the results. Section 5 presents a summary and potential future work.

## 2 Related work and our contribution

CUDA Graphs, introduced in CUDA 10, allow to express more complex execution scenarios in a form of a graph. Nodes of the latter can refer to kernel launches, memory copies, and synchronization events. Edges between nodes define the dependencies. A particular node begins execution only after all its predecessor nodes have completed. This effectively removes the overhead of CPU-side stream synchronization. CUDA Graphs can effectively reduce the overhead associated with launching individual kernels and improve overall application performance [4]. One way of adopting the code to use CUDA Graphs is to encapsulate and capture existing operations put into a CUDA stream, into a graph. This can be done by calling `cudaStreamBeginCapture()` and `cudaStreamEndCapture()` API calls. Alternatively, a graph can be created explicitly using `cudaGraphCreate()` and appropriate functions for definition of nodes and dependencies.

The two CUDA Graphs creation modes can be combined for effective code in a similar scenario. In article [11], the author proposes code with 3 kernel invocations where the first two are captured in a stream with static parameters while a third kernel is being added manually with dynamic parameters to the graph extracted with a call to `cudaStreamGetCaptureInfo_v2()`. In terms of performance, three versions were compared: without CUDA Graphs, with CUDA Graphs with the recapture-then-update approach, using CUDA Graphs with the aforementioned combined approach, the latter two giving speed-ups over the first one: 1.22 and 1.63 respectively. Obviously, potential gains depend on the ratio of the time spent on computations to communication as well.

Article [9] discusses combining CUDA Graphs with an Image Processing Domain-Specific Language (DSL) and Hipacc, a source-to-source compiler. Benchmarks were conducted across ten different image processing applications on two different NVIDIA GPUs: the RTX2080 and the GTX680. The proposed approach allowed to achieve a geometric mean speedup of 1.30 over Hipacc without CUDA graph, 1.11 over CUDA graph without Hipacc.

Paper [12] discusses a novel compiler transformation technique that converts OpenMP code into CUDA Graphs to enhance NVIDIA device programmability. The approach combines high-level OpenMP's high-level programmability with CUDA's performance benefits. The performance evaluation involved two benchmarks: Saxpy, a structured benchmark with 1024 tasks, and Cholesky decomposition, an unstructured benchmark with 1540 tasks. The evaluation showed that,

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

for the Saxpy benchmark, the CUDA graph implementation reduced execution time by approximately 78%, while the Cholesky decomposition saw a reduction in execution time by approximately 93% when compared to their OpenMP offloading counterparts. It was noted, though, that CUDA Graphs significantly increase the complexity and the number of lines of code needed, especially for Cholesky decomposition. In work [3] authors conducted a study that involved modifying a Breadth-First Search (BFS) application from a benchmark suite to utilize CUDA Graphs, and comparing its performance to the original non-graph version. The testbed system was equipped with an NVIDIA RTX 2060 GPU and a Ryzen 5 3600x CPU. The BFS application from the Rodinia benchmark suite was selected, duplicated, and modified to implement code with CUDA Graphs. The results, for runs repeated 100 times, showed that for input sizes of 524,288 nodes, the CUDA Graph implementation allowed to obtain a 14% speedup compared to the non-graph counterpart, with a running time reduction from 1836$\mu$s to 1610$\mu$s. On the other hand, for smaller input sizes, such as 8,192 nodes, the CUDA Graph version actually exhibited worse performance, with the execution time 306$\mu$s – 7% slower than 285$\mu$s for the standard non-graph version. The study indicates that while CUDA Graphs can offer performance benefits, particularly for medium-sized workloads, the overhead associated with graph instantiation can negate these benefits for smaller workloads.

Article [6] discusses the integration of CUDA Graphs in PyTorch, which is aimed at accelerating PyTorch with advanced CUDA features. Using CUDA Graphs resulted in up to a 50% reduction in CPU usage and an increase in GPU utilization, which led to a significant speedup of up to 30% in the Mask R-CNN model's overall execution time. The article also illustrates how employing CUDA graphs to capture the model leads to the elimination of CPU overhead and synchronization issues resulting in a performance boost of 1.12 times for a large-scale BERT model.

In the context of existing works, we aim at thorough assessment of CUDA Graphs versus the traditional CUDA model. We do that by extending the well established NAS Parallel Benchmarks (NPB) [2] (available at GMAP/NPB-GPU: NAS Parallel Benchmarks for GPU) [1] with CUDA Graphs. We then performed comprehensive comparison on four different GPUs. We discuss relative speed-up, which refers to the difference in execution time between the approach without and with implemented CUDA Graphs in relation to the time obtained for the approach without CUDA Graphs.

## 3 Testbed workloads and our implementations with CUDA Graphs

The collection of evaluated benchmarks consisted of algorithms Conjugate Gradient (CG), Embarassingly Parallel (EP), Integer Sort (IS), Multi-Grid (MG), CFD-related tasks (LU, SP) and one CFS-related task (BT), without Fourier Transform (FT). For the latter, we did encounter unsuccessful verification of the original code on one of the cards.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

For the CUDA Graphs implementations, we decided to use the stream capture mode. Details regarding implementation are available in the github repository[1]. The implementation began with the creation of a stream to which kernel functions would be assigned, so that they could then be captured. After that, variables that will describe our graph were created. A bool variable ensures that the graph will be created only one time, when set to false, and true flag allow executing an already existing graph. To capture a stream it was necessary to create a new, non default one, and assign desirable functions to it. After that, with a call to the `cudaStreamBeginCapture(stream, mode)` function it was possible to start capturing. The capturing of the graph could be easily terminated with a `cudaStreamEndCapture(stream, &graph1)` call. To connect captured graph with its execution name, function `cudaGraphInstantiate(graphExec_1, graph_1, NULL, NULL, 0)` was used. Having that it was possible to finally call an instance of created graph by `cudaGraphLaunch(graphExec_1, stream)` [8].

Unfortunately, the presented procedure would not work in every case. It would not be optimal in situations where kernel functions calls are interspersed with operations performed on the CPU, where, for example, the value of a variable used in the kernel had to be changed. Such situations occurred in several places in our implementation and we decided to use other CUDA functionalities so that the mentioned values could always be available in the graph. One option was to use CUDA Graphs with Dynamic Parameters as outlined in [11]. We eventually chose the task-related variable which is allocated on the device and stores the value calculated on the host, transferred via `cudaMemcpy()`.

Another problem is a situation where the function we want to capture contains a set of various other functions calling kernel functions. This situation is problematic, because: firstly – it is possible to encounter a case similar to the one presented previously, and secondly – it is possible that in certain functions it will be necessary to copy data from device to host, e.g., using function `cudaMemcpy()`. The solution to this is to replace the synchronous operation with an asynchronous one. This also effectively requires to prepare an analogous function and add the stream used for capturing within the function parameters. Finally, we verified correctness of results by checking an appropriate flag already implemented in the NAS NPB software.

## 4 Experiments

The foregoing benchmarks were evaluated on GPUs of various classes/architectures: desktop/Turing NVIDIA GeForce RTX 2080, desktop/Ada Lovelace GeForce RTX 4070 Ti, workstation/Turing Quadro RTX 8000 and server/Ampere A100 80GB PCIe. System configurations for the 4 nodes are as follows: 2 x Intel(R) Xeon(R) Gold 6130, NVIDIA GeForce RTX 2080, 256 GB RAM; 13th Gen Intel(R) Core(TM) i7-13700K, NVIDIA GeForce RTX 4070 Ti, 32 GB RAM; Intel(R) Xeon(R) Gold 6248R CPU, 2 x Quadro RTX 8000 + Quadro RTX 5000,

[1] https://github.com/odiakun/NPB-GPU-CUDA-Graphs

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

192 GB RAM; 2 x Intel(R) Xeon(R) Silver 4316 CPU, NVIDIA RTX A4500 + NVIDIA A100 80GB PCIe, 256 GB RAM.

We have obtained results on every card for certain algorithm. We conducted 10 iterations for each application across different classes. The average value was calculated alongside with the standard deviation. Detailed execution times are presented for NVIDIA GeForce RTX 4070 Ti and A100 in Figure 1. Tables with more detailed information are available under the link[2].

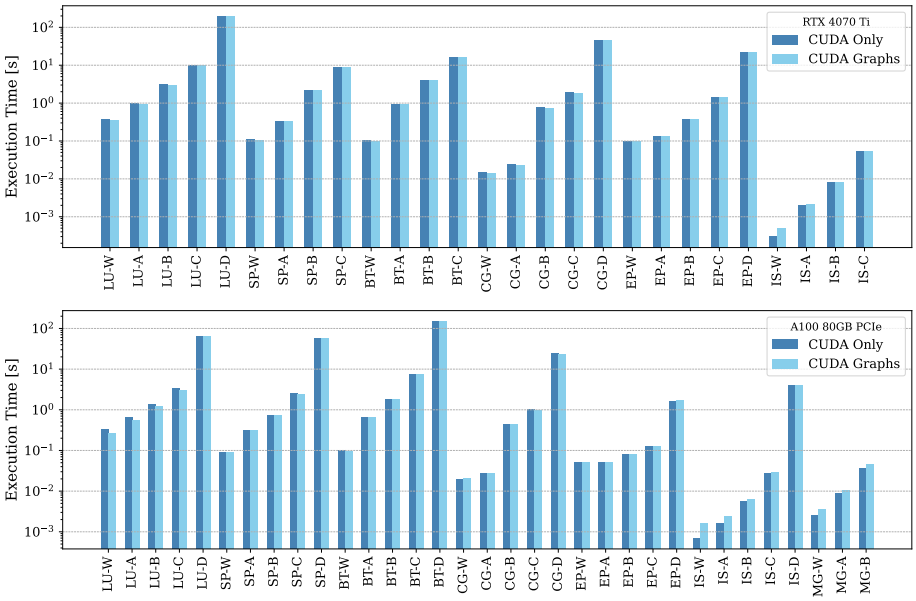


Fig. 1: Execution times per NAS Benchmarks and Class (for two cards)

For each of the cases a relative speed-up was calculated. The values of the execution time obtained for the variant that does not take into account the implementation of CUDA Graphs, were considered as the relative value.

In the following analysis, we focus on the comparison mainly of the larger classes of the benchmarks and assume that gains/losses within [-1,+1]% range are considered non-conclusive. This is because, for these classes, for the vast majority of cases, standard deviation in mean (percentage) is also within the [0,1]% range, although very often significantly below 1%. In very rare cases it exceeds 1% but it is for relatively short execution times, for these classes. Instance time is the execution time of a single kernel instance, and instance number is the number of kernel instances that have been executed for a given algorithm. These values have been weighted by the proportion of total execution

[2] https://cdn.files.pg.edu.pl/eti/KASK/CUDA_Graphs_additional/CUDA_Graphs_paper_detailed_data.pdf

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

time that each kernel consumes. Data relevant to configurations giving the best results are shown under the link[2]. Weighted values are used for axes of the following figures.

Speed-ups for the tested configurations are shown in Figure 2. It can be observed that gains in application duration were achieved primarily in cases where there were relatively many short-lived kernels.

In the case of the NVIDIA GeForce RTX 2080, there are gains for the CG algorithm for classes B and C with values of 2.97% and 3.41%. As for the other algorithms, LU, SP, IS, BT and EP reach values that are close to zero. On the other hand, one can notice a negative value for the MG (-9.44%) algorithm. In the latter case, however, absolute execution times are very low, in which case additional overheads might not be mitigated by potential benefits.

For the NVIDIA GeForce RTX 4070 Ti, for the LU algorithm, the profit values are: 0.36%, 1.58% and 2.57%, with the value of 0.36% again being too small to make a conclusion. For the CG algorithm, the results are: 2.89%, 3.83% and 4.13%. The values for the IS, SP and BT algorithms are again too small for conclusive statements. The MG algorithm (-5.60%) again shows a negative value, but its execution times are very small in absolute terms.

For the NVIDIA Quadro RTX 8000, there are consistent gains for CG (2.11%, 3.46% and 3.84%). The BT (-1.34% and -4.08%) and MG (-8.56%) algorithms are characterized by slight negative values and in this case the result of the IS algorithm for one of the computing classes (-3.88%) also signals a decrease in the value of the application duration.

For the NVIDIA A100 80GB PCIe, the noticeable difference from the previous GPUs is that the LU algorithm here features significant gains in application time with values of 1.45%, 8.05% and 11.87%. With the values decreasing as the weighted instance time of a single instance increases. For the CG algorithm, gains for two classes amount to 2.78% and 3.56%, similarly to the results on the other GPUs. Negative values, on the other hand, are visible for MG (-19.98%) and IS algorithms for two classes (-2.27% and -10.28%). The remaining cases have values that are too small to be considered significant.

For all the cards, the MG and IS algorithms are characterized by short execution times, even for larger computational classes (B and C). In most cases, the execution times of these applications, in their unmodified forms, do not exceed 1 second. For class D and NVIDIA Quadro RTX 8000, these times are 15.42 seconds for the IS algorithm and 17.04 seconds for MG. In comparison, the LU algorithm for class D on the same card has an execution time of 301.76 seconds, and the CG algorithm 273.48 seconds. On NVIDIA A100 80GB PCIe, where the MG algorithm achieves a time of 6.17 seconds and IS 3.96 seconds. For the LU and CG algorithms, these times are 66.18 seconds and 24.16 seconds, respectively. For both of these cards, these times are of different orders of magnitude. With such short execution times, the implementation of CUDA Graphs may introduce some fixed overheads and could slow down the application.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

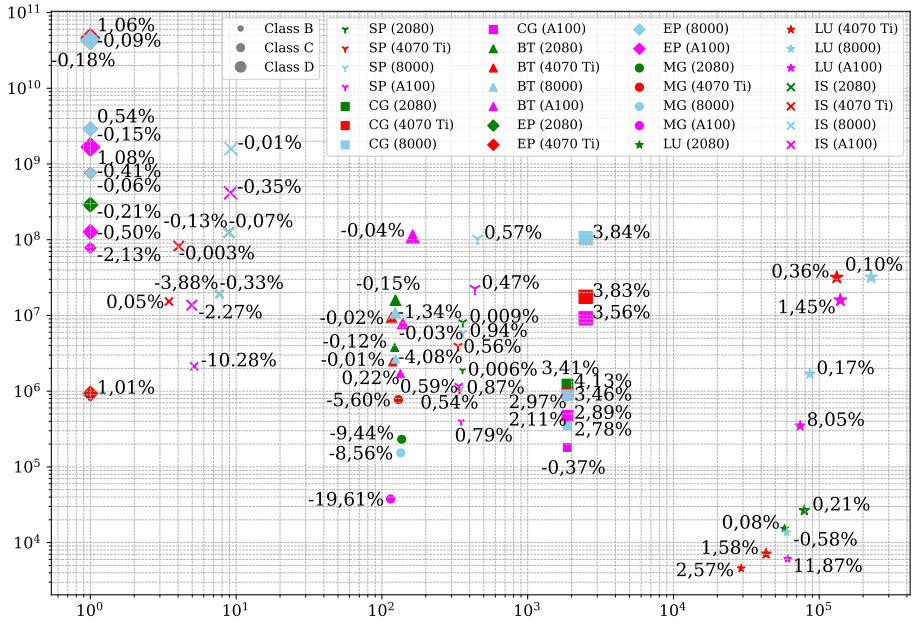


Fig. 2: Speed-ups, Weighted Instance Time in relation to Weighted Instance Number for every GPU

## 5 Summary and Future Work

This work is one of the first to explicitly compare CUDA and CUDA Graphs. This paper presents an experimental study based on the GPU implementation of NAS Parallel Benchmark. The research presented in this work includes the evaluation of the aforementioned benchmarks in their unaltered form on NVIDIA GeForce RTX 2080, NVIDIA GeForce RTX 4070 Ti, NVIDIA Quadro RTX 8000, and NVIDIA A100 80GB PCIe high-performance cards. Usage of CUDA Graphs was then incorporated by us into all applications comprising the NAS Parallel Benchmarks, followed by reevaluation on the same hardware, which allowed to compute a relative speed-up. The CG algorithm shows the most consistent gains across various cases. Depending on the card, CUDA Graphs yielded gains of 2 to over 4%. For LU, gains were largest on the A100 and reached even 11.87%. For all of the aforementioned cards, low negative gain values were observed for the MG algorithm (-9.44%, -5.60%, -8.56%, and -19.61%) that we attribute to small absolute running times and impact of fixed overheads of CUDA Graphs. It can be concluded that the CG algorithm consistently yields gains, and the LU algorithm shows improvements in about half of the analyzed cases. Other benchmarks resulted in execution times similar for both cases.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16

For the future, detailed investigation of the GPU architecture impact could be performed, along with integration of CUDA Graphs into a higher-level DAG processing framework in a cluster [10].

## References


1. Araujo, G., Griebler, D., Rockenbach, D.A., Danelutto, M., Fernandes, L.G.: NAS Parallel Benchmarks with CUDA and beyond. Software: Practice and Experience (2023)
2. Araujo, G.A., Griebler, D., Danelutto, M., Fernandes, L.G.: Efficient NAS Parallel Benchmark kernels with CUDA. In: 2020 28th Euromicro international conference on parallel, distributed and network-based processing (PDP). IEEE (2020)
3. Demirsu, M., Lervik, A.: Evaluating the performance of CUDA Graphs in common GPGPU programming patterns (June 2023), school of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, https://www.diva-portal.org/smash/get/diva2:1779194/FULLTEXT01.pdf
4. Gray, A.: Getting Started with CUDA Graphs (September 2019), NVIDIA Developer, Technical Blog, https://developer.nvidia.com/blog/cuda-graphs/
5. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 3rd edn. (2016)
6. Nguyen, V., Carilli, M., Eryilmaz, S.B., Singh, V., Lin, M., Gimelshein, N., Desmaison, A., Yang, E.: Accelerating PyTorch with CUDA Graphs (October 2021), https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/
7. NVIDIA Corporation: CUDA Graphs API (2019), `https://developer.nvidia.com/cuda-graphs`, NVIDIA Developer Documentation
8. NVIDIA Corporation: Cuda c++ best practices guide (September 2024), https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
9. Qiao, B., Özkan, M.A., Teich, J., Hannig, F.: The best of both worlds: combining cuda graph with an image processing dsl. In: Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference. DAC '20, IEEE Press (2020)
10. Rościszewski, P., Czarnul, P., Lewandowski, R., Schally-Kacprzak, M.: Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with cpus and gpus. Concurrency and Computation: Practice and Experience **28**(9), 2586–2607 (2016). `https://doi.org/https://doi.org/10.1002/cpe.3719`, `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3719`
11. Tu, J.: Constructing CUDA Graphs with Dynamic Parameters (August 2022), NVIDIA Developer, Technical Blog, https://developer.nvidia.com/blog/constructing-cuda-graphs-with-dynamic-parameters/
12. Yu, C., Royuela, S., Quiñones, E.: OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. In: Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems. p. 42–47. SCOPES '20, Association for Computing Machinery, New York, NY, USA (2020). `https://doi.org/10.1145/3378678.3391881`, `https://doi.org/10.1145/3378678.3391881`


ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97635-3_16