Received: 31 May 2021 | Revised: 23 October 2021 | Accepted: 5 November 2021

DOI: 10.1002/spe.3056

SPECIAL ISSUE PAPER

WILEY

NAS Parallel Benchmarks with CUDA and beyond

Gabriell Araujo[1] | **Dalvan Griebler**[1,2] | **Dinei A. Rockenbach**[1,2] | **Marco Danelutto**[3] | **Luiz G. Fernandes**[1]

[1] School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

[2] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Educational Society (Setrem), Três de Maio, Brazil

[3] Department of Computer Science, University of Pisa, Pisa, Italy

Correspondence
Gabriell Araujo and Dalvan Griebler, School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre 90619-900, Brazil.
gabriell.araujo@edu.pucrs.br and dalvan.griebler@pucrs.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Number: 437693/2018-0; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Grant/Award Number: 001; Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, Grant/Award Numbers: 19/2551-0001895-9, 21/2551-0000725-7

Abstract

NAS Parallel Benchmarks (NPB) is a standard benchmark suite used in the evaluation of parallel hardware and software. Several research efforts from academia have made these benchmarks available with different parallel programming models beyond the original versions with OpenMP and MPI. This work joins these research efforts by providing a new CUDA implementation for NPB. Our contribution covers different aspects beyond the implementation. First, we define design principles based on the best programming practices for GPUs and apply them to each benchmark using CUDA. Second, we provide ease of use parametrization support for configuring the number of threads per block in our version. Third, we conduct a broad study on the impact of the number of threads per block in the benchmarks. Fourth, we propose and evaluate five strategies for helping to find a better number of threads per block configuration. The results have revealed relevant performance improvement solely by changing the number of threads per block, showing performance improvements from 8% up to 717% among the benchmarks. Fifth, we conduct a comparative analysis with the literature, evaluating performance, memory consumption, code refactoring required, and parallelism implementations. The performance results have shown up to 267% improvements over the best benchmarks versions available. We also observe the best and worst design choices, concerning code size and the performance trade-off. Lastly, we highlight the challenges of implementing parallel CFD applications for GPUs and how the computations impact the GPU's behavior.

KEYWORDS

graphics processing units, high-performance computing, NPB, parallel applications, parallel programming, performance analysis

1 | INTRODUCTION

Graphics processing units (GPUs) were originally designed to supply the growing demands of the gaming industry. These specialized boards offer efficient computation and high throughput in floating point operations per second (FLOPs) by means of massive parallelism. The processing power offered by this execution model sparked interest in many areas, such

Abbreviations: CFD, computational fluid dynamics; CG, conjugate gradient; FLOPs, floating point operations per second; GPUs, graphics processing units.

as artificial intelligence, data science, physics simulation, computational fluid dynamics, and robotics. These new applications largely increased the popularity of the GPUs, up to the point that nowadays they are included into almost every modern computational system, including notebooks, smartphones, embedded systems, and supercomputers. However, efficiently using the massive parallelism available in the GPUs is a challenging task because it requires the developer to use a different programming paradigm and usually involves architecture- and application-specific optimizations.

Benchmarks are used to overcome hardware and application differences in order to study and promote programming techniques and strategies that can be used to exploit the available hardware. One of the most used benchmarks suites to evaluate parallel software and hardware is the NAS Parallel Benchmarks (NPB).[1] It was developed by the NASA Advanced Supercomputing Division. The NPB suite consists of five kernels and three pseudo-applications that mimic features such as data movement and intensive numeric computations present in the domain of computational fluid dynamics (CFD). The benchmarks are similar to real applications, supporting with scientific reports, different workloads, and tests for correctness. NPB was originally developed in Fortran and consisted of two versions: a serial version[1] for reference and an OpenMP version[2] targeting multi-core architectures. Later, NASA introduced newer versions targeting different architectures: a version for clusters with MPI named NPB-MPI,[3] a version for hybrid programming named NPB Multi-zone (NPB-MZ),[4] and a version targeting grid computing named as NAS Grid Benchmarks (NGB).[5] The popularity of NPB led researchers to port it to other languages, such as C++,[6] including different parallel APIs such as FastFlow and TBB.[6]

Given the pervasiveness of GPUs and the importance of the NPB suite in the area of high performance computing and parallel programming, it is of much interest to investigate this set of benchmarks for GPUs. Although it was originally focused on CPU architectures, the NPB suite was already implemented for GPU architectures using OpenACC,[7] OpenCL,[8,9] and more recently CUDA.[9,10] CUDA is the standard way to exploit the parallelism of GPUs from NVIDIA, which is currently the single biggest manufacturer of GPUs. In addition to re-implementing the five NPB kernels, our goal is to provide new CUDA versions for the three NPB's pseudo-applications as well as provide a set of parallel design principles for GPUs. These design principles were based on our expertise of years with GPU programming in the research laboratory and on a careful analysis of the GPU literature, which includes books and scientific papers. These can be viewed as a new guideline for GPU developers. Moreover, we conduct a deeper performance analysis, including a comparative study with other implementations. Since we provided support for configuring more parameters in NPB, it allows us to investigate different configurations with CUDA, revealing new insights.

Given this context, our research is motivated by the following questions: (1) How much the GPU performance is impacted by design principles and optimizations? (2) What are the challenges when implementing parallel CFD applications for GPUs with CUDA? (3) How much does the different numbers of threads per block impacts the performance of NPB programs in CUDA? and (4) Which is the best number of threads per block for a specific hardware architecture and benchmark?

We provide the following scientific contributions:

- **A new parallel version for NPB kernels and pseudo-applications with CUDA**. In addition to re-implementing the five kernels of our previous work[10] using new design principles and parallelism strategies, we also provide the three NPB pseudo-applications. With respect to the other approaches, our CUDA version reached performance improvements up to 267% over the best benchmarks versions available.
- **A set of design principles to guide GPU developers for efficient parallelism exploitation**. This is a guideline for developers to efficiently exploit parallelism on GPUs. We evaluated these design principles in NPB and compared with other studies.
- **A parametrization support for configuring the number of threads per block in our CUDA version of NPB**. NPB users can increase the performance analysis space by configuring the number of threads per block without changing the source code to test how the benchmark performs in different GPU architectures and benchmarks.
- **A discussion and comparison among our and other NPB parallel versions for GPUs**. We highlighted the differences of parallel programming approaches, techniques, and strategies as well as the number of source lines of code (SLOC) required.
- **An evaluation of the performance impacts for different configurations of the number of threads per block**. These experiments provide new insights and a deeper understanding about the NPB programs with CUDA running on GPUs.

- **A set of strategies for helping to choose a better number of threads per block for a specific hardware**. Finding the best configuration is not a trivial task. We provide experiments evaluating the advantages and disadvantages of five different strategies, and discuss their performance.
- **A performance comparison with other NPB GPU implementations**. A set of experiments was conducted comparing our version of NPB in CUDA with the available implementations of NPB for GPUs made by other works in the academia.

The article is organized as follows. Section 2 describes our implementation of the NPB programs for GPUs. Section 3 presents the experiments, their methodology, and the obtained results. Section 4describes the related work. Finally, we present our conclusions in Section 5.

2 | NPB PROGRAMS

NPB is a consolidated set of benchmarks consisting of five kernels and three pseudo-applications. All eight benchmarks mimic computations present in the domain of CFD. While the kernels reproduce cores of five different numerical methods present in CFD applications, the pseudo-applications reproduce full data movement and computations, being larger and more complex. The original NPB version was written in Fortran language targeting CPU architectures. Later, other versions for clusters,[3] hybrid programming,[4] and grid computing[5] were released. The NPB suite provides a set of workloads called classes, which are ordered by the size, S, W, A, B, C, D, E, and F. The classes S and W are used for simple tests while the other classes are used for experiments. The full description of the workloads can be found at the NPB website.[11] NPB also provides tests of correctness integrated into the benchmarks that are specially useful when applying parallelism to ensure correctness.

Figure 1 presents a flowchart of the NPB programs. Each benchmark is composed of functions (presented as rounded rectangles with solid lines) and routines (dotted rectangles). The most expensive function of each benchmark in terms of time spent to compute is presented with dashed blue borders while the functions with green background are the functions that we offloaded to the GPU in our CUDA implementation. From left to right, the first five benchmarks are the kernels (CG, EP, FT, IS, and MG), and the last three benchmarks are the pseudo-applications (BT, LU, and SP). Each benchmark starts with an initialization step and finishes with a verification procedure that ensures the correctness of the result. Between the initialization and the verification steps, the benchmark performs the main calculations by calling the most expensive functions, usually inside an iterative process. EP benchmark is different since the most expensive function is not inside an iteration. In addition, EP and IS are the only benchmarks where the main computations are inside a single

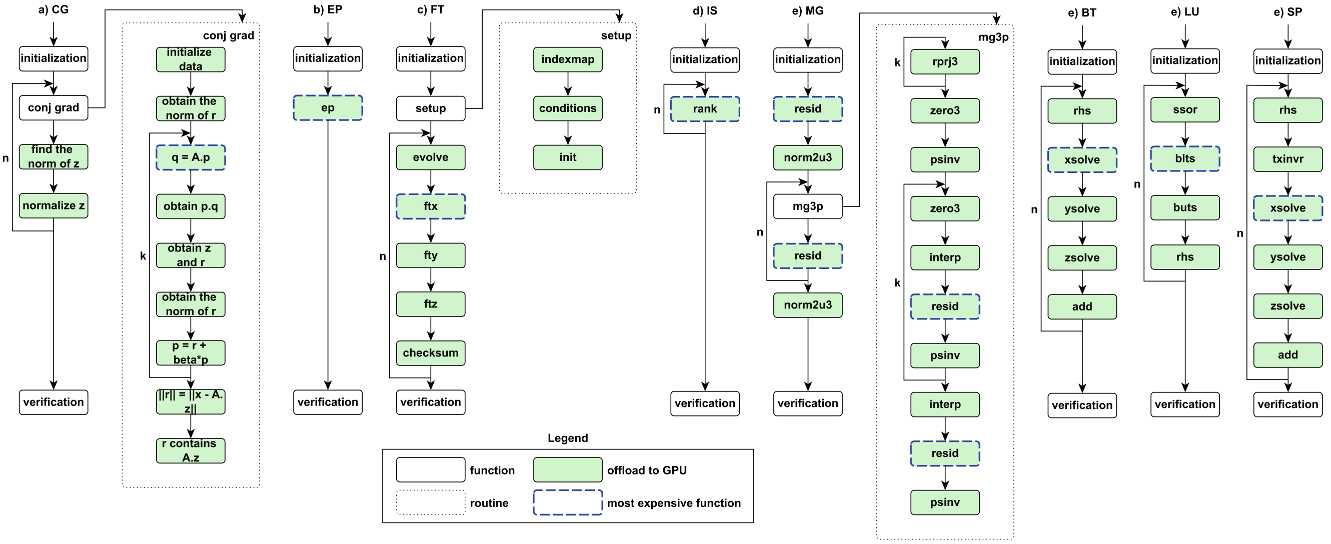


FIGURE 1 NPB programs' flowchart

function. The measurement of the total execution time starts right after the initialization and ends when the verification function is called.

Overall, the NPB suite requires a large effort to efficiently port computations to GPUs.[12] There are lots of routines with complex execution flows. The programs present different characteristics that are challenging to port to GPUs, such as tasks with irregular computation, computations on multi-dimensional arrays composed by up to five dimensions (which require different data access patterns to improve the GPU performance), data dependencies (which require strategies to eliminate the dependencies or at least to reduce the performance hit), large chunks of data that does not fit in the GPU memory. These challenges often require a complete refactoring of the source code in order to achieve good performance numbers, for which the programmer must possess a deep knowledge about the specific application algorithms and about GPU programming.

Our CUDA version was implemented using a C++ conversion from the NPB 3.4,[6] which follows strictly the original Fortran version. We checked to ensure that our CUDA version passed in all the NPB correctness tests. Our source codes are available in the following GitHub repository: https://github.com/GMAP/NPB-GPU. In the next sections we describe the design principles adopted in our CUDA version of the NPB (Section 2.1), describe the main loop transformations used when refactoring the NPB programs (Section 2.2), present the main aspects of the kernels (Section 2.3), discuss in details the pseudo-applications (Section 2.4), compare implementations of the literature (Section 2.5), and briefly describe the parameters for configuring the number of threads per block in the benchmarks (Section 2.6).

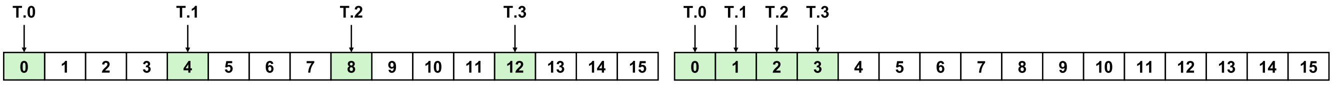
2.1 | Design principles

Parallel programming for GPUs is not a trivial task and it is even more difficult to efficiently exploit the GPU parallel resources. During the last years working on GPU parallel programming, we have faced different problems and issues, where only a small portion of them were actually reported by other research papers. We have several lessons learned that could help other GPU developers to achieve better performance in their GPU applications. Therefore, we are proposing GPU design principles that serve as a guideline for helping developers achieve better application performance on GPUs.

Our design principles were built based on our expertise and experience on GPU parallel programming and on a careful analysis of the GPU literature. The analysis of the literature included books, documentations, and scientific papers (including our previous work[10]). From the books and documentations, we collected mainly the concepts of GPU architectures, their programmability, and instructions on how to port legacy code in GPUs to best fit the architecture of those accelerators. Not all of these recommendations actually worked in practice. For instance, CUDA's design guide recommends using a small amount of threads per block when a function offloaded to the GPU has several barriers,[13] however, our experiments show that a small number of threads per block increases the overhead of synchronizations, imposing a larger overhead. In contrast, a larger number of threads per block, decreases the amount of synchronizations and improve the performance, as we observed in our experiments (Section 3.1). With respect to scientific papers, we analyzed the findings of other authors at porting legacy code and investigating GPU programming techniques and their performance. The papers often present more empiric approaches than books, which helped us to analyze scenarios and observe the effects of different kinds of computations when ported to GPUs. Unfortunately, findings such as the author's choices and implementation techniques to deal with each scenario presented some inconsistencies and drive developers to bad choices in terms of performance.

Our design principles organize the concepts, techniques, and best practices applied for GPU programming and provide a concise guideline. Our inspiration comes from the design patterns from Software Engineering.[14,15] We expect that developers applying it in different algorithms and GPU architectures, will drive their parallel code implementations to results as the ones presented in NPB. Take into account that when approaching an application to implement GPU parallelism, the programmer must analyze the algorithms to decide how to apply our principles, as the GPU performance depends on several variables. The design principles are briefly described as follows:

1. **Avoid offloading complex functions to the GPU**. Given that the GPU's simpler processing cores benefits from massive parallelism with few complex instructions, it is important to keep the offloaded functions as simple as possible to extract the best performance from the GPU execution model. Therefore, the programmer should avoid offloading complex instruction flows with many loops, branches, synchronizations, jumps over different blocks of memory, or excessive use of registers.



(A) Uncoalesced access pattern.

(B) Coalesced access pattern.

FIGURE 2 Illustrating memory coalescing patterns. (A) Uncoalesced access pattern, (B) coalesced access pattern

2. **Avoid branch divergences**. GPU threads are grouped in warps that execute a single common instruction at each clock cycle. Thus, when a branch divergence occurs, only a single path is executed and the diverging threads (following other paths) are kept inactive waiting for their path to be executed.
3. **Avoid global memory accesses**. The GPU global memory is off the chip and has high access latency. To extract the best of GPU performance it is important to reduce input and output (I/O) times by using other memory levels that offer lower latencies such as local memory, shared memory, and constant memory. The local memory is off chip, but its accesses are automatically coalesced by the GPU to improve the bandwidth. Shared memory is on the chip and offers lower latency access. Constant memory is a read only memory also inside the chip.
4. **Avoid memory transfers**. Transferring data between the host (CPU) and the device (GPU) memories are very expensive tasks in terms of time. When performing computations in the GPU, even if some specific routine during the computation runs faster on the CPU, it is preferable to run them on the GPU because it would be more expensive to copy data from the GPU to the host, perform the computation, and copy the resulting data back to the GPU afterwards.
5. **Avoid synchronizations**. Synchronizations (also known as barriers) impose performance degradation. Even in the cases when they are inevitable, they should be delayed as long as possible.
6. **Support memory coalescing**. Coalesced accesses happens when the GPU threads access contiguous positions in the memory. The GPU detects the coalesced pattern, and the memory operation is executed in a single clock cycle, loading the whole block of memory accessed by the threads. Figure 2 illustrates an example of memory coalescing. Figure 2A illustrates an array of sixteen positions of memory being accessed by four threads (T.0, T.1, T.2, and T.3), however every thread must issue a separate load instruction, since they are accessing different blocks of memory. In contrast, Figure 2B illustrates the same four threads accessing contiguous positions of memory. Thus the load instruction is executed by a single thread and loads the whole block of memory requested by all threads in a single clock cycle.
7. **Explore as much parallelism as possible**. GPUs are massively parallel accelerators, thus it is essential to expose as much parallelism as possible. Moreover, massive parallelism helps hiding thread latency. This optimization kicks in when a GPU thread requests an operation that take several clock cycles to be completed (e.g., branch instructions or memory accesses). The thread must wait until the operation is finished, then another thread is selected by the scheduler to be executed. This mechanism helps avoiding idle time in the GPU cores, exploiting the hardware resources more efficiently because the cores are performing computations even when some threads need to perform long-latency operations.[16,17]

2.2 | Loop transformations

CPUs have some optimizations that do not exist in GPUs (e.g., branch prediction). Moreover, the compilers optimize the loops applying different transformations such as vectorization (attempt to run loop iterations concurrently) and tiling (iterates over blocks of data to fit the cache), reducing overheads and increasing the performance of CPU cores.[16,18] GPUs have none or limited versions of such optimizations, thus loops are expensive operations for GPU threads. When programming for GPUs, manually applying techniques for loop transformations is crucial to increase the overall performance. Moreover, refactoring of loops is usually necessary to expose massive parallelism suitable to GPU offloading, which also simplifies the execution flow.[16,18] We briefly describe the main loop transformations used in the refactoring of the NPB Programs as follows:

1. **Collapsing**. Loop collapsing occurs when the iteration space of nested loops is combined into a single loop. It is useful to map computations to GPU threads and also simplifies the execution flow by reducing the amount of loops.
2. **Fission**. Loop fission occurs when a loop is divided into multiple loops. For instance, if a large and complex computation can be separated into two or more simplified functions to be offloaded to GPU, it can improve the overall performance of the GPU threads.

3. **Fusion**. Loop fusion is the opposite of fission, it occurs when the computations of different loops are grouped together into a single loop. When porting an algorithm to GPUs, fusion is useful because it reduces the amount of loops that a thread must perform. However, one must be careful about the added complexity of merging multiple code blocks into a single loop.
4. **Unrolling**. Loop unrolling occurs when the loop is replaced by many copies of the body of the loop itself, essentially removing the iterations from the code. It is useful for GPU code because this eliminates the burden of performing tests and branches as well as operations that are not optimized in GPUs. However, only loops with a fixed small number of iterations are candidates for loop unrolling.

2.3 | Kernels

In this section we briefly describe the main aspects of our CUDA implementation for the five NPB kernels. Also, analysis of the execution time of each function offloaded to the GPU using the workload class C is presented. For the number of threads per block we used the GPU warp size in order to collect the execution time of each function. We executed those experiments on a machine equipped with a processor Intel Xeon E5-2620, and a GPU NVIDIA Titan X. Since we re-implemented all benchmark kernels from our previous work,[10] we summarize the main differences as follows. First, we used dynamic shared memory instead of static shared memory in GPU. Second, we implemented different memory access patterns to boost the GPU performance based on the application's characteristics. Third, we introduced the use of atomic operations for combining results from different GPU threads.

2.3.1 | Conjugate gradient (CG)

Conjugate gradient (CG) is a benchmark characterized by irregular computations that perform the multiplication of unstructured matrices.[1,2] There is a set of tasks composed by unbalanced computations that access different blocks of memory in the most expensive function of the benchmark (`q = A.p` presented in Figure 1). Those unbalanced computations were isolated assigning each task (a range of a global array) to a block of threads. Then, the computations of a single task were distributed between the threads of a single block of threads. We also modified the data access patterns of the tasks to allow memory coalescing. Consequently, we were able to apply all design principles to this benchmark due to its characteristics of low data dependency and the possibility of changing the memory access pattern. Loop transformations were not needed because efficient parallelism is reached just by mapping tasks to blocks of threads.

We also followed this same parallelism strategy in the function `||r|| = ||x - A.z||`. The other functions are essentially composed by non-nested loops, where we simply assigned a GPU thread to each iteration. The most expensive function of CG in terms of time (`q = A.p`) takes 90.13% of the total execution time with the workload class C. In addition to being an intensive function, it is called several times. The second most expensive function is `||r|| = ||x - A.z||` which represents 3.61% of the total execution time. It is executed only a few times, although it is also intensive due to the irregular computations. The other parallelized functions are not intensive and represent only a fraction of the total execution time.

2.3.2 | Embarrassingly parallel (EP)

Embarrassingly parallel (EP) generates a set of `n` pseudo random numbers and computes Gaussian deviates for each one of them.[1,2] In our CUDA version, the `n` pseudo random numbers were splitted in subsets and each subset is assigned to a block of threads. Inside each block of threads, each thread computes a single pseudo random number. Each pseudo random number requires a very large array to complete the computations, resulting in a high memory consumption that limits the number of threads created in order to fit the GPU memory capacity. Then, we refactored the algorithm in such a way that a small array is allocated and it is reused until the computation is finished. We were able to apply each one of the design principles since EP has low data dependency and enables to change the memory access pattern. No loop transformations were required, except for the reuse of arrays that are necessary to decrease the requirements of memory consumption. Since a single function is offloaded to the GPU, it consumes 100% of the execution time.

2.3.3 | Fourier transform (FT)

Fourier Transform (FT) calculates a fast FT in 3D matrices of complex numbers.[1,2] The main computation of this benchmark is composed by the functions `ftx`, `fty`, and `ftz`, which are computing the FT in the x, y, and z dimensions. The serial algorithm is equal in all of them and contains a complex instruction flow with data dependencies and six nested loops. The algorithm performs three consecutive steps on chunks of data: (1) it reads a chunk of data; (2) computes the chunk; and (3) writes the chunk in the output. Then the steps are repeated to the other chunks. Since complex instruction flows do not perform well on GPUs, in our CUDA version we rewrote the algorithm by applying two loop transformation techniques. Using the loop fission technique, we divided the algorithm into three stages (e.g., we split `ftx`into `ftx1`, `ftx2`, and `ftx3`), and with the loop collapsing technique we removed the nested loops from each stage. Then, each stage was offloaded to the GPU to compute over the whole data instead of small chunks. Lastly, we coalesced the accesses in the computations of the function `ftx`, while `fty` and `ftz` were already coalesced. This strategy allows a higher degree of parallelism upon eliminating data dependencies, besides simplifying the original complex instruction flow.

The other functions are composed of nested loops, where we applied the loop collapsing technique and assigned a GPU thread to each iteration. In the routine `setup`, the functions `indexmap`, `conditions`, and `init` are offloaded concurrently to the GPU because there are no data dependencies between them. In this benchmark we were able to apply all of the design principles. Although FT has several data dependencies, they can be eliminated through refactoring, allowing a high degree of parallelism. Coalescing the accesses in this benchmark is more complex than in CG and EP because FT compute 3D arrays, while CG and EP operate over linear arrays. This requires more effort for calculating contiguous accesses for the GPU threads in the routines. Loop transformations that we implemented are loop collapsing to increase the degree of parallelism and loop fission to decrease the amount of loops and branches.

The most expensive functions of FT are the second stage from `ftx`, `fty`, and `ftz`, representing 25% of the execution time each. Functions for loading or writing data are less expensive and spend less than 5% of the execution time each, which includes `evolve`, and the first and the third stages from `ftx`, `fty`, and `ftz`.

2.3.4 | Integer sort (IS)

Integer sort (IS) sorts integer numbers.[1,2] The `rank` function is composed by a sequence of non-nested loops, where we applied the parallelism assigning a GPU thread to each iteration from each loop. A synchronization between the CPU and the GPU is required after the execution of each loop offloaded to the GPU. We applied all of the design principles because it does not have data dependencies and offers the possibility to change the memory access pattern. However, the benchmark is not computationally intensive and requires several synchronizations, resulting in low GPU usage. No loop transformations were required as it is just a matter of assigning a GPU thread to a single iteration of the non-nested loops. Since the IS benchmark is composed of a single function, it represents 100% of the execution time.

2.3.5 | Multi-grid (MG)

Multi-grid (MG) is described by NASA reports as a simplified multi-grid benchmark.[1,2] The MG main routine, `mg3p`, executes the `interp`, `psinv`, `resid`, `rprj3`, and `zero3` functions. The computations of `psinv` and `resid` are double nested loops that perform a sequence of non-nested loops. These are the two most expensive functions and represent 25.33% and 50.41% of the total execution time, respectively. In our approach, we applied the loop collapsing technique in the double nested loops, and assigned a GPU thread to each iteration, and then each thread performs a sequence of non-nested loops. The computations of `interp` and `rprj3` also have double nested loops that perform a sequence of loops and represent 9.02% and 10.94% of the total execution time, respectively. However, these loops have the same iteration space and no dependencies between the data. We then applied a loop fusion, grouping the computations into a single non-nested loop. With this refactoring, the execution flow is transformed into a triple nested loop. We applied collapsing in the triple nested loop and assigned a GPU thread to each iteration, offloading to the GPU computations without loops and no branch divergences.

The functions `zero3` and `comm3` (`comm3` is a routine that is executed at the end of other functions, such as `resid`, `rprj3`, `psinv`, and `interp`) only update values from arrays. We applied the parallelism in these two functions by assigning a GPU thread to each array index. The function `norm2u3` combines results where we implemented a binary tree parallel reduce to perform the computations on the GPU.[16] Finally, we applied memory coalescing in all the functions offloaded to the GPU. We were able to apply all of the design principles in this benchmark due to the low data dependency and possibility that data access can be grouped in the memory. Applying the memory coalescing required a large effort similar to the FT benchmark, because MG also computes 3D arrays. Implementing the design principles required the use of loop transformations, mainly loop collapsing for increasing the degree of parallelism and loop fusion for decreasing the amount of loops and branches.

2.4 | Pseudo-applications

In the next subsections we describe our CUDA implementation for the NPB pseudo-applications. We also present an analysis of the execution time for each function offloaded to the GPU, using the workload class C and the GPU warp size as the number of threads per block.

2.4.1 | Block tri-diagonal solver (BT)

This benchmark is a block tri-diagonal solver for Navier–Stokes equations.[1,2] The most expensive function from BT is `xsolve` that is a 3D solver. The algorithm from the `xsolve` function has double nested loops that perform three stages of computations: (1) initializes the data; (2) performs a sequence of non-nested loops that do not have data dependencies; and (3) performs a sequence of loops that have data dependencies.

In our CUDA version, we splitted the algorithm inside the `xsolve` function into three stages applying the loop fission technique. The first stage initializes the data, in which there are double nested loops. We applied loop collapsing in the double nested loop and assigned a GPU thread to each iteration compute independently. This stage represents 0.04% of the total execution time. The second stage has double nested loops performing a sequence of loops that have no dependencies between the data. This feature allowed us to apply the loop fusion technique in this sequence of loops, grouping the computations into a single loop. At this step of refactoring, the second stage was transformed into a triple nested loop. After, we applied loop collapsing and assigned a GPU thread to each iteration of the final loop to compute independently. This stage represents 8.33% of the total execution time. The third stage has double nested loops performing a sequence of loops that have dependencies between the data being processed. Those dependencies are a constraint for applying the loop fusion technique in the same way it was done in the second stage. Then, we only applied collapsing in the double nested loops and assigned a GPU thread to each iteration, where each GPU thread performs the sequence of the non-nested loops with data dependencies. This stage represents 29.06% of the total execution time.

The functions `ysolve` and `zsolve` are composed by the same algorithm of `xsolve`. Therefore, we applied the same parallelism strategy adopted in the `xsolve` function. The third stage of `ysolve` and `zsolve` represents 27.68% and 15.42% of the total execution time, respectively, while the second stages represent a little more than 5% each. Therefore, over 90% of the execution time is distributed among these three routines (`xsolve`, `ysolve`, and `zsolve`). The function `rhs` has a large amount of code composed by a double nested loop that performs a sequence of loops. In our approach, we applied the loop fission technique and splitted the computations into nine small stages, where each stage is composed by a triple nested loop. Then, we applied loop collapsing in each triple nested loop and assigned a GPU thread to each iteration. The function `add` has a nested loop where we also applied loop collapsing and assigned a GPU thread to each iteration compute independently. Moreover, we applied memory coalescing pattern in all functions that were offloaded to the GPU.

This benchmark has data dependencies and a complex instruction flow, however, we were able to apply all of the design principles after refactoring the application for eliminating data dependencies and reducing the amount of long latency operations. Applying memory coalescing pattern in BT was much more complex than in the kernel benchmarks, because this benchmark performs computations over 5D arrays. The amount of refactoring was also larger than in the kernel benchmarks while BT widely required all of the main loop transformations (collapsing, fission, fusion, and unrolling). Loop unrolling is specially useful in the pseudo-applications as almost all functions in the pseudo-applications have loops with a small fixed number of iterations.

2.4.2 | Lower-upper Gauss–Seidel solver (LU)

Lower-Upper Gauss-Seidel (LU) is a solver for a seven-block-diagonal system.[1,2] The most computational expensive function from LU is `blts`, which contains wave-front data dependencies[2] and represents 32.44% of the total execution time.

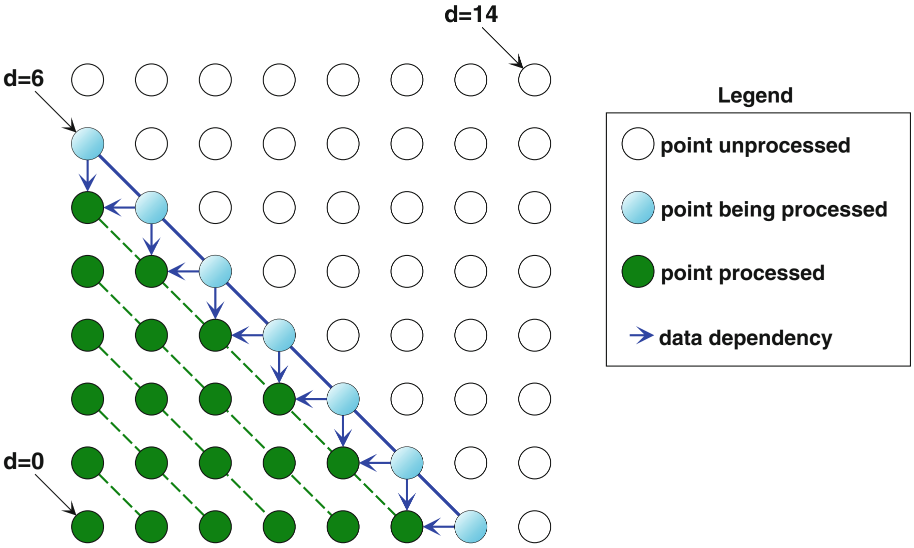


FIGURE 3 LU's `blts` algorithm

The algorithm is composed by `n` iterations, where each iteration computes the points of a diagonal from the matrix and each iteration depends on the results from the previous iteration. The points of a diagonal can be computed in parallel, however, two diagonals can not compute concurrently, because computing the points of a diagonal requires the values from the points of the previous diagonal. Figure 3 illustrates how the computations from `blts` are performed in a matrix of 8 × 8. The direction of the computations is from left to right, totaling 15 diagonals (d) of points to be computed. The diagonals from 0 ($d = 0$) to 5 ($d = 5$) are already computed and the algorithm is computing the points from the diagonal 6, while diagonals 7 to 14 wait to be computed.

The algorithm from `blts` is composed by two nested loops, where the outermost loop must be executed sequentially (it iterates over the diagonals of a matrix) and the innermost loop can be executed in parallel (it iterates over the points of a diagonal). As previously discussed, a synchronization must be performed before the execution of the innermost loop due to the data dependencies. In our CUDA approach, we offloaded to GPU a routine for each diagonal of the matrix, where each GPU thread is assigned to a single point of the diagonal. This way, the GPU threads does not perform expensive operations such as branches or synchronizations. We performed a synchronization between the CPU and the GPU after offloading a routine to the GPU. Since the algorithm always processes a diagonal of a matrix, we were not able to modify the memory access patterns.

The function `buts` has the same algorithm of `blts`, so we applied the same parallelism strategy. It represents 30.63% of the total execution time. The function `ssor` is composed by nested loops and updates the values of the arrays used in the benchmark. We applied the parallelism by collapsing the loops and assigning a GPU thread to each iteration, where each thread is mapped to a position of the array. Also, we divide the function in two routines (`ssor1` and `ssor2`), and each one of them updates different arrays. The `rhs` function is composed by a double nested loop that performs a sequence of non-nested loops. In our approach, we applied the loop fission technique and broke the routine in four stages. Then, we applied the loop fusion technique in each stage of computation to reduce the amount of loops. After these transformations, we applied the loop collapsing technique and assigned a GPU thread to each iteration. These four stages of the `rhs` function account for 31.52% of the total execution time.

This benchmark has complex data dependencies and barriers, but we were able to apply almost all of our design principles. Unfortunately, it was not possible to implement memory coalescing access pattern because this benchmark computes the diagonals of matrices. Similar to BT, the LU benchmark also required all loop transformations (collapsing, fission, fusion, and unrolling) in order to efficiently offload computations for GPU.

2.4.3 | Scalar penta-diagonal solver (SP)

Scalar penta-diagonal (SP) is a block penta-diagonal solver.[1,2] The most computationally intensive function is `xsolve`, which represents 35.99% of the total execution time. This algorithm contains a loop, where each iteration performs a sequence of double nested loops. In our approach we refactored the algorithm so that we have two nested loops that perform a sequence of loops. Then, we collapsed the two outer nested loops and assigned a GPU thread to each iteration of this new collapsed loop, where each thread performs a sequence of loops. The functions `ysolve` (14.72% of the execution time) and `zsolve` (16.52% of the execution time) follows the same algorithm and we applied the same parallelism strategy for them. The functions `txinvr` and `add` are composed by nested loops, where we applied loop collapsing and assigned a GPU thread to each iteration. In the `rhs` function from SP, there is a double nested loop that performs a sequence of loops. In our approach, we applied a loop fission to split the computations in two stages, which represent 3.25% and 20.69% of the total execution time, respectively. The first stage has a triple nested loop, where we applied a loop collapsing and assigned a GPU thread to each iteration. The second stage still has a double nested loop that performs a sequence of loops, so we applied the loop fusion technique to combine the sequence of loops in a single loop. The result of this refactoring is a triple nested loop, where we applied the loop collapsing transformation and assigned a GPU thread to each iteration. Finally, we also applied coalesced access patterns to all functions offloaded to the GPU. The second stage of the `rhs` function does not perform branch instructions, however the grouping of different routines (through the loop fusion) requires it to access different blocks of memory, which hinders memory coalescing access pattern.

The SP benchmark has a complex instruction flow for GPU threads with lots of loops and branches, but we were able to apply all of our design principles after refactoring the routines. Applying coalesced access patterns is as difficult as in BT because SP also computes 5D arrays. SP also widely required all of the main loop transformations (collapsing, fission, fusion, and unrolling) for efficient porting the routines to the GPU.

2.5 | Comparison with other parallel implementations available in the literature

Different ways of describing the GPU parallelism implementations are presented in the literature. For comparison purpose and since they were used for different GPU programming models, we adopt the definition of Do et al.[9] We do not agree with the term "optimiztions" used by the authors because they are not actually all implementations that optimize the performance achieved by the GPU. Therefore, we enumerated and referred to them as the 15 GPU implementations that are shortly described below:

1. Transform loops (simple or nested) into a function to be offloaded to the GPU.
2. Use loop fission (as described in Section 2.2).
3. Transform a block of code into a loop (this is the opposite of the loop unrolling technique described in Section 2.2).
4. Implement parallel reduce using a binary tree reduction.[16]
5. Use a variation of a parallel binary tree, where intermediate results of the array are not discarded.[16] Also known as parallel prefix sum.
6. Insert synchronization points to preserve the correctness inside a loop with dependencies.
7. Use local or shared memory, instead of global memory (this is related to our third design principle presented in Section 2.1).
8. Refactor an algorithm to allow the use of shared memory.
9. Use loop fusion (as described in Section 2.2).
10. Use memory coalescing (this is related to our sixth design principle presented in Section 2.1).
11. Transform the memory layout to improve memory access times.
12. Use synchronization between the threads of a block instead of synchronizations between the CPU and the GPU (this is only applicable when the computation can be performed by a single block of threads).
13. Use data tiling on global memory to fit the GPU memory capacity.[16]
14. Overlap data copies and computation, that is, offload computations to the GPU and transfer data from the host memory to the GPU memory at the same time.[16]
15. Offload more computation to the GPU, instead of transforming data in the CPU and performing data transfer between the host memory and the GPU memory (this is related to our fourth design principle presented in Section 2.1).

Table 1 relates these 15 GPU implementations with different works.[7-9] Take into account that we carefully inspected all benchmark source codes to provide these data, which was double checked by our team to avoid mistakes. The first and second columns of the table presents the references of the compared works and the respective benchmarks. The subsequent 15 columns represent each of the implementation and shows which works have applied them. For instance,

TABLE 1 GPU implementations applied to the NPB programs

Work	Bench.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	CG	✓						✓								✓
8	CG	✓			✓			✓			✓					
9	CG	✓			✓			✓			✓			✓	✓	
Our work	CG	✓			✓			✓			✓					✓
7	EP	✓						✓								✓
8	EP	✓			✓			✓								
9	EP	✓			✓			✓								
Our work	EP	✓						✓								✓
7	FT	✓						✓			✓					✓
8	FT	✓						✓			✓					
9	FT	✓		✓	✓			✓	✓		✓			✓	✓	
Our work	FT	✓	✓		✓			✓			✓					✓
7	IS	Unavailable														
8	IS	✓				✓		✓								
9	IS	✓				✓		✓						✓	✓	
Our work	IS	✓				✓		✓								✓
7	MG	✓						✓								✓
8	MG	✓			✓			✓		✓	✓					
9	MG	✓			✓			✓		✓	✓			✓	✓	
Our work	MG	✓			✓			✓		✓	✓					✓
7	BT	✓						✓			✓					✓
8	BT	✓						✓								
9	BT	✓	✓				✓	✓		✓	✓	✓		✓	✓	✓
Our work	BT	✓	✓				✓	✓		✓	✓					✓
7	LU	✓						✓			✓					✓
8	LU	✓			✓			✓								
9	LU	✓	✓		✓			✓		✓	✓		✓	✓	✓	✓
Our work	LU	✓	✓					✓		✓	✓					✓
7	SP	✓						✓			✓					✓
8	SP	✓						✓								
9	SP	✓	✓					✓		✓	✓	✓		✓	✓	✓
Our work	SP	✓	✓					✓		✓	✓					✓

implementation 1 was applied by all of the works and for each of the eight NPB programs. In contrast, the implementation 12 was implemented only by Do et al.[9] in LU.

The oldest work by Seo et al.[8] mostly applied the considered basic GPU implementations (1, 4, and 7) while other implementations such as 10 were applied only inside the kernels. In contrast, Xu et al.[7] used partially the basic implementations 1 and 7, because they use local memory instead of shared memory. However, the authors implemented the 10th for pseudo-applications and the 15th for all the benchmarks. These two additional implementations make this work[7] more efficient than the previous one[8] in BT and SP (see more details in Section 3.4). The source code of IS benchmark was not made available, therefore, we were not able to check what implementations were applied to this program.

Do et al.[9] is the most recent and uses the largest set of implementations. It was expected since they have proposed them. However, the authors followed distinct goals compared to ours. The main goal of our approach is the application of design principles targeting mainly a high degree of parallelism and transforming the instruction flows as simple as possible for GPU threads. The NPB implementation from Do et al.[9] did 8 and 12 implementations where the refactoring increases the complexity of the instruction flow for the GPU threads and reduces the degree of parallelism (a function implementing the implementation 12 must be executed by a single block of threads). Besides that, when data tiling (the implementation 13) is applied on the GPU global memory it reduces the performance to levels that are worse than the serial code executing on CPUs (this is also observed in the results reported by Do et al.[9]).

The GPU implementation 14 that overlaps data computing and data communication was implemented combined with the implementation 13. Consequently, it harms the GPU performance as it transfers chunks of data between CPU and GPU. In the NPB programs, computing the whole data in GPU presents a better performance than computing chunks that must be transferred between the host and the GPU memories, because it increases the burden of communication. Moreover, this implementation requires the GPU threads to perform a more complex instruction flow to compute the chunks of data. The GPU implementation 11 requires the creation of routines to transform the data layout, which implies in additional computations for the GPU. As an example, a routine can be offloaded to the GPU to rotate the dimensions of a 3D array (which also requires a synchronization) so that the same access pattern is used to compute different dimensions. This also can be used to organize the data to allow memory coalescing access pattern. However, it is possible to compute different memory access patterns operating directly in the array or the array can be already created with an organization that allows memory coalescing access pattern.[16-18] The implementation 11 improves the code productivity because it requires less effort from the programmer to compute different access patterns, however, it increases the work performed by the GPU with additional routines and synchronization barriers to update the data.

Therefore, based on our design principles, we did not apply the implementations 8, 11, 12, 13, and 14 for our CUDA versions. Another main difference is that we implemented 15 in all benchmarks, avoiding memory transfer between the CPU and the GPU. The impact of our design principles in the performance will be further discussed in Section 3.4.

We also measured the number of SLOC from each work using David A. Wheeler's "SLOCCount."[19] This metric serves as a rough measure of the effort required to refactor each implementation. Figure 4 compares the SLOC numbers of each implementation of NPB discussed in this section. The groups of columns in x-axis lists the eight benchmarks in NPB, while the y-axis presents the number of SLOC. The C++ Serial and OpenMP versions were provided by Löff et al.,[6] and are named as `Löff-Serial-2021` and `Löff-OMP-2021`. The OpenACC version was provided by Xu et al.,[7] and is named as `Xu-ACC-2015`. The OpenCL versions provided by Seo et al.[8] and Do et al.[9] are named as `Seo-OCL-2011` and `Do-OCL-2019`, respectively. The CUDA version provided by Do et al.[9] is named as `Do-CUDA-2019`. Finally, our CUDA version is named as `Our-CUDA-2021`.

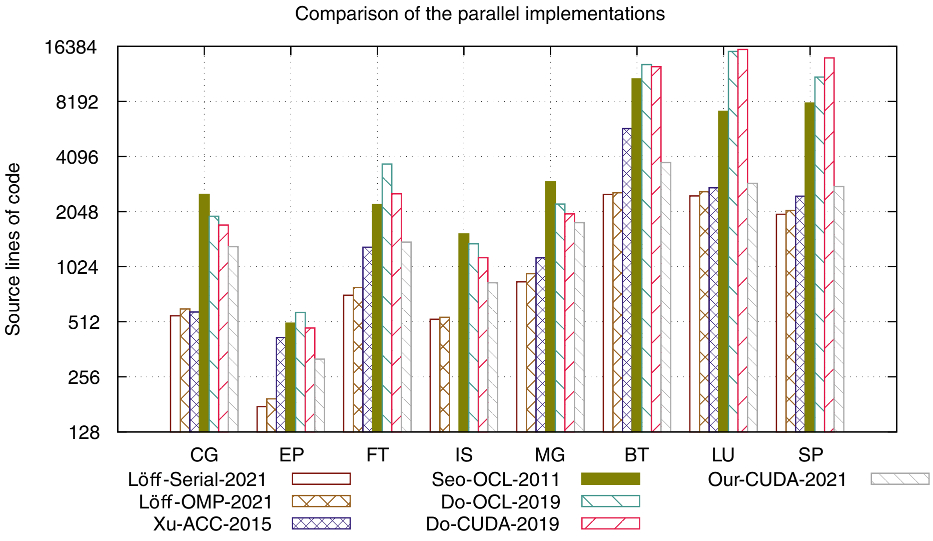


FIGURE 4 Source lines of code of each NPB implementation

Observe that the Löff-Serial-2021 and Löff-OMP-2021 versions have a similar amount of SLOC, which is explained because the OpenMP programming model uses loop-based directives. Although the OpenACC programming model is also directive-based, the Xu-ACC-2015 version presents similar or even bigger numbers of SLOC than our CUDA version (Our-CUDA-2021) in most benchmarks. This shows that even with a directive-based API, a parallel implementation for GPUs still requires non-trivial code refactoring. Moreover, Xu-ACC-2015 implements an optimization to reduce branch divergences that significantly increases the number of SLOC. The optimization consists in offloading to the GPU different versions of a single function, where each version contains a specific path of the original code branches. This reduces the number of branches for the GPU threads and increases the number of SLOC.

The OpenCL programming model is known to be more verbose than CUDA.[16] Thus the OpenCL versions (Seo-OCL-2011 and Do-OCL-2019) are expected to present more SLOC than other versions. Additionally, the Seo-OCL-2011 implementation uses different configurations of thread hierarchy for the GPU and each parallel implementation of the NPB functions for GPU has a version optimized for CPUs. The versions from Do et al.[9] (Do-OCL-2019 and Do-CUDA-2019) implement the same parallelism strategy. These versions are very verbose, presenting the largest number of SLOC for many of the benchmarks. They contain the most complex set of implementations, which turn the code very complex for implementing other parallelism features or strategies. Overall, our CUDA version is less verbose and it uses routines that are less complex than Do-OCL-2019 and Do-CUDA-2019. We did so to ease implementing new features and test different parallelism strategies.

2.6 | Parametrization support for configuring number of threads per block

GPU threads are organized hierarchically. A function offloaded to a GPU is executed by a grid that is composed by blocks. A block is composed by threads and the number of threads per each block modifies the grain of parallelism. In this section, we discusses the steps to make the threads per block configurable for each GPU function in NPB as well as the ease of use via parameters. User specifies the number of GPU threads in a simple text file named gpu.config with our standard configuration file syntax, which has the name of one parameter and its value in each line of the file.

The user can define the number of threads per block for each function of each benchmark with the known names (Figure 1) following this syntax: <benchmark-name> underscore "THREADS_PER_BLOCK_ON" underscore <function-name>. Where <benchmark-name> is the benchmark's two-letter acronym as defined by NPB and <function-name> is the name of the function that is being offloaded to the GPU. Each parameter name is followed by the equal character (=) and an integer value, which is the number of threads per block. For example, the line IS_THREADS_PER_BLOCK_ON_RANK=256 defines the number of threads per block for the function rank of the IS benchmark as 256. By default, if there is no specification or an invalid number, the program uses the GPU's warp size information as the number of threads per block. To support the dynamic number of threads per block, we implemented dynamic shared memory allocation instead of the default NPB that allocate statically the memory for the arrays. If there is not enough shared memory for executing the program with the specified number of threads per block, the configured number of threads is divided by two until it fits the GPU shared memory capacity. The same is done if there are other limitations of resources in the GPU. For instance, if a GPU multiprocessor does not have enough registers to handle at least a block of threads, a memory corruption occurs and it leads to incorrect results in GPU.

3 | EXPERIMENTS

The experiments were conducted in a computer with a processor Intel Xeon E5-2620 (6 cores/12 threads), 16 GB of RAM, and a GPU NVIDIA Titan X Pascal (3584 CUDA Cores) with 12 GB of VRAM. The operating system was Ubuntu 16.04 LTS. The software used was CUDA 10, GCC 9, OpenCL 1.1, OpenACC 2.5, and OpenMP 4.5. We used the compiler flag -O3 to enable automatic compiler optimizations. The implementations provided by the literature[7-9] required the use of the flag -mcmodel=large and the command ulimit -s unlimited due to the stack memory overflow problems. Additionally, we deactivated the data tiling on global memory (implementation 13 presented in the Section 2.5) in the implementations provided by Do et al.[9] because it significantly reduces the GPU performance, as discussed in Section 2.5. We used the workload classes B and C. Although NPB has bigger workloads such as classes D, E, and F, our GPU (NVIDIA Titan X) does not have enough memory to execute them. Each test was executed ten times and averaged, obtaining a negligible standard deviation (SD) that are plotted with error-bars. GPU speedups were computed with respect to the execution time of the sequential code in CPU.

3.1 | Impact of the number of threads per block

In this section, we select the most computational expensive function regarding the execution time of each NPB program to discuss the impact of changing the number of threads per block. These functions were already identified in Sections 2.3 and 2.4. We ran experiments varying the number of threads per block and collected the execution time of each function individually.

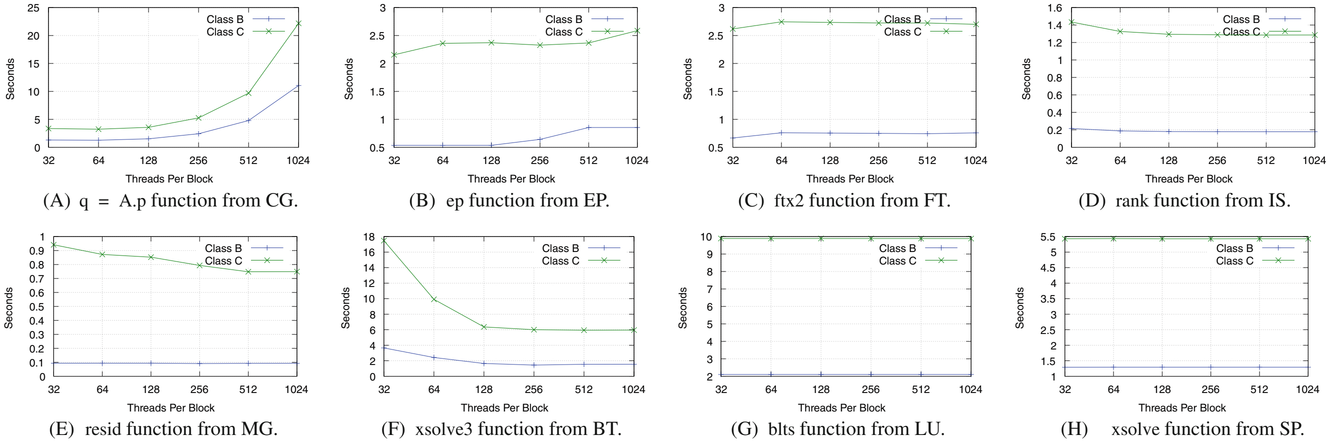
Figure 5 presents the results for each NPB program with different numbers of threads per block. Each plotted graph contains two lines, representing the results for classes B (in blue color with vertical ticks) and C (in green color with × ticks). The x-axis shows different configurations of the number of threads per block, starting from the GPU warp size (32) up to the maximum number of threads per block supported by the GPU (1024). The y-axis presents the execution time in seconds of that specific function for each configuration.

CG's most expensive function (`q = A.p` in Figure 5A) was the most impacted varying the number of threads per block among all the benchmarks. It is best case (using 64 threads per block) is 8.76 times faster than the worst case (1024 threads per block) with class B and 6.84 times faster with class C. This function generates workloads of random sizes where a large number of them are relatively small. Thus, many threads do nothing wasting GPU cycles when a large number of threads per block is used.

BT's most expensive function (`xsolve3` in Figure 5F) was the second most impacted. The best case (256 threads per block for class B and 512 threads per block for class C) is 2.55 times faster than the worst case (32 threads per block) with class B, and 2.94 times faster with class C. This function performs a large number of synchronizations inside each block of threads, so using smaller number of threads per block requires more blocks of threads to be created to conclude the computation. Consequently more synchronizations are executed. In contrast, when using a larger number of threads per block, there are fewer blocks of threads and fewer synchronizations are being executed.

The `ep` function from EP benchmark (Figure 5B) was less impacted by changing the number of threads of each block. The best case (128 threads per block for class B and 32 threads per block for class C) was 1.6 times faster than the worst case for class B (using 512 threads block) and 1.2 times faster with class C (using 1024 threads per block). GPU's Stream Multiprocessors are responsible to execute blocks of threads and they have a limited number of registers. When there is not enough registers, fewer blocks of threads are executed in parallel. Since the `ep` function consumes a large number of registers, when using a large number of threads per block, fewer blocks of threads are executed in parallel.

The `resid` function from MG (Figure 5E) shows a tendency to improve the performance with a larger number of threads per block. The best case (256 threads per block for class B, and 1024 for class C) is 1.02 times faster than the



(A) q = A.p function from CG.

(B) ep function from EP.

(C) ftx2 function from FT.

(D) rank function from IS.

(E) resid function from MG.

(F) xsolve3 function from BT.

(G) blts function from LU.

(H) xsolve function from SP.

FIGURE 5 Performance of the most expensive function of each benchmark offloaded to GPU with different number of threads per block. (A) `q = A.p` function from CG, (B) `ep` function from EP, (C) `ftx2` function from FT, (D) `rank` function from IS, (E) `resid` function from MG, (F) `xsolve3` function from BT, (G) `blts` function from LU, (H) `xsolve` function from SP

worst case with class B (using 64 threads per block) and 1.25 times faster with class C (using 32 threads per block). This function performs synchronizations between the threads of each block, having an effect similar to BT's xsolve3. Yet MG has fewer synchronizations and therefore, is less impacted by the number of threads per block configurations. Similarly, IS's rank function (Figure 5D) performs synchronizations between the threads of a block and also depends on atomic operations. For this function, the best case (512 threads per block for classes B and C) is 1.2 times faster than the worst case (32 threads per block for classes B and C) with class B and 1.11 with class C.

Different from the functions previously discussed in this Section, the fftx2 function from FT (Figure 5C) does not have limitations such as synchronizations (like BT, MG, IS), atomic operations (like IS), small tasks (like CG), or excessive use of registers (like EP). On the other hand, this function has a complex instruction flow composed by loops, nested loops, and conditional statements, causing branch divergences where some threads will do nothing at all. Using a smaller number of threads per block reduces the number of threads in divergent paths, and threads from other blocks can be scheduled to increase GPU occupation. The best case (32 threads per block for classes B and C) is 1.14 times than the worst case (64 threads per block for both classes) with class B and 1.04 with class C.

The blts function from the LU benchmark (Figure 5G) has a simple execution flow, where each thread computes a sequence of instructions without loops, branches, or other complex operations such as synchronizations or atomic operations. The xsolve function from SP benchmark (Figure 5H) is similar to blts. It also has a simple instruction flow, besides the GPU threads still need to perform a few non-nested loops. In both cases, the number of threads per block configuration does not affect the performance.

These experiments highlighted how the number of threads per block configuration can impact on the performance for a function offloaded to the GPU. It also motivates deeper investigations for evaluating the impact on other benchmark functions that are offloaded to the GPU. This is the total impact on performance when varying the number of threads for every function offloaded to the GPU. In Section 3.2, we present different strategies to define and test the number of threads per block combining different functions of the NPB benchmarks.

3.2 | Strategies for choosing the number of threads per block

Our experiments presented in Section 3.1 showed that the number of threads per block may affect the performance of a function offloaded to the GPU. Therefore, it is important to propose and evaluate different strategies for helping to choose a better number of threads per block configuration. The available literature does not agree on the optimal strategy for choosing the number of threads per block.[20,21] The most common strategies involve using the maximum number of threads per block supported by the GPU[22] or a fixed number of threads per block chosen by the programmer.[21,23] While using fixed numbers like the warp size or the maximum number of threads supported by the GPU are useful to identify bottlenecks in a GPU kernel, these numbers may not be the best possible configuration for the thread block size. Another possibility is to evaluate different numbers of threads per block and check the impact of this parameter in the program performance in order to identify a better configuration.[20] Therefore, we classified this configuration space in five strategies that were tested for the GPU functions discussed in Section 3.1. We named and described the strategies as follows:

1. **Warp**. The number of threads per block of each benchmark function offloaded to the GPU will be the GPU warp size.
2. **Max**. The number of threads per block of each benchmark function offloaded to the GPU will be the maximum number of threads per block supported by the GPU device.
3. **Manual**. The number of threads per block will be manually chosen by the expert GPU programmer using their experience and "intuition."
4. **Profiling**. First, a screening is performed in which the same number of threads per block will be assigned to all function of each benchmark so that the number of threads per block will vary from **Warp** until **Max**. Then, the lowest execution time for each benchmark function that was offloaded to the GPU is captured to set the number of threads per block. This strategy's final configuration will have distinct number of threads per block among the functions, the best configuration based on the screening, for a given benchmark.
5. **Exhausting**. Similarly to the Profiling strategy, there is a screening phase varying the number of threads per block. Differently, it test all possible combinations of the number of threads per block for each of the benchmark functions that are offloaded to the GPU instead of assigning the same number to all benchmark functions. Also, the execution time of the whole benchmark is taken into account instead of looking for each benchmark function individually. We capture the configuration that presented the lowest execution time. This screening could take years to finish if all

offloaded functions would be considered. Therefore, we selected only the functions that spend at least 10% of the total execution time of the benchmarks. The numbers of threads per block for the other less significant functions were configured based on the `Profiling` strategy.

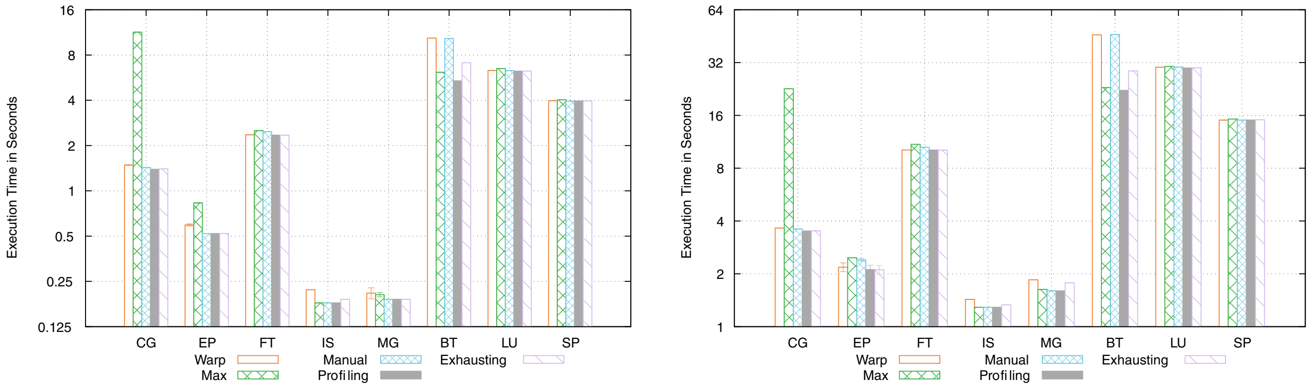
While some strategies agree in some numbers of specific functions, there were no consensus in most of the tests. For example, for the `rank` function of the IS benchmark using class B, `Warp` used 32 threads per block, `Max` used 1024, `Manual` used 256 (this was extracted from[10]), `Profiling` used 512, and `Exhausting` used 128. Our GPU version of NPB contains 94 functions that are offloaded to the GPU, therefore, for the sake of space we do not present all configurations for the benchmarks. Also, even reducing the number of functions evaluated in the `Exhausting` strategy by using the 10% execution time threshold, we still had to test 6156 different configurations of threads per block. The complete set of experiments including the five strategies resulted in 6300 combinations and took a few weeks to finish. This kind of experiments are only possible due to our ease of use of the parametrization support for configuring the number of threads per block provided now in our NPB version. It does not require the user to change the source code or recompile it for testing different numbers of threads per block.

3.3 | Performance of the strategies for threads per block configuration

To evaluate the performance of the five strategies, we performed a set of experiments by executing 10× each benchmark with the configuration found by each strategy. Figure 6 shows the average execution time of both classes B (Figure 6A) and C (Figure 6B) across the whole NPB suite. The benchmarks are identified in the x-axis and the y-axis presents the execution time in seconds (lower is better) with the SD as error-bars. Each column represent a different strategy in the graph.

CG's most expensive function is heavily affected by the number of threads per block and performs better with small amounts of threads, as discussed in Section 3.1, thus the other strategies easily outperform the `Max` strategy. `Profiling` was the best strategy that is 8.16 (class B) and 6.46 (class C) times faster than the worst strategy (`Max`). The reason for this difference in performance is mostly due to the size of the tasks that are generated in the most intensive computations of the benchmark, mainly by the GPU functions `q = A.p` and ||`r`|| `=` ||`x - A.z`|| (together they represent 93% of the total execution time).

In the EP benchmark, the best result for class B was achieved by three different strategies: `Manual`, `Profiling`, and `Exhausting`, which are all 1.6 faster than the worst strategy (`Max`). For class C, the `Profiling` and `Exhausting` strategies presented the best performance, being 1.16 times faster the execution time than the worst strategy (`Max`). However, in this workload (class C) the `Manual` strategy is not among the best ones and its performance is closer to the `Max` strategy. Since EP consumes a large amount of registers to compute the pseudo random numbers and each GPU multiprocessor has a limited number of registers available, it has to execute fewer blocks of threads concurrently. In such case,



(A) Using the workload class B.

(B) Using the workload class C.

FIGURE 6 Total execution time using different strategies to define the number of threads per block. (A) Using the workload class B, (B) using the workload class C

the best configuration found by the strategies Profiling and Exhausting was to create blocks with a lower number of threads. The best number of threads per block vary according to the workload size, so these strategies chose a different number of threads per block for each workload class.

In the FT benchmark the Profiling and Exhausting strategies presented the best results, which represent a performance improvement of 7% for both classes B and C in relation to the worst strategy (Max). The main computations of FT are in the functions that compute the FT in axis x, y, and z (they represent 90% of the execution time). Given that the performance of FT's most expensive function is not very dependent of the number of threads per block, as discussed in Section 3.1, the performance of this benchmark is very similar among all strategies.

For the IS benchmark in both classes B and C the Max, Manual, and Profiling strategies presented the best performance, with is 1.22 (class B) and 1.10 (class C) times faster than the worst case (Warp). Exhausting presented a performance slightly lower than the best cases. The main computation of the benchmark has a few synchronizations per blocks of threads. Using a small number of threads per block, more blocks are created and more synchronizations are executed. This is the reason why Warp presented the worst performance. The MG benchmark presents very similar behavior and for the same reasons that is intra-block synchronizations. In this case, the Manual and Profiling strategies presented the best performance, showing 1.10 (class B) and 1.15 (class C) times faster execution times than the worst case (Warp). It is important to recall that Profiling measures the execution time of each function individually, while Exhausting defines the number of threads per block based on the total execution time.

BT is among the most significantly impacted benchmarks, with a large performance difference when using different strategies. Profiling was the best strategy and it is 1.92 (class B) and 2.07 (class C) times faster than the worst case (Warp). The main computation of the benchmark is the solver (functions xsolve, ysolve, and zsolve) that represents 75% of the total execution time. The solver has a large and variable amount of synchronizations. Using 1024 threads per block, 10 times fewer synchronizations are executed than using 32 threads per block. That is why a very different performance can be observed varying the number of threads, so Warp is the worst strategy. In addition, BT is another case where Exhausting is less efficient than Profiling.

Both LU and SP benchmarks are not significantly impacted with different numbers of threads per block, thus showing less than 5% of performance improvements across all strategies and workloads. However, the best strategy for all cases is Profiling and the worst is Max.

Table 2 presents a summary of results, highlighting the performance improvement of Profiling over the other strategies. We used Profiling as a baseline since it presented the best performance in all benchmarks. The first column lists the benchmark and the workload class. The second column lists which strategy presented the worst performance. The third column lists which strategies presented a performance equivalent to Profiling. The fourth column lists the performance improvement (reduction in execution time) of Profiling over the worst case (second column). The fifth columns lists the performance improvement (reduction in execution time) of Profiling over the Manual strategy.

Compared to Profiling, Exhausting is slower in most cases. Profiling evaluates each function offloaded to the GPU individually, while Exhausting measures only the impact of these functions in the total execution time. In other words, the evaluation from Exhausting is interfered by any procedure beyond the time of the GPU computing. In the NPB programs, as in most of the real-world applications, the CPU performs loops, conditionals, and interacts with the GPU by performing synchronizations and offloading routines. Therefore, we have seen that Exhausting has a worse accuracy at predicting a better number of threads per block for each GPU function.

We also observed that static strategies such as Warp, Max, and Manual can compromise the GPU performance. Warp and Max are opposites and can perform very differently on routines with complex instruction flows. The problem with a static strategy is defining a single configuration of threads per block only once, while the best number of threads per block can vary according to the workload size and the specific characteristics of the GPU function. As an example, we discuss the case of EP with the Manual strategy. When using the class B, Manual has a performance as good as Profiling. However, when using the class C, Manual is 14% slower than Profiling. The differences are due to the fact that Profiling chose a set of configurations for the class B and another set for the class C. The largest improvement of Profiling over the worst case was 717% (CG with class B) and the best improvement over Manual was 108% (BT with class C), which are significant improvements. Our results highlight the importance of taking into account the number of threads per block when evaluating the performance of functions offloaded to GPUs, and how they can provide insights about the software and hardware characteristics.

TABLE 2 Summary of Profiling compared to other strategies

Bench. class	Worst strategy	Same as Profiling	Improve over worst	Improve over Manual
CG.B	Max	-	**717%**	3%
CG.C	Max	-	546%	3%
EP.B	Max	Manual, Exhausting	60%	0%
EP.C	Max	Exhausting	17%	14%
FT.B	Max	-	8%	6%
FT.C	Max	Exhausting	8%	4%
IS.B	Warp	Max, Manual	22%	0%
IS.C	Warp	Max, Manual	11%	0%
MG.B	Warp	Manual, Exhausting	11%	0%
MG.C	Warp	Manual	16%	0%
BT.B	Warp	-	92%	91%
BT.C	Manual	-	108%	**108%**
LU.B	Max	-	4%	1%
LU.C	Max	-	2%	1%
SP.B	Max	Manual, Exhausting	2%	0%
SP.C	Max	Manual	1%	0%

Note: Bold Values are highlighting the better performance.

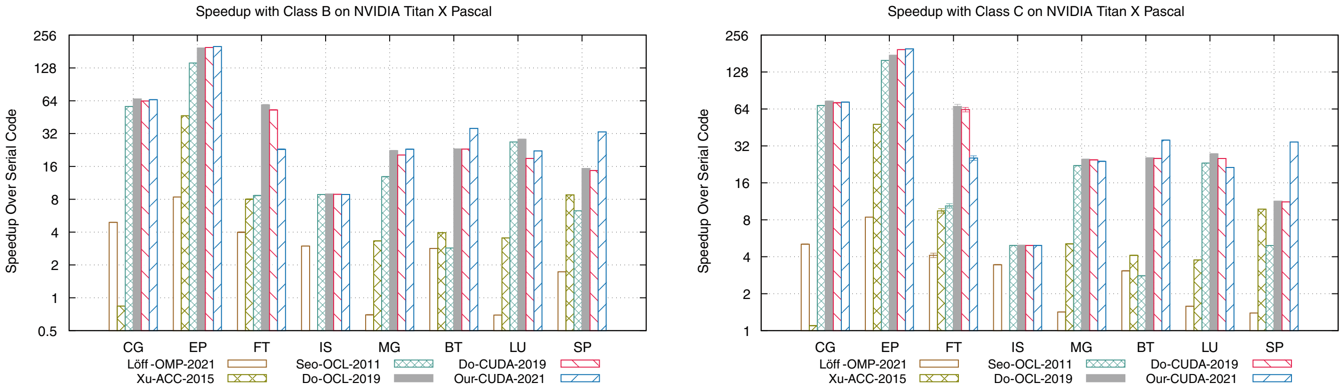
3.4 | Performance comparison with implementations available in the literature

In this section, we evaluate the performance of our NPB implementation with CUDA compared to the other approaches that are available in the literature. For the number of threads per block setup, we used the configurations generated by the strategy Profiling, according to the findings discussed in Section 3.3. Each NPB version evaluated is named as the following: Serial version[6] as Löff-Serial-2021. OpenMP version[6] as Löff-OMP-2021 (this version uses the number of logical cores of the processor as the number of threads, that is 12 in our setup). OpenACC version[7] as Xu-ACC-2015 (the IS benchmark was not made available by the authors, so it is not used in the experiments). OpenCL version by Seo et al.[8] as Seo-OCL-2011. OpenCL version by Do et al.[9] as Do-OCL-2019. CUDA version by Do et al.[9] as Do-CUDA-2019. Our CUDA version as Our-CUDA-2021.

In the experiments, we deactivated the optimization of data tiling on global memory (optimization 13 presented in Section 2.5) for the versions Do-OCL-2019 and Do-CUDA-2019, because it severely degrades the GPU performance (these two versions implement the same parallelism strategy). Figure 7 presents the speedup over the serial code on the CPU for the workloads B (Figure 7A) and C (Figure 7B). Figure 8 presents the GPU memory consumption in MB for the workloads B (Figure 8A) and C (Figure 8A). The x-axis of each figure represent the different benchmarks and the y-axis present the execution time, speedup, and memory consumption, respectively. Table 3 presents the execution time and SDs. The first column lists the NPB versions according to our aforementioned naming convention, the second column specifies the metric (execution time or SD), and the third column onward present the results for each combination of benchmark and workload.

3.4.1 | Execution time and speedup

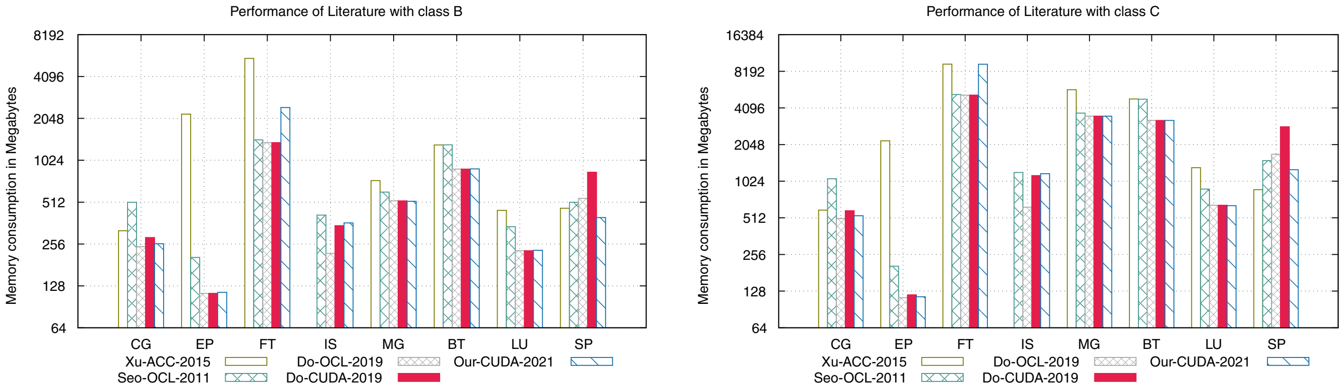
In CG, our implementation achieved speedups of 65.44 (class B) and 72.73 (class C) times with respect to the sequential version on CPU. Xu-ACC-2015 presented the lowest speedups because it does not isolate irregular computations. This version is slower than serial code with class B and only 10% faster with class C. Our approach is 15% (class B) and 6% (class C) faster than Seo-OCL-2011, because we offload concurrent routines to the GPU when functions that compute



(A) Using the workload class B.

(B) Using the workload class C.

FIGURE 7 Speedup of the NPB versions executing on a NVIDIA Titan X GPU over serial code in the CPU. (A) Using the workload class B, (B) using the workload class C



(A) Using the workload class B.

(B) Using the workload class C.

FIGURE 8 GPU memory consumption of the NPB versions executing on a NVIDIA Titan X GPU. (A) Using the workload class B, (B) using the workload class C

different arrays have no dependencies among them (a different function is offloaded to the GPU for each array). This approach increases the GPU usage and also simplifies the execution flow. Our approach is 1% (classes B and C) slower than Do-OCL-2019 and 2% (class B) and 3% (class C) faster than Do-CUDA-2019. Do-OCL-2019 and Do-CUDA-2019 merged the CG functions obtain p.q and obtain z and r into a single routine offloaded to GPU and offloading fewer routines to GPU improved the total execution time of the Do-OCL-2019 version due to the OpenCL runtime.

Our EP implementation was able to achieve 200.99 (class B) and 197.71 (class C) speedup. Our version is up to 330% (class B) and 312% (class C) faster than Xu-ACC-2015, 41% (class B) and 24% (class C) faster than Seo-OCL-2011, 3% and 12% faster than Do-OCL-2019 as well as 2% and 1% faster than Do-CUDA-2019. Xu-ACC-2015 implements a strategy of coarse grain parallelism similar to CPUs, where a subset of the pseudo random numbers is assigned to a GPU thread to be computed. This low degree of parallelism does not maintain a high GPU usage, which is required to achieve good performance results using the GPUs. The other GPU versions exploited finer grain parallelism by assigning each pseudo random number to a GPU thread. However, Seo-OCL-2011 uses a large block of memory to perform the computations that results in memory access conflicts and cache-misses. Moreover, the excessive use of registers prevent GPU multiprocessors to launch more blocks of threads in parallel. Do-OCL-2019 and Do-CUDA-2019 perform a parallel reduce that increases the complexity of the execution flow and requires additional loops, synchronizations, and branches. In contrast, our version exploits fine grain parallelism, uses small blocks of memory, and atomic operations. Atomic operations are expensive, but in EP they help in maintaining the execution flow simpler and thus worth the cost.

TABLE 3 Execution time and standard deviation (in seconds)

Bench. class	Metric	Löff-Serial-2021	Löff-OMP-2021	Xu-ACC-2015	Seo-OCL-2011	Do-OCL-2019	Do-CUDA-2019	Our-CUDA-2021
CG.B	Time	90.89	18.49	108.42	1.60	**1.37**	1.43	1.39
	StDev	0.27	0.15	0.06	0.00	0.01	0.01	0.00
CG.C	Time	254.93	50.33	232.34	3.73	**3.44**	3.55	3.51
	StDev	1.12	0.20	0.13	0.01	0.01	0.01	0.01
EP.B	Time	104.51	12.46	2.24	0.74	0.54	0.53	**0.52**
	StDev	0.04	0.06	0.00	0.01	0.00	0.00	0.00
EP.C	Time	418.16	49.78	8.72	2.63	2.39	2.15	**2.12**
	StDev	0.48	0.22	0.00	0.02	0.08	0.10	0.12
FT.B	Time	53.81	13.49	6.71	6.21	**0.92**	1.02	2.34
	StDev	0.42	1.10	0.01	0.00	0.00	0.02	0.00
FT.C	Time	258.62	62.91	27.35	24.91	**3.86**	4.10	10.15
	StDev	10.72	2.64	0.01	0.03	0.05	0.03	0.00
IS.B	Time	1.59	0.53	-	0.18	0.18	0.18	0.18
	StDev	0.00	0.02	-	0.00	0.00	0.00	0.00
IS.C	Time	6.38	1.85	-	1.29	1.29	1.29	1.29
	StDev	0.03	0.08	-	0.00	0.00	0.00	0.00
MG.B	Time	4.37	6.26	1.32	0.34	0.20	0.21	**0.19**
	StDev	0.02	0.13	0.02	0.00	0.00	0.01	0.00
MG.C	Time	38.27	26.95	7.51	1.73	**1.54**	1.55	1.60
	StDev	0.17	0.15	0.02	0.00	0.00	0.00	0.00
BT.B	Time	192.24	67.79	48.67	67.13	8.30	8.34	**5.39**
	StDev	0.45	0.79	0.02	0.02	0.02	0.03	0.01
BT.C	Time	791.19	257.54	192.67	282.59	30.93	31.28	**22.18**
	StDev	3.12	1.07	0.06	0.02	0.21	0.01	0.10
LU.B	Time	138.67	199.78	39.18	5.16	**4.88**	7.32	6.25
	StDev	0.23	9.79	0.08	0.00	0.02	0.01	0.01
LU.C	Time	636.45	402.81	168.93	27.47	**23.05**	25.18	29.79
	StDev	1.68	9.68	0.16	0.00	0.18	0.01	0.02
SP.B	Time	131.41	75.83	14.98	20.92	8.58	8.95	**3.96**
	StDev	0.21	0.35	0.31	0.00	0.00	0.00	0.00
SP.C	Time	518.47	372.63	53.11	105.01	45.72	46.28	**15.06**
	StDev	0.48	2.42	0.01	0.01	0.02	0.01	0.01

Note: Bold Values are highlighting the better performance.

We reached up to 22.93 (class B) and 25.48 (class C) times of speedup in FT benchmark. This is 186% and 169% faster than `Xu-ACC-2015` and 165% and 145% faster than `Seo-OCL-2011` in classes B and C, respectively. `Xu-ACC-2015` and `Seo-OCL-2011` applied a coarse grain parallelism in the main FT functions (`ftx`, `fty`, and `ftz`) because they have several dependencies. In our approach, we refactored the Fourier computations in smaller functions with simpler instruction flows, where data dependencies are eliminated and it is possible to explore finer grain parallelism for increasing the GPU usage. Additionally, memory coalescing access improved the memory bandwidth. Those are the main reason for the performance difference. However, our version is 61% (class B) and 62% (class C) slower than `Do-OCL-2019`, and 56% (class B) and 60% (class C) slower than `Do-CUDA-2019`. `Do-OCL-2019` and `Do-CUDA-2019` applied a large and complex refactoring in the FT main computations for strictly using the GPU shared memory (implementation 8 in the Section 2.5). Since FT is a memory bound application, the algorithm presented an improved performance using shared memory and lowered significantly the memory latency. In the technique of tiling data with shared memory,[16] the algorithm is refactored in a set of successive steps. Each step loads and copy a portion of global data into the shared memory, computes the chunk using the shared memory, and writes it back to the global memory. This technique increases the complexity of the instruction flow of the GPU threads. However, in the case of FT, where the data is frequently recomputed, reducing the latency of memory with shared memory reached up better results than using a higher degree of parallelism combined with a simpler execution flow.

IS benchmark results are equivalent for all GPU implementation approaches. The `Xu-ACC-2015`version for IS has no source code available and thus we could not test it. IS has a low degree of parallelism and requires several synchronizations between the CPU and the GPU, having a low GPU usage. The speedup of our version is 8.84 (class B) and 4.94 (class C) times. The other approaches present very similar performance numbers since IS is a limited benchmark for GPUs.

The speedups of our MG version are 23.01 (class B) and 23.96 (class C) times. These performance results were possible because we implemented `interp` and `rprj3` functions with a simple instruction flow without branch divergences. In the `psinv` and `resid` functions, the instruction flow is more complex, however, we implemented the memory coalescing access pattern to reduce the memory access latency faced by the GPU threads. `Xu-ACC-2015` also eliminated the branch divergences of the functions `interp` and `rprj3`, but the computations are distributed in several routines offloaded to the GPU. In MG, it results an overhead with lots of routines being offloaded to GPU (each one of them requires a synchronization with the CPU). Additionally, the memory accesses are not coalesced in the functions `psinv` and `resid`. Our version is also faster than `Seo-OCL-2011`: 78% faster with class B and 8% faster with class C. The strategies are similar, but the access patterns are different, which reflects in the performance. Our parallelism strategy is also similar to `Do-OCL-2019` and `Do-CUDA-2019`, consequently, these versions presented a similar performance.

Our CUDA version for BT benchmark achieved 35.69 (class B) and 35.66 (class C) times of speedup. BT is a complex pseudo-application with several dependencies and require refactoring to expose a degree of parallelism suitable to GPUs. `Seo-OCL-2011` keeps the original structure of the benchmark and applies coarse grained parallelism, consequently, this version presented the lowest speedups, 2.86 (class B) and 2.8 (class C) times with respect to the sequential version. `Xu-ACC-2015` applied optimizations such as memory coalescing access pattern, being able to achieve higher speedups of 3.95 (class B) and 4.11 (class C) times. `Do-OCL-2019` and `Do-CUDA-2019` refactored the main algorithm (functions `xsolve`, `ysolve`, and `zsolve`) to expose the parallelism and presented interesting results for executing on GPUs, with speedups from 23.05 to 25.29 times. Our approach is up to 54% (class B) and 55% (class C) faster than `Do-OCL-2019` and `Do-CUDA-2019`. We exploit finer grain parallelism in all GPU functions of this benchmark (which is only possible due our refactoring) and our implementation has a simpler instruction flow than `Do-OCL-2019` and `Do-CUDA-2019`.

LU benchmark uses a data access pattern that does not allow memory coalescing access, because it requires computing the diagonal of matrices. In our approach, the main functions offloaded to the GPU (`blts` and `buts`) have a simple instruction flow, where each GPU thread computes a single element of a diagonal. However, the routine must be offloaded to the GPU several times to compute the whole matrix. This resulted on a overhead, decreasing the overall performance of the benchmark. Our approach was able to reach up to 22.20 and 21.37 times of speedup for classes B and C, respectively.

On the other hand, `Do-OCL-2019` and `Do-CUDA-2019` uses a parallelism strategy to offload to GPU a routine that is executed a single time and compute all the diagonals of the matrix. The routine is executed by a single block of threads, resulting in a low degree of parallelism and the GPU threads perform a complex instruction flow with loops, branches, and synchronizations. Our version is slower than `Do-OCL-2019`, 22% with class B and 23% class C. When compared to `Do-CUDA-2019`, our version is 17% faster with class B and 15% slower with class C. This is because the overhead of offloading several routines to the GPU was worse than the overhead of offloading a single routine with a complex instruction flow and performing several synchronizations between the threads of the block. `Seo-OCL-2011` has a similar strategy to ours concerning `blts` and `buts`, but it has more optimized routines for the `rhs`function, with fewer branches. `Xu-ACC-2015` presented the lowest speedups due to the use of coarse grained parallelism.

The speedups of our SP version are 33.19 and 34.42 times for classes B and C, respectively. SP is a complex benchmark that needs refactoring to expose massive parallelism. In our version we rewrote the main computations to allow a higher degree of parallelism (we increased the total amount of threads from nz to $nz * ny$ with our refactoring) and simplify the instruction flow (we reduced the total amount of branches from $O(ny * nx)$ to $O(nx)$). We also applied different memory access patterns that were also crucial for the performance improvement, because SP computes arrays with up to five dimensions and significantly benefit from memory coalescing access. Our CUDA version is 428% (class B) and 597% faster (class B) than `Seo-OCL-2011` that exploits coarse grained parallelism as well as 278% (class B) and 253% faster (class B) than `Xu-ACC-2015` that implemented memory coalescing access, but used a limited degree of parallelism. `Do-OCL-2019` and `Do-CUDA-2019` also refactored the SP benchmark, but our CUDA version has a higher degree of parallelism and a simpler instruction flow. The performance of our version after refactoring the main computations from SP is similar to `Do-OCL-2019` and `Do-CUDA-2019`. Nonetheless, the speedup was significantly improved after modifying the memory access patterns. Our version is 117% (class B) and 204% (class C) faster than `Do-OCL-2019` as well as 126% (class B) and 207% (class C) faster than `Do-CUDA-2019`.

3.4.2 | Memory consumption

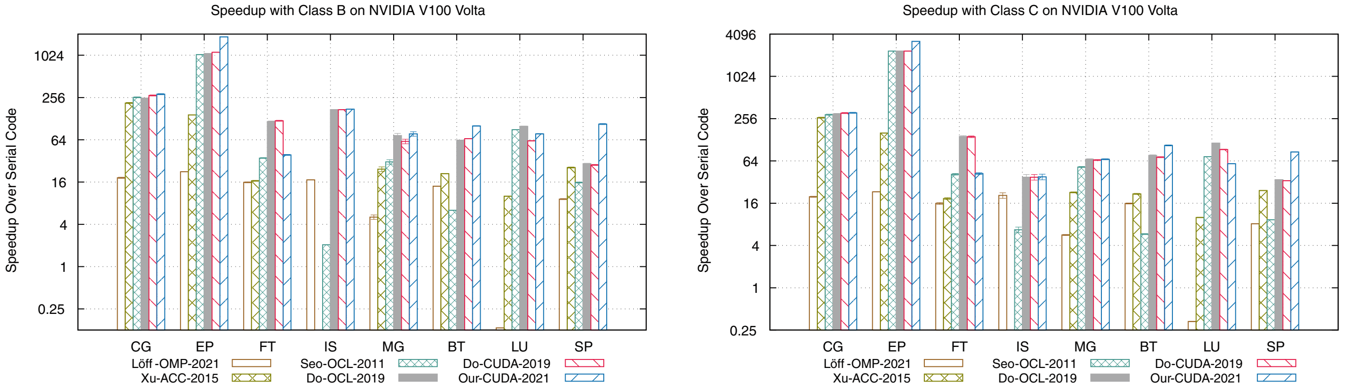
Evaluating memory consumption can provide other insights when approaching a benchmark for GPUs. For example, without refactoring the EP benchmark for data recomputing it is not possible to apply massive parallelism, because the original algorithm requires large amounts of memory for each GPU thread, forcing the creation of a small number of threads. Observing the memory footprint of an algorithm also helps to predict if a given GPU has enough memory to perform a specific workload.

Overall, our CUDA implementation presented an amount of GPU memory consumption similar to the best case from other approaches, as presented in Figure 8. The only exception is for FT, where `Seo-OCL-2011`, `Do-OCL-2019`, and `Do-CUDA-2019` allocate just a small amount of memory and reuse it until the computation is finished. Another interesting case happens in the IS benchmark, where `Do-OCL-2019` performs some routines using the CPU, lowering the requirements for memory usage from the GPU. `Xu-ACC-2015` presented some of the highest requirements of memory overall. In EP, the memory consumption is higher than other approaches because the algorithm was not rewritten to consume fewer memory resources, so this version uses a lower amount of GPU threads to fit the problem in the GPU memory. However, as previously discussed, this negatively affects the performance. In FT, this version allocates the 3D arrays always using the size of the largest dimension. In workloads such as class C, where the arrays have dimensions of different size, not all amount of memory that was allocated is used.

3.4.3 | Additional performance experiments on NVIDIA V100 Volta and NVIDIA T4 Turing

This section shows the performance achieved of NPB for different GPUs. Due to space constraints, we are just presenting a summary of the experiments since they behave similar to the results on Titan X Pascal discussed previously in this article. Figure 9 presents the speedup over the serial code on the CPU for the workloads B (Figure 9A) and C (Figure 9B), when executing the experiments on a machine equipped with a processor E5-2698 v3 (16 cores/32 threads) with 16 GB of RAM, and a GPU NVIDIA V100 Volta (5120 CUDA Cores) with 32 GB of VRAM. Figure 10 presents the speedup over the serial code on the CPU for the workloads B (Figure 10A) and C (Figure 10B), when executing the experiments on a machine equipped with a processor E5-2698 v3 (16 cores/32 threads) with 16 GB of RAM, and a GPU NVIDIA T4 Turing (2560 CUDA Cores) with 32 GB of VRAM.

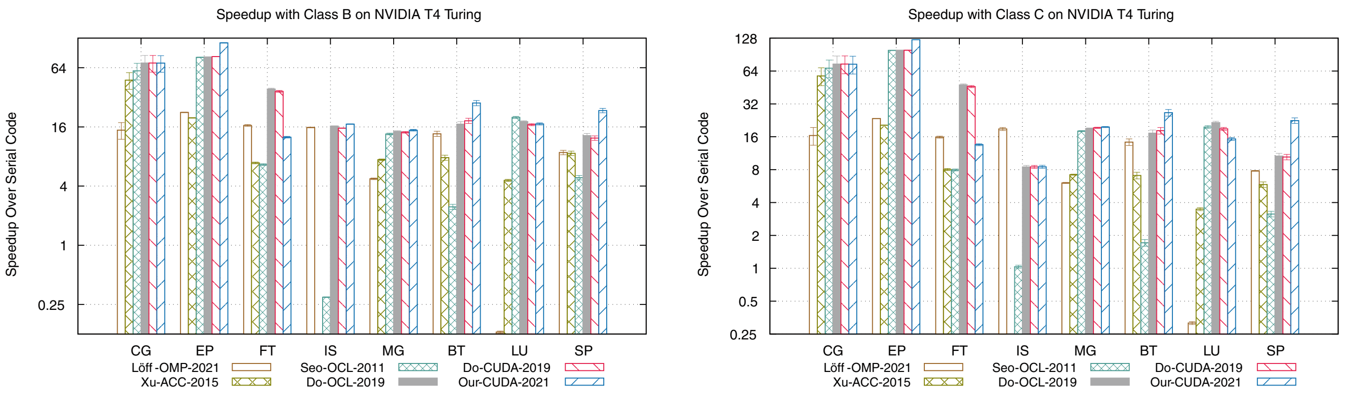
Overall, we observed that the difference of performance among the GPU implementations is similar when they are tested on other GPUs. However, we observed an improvement when executing our version of EP on these newer GPU



(A) Using the workload class B.

(B) Using the workload class C.

FIGURE 9 Speedup of the NPB versions executing on a NVIDIA V100 GPU over serial code in the CPU. (A) Using the workload class B, (B) using the workload class C



(A) Using the workload class B.

(B) Using the workload class C.

FIGURE 10 Speedup of the NPB versions executing on a NVIDIA T4 GPU over serial code in the CPU. (A) Using the workload class B, (B) using the workload class C

architectures. Our EP version is only 2% faster than `Do-CUDA-2019` when executing on the Titan X (Figure 7), nonetheless, our version of EP is up to 37% (Class C) and 73% (Class B) faster than `Do-CUDA-2019` when executing on the V100 (Figure 9) and T4 (Figure 10). This kind of improvement occurs due to automatic optimizations provided by newer NVIDIA GPUs. In this case, we used atomic operations in contrast to the other approaches that implemented reduction algorithms. Another difference noted occurs in the IS benchmark when executing on the T4 GPU (Figure 10). In this benchmark, the OpenMP version is faster than the GPU versions. This occurs because the IS benchmark has limitations for GPU parallelism. The T4 is less powerful than the other GPUs, and the processor used in this test has 16 physical cores, contrasting with 6 cores of the processor used in the test with the Titan X GPU. Finally, the CG version from `Xu-2015-ACC` presented best speedups when using the GPUs V100 and T4. The reason is because we ran those experiments with OpenACC using the PGI 19 instead of GCC-9. PGI was able to generate a more efficient code for GPUs. Unfortunately, the cluster we used had no compatible GCC version for running the OpenACC codes.

3.4.4 | Final remarks

The NPB programs provided interesting cases of study, because each benchmark has a set of features that impacted on the GPU performance in different ways. Table 4 presents a summary of the results and highlights the speedup of our approach over the results of other works (the summary considers the results on the GPUs Titan X, V100, and T4). The first column lists the benchmark and workload class. The second and third column lists the versions that presented the worst and best performances, respectively. The fourth to seventh columns lists the performance differences of our approach compared to `Xu-ACC-2015`,[7] `Seo-OCL-2011`,[8] `Do-OCL-2019`,[9] and `Do-CUDA-2019`.[9]

EP offers the best opportunity for the massive parallelism of the GPUs, given that the whole computation can be performed offloading a single routine to the GPU, without having data dependencies or synchronizations. However, the algorithm requires refactoring to fit the larger workloads in the GPU memory capacity. Unlike EP, CG and other benchmarks require offloading and synchronizing several routines to the GPU, with more complex computations. CG presents a case for unbalanced computations that access different blocks of memory. However, isolating the irregular computation allows one of the best speedups from the benchmarks. FT and MG manipulate 3D data and require different memory access patterns to improve its performance. While FT often recomputes data, MG has irregular computations and the routines must be offloaded several times to the GPUs. These features need different refactoring to favor the hardware characteristics of GPU, including breaking large routines into multiple smaller routines to simplify the instruction flow for the GPU threads and merging multiple small routines into a single routine to avoid excessive synchronizations between CPU and GPU. The pseudo-applications BT and SP work with 5D arrays and require considerable refactoring to expose massive parallelism that fits the GPU programming model. When refactored, these programs are able to achieve similar speedups to the kernels such as FT and MG. The pseudo-application LU exposes complex data dependencies and highlights the overhead of synchronizations for GPU. In contrast to the benchmarks, IS is the most difficult case to obtain

TABLE 4 Summary of our approach compared to the other works available in the literature (the results on the Titan X, V100, and T4 GPUs are considered)

Bench. class	Worst version	Best version	Improve over Xu-ACC-2015	Improve over Seo-OCL-2011	Improve over Do-OCL-2019	Improve over Do-CUDA-2019
CG.B	Xu-ACC-2015	Do-OCL-2019	**7706%**	20%	15%	4%
CG.C	Xu-ACC-2015	Do-OCL-2019	6529%	9%	4%	1%
EP.B	Xu-ACC-2015	Do-CUDA-2019	1194%	79%	73%	66%
EP.C	Xu-ACC-2015	Do-CUDA-2019	1919%	37%	37%	37%
FT.B	Xu-ACC-2015	Do-OCL-2019	186%	165%	−61%	−56%
FT.C	Xu-ACC-2015	Do-OCL-2019	169%	145%	−62%	−60%
IS.B	Equal	Equal	-	0%	0%	0%
IS.C	Equal	Equal	-	0%	0%	0%
MG.B	Xu-ACC-2015	Do-OCL-2019	593%	151%	6%	28%
MG.C	Xu-ACC-2015	Do-OCL-2019	370%	29%	3%	3%
BT.B	Seo-OCL-2011	Do-OCL-2019	804%	1491%	64%	55%
BT.C	Seo-OCL-2011	Do-OCL-2019	769%	**1734**%	54%	47%
LU.B	Xu-ACC-2015	Do-OCL-2019	673%	−13%	−5%	26%
LU.C	Xu-ACC-2015	Do-OCL-2019	483%	−8%	−23%	−15%
SP.B	Seo-OCL-2011	Do-OCL-2019	316%	580%	**267%**	**282%**
SP.C	Seo-OCL-2011	Do-OCL-2019	285%	821%	204%	207%

Note: Bold Values are highlighting the better performance.

high speedups, because it has the smallest potential of parallelism exploitation from the NPB programs influenced by its data dependencies.

As the benchmarks have different characteristics, we followed different strategies to apply our design principles presented in Section 2.1 regarding FT and MG as an example. In FT, the algorithms have a complex instruction flow and imply a performance penalty to GPU threads, with excessive amounts of branches. In this case, it is worthy to apply refactoring in the algorithm and split it into a few CUDA kernels, where the excessive amount of branches per thread is significantly reduced. In contrast, the opposite occurs in MG since it has a few routines with simple instruction flow and no data dependency. In this case, it is worthy merging the computation into a single CUDA kernel and reducing the number of times that we need to launch CUDA kernels, which are also expensive operations. In FT we amortize the number of branches, and in MG we amortize the number of kernel launches.

Porting the NPB programs to GPU requires a large effort and a deep knowledge about the algorithms. Some of the main challenges include: eliminating data dependencies, rewriting the algorithms in order to expose the parallelism, elaborating fine grained parallelism strategies, breaking large problems into smaller and simpler tasks, and providing different access patterns for several routines of a single program to allow memory coalescing access pattern. Those challenges led us to implement different versions of the benchmarks, applying different programming techniques and using different CUDA features. Despite the effort put in the benchmarks, some programming techniques or CUDA features such as CUDA streams and CUDA Dynamic Parallelism did not show any improvement in the routines that we tested, and imposed additional overheads in the benchmarks. Nonetheless, we observed relevant performance improvements compared to the parallel version of the programs using OpenMP. For instance, when running the experiments on the Titan X GPU (Figure 7), our CUDA version was up to 1333% faster than OpenMP in CG, 2296% in EP, 519% in FT, 194% in IS, 3194% in MG, 1157% in BT, 3096% in LU, and 2374% in SP. These results show that is worth porting the benchmarks to GPUs, even though there is a large programming effort involved. Additionally, some NPB programs have limitations when running on CPUs. For example, in the MG benchmark the OpenMP version does not present speedup over the serial code. The reason is because MG has irregular computations, so the CPU can not benefit from cache memory, while the GPU can hide the memory latency through massive parallelism. The previous work of our research group discusses in details the performance and limitations of the parallel version for the NPB programs with OpenMP.[6]

We also provided a comprehensive discussion over interesting cases of study when applying our design principles (presented in Section 2.1) in the NPB programs with CUDA. When compared to the other works where principles such as exploiting a high degree of parallelism, simplifying the instruction flow, and applying memory coalescing access pattern were not used. This is why in several cases we observed a relevant performance difference. The only exceptions are in the benchmarks FT and LU. In FT that often recomputes data, it is worth to increase the complexity of the instruction flow to reduce the memory latency. In LU it is worth to provide lower degrees of parallelism and increase the complexity of the instruction flow to amortize the overhead of offloading several routines to the GPU. As show in the Table 4, our improvements over the best CUDA version of literature (`Do-CUDA-2019`) were up to 4% in CG, 66% in EP, 28% in MG, 55% in BT, 26% in LU, and 282% in SP.

4 | RELATED WORK

As related work we selected papers that manually implemented the NPB standard version, excluding papers focused on other versions (e.g., NPB-MZ or NPB-MPI) or related to automatic code generation.

Gong et al.[21] implemented the EP with CUDA and compared the performance to OpenMP. The authors discussed several aspects about the implementation strategy. The complete NPB was implemented with OpenCL by Seo et al.[8] The OpenCL version was written targeting CPUs and GPUs. A comparative study with CUDA, MATLAB, OpenACC, and OpenCL was done by Malik et al.[24] The authors evaluated manual effort, conceptual programming effort, memory requirements, and performance. A study was carried out implementing the benchmark SP with CUDA and OpenACC by Jin et al.[25] The authors introduced several versions of the implementation where different GPU optimizations were explored. An OpenACC version of the NPB is provided by Xu et al.[7] The authors discussed programmability, performance portability, steps to approach an application to apply parallelism, and listed several GPU optimization strategies. An improvement of the work by Seo et al.[8] was done in Reference 9. The authors presented a set of 15 optimizations for GPUs and applied them to the original OpenCL code. In our previous work,[10] we described the design principles and strategies we followed to implement the five kernels of the NPB with CUDA. We also compared the performance and memory consumption to other works.

Table 5 presents an overview of the related works. The first column shows the reference and the publication year. The second column lists the NPB programs implemented. The third column lists which Application Programming Interface (API) were used to implement the GPU parallel versions. The fourth column identifies if the source code is available to download. The fifth column specifies the programming language used in the implementation. The sixth column identifies if the number of threads per block is configurable through parameters. Finally, the seventh column lists the CPU and GPU that composes the evaluation environment that was used in the experiments.

The works[7,8] provided the first complete GPU implementation of the NPB programs and their source code are used as a starting point for many other works. The works[7,8,21,24,25] are relatively old (for the GPU research pace) and presented several limitations related to the GPU architectures that were available at that time, such as: (a) the low amount of inboard memory that makes prohibitive the evaluation of workloads of relevant size to analyze the performance; (b) the low amount of cores, that makes difficult evaluating the suitability of parallel strategies. These characteristics limited the GPUs to small performance improvements, with the program sometimes running even slower than the code running on the CPU.[7,8]

The works[9,10] are the most recent approaches of the literature and presented the most effective programming techniques and the highest speedups. We extend[10] by providing a CUDA version for the pseudo-applications from NPB, a new version of the NPB kernels implemented by applying a novel set of design principles, and we further discuss its implications. In addition, we make a broader study with emphasis on strategies for choosing different numbers of threads per block and their impact on performance. The works[21,24] were the only ones to explore the performance of the NPB programs related to the number of threads per block. The performance of the EP benchmark is observed when varying the number of threads per block in the work by Gong et al.[21] However, besides evaluating a single benchmark, the authors only ran experiments with 64, 128, and 256 threads per block. A strategy for choosing the number of threads per block is described by Malik et al.[24] However, the strategy was not efficient in some cases, where CUDA performance was worse than MATLAB that automatically chooses the number of threads per block for builtin functions that are offloaded to GPUs. The main problem of their strategy is that it assigns the same number of threads per block to all functions that are offloaded to the GPU, while we observe that the functions have different characteristics, thus each one of them perform better with a different number of threads per block. The worst case was in the CG kernel, where the author performed the

TABLE 5 Overview of the related works

Year	Benchmarks	GPU API	Source avail.	Lang.	Thread Param.	Hardware
2010[21]	EP	CUDA	No	C	No	(1) Intel Q6600, NVIDIA GT200
2011[8]	BT, CG, EP, FT, LU, IS, MG, SP	OpenCL	Yes	C	No	(1) 2 × Intel X5660, NVIDIA GTX 480
2012[24]	CG, EP, FT, MG	CUDA, OpenCL, OpenACC, MATLAB	No	C	No	(1) Intel X5560, NVIDIA C2050
2012[25]	SP	OpenACC	No	Fortran	No	(1) Intel X5670, NVIDIA M2090, (2) AMD Opteron 2354, NVIDIA GTX 480
2015[7]	BT, CG, EP, FT, IS, LU, MG, SP	OpenACC	Yes	C	No	(1) Intel Xeon (unspecified version), NVIDIA Kepler K20
2019[9]	BT, CG, EP, FT, IS, LU, MG, SP	CUDA, OpenCL	Yes	C	No	(1) Intel Xeon Gold 6130, NVIDIA V100, (2) Intel Xeon Gold 6130, AMD Radeon VII
2020[10]	CG, EP, FT, IS, MG	CUDA	Yes	C++	No	(1) Intel Xeon E5-2620, NVIDIA Titan X Pascal
Our work 2021	BT, CG, EP, FT, IS, LU, MG, SP	CUDA	Yes	C++	Yes	(1) Intel Xeon E5-2620, NVIDIA Titan X Pascal, (2) Intel Xeon E5-2698 v3, NVIDIA V100 Volta, (3) Intel Xeon E5-2698 v3, NVIDIA T4 Turing

benchmark using 256 threads per block and the main computations of this benchmark suffer an overhead when executing with a large number of threads per block.

5 | CONCLUSION

In this article we presented a new CUDA implementation of the NPB kernels and pseudo-applications applying a specific set of best programming practices for GPUs, which was referred during the paper as being design principles. We presented a study for evaluating how the number of threads per block can impact the GPU performance, where the results showed that the computations offloaded to GPU can be highly impacted when choosing different strategies to pick up the number of threads per block. We also compared our CUDA version to the parallel versions available in the literature, analyzing performance, memory consumption, programming effort, and design choices. The comparative study with the literature highlighted how different design choices can affect the GPU behavior in the NPB benchmarks. We obtained a significant amount of speedup improvement compared to the different approaches. In addition to that, our investigations and analysis contributed to the discussion about the challenges of porting CFD applications to GPUs, and how the computations from this domain can impact the performance of GPUs.

This study can be extended in several ways, thus we suggest some investigations that can be followed in the future. Design principles for GPUs can be further investigated, reasoning about exceptional cases where specific optimizations are more suitable to boost the GPU performance. Studies may suggest a newer set of optimizations to be applied in the CUDA version of NPB, such as the possibility of executing workloads larger than the memory capacity of GPU without compromising the application performance. Our design principles can be applied in NPB using other GPU frameworks such as Thrust from NVIDIA or OpenCL. Further studies may exploit multi-GPU parallelism since our work was limited by a single GPU board. Different GPU parameters can also be investigated and implemented in the NPB, such as the amount of shared memory or different memory access patterns.

ACKNOWLEDGMENTS

This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 05/2019-PQG project PARAS (No 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPAR4.0 (No 21/2551-0000725-7), and Universal MCTIC/CNPq No 28/2018 project SPARCLOUD (No 437693/2018-0). We thank NVIDIA's GPU Grant program for the GPU donation, and also for the access to the NVIDIA Solutions Lab.

AUTHOR CONTRIBUTIONS

Gabriell Araujo: Software; investigation; validation; conceptualization; formal analysis; visualization; writing – original draft. **Dalvan Griebler**: Conceptualization; formal analysis; project administration; validation; writing – review and editing. **Dinei A. Rockenbach**: Writing – review and editing. **Marco Danelutto**: Writing – review and editing. **Luiz G. Fernandes**: Supervision; funding acquisition.

DATA AVAILABILITY STATEMENT

The source codes are open source available in a GitHub repository: https://github.com/GMAP/NPB-GPU

ORCID

Gabriell Araujo https://orcid.org/0000-0001-8179-2318
Dalvan Griebler https://orcid.org/0000-0002-4690-3964
Dinei A. Rockenbach https://orcid.org/0000-0002-2091-9626
Marco Danelutto https://orcid.org/0000-0002-7433-376X
Luiz G. Fernandes https://orcid.org/0000-0002-7506-3685

REFERENCES

1. Bailey DH, Barszcz E, Barton JT, et al. The NAS parallel benchmarks. Technical report RNR-94-007, NASA Advanced Supercomputing Division; California; 1994.
2. Jin H, Frumkin M, Yan J. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report NAS-99-011, NASA Advanced Supercomputing Division; California; 1999.
3. Bailey D, Harris T, Saphir W, Wijngaart RVD, Woo A, Yarrow M. The NAS parallel benchmarks 2.0. Technical report NAS-95-020, NASA Advanced Supercomputing Division; California; 1995.
4. Wijngaart RVD, Jin H. The NAS parallel benchmarks, multi-zone versions. Technical report NAS-03-010, NASA Advanced Supercomputing Division; California; 2003.
5. Wijngaart RVD, Frumkin M. NAS grid benchmarks version 1.0. Technical report NAS-02-005, NASA Advanced Supercomputing Division; California; 2002.
6. Löff J, Griebler D, Mencagli G, et al. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Gener Comput Syst*. 2021;125:743-757. doi:10.1016/j.future.2021.07.021
7. Xu R, Tian X, Chandrasekaran S, Yan Y, Chapman B. NAS parallel benchmarks for GPGPUs using a directive-based programming model. In: Brodman J, Tu P, eds. *Languages and Compilers for Parallel Computing*. Springer International Publishing; 2014:67-81.
8. Seo S, Jo G, Lee J. Performance characterization of the NAS parallel benchmarks in OpenCL. Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC); 2011:137-148; IEEE.
9. Do Y, Kim H, Oh P, Park D, Lee J. SNU-NPB 2019: parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs; 2019:93-105.
10. Araujo GA, Griebler D, Danelutto M, Fernandes LG. Efficient NAS parallel benchmark kernels with CUDA. *PDP'20*. IEEE; 2020:9-16.
11. NASA Advanced Supercomputing Division. NAS parallel benchmarks; 2021.
12. Tian X, Xu R, Yan Y, Chandrasekaran S, Eachempati D, Chapman B. Compiler transformation of nested loops for general purpose GPUs. *Concurr Comput Pract Exper*. 2016;28(2):537-556.
13. NVIDIA. CUDA C++ best practices guide. Technical report, NVIDIA Corporation; California; 2021.
14. Mattson TG, Sanders BA, Massingill BL. *Patterns for Parallel Programming*. Software Patterns Series. 1st ed. Pearson Education; 2004.
15. McCool M, Robison AD, Reinders J. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st ed. Morgan Kaufmann; 2012.
16. Kirk DB, Hwu WW. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st ed. Morgan Kaufmann Publishers Inc; 2010.
17. Cheng J, Grossman M, McKercher T. *Professional CUDA C Programming*. 1st ed. Wrox Press Ltd; 2014.
18. NVIDIA. *CUDA C programming guide. Technical report*. NVIDIA Corporation; 2019.
19. Wheeler DA. SLOCCount; 2016.
20. Vollmer M, Svensson BJ, Holk E, Newton RR. Meta-programming and auto-tuning in the search for high performance GPU code; 2015:1-11; ACM.

21. Gong C, Liu J, Qin J, Hu Q, Gong Z. Efficient embarrassingly parallel on graphics processor unit. Proceedings of the 2010 2nd International Conference on Education Technology and Computer; Vol. 4, 2010:V4-400-V4-404; IEEE.
22. Sundin P. *Adaptation of Algorithms for Underwater Sonar Data Processing to GPU-Based Systems*. Master's thesis. IDA, Department of Computer and Information Science, Linköping; 2013:581 83.
23. Woolley C. GPU optimization fundamentals; 2013.
24. Malik M, Li T, Sharif U, Shahid R, El-Ghazawi T, Newby G. Productivity of GPUs under different programming paradigms. *Concurr Comput Pract Exper*. 2012;24(2):179-191. doi:10.1002/cpe.1860
25. Jin H, Kellogg M, Mehrotra P. Using compiler directives for accelerating CFD applications on GPUs. In: Chapman BM, Massaioli F, Müller MS, Rorro M, eds. *OpenMP in a Heterogeneous World*. Springer; 2012:154-168.

How to cite this article: Araujo G, Griebler D, Rockenbach DA, Danelutto M, Fernandes LG. NAS Parallel Benchmarks with CUDA and beyond. *Softw Pract Exper*. 2023;53(1):53-80. doi: 10.1002/spe.3056