Efficient NAS Parallel Benchmark Kernels with CUDA

Gabriell Alves de Araujo*, Dalvan Griebler*†, Marco Danelutto‡, and Luiz Gustavo Fernandes*
*School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
†Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil.
‡Computer Science Department, University of Pisa (UNIPI), Pisa, Italy.
Corresponding authors: {gabriell.araujo, dalvan.griebler}@edu.pucrs.br

***Abstract*—NAS Parallel Benchmarks (NPB) are one of the standard benchmark suites used to evaluate parallel hardware and software. There are many research efforts trying to provide different parallel versions apart from the original OpenMP and MPI. Concerning GPU accelerators, there are only the OpenCL and OpenACC available as consolidated versions. Our goal is to provide an efficient parallel implementation of the five NPB kernels with CUDA. Our contribution covers different aspects. First, best parallel programming practices were followed to implement NPB kernels using CUDA. Second, the support of larger workloads (class B and C) allow to stress and investigate the memory of robust GPUs. Third, we show that it is possible to make NPB efficient and suitable for GPUs although the benchmarks were designed for CPUs in the past. We succeed in achieving double performance with respect to the state-of-the-art in some cases as well as implementing efficient memory usage. Fourth, we discuss new experiments comparing performance and memory usage against OpenACC and OpenCL state-of-the-art versions using a relative new GPU architecture. The experimental results also revealed that our version is the best one for all the NPB kernels compared to OpenACC and OpenCL. The greatest differences were observed for the FT and EP kernels.**

I. Introduction

The utilization of graphic processing units (GPUs) has significantly increased over the years. GPUs are nowadays equipped by thousands of computing unites (cores), offering high-performance, high energy-efficiency, and low cost. The use of GPUs is more and more fundamental and necessary for accelerating enterprise and scientific software, solving complex and large problems [1]. This amount of parallelism available imposes parallel programming and hardware design challenges. To mitigate this, benchmarks become the standard way to help in research and development for improving hardware and software performance. They are used to evaluate computing architectures, programming techniques, tools, and other solutions.

The NASA Advanced Supercomputing Division has developed the NAS Parallel Benchmarks (NPB) [2]. It has become an important and standard benchmark set that has been used in many types of research over the years. NPB consists of five kernels and three pseudo-applications based on computational fluid dynamics (CFD). The NASA expertise and the knowledge along with the documentation provided by scientific reports, which includes math and algorithmic definitions, assigned major relevancy to NPB. Also, NPB gained great popularity due to the extensive use of different evaluations and tests. The benchmarks offer algorithmic kernels close to real applications and also provide several workload options, from small to very large sizes. The NPB was originally developed with Fortran language and is available in several versions such as sequential code [2], OpenMP [3], MPI [4]–[6], High Performance Fortran (HPF) [7] and Multi-Zone (NPB-MZ) [8].

Concerning the GPU context, NPB is receiving attention for evaluating the performance behavior, compare different architectures, study specific programming techniques, and evaluate the efficiency of compilers' code generation [9]–[15]. When analyzing the literature, we realize that the main works are relatively old, while programming techniques and hardware architecture have evolved. For instance, GPUs got from tens to thousands of cores, have new features, and more efficient mechanisms. A problem with the available GPU parallel implementations is that there is no test with large workloads such as NPB class B and C due to the limitations that this kind of architectures had in the past. Also, a low expectation of acceleration for GPUs was given, even if that small workloads do not allow to evaluate the GPU performance precisely in current architectures. Therefore, it is unclear the real performance that a GPU can obtain in these applications. It is also hard to analyze the performance gap between high-level approaches and robust low-level implementations. To the best of our knowledge, there is no complete parallel implementation of the NPB kernels provided with CUDA. The only implementations available use OpenCL [10] and OpenACC [12]. There are also works reporting the lack of CUDA version for the NPB benchmarks as a limiting point for their research [12], [16].

To fill these gaps, our goal is to provide new NPB parallel versions with CUDA, starting from the five kernels. We also intend to compare the performance and discuss the implementations with respect to the state-of-the-art OpenCL and OpenACC versions using larger workloads (classes B and C). The scientific contributions are summarized as follows:

- **it is the first work to provide NPB's five kernels with CUDA**. It is an important progress to research and development since CUDA is the standard parallel programming framework for Nvidia's GPUs. Research

2377-5750/20/$31.00 ©2020 IEEE
DOI 10.1109/PDP50117.2020.00009

on hardware and software can now be expanded due to the availability and possibility to compare the existing frameworks, programming techniques, code generation techniques, and parallel algorithms for GPUs.

- **it is the first work to support and compare classes B and C for the NPB's five kernels**. Previous works were not able to run experiments with these classes. Consequently, there was a limitation for analyzing the current GPU architecture's behavior under certain computations as well as the scalability of the parallel implementations provided.
- **we revealed new facts about NPB**. Previous works reported a low expectation of acceleration for the NPB's kernels with GPUs. However, the performances we obtained are significantly better w.r.t. the previous ones. We have shown that performing a careful analysis, code refactoring, and smart parallel implementation, it is possible to make the NPB kernels suitable for GPUs . We achieved more than double of the state-of-art speed-up and made the memory usage more efficient.
- **we run new experiments covering performance and memory usage as well as comparing with the OpenACC and OpenCL state-of-the-art versions**. We used a relative new GPU architecture to measure the memory usage, and to discuss the parallel implementation techniques and algorithms, comparing the state-of-the-art GPU parallel programming frameworks (CUDA, OpenACC, and OpenCL) for the five NPB kernels.

The paper is organized as follow. Section II describes the related work. Section III describes the CUDA implementation of the NPB five kernels. Section IV presents the experiments and results of this work. Section V presents the conclusion.

II. Related Work

Here, we selected papers that studied NPB for GPUs and discard approaches where the focus was to provide tools for automatic code generation or tackled Multi-GPU for NPB.

The design and implementation of the EP NPB kernel with CUDA are presented and compared with others works by Gong et. al. [9]. Their goal was to share the programming experience to register possible parallelism strategies applied to EP that improved the performance, and consequently, may be used on other applications to gain performance. It was essentially one of the first works about NPB for GPUs.

The NPB kernels and pseudo-applications were implemented with OpenCL in [10]. The work was the first complete implementation of the NPB targeting accelerators. Differently from [9], they did not made a comparison with other works. The work compared performance differences between multi-core and many-core architectures. The optimization design principles adopted in the implementations were also described. The main contribution was the characterization of the OpenCL performance and the availability of the source codes for the parallel versions.

The work [11] implemented the CG, EP, FT and MG kernels with CUDA, OpenCL, OpenACC and MATLAB. The paper made a comparative study between these frameworks, analyzing the programming effort and programmer productivity. Differently from the other works [9] and [10], no details about the implementations were given. The focus was to provide a discussion about the evaluated frameworks. Additionally, they evaluated how much impact have high-level abstractions like OpenACC or MATLAB on the performance compared to CUDA and OpenCL.

The SP pseudo-application was studied with OpenACC in [14]. The work shared the development experience of the authors, analyzed the usability, programming effort, and performance of OpenACC compared to CUDA. The authors implemented several versions of the SP, applying a closer strategy to OpenACC and CUDA. As of this, the work is similar to [11]. It is the only paper published by NASA about NPB with GPUs.

BT-MZ, LU-MZ and SP-MZ from the NPB Multi-Zone (NPB-MZ) [8] were implemented with CUDA in [15]. The focus was on hybrid programming techniques. This is the single work that ported the complete NPB-MZ to GPUs. They presented the deepest hybrid parallel programming approach of the NPB GPU literature, however, researchers commonly utilize the NPB standard instead of NPB-MZ.

The NPB kernels and pseudo-applications were implemented with OpenACC by et. al Xu [12]. This was the second complete implementation of the NPB benchmarks targeting GPU accelerators. The authors shared their OpenACC experience and described several techniques that improved the GPU performance using OpenACC. Also, the authors compared their OpenACC implementation to the OpenCL work [10].

SP-MZ from the NPB Multi-Zone (NPB-MZ) was studied with OpenACC by Stone et. al. [13], using different programming paradigms, single GPU, hybrid CPU and GPU, and Multi-GPU. The authors compared the performance of the approaches. The work is similar to [9] and [14] in the goal of presenting a deeper study for a single application.

Table I presents an overview on the related work characteristics and features. First column presents the work followed by the publication year. Second column lists the benchmarks investigated. Third column lists the GPU Frameworks to provide the parallel versions. Fourth column indicates if the author made the source code of the work available. Fifth column shows the programming language used. Sixth column describes the hardware used in the experiments.

Based on our survey, the main research works are [10] and [12], mostly because they implemented the complete NPB for GPU or with OpenACC or OpenCL and their source code is available. Therefore, other scientists can use or compare their parallel implementations in the research works. Since there is no source code available of [9], [11], [13], [14], they are less used in other researches. Although the work [15] has made the source code available, NPB-MZ is less used in the literature and its focus is on hybrid parallel programming.

All related works cited have common characteristics, they are relatively old. The works have also not tested bigger workloads (e.g., class B or C) for NPB due to the GPU limi-

TABLE I
RELATED WORK GENERAL INFORMATION

Work	Apps	Tools	Available	Lang.	Hardware
[9] 2010	EP	CUDA	No	C	**(1)** Intel Q6600, Nvidia Tesla GT200
[10] 2011	BT, CG, EP, FT, LU, IS, MG, SP	OpenCL	Yes	C	**(1)** two Intel X5660, Nvidia GTX 480
[11] 2012	CG, EP, FT, MG	CUDA, OpenCL, OpenACC, MATLAB	No	C	**(1)** Intel X5560, NVIDIA Tesla C2050
[14] 2012	SP	OpenACC	No	Fortran	**(1)** Intel X5670, Nvidia Tesla M2090, **(2)** AMD Opteron 2354, Nvidia GTX 480
[15] 2013	BT-MZ, LU-MZ, SP-MZ	CUDA	Yes	C++	**(1)** two Intel X5670, Nvidia GTX 570
[12] 2015	BT, CG, EP, FT, IS, LU, MG, SP	OpenACC	Yes	C	**(1)** Intel Xeon (version unspecified), Nvidia Kepler K20
[13] 2015	SP-MZ	OpenACC	No	C	**(1)** Intel E5-2670, Nvidia Kepler K40c, **(2)** Intel E5-2670 v2, Nvidia Tesla K40s
our work 2020	CG, EP, FT, IS, MG	CUDA	Yes	C++	**(1)** Intel E5-2620, Nvidia Titan X Pascal

tations at the time that those papers were written. In addition, the conclusions were relatively basic as GPU programming techniques were not mature and most of the implementation choices were related to architecture characteristics instead of fine-grained and massive thread parallelism.

This work has common characteristics with respect to all related works cited. Our goal is to provide for the community NPB for GPU like [10] that implemented the NPB with OpenCL and [12] that implemented the NPB with OpenACC. Differently, we presented the first NPB kernels parallel implementation with CUDA. We implemented different techniques and evaluate the performance of our approach like [9], [14], [15]. We compare performance of our work with other works like [9]. Also, our experiment are broader since we compare the performance of CUDA, OpenCL and OpenACC using different workloads (class B and C).

III. PARALLEL IMPLEMENTATIONS

To implement our NPB kernels with CUDA, we used a well tested conversion from the original NPB 3.3.1 to C++ [17]. This section first describes the design principles used to implement the NPB kernels and then discusses in detail how each one of the kernels has been implemented. Our source codes are available in a GitHub repository[1].

A. Design Principles

The design principles we used in this work to implement NPB with CUDA can be described as follows:

[1]NPB with CUDA is available here https://github.com/GMAP/NPB-GPU

1) **Implement the GPU kernels as simple as possible to avoid large operation cycles in the GPU**. The idea is to allow more lightweight threads execute in parallel. For instance, if a sequential routine has two array computations that can compute independently, we can create two GPU kernels. In principle, this approach can improve performance because threads can do less or no jumps among memory blocks.
2) **Avoid branch divergences as much as possible**. Branches inside GPU kernels prevent the threads parallel execution. The usual strategy is to collapse the loops so that each thread computes one iteration. However, sequential source codes may naturally have branches for optimizing the performance of the sequential algorithm by using conditions and jumps. There are also situations where nested loops can not be simply collapsed due to the data dependencies. We used different strategies that are explained later to avoid and minimize the impact of the branches inside GPU kernels and therefore to increase the performance of kernel parallel executions.
3) **Support memory coalescing.** Accessing the global GPU memory adds a significant latency. Therefore, it is important to perform contiguous data access in parallel threads to save load instructions and therefore to increase performance [1].
4) **Avoid as much as possible global memory accesses**. The GPU memory is hierarchical organized. The Shared memory is on the chip offering high bandwidth. Constant memory, texture, and surface memory are the cache memory offering low latency. Then, there is the local memory where data coalescing is automatically done by the GPU to speed up the memory access. Therefore, each code has unique characteristics regarding memory access. For example, if there is a recurrent access to specific data, data can be copied to the shared memory. In case data is read but never written, we can copy it to the constant memory. We discuss the specific memory improvements made so far to each NPB kernel in the next sections.
5) **Implement GPU parallelism only when it is worth**. Since to compute on GPU we need to pay for the data transferring overhead, a careful analysis of the code should be done about the potential parallelism that the application can obtain as well as the cost of the data transfers and GPU kernel computations. As a consequence, only if it adds performance improvements, the parallel implementation to GPU should be performed.

B. Conjugate Gradient (CG)

CG computes an approximation to the smallest eigenvalue of a large matrix, and tests irregular communication and unstructured matrix vector multiplication [2]. The CG execution flow is organized into a global loop, where inside `conj_grad` function, find norm of `z` operation, and normalize `z` to obtain `x` are executed for each iteration. Overall, achieving a reasonable parallel performance is a relatively simple task due to the

changes required compared to other NPB kernels. Almost all parallel regions consist of nested `for` loops that can be collapsed and mapped into GPU threads independently. Several of these loops can perform parallel reduce operations.

Nonetheless, the two main computations of CG inside the `conj_grad` function cannot achieve a good speedup performing a simple implementation as we did for all other operations. There is a sub-matrix multiply of irregular computations demonstrated in the code snippet of Listing 1. In the sub-matrix multiply, there are different workloads sizes that begins on `rowstr[j]` and goes until `rowstr[j+1]` for the inner loop. For each iteration `j`, different blocks of data are accessed in the global memory and the size of these blocks vary because `rowstr` has aleatory numbers. If we simply apply the map pattern (assign each thread to a `j` iteration), the workload will be unbalanced and the memory access will be irregular. If we simple collapse both loops, the computation grain is smaller, but the threads execution is still irregular. In this case, GPU threads from a block may compute different groups of computations. Although each computation group has a regular data access inside, it may happen that threads from the same block execute different computation groups. Finally, decompose this part of code in a streaming fashion where each kernel executes a `j` computation group is not a good performance strategy, because it will launch GPU kernels too frequently. We tested and evaluated all these strategies to highlight the drawbacks.

```
1 for(j = 0; j < lastrow - firstrow + 1; j++){
2   sum = 0.0;
3   for(k = rowstr[j]; k < rowstr[j+1]; k++){
4     sum = sum + a[k]*p[colidx[k]];}
5   q[j] = sum;}
```

Listing 1. CG's irregular computations.

```
1  int j = blockIdx.x;
2  int local_id = threadIdx.x;
3  int begin = rowstr[j];
4  int end = rowstr[j+1];
5  double sum = 0.0;
6  for(int k=begin+local_id; k<end; k+=blockDim.x){
7    sum = sum + a[k]*p[colidx[k]];}
8  share_data[local_id] = sum;
9  /* reduce sum on this block */
10 if(local_id==0){q[j]=share_data[0];}
```

Listing 2. Strategy for CG's irregular computations on GPU

To overcome these problems, our strategy was to create a group of threads for each iteration `j` as presented in Listing 2. Since we are not able to know in advance the number of iterations of the inner loop, we defined a default number of threads per block, which is defined according to the maximum number of threads that the GPU warp scheduler is able to handle at once. If the inner loop has more iterations than the number of threads in the block, line 6 implements the loop where threads will execute again the computation. Observe that the memory access pattern is vertical to increase the performance. We made so that the global data `rowstr` is accessed only two times to read the `start` and `end` of the computation group. All other accesses are made through local variables. The results of each thread are then stored in the shared memory to apply the parallel reduce pattern with interleaved addressing [1].

C. Embarrassingly Parallel (EP)

The EP kernel estimates the floating point capacity. It was designed to be an embarrassingly parallel computation with almost no communication/synchronization [2]. The EP execution flow is organized into a loop, where each iteration executes two distinct computations. Firstly, it computes pseudo-random numbers from the given iteration as the initial value. Secondly, it computes Gaussian deviates from the pseudo random numbers generated. The parallel implementation of EP could be performed executing each iteration in parallel. However, this approach requires special attention regarding the memory consumption on GPUs. Each iteration needs a very large array `x` to store the pseudo random numbers, and each thread has its own copy of the array `x`. A possible strategy to approach this problem is assign a group of iterations to each thread. Consequently, a thread will reuse the array `x`. This coarse grained approach does not scales very well on GPUs, because the number of threads will be limited. Also, the number of serial computations in the GPU kernel will be too big, as for each iteration a very large array of pseudo-random numbers is computed.

To overcome memory and scalability limitations, the strategy was to create a small array `x` for each thread so that each one of them recomputes the array `x` until all the pseudo random-numbers are generated. Therefore, a larger number of threads is created and each iteration is assigned to a block of threads instead of a single thread. The array `x` is also in the GPU shared memory. We also applied a vertical access pattern for coalescing the memory access.

D. Fast Fourier Transform (FT)

FT solves differential equations using forward and inverse Fast Fourier Transforms [2]. The FT execution flow is organized in a loop where each iteration executes five functions, *evolve*, *fft-z*, *fft-y*, *fft-x* and *checksum*. The *evolve* computes the initial values of the arrays for each iteration, and consists of three nested loops that can be simple collapsed into a GPU kernel for executing multiple threads in parallel. Then, a function called *checksum* combines the arrays values at the end of each iteration. *checksum* also have the potential to implement the parallel reduce pattern on GPU.

The `fft-x`, `fft-y` and `fft-z` functions have a similar pattern of behavior. The main difference is that `fft-x` computes the Fourier Transformation on the x axis, `fft-y` on y axis and `fft-z` on z axis. These functions have several dependencies in the loops and exploit coarse grain parallelism as they were developed to target CPU architectures. The original `fft-x` sequential code is shown in Listing 3. Inside the most external loop (line 1), dependencies are established. The computation is done in blocks (the loop at line 2), where a sequence of different operations are performed for each block. The first operation copies a data of a block (loops at lines 3 and 4). The second operation processes this block (line 6). The

third operation writes the block in the output (loops at lines 7 and 8).

```
1 for(k=0; k<NZ; k++){
2   for(jj=0; jj<=NY-FFTBLOCK; jj+=FFTBLOCK){
3     for(j=0; j<FFTBLOCK; j++){
4       for(i=0; i<NX; i++){
5         y1[i][j] = x[k][j+jj][i];}}
6     cfftz(y, ...);
7     for(j=0; j<FFTBLOCK; j++){
8       for(i=0; i<NX; i++){
9         xout[k][j+jj][i] = y1[i][j];}}}}
```

Listing 3. The original fft-x code.

To add parallelism without refactoring the source code in the original disposition, we can just apply the map pattern in the most external loop so that each GPU thread performs an iteration. However, the GPU kernel will have branch divergences with five nested loops. For a proper parallel implementation on GPUs, we rewrote *fft-x*, *fft-y* and *fft-z*, eliminating the dependencies to allow fine-grained parallelism exploitation. We organized the operations in three stages that can be visualized in Listing 4. The loops of each stage are then simply collapsed to apply the parallel map pattern and launched as a GPU kernel without branch divergences.

```
1 /* stage one, copy in */
2 for(k=0; k<NZ; k++){
3   for(j=0; j<NY; j++){
4     for(i=0; i<NX; i++){
5       y[k][j][i] = x[k][j][i];}}}
6 /* stage two, compute FT on x axis */
7 for(k=0; k<NZ; k++){
8   for(j=0; j<NY; j++){
9     cfftz(y, k, j, ...);}}
10 /* stage three, copy out */
11 for(k=0; k<NZ; k++){
12   for(j=0; j<NY; j++){
13     for(i=0; i<NX; i++){
14       x[k][j][i] = y[k][j][i];}}}
```

Listing 4. The fft-x code refactored.

As `fft-x`, `fft-y` and `fft-z` compute on different axis, each function has a different memory access pattern. `fft-x` have horizontal accesses to x in the global array. We change the access pattern by refactoring the code, for instance, from `array[z][y][x]` to `array[z][x][y]`. Consequently, the memory access will be coalesced in the GPU for `fft-x` while this is done by default in `fft-y` and `fft-z` functions. This modification adds significant performance improvement.

E. Integer Sort (IS)

IS performs a sort operation for testing the integer computation speed and communication [2]. The kernel is organized into a loop where each iteration calls a rank function, which consists of a sequence of simple loop routines. The way to exploit the parallelism is to execute each iteration of a loop by a GPU thread. Therefore, our developing for GPU was quite fast in IS. The workload is small in each loop and synchronization between GPU and host is necessary. Differently from the other NPB kernels, IS have a relative small portion of code, provides a small workload, the degree of parallelism is small, and it requires several GPU atomic operations and synchronizations between the GPU kernels.

F. MultiGrid (MG)

MG is a simplified multi-grid kernel implementation [2]. The kernel is organized into a global loop, where each iteration executes the function `mg3P`. `mg3P` has five main routines: `interp`, `psinv`, `resid`, `rprj3` and `zero3`. `interp` and `rprj3` have the same code structure. The computation consists of two nested loops where inside there is a sequence of a few loops of the same number of iterations.

A possible way to implement the parallelism is to collapse the two most external loops and launch as a GPU kernel. Therefore, each thread corresponds to a iteration of the collapsed loops. Then, the threads will execute a sequence of internal loops. This is a coarse grain parallelism and results in a poor GPU usage. Another possibility is to create a sequence of GPU kernels, where each GPU kernel is the collapsing of the two most external loops with one of the most internal loops. As the sequence of internal loops must be kept, it is possible to launch a sequence of GPU kernels with no branch divergences. However, a synchronization between each kernel must done, which generates an extra overhead. The sequence of internal loops has an important feature. For instance, the iteration `i` of a loop computes `data[i]`. The iteration `i` of the next loop computes also exactly `data[i]` and so on. Consequently, it is possible to join the most internal loops into a single loop. Our parallelism strategy is to collapse the two most external loops, where each block of threads corresponds to an iteration of the collapsed loop. We also join the most internal loops and each thread of the block computes an iteration of the joined loop. This strategy exploits fine-grained parallelism, eliminates the branch divergences, and does not requires synchronization between the host.

`psinv` and `resid` are organized in the same way. In these functions, there are two nested loops. Inside these loops, there are two consecutive serial loops that should execute one after the other. We collapsed the most external loops into a GPU kernel where each block of threads correspond to an iteration of the collapsed loops. Each thread computes a subset of the internal loops. Also, we implemented the vertical access pattern in the two most internal serial loops to provide coalesced access to the memory. Function `zero3` just clears matrix values, using three simple nested loops that can be collapsed into a GPU kernel.

IV. Experiments

A. Methodology

All experiments were executed in the same machine equipped with an Intel E5-2620 2.0 GHz processor, 16 Gigabytes of RAM, and a GPU Nvidia Titan X Pascal with 3584 CUDA Cores and 12 Gigabytes of dedicated memory. The operating system was Ubuntu 14.04 LTS. The software used are CUDA 10, GCC-9, OpenCL 1.1, and OpenACC 2.5. We set the *-O3* compiler flag to compile all benchmarks. We repeated 10 times each test to compute the performance metrics.

In order to execute classes B and C workloads for the related work `NPB-OpenCL` [10] and `NPB-OpenACC` [12], it

was necessary to add `-mcmodel=large` compilation flag as well as use the command `ulimit -s unlimited`, which allows more memory in the stack. As mentioned before, the related works implemented inefficient memory allocation with large multi-dimensional arrays. Moreover, we must to refactor the source code of `NPB-OpenCL` and `NPB-OpenACC` to solve compilation errors. We faced problems with macros for `NPB-OpenCL` and some gang/work/vector configurations that were present in the `NPB-OpenACC` codes were not compatible with the GPU used in the experiments .

B. Results

The execution time and standard deviations of the experiments are shown in Table II. First column presents the NPB kernel followed by the workload evaluated. Second column lists the mean of 10 execution times in seconds (Time) and standard deviation in seconds (STDEV). Third column lists the results of the serial code, labeled as `Serial`. Fourth column lists the results of the work [10] labeled as `NPB-OpenCL`. Fifth column lists the results of the work [12] labeled as `NPB-OpenACC`. Sixth column lists the results of our work labeled as `NPB-CUDA`.

TABLE II
EXECUTION TIME AND STANDARD DEVIATIONS

Kernel. Class	Metrics	Serial	NPB-OpenCL	NPB-OpenACC	NPB-CUDA
CG.B	Time (s)	90.89	1.60	108.42	1.54
	STDEV (s)	0.27	0.00	0.06	0.02
CG.C	Time (s)	254.93	3.73	232.34	3.56
	STDEV (s)	1.12	0.01	0.13	0.03
EP.B	Time (s)	104.52	0.74	2.24	0.72
	STDEV (s)	0.04	0.01	0.00	0.01
EP.C	Time (s)	418.16	2.63	8.72	2.39
	STDEV (s)	0.48	0.02	0.00	0.01
FT.B	Time (s)	53.81	6.21	6.71	2.4
	STDEV (s)	0.42	0.00	0.01	0.00
FT.C	Time (s)	258.62	24.91	27.35	10.21
	STDEV (s)	10.72	0.03	0.01	0.01
IS.B	Time (s)	5.50	0.88	-	0.86
	STDEV (s)	0.01	0.01	-	0.01
IS.C	Time (s)	22.87	2.35	-	2.34
	STDEV (s)	0.04	0.01	-	0.00
MG.B	Time (s)	4.37	0.34	1.32	0.32
	STDEV (s)	0.02	0.00	0.02	0.00
MG.C	Time (s)	38.27	1.73	7.51	1.72
	STDEV (s)	0.17	0.00	0.02	0.01

The speedup and GPU memory consumption of the experiments are shown in the graphs presented in this section. The X-axis of each graph presents the NPB kernels and the Y-axis presents the metric evaluated. Figure 1 presents the speedup for the class B and Figure 2 presents the speedup for the class C. The GPU memory consumption of the NPB kernels for the class B is shown in the Figure 3, and the GPU memory consumption of the NPB kernels for the class C workload is shown in the Figure 4.

In CG, our CUDA implementation achieved 58.94 of speedup with the class B and 71.55 with the class C (Figures 1 and 2), which is significantly better than OpenACC version as well as 3.74% (class B) and 4.48% (class C) better than OpenCL version. Our performance improvement is due to the isolation of irregular computations for load balancing, fine-grained parallelism exploitation, and memory coalescing support. `NPB-OpenCL` achieved a closer performance to

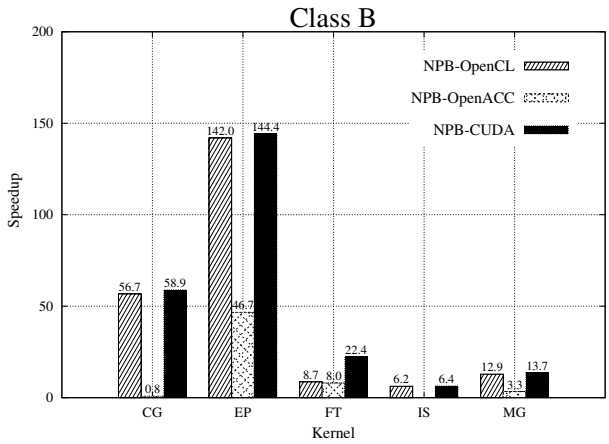

Fig. 1. Speedup with Class B

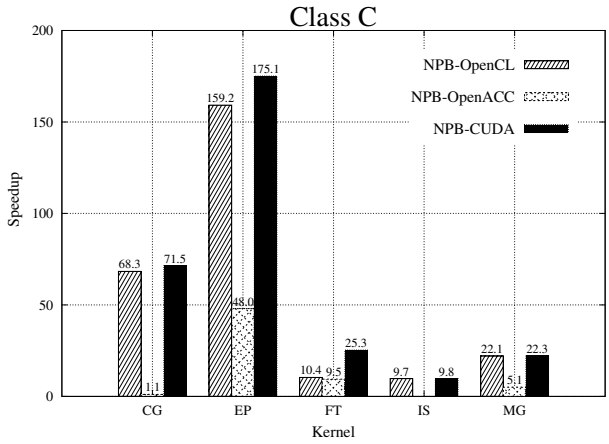

Fig. 2. Speedup with Class C

our parallel implementation because a similar isolation of irregular computations has been applied. When applying our design principles in `NPB-OpenCL` for CG, we believe that a better performance could be achieved for it. `NPB-OpenACC` achieved a very low performance due to the poor GPU parallelism exploitation given that non of our design principles were followed in their work.

When comparing memory consumption, `NPB-OpenCL` presented the higher values. The main reason is that the whole CG code is offloaded to GPU. As we only offloaded the code where parallelism is implemented, our parallel implementation consumes less GPU resources. Due to the same reason, `NPB-OpenACC` presented a memory consumption very similar to our version.

A expressive performance difference was observe for EP. Our CUDA version achieved 144.36 of speedup with class B and 175.11 with class C. This was possible due to the fine-grained parallelism exploitation and the memory coalescing. `NPB-OpenCL` achieved a similar performance to our work only with class B. Using class C, `NPB-OpenCL` was

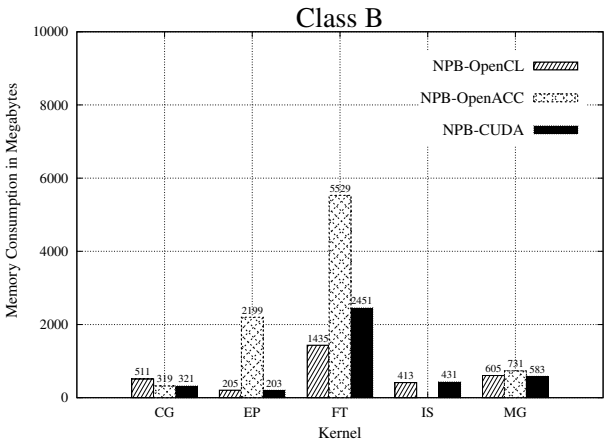

Fig. 3. Memory Consumption with Class B

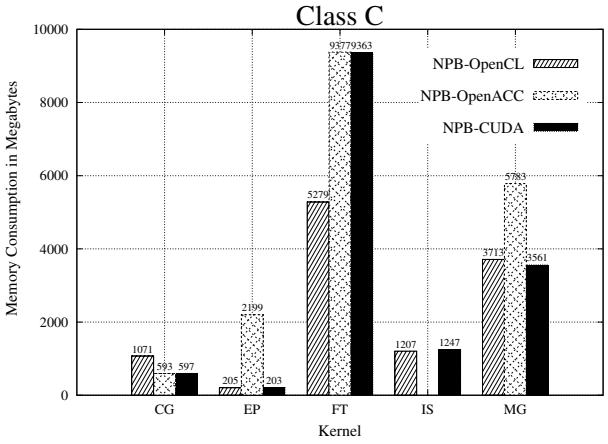

Fig. 4. Memory Consumption with Class C

significantly lower than our work. Although NPB-OpenCL implemented a similar parallelism strategy to ours, the worst performance in class C is due to the use of a larger block of memory for each thread, which generates memory bank conflicts and misses. NPB-OpenACC presents significantly lower performance than our work and has almost no improvement with a larger workload. This work implemented a coarser-grained parallelism exploitation. While our implementation creates a block of threads per iteration and each thread of each block computes a subset of pseudo-random numbers, NPB-OpenACC creates a GPU thread per iteration and the single thread computes the complete set of the pseudo-random numbers. This reduces significantly the total number of GPU threads. Consequently, the GPU utilization is low and it results in a larger GPU kernel. Therefore, the low degree of parallelism was the main factor for the lower speedup. In addition, they did not implemented memory coalescing. Concerning GPU memory consumption, our work is very similar to the NPB-OpenCL, except the memory requirements per GPU threads. We created a small array x for each thread and each thread reuses it for the computations of the GPU kernel. This way, it was possible a low memory consumption. NPB-OpenACC implementation created a complete copy of the EP larger array x for each GPU thread, increasing significantly the GPU memory consumption. Avoiding EP to run out of memory, they created a fixed number of threads, which has impacted negatively in the performance scalability when comparing class B and C.

The main reason of the performance gain in FT for the CUDA version is that we split the main routines of fft-dimension in three stages, eliminating branch divergences and exploiting a finer grain parallelism. Moreover, the memory accesses were coalesced in fft-x. NPB-OpenCL obtained less than half of our performance. This work implemented a coarser grain parallelism exploitation. They did not refactor the FT original code presented before in Section III-D. They simple mapped each iteration of the most external loop of fft-dimension to a GPU thread. This approach lowered significantly the GPU utilization with a smaller number of threads. Also, the branch divergences were not eliminated and the memory was not coalesced in fft-x. NPB-OpenACC also obtained less than half of our performance. The reason for this poor performance is similar to NPB-OpenCL.

When comparing memory consumption from the FT parallel versions (Figures3 and 4), NPB-OpenCL presented the lowest values. As they used a strategy with a smaller number of threads, the GPU memory requirements were also smaller. In contrast, this impacted on performance. NPB-OpenACC presented the highest memory requirements. They created multi-dimensional global arrays where the size of each dimension corresponds to the size of the bigger dimension given by the workload. Therefore, NPB-OpenACC uses more memory than necessary. when the dimensions of the workloads are not equal. Also, the values of the arrays will not be contiguous in the memory, which also impacts on GPUs' performance. Finally, although they transformed the accesses of the functions fft-x and fft-z, the memory coalescing is not work. The memory consumption of our CUDA version corresponds to the minimum requirements for executing FT with a finer grain parallelism. In class B, our memory consumption is significantly lower than NPB-OpenACC because the arrays have different size dimensions. In class C, the dimensions are equal, consequently, NPB-CUDA memory consumption is close to NPB-OpenACC.

NPB-CUDA and NPB-OpenCL achieved a similar performance and memory consumption in IS as both implemented similar parallelism strategies. IS has the smallest workload and degree of parallelism of the NPB. The kernel needs atomic operations and several synchronizations with the host. These characteristics impacted significantly on GPU performance at the point that it is the only kernel where our approach achieved less than 20 of speedup. NPB-OpenACC results are not shown because their IS implementation was not available.

The NPB-CUDA performance in MG is achieved due to the strategy adopted. In the functions interp and rprj3, we were able to launch a single GPU kernel with no branch diver-

gences. Since this was not possible for `psinv` and `resid`, we applied memory coalescing. `NPB-OpenCL` implemented a similar strategy and obtained also a similar performance. `NPB-OpenACC` obtained a significant lower performance. This work implemented a different parallelism strategy compared to ours. In `interp` and `rprj3` branch divergences were eliminated, however, they splitted the functions in several GPU kernels. Therefore, synchronization with the host is necessary among each GPU kernel launch in order to maintain the correctness. It adds an extra overhead, as these functions are called recurrently in MG. Although the parallelism strategy were similar to ours in `psinv` and `resid`, no memory coalescing was implemented. The GPU global memory latency and the additional synchronizations between CPU and GPU were the main reasons for `NPB-OpenACC`'s low performance. Finally, our version presented the smallest memory consumption as we just offload to the GPU only the code regions needed. `NPB-OpenCL` and `NPB-OpenACC` implemented in way that the complete MG is offloaded to GPU, requiring more GPU memory usage. Observe that `NPB-OpenACC` presented the highest memory consumption.

We have to highlight that the performance of NPB kernels varied significantly due to the different code characteristics. EP achieved higher speed-ups because it is able to exclusively use the GPU with almost no communications/synchronizations. CG and other kernels have more synchronizations between GPU and CPU. FT and MG have branch divergences that were not eliminated in significant parts of the code. FT also has different memory accesses patterns that impact speedup. MG has similar code characteristics to FT, however, it has more GPU kernel calls, synchronizations, and more branch divergences, which resulted in lower speedups than FT. Finally, IS has the smallest degree of parallelism from the NPB kernels, demands GPU atomic operations, has branch divergences that can not be eliminated, and also has many synchronizations between host and GPU.

V. Conclusion

This paper presented new and efficient NPB kernels for GPUs using CUDA. The solution was compared to the state-of-the-art NPB OpenACC and OpenCL versions. The experimental results have shown that it is possible to obtain good performance on GPUs. Our parallel versions were significantly better concerning performance and memory consumption. Since NPB represents the fluid dynamic application domain, our investigations and analysis are important to reason about the parallel implementation challenges and behaviors on GPUs in this application domain. With these contributions, other opportunities for future works remain opened. We plan to apply our design principles for improving performance and reducing the memory consumption of OpenACC and OpenCL versions. Also, to expand the research to the NPB pseudo applications using CUDA as well as to other GPU parallel programming frameworks. Another opportunity is to exploit parallelism using multi-GPU.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoa-mento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 "DECLware", the FAPERGS 01/2017-ARD project called ParaElastic (No. 17/2551-0000871-5), and the Universal MCTIC/CNPq N 28/2018 project called SParCloud (No. 437693/2018-0). We thank Nvidia's GPU Grant program for the GPU donation.

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks RNR-94-007," NASA Advanced Supercomputing Division, Tech. Rep., 1994.

[3] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance NAS-99-011," NASA Advanced Supercomputing Division, Tech. Rep., 1999.

[4] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0 NAS-95-020," NASA Advanced Supercomputing Division, Tech. Rep., 1995.

[5] W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "New Implementation and Results for the NAS Parallel Benchmarks 2," NASA Advanced Supercomputing Division, Tech. Rep., 1998.

[6] R. V. D. Wijngaart, "The NAS Parallel Benchmarks Version 2.4 NAS-02-007," NASA Advanced Supercomputing Division, Tech. Rep., 2002.

[7] M. Frumkin, H. Jin, and J. Yan, "Implementation of NAS Parallel Benchmarks in High Performance Fortran NAS-98-009," NASA Advanced Supercomputing Division, Tech. Rep., 1998.

[8] R. V. D. Wijngaart and H. Jin, "The NAS Parallel Benchmarks, Multi-Zone Versions NAS-03-010," NASA Advanced Supercomputing Division, Tech. Rep., 2003.

[9] C. Gong, J. Liu, J. Qin, Q. Hu, and Z. Gong, "Efficient Embarrassingly Parallel on Graphics Processor Unit," in *2010 2nd International Conference on Education Technology and Computer*, vol. 4, June 2010, pp. V4–400–V4–404.

[10] S. Seo, G. Jo, and J. Lee, "Performance Characterization of the NAS Parallel Benchmarks in OpenCL," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 137–148.

[11] M. Malik, T. Li, U. Sharif, R. Shahid, T. El-Ghazawi, and G. Newby, "Productivity of GPUs Under Different Programming Paradigms," *Concurr. Comput. : Pract. Exper.*, vol. 24, no. 2, pp. 179–191, Feb. 2012.

[12] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman, "NAS Parallel Benchmarks for GPGPUs Using a Directive-Based Programming Model," in *Languages and Compilers for Parallel Computing*, J. Brodman and P. Tu, Eds. Cham: Springer International Publishing, 2015, pp. 67–81.

[13] C. P. Stone and B. H. Elton, "Accelerating the Multi-zone Scalar Pentadiagonal CFD Algorithm with OpenACC," in *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, ser. WACCPD '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:7.

[14] H. Jin, M. Kellogg, and P. Mehrotra, "Using Compiler Directives for Accelerating CFD Applications on GPUs," in *OpenMP in a Heterogeneous World*, vol. 7312, 06 2012, pp. 154–168.

[15] J. Dummler and G. Runger, "Execution Schemes for the NPB-MZ Benchmarks on Hybrid Architectures: A Comparative Study," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, vol. 25, 09 2013.

[16] X. Tian, R. Xu, Y. Yan, S. Chandrasekaran, D. Eachempati, and B. Chapman, "Compiler Transformation of Nested Loops for General Purpose GPUs," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 537–556, 2016.

[17] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, and L. G. Fernandes, "Efficient NAS Benchmark Kernels with C++ Parallel Programming," in *26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, ser. PDP'18. Cambridge, UK: IEEE, March 2018, pp. 733–740.