

Impact of Reduced and Mixed-Precision on the Efficiency of a Multi-GPU Platform on CFD Applications

Gabriel Freytag[1(✉)], João V. F. Lima[2], Paolo Rech[1], and Philippe O. A. Navaux[1]

[1] Informatics Institute, UFRGS, Porto Alegre, Brazil
gfreytag@inf.ufrgs.br
[2] Informatics Institute, UFSM, Santa Maria, Brazil

Abstract. The heterogeneity of modern computing architectures allows software engineers to refine their algorithms and assign the procedures with peculiar characteristics to the device with the most suitable architecture. In particular, the latest computing architectures, such as Nvidia's Graphics Processing Units (GPUs), feature mixed-precision Floating-Point Units (FPUs) that natively support double, single, and half-precision arithmetic operations. By using mixed precisions in a specific portion of the code can significantly improve both performances and computing efficiency. The challenge is to identify which procedure to run in lower precision without jeopardizing the accuracy loss.

This work explores the use of mixed-precision FPUs in Nvidia's P100 GPU for improving the performance and efficiency of High-Performance Computing (HPC) applications. We investigate the performance and power efficiency of a Computational Fluid Dynamic (CFD) application. We first use the same precision for both arithmetic operations and storage and then use single or half-precision for arithmetic operations and double, single, or half for storage. We also explore the CUDA platform's intrinsic mathematical functions, and the results show a speedup of 1.44× with only 0.02% of accuracy loss in comparison to regular functions. By using half-precision for storage and single-precision for arithmetic operations, we were able to improve the performance up to more than 11× and power consumption of the application up to more than 3.4×, with only 0.02% of accuracy loss compared to double-precision.

1 Introduction

On the path to the exascale, both hardware and software has become increasingly heterogeneous. Naturally, the plethora of existing computational problems has several distinct intrinsic characteristics that, to be quickly and efficiently computed, require a high level of affinity with the hardware's architecture. While some characteristics have a limited level of parallelism, others can be highly parallelized, require special operations, require a large amount of memory, or

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13380, pp. 570–587, 2022.
https://doi.org/10.1007/978-3-031-10542-5_39

need high-speed interconnections. Therefore, to match all distinct characteristics of the problems with the most suitable hardware maximizing performance, the software itself becomes heterogeneous.

Today's High-Performance Computing (HPC) systems connect a large number of devices with distinct hardware architectures that are themselves becoming heterogeneous. The most common devices are Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field-Programmable Gate Array (FPGAs), and others. While some CPU devices deliver a portion of robust cores and others of reduced computational power but more energy-efficient, latest GPUs deliver a set of heterogeneous Floating-Point Units (FPUs), which have discrete floating-point precision and consequently distinct performance and power consumption.

At the same time that the heterogeneity of systems increases to support different computational characteristics, the demand for computational and storage resources exceeds current availability. Although there are a large number of optimization strategies and highly parallel systems, they do not account for the computational demand of modern large-scale applications such as scientific computing, artificial intelligence, data mining, social media, and many others. However, while some problems require exact results, others admit approximate results reducing the resource requirements and consequently improve the performance.

One of the most prevalent strategies in the approximate computing paradigm is precision scaling, which takes advantage of discrete FPUs [33]. Mixed-precision architectures usually support two or more floating-point precision arithmetic operations and allow the reduction of both storage, energy, and computational requirements. By reducing the precision of some data and arithmetic operations of the problems, it is possible to trade-off the quality of the result by the performance and energy efficiency of the execution.

Several studies explored reduced and mixed-precision in neural network training in deep learning and machine learning algorithms, increasing the performance significantly with little to no degradation in the networks' accuracy [12,22,31] as well as energy consumption reduction [16,37]. However, reduced and mixed-precision also improve both performance and energy consumption of scientific applications as in iterative refinement methods [17,18], molecular dynamics [24], and computational fluid dynamics [9]. Although the last one accelerated a CFD using mixed-precision, no half-precision arithmetic operations were available in the hardware used by the authors. Moreover, they do not exploit the CUDA platform's intrinsic mathematical functions that have improved performance in comparison to the regular ones.

In this paper, we evaluate the impact of reduced and mixed-precision on the performance, energy efficiency, and accuracy of stencil computing on a multi-GPU platform that natively supports mixed-precision floating-point operations. As a case study, we use the well-known Computational Fluid Dynamics (CFD) application called Lattice-Boltzmann Method (LBM) with a three-dimensional model, namely, D3Q19.

2 Reduced and Mixed-Precision Algorithms

In this section, we present the three multi-GPU implementations of the 3D CFD application LBM based on [27]. One version has fixed precision for data storage and computations defined at compile time. The other two versions have mixed precision for arithmetic operations, single or half, and data storage defined at compile time in any of the three precision types.

The three versions are:

multigpu The floating-point arithmetic operations and storage are in the same floating-point precision selected at compile time. This version performs all arithmetic operations with regular mathematical functions from CUDA.

multigpu_float This implementation performs all floating-point arithmetic operations in single-precision using intrinsic mathematical functions from CUDA. Data storage is defined at runtime in double, single or half.

multigpu_half2 This version does the arithmetic operations in half-precision with intrinsic mathematical functions from CUDA. In addition, it uses vectorized data types and NVIDIA's 2-way half-precision functions that perform two half arithmetic operations in parallel on a single-precision FPU.

2.1 Reduced-Precision Implementation

Our multi-GPU algorithm decomposes the data domain into subdomains where each subdomain is assigned to a device. Figure 1 shows the pseudocode of the multi-GPU implementation of our CFD application. It has five kernels, and the first (line 3) initializes the density of each particle. This first kernel runs at the pre-processing stage since this kernel runs only once. Afterwards, the simulation of the fluid flow starts by redistributing the propagation forces of each particle, as can be seen at line 8.

The next step of the simulation is the propagation of the particles (line 19) in which particles can leave one subdomain and move to another one. The fluid particles have nineteen propagation directions. After decomposing the data domain into one subdomain per device, the propagation becomes restricted to the particles of each subdomain. Consequently, each GPU device will perform disjoint simulations just with its fragment of the domain and do not contribute to the global simulation. In order to perform the simulation in parallel and synchronize each time step, we used ghost zones to synchronize the local subdomains of each GPU device with its neighbors. Each time step copies the boundaries of each subdomain to temporary buffers owned by the neighbors (line 12), and then it copies the neighbor's boundaries from the buffers into the subdomains (line 15).

As the platform that we are exploring has only four GPU devices, the decomposition of the domain in more than just one dimension is not necessary. Splitting just one of the dimensions of a domain in half results in two subdomains, and each one requires a copy to each of its neighboring subdomains of two faces and four diagonals, which in this case the faces already have they included. As we divide more dimensions in half, the number of subdomains and copies of faces

doubles and diagonals triples. Thus, using only four GPUs, splitting just one of the dimensions in the number of devices involved in the computation is enough and minimizes the ghost zones' synchronization overhead in our case.

Finally, as we use up to four GPU devices in a collaborative way, manage these devices using only one CPU thread substantially harms the performance of the implementation. To manage each GPU independently, we use one CPU thread per GPU device by parallelizing the for loops We use `#pragma omp parallel for num_threads(deviceCount)` on the for loops of lines 6, 11, 14, and 17 of pseudocode from Fig. 1 to manage multiple devices in parallel.

2.2 Single-Precision Implementation

This implementation performs all arithmetics in single-precision instead of performing on the same precision used to store the domain's data in memory. Moreover, it uses intrinsic mathematical functions from CUDA platform rather than regular mathematical operations. This mixed-precision implementation uses the same multi-GPU algorithm previously described in Sect. 2.1 and only changes the kernels' arithmetic operations precision and mathematical functions.

Figure 2 shows an example of the use of the CUDA platform's intrinsic mathematical functions. To add up two single-precision (or float) numbers, instead of using the regular addition function +, we use the intrinsic addition function *__fadd()*. We do the same for the other mathematical functions and, instead of using the regular subtraction function −, the regular multiplication function ∗, and the regular division function /, we use the respective intrinsic functions *__fsub()*, *__fmul()*, and *__fdiv()*. For each of the single-precision intrinsic mathematical functions, we use the round to nearest even rounding mode by suffixing each function with *_rn*. This intrinsic functions map to fewer native instructions and, therefore, are faster than the regular functions but less accurate.

2.3 2-Way Half-Precision Implementation

This version makes use of arithmetic operations in half-precision and two parallel operations in a single FPU. The implementation packs data into a *half2* vector data type in order to issue two operations per FPU. As we only use double, single, or half-precision to store the domain's data in memory, we need to convert these data to a *half2* data type before the call of computation kernels. Moreover, we need to rearrange the operands in order to avoid two computations on the same data.

Figure 3 shows an example of the 2-way half-precision arithmetic operations. At lines 1, 2, 3, and 4 of the pseudocode, we create local registers with type *half2* placing in the first position of the intrinsic function *__floats2half2()* the data of one operation, and in the second position of the function the data of the second operation. Therefore, to add up the data of *a1* and *a2* and *b1* and *b2*, first, we need to create a register *a1b1* containing the data of *a1* and *b1* (line 1) and other *a2b2* containing the data of *a2* and *b2* (line 2). Then we can add up

```
1: Initialize parameters
2: for Each N_x × N_y × N_z do
3:    Initialize conditions
4: end for
5: for Each time step do
6:    for Each device do
7:       for Each N_x × N_y × N_z do
8:          Redistribute
9:       end for
10:   end for
11:   for Each device do
12:      CopyBoundariesToNeighborBuffers
13:   end for
14:   for Each device do
15:      CopyNeighborBoundariesFromBuffers
16:   end for
17:   for Each device do
18:      for Each N_x × N_y × N_z do
19:         Propagate
20:         Bounceback
21:         Relaxation
22:      end for
23:   end for
24: end for
```

Fig. 1. CFD Multi-GPU algorithm.

```
1: float a = __fadd_rn(a1, a2)
2: float b = __fsub_rn(b1, b2)
3: float c = __fmul_rn(c1, c2)
4: float d = __fdiv_rn(d1, d2)
```

Fig. 2. CUDA single-precision intrinsic math functions.

```
1:  __half2 a1b1 = __floats2half2_rn(a1, b1)
2:  __half2 a2b2 = __floats2half2_rn(a2, b2)
3:  __half2 c1d1 = __floats2half2_rn(c1, d1)
4:  __half2 c2d2 = __floats2half2_rn(c2, d2)
5:
6:  __half2 ab = __hadd2(a1b1, a2b2)
7:  __half2 cd = __hmul2(c1d1, c2d2)
8:
9:  __half a = ab.x
10: __half b = ab.y
11: __half c = cd.x
12: __half d = cd.y
```

Fig. 3. CUDA Half2 intrinsic math functions.

these four data by calling the intrinsic addition function *__hadd2(a1b1, a2b2)* (line 6) and saving the result to the *ab* register. Finally, to get the answer to the addition of *a1* and *a2*, and *b1* and *b2*, we call *ab.x* (line 9) and *ab.y* (line 10), respectively.

3 Tools and Methods

This section details the hardware configuration and methodology used in our reduced and mixed-precision experiments.

3.1 Evaluation Platform

We target a experimental platform, called Blaise, enhanced with GPUs capable of native mixed-precision floating-point arithmetics. It consists of two NUMA (Non-Uniform Memory Access) nodes, each with one Intel Xeon E5-2699 v4 (Broadwell) processor with 22 cores running at 2.2 GHz or 3.6 GHz with Turbo Boost (44 cores total), and 256 GB of main memory. The machine has 4 NVIDIA Tesla P100 (Pascal architecture) GPUs interconnected with NVIDIA's NVLink, each having 3,584 CUDA cores running at 1.48 GHz and 16 GB HBM2 global memory (a total of 14,336 CUDA cores and 64 GB of GPU global memory).

We used as software environment the C/C++ compiler GCC version 8.3 and NVIDIA's CUDA toolkit release 10.1, version 10.1.168. The flags used to compile all the three implementations were *-O3*, *-m64*, *-arch=sm_60*, and *-Xcompiler -fopenmp -lgomp*. All this was running over a Debian GNU/Linux 10 (buster) operational system.

3.2 Methodology

We only report the execution time of the processing stage, which is the time taken to run all the iterations of the time step for loop (lines 5 to 24 of the pseudocode in Fig. 1), that we fixed to 200 iterations in all our experiments. Since we investigate the performance of the three implementations using from 1 up to 4 GPU devices, we split the domain in the same number of subdomains as devices involved in the computation. The split step occurs in the pre-processing stage and does not appear in the execution time measurements.

The results reported are a mean of 10 samples using domains of size up to the largest one that fits in single GPU memory using double-precision. Although splitting the domain across multiple GPUs allows the use of larger domains, to investigate the impact of incrementing the number of GPUs from one to four, we run experiments using five 3D domains with size 128, 192, 256, 320, and 352. All these domains have a rectangular obstacle placed in the middle of the flow. Moreover, for each implementation, we evaluate the performance of storing the domain in memory using double, single, and half-precision floating-point.

The accuracy of the implementations was measured by the absolute floating-point error. To determine the absolute error of an experiment, we subtract its final fluid density from the exact fluid density computed using double-precision. We show the difference between the exact density and the density returned by the experiment (absolute error) as a percentage of the exact density.

To collect the GPU's resource utilization metrics, we used the NVIDIA's System Management Interface (*nvidia-smi*) command-line utility. This utility

allows to query the GPU device state and retrieve information about its configuration and its instant resource utilization. During each experiment, we collect the instantaneous state of all GPUs at every teen milliseconds. In the end, we sum up all the records related to each execution and divide it by the respective number of records. Moreover, we only collect the informations of the devices involved in the respective experiment, does not including data from idle devices. Therefore, we report the mean of teen executions of each experiment for the metrics: power consumption, GPU utilization, memory usage, and GPU memory read/write utilization.

4 Experimental Results

In this section, we present the experimental results of our reduced and mixed-precision implementations by using double, single, and half-precision floating-point (FP) data types to store the CFD's domain in memory and from one up to four GPU devices.

4.1 Performance and Accuracy Evaluation

Table 1 shows the execution time, speedup with *multigpu* and single GPU on each FP as baseline, floating-point absolute error, total power consumption, GPU capacity utilization, memory usage and memory read or write overtime utilization for each of our three implementations using double, single and half-precision floating-point data types from one up to 4 GPU devices.

Looking at the execution times of the *multigpu* implementation on a single GPU, we observe that the reduction of the floating-point precision substantially improves its performance. Compared to the execution time using double-precision to store the data in memory, single-precision improves the performance of the implementation by up to 1.72× with no accuracy loss. With half-precision, performance improves by up to 2.57× with an accuracy loss of 0.53%.

Using more GPUs to perform the computation in parallel made it possible to improve the implementation's performance further. Compared to double-precision and single GPU, single-precision improved the performance of the *multigpu* implementation by up to 3.24, 4.67×, and 5.9× using two, three, and all four GPUs, respectively, and no accuracy loss. With half-precision, improvements were up to 4.86×, 7.10×, and 8.59× with an accuracy loss of 0.84%, 1.05%, and 0.22% with two, three, and four GPUs, respectively.

If we look at the performance of the *multigpu_float* and *multigpu_half2* implementations over *multigpu* with double-precision and single GPU, it is possible to see that both implementations improve the performance significantly. More precisely, *multigpu_float* improved by up to 1.21x with no accuracy loss, and *multigpu_half2* improved by 1.17x with an accuracy loss of 0.09%.

Table 1. Execution time, speedup, floating-point absolute error, power consumption, GPU utilization, amount of GPU memory used, and GPU memory utilization (read or write). It corresponds to the use of double, single, and half-precision floating-point, one up to 4 GPU devices, and a 3D domain of size $352 \times 352 \times 352$. The speedup is calculated over *multigpu*'s double-precision performance on a single GPU.

Implementation	FP	GPUs	Execution (s)	Speedup	FP Error (%)	Power (W)	GPU (%)	Memory (MiB)	Memory R/W (%)
multigpu	double	1	65.82	1.00	0.00	1985.93	99.39	12995.00	48.78
		2	35.02	1.88	0.00	1939.48	96.25	6743.00	45.53
		3	24.40	2.70	0.00	1952.66	92.44	4623.00	43.08
		4	19.47	3.38	0.00	1986.84	90.44	3583.00	40.87
	single	1	38.17	1.72	0.00	1266.33	99.35	6671.00	76.80
		2	20.34	3.24	0.00	1234.23	96.01	3547.00	72.09
		3	14.10	4.67	0.00	1240.12	93.27	2487.00	69.27
		4	11.16	5.90	0.00	1255.06	90.62	1967.00	65.60
	half	1	25.57	2.57	0.53	817.95	99.12	3511.00	86.62
		2	13.56	4.86	0.84	810.77	95.39	1947.00	81.60
		3	9.27	7.10	1.05	812.82	93.58	1419.00	79.30
		4	7.67	8.59	0.22	836.52	87.34	1159.00	72.85
multigpu_float	double	1	54.49	1.21	0.00	1492.80	99.44	12995.00	48.47
		2	29.44	2.24	0.00	1480.95	95.33	6743.00	44.42
		3	20.71	3.18	0.00	1514.98	91.28	4623.00	41.63
		4	16.81	3.92	0.00	1557.10	88.05	3583.00	39.00
	single	1	26.42	2.49	0.00	802.79	99.00	6671.00	64.99
		2	14.60	4.51	0.00	812.39	93.87	3547.00	58.64
		3	10.35	6.36	0.00	830.05	89.31	2487.00	55.13
		4	8.52	7.73	0.00	849.41	84.29	1967.00	50.44
	half	1	17.19	3.83	0.02	521.63	98.58	3511.00	73.62
		2	9.34	7.05	0.02	520.19	94.15	1947.00	68.15
		3	6.67	9.86	0.02	534.70	89.62	1419.00	64.14
		4	5.66	11.64	0.02	566.10	82.28	1159.00	57.29
multigpu_half2	double	1	56.49	1.17	0.09	1507.69	99.47	12995.00	45.40
		2	30.58	2.15	0.12	1503.53	95.55	6743.00	41.91
		3	21.53	3.06	0.15	1542.11	92.31	4623.00	39.80
		4	17.37	3.79	0.06	1587.30	88.73	3583.00	36.89
	single	1	27.40	2.40	0.09	816.62	99.06	6671.00	69.55
		2	15.12	4.35	0.12	820.43	93.92	3547.00	63.28
		3	10.66	6.18	0.15	837.15	89.90	2487.00	59.68
		4	8.72	7.55	0.06	863.72	85.72	1967.00	54.81
	half	1	18.33	3.59	0.09	560.20	98.74	3511.00	73.98
		2	10.03	6.56	0.12	562.62	93.35	1947.00	67.97
		3	6.91	9.53	0.15	570.02	91.38	1419.00	65.93
		4	5.85	11.26	0.06	595.69	84.03	1159.00	59.16

By using all four GPUs, the performance increases more than 11×. While *multigpu* improves by up to 3.38×, 5.9×, and 8.59× with double, single, and half-precision, respectively, *multigpu_float* improves by up to 3.92×, 7.73×, and 11.64×, respectively. Already *multigpu_half2* improves by up to 3.79×, 7.55×, and 11.26×, respectively. *multigpu* and *multigpu_float* had an accuracy loss of 0.22% and 0.02%, respectively, only with half-precision, while *multigpu_half2* had an accuracy loss of 0.06% for all three precisions.

In Fig. 4, it is possible to see that difference in performance between both *multigpu_float* and *multigpu_half2* and *multigpu* becomes more prominent as the size of the domain increases. Furthermore, it is also possible to observe that the execution times of both *multigpu_float* and *multigpu_half2* are very similar on all domain sizes. Moreover, as the domain size increases, the accuracy loss of all three implementations reduces.

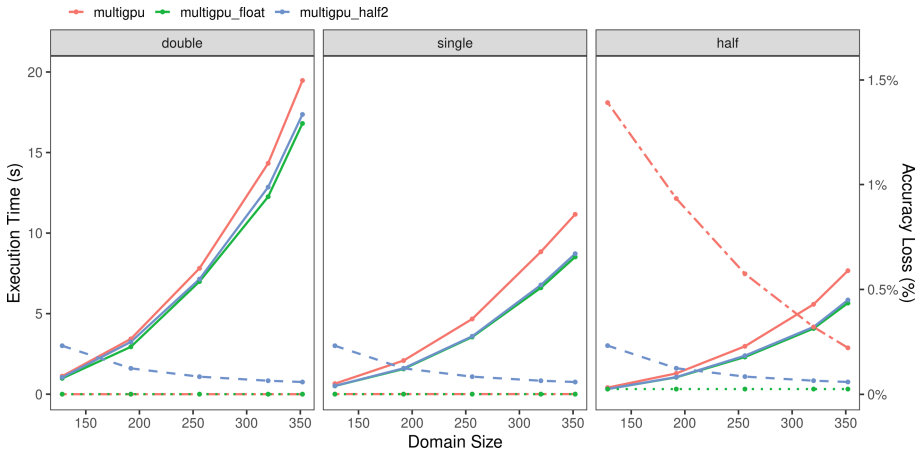

Fig. 4. Execution times of the three implementations (solid lines) using double, single, and half-precision floating-point to store the data in memory and their accuracies (dashed lines) on three-dimensional domains of size 128, 192, 256, 320, and 352 using four GPU devices.

4.2 Power Consumption Evaluation

In column seven of Table 1, we present the power consumption for each of the three implementations using from 1 up to 4 GPUs and double, single, and half-precision data types to store data. Looking only at the consumption while using a different number of GPUs within the same implementation and storage data type, it is possible to see that the power consumption remains almost the same when using one or more GPUs in parallel. The difference between the lowest and highest values for power consumption on the *multigpu* implementation is 2.38%, 2.53%, and 3.07%, for double, single, and half-precision as storage data types, respectively. Already for the *multigpu_float* implementation, the differences are 4.89%, 5.48%, and 8.10%, and for the *multigpu_half2* implementation, 5.27%, 5.45%, and 5.95% for double, single, and half-precision storage data types, respectively.

Let us compare the power consumption of the different storage data types on the same implementation. It is possible to observe that in all three implementations, the consumption was reduced by more than 50%, going from double to half-precision. Using half instead of double-precision on the *multigpu* implementation reduces approximately 57% of the power consumption with all four GPUs. Already on the *multigpu_float* implementation, it reduces more than 63% and more than 62% on the *multigpu_half2* implementation.

Now, let us analyze the power consumption between different implementations using the same data type for storage. We can see that using CUDA's intrinsic mathematical functions instead of the regular mathematical functions results in a significant reduction in power consumption. While the *multigpu* implementation consumes 1986.84, 1255.06, and 836.52 W with all for GPUs

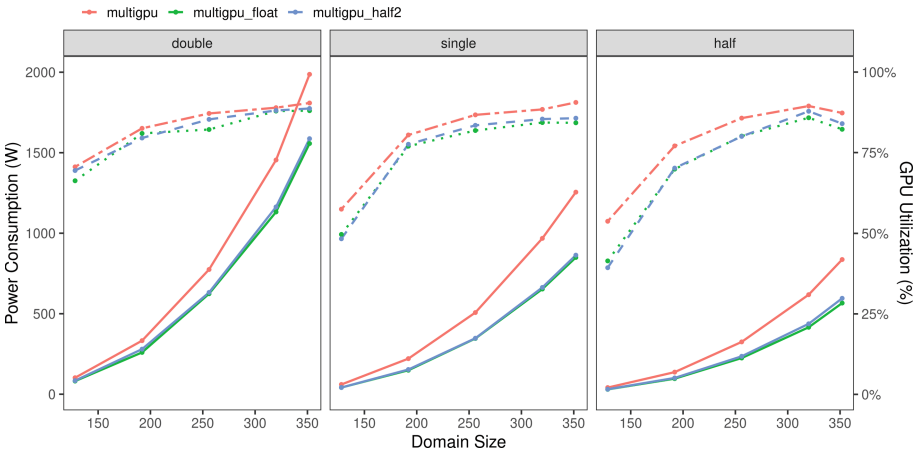

Fig. 5. Power consumption (solid lines) and the GPU utilization (dashed lines) of the three implementations using double, single, and half-precision to store the data. It corresponds to the use of four GPUs and domain size 128, 192, 256, 320, and 352.

in double, single, and half-precision, respectively, *multigpu_float* consumes only 1557.10, 849.41, and 566.10 W, and *multigpu_half2* consumes 1587.30, 863.72, and 595.69 W. This corresponds to a reduction of approximately 21%, 32%, and 32% for double, single, and half-precision on the *multigpu_float* implementation, and 20%, 31%, and 28% on *multigpu_half2*, respectively.

Figure 5 shows the power consumption of each implementation on five different domain sizes using all four GPU devices. As can be observed, the power consumption of all three implementations increases with larger domain sizes. However, let us look at the consumption of the *multigpu* implementation, which uses regular mathematical functions from CUDA, and compare it to the consumption of the *multigpu_float* and *multigpu_half2* implementations, which in turn use CUDA's intrinsic mathematical functions. It is clear that the difference in power consumption increases as the domains' size increases.

4.3 Memory Utilization Evaluation

In Table 1, we present the amount of GPUs' memory used (column *Memory*) by the three implementations to store the data in double, single, and half-precision. As can be seen, each of the three implementations used exactly the same amount of memory for each of the storage data types. With four GPUs, a total of 3583 MB, 1967 MB, and 1159 MB of global memory was used to store the data in double, single, and half-precision. However, while with a single GPU in double-precision the total amount of memory used was 12995 MB, with all four GPUs the total amount of memory used was 14332 MB, since 3583 MB of memory was used on each of the four GPUs. This is an increase of approximately 10%.

In the column *Memory R/W* of the Table 1, we also show the percentage of time in which read or write operations were performed in memory during

the runtime. First, if wee look at the percentage of operations performed when using a different number of GPUs with the same data type to store the data, it is possible to see that the percentage reduces significantly as the number of GPUs increases. Second, if we compare the percentage of operations for the different storage data types, it is possible to see that it increases as the precision decreases. Third, if we look at the percentage of operations performed by the *multigpu_float* and *multigpu_half2* implementations in comparison to *multigpu*, we can see that the two implementations performing arithmetic operations sing CUDA's intrinsic mathematical functions have a higher percentage of memory read or write operations than the one using regular mathematical functions from CUDA.

As well as with the amount of memory, the percentage of read or write operations performed during runtime is given per GPU. Therefore, while using *multigpu*, double-precision, and a single GPU, the percentage of the time were memory operations were performed was about 48%. With all four GPUs, each GPU performed memory operations about 40% of the time.

Figure 6 shows the amount of memory and memory utilization (percentage of read or write operations performed during time) of each of the three implementations on five different domain sizes using all four GPU devices. As expected, the amount of memory used increases with larger domain sizes. Moreover, while the domain size increases, the memory read or write operations performed during the runtime increases at an even-slowing pace after domain size 256.

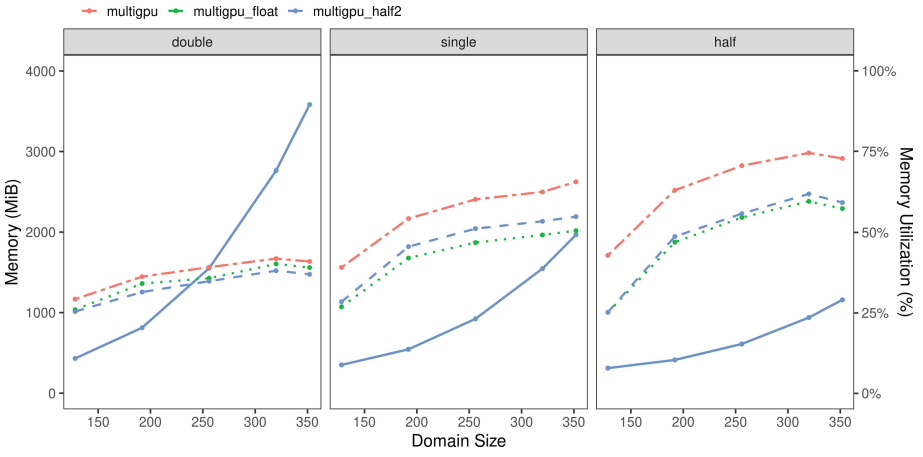

Fig. 6. Amount of memory (solid lines) and memory read/write utilization (dashed lines) of the three implementations using double, single, and half-precision to store the data. It corresponds to the use of four GPUs and domain size 128, 192, 256, 320, and 352. As the amount of memory used by the three implementations is the same on each precision, the solid lines overlap.

5 Discussion

In Sect. 4.1, it was possible to see that fastening the arithmetic operations to higher precision and storing the data in memory using a lower precision substantially improves the performance while minimizes the accuracy loss. As higher precision arithmetic operations are less likely to accumulate errors, at the moment of casting the final results to a lower precision, the loss of data is smaller and reduces the accuracy loss. Comparing the performance of the *multigpu_float* implementation using half-precision and all four GPUs against the *multigpu* implementation using double-precision and a single GPU, it is possible to see that the performance is improved by more than 11× with only an accuracy loss of 0.02%.

As seen in Sect. 4.2, by reducing the precision of floating-point arithmetic operations, the total power consumption reduces substantially compared to more accurate operations. Since reduced-precision floating-point units require a lower number of hardware instructions to perform the arithmetic operations, the hardware area needed to implement the instructions is much smaller than that of higher precision FPUs. Consequently, the total amount of power consumed nearly drops by half compared to FPUs with twice the floating-point precision.

Despite reducing the floating-point precision improves the power consumption, it harms the accuracy of the arithmetic operations. As can be observed in Table 1, while using half-precision for the arithmetic operations and storage of data (*multigpu* implementation) reduces the total power consumed by nearly 58% compared to double-precision, the accuracy drops by up to 1.05% using three GPU devices. By using all four GPUs, the accuracy loss reduces to just 0.22%.

However, by mixing different floating-point precisions, it is possible to mitigate the accuracy loss maintaining the improved performance and reduced power consumption of lower precision floating-point arithmetics. By storing the data using half-precision and performing all the arithmetic operations in single-precision, it is possible to improve the performance with nearly the same power consumption than using just half-precision for both arithmetics and storage with minimal accuracy loss. As can be seen in Table 1, the performance of the *multigpu_float* implementation using half-precision and all four GPUs is approximately 35% higher than *multigpu*, consuming about 32% less power and with an accuracy loss of only 0.02%, compared to 0.22% on the *multigpu* implementation. Comparing the power consumption of half-precision in the *multigpu_float* implementation against the power consumption of double-precision in *multigpu* implementation using all four GPUs, the *multigpu_float* consumes approximately 3.5× less power.

These substantial improvements, however, are not only due to the use of mixed precision, but also due to the use of faster mathematical functions. As previously mentioned in Sect. 2.2, the *multigpu_float* implementation uses the single-precision intrinsic mathematical functions of the CUDA platform, which map to a lower number of hardware instructions and, hence, are faster than the

regular functions. Therefore, by storing the data using a lower precision but performing the arithmetic operations in a higher floating-point precision using faster mathematical functions, it was possible to improve the performance and power consumption while minimizing the absolute floating-point error. Comparing the performance of the *multigpu_float* implementation against the *multigpu* implementation using single-precision and a single GPU, it is possible to see that the intrinsic mathematical functions improve the performance of the *multigpu_float* implementation by up to 1.44× with an accuracy loss of only 0.02%.

Moreover, if we look at the power consumption between the *multigpu_float* and the *multigpu* implementations, it is possible to see that the use os intrinsic mathematical functions also reduce the power consumption significantly. While using single-precision and a single GPU the *multigpu* consumes 1266.33 W, the *multigpu_float* consumes only 802.79 W, a reduction of approximately 1.58× without accuracy losses. This corresponds to a reduction of about 36% only by using faster (and therefore, less precise) CUDA mathematical functions.

Furthermore, by storing the data using lower precision while performing all the arithmetic operations using a higher one, the amount of memory required to store the data reduces and contributes to the improvement of the performance and power consumption. As can be seen in Table 1, the use of half-precision instead of single-precision to store the data in the *multigpu_float* implementation reduces the amount of memory required up to 47% and the power consumption and execution time by up to 35% on a single GPU.

Not only that, but the use of intrinsic mathematical functions instead of regular functions helps to reduce the pressure on the memory. As shown in Table 1, with a single GPU and half-precision, in *multigpu* implementation, read or write operations were performed 86.62% of the time. In contrast, in *multigpu_float* implementation, operations were performed 73.62% of the time. With all four GPUs, in *multigpu* implementation, each GPU performed read or write operations 72.85% of the time and *multigpu_float* 57.29%, a reduction of about 21%.

6 Related Work

Most research on reduced and mixed precision has been devoted to neural network training in deep learning and machine learning for performance [7,12,22,23,26,31,36] and energy efficiency [16,37,40]. In [14] the authors evaluate the reliability of reduced precision on DNNs through fault injection.

In addition, many linear algebra algorithms are subject of study on reduced precision. The authors in [17,18] evaluate the impact on in Linear solvers using NVIDIA Tensor cores, while [2] evaluated performance and energy efficiency of a GMRES solver. The works with BLAS algorithms include GPU and FPGA usage [32], batched matrix multiplication [1], and mixed precision [25]. Mixed precision in sparse linear algebra algorithms was evaluated over multiple platforms [6] and a Cell BE processor [5].

Libraries and tools have been proposed to program reduced or mixed precision algorithms in NVIDIA GPUs. In [28] the authors survey programming interfaces for half precision, and [11] proposes optimized reduction and scan operations. A library for mixed precision computations is described in [30], while [20] designed a tool to convert CUDA code into half precision for NVIDIA Pascal GPUs. An architectural model was designed in [35] in order to simulate half precision and tensor cores on NVIDIA GPUs.

Studies on scientific applications have showed an acceptable tradeoff between performance and precision error. Scientific benchmarks were evaluated on GPU [15] and Intel KNL/KNM [13]. Other applications are radio astronomy [38], earthquake city simulations [21], weather forecast [19], robotics [29], and IoT devices [34]. In CFD several works related evaluation of molecular dynamics [24] and Lattice Quantum Chromodynamics [3,4,8–10,39] on GPUs.

Compared to previous works, our paper explores a new approach with a multi-GPU algorithm based on [27] and optimized mixed-precision methods. Besides, we evaluate performance and energy in the context of multiple GPUs.

7 Conclusion and Future Work

In this paper, we presented a reduced and two mixed-precision floating-point multi-GPU implementations using as a case study, a well-known CFD application. In the first one, all arithmetic operations were performed using the same precision of the stored data and regular mathematical functions from the CUDA platform. The second and third implementations performed all arithmetics using single and half-precision, respectively, and the faster but slightly less accurate intrinsic mathematical functions. We provide results and analysis using double, single, or half-precision to store the CFD domains' data in GPU's memory on each implementation.

We show that by fastening the precision of the arithmetic operations to single and storing the CFD domains' data in half-precision improves the performance by more than 11× and power consumption by more than 3.4× with just an accuracy loss of up to 0.02%. These findings show that storage precision plays an essential role in the performance of reduced and mixed-precision implementations, once the reduction in storage precision minimizes pressure in memory while fixed higher precision arithmetics minimizes accuracy loss in lower storage precision. Furthermore, we point out that part of this improvement is due to the use of intrinsic mathematical functions instead of the regular ones, which improves the performance of single-precision arithmetic operations by up to 1.44× with only 0.02% of accuracy loss.

As future works, we aim to extend our work on mixed-precision towards a broader set of applications, especially for computing-intensive ones. Since the performance of the implementation in which we explored the 2-way half-precision arithmetics in parallel on a single-precision FPU was nearly the same as the single-precision one, we aim to explore optimizations on these procedures. Moreover, we also intend to explore the use of intrinsic mathematical functions over the regular functions more deeply.

Acknowledgment. This research received funding from the Petrobras project, grant n. 2016/00133-9, by CAPES (Brazil) - Finance Code 001, by the project "GREEN-CLOUD" (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014, and the project 406182/2021-3 from CNPq.

References

1. Abdelfattah, A., Tomov, S., Dongarra, J.: Fast batched matrix multiplication for small sizes using half-precision arithmetic on GPUs. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 111–122, May 2019. https://doi.org/10.1109/IPDPS.2019.00022
2. Anzt, H., Rocker, B., Heuveline, V.: Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. Comput. Sci. Res. Dev. **25**(3), 141–148 (2010). https://doi.org/10.1007/s00450-010-0124-2
3. Babich, R., Clark, M.A., Joó, B., Shi, G., Brower, R.C., Gottlieb, S.: Scaling lattice QCD beyond 100 GPUs. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 70:1–70:11. ACM, New York, NY, USA (2011). https://doi.org/10.1145/2063384.2063478, http://doi.acm.org/10.1145/2063384.2063478
4. Babich, R., Clark, M.A., Joó, B.: Parallelizing the QUDA library for multi-GPU calculations in lattice quantum chromodynamics. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC, USA (2010). https://doi.org/10.1109/SC.2010.40
5. Baboulin, M., et al.: Accelerating scientific computations with mixed precision algorithms. Comput. Phys. Commun. **180**(12), 2526–2533 (2009). https://doi.org/10.1016/j.cpc.2008.11.005, http://www.sciencedirect.com/science/article/pii/S0010465508003846
6. Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., Tomov, S.: Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. ACM Trans. Math. Softw. **34**(4), 17:1–17:22 (2008). https://doi.org/10.1145/1377596.1377597, http://doi.acm.org/10.1145/1377596.1377597
7. Chu, T., Luo, Q., Yang, J., Huang, X.: Mixed-precision quantized neural networks with progressively decreasing Bitwidth. Pattern Recogn. **111**, 107647 (2021)
8. Clark, M.A., Joó, B., Strelchenko, A., Cheng, M., Gambhir, A., Brower, R.C.: Accelerating lattice QCD multigrid on GPUs using fine-grained parallelization. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 68:1–68:12. IEEE Press, Piscataway, NJ, USA (2016). http://dl.acm.org/citation.cfm?id=3014904.3014995
9. Clark, M., Babich, R., Barros, K., Brower, R., Rebbi, C.: Solving lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. **181**(9), 1517–1528 (2010). https://doi.org/10.1016/j.cpc.2010.05.002, http://www.sciencedirect.com/science/article/pii/S0010465510001426
10. Clark, M., Strelchenko, A., Vaquero, A., Wagner, M., Weinberg, E.: Pushing memory bandwidth limitations through efficient implementations of Block-Krylov space solvers on GPUs. Comput. Phys. Commun. **233**, 29–40 (2018). https://doi.org/10.1016/j.cpc.2018.06.019, http://www.sciencedirect.com/science/article/pii/S0010465518302273

11. Dakkak, A., Li, C., Xiong, J., Gelado, I., Hwu, W.M.: Accelerating reduction and scan using tensor core units. In: Proceedings of the ACM International Conference on Supercomputing, ICS 2019, pp. 46–57. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3330345.3331057, http://doi.acm.org/10.1145/3330345.3331057
12. Das, D., et al.: Mixed precision training of convolutional neural networks using integer operations, pp. 1–11 (2018). http://arxiv.org/abs/1802.00930
13. Domke, J., et al.: Double-precision FPUs in high-performance computing: an embarrassment of riches? In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 78–88, May 2019. https://doi.org/10.1109/IPDPS.2019.00019
14. dos Santos, F.F., Navaux, P., Carro, L., Rech, P.: Impact of reduced precision in the reliability of deep neural networks for object detection. In: 2019 IEEE European Test Symposium (ETS), pp. 1–6, May 2019. https://doi.org/10.1109/ETS.2019.8791554
15. Fernandes Dos Santos, F., Lunardi, C., Oliveira, D., Libano, F., Rech, P.: Reliability evaluation of mixed-precision architectures. In: Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, pp. 238–249 (2019). https://doi.org/10.1109/HPCA.2019.00041
16. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: 32nd International Conference on Machine Learning, ICML 2015, vol. 3, pp. 1737–1746 (2015). http://www.jmlr.org/proceedings/papers/v37/gupta15.pdf
17. Haidar, A., Tomov, S., Dongarra, J., Higham, N.J.: Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, pp. 47:1–47:11. IEEE Press, Piscataway, NJ, USA (2018). https://doi.org/10.1109/SC.2018.00050
18. Haidar, A., Wu, P., Tomov, S., Dongarra, J.: Investigating half precision arithmetic to accelerate dense linear system solvers. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA 2017, pp. 10:1–10:8. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3148226.3148237, http://doi.acm.org/10.1145/3148226.3148237
19. Hatfield, S., Chantry, M., Düben, P., Palmer, T.: Accelerating high-resolution weather models with deep-learning hardware. In: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2019, pp. 1:1–1:11. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3324989.3325711, http://doi.acm.org/10.1145/3324989.3325711
20. Ho, N., Wong, W.: Exploiting half precision arithmetic in Nvidia GPUs. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7, September 2017. https://doi.org/10.1109/HPEC.2017.8091072
21. Ichimura, T., et al.: A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, pp. 49:1–49:11. IEEE Press, Piscataway, NJ, USA (2018). https://doi.org/10.1109/SC.2018.00052
22. Jia, X., et al.: Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes (2018). http://arxiv.org/abs/1807.11205

23. Jordà, M., Valero-Lara, P., Peña, A.J.: Performance evaluation of cuDNN convolution algorithms on NVIDIA Volta GPUs. IEEE Access **7**, 70461–70473 (2019). https://doi.org/10.1109/ACCESS.2019.2918851
24. Le Grand, S., Götz, A.W., Walker, R.C.: SPFP: speed without compromise - a mixed precision model for GPU accelerated molecular dynamics simulations. Comput. Phys. Commun. **184**(2), 374–380 (2013). https://doi.org/10.1016/j.cpc.2012.09.022
25. Li, X.S., et al.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2002). https://doi.org/10.1145/567806.567808, http://doi.acm.org/10.1145/567806.567808
26. Li, Y., Wang, W., Bai, H., Gong, R., Dong, X., Yu, F.: Efficient bitwidth search for practical mixed precision neural network. arXiv preprint arXiv:2003.07577 (2020)
27. Lima, J.V.F., Freytag, G., Pinto, V.G., Schepke, C., Navaux, P.O.A.: A dynamic task-based D3Q19 lattice-Boltzmann method for heterogeneous architectures. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 108–115, February 2019. https://doi.org/10.1109/EMPDP.2019.8671583
28. Markidis, S., Chien, S.W.D., Laure, E., Peng, I.B., Vetter, J.S.: NVIDIA tensor core programmability, performance precision. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 522–531, May 2018. https://doi.org/10.1109/IPDPSW.2018.00091
29. Medhat, R., Lam, M.O., Rountree, B.L., Bonakdarpour, B., Fischmeister, S.: Managing the performance/error tradeoff of floating-point intensive applications. ACM Trans. Embed. Comput. Syst. **16**(5s), 184:1–184:19, October 2017. https://doi.org/10.1145/3126519, http://doi.acm.org/10.1145/3126519
30. Menon, H., et al.: Adapt: algorithmic differentiation applied to floating-point precision tuning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, pp. 48:1–48:13. IEEE Press, Piscataway, NJ, USA (2018). https://doi.org/10.1109/SC.2018.00051
31. Micikevicius, P., et al.: Mixed precision training, pp. 1–12 (2017). http://arxiv.org/abs/1710.03740
32. Minhas, U.I., Bayliss, S., Constantinides, G.A.: GPU vs FPGA: a comparative analysis for non-standard precision. In: Goehringer, D., Santambrogio, M.D., Cardoso, J.M.P., Bertels, K. (eds.) Reconfigurable Computing: Architectures, Tools, and Applications, pp. 298–305. Springer, Cham (2014)
33. Mittal, S.: A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 1–33 (2016). https://doi.org/10.1145/2893356
34. Qi, X., Liu, C., Schuckers, S.: IoT edge device based key frame extraction for face in video recognition. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 641–644, May 2018. https://doi.org/10.1109/CCGRID.2018.00087
35. Raihan, M.A., Goli, N., Aamodt, T.M.: Modeling deep learning accelerator enabled GPUs. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 79–92, March 2019. https://doi.org/10.1109/ISPASS.2019.00016
36. Ramirez-Gargallo, G., Garcia-Gasulla, M., Mantovani, F.: TensorFlow on state-of-the-art HPC clusters: a machine learning use case. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 526–533, May 2019. https://doi.org/10.1109/CCGRID.2019.00067

37. Rojek, K.: Machine learning method for energy reduction by utilizing dynamic mixed precision on GPU-based supercomputers. Concurr. Comput. Pract. Experience **31**(6), e4644 (2019). https://doi.org/10.1002/cpe.4644, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4644, e4644 cpe.4644
38. Seznec, M., Gac, N., Ferrari, A., Orieux, F.: A study on convolution using half-precision floating-point numbers on GPU for radio astronomy deconvolution. In: 2018 IEEE International Workshop on Signal Processing Systems (SiPS), pp. 170–175, October 2018. https://doi.org/10.1109/SiPS.2018.8598342
39. Shi, G., Gottlieb, S., Torok, A., Kindratenko, V.: Design of MILC lattice QCD application for GPU clusters. In: 2011 IEEE International Parallel Distributed Processing Symposium, pp. 363–371, May 2011. https://doi.org/10.1109/IPDPS.2011.43
40. Vasquez, K., Venkatesha, Y., Bhattacharjee, A., Moitra, A., Panda, P.: Activation density based mixed-precision quantization for energy efficient neural networks. arXiv preprint arXiv:2101.04354 (2021)