Mixed precision applied on common mathematical procedures over GPU

Marcelo A. Sudo[1], **Álvaro L. Fazenda**[1], **Roberto P. Souto**[2]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (Unifesp)
São José dos Campos – SP – Brazil

[2]Laboratório Nacional de Computação Científica (LNCC)
Petrópolis – RJ – Brazil

{marcelo.sudo,alvaro.fazenda}@unifesp.br, rpsouto@lncc.br

Abstract. *Approximate Computing is a paradigm used by researchers as alternative to the diminishing of the evolution of hardware performance in the ongoing race for computational throughput in HPC. Precision reduction and mixed precision are the most studied among the existing techniques. In addition, some NVIDIA GPUs have Tensor Core architecture to speed up some classes of algorithms, such as matrix multiplication. This study aims to apply Approximate Computing techniques, like mixed precision, in matrix multiplication and stencil algorithms using OpenACC directives and cuTensor library to analyze performance gains versus accuracy losses. Results showed that it was possible to obtain a speedup of* $16.60\times$ *with an optimized matrix multiplication algorithm present in the matmul intrinsic function using 16-bit floating-point data with Tensor Core, compared to a naive version using 64-bit floating-point. For this same case, accuracy loss went from* 10^{-26} *up to* 10^{-1}*, approximately. For the stencil algorithm, it was possible to obtain a gain of* $1.60\times$ *by only reducing variables precision from 64-bit to 16-bit floating-point, with accuracy loss from 0 to* 10^{-9}*, for 300 iterations.*

1. Introduction

Approximate Computing (AC) is emerging as a promising paradigm to increase peak performance, once HPC programmers live an incessant search for computing techniques to increase computational throughput, and hardware isn't evolving at the same pace as in the past. Its principles preach that although performing the most possible exact computations at scale requires a high amount of computational resources, allowing certain approximations or occasional violations of numerical consistency can provide significant gains in efficiency. [Parasyris et al. 2020] states that Approximate Computing exploits the gap between the level of accuracy required by the applications and that provided by the system and has the potential to benefit a wide range of applications, such as scientific computing and machine learning. [Mittal 2015] stands that AC is based on the intuitive observation that while performing the most possible exact computation or maintaining peak-level service requires a high amount of resources, allowing selective approximation or occasional violation of the specification can provide gains in efficiency.

Among all AC methods, one of the most explored is floating-point precision reduction, or in a broader view, mixed precision. Modern computer architectures sup-

port multiple levels of precision for floating-point computations to provide trade-offs between accuracy and performance. Several recent studies, such as [Fogerty et al. 2017] and [Parasyris et al. 2020], have demonstrated the use of mixed precision, which means using multiple levels of precision, to increase significantly the performance of scientific applications. With accelerators supporting several levels of floating-point precision, such as half, single, and double precision in NVIDIA GPUs, and with higher peak performance in lower precision in these accelerators, this technique has become a promising approach to boost performance, especially using Tensor Cores [Parasyris et al. 2020].

Considering the strategies for approximation, variables and operations can be approximated using a variety of them, such as reducing their precision, skipping tasks, memory accesses or some iterations of a loop, operating on inexact hardware, etc. And one of the most exploited techniques is precision scaling, which is known for changing the precision (bit-width) of input or intermediate operands to reduce storage/computing requirements [Mittal 2015]. Another variant of this technique, defined by [Agrawal et al. 2016], is reduced precision which is a technique that represents variables and data structures in a program with fewer bits (compared with regular integer and floating point numbers). This allows users to utilize less expensive and more energy-efficient hardware to perform the reduced precision computation using the arithmetic logic unit. A Systematic Review analyzed by the authors [Sudo and Fazenda 2020] pointed out that "precision scaling" or "mixed precision" is mentioned by almost half of the papers selected when filtered by software techniques from AC.

To complement AC, but in the hardware area, the technology seems to be in evolution, such as the NVIDIA GPUs which were launched with Tensor Cores some years ago.

The objective of this study is to apply mixed precision as AC technique in basic algorithms such as matrix multiplication and stencil in FORTRAN code by using OpenACC directives and cuTensor library, analyzing possible performance gains versus its accuracy losses, and looking for opportunities to explore the power of Tensor Core technology.

The contribution to scientific community will be to show that it is possible to have some performance gains, against insignificant losses in accuracy when reducing the precision of floating point variables, in different mathematical numeric models. The motivation of the study is the fact that in the future we will carry out this same analysis in a Weather Forecasting System called BRAMS, developed in this same programming language, in a way that impacts the original code as little as possible, hence the use of OpenACC.

This paper follows with Related Work in subsection 1.1, and then the Methodology in Chapter 2, with a brief explanation of Tensor Core in subsection 2.1 and in sequence the Experimental Setup in subsection 2.2, following by the Results in Chapter 3 and finally the Conclusion in Chapter 4.

1.1. Related Work

Considering the state-of-the-art in this field of study, although we could find many papers mentioning Approximate Computing, our focus was the software techniques, specifically the ones focused on precision scaling and its variations, and in this scope, the ones that analyzed performance also, as most of them studied only the energy reduction. [Fogerty et al. 2017] analyzed approximate computing in some relevant mini-

applications. They intended to see the effects that reducing precision has on the power consumption and cost of the computation as well as the validity and correctness of the computational solution. The models were CLAMR, which is a mini-app that simulates fluid motion using the Shallow Water equations, and the Spectral Element Libraries in Fortran (SELF) which is a set of Fortran modules that define data structures and procedures that facilitate rapid implementation of Spectral Element Methods. The mini-apps were tested on both Intel processors and NVIDIA GPUs (NVIDIA GPU Tesla K40m, NVIDIA Quadro K6000, NVIDIA Tesla P100 SXM2-16GB, and GeForce GTX TITAN X). In the results, considering only GPUs, they presented a speedup gain from half-precision (16 bits) compared to double precision (64 bits) as for CLAMR 261% in Tesla K40m, 252% in Quadro K6000, and 453% in GTX TITAN X, and for SELF a gain of 34% in Tesla K40m, 31% in Quadro K6000, 28% in Tesla P100 and 309% in GTX TITAN X.

[Matoussi et al. 2019] proposed an analytical approach to study the impact of floating point precision variation on the square root operation, in terms of computational accuracy and performance gain. They estimated the round-off error resulting from reduced precision and also inspected the Newton-Raphson algorithm used to approximate the square root to bound the error caused by algorithmic deviation. Their case study was the K-means algorithm to reduce its energy footprint. The experimental results showed that energy savings could be achieved without penalizing the quality of the output (e.g., up to 41.87% of energy gain for output quality, measured using structural similarity, within a range of [0.95,1]).

[Koliogeorgi et al. 2019] proposed a highly optimized approximate SVM FPGA accelerator, utilizing arrhythmia detection in ECG signals as a case study. In methodology, they applied two algorithmic approximation techniques, i.e., precision scaling and loop perforation. They used fixed point representation for each data type and examined varying precision for the decimal part that ranges from 12 up to 22 bits. In the results, considering only the precision scaling point of view, it delivers a configuration with 4× speedup and 97.32% accuracy.

[Parravicini et al. 2021] proposed an implementation of Coordinate Format (COO) sparse matrix-vector multiplication, and studied its effectiveness when applied to the Personalized PageRank algorithm. Their implementation in FPGA achieved speedups up to $6.8\times$, when reducing to 20 bits, over a reference multi-threaded CPU implementation on 8 different data-sets, while preserving the numerical fidelity of the results, and reaching up to $42.0\times$ higher energy efficiency.

Our study, compared to all these, intended to modify the minimum possible the original code, and only via software techniques, so the reduction precision was performed only in the variable types and in the algorithm core, i.e. comparing double, float, and half precision in floating point numbers, sometimes also using lightly different instructions codes, and our purpose was to verify the speedup gain with these variations versus the precision loss.

2. Methodology

The study consists of the analysis of two algorithms: matrix multiplication, used often employed for deep learning methods, and stencil computation [Sloot et al. 2003], commonly used in scientific computing simulation. The first was intentionally chosen to analyze

the gains from using Tensor Core, since it is a specific technology for matrix multiplication, while the other was chosen because BRAMS contains many numerical processing functions that are based on stencil. Both use OpenACC directives to run on the GPU.

Considering the Matrix Multiplication, a naive source-code, an optimized version, and a *matmul* algorithm were analyzed. The naive program is the simplest version of the algorithm, based on three nested loops. The optimized version include some OpenACC directives aimed at improving performance, such as *parallel loop* and *collapse*, so it was expected that there would be a certain gain over the original algorithm. And finally, the *matmul* version consisted of calling the intrinsic function of the same name. In addition, some other variations were tested, such as the Kahan Algorithm, a variable with a higher precision accumulator, and also a program with calls to intrinsic functions *float2half* and *half2float* to perform the operations with greater precision. The Kahan summation algorithm, also known as compensated summation, significantly reduces the numerical error in the total obtained by adding a sequence of finite-precision floating-point numbers, compared to the obvious approach. This is done by keeping a separate running compensation (a variable to accumulate small errors), in effect extending the precision of the sum by the precision of the compensation variable [Higham 2002]. It was selected because it minimizes the accuracy loss when working with floating-point equations. The pseudo-code for it is described in algorithm 1.

Algorithm 1 Kahan Summation Algorithm. Source: [Higham 2002]

1: $sum \leftarrow 0.0$ ▷ Accumulator.
2: $c \leftarrow 0.0$ ▷ Compensation for lost low-order bits.
3: **for** $i = 1, 2, \ldots, input.length$ **do**
4: $y \leftarrow input[i] - c$ ▷ y gets the greatest part of the number.
5: $t \leftarrow sum + y$ ▷ t accumulates the rounding values.
6: $c \leftarrow (t - sum) - y$ ▷ (t - sum) cancels the high-order part of y; subtracting y recovers negative (low part of y)
7: $sum \leftarrow t$
8: **end for** ▷ Next time around, the lost low part will be added to y in a fresh attempt.

Regarding the second algorithm, for stencil calculation, the measurements were performed with 100, 300, and 500 iterations. These values were chosen because it is necessary 248 iterations to reach a steady state with a 10^{-3} precision, regarding the maximum difference between the same element position of two consecutive matrices. Interactive stencil loops perform a sequence of sweeps (called timesteps) through a given array. Generally, this is a 2- or 3-dimensional regular grid. The elements of the arrays are often referred to as cells. In each timestep, all array elements are updated. Using neighboring array elements in a fixed pattern (the stencil), each cell's new value is computed. In most cases, boundary values are left unchanged, but in some cases, those need to be adjusted during the computation as well. Since the stencil is the same for each element, the pattern of data accesses is repeated. The formula is as follow: $d(i, j) = alpha * (a(i, j - 1) + a(i, j + 1) + a(i - 1, j) + a(i + 1, j))$, where d is the resultant matrix, a is the input matrix, i and j corresponds to the position of the element in the matrix, and $alpha$ is a constant based on the number of elements considered, in this case 0.25 as there are 4 elements in the equation. The main point in stencil is that the

elements from the input are the neighbors of the resultant, from the 4 borders of it.

In both benchmarks, 5120 x 5120 matrices were used with the same precision reduction tests performed, comparing FP64 with FP32 and FP16. An evaluation using FP32 as a reference was also implemented, comparing only with FP16. Furthermore, all of them have been compared with Kahan's Algorithms, accumulator variables version in higher precision and intrinsic functions version (intrinsic numerical data type conversion functions *float2half* and *half2float*), with FP16. The optimized version was postponed for future work. There is no intrinsic function specific to performing this operation and Kahan Algorithm did not show any increase in accuracy.

Performance and accuracy analyses were performed for both algorithms. For matrix multiplication, the performance measurements consider the time spent to execute a sequence of ten times the same operation for each instance, resulting in a final matrix consisting of the sum of all executions. For the stencil benchmark, the performance and accuracy measures consider three different moments in interactive advance in time, for 100, 300, and 500 interactions. The total elapsed time measured for both benchmarks comprehends the data transfers between the host/device and the kernel execution. The data transfer phase between host and device and vice-versa was executed once for both algorithms, by a specific OpenACC directive before the main loop began. The specific data transfer elapsed time for matrix multiplication varies from 11%, related to total time, for *matmul* intrinsic function to 0.5% for Naive or Optimized version, considering all precision variations (FP64, FP32, and FP16). In the Stencil tests, data transfers vary from 19.6% for only 100 interactions to 4% for 500 interactions, related to the total elapsed time. The time spent for ten matrix multiplications in sequence and hundreds of executions in the Stencil algorithm dominates the total execution time. All tests execute six executions discarding the first one, since the execution time was generally significantly greater, probably due to GPU spin-up. The resulting time considers the average of the remaining five instances.

The initial data to fulfill the arrays in matrix multiplication uses a constant seed in a pseudo-random number generator, guaranteeing identical initial data for all instances. This procedure was done on the CPU, which transfers data to the GPU before the first kernel execution. The initial array in the Stencil benchmark is a matrix full of zeros and a left border initialized with the value. The values in the array's boundaries remain constant during all executions.

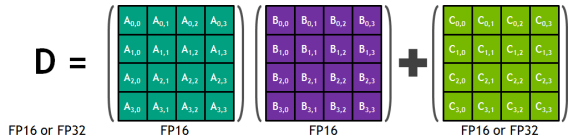
The performance metrics consist of elapsed time, percentage of gain, and speedup, using FP64 or FP32 as a reference state. The accuracy metrics use Mean Square Error (MSE) and the maximum difference between the precision target and a reference state considering all grid points.

2.1. Tensor Core Technology

In 2017 NVIDIA launched its GPU with Tensor Core technology performing small-size matrix multiplication in a single processor clock. The procedure uses FP16 or FP32 matrices with temporary reduction variable in greater precision, as seen in Figure 1, which means the multiplication is under FP16 and the accumulator may be on FP16 or FP32, which determines the resultant matrix [Appleyard and Yokim 2022].

The NVIDIA Fortran compiler supports Tensor Cores with NVIDIA's Volta V100

Figure 1. Tensor Core 4x4x4 matrix multiply. Source: [Appleyard and Yokim 2022]



and more recent GPUs (Turing, Ampere, and so on). Enabling scientific programmers using Fortran to take advantage of FP16 matrix operations accelerated by Tensor Cores [Leback 2022b]. This was considered an advance in technology, since Tensor Cores offer substantial performance gains over typical CUDA GPU core programming on Tesla V100 GPUs for certain classes of matrix operations running at FP16 (half-precision) [Leback 2022b].

The nvfortran automatically generates calls to tuned math libraries promoting portability by mapping Fortran statements to the functions available in the NVIDIA cuTENSOR library, providing tensor contraction, reduction, and element-wise operations [Leback 2022a].

2.2. Experimental Setup

The hardware used for the experiment is an NVIDIA V100 graphics card hosted by the National Scientific Computing Laboratory (LNCC) which is a Brazilian scientific research and technological development institution of the Ministry of Science, Technology and Innovation and Communications (MCTIC) specialized in scientific computing, located in Petropolis, Rio de Janeiro. The compiler used was NVIDIA Fortran Compiler v.20.11-0 with directives: $nvfortran - acc - cuda - cudalib[file].f90$. The source-code is available at: `https://github.com/marcelosudo/matmul`.

Although LNCC is shared with many researchers from all over the country, the executions in the operational environment force an exclusive running experiment, avoiding interferences in the measurements. All six running instances were defined in the same job file, so they were executed in sequence, with no interval between them. As mentioned before, the first execution was discarded due to probable GPU spin-up.

3. Results

Tables 1 and 2 show the computational performance for the matrix multiplication, considering a program using precision FP64 executing on a CPU as a reference state. The metrics used for performance measurements are the elapsed time and standard deviation (std) in seconds, the percentage of performance gain, and the Speedup. A performance gain greater than 40% was obtained by just decreasing the floating-point precision to FP32 and FP16, reaching $1.73\times$ and $1.97\times$ speedups, respectively. Using an optimized version with the same reference state achieved a speedup of $2.04\times$ for FP16, but the most impressive was the call to the intrinsic *matmul* function, where even in FP64 a speedup of $10.49\times$ was achieved, and in FP16 the speedup is $16.60\times$, demonstrating the power of the Tensor Core. When analyzing the performance with FP32 as a reference state, the speedup was 8.73 for the algorithm using FP16 *matmul*. The other algorithms evaluated: Kahan, Accumulator, and Intrinsic, did not obtain significant performance gains.

algorithm	precision	time $\pm$ std (s)	gain (%)	speedup
naive	FP64	$\mathbf{13.42} \pm 0.10$	0.00%	1.00
	FP32	7.75 ± 0.09	42.29%	1.73
	FP16	6.82 ± 0.14	49.21%	1.97
optimized	FP64	12.10 ± 0.01	9.84%	1.11
	FP32	7.05 ± 0.18	47.49%	1.90
	FP16	6.56 ± 0.03	51.09%	2.04
matmul	FP64	1.28 ± 0.13	90.47%	10.49
	FP32	1.01 ± 0.06	92.50%	13.33
	FP16	0.81 ± 0.01	93.98%	16.60
Kahan	FP16	6.63 ± 0.03	50.59%	2.02
accumulator	FP16	6.59 ± 0.00	50.90%	2.04
intrinsic	FP16	6.45 ± 0.00	51.95%	2.08

Table 1. Matrix Multiplication (5120 x 5120), reference FP64, PERFORMANCE.

algorithm	precision	time $\pm$ std (s)	gain (%)	speedup
naive	FP32	$\mathbf{7.62} \pm 0.08$	0.00%	1.00
	FP16	6.75 ± 0.13	11.44%	1.13
optimized	FP32	6.96 ± 0.01	8.65%	1.09
	FP16	6.34 ± 0.01	16.71%	1.20
matmul	FP32	1.00 ± 0.07	86.92%	7.64
	FP16	0.87 ± 0.19	88.54%	8.73
Kahan	FP16	6.47 ± 0.23	15.01%	1.18
accumulator	FP16	6.27 ± 0.11	17.71%	1.22
intrinsic	FP16	6.10 ± 0.07	19.93%	1.25

Table 2. Matrix Multiplication (5120 x 5120), reference FP32, PERFORMANCE

algorithm	precision	MSE	max diff
naive	FP64	$1.72\text{E}-26$	$2.00\text{E}-12$
	FP32	$2.00\text{E}-06$	$7.07\text{E}-03$
	FP16	$2.29\text{E}+04$	$1.98\text{E}+02$
optimized	FP64	$1.72\text{E}-26$	$2.00\text{E}-12$
	FP32	$2.00\text{E}-06$	$7.07\text{E}-03$
	FP16	$2.29\text{E}+04$	$1.98\text{E}+02$
matmul	FP64	$1.72\text{E}-26$	$2.00\text{E}-12$
	FP32	$4.95\text{E}-09$	$1.00\text{E}-03$
	FP16	$1.04\text{E}-01$	$6.74\text{E}-01$
Kahan	FP16	$8.34\text{E}-02$	$5.37\text{E}-01$
accumulator	FP16	$8.34\text{E}-02$	$5.34\text{E}-01$
intrinsic	FP16	$8.34\text{E}-02$	$5.28\text{E}-01$

Table 3. Matrix Multiplication (5120 x 5120), reference FP64, ACCURACY.

algorithm	precision	MSE	max diff
naive	FP32	$4.95\text{E}-09$	$1.00\text{E}-03$
	FP16	$2.10\text{E}+04$	$1.98\text{E}+02$
optimized	FP32	$4.95\text{E}-09$	$1.00\text{E}-03$
	FP16	$2.10\text{E}+04$	$1.98\text{E}+02$
matmul	FP32	$4.95\text{E}-09$	$1.00\text{E}-03$
	FP16	$9.71\text{E}-02$	$6.76\text{E}-01$
Kahan	FP16	$8.07\text{E}-02$	$5.33\text{E}-01$
accumulator	FP16	$8.07\text{E}-02$	$5.36\text{E}-01$
intrinsic	FP16	$8.07\text{E}-02$	$5.27\text{E}-01$

Table 4. Matrix Multiplication (5120 x 5120), reference FP32, ACCURACY.

The accuracy analysis is shown in Tables 3 and 4 through the MSE shows an error in the order of 10^{-26} for FP64 to 10^{4} in FP16, which would be an undesirable result, but *matmul* function still kept an acceptable precision loss on the order of 10^{-1}. Improving the algorithm by using the Kahan method, or an accumulator as extra variables with higher precision, or using intrinsic function to convert numerical data-types, it was possible to reach 10^{-2} in MSE. Similar behavior was obtained in FP32 reference state, and better than *matmul* version.

Regarding the stencil, as seen in Tables 5 and 6, we did not obtain comparable gains related to matrix multiplication. A speedup of $1.57\times$ was obtained by using an FP16 code with an FP64 reference state. The same behavior was maintained in the various analyzed iterations. With FP32 reference state, the speedup reach $1.30\times$ to $1.34\times$ for the different variations of analyzed algorithms, when the precision was reduced to FP16.

In contrast to the smaller speedups, as seen in Tables 7 and 8, the accuracy loss was also smaller, starting with 10^{-19} in FP32 and reaching up to 10^{-10} in FP16. In this case, there was a small increase in the loss of accuracy with increasing iterations, reaching 10^{-8} in 500 iterations. The same behavior was observed for FP32 reference state.

iterations	precision	time $\pm$ std (s)	gain (%)	speedup
100	FP64	84.02 ± 0.07	0.00%	1.00
	FP32	69.96 ± 0.01	16.73%	1.20
	FP16	53.68 ± 0.01	36.11%	1.57
300	FP64	256.65 ± 6.24	0.00%	1.00
	FP32	212.06 ± 2.61	17.38%	1.21
	FP16	160.74 ± 1.06	37.37%	1.60
500	FP64	419.97 ± 4.58	0.00%	1.00
	FP32	360.67 ± 0.60	14.12%	1.16
	FP16	268.05 ± 1.83	36.17%	1.57

Table 5. Stencil (5120 x 5120), 100, 300 and 500 iterations, reference FP64, PERFORMANCE.

iterations	algorithm	precision	time $\pm$ std (s)	gain (%)	speedup
100	naive	FP32	71.94 ± 0.67	0.00%	1.00
		FP16	54.67 ± 0.26	24.01%	1.32
	Kahan	FP16	53.65 ± 0.03	25.43%	1.34
	accumulator	FP16	53.86 ± 0.36	25.14%	1.34
	intrinsic	FP16	53.77 ± 0.26	25.26%	1.34
300	naive	FP32	210.83 ± 1.94	0.00%	1.00
		FP16	160.80 ± 0.98	23.73%	1.31
	Kahan	FP16	161.10 ± 0.99	23.59%	1.31
	accumulator	FP16	161.05 ± 1.49	23.61%	1.31
	intrinsic	FP16	161.47 ± 1.17	23.41%	1.31
500	naive	FP32	355.20 ± 6.05	0.00%	1.00
		FP16	272.71 ± 1.22	23.22%	1.30
	Kahan	FP16	272.45 ± 1.69	23.30%	1.30
	accumulator	FP16	268.27 ± 1.65	24.47%	1.32
	intrinsic	FP16	272.71 ± 0.48	23.22%	1.30

Table 6. Stencil (5120 x 5120), 100, 300 and 500 iterations, reference FP32, PERFORMANCE.

iterations	precision	MSE	max diff
100	FP64	$0.00\mathrm{E}+00$	$0.00\mathrm{E}+00$
	FP32	$3.11\mathrm{E}-19$	$1.44\mathrm{E}-07$
	FP16	$1.68\mathrm{E}-10$	$5.70\mathrm{E}-04$
300	FP64	$0.00\mathrm{E}+00$	$0.00\mathrm{E}+00$
	FP32	$4.60\mathrm{E}-18$	$1.38\mathrm{E}-07$
	FP16	$3.77\mathrm{E}-09$	$4.41\mathrm{E}-03$
500	FP64	$0.00\mathrm{E}+00$	$0.00\mathrm{E}+00$
	FP32	$4.19\mathrm{E}-18$	$1.53\mathrm{E}-07$
	FP16	$6.68\mathrm{E}-08$	$7.28\mathrm{E}-03$

Table 7. Stencil (5120 x 5120), 100, 300 and 500 iterations, reference FP64, ACCURACY.

iterations	algorithm	precision	MSE	max diff
100	naive	FP32	$0.00\mathrm{E}+00$	$0.00\mathrm{E}+00$
		FP16	$1.68\mathrm{E}-10$	$5.70\mathrm{E}-04$
	Kahan	FP16	$3.10\mathrm{E}-10$	$1.23\mathrm{E}-03$
	accumulator	FP16	$1.68\mathrm{E}-10$	$5.70\mathrm{E}-04$
	intrinsic	FP16	$2.29\mathrm{E}-10$	$6.21\mathrm{E}-04$
300	naive	FP32	$0.00\mathrm{E}+00$	$0.00\mathrm{E}+00$
		FP16	$3.77\mathrm{E}-09$	$4.41\mathrm{E}-03$
	Kahan	FP16	$4.85\mathrm{E}-09$	$7.29\mathrm{E}-03$
	accumulator	FP16	$3.77\mathrm{E}-09$	$4.41\mathrm{E}-03$
	intrinsic	FP16	$3.58\mathrm{E}-09$	$2.99\mathrm{E}-03$
500	naive	FP32	$0.00\mathrm{E}+00$	$0.00\mathrm{E}+00$
		FP16	$6.69\mathrm{E}-08$	$7.28\mathrm{E}-03$
	Kahan	FP16	$1.81\mathrm{E}-07$	$1.67\mathrm{E}-02$
	accumulator	FP16	$6.69\mathrm{E}-08$	$7.28\mathrm{E}-03$
	intrinsic	FP16	$6.65\mathrm{E}-08$	$6.63\mathrm{E}-03$

Table 8. Stencil (5120 x 5120), 100, 300 and 500 iterations, reference FP32, ACCURACY.

4. Conclusion

In this study, applications of AC techniques such as reduced precision and mixed precision were analyzed to verify the trade-off between performance gains and accuracy losses. We compared results from changing FP64, FP32, and FP16 in floating point operations in matrix multiplication and stencil algorithm. The results showed an exceptional speedup gain of $16.60\times$ in matrix multiplication when comparing a naive version with a *matmul* intrinsic function call with the cuTensor library, which invokes Tensor Core GPU hardware. Also, a speedup of $1.60\times$ was obtained in stencil simply by reducing the precision from FP64 to FP16. In both cases, the losses were acceptable, being from 10^{-26} to 10^{-1} in matrix multiplication and 0 to 10^{-9} in the stencil benchmark.

For future works, we aim to study the energy savings in these algorithms, the effects of different matrix dimensions in performance, and also investigate the power of Tensor Core in other applications, such as in convolution method. There are plans to investigate more mathematical methods to mitigate round-off errors when using reduced precision.

Acknowledgements

The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper. URL: http://sdumont.lncc.br.

This work was also partially financed by grant #2019/26702-8, São Paulo Research Foundation (FAPESP).

References

Agrawal, A., Choi, J., Gopalakrishnan, K., Gupta, S., Nair, R., Oh, J., Prener, D. A., Shukla, S., Srinivasan, V., and Sura, Z. (2016). Approximate computing: Challenges and opportunities.

Appleyard, J. and Yokim, S. ((accessed July 07, 2022)). Programming tensor cores in cuda 9. `https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/`.

Fogerty, S., Bishnu, S., Zamora, Y., Monroe, L., Poole, S., Lam, M., Schoonover, J., and Robey, R. (2017). Thoughtful precision in mini-apps. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 858–865.

Higham, N. (2002). Accuracy and stability of numerical algorithms (2 ed). In SIAM, editor, *Accuracy and Stability of Numerical Algorithms (2 ed)*, page 110–123. Society for Industrial and Applied Mathematics Philadelphia.

Koliogeorgi, K., Zervakis, G., Anagnostos, D., Zompakis, N., and Siozios, K. (2019). Optimizing svm classifier through approximate and high level synthesis techniques. In *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4.

Leback, B. (2019 (accessed June 26, 2022)b). Tensor core programming using cuda fortran. `https://developer.nvidia.com/blog/tensor-core-programming-using-cuda-fortran/`.

Leback, B. (2020 (accessed June 26, 2022)a). Bringing tensor cores to standard fortran. `https://developer.nvidia.com/blog/bringing-tensor-cores-to-standard-fortran/`.

Matoussi, O., Durand, Y., Sentieys, O., and Molnos, A. (2019). Error analysis of the square root operation for the purpose of precision tuning: A case study on k-means. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 75–82.

Mittal, S. (2015). A survey of techniques for approximate computing. DOI: 10.1145/2893356.

Parasyris, K., Laguna, I., Menon, H., Schordan, M., Osei-Kuffuor, D., Georgakoudis, G., Lam, M. O., and Vanderbruggen, T. (2020). Hpc-mixpbench: An hpc benchmark suite for mixed-precision analysis. DOI: 10.1109/IISWC50251.2020.00012.

Parravicini, A., Sgherzi, F., and Santambrogio, M. D. (2021). A reduced-precision streaming spmv architecture for personalized pagerank on fpga. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 378–383.

Sloot, P. M. A., Tan, C. J. K., Dongarra, J. J., and Hoekstra, A. G., editors (2003). *Computational science - ICCS 2002*. Lecture Notes in Computer Science. Springer Berlin, Berlin, Germany, 2002 edition.

Sudo, M. and Fazenda, (2020). A review on approximate computing applied to meteorological forecast models using software-based techniques.