Optimization of asynchronous logging kernels for a GPU accelerated CFD solver

Paul Zehner[0000−0002−4811−0079] and Atsushi Hashimoto[0000−0003−4428−0406]

JAXA, 7-44-1, Jindiaiji Higashi-machi, Choufu, 182-0012 Tokyo, Japan
zehner.paul@jaxa.jp, hashimoto.atsushi@jaxa.jp

Abstract. Thanks to their large number of threads, GPUs allow massive parallelization, hence good performance for numerical simulations, but also make asynchronous execution more common. Kernels that do not actively take part in a computation can be executed asynchronously in the background, in the aim to saturate the GPU threads. We optimized this asynchronous execution by using mixed precision for such kernels. Implemented on the FaSTAR solver and tested on the NASA CRM case, asynchronous execution gave a speedup of 15 % to 27 % for a maximum memory overhead of 4.5 % to 9 %.

Keywords: asynchronous · CFD · GPU · mixed precision · OpenACC.

1 Introduction

General-Purpose computing on GPU (GPGPU) allows massive parallelization thanks to its large number of threads, and is considered a promising path to exascale computing (10^{18} FLOP s^{-1}) [18, 22]. Its use in Computational Fluid Dynamics (CFD) allows to perform more complex, more precise, and more detailed simulations, but requires either to adapt existing codes or to create new ones [3], with the help of specific techniques such as CUDA, OpenCL or OpenACC [17]. CFD codes can not only benefit from the computational power of Graphical Processing Units (GPUs), but also from their asynchronous execution capabilities.

The asynchronous execution of kernels stems from the capacity of the Streaming Multiprocessor (SM) to execute different instructions on its different warps [8, 15]. The programmer specifies an asynchronous execution by using *streams* for CUDA (with a fourth argument to the triple chevron `<<<>>>` syntax), or by using the `async` clause for OpenACC. Uncommon among traditional High Performance Computing (HPC) paradigms, asynchronous execution offers interesting uses.

Its immediate benefit is to allow kernels execution and memory transfers to overlap. Multi-GPU performance is improved by this approach, as it allows to hide network transfers. Micikevicius [13], on a CUDA-accelerated three-dimensional structured finite difference code, used asynchronous execution along with non-blocking Message Passing Interface (MPI) calls. He proposed a two-step algorithm: for each timestep, the first step consists in computing only the boundary cells; the

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

second step consists in computing all the other cells and, at the same time, transferring the boundary cells. McCall [12] accelerated an artificial compressibility Navier–Stokes structured solver using OpenACC. The same approach was used, but reversely: the first step consists in calculating the artificial compressibility and numerical damping terms on the interior cells and, at the same time, transferring the boundary cells with MPI; the second step consists in calculating the remaining cells. McCall considered that this overlapping technique improved weak scaling significantly. Shi et al. [20], accelerating an incompressible Navier–Stokes solver with CUDA, also used streams to overlap computations and non-blocking MPI communications. Choi et al. [2] used a more aggressive approach with an overdecomposition paradigm which uses Charm++ instead of MPI. Charm++ [11] is a C++ library which allows to decompose a problem in several tasks, named *chares*, that are executed asynchronously by processing elements and communicate with messages. In their paper, Choi et al. applied this technique to a test Jacobi solver, and obtained a weak scaling performance gain of 61 % to 65 % on 512 nodes, compared to a traditional MPI approach, and a strong scaling gain of 40 % on 512 nodes.

Another benefit resides in the concurrent execution of independent kernels; where each kernel operates on different arrays at the same time. Hart et al. [5], for the OpenACC acceleration of the Himeno benchmark on the Cray XK6 supercomputer, overlapped data transfers and computation, and executed computation and MPI data packing concurrently. Compared to synchronous GPU code, performances were increased by 5 % to 10 %. The authors predicted that a higher gain could be obtained for larger codes, as they considered Himeno simple. Searles et al. [19] accelerated the wavefront-based test program Minisweep with OpenACC and MPI, in order to improve the nuclear reactor modeling code Denovo. They proposed to parallelize the Koch–Baker–Alcouffe algorithm, which consists in different sweeps. They used asynchronous kernel execution to realize the 8 sweeps simultaneously, in order to saturate the GPU threads. The authors reported good performance.

Asynchronous execution also allows to run background tasks with low resources, while a main task is running synchronously, in the objective to saturate the GPU threads. The unstructured CFD solver FaSTAR, developed at Japan Aerospace eXploration Agency (JAXA) [6, 7, 10] and accelerated with OpenACC [28,29], uses this feature in order to run logging kernels asynchronously. These kernels are executed with a certain number of OpenACC gangs, and require memory duplication to avoid race conditions. This process allowed a speedup of 23 % to 35 %, but increased memory occupancy by up to 18 %. In this paper, we propose to optimize this process by reducing the memory overhead with lower precision storage of floating point arrays.

This paper is organized as follows. We introduce the FaSTAR solver, its current state of acceleration, and its asynchronous execution of logging kernels in section 2. The use of mixed precision to reduce the memory overhead of the asynchronous execution is presented in section 3. Then, we validate and check

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

the performance of this improvements on the NASA Common Research Model (NASA CRM) test case in section 4. To finish, we conclude in section 5.

2 Description of the FaSTAR solver

2.1 Solver introduction

The FaSTAR code solves the compressible Reynolds-Averaged Navier–Stokes (RANS) equations on unstructured meshes. It consists in 80,000 lines of Fortran code, has an Array of Structures memory layout, is parallelized with MPI, and uses Metis for domain decomposition. The Cuthill–McKee algorithm is used to reorder cells. The solver was designed to run a simulation of 10 million cells on 1000 Central Processing Unit (CPU) cores within 2 min [6].

2.2 Acceleration of the solver

The solver was partially accelerated with OpenACC in a previous work [29]. OpenACC was selected in favor of CUDA, as we wanted to keep CPU compatibility without having to duplicate the source code.

Parts of the code that were accelerated so far are the modified Harten-Lax-van Leer-Einfeldt (HLLEW) scheme [16], the Lower-Upper Symmetric Gauss–Seidel (LU-SGS) [27] and the Data-Parallel Lower-Upper Relaxation (DP-LUR) [26] implicit time integration methods, the GLSQ algorithm [21] (a hybrid method of Green-Gauss and Least-Square), the Hishida slope limiter (a Venkatakrishnan-like limiter that is complementary with difference of neighboring cell size) and the Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL) reconstruction method. During execution, all of the computation takes place on the GPU, and data movements between the CPU and the GPU memories are minimized.

2.3 Asynchronous execution of logging kernels

Some kernels only used for logging purpose can take a significant computation time, while their result is not used anywhere in the computation. Such kernels are run asynchronously in the background [28], and use just enough resources to saturate the GPU threads. In the case of FaSTAR, the kernel to log the right-hand side (RHS) values of the Navier–Stokes equation and the kernel to compute the L^2 norm of residuals are concerned by this optimization. These two kernels are called at different times during the iteration, respectively after the computation of the RHS term, and after the time integration.

The subroutine hosting such a kernel is separated in two parts: one computing the values, offloaded on the GPU, and another one writing the values (on disk or on screen), executed by the CPU. A module stores the value between the two subroutines:

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

```
1  module store
2    real :: value(n_value)
3  end module store
4
5  module compute
6  contains
7    subroutine compute_value(array)
8      use store, only: value
9      ...
10
11     !$acc kernels async(STREAM) &
12     !$acc          copyout(value) &
13     !$acc          copyin(...)
14     ! compute value from array
15     !$acc end kernels
16   end subroutine compute_value
17
18   subroutine write_value
19     use store, only: value
20
21     !$acc wait(STREAM)
22     print "('value(1) = ', g0)", value(1)
23     ...
24   end subroutine write_value
25 end module compute
```

As seen on line 11, the kernel in the compute part is executed asynchronously with the OpenACC `async` clause in a specific `STREAM`, while the write part is called as late as possible and, on line 21, waits for the asynchronous kernel to end with the `wait` directive. At the same time, other kernels are executed synchronously. The write part (for each logging subroutine in the case of FaSTAR) is called at the end of the iteration.

In order to avoid read race conditions as the main synchronous kernels may modify the memory, arrays used by the logging kernel must be duplicated. The following modifications are added to the code:

```
1  module store
2    ...
3    real, allocatable :: array_d(:)
4  end module store
5
6  module compute
7  contains
8    subroutine compute_value(array)
9      use store, only: array_d
10     ...
11
12     !$acc kernels
13     array_d(:) = array(:)
14     !$acc end kernels
```

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

```
15
16        ...
17      end subroutine compute_value
18      ...
19    end module compute
```

In subroutine `compute_value`, the memory duplication takes place before the asynchronous kernel. This is however a costly operation in term of memory. The complete process workflow for FaSTAR is sketched in fig. 1. The two asynchronous kernels use the same stream to not be concurrent to each other.

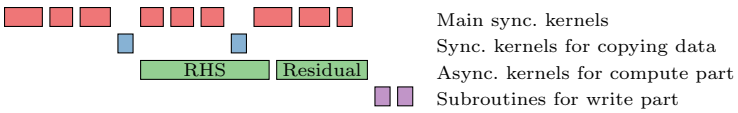


Fig. 1: Workflow of asynchronous execution of the two logging kernels.

Finding the optimal number of threads for the logging kernels is an optimization problem, as too few dedicated threads makes its execution time too long (hence increasing the simulation time), and too much slows down the synchronous kernels (hence increasing the simulation time again). The number of cells of the simulation as well as the hardware itself has an influence on this number of threads. Some assumptions are used to simplify the problem. We consider that the compute kernel represents enough operations to fill at least the threads of one SM, which translates in one gang in the OpenACC terminology of *gang* (SM), *worker* (Streaming Processor (SP)) and *vector* (thread). We consequently investigate the number of gangs, instead of the number of threads, which can be set with the OpenACC clause `num_gangs`. We also consider that each iteration is similar and takes the same amount of time to compute (with the same amount of resources). The number of gangs can either be set by the user, or can be automatically estimated with what we name a sloped descend gradient method:

$$n_g^{(n+1)} = n_g^{(n)} - \operatorname{sign} \frac{\Delta t^{(n)} - \Delta t^{(n-1)}}{n_g^{(n)} - n_g^{(n-1)} + \epsilon}, \tag{1}$$

where $n_g^{(n+1)}$, respectively $n_g^{(n)}$ and $n_g^{(n-1)}$, is the number of gangs for the next iteration (or group of iterations), respectively for the current iteration and the previous iteration, $\Delta t^{(n)}$, respectively $\Delta t^{(n-1)}$, is the duration of the current iteration, respectively of the previous iteration, and ϵ is a small value. We consider, as for many Fortran implementations, that $\operatorname{sign} 0 = 1$. This formulation is robust and increases the number of gangs by ± 1 for each evaluation, but has two drawbacks: the optimal value may be slow to reach, depending on the initial value to start the evaluation from, and the number of gangs can only fluctuate close to this optimum. Using the iteration time to compute the number of gangs makes

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

this algorithm vulnerable to external factors, such as network flutters in case of MPI parallelization, or input/output latency when writing intermediate results to disk. This problem is mitigated by taking a group of iterations in eq. (1), by default 100, instead of one, and by disabling the evaluation of the number of gangs for iterations when results are outputted to a file.

3 Implementation of mixed precision kernels

The asynchronous execution process presented in the previous section has the disadvantage to trade GPU memory for execution speed, while memory is limited on this hardware. FaSTAR uses double precision floating point arrays, which means by the Institute of Electrical and Electronics Engineers (IEEE) 754 norm [9] that each element in an array is stored on 64 bits (8 bytes). Such a precision is the standard in CFD, but is not required for logging purpose, where the order of magnitude is usually enough as logs are read by a human. Consequently, we propose to use a lower precision for the logging kernels, such as single precision (32 bits per element), and even half precision (16 bits per element). This can reduce the memory cost of the asynchronous execution process by respectively two and four. Even if mixed precision is commonly used in a context of performance gain [24], we only aim in this paper to reduce memory occupancy.

3.1 Single precision

We first implement the storage of duplicated arrays and the computation of the value using single precision. As the selection of precision has to be decided at compile time, we use preprocessor commands, which are only executed by the compiler before converting the source code in binary instructions. Since the Fortran standard does not specify a preprocessor, we use the traditional C preprocessor syntax, which is commonly available in compilers.

We use a set of two preprocessor macros, that are substituted by the compiler in the rest of the source code:

```
1 #if LOGGING_REDUCED_PRECISION == 4
2 #  define TYPE 4
3 #  define STORE(value) (real(value, TYPE))
4 #else
5 #  define TYPE 8
6 #  define STORE(value) (value)
7 #endif
```

The macro `TYPE` qualifies the kind of the floating point variables, and is used in the declaration of the duplicated arrays and in the compute and write subroutines, using the `kind` argument of type `real`. Line 2 defines it to single precision (for most environments), and line 5 to double precision. `STORE` explicitly converts variables to the desired precision using the `real` Fortran intrinsic function; it is used when duplicating the arrays. At line 3, it converts values to single precision, and at line 6 it is an invariance. Using mixed precision can be disabled completely by the `LOGGING_REDUCED_PRECISION` macro at line 1.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

3.2 Half precision

Half precision has several limitations that must be taken into consideration first. The Fortran standard does not explicitly define which precision is available for the programmer, and this choice is left to the compiler maker. Single and double precision are usually always implemented, but half precision is less common, hence not supported by all compilers. In this study, we chose to use the NVIDIA HPC Software Development Kit (NVIDIA HPC SDK) compiler, as it is the most common and mature OpenACC compiler available when writing this paper, and as it has support for half precision. Also, half precision has an exponent range limited to $\pm 10^4$, with any other value replaced by $\pm\infty$. This limitation may be unacceptable if the values to manipulate fall outside of this range. Another representation considered was *bfloat16* [25], which has a larger exponent range of $\pm 10^{38}$ while using the same amount of memory, but is not covered by the Fortran standard [1]. Another shortcoming of half precision is that not all operations may be implemented for this precision. Consequently, when using half precision, only the duplicated arrays are stored using this precision, whereas work on these arrays is done using single precision.

The set of preprocessor macros is extended with two other macros, `TYPE_DUPLICATED` and `UNSTORE`:

```
1  #if LOGGING_REDUCED_PRECISION == 2
2  #  define TYPE 4
3  #  define TYPE_DUPLICATED 2
4  #  define STORE(value) (real(value, TYPE_DUPLICATED))
5  #  define UNSTORE(value) (real(value, TYPE))
6  #if LOGGING_REDUCED_PRECISION == 4
7  #  define TYPE 4
8  #  define TYPE_DUPLICATED 4
9  #  define STORE(value) (real(value, TYPE))
10 #  define UNSTORE(value) (value)
11 #else
12 #  define TYPE 8
13 #  define TYPE_DUPLICATED 8
14 #  define STORE(value) (value)
15 #  define UNSTORE(value) (value)
16 #endif
```

The macro `TYPE` is now only used for the kind of arrays in the compute and write subroutines. At line 2 it is defined for single precision. `TYPE_DUPLICATED` is only used for the kind of the duplicated arrays. At line 3 it is defined for half precision, at lines 8 and 13 as the current `TYPE`. `UNSTORE` explicitly converts back duplicated variables to the desired precision, and is used in the compute part. At line 5, it converts values from half precision to single precision, at lines 10 and 15 it is an invariance. The `LOGGING_REDUCED_PRECISION` macro is used to select the desired precision.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

4 Validation and performance

4.1 Description of the cases

Now mixed precision is implemented, we test the validity and the performance of the modified solver on the NASA CRM test case [23]. This geometry aims to propose a research model of a commercial airplane, it has a wing and a cabin. We simulated a cruise flight at Mach number $M = 0.85$, angle of attack $\alpha = 2.5°$, and Reynolds number $Re = 5 \cdot 10^6$. The upstream temperature was $T_\infty = 100\,°\mathrm{F} \approx 310.93\,\mathrm{K}$ and the air specific heat ratio was $\gamma = 1.4$.

Table 1: Number of cells for the different meshes.

Mesh	Tetrahedron	Pyramid	Prism	Hexahedron	Total
Coarse	146,062	587,161	0	2,946,009	3,679,232
Medium	258,724	1,040,044	27,424	5,260,349	6,586,541
Fine	467,168	1,897,302	77,256	9,491,406	11,933,132
Extra-fine	1,633,430	6,554,906	147,004	38,410,538	46,745,878

We used a set of four meshes of increasing number of cells, as described in table 1, created with HexaGrid. Minimum cell size was 5 in, 3.5 in, 2 in and 1 in ($1.27 \cdot 10^{-1}$ m, $8.89 \cdot 10^{-2}$ m, $5.08 \cdot 10^{-2}$ m and $2.54 \cdot 10^{-2}$ m) for the coarse, medium, fine and extra-fine mesh respectively. Maximum cell size was 5 in ($1.27 \cdot 10^{-1}$ m), and the dimensionless wall distance was $y^+ = 1.00$.

Methods cited in section 2.1 were used; the DP-LUR implicit time integration method was used with 6 sub-iterations. We performed steady state simulations of 10,000 iterations, using local time stepping and setting a Courant–Friedrichs–Lewy (CFL) number $CFL = 50$. The solver was compiled using NVIDIA HPC SDK version 22.7, simulations were executed on JAXA Supercomputer System generation 3 (JSS3) [4] using NVIDIA V100 GPUs [14]. Cases were executed on single GPU and multi-GPU using 2 and 4 domains. The extra-fine case memory occupancy was too high for some executions: for single GPU and multi-GPU on 2 domains with double precision.

Compared to reference computations using synchronous logging kernels, we aim to analyze the speedup and the memory overhead of the different implementations. Two batches of computations were executed: one for which the number of gangs was automatically estimated by the algorithm presented in section 2.3, another one with a fixed number of gangs based on the outcome of the first batch.

4.2 Accuracy analysis

Before analyzing performance, we analyze the accuracy of the results for computations on the fine mesh on single GPU. We first made sure that solution files were similar with an absolute tolerance of 10^{-5} for the different simulations.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

Letting the algorithm estimate the number of gangs or setting it manually did not change the results.

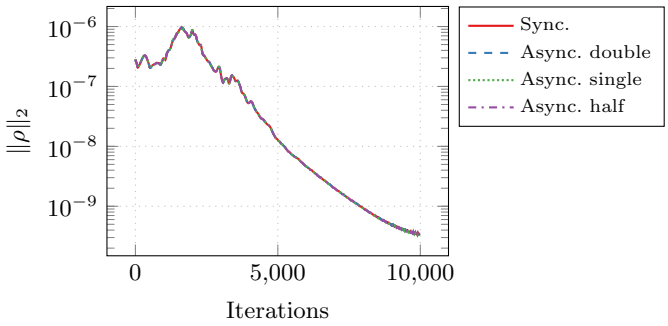


Fig. 2: Evolution of the L^2 norm of residual.

The evolution of the L^2 norm of the residual of density $\|\rho\|_2$ is displayed in fig. 2. Residual of computations using asynchronous execution with double, single, or half precision are compared with a reference synchronous computation. Results are indistinguishable. A closer analysis reveals that residual for single and half precision are a bit lower than reference results, which can be explained as they are both computed with single precision.

The evolution of the sum of the RHS terms, not displayed in this paper, was also indistinguishable from the results of the synchronous execution. However, some values were replaced by $+\infty$ in results of the asynchronous execution using half precision.

We conclude that using reduced precision when computing residual did not affect the quality of the solution. The quality of the logging files was almost identical when using single precision; on the other hand, half precision gave almost identical results in most cases, but some values could not be represented.

4.3 Performance analysis

We analyze now the performance of the new implementations, both in term of speedup and in term of memory overhead.

We define the speedup η as the ratio of wall clock times spent in the time integration loop, which excludes the initial time for loading the mesh and the final time for writing the solution:

$$\eta = \frac{t_{\text{sync}}}{t_{\text{async}}} \tag{2}$$

where t_{sync} is related to the reference synchronous execution, and t_{async} is related to the asynchronous execution of the different implementations. The speedup is displayed in fig. 3 as a function of the number of cells, when the number of

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

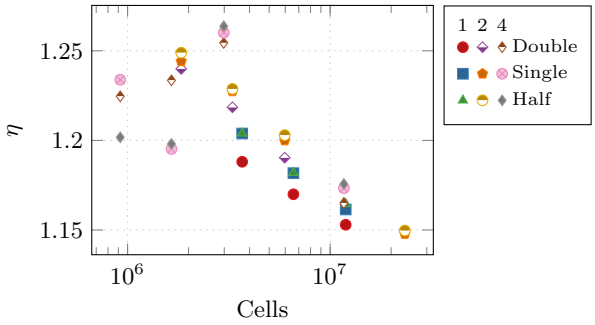
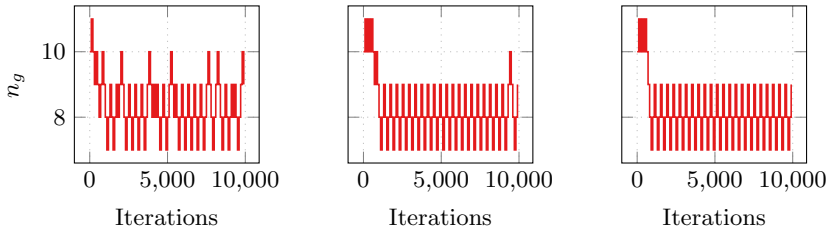


Fig. 3: Evolution of the speedup compared with synchronous execution.



(a) Double precision. (b) Single precision. (c) Half precision.

Fig. 4: Evolution of the number of gangs for asynchronous execution on the fine mesh on 1 domain.

gangs is automatically estimated. We represented speedups for 1 domain (single GPU), 2 and 4 domains (multi-GPU) on the same graph. For any precision, the overall tendency is that the speedup decreases as the number of cells increases. As the number of cells increases, there are less idle threads to saturate, and the asynchronous execution is hence less efficient. The speedup ranged from 1.15 to 1.21 for 1 domain, from 1.15 to 1.25 for 2 domains, and from 1.17 to 1.27 for 4 domains. Computations on 1 and 2 domains exhibited a similarly decreasing speedup, whereas computations on 4 domains exhibited a more chaotic trend with high variability, which can be explained by other factors contributing more to the speedup, such as network latency. Executions with single precision had a higher speedup compared with double precision by on average 1 % for 1 domain and 0.6 % for 2 domains. Half precision had a higher speedup by on average 1.1 % for 1 domain and 0.9 % for 2 domains. This can either be due to less time spent in duplicating the memory, or by more resources available for the main kernels. Using reduced precision had a limited influence on speedup.

When automatically estimated by the algorithm, the number of gangs stabilized close to a single value. Figure 4 displays the evolution of the number of gangs as a function of the number of iteration for the computations on the fine mesh on single GPU only. Despite being specific to this mesh, the evolution of

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

the number of gangs is representative of the other simulations. For all cases, the initial number of gangs was set to 10. The optimum value was 8 for the different precision values, meaning that the precision of the logging kernels did not have an impact on the estimation of the number of gangs. Compared with the maximum number of 84 gangs available on a V100 GPU, the number of gangs dedicated to the logging kernels remained reasonable: 9 %. For multi-GPU execution, the number of gangs was usually the same on the different devices. However, for the simulation of the coarse case on 4 domains using half precision, the number of gangs on one device converged to a different value.

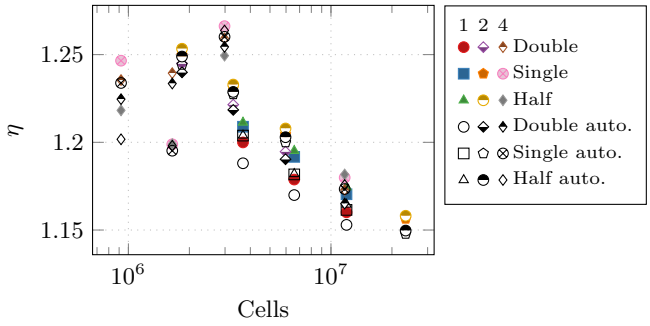


Fig. 5: Evolution of the speedup compared with synchronous execution for a fixed number of gangs.

The speedup for a fixed number of gangs is displayed in fig. 5. Speedup for automatic estimation of the number of gangs is also represented for reference with black empty markers. Using a fixed number of gangs improved speedup by 0.6 % on average (−1.1 % to 1.3 %). The gain is very moderate, the automated estimation gave satisfactory performance without any input from the user.

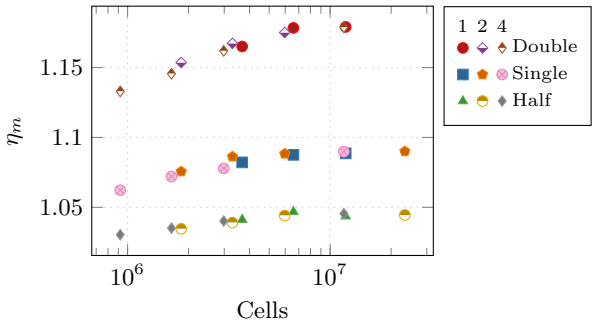


Fig. 6: Evolution of the memory overhead compared with synchronous execution.

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

We analyze now the performance in term of memory overhead. We define the memory overhead η_m as the ratio of the GPU average memory used during execution and reported by the job submission system on JSS3:

$$\eta_m = \frac{m_{\text{async}}}{m_{\text{sync}}} \tag{3}$$

where m_{sync} is the memory of the synchronous execution, and m_{async} is the memory of the asynchronous execution of the different implementations. Evolution of the memory overhead is displayed in fig. 6 as a function of the number of cells. Independently from the precision, the memory overhead of the the asynchronous execution increases with the number cells up to 10^7 cells. Simulations on different number of domains coincide well. Asynchronous execution using double precision has a maximum memory overhead of 18 %, while single precision has a maximum overhead of 9 % (i.e. a half of double precision), and half precision has an overhead of 4.5 % (i.e. a fourth of double precision). This is in agreement with theoretical expectations. The overhead on the CPU memory was also analyzed, and was close to 1 for all simulations.

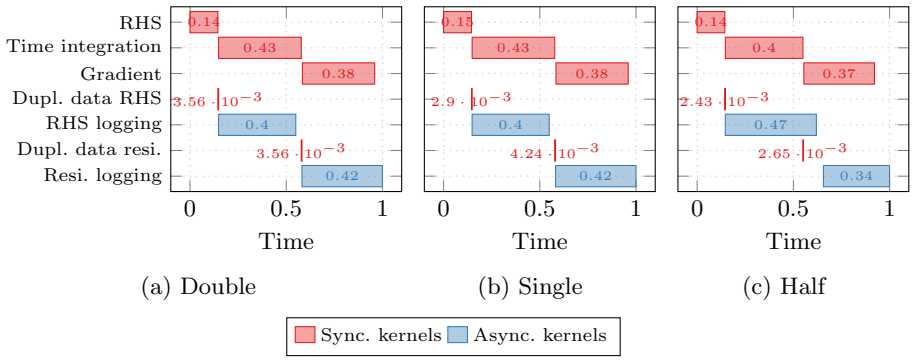


Fig. 7: Timeline execution of the different parts of the solver for asynchronous execution on the fine mesh on 1 domain.

We finalize this performance analysis by profiling the execution of the code for a fixed number of 8 gangs for one iterations with NVIDIA Nsight Systems. A synthetic timeline of the execution of the kernels is displayed in fig. 7 for the asynchronous execution on the fine mesh on 1 domain, for the three precision values. In the figure, the execution time is expressed as a ratio of the iteration time. The asynchronous logging kernels (named *RHS logging* and *resi. logging*) were executed concurrently with the other kernels, and the end of the second logging kernel coincides with the end of the last group of main kernels (named *gradient*), but occurs always a bit later. The timings for each kernels is similar for any precision requested. The time spent in arrays duplication for asynchronous execution (named *dupl. data RHS* and *dupl. data resi.*) was always negligible (less

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

than 0.5 % for any precision value). In the first iteration only, not shown in the graph, a consequent time is spent in creating the pinned memory buffer when preparing to transfer the computed logging values back to the host memory.

5 Conclusion

In this study, we improved the GPU execution of asynchronous logging kernels used in the CFD solver FaSTAR that was initiated in a previous work. We implemented mixed precision in the execution of the logging kernels in order to reduce the memory overhead, by reducing the floating point precision of the arrays they work on. The optimization was tested on four different meshes of the NASA CRM case, ranging from 3.7 to 46.7 million cells, on single GPU and multi-GPU (2 and 4 domains). Simulation results were identical, but a few logged values were incorrectly rendered as $+\infty$ using half precision. Asynchronous execution gave a speedup of 15 % to 27 %, which decreased as the number of cells increased, as there are less remaining threads to saturate. Simulations on 4 domains had more variation in speedup. Using double precision for logging kernels had a maximum memory overhead of 18 %, single precision 9 %, and half precision 4.5 %; it did not have a significant impact on the speedup.

We conclude from this study that background asynchronous execution of logging kernels can improve the speedup of computations for an acceptable memory cost. Single precision is enough to have good fidelity, while half precision is a more aggressive approach that may not be suited for all computations. We believe that the optimization technique presented in this paper can be beneficial to other numerical solvers and improve their performance.

References

1. Bleikamp, R.: BFLOAT16 (Feb 2020), https://j3-fortran.org/doc/year/20/20-118.txt
2. Choi, J., Richards, D.F., Kale, L.V.: Improving Scalability with GPU-Aware Asynchronous Tasks. In: 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 569–578. IEEE, Lyon, France (May 2022). https://doi.org/10.1109/IPDPSW55747.2022.00097
3. Duffy, A.C., Hammond, D.P., Nielsen, E.J.: Production Level CFD Code Acceleration for Hybrid Many-Core Architectures. Tech. Rep. NASA/TM-2012-217770, L-20136, NF1676L-14575, NASA (Oct 2012), https://ntrs.nasa.gov/citations/20120014581
4. Fujita, N.: JSS3/TOKI Overview and Large-Scale Challenge Breaking Report. In: Proceedings of the 53rd Fluid Dynamics Conference/the 39th Aerospace Numerical Simulation Symposium. vol. JAXA-SP-21-008, pp. 95–100. JAXA, Online (Feb 2022), http://id.nii.ac.jp/1696/00048362/
5. Hart, A., Ansaloni, R., Gray, A.: Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. The European Physical Journal Special Topics **210**(1), 5–16 (Aug 2012). https://doi.org/10.1140/epjst/e2012-01634-y

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

6. Hashimoto, A., Ishida, T., Aoyama, T., Hayashi, K., Takekawa, K.: Fast Parallel Computing with Unstructured Grid Flow Solver (May 2016), 28th International Conference on Parallel Computational Fluid Dynamics, Parallel CFD'2016
7. Hashimoto, A., Ishida, T., Aoyama, T., Takekawa, K., Hayashi, K.: Results of Three-dimensional Turbulent Flow with FaSTAR. In: 54th AIAA Aerospace Sciences Meeting. American Institute of Aeronautics and Astronautics, San Diego, California, United States of America (Jan 2016). https://doi.org/10.2514/6.2016-1358
8. Hennessy, J.L., Patterson, D.A.: Chapter Four: Data-Level Parallelism in Vector, SIMD, and GPU Architectures. In: Computer Architecture: A Quantitative Approach, pp. 281–365. Morgan Kaufmann Publishers, Cambridge, Massachusetts, United States of America, sixth edn. (2019)
9. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp. 1–84 (Jul 2019). https://doi.org/10.1109/IEEESTD.2019.8766229
10. Ito, Y., Murayama, M., Hashimoto, A., Ishida, T., Yamamoto, K., Aoyama, T., Tanaka, K., Hayashi, K., Ueshima, K., Nagata, T., Ueno, Y., Ochi, A.: TAS Code, FaSTAR, and Cflow Results for the Sixth Drag Prediction Workshop. Journal of Aircraft **55**(4), 1433–1457 (Jul 2018). https://doi.org/10.2514/1.C034421
11. Kale, L.V., Krishnan, S.: CHARM++: A portable concurrent object oriented system based on C++. In: Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications - OOPSLA '93. pp. 91–108. ACM Press, Washington, D.C., United States of America (1993). https://doi.org/10.1145/165854.165874, http://portal.acm.org/citation.cfm?doid=165854.165874
12. McCall, A.J.: Multi-Level Parallelism with MPI and OpenACC for CFD Applications. Master of Science thesis, Virginia Tech (Jun 2017), https://vtechworks.lib.vt.edu/handle/10919/78203
13. Micikevicius, P.: 3D finite difference computation on GPUs using CUDA. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2). pp. 79–84. ACM Press, Washington, D.C., United States of America (Mar 2009). https://doi.org/10.1145/1513895.1513905
14. NVIDIA: NVIDIA V100 Datasheet. Tech. rep. (Jan 2020), https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf
15. NVIDIA: CUDA Toolkit Documentation (Mar 2022), https://docs.nvidia.com/cuda/index.html
16. Obayashi, S., Guruswamy, G.P.: Convergence acceleration of a Navier-Stokes solver for efficient static aeroelastic computations. AIAA Journal **33**(6), 1134–1141 (Jun 1995). https://doi.org/10.2514/3.12533
17. OpenACC: OpenACC API 2.7 Reference Guide (Oct 2018), https://www.openacc.org/sites/default/files/inline-files/API%20Guide%202.7.pdf
18. Riley, D.: Intel 4th Gen Xeon series offers a leap in data center performance and efficiency (Jan 2023), https://siliconangle.com/2023/01/10/intel-4th-gen-xeon-series-offers-leap-data-center-performance-efficiency/
19. Searles, R., Chandrasekaran, S., Joubert, W., Hernandez, O.: MPI + OpenACC: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems. Computer Physics Communications **236**, 176–187 (Mar 2019). https://doi.org/10.1016/j.cpc.2018.10.007
20. Shi, X., Agrawal, T., Lin, C.A., Hwang, F.N., Chiu, T.H.: A parallel nonlinear multigrid solver for unsteady incompressible flow simulation on multi-GPU cluster. Journal of Computational Physics **414**, 109447 (Aug 2020). https://doi.org/10.1016/j.jcp.2020.109447

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32

21. Shima, E., Kitamura, K., Haga, T.: Green–Gauss/Weighted-Least-Squares Hybrid Gradient Reconstruction for Arbitrary Polyhedra Unstructured Grids. AIAA Journal **51**(11), 2740–2747 (Nov 2013). https://doi.org/10.2514/1.J052095
22. Trader, T.: How Argonne Is Preparing for Exascale in 2022. HPCwire (Sep 2021), https://www.hpcwire.com/2021/09/08/how-argonne-is-preparing-for-exascale-in-2022/
23. Vassberg, J., Dehaan, M., Rivers, M., Wahls, R.: Development of a Common Research Model for Applied CFD Validation Studies. In: 26th AIAA Applied Aerodynamics Conference. American Institute of Aeronautics and Astronautics, Honolulu, Hawaii, United States of America (Aug 2008). https://doi.org/10.2514/6.2008-6919
24. Walden, A., Nielsen, E., Diskin, B., Zubair, M.: A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs. In: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3). pp. 23–30. IEEE, Denver, Colorado, United States of America (Nov 2019). https://doi.org/10.1109/IA349570.2019.00010
25. Wang, S., Kanwar, P.: BFloat16: The secret to high performance on Cloud TPUs (Aug 2019), https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus
26. Wright, M.J., Candler, G.V., Prampolini, M.: Data-parallel lower-upper relaxation method for the Navier-Stokes equations. AIAA Journal **34**(7), 1371–1377 (Jul 1996). https://doi.org/10.2514/3.13242
27. Yoon, S., Jameson, A.: An LU-SSOR scheme for the Euler and Navier-Stokes equations. In: 25th AIAA Aerospace Sciences Meeting. American Institute of Aeronautics and Astronautics, Reno, Nevada, United States of America (Mar 1987). https://doi.org/10.2514/6.1987-600
28. Zehner, P., Hashimoto, A.: Asynchronous Execution of Logging Kernels in a GPU Accelerated CFD Solver. In: Proceedings of the 54th Fluid Dynamics Conference/the 40th Aerospace Numerical Simulation Symposium. vol. JAXA-SP-22-007, pp. 331–339. Japan Aerospace Exploration Agency (JAXA), Morioka, Japan (Jun 2022), http://id.nii.ac.jp/1696/00049141/
29. Zehner, P., Hashimoto, A.: Acceleration of the data-parallel lower-upper relaxation time-integration method on GPU for an unstructured CFD solver. Computers & Fluids p. 105842 (Mar 2023). https://doi.org/10.1016/j.compfluid.2023.105842

ICCS Camera Ready Version 2023
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-36024-4_32