Overcoming GPU Memory Capacity Limitations in Hybrid MPI Implementations of CFD

Jake Choi[1(✉)], Yoonhee Kim[2], and Heon-young Yeom[1]

[1] Department of Computer Science, Seoul National University, Seoul, South Korea
kidcoder@snu.ac.kr

[2] Department of Computer Science, Sookmyung Women's University, Seoul, South Korea

Abstract. In this paper, we describe a hybrid MPI implementation of a discontinuous Galerkin scheme in Computational Fluid Dynamics which can utilize all the available processing units (CPU cores or GPU devices) on each computational node. We describe the optimization techniques used in our GPU implementation making it up to 74.88x faster than the single core CPU implementation in our machine environment. We also perform experiments on work partitioning between heterogeneous devices to measure the ideal load balance achieving the optimal performance in a single node consisting of heterogeneous processing units. The key problem is that CFD workloads need to allocate large amounts of both host and GPU device memory in order to compute accurate results. There exists an economic burden, not to mention additional communication overheads of simply scaling out by adding more nodes with high-end scientific GPU devices. In a micro-management perspective, workload size in each single node is also limited by its attached GPU memory capacity. To overcome this, we use ZFP, a floating-point compression algorithm to save at least 25% of data usage in our workloads, with less performance degradation than using NVIDIA UM.

Keywords: CFD · MPI · CUDA · GPU · Compression · Memory

1 Introduction

The rising demand for analyzing more complex aerodynamic applications has led to the development of high-order methods that break through the limitations of conventional 2nd-order finite volume methods (FVM). The high-order methods possess many attractive features: the capability to achieve arbitrary high accuracy with compact stencils, high spectral resolvability fitted to turbulence simulation, and high scalability under large parallel computing systems.

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2015M3C4A7065646).

In this paper, the three-dimensional compressible Navier-Stokes equations given by

$$\frac{\partial \mathbf{Q}}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{F_c}(\mathbf{Q}) = \boldsymbol{\nabla} \cdot \mathbf{F_v}(\mathbf{Q}, \boldsymbol{\nabla}\mathbf{Q}), \tag{1}$$

where $\mathbf{Q}$ are conservative variables and $\mathbf{F_c}(\mathbf{Q}), \mathbf{F_v}(\mathbf{Q}, \boldsymbol{\nabla}\mathbf{Q})$ are convective and viscous fluxes respectively, are considered.

Among various high-order methods, we focus on the discontinuous Galerkin method [21] that is the most widely used in CFD society because of its intuitive form and rigorous mathematical backgrounds. Applying the discontinuous Galerkin method into Eq. (1), we finally get the following weak formulation on each element Ω:

$$\begin{aligned}&\int_{\Omega} \frac{\partial \mathbf{Q}}{\partial t} \phi dV + \int_{\partial\Omega} \left(\widehat{\mathbf{F_c} \cdot \mathbf{n}} - \widehat{\mathbf{F_v} \cdot \mathbf{n}} \right) \phi dA = \\ &\int_{\Omega} \boldsymbol{\nabla}\phi \cdot (\mathbf{F_c} - \mathbf{F_v})\, dV,\end{aligned} \tag{2}$$

where ϕ is the orthogonal basis computed from the modified Gram-Schmidt process [22]. Here, $\widehat{\mathbf{F_c} \cdot \mathbf{n}}$ is a monotone numerical convective flux used in FVM, and $\widehat{\mathbf{F_v} \cdot \mathbf{n}}$ is a numerical viscous flux computed via BR2 method [23].

Both massively multicore GPUs and clusters of CPUs can be used to accelerate such high-order methods of computational fluid dynamics (CFD). CFD workloads are composed of calculation-heavy tasks, where input and output data can be divided into independent portions, with little communication necessary in each iterative step. Therefore, the benefit of performing calculations in parallel outweigh the costs of communication among different independent tasks. In this paper, we initially implement a hybrid MPI implementation to accelerate CFD workloads on both CPUs and GPUs in parallel. We show performance improvements of up to 22% compared to the GPU-only version.

One potential problem that can arise is that data allocated to the GPU could exceed device memory capacity. Workload sizes beyond hundreds of GB cannot be fully contained in even high-end GPU devices like the Tesla P100, which only possess a mere 16 GB of memory capacity. To rectify this, previously unnecessary memory management operations transferring data to and from host memory need to be taken in each iteration step, leading to large overheads. In order to not deal with such nuisances, NVIDIA UM (Unified Memory) [25] can be used to unify the GPU device memory with host memory.

Introduced in CUDA 6, NVIDIA UM is a single memory address space that is accessible from any processor in the system [26]. It allows applications to allocate data that can be written or read to from code running on either CPUs or GPUs. Using UM eliminates the need for explicit memory copies from host memory to GPU device memory. The system will automatically perform page migration on-demand to the memory of the accessing processor. Even though UM provides simplicity in its usage, the system is ignorant of the actual data access patterns of applications using it. In our evaluation, we show that total performance drops significantly if the working set size for the GPU process exceeds its total device memory.

In order to overcome such performance limitations, our GPU implementation utilizes CUDA-based ZFP [31], a floating point compression algorithm, to compress the working set data in the pre-processing step. We achieve lossless compression rates of up to 50%, and perform partial block decompression on the GPU. We evaluate the performance overheads of decompression for various CFD workloads in comparison to the baseline UM performance. We also evaluate the amount of compression we can achieve on the working set without incurring data loss.

To summarize, our paper makes the following contributions:

- We derive GPU kernel implementations of CFD from the CPU version that are optimized to perform better than the cuBLAS library on general consumer commodity GeForce GPUs.
- Our GPU implementation is capable of utilizing NVIDIA UM.
- We utilize MPI to allow parallel execution on both CPUs and GPUs in a heterogeneous environment.
- We use two techniques, ZFP partial block decompression, and GPU memory overwriting to reduce data usage by up to a maximum of 50% with less overhead than simply managing data automatically with UM.

The rest of this paper is structured as follows. We first examine related work in Sect. 2. Subsequently, we describe the implementation of our techniques in three subsections in Sect. 3. We describe our experimental setup and show evaluation results in Sect. 4, and conclude with directions for future work in Sect. 5.

2 Related Work

The field of CFD has an extensive amount of existing literature mainly consisting of GPU implementations that boast magnitude-of-order speedups over sequential single-threaded CPU code. Work using multi-threaded CPU implementations also seem to compete against their GPU counterparts. The common thread of such works is that most advocate using purely homogeneous multi-threaded CPU [19,20] or GPU [8,9,16] implementations, but rarely are hybrid implementations considered. Such works essentially recommend competing implementations with a preference towards one type of architecture. However, our work uses an heterogeneous MPI implementation which can take full advantage of all available processing units. Some GPU implementations are able to utilize multiple devices [11,17,18] for scalability, but these works also do not consider CPU usage in parallel. Albeit there are less common works which suggest efficient methods of using a combination of either MPI or OpenMP to solve CFD problems in heterogeneous systems [28], these works deal with less complex methods that are lower-order. They also do not mention GPU memory capacity issues regarding large workloads at all. Moreover, to the best of our knowledge, we were unable to find any related work that dealt with reducing the data usage of CFD workloads in GPU memory.

3 Implementation

Our GPU implementation has two versions, one using NVIDIA UM, and the other using manual CUDA memory operations. Workload size is limited to GPU device capacity when ordinary `cudaMalloc` is used, but using UM allows the workload size to reach host memory limits. In the latter case we prefetch data from the host to GPU memory by using `cudaMemPrefetchAsync`. Table 1 shows the size of major data arrays along with their access type in our GPU kernels.

When executing our GPU implementation, only one CPU core is responsible for the management of GPU operations. This leaves the other cores idle. By partitioning the workload in the pre-processing stage using ParMETIS [29], we assigned different weights to different MPI processes, allocating a subset of the data to each process. By doing so, we are able to independently execute different processes on either GPU devices or CPU cores, based on the rank of the process. In the following Subsects. 3.1, 3.2, and 3.3, we will explain the GPU optimizations we performed in detail, along with how we used ZFP compression or memory buffer overwriting to save data usage when GPU memory capacity is not sufficient to completely contain the given workload.

3.1 CUDA Optimizations

The CUDA implementation consists of a total of 17 separate kernels. Among these 17 kernels, 9 are used in calculations needed for computing intermediate **rhs** values, 5 are used for updating the **solution** values and the remaining 3 is for calculating the time step after each iteration. A profiling of the kernels using *nvprof* [10] is performed to show which kernels take up the majority of program time in Table 1. Kernels with less execution time are omitted. Among the nine major kernels, three kernels each are responsible for calculating the face, periodic boundary and boundary, and cell values, respectively. We prefix such kernels as *first_loop_#*, *third_loop_#*, and *fourth_loop_#* respectively. Each kernel runs with a different number of spawned threads and blocks. Intermediate data is stored in global memory. If there are no data dependencies among the kernels, we run them in different CUDA streams so that they can run simultaneously when GPU streaming multiprocessors are not fully utilized.

Algorithm 1 shows the CPU and GPU version of *(fourth_loop_3)*. We store spatially close data in shared memory to take advantage of global memory coalescing, exemplified in line 2. We usually set the *blockDim* and *gridDim* of each kernel to match the respective number of cells, points, states or basis values, shown as loop indices in the CPU version, with some exceptions, where the number of spawned threads of the kernel is set to 32 to match the warp thread count, for performance optimization reasons. Using atomic functions for Algorithm 1 shows better performance than shared memory reduction because there

Algorithm 1. Third stage of calculation of cells

```
Input: 2D vectors cell_coefficients as $cc$, $flux$
Output: 2D vector rhs as $rhs$

CPU Version
1: for $i \leftarrow 0$ to $num\_cells$ do
2:     for $j \leftarrow 0$ to $num\_points$ do
3:         for $k \leftarrow 0$ to $num\_states$ do
4:             for $l \leftarrow 0$ to $num\_basis$ do
5:                 for $m \leftarrow 0$ to $dimensions$ do
6:                     $rhs_{i,k,l} \leftarrow rhs_{i,k,l} - (cc_{i,j,l,m} * flux_{i,j,k,m})$
7:                 end for
8:             end for
9:         end for
10:    end for
11: end for

GPU Version
$i \leftarrow$ blockIdx.x, $j \leftarrow$ blockIdx.y
$l \leftarrow$ threadIdx.x
1: procedure FOURTH_LOOP_3                          ▷ Performed in parallel
2:     Declare __shared__ memory $cc\_$
3:     $cc\_{k,m} \leftarrow cc_{i,j,k,m}$
4:     for $k \leftarrow 0$ to $num\_states$ do
5:         $temp \leftarrow 0$
6:         for $m \leftarrow 0$ to $dimensions$ do
7:             $temp \leftarrow temp - (cc\_{k,m} * flux_{i,j,k,m})$
8:         end for
9:         atomicAdd $temp$ to $rhs_{i,k,l}$
10:    end for
11: end procedure
```

is little contention amongst the *blockIdx.y* axis blocks for the same memory location, and using reduction will cause poor memory access patterns and lower functional utilization.

3.2 ZFP Compression and Block Decompression

ZFP is a fixed-rate, near-lossless compression scheme that maps small blocks of 4^d values in d dimensions to a fixed, user-specified number of bits per block, thereby allowing read and write random access to compressed floating-point data at block granularity [2]. ZFP shows much higher compression rates for floating-point data compared to other generic lossless compression schemes like Gzip [4], bzip2 [5], or even floating point compression schemes like FPZIP [6]. This is because floating-point values have tailing mantissa bits that are too random to compress effectively [7]. ZFP is also relatively accurate because of its bounded relative error [3]. It supports three modes: fixed-rate, fixed-accuracy, and fixed-

Table 1. List of major data arrays referenced in GPU Kernels and Major Kernels

Array name	RW	% of data usage	Kernel name	Avg. time	% of time
cell_coefficients	R	50.81%	fourth_loop_3	9.74 ms	38.18%
cell_basis_value	R	16.94%	fourth_loop_1	5.93 ms	23.24%
face_owner_basis_value	R	5.77%	third_loop_1	3.95 ms	15.47%
face_neighbor_basis_value	R	5.77%	third_loop_3	2.81 ms	11.01%
face_owner_coefficients	R	5.77%	first_loop_1	1.22 ms	4.79%
face_neighbor_coefficients	R	5.77%	first_loop_3	583.88 µs	2.29%
peribdry_owner_basis_value	R	1.15%	fourth_loop_2	490.32 µs	1.92%
peribdry_neighbor_basis_value	R	1.15%	third_loop_2	334.34 µs	1.31%
peribdry_owner_coefficients	R	1.15%	memcpy HtoD	7.38 µs	1.10%
Flux	RW	3.02%	first_loop_2	125.88 µs	0.49%
solution	RW	0.17%	memcpy DtoD	15.39 µs	0.05%
rhs	W	0.17%			

precision. In fixed-rate mode, each d-dimensional compressed block of 4^d values is stored using a fixed number of bits that we can set as a parameter.

Our scheme uses CUDA ZFP to compress and decompress data in the GPU in parallel. We use fixed-rate mode as the other modes are not supported by CUDA ZFP. The reason for this is because we need random access to the compressed blocks. In fixed-rate mode, the size of all compressed blocks are constant, allowing partial decompression to take place at any order. The number of bits that are used to store each block is input as a parameter to the compression and decompression functions. This parameter needs to exactly be a power of two otherwise the number of actual compressed bits per block will be rounded up to the next largest power of two. Therefore in fixed-rate mode, we can only achieve exact compression ratios of 50%, 75%, 87.5%, and so forth. We directly incorporate the encoding and decoding functions from source, because the GPU API is not exported into the shared library when compiled. Before application modifications are made, we take note of which read-only data buffer would potentially use the greatest amount of space in GPU device memory, and is referenced scarcely. This is because our goal is to minimize decompression overheads and data loss. We encode the largest buffer directly into GPU device memory only once in the pre-processing stages by directly calling the encode launch kernel function for one-dimensional arrays of type *double*.

Decompression is performed directly in the kernels before references to the compressed arrays are made. We modified the device decode functions to work with larger block indices, up to the value limit of type *unsigned long*. Then, we used block decompression in each thread of our calculation kernel which references the data values in the compressed buffer. Because fixed-rate mode requires each block to contains 4^d values, where d = 1, we can obtain 4 decoded 64-bit values from the decode function per CUDA thread. Because the number of threads per CUDA block is equivalent to the **basis value**, some of the threads would have to wait in a synchronization step before the results from the other threads are all stored in shared memory. Once block decompression is finished, the kernel continues with its task. By using block decompression, we can amortize the overhead of decompression in the calculation kernel itself without incurring additional kernel launch overheads, and can discard the decompressed values from shared memory after they are used.

3.3 Overwriting GPU Data Buffers Using Memory Copy

A naive method of saving GPU memory capacity is to keep only the absolute necessary data in GPU device memory. We can perform this by calling `cudaMemcpy` before the kernel referencing the data is launched and calling `cudaFree` immediately after the kernel execution completes before the next kernel is launched. Not only is this method extremely inefficient, it is also impractical to pipeline. Data residing in GPU memory would have to be constantly freed and copied before each calculation step, leading to large data transfer overheads. Additionally, `cudaFree` is a synchronous operation with an internal synchronization call. Therefore we cannot pipeline the kernels with the data transfers.

We use a different technique of avoiding `cudaFree` altogether by sharing buffers across multiple kernels. Like in Sect. 3.2, we select the **cell_coefficients** buffer because it utilizes the largest capacity. Once the kernel using this buffer finishes execution, we simply overwrite its contents using `cudaMemcpy` with the host data that we do not want to store in GPU device memory. We are able to save GPU space because we do not have to `cudaMalloc` distinct buffers for such arrays. Based on Table 1, we are able to theoretically save a maximum of 50% of GPU space if we utilize the largest buffer for all of the required data.

4 Evaluation

4.1 Experimental Setup

We use two experimental environments. The first environment is a single private machine equipped with NVIDIA GeForce GTX 1050 Ti using the Pascal architecture. The CPU we use is an Intel i7-7700 @ 3.6GHz with 4 physical cores. The second environment consists of a single server node equipped with 2 NVIDIA GeForce Titan XP also using the Pascal architecture. The CPU for this node is an Intel Xeon E5-2683 @ 2.1GHz with 2 sockets equipped with 16 cores each.

4.2 Performance Results

Figure 1 shows the performance results of both the CPU and GPU implementation in both environments. The x-axis shows the size of the workload that we used, and the y-axis is the execution time in seconds or the speedup. We notice that the multi-core version utilizing all 8 cores with hyper-threading on shows a maximum speedup of 4.45 times the sequential version. Using the GTX 1050 Ti speeds up performance up to 9.02 times sequential code. Titan XP shows

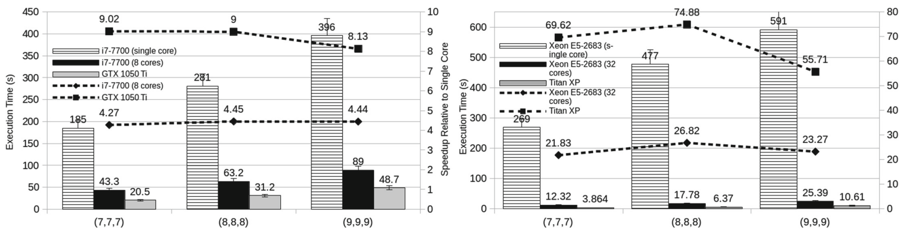

Fig. 1. GeForce GTX 1050 Ti and Titan XP performance results (100 iterations)

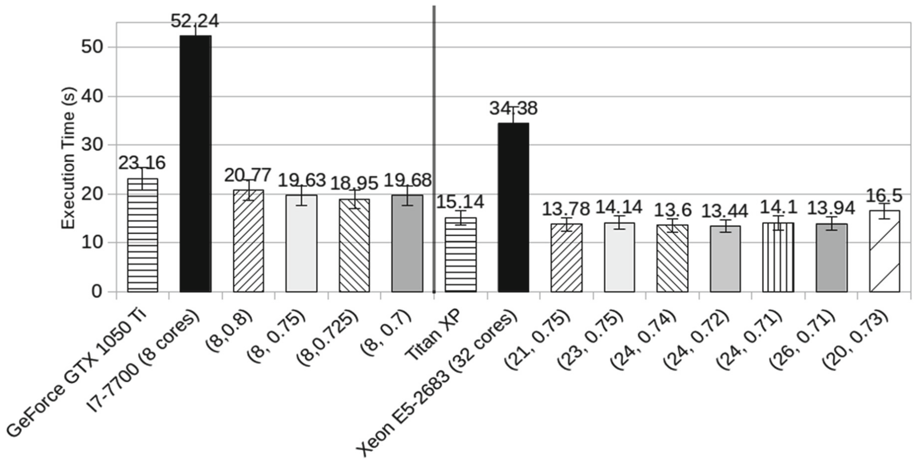

Fig. 2. Hybrid run using MPI on (7, 7, 7) and (10, 10, 10) workloads

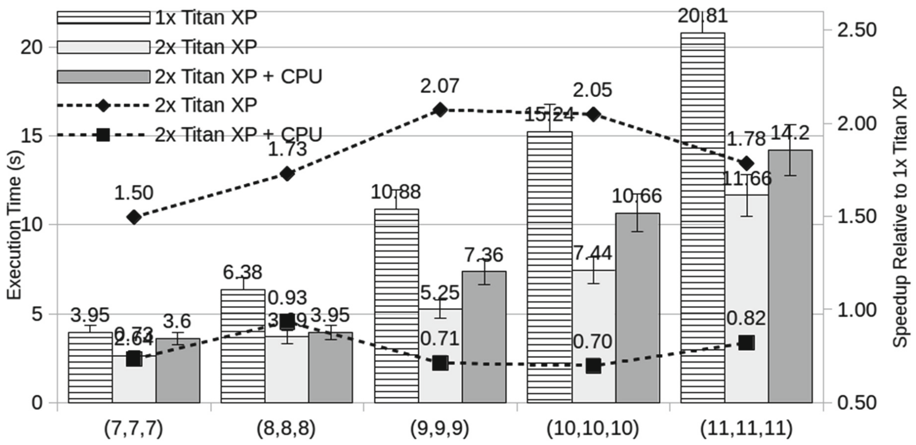

Fig. 3. Hybrid run using MPI on multiple GPU devices

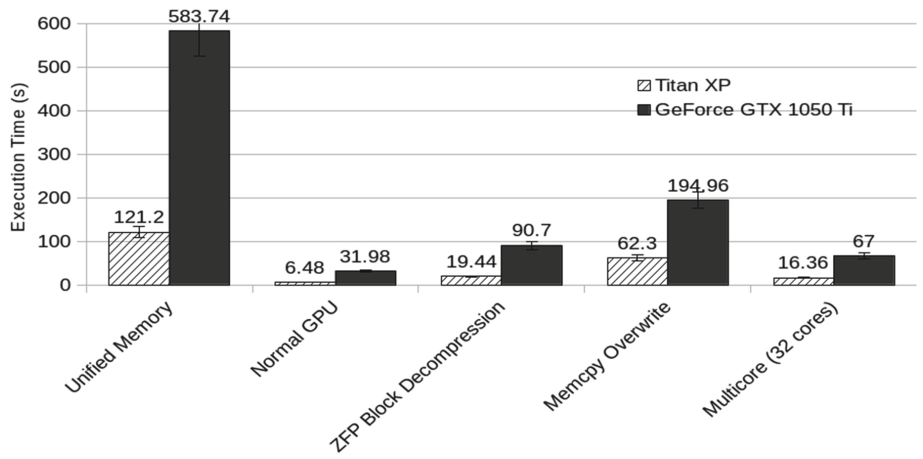

Fig. 4. Comparison of data saving techniques with unified memory

speedups of up to 74.88 times the sequential version. Both experiments ran each workload for 100 iterations.

Figure 2 shows the results when a small workload is run on the private machine, and a bigger one on the server machine. The first two columns of each division denote sole GPU or CPU execution times, and the remaining columns show heterogeneous execution times. The two parameters of the x-axis show the number of MPI processes, and weight given to a single GPU device. Results show that assigning a workload weight of 72% to the GPU causes the most speedup (about 11% compared to GPU-only) for the server machine. We believe the speedup is not completely scalable because of the MPI communication overheads. There are a fewer number of cores co-located near each other in the private machine, causing less communication overheads. Each individual core also has a greater clock frequency, contributing to higher performance per core. In the private machine, we experience a speedup of about 22% compared to when the GTX 1050 Ti executes the code exclusively.

Using multiple GPU devices also further reduce execution time, as shown in Fig. 3. We run the workload in two Titan XPs in parallel using MPI, and assign half the weight to each device. As the workload size becomes larger, performance generally becomes better as the divisibility of the workload from the ParMETIS library becomes more congruent. The middle cases (9, 9, 9) and (10, 10, 10) actually experience a greater than linear speedup, most likely due to the kernels being more optimal in block and thread size to allow the kernel optimizations to work more efficiently.

Running the workload on all processing units actually performs worse than dividing the workload equally into multiple GPU devices. We do not execute the program with too many CPU cores, because partitioning the workload too finely will prevent some MPI processes from receiving any data at all, leading to errors. In order to properly experience program speedup, the GPU device must be assigned the proper ratio of workload depending on its processing power relative to a single CPU core. We use heuristics to find the optimal weight that we have to give each processing unit in the cluster. However, ParMETIS cannot allocate an exactly equal amount of data to each process, which leads to synchronization discrepancies. This could cause CPU cores to lag behind the GPU devices, which would be idle in `MPI_Wait()` while the CPU cores are still busy.

We compare our data usage saving techniques with NVIDIA UM on both GPU devices in Fig. 4. We run a (8, 8, 8) workload for 100 iterations. We performed the experiment by filling up the GPU memory using `cudaMalloc` to guarantee that the UM paging mechanism will be called when the workload executes. When GPU memory is insufficient, resorting to UM slows down program execution more than one order of magnitude because of page faults. We compressed **cell_coefficients** (see Table 1) with a fixed-rate of 50% so that data loss would be minimized. Therefore we were able to save 25% of total data usage and reduced execution time by up to 6.4 times UM. Likewise, we eliminated the **cell_basis_value**, **face_owner_basis_value** and **face_neighbor_basis_value** buffers for our memory copy overwrite technique, saving a total of 28.48% of data usage while achieving a speedup of up to 3 times UM. Memory copy overwrite has more overhead than block decompression, and that is because the time spent in additional `cudaMemcpy` operations exceed the amortized kernel decompression time by more than a factor of two. These operations are repeated five times in each iteration, causing large overheads.

5 Conclusion and Future Work

CFD is a widely researched application relevant to many scientific fields. Implementations of CFD are scalable, as the application running time generally decreases when the number of processing units (GPU or CPU) are increased. However, scarcity of GPU memory compared to host memory limits CFD workloads to GPU device memory capacities.

Our solution, which includes ZFP block decompression and memory copy overwrites allows a minimum of 25% larger CFD workloads to run with adequate performance on different types of commodity GPUs, without resorting to additional money spent on purchasing more GPU devices, or cluster nodes for the need of adding more GPU slots. The effect is for more data to be packed in the GPUs of each cluster node, reducing MPI communication overheads which can potentially hamper the scalability of execution time of the application. Our GPU implementation performs up to 74.88 times faster than the sequential version on cheap, commodity GPUs. Finally, we are able to further increase performance by partitioning the application workload to different MPI processes to utilize all heterogeneous processing units in the cluster.

In future work, we shift our focus to how we can manage multiple CFD applications running simultaneously. We also want to automate the process of finding optimal weights, especially when network nodes are used. Finally, we pursue a generalized compression method in the system layer.

References

1. Lai, J., Li, H., Tian, Z.: CPU/GPU heterogeneous parallel CFD solver and optimizations. In: Proceedings of the 2018 International Conference on Service Robotics Technologies (ICSRT '18), pp. 88–92. ACM, New York (2018). https://doi.org/10.1145/3208833.3208847
2. Lindstrom, P.: Fixed-rate compressed floating-point arrays. IEEE Trans. Vis. Comput. Graph. **20**(12), 2674–2683 (2014). https://doi.org/10.1109/TVCG.2014.2346458
3. Lindstrom, P.: Error distributions of lossy floating-point compressors. Joint Stat. Meet. **2017**, 2574–2589 (2017)
4. Deutsch, P.: GZIP file format specification version 4.3. RFC, vol. 1952, pp. 1–12 (1996). https://doi.org/10.17487/RFC1952
5. Bzip2 (2018). http://www.bzip.org/
6. Lindstrom, P., Isenburg, M.: Fast and efficient compression of floating-point data. IEEE Trans. Vis. Comput. Graph. **12**(5), 1245–1250 (2006)
7. Tao, D., Di, S., Liang, X., Chen, Z., Cappello, F.: Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP (2019). https://doi.org/10.1109/TPDS.2019.2894404
8. Niksiar, P., Ashrafizadeh, A., Shams, M., Madani, A.H.: Implementation of a GPU-based CFD Code. In: 2014 International Conference on Computational Science and Computational Intelligence, Las Vegas, NV, pp. 84–89 (2014). https://doi.org/10.1109/CSCI.2014.21
9. Mintu, S.A., Molyneux, D.: Application of GPGPU to accelerate CFD simulation. In: ASME International Conference on Offshore Mechanics and Arctic Engineering, vol. 2: CFD and FSI ():V002T08A001. https://doi.org/10.1115/OMAE2018-77649
10. NVIDIA Corp: Profiler user's guide (2017). https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview. An optional note
11. Griebel, M., Zaspel, P.: Comput. Sci. Res. Dev. **25**, 65 (2010). https://doi.org/10.1007/s00450-010-0111-7
12. Xu, H., et al.: Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer. J. Comput. Phys. **278**(C), 275–297 (2013). https://doi.org/10.1016/j.jcp.2014.08.024
13. Videocardbenchmark.net. PassMark Software - Video Card (GPU) Benchmark Charts (2019). https://www.videocardbenchmark.net/. Accessed 24 May 2019
14. Cpubenchmark.net. PassMark Software - CPU Benchmark Charts (2019). https://www.cpubenchmark.net/. Accessed 24 May 2019
15. Ark.intel.com. Intel product specifications (2019). https://ark.intel.com/content/www/us/en/ark.html. Accessed 24 May 2019
16. Wang, Y., Malkawi, A., Yi, Y.K.: Implementing CFD (computational fluid dynamics) in OpenCL for building simulation (2011)
17. Gorobets, A., Soukov, S., Bogdanov, P.: Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers. Comput. Fluids **173** (2018). https://doi.org/10.1016/j.compfluid.2018.03.011
18. Oyarzun, G., Borrell, R., Gorobets, A., Mantovani, F., Oliva, A.: Efficient CFD code implementation for the ARM-based mont-blanc architecture. Future Gener. Comput. Syst. **79** (2017). https://doi.org/10.1016/j.future.2017.09.029
19. Wang, Y.X., Zhang, L.L., Liu, W., Cheng, X.H., Zhuang, Y., Chronopoulos, A.: Performance optimizations for scalable CFD applications on hybrid CPU+MIC heterogeneous computing system with millions of cores. Comput. Fluids (2018). https://doi.org/10.1016/j.compfluid.2018.03.005

20. Che, Y., Zhang, L., Xu, C., Wang, Y., Liu, W., Wang, Z.: Optimization of a parallel CFD code and its performance evaluation on Tianhe-1A. Comput. Inf. **33**, 1377–1399 (2014)
21. Cockburn, B., Shu, C.W.: The Runge-Kutta discontinuous Galerkin method for conservation laws V. J. Comput. Phys. **141**, 199–224 (1998)
22. You, H., Kim, C.: High-order multi-dimensional limiting strategy with subcell resolution I. Two-Dimension. Mixed Meshes, J. Comput. Phys. **375**, 1005–1032 (2018)
23. Bassi, F., Crivellini, A., Rebay, S., Savini, M.: Discontinuous Galerkin solution of the Reynolds-averaged Navier-Stokes and k-ω turbulence model equations. Comput. Fluids **34**, 507–540 (2005)
24. Cohen, J., Molemaker, M.J.: A fast double precision CFD code using CUDA. Parallel Computational Fluid Dynamics: Recent Advances and Future Directions (2009)
25. Li, W., Jin, G., Cui, X., See, S.: An evaluation of unified memory technology on NVIDIA GPUs. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen, pp. 1092-1098 (2015). https://doi.org/10.1109/CCGrid.2015.105
26. Harris, M., Harris, M., Harris, M., Sakharnykh, N., Harris, M.: Unified memory for CUDA beginners—NVIDIA developer blog. NVIDIA Developer Blog (2019). https://devblogs.nvidia.com/unified-memory-cuda-beginners/. Accessed 17 May 2019
27. Harris, M., Perelygin, K., Luitjens, J., Karras, T., Karras, T., Karras, T.: Cooperative groups: flexible CUDA thread programming—NVIDIA developer blog. NVIDIA Developer Blog (2019). https://devblogs.nvidia.com/cooperative-groups/. Accessed 22 May 2019
28. Oteski, L., Colin de Verdiere, G., Contassot-Vivier, S., Vialle, S., Ryan, J.: Towards a unified CPU-GPU code hybridization: a GPU based optimization strategy efficient on other modern architectures (2018)
29. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM), Ser. Supercomputing '96. IEEE Computer Society, Washington, DC, USA (1996). https://doi.org/10.1145/369028.369103
30. NVIDIA: NVIDIA CUBLAS Library (2019). https://developer.nvidia.com/cublas
31. Larsen, M.: mclarsen/cuZFP. GitHub (2019). https://github.com/mclarsen/cuZFP. Accessed 22 May 2019