A High-productivity Framework for Adaptive Mesh Refinement on Multiple GPUs

Takashi Shimokawabe[1] and Naoyuki Onodera[2]

[1] Information Technology Center, The University of Tokyo, Japan
shimokawabe@cc.u-tokyo.ac.jp
[2] Center for Computational Science and e-Systems, Japan Atomic Energy Agency, Japan

Abstract. Recently grid-based physical simulations with multiple GPUs require effective methods to adapt grid resolution to certain sensitive regions of simulations. In the GPU computation, an adaptive mesh refinement (AMR) method is one of the effective methods to compute certain local regions that demand higher accuracy with higher resolution. However, the AMR methods using multiple GPUs demand complicated implementation and require various optimizations suitable for GPU computation in order to obtain high performance. Our AMR framework provides a high-productive programming environment of a block-based AMR for grid-based applications. Programmers just write the stencil functions that update a grid point on Cartesian grid, which are executed over a tree-based AMR data structure effectively by the framework. It also provides the efficient GPU-suitable methods for halo exchange and mesh refinement with a dynamic load balance technique. The framework-based application for compressible flow has achieved to reduce the computational time to less than 15% with 10% of memory footprint in the best case compared to the equivalent computation running on the fine uniform grid. It also has demonstrated good weak scalability with 84% of the parallel efficiency on the TSUBAME3.0 supercomputer.

Keywords: Adaptive mesh refinement · GPU · Stencil computation

1 Introduction

The stencil-based applications are important applications running on the GPU supercomputers. Thanks to the wide bandwidth and high computational power of GPU, various stencil applications have successfully achieved high performance [8, 7, 11]. Recently grid-based physical simulations with multiple GPUs require effective methods to adapt grid resolution to certain sensitive regions of simulations. An adaptive mesh refinement (AMR) method is one of the key technique to compute certain local regions that demand higher accuracy with higher resolution [1, 3, 6]. While GPU computation has the potential to achieve high performance, it forces the programmer to learn multiple distinctive programming models such as CUDA or OpenACC and introduce various complicated

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

optimizations. For this reason, most of existing AMR libraries supporting GPU provide only several numerical schemes that optimized for GPU, or the programmer has to provide GPU optimized kernels written in CUDA [6].

In order to improve productivity and achieve high performance, various types of high-level programming models for GPU were proposed [5, 13, 2, 4, 9, 10]. However, since these programming models focus on stencil computations on uniform grids, it is difficult to apply them to the AMR applications where additional data structures such as tree structures are essential. Although Daino was proposed as a directives-based programming framework for AMR on GPUs, it needed to use its own directives [14]. To enhance the portability and transparency of frameworks themselves and the user codes using them, the framework should be written in standard languages without language extension.

In this paper, we propose a high-productivity framework for a block-based AMR for grid-based applications running on multiple GPUs. In previous research, we proposed a high-productivity GPU programming environment for stencil computations on uniform grids [9, 10]. By extending this framework and adding the AMR data structure with halo exchange functions and mesh refinement mechanisms, we construct this AMR framework. The framework is implemented in the C++ language with CUDA and can be used in the user code is written just in C++, which improves portability of both framework and user code and facilitates cooperation with the existing codes. The framework provides data structure suitable for AMR method and class which can easily express stencil calculation on grid with various resolutions.

2 Overview of AMR framework

The proposed block-based AMR framework is designed to provide highly-productive programming environment for stencil applications with explicit time integration and adapting grid resolution to certain sensitive regions of simulations. The framework is intended to execute the user program on NVIDIA's GPU. The programmer can develop user programs just in the C++ language. The programmer simply describes a C++11 lambda expression that updates a grid point, which is applied to the entire grids with various resolution over a tree-based AMR data structure effectively.

The framework can locally change the resolution of the grids for arbitrary regions in the time integration loop of applications. An entire computational domain is divided into a large number of the small uniform grid blocks with the same size recursively. The computation for all grid blocks can be solved with a single execution of a conventional stencil calculation for Cartesian grid regardless of their resolutions. This strategy may be effective for performance improvement because GPU can often derive high performance when accessing contiguous memory. The framework also provides some functions and C++ classes to realize other processes required for the AMR computations, such as mesh refinement, exchanging data in halo regions between grid blocks with different resolutions, and data migration to maintain load balancing.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

3 Implementation and programming model of AMR framework

This section describes the implementation and programming model of this proposed framework.

3.1 Data structure for AMR framework

In order to realize AMR computations, this framework recursively divides a computational domain into a large number of uniform grid blocks and represent their spatial distributions by tree structures. Each leaf node of the tree structures has a uniform grid block per each physical variable. Each block contains the same number of cells regardless of the resolution to be expressed. A grid block, for example, contains 16^3 cells in 3D with halo regions, which size depends on the numerical schemes adopted by the application. Figure 1 shows a schematic diagram of the physical spatial distributions of grid blocks with trees and the memory space layout that holds the actual data. A quadtree or an octree tree is used as the tree structure in 2D or 3D, respectively. Since the GPU often achieves high performance when accessing consecutive memory areas, the grid blocks are allocated in one large contiguous memory area for each physical variable.

Each leaf node does not directly hold a grid block itself but holds an ID that specifies an assigned grid block. From these IDs, the position of the assigned grid block in the contiguous memory area can be determined. By based on ID mapping, a single tree structure can be associated with an arbitrary number of physical variables, which is important in developing a framework. Changing the positions and the number of grid blocks with time integration can be made only by changing the tree structure with varying the values of the IDs on the leaf nodes. It is unnecessary to allocate and deallocate the memory for grid blocks that may cause performance degradation especially on GPU. In order to express arbitrary shapes of the computational domains flexibly, the framework arranges multiple tree structures in an entire computational domain as shown in Figure 2.

In order to represent the AMR data structure by the multiple tree structures, this framework provides `Field` class, which is used as follows.

```
Field field(3, {4, 4, 8}); // dimension and size of trees
field.grow(2);             // grow all trees by 2 levels
```

In the multi-GPU computation, the entire program is parallelized by MPI and each process handles a single GPU. Each process independently holds the same tree structures as an object of the `Field` class at all times. The change of the tree structures, which means the change of the spatial resolution of the computational domain, is determined by (1) instructions to change mesh resolution and (2) instructions to migrate grid blocks between GPUs. Only the instructions in (1) and (2) are synchronized with MPI without explicit synchronization of the tree structures themselves. By sharing all the instructions in all processes, every process can change the own tree structures in the same way, which allows us

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

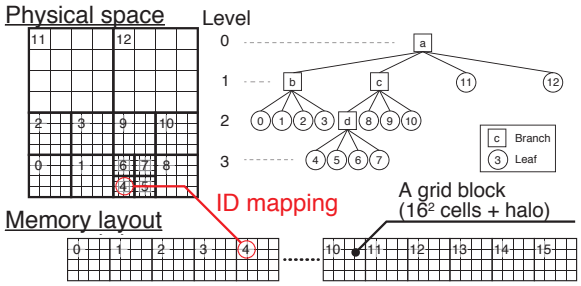

Fig. 1. AMR data structure.

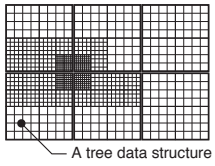

Fig. 2. Multiple trees to represent arbitrary shapes of the computational domains.

to always keep the same tree structures among all processes. In addition to the ID indicating the grid block, each leaf node holds a rank number of MPI that handles a GPU in which the grid block data are actually stored.

3.2 Array structure for multiple grid blocks

In order to represent the entire computational domain by a large number of grid blocks, the framework provides an unique data type `MArray` with `Range` type, which represents a 1D/2D/3D rectangular range. An object of `MArray` holds a single large array data, which is virtually divided into and used as multiple grid blocks as shown in Figure 1, with the number of grid blocks and one object of `Range`. By using these types, the multiple grid blocks are allocated as follows:

```
unsigned int length[] = {16+2*mgn, 16+2*mgn, 16+2*mgn};
int begin [] = {-mgn, -mgn, -mgn};
int narrays = 4096;            // number of grid blocks
Range3D whole(length, begin); // size of each grid block
MArray<float, Range3D> f(whole, narrays, MemoryType::DEVICE);
```

`MArray` is initialized with parameters that specify a `Range` that represents the range of a grid block, the number of grid blocks the `MArray` contains, and a location of memory to allocate. This `Range` object is used to determine the halo regions of each grid block. These grid blocks are also exploited as temporary areas for storing data used for mesh refinement and halo exchange with MPI.

3.3 Writing and executing stencil functions

In this framework, a stencil calculation must be defined as a C++11 lambda expression called a *stencil function* with `MArrayIndex` provided by the framework. The stencil function for 3D diffusion equation is defined as follows:

```
auto diffusion3d = [] __device__ (const MArrayIndex &idx,
  int level, float ce, float cw, float cn, float cs, float ct,
```

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

```
float cb, float cc, const float *f, float *fn) {
fn[idx.ix()] = cc*f[idx.ix()]
  + ce*f[idx.ix(1,0,0)] + cw*f[idx.ix(-1,0,0)]
  + cn*f[idx.ix(0,1,0)] + cs*f[idx.ix(0,-1,0)]
  + ct*f[idx.ix(0,0,1)] + cb*f[idx.ix(0,0,-1)]; }};
```

`MArrayIndex` holds the size of given grid block n^3 and represents a certain grid point (i, j, k), which is the coordinate of the point where this function is applied. It provides a function for accessing to the (i, j, k) point and its neighboring points for the stencil access; `idx.ix(-1, -2, 0)`, for example, returns the index representing $(i-1, j-2, k)$ point. Stencil functions can be defined as device (i.e., GPU) functions by using the qualifier `__device__` provided by CUDA.

To update `MArray` by the user-written stencil functions, the framework provides the `Engine` class, which is used to invoke the diffusion equation on the three-dimensional grid as follows:

```
Range3D inside; // where stencil functions are applied.
Engine_t engine;
engine.run(amrcontroller, inside, LevelGreaterEqual(1),
  diffusion3d, idx(f.range()), level(), ce,cw,cn,cs,ct,cb,cc,
  ptr(f), ptr(fn));
```

The parameters of `Engine::run` must begin with an object of `AMRController` that holds `Field` and another data structures required for AMR. The fourth parameter is a stencil function defined as a lambda expression, followed by any number of different types of additional parameters that are provided to this function. `f` and `fn` are `MArray` data. `Engine::run` applies a given stencil function to the grid blocks of the given `MArray` `fn` in the region represented by the second parameter `inside` and satisfying the condition for the AMR level given as the third parameter. Typically, `inside` specifies an inside region that is a region excluding the halo region from the computational domain as shown in Figure 3. By specifying `LevelGreaterEqual(1)`, this stencil function is applied to the grid blocks on level 1 or higher. The `ptr` function provides the pointer pointing to (i, j, k) of the given grid block in the `MArray` to the user-defined stencil function. Similarly, `level` is used to obtain the AMR level of the applied given block inside the stencil function, which allows us to perform level-dependent computation. Since the grid blocks are allocated in the contiguous memory area as described above, the framework can apply a single stencil function to all grid blocks at various levels that are contained inside a single `MArray` simultaneously.

3.4 Data transfer of halo regions

In this framework, each grid block on a leaf node has halo regions for stencil calculations. To advance the time step, it is necessary to exchange data in the halo regions between adjacent grid blocks with the same and different resolutions.

Data exchange of the halo regions inside a GPU is performed in the following order. First, data exchange of the halo regions is performed between adjacent

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

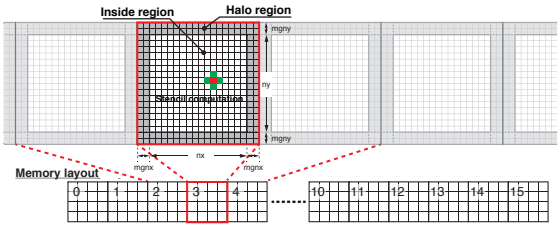

Fig. 3. Executing a stencil function with multiple grid blocks allocated in a large array.

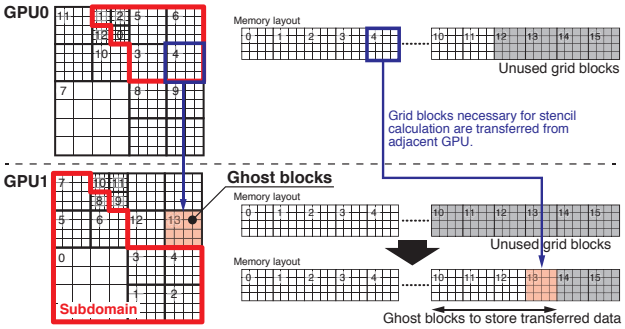

Fig. 4. Halo exchange between grid blocks allocated in the different GPUs.

grid blocks with the same resolution (i.e., the same level), which do not need the interpolation of values. Next, the data of the halo regions are transferred from the high-resolution grid blocks to the low-resolution grid blocks. Finally, the data of the halo regions are transferred from the low-resolution grid blocks to the high-resolution grid blocks. The framework can handle values defined at cell center and node center points. It can copy values at the same physical location between high- and low-resolution with interpolation functions. Currently, the interpolation values are calculated by a linear function.

Figure 4 shows exchanging data in the halo regions between the grid blocks allocated in the different GPUs. First, the framework designates several pieces of current unused grid blocks from the continuous memory area as temporary regions in each process. They are placed in the surround area of the subdomain of each process. These temporary grid blocks are called the ghost blocks in our framework. Next, the data in the grid blocks that are necessary for the stencil computation are actually transferred from the adjacent GPU using the CUDA APIs with MPI and stored in the ghost blocks. Referring to these ghost blocks, the stencil functions are executed at each process independently.

To execute the stencil computations, only the halo regions of the ghost blocks are required. However, in this framework, the whole regions of the ghost blocks are transferred between neighboring GPUs instead of the halo regions of them. In order to make full use of transferred data of grid blocks, we exploit a tempo-

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

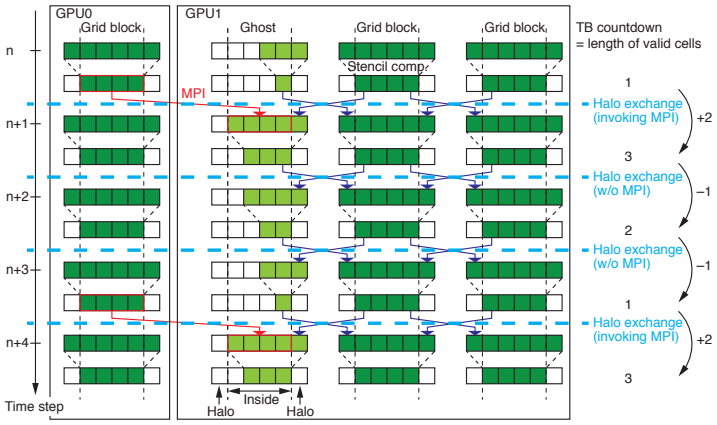

Fig. 5. Scheme of halo exchange using the temporal blocking with multiple GPUs. For the sake of simplicity, this figure is supposed to exchange halo regions at the same level.

ral blocking method that is a well-known technique for locality improvement in stencil computations [15, 12]. By using this method with several decomposed subdomains, several time steps can be advanced in each subdomain independently of the others. This also contributes to reducing the number of communications.

The framework exploits the temporal blocking based on the countdown proposed in our previous research [12]. Figure 5 shows the scheme of halo exchange using the temporal blocking with multiple GPUs. The number of executions of the function of halo exchange is counted. Based on this count, when it is expected that there will be no more effective data for performing the stencil calculation, actual communication will be carried out. Otherwise, the function of halo exchange does not perform any communication. As a result, the programmers can use the temporal blocking method without modifying their user codes.

In order to perform the halo exchange inside a GPU and between different GPUs with the temporal blocking method, this framework provides the `AMRController::exchange_halo`. This function is typically used as follows:

```
amrcontroller.exchange_halo(f, u, v, w);
```

`f`, `u`, `v` and `w` are `MArray` data. By using the C++11 variadic templates, this function can apply halo exchange between grid blocks to any number of different types of `MArray` data simultaneously. In this function, first inter-GPU communication with MPI is performed, which updates values on ghost blocks allocated each GPU. After that, inside each GPU, the halo exchanges between grid blocks including the ghost blocks are performed.

3.5 Mesh refinement

Modifying the resolution of the grid blocks on the leaf nodes is not done automatically on the framework side because it is necessary to take care of the change

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

of arbitrary physical quantities and variables in the user codes. To change the resolution of the grid blocks, the programmers explicitly specify the leaf nodes that having these grid blocks in the user code and issue the instructions of changing their resolutions by using the functions provided by the framework. These instructions issued to some leaf nodes in each process are shared by all processes before mesh refinement is actually executed.

After all instructions are shared by all processes, each process changes its own tree structure as follows. When a leaf node is specified to be fine resolution by an instruction for refining mesh, the framework forcibly raises its level by 1. To maintain a 2:1 balance of the resolution, the levels of its adjacent leaf nodes are also increased by 1 if necessary. When a leaf node is specified to be coarse resolution by an instruction, the framework decreases its level if it is able to continue to meet a 2:1 balance with its surrounding leaf nodes.

The resolution of the grid blocks is actually changed, after the new levels of the all leaf nodes after mesh refinement are determined on the tree structures. First, some of the unused grid blocks pooled in the continuous memory area are assigned to the grid blocks that store fine or coarse values after the mesh refinement. The framework assigns the grid blocks for this purpose in order from the smallest numeral to prevent fragmentation of memory. After that, the framework actually copies the values between grid blocks for the mesh refinement with interpolation in parallel. The unnecessary grid blocks that hold original values are returned to a group of the unused grid blocks for future use.

When several grid blocks with a high resolution that are not allocated on the same single GPU are changed to a single grid block with a low resolution, data migration is executed before mesh refinement in order to collect those original data on the same GPU.

3.6 Data Migration between GPUs and load balancing

In the AMR method, the sizes and the physical positions of local regions with high resolution change in the time integration loop of applications. The load balancing among GPUs using data migration is necessary to make efficient use of computational resources and improve performance.

This framework provides a function to issue an instruction to migrate grid blocks from a GPU to another GPU. When migrating a grid block, the programmer first issues this instruction with specifying a new process for the leaf node handling this grid block. All instructions issued in each process are shared by all processes with MPI Allreduce. After that, the framework actually performs the migration of the grid blocks using MPI according to these instructions. The some of the unused grid blocks are assigned to store the migrated grid blocks.

By using this migration mechanism, dynamic load balancing is realized as follows. In our framework, a computational domain is represented by multiple trees (Figure 2). While traversing trees in turn, the leaf nodes are assigned to each process in a depth-first search on each tree. The leaf nodes assigned to a certain process are typically owned by a few adjacent trees. By using this strategy, our applications can achieve localizing the distribution of the leaf nodes handled by

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

each process and load balancing of them. Localizing their distribution contributes to making inter-process communication more effective. In our application, when the number of leaf nodes assigned to a certain process increases by 10% compared to the average number of leaf nodes assigned to each process, the redistribution of all leaf nodes based on the migration described above is carried out.

4 Performance analysis and discussion

This section presents the performance of compressible flow simulation based on the proposed framework on a NVIDIA Tesla P100 GPU and its weak scaling results obtained on TSUBAME3.0. TSUBAME3.0 is equipped with 2,160 P100 GPUs. The peak performance of each GPU in double precision is 5.3 TFlops. Each node of it has four P100 attached to the PCI Express bus 3.0 $\times$16 (15.8 GB/s), four Intel Omni-Path Architecture HFI (12.5 GB/s) and two sockets of the Intel CPU Xeon E5-2680 V4 2.4 GHz 14-core.

4.1 Application: 3D Compressible Flow

We perform 3D compressible flow computation written by this framework and show computational results of the Rayleigh-Taylor instability. To simulate this, we solve 3D Euler equations described as follows:

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} + \frac{\partial G}{\partial z} = S, \quad U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix}, \; E = \begin{bmatrix} \rho u \\ \rho uu + p \\ \rho vu \\ \rho wu \\ (\rho e + p)u \end{bmatrix},$$
$$F = \begin{bmatrix} \rho v \\ \rho uv \\ \rho vv + p \\ \rho wv \\ (\rho e + p)v \end{bmatrix}, \; G = \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho ww + p \\ (\rho e + p)w \end{bmatrix}, \; S = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \rho g \\ \rho wg \end{bmatrix}, \tag{1}$$

where ρ is density, (u, v, w) are velocity, p is pressure, and e is energy. Here, g is gravitational acceleration. An advection term is solved using three-order upwind scheme with three-order TVD Runge-Kutta method. Time integration of five variables ρ, ρu, ρv, ρw, and ρe is solved, which requires 13 neighbor elements of each variable are used to update them on a center point of the grid.

4.2 Performance evaluation on single GPU

We show the performance results of the application on a single GPU by varying the size of the grid blocks assigned to leaf nodes. We change the number of cells that each grid block contains to 8^3, 12^3 or 16^3, and evaluate performance using 5 levels of resolution of AMR. In each grid block, halo regions having a width of

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

Table 1. Performance on a single NVIDIA Tesla P100 GPU.

# of cells	Equivalent domain size	# of leaves (level 1/2/3/4/5)	Kernels' performance (GFlops)	Overall performance (GFlops)
8^3	$512 \times 512 \times 2048$	180/449/705/2385/17208	765.2	102.4
12^3	$576 \times 576 \times 2304$	20/176/210/786/4848	811.9	178.1
16^3	$512 \times 512 \times 2048$	16/80/220/718/4752	913.9	245.9

2 are added around those cells in this simulation due to the adopted numerical schemes. The maximum width of a grid block is 16 times the minimum one in physical space on the 5-level AMR. When the number of cells each grid block contains is increased, the volume occupied by one grid block becomes large and it is difficult to finely adjust the resolution locally. Then, we set the maximum value of the length of one side of a grid block to 16 in this measurement.

Table 1 shows the total performance of computational kernels themselves and the overall performance of an entire time step at a certain time step when the number of cells in each grid block is changed. The 15 different computational kernels are executed at each time step. The entire time step includes computational times for exchange of the halo regions and control of the AMR structure as well as the above 15 kernels. These results are evaluated by using NVIDIA GPU profiler nvprof. This table also shows the number of leaves from the coarsest level 1 to the finest level 5. Note that the length of the whole computational domain needs to be a constant multiple of the length of a grid block in the current implementation. Then, when the number of cells each grid block contains is 12^3, the size of whole computational domain is different from others.

As shown in Table 1, comparing the total performance of the 15 kernels, the performance is higher when the length of one side of a grid block is longer. This is because when the volume of the halo regions with respect to that of the inside region decreases in each grid block, the memory access needed for updating values in the inside region is reduced, resulting in performance improvement. When the length of one side of a grid block is 16, the total performance of the kernels is 914 GFlops, which is 65% of the performance obtained by the same computation on the normal structure grid with the size of 128^3 (i.e., 1.41 TFlops). Considering that the ratio of the inside region to the entire computational region including the halo regions is 51% in each grid block and the cache can be used as part of the memory access, the ratio of 65% is considered appropriate.

In this framework, each grid block has halo regions so that the stencil functions for the structure grid can be used without any modification. However, the cost of exchanging these halo regions is relatively high. Due to this overhead, the observed overall performance decreases to 246 GFlops in the above case.

4.3 Time to solutions

We evaluate the computation time of two versions of simulation codes. The first version uses the temporal blocking method to reduce the number of communications and the second one does not exploit it. The latter version is used as

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

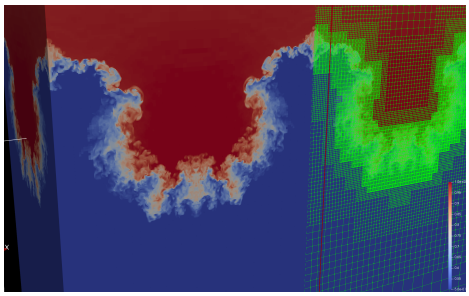

Fig. 6. A snapshot of density distribution results obtained by the simulation of 3D compressible flow. The boundary lines of the grid blocks are also shown in part.

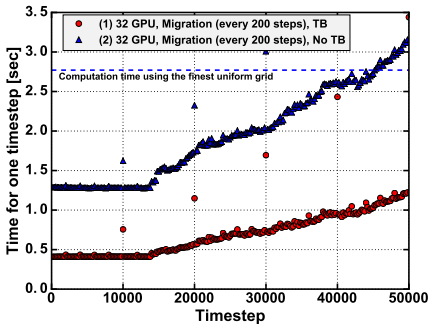

Fig. 7. Computational times for each time step using 32 GPUs.

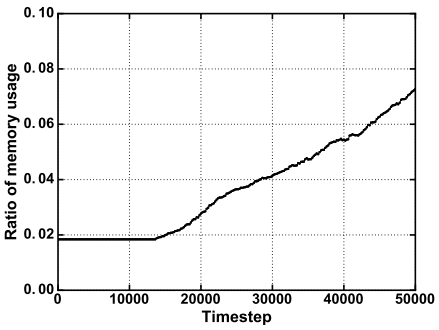

Fig. 8. Ratio of memory usage of AMR simulation for each time step in comparison with the computation on the finest grid.

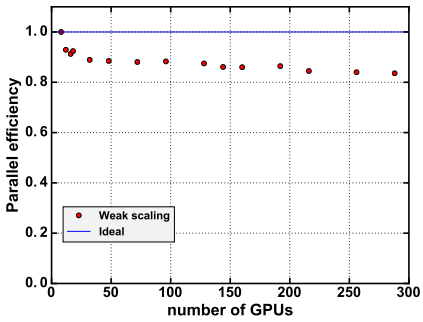

Fig. 9. Weak scaling on TSUBAME3.0.

references for this performance evaluation. Both versions may migrate data on grid blocks every 200 steps to improve load balancing if necessary. We perform simulations on a physical volume equivalent to the finest uniform grid with the size of $2,048 \times 1,024 \times 4,096$ using 5 levels of AMR by using 4 GPUs on each node and total 32 GPUs on TSUBAME 3.0. We use grid blocks having 16^3 cells with 2-width halo region from the results of the previous section.

Figure 6 shows a snapshot of computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this AMR framework. By applying the AMR method to fluid simulation, we have succeeded in simulating with a fine structure around the interface of two fluids.

Figure 7 shows the computation time taken for the calculation of each time step in the above two versions. At the 10,000th step, the first version takes 0.41 sec for the computation on this time step, while the second version takes 1.28 sec for the same computation. With the benefits of the framework, programmers can easily introduce the temporal blocking to this application and achieve approxi-

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

mately 3.1 times speedup without any additional development cost. When the finest uniform grid is used over entire computational domain instead of AMR, the computational time of 2.7 sec per each time step is required, which is depicted as a blue dashed line in Figure 7. It indicates that the first version is 6.7 times faster than the same computation on the finest uniform grid. Since the restart files are output every 10,000 steps, the computational times for every 10,000 steps is longer than the those required for other time steps.

Figure 8 shows the memory consumption ratio of this AMR simulation at each time step, compared to the simulation performed using the finest uniform grid over the physical volume having the same size. By using 5 levels of AMR, this memory consumption rate is kept to be less than 10% in overall runtime.

4.4 Weak scaling results

We show the weak scaling results of AMR applied simulation for the Rayleigh-Taylor instability using multiple GPUs on TSUBAME3.0. Figure 9 shows the performance results of the simulation using 5-level AMR with the temporal blocking method and the data migration to improve load balancing. We use 4 GPUs per each node for this simulation. We assign a physical volume equivalent to the finest uniform structure grid with the size of 1024^3 to each GPU. As shown in this figure, the weak scaling efficiency is above 84% for a physical volume equivalent to the finest uniform grid with the size of $6144 \times 6144 \times 8192$ on 288 GPUs with respect to the 8-GPU performance.

In order to further analyze the weak scaling results, Figure 10 shows the breakdown results of the computation time using 8 and 288 GPUs at the 1000th step. The computation time obtained by the stencil functions and the time taken by the halo exchange inside each GPU are almost the same in both figures. On the other hand, the communication time among GPUs with MPI is greatly affected by the number of GPUs to be used. Because of the complex geometry of the subdomains, each GPU needs to communicate with more number of GPUs in AMR than in the case of computation using a structure grid with multiple GPUs. When the number of GPUs used increases, the number of GPUs each GPU communicates increases, resulting that the communication time takes longer. In the refinement and data migration, MPI Allreduce is used to share the instructions among all processes to update the tree structures held by each process. As the number of GPUs increases, the communication between all processes increases, resulting in increasing the total computation time in one time step.

5 Conclusion

This paper has presented the programming model and implementation of the high-productivity framework for a block-based AMR for stencil applications, and evaluation of 3D compressible flow based on the proposed framework performed on a supercomputer equipped with multiple GPUs. The framework can

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

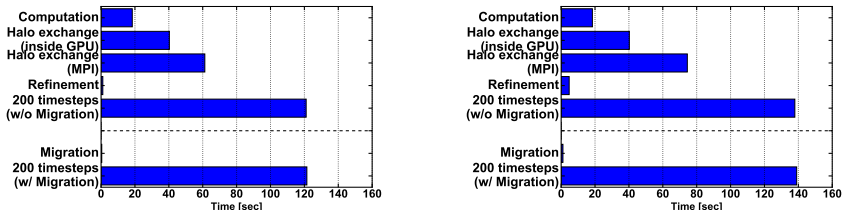

Fig. 10. Breakdown of the computation time at one time step using 8 GPUs (*left figure*) and 288 GPUs (*right figure*).

execute the user-written stencil functions that update a grid point on Cartesian grid over a tree-based AMR data structure effectively. This framework also provides mesh refinement mechanism and data migration that are required for AMR applications. The countdown based temporal blocking method, which is applied to the user codes without any modification, are contributes to reducing the number of communications and making full use of transferred data. With our proposed framework, we have conducted performance studies of the framework-based compressible flow simulation on a single GPU and using multiple GPUs on TSUBAME 3.0. The framework-based compressible flow simulation has achieved to reduce the computational time to less than 15% with 10% of memory footprint compared to the equivalent computation running on the fine uniform grid. The good weak scaling is obtained using 288 GPUs of TSUBAME 3.0 with the efficiency reaching 84%.

Acknowledgments

This research was supported in part by JSPS KAKENHI Grant Number JP17K00165, JP26220002 and in part by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" and "High Performance Computing Infrastructure" in Japan (Project ID: jh180061-NAH, jh180041-NAH).

References

1. Berger, M.J., Oliger, J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Journal of Computational Physics **53**, 484 (Mar 1984)
2. Christen, M., Schenk, O., Burkhart, H.: PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International. pp. 676–687 (2011)
3. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series **131**(1), 273 (2000)

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21

4. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: STELLA: A domain-specific tool for structured grid methods in weather and climate models. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 41:1–41:12. SC '15, ACM, New York, NY, USA (2015)
5. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 11:1–11:12. SC '11, ACM, New York, NY, USA (2011)
6. Schive, H.Y., Tsai, Y.C., Chiueh, T.: GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. The Astrophysical Journal Supplement Series **186**(2), 457 (2010)
7. Shimokawabe, T., Aoki, T., Ishida, J., Kawano, K., Muroi, C.: 145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction. Procedia Computer Science **4**, 1535 – 1544 (2011), proceedings of the International Conference on Computational Science, ICCS 2011
8. Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N., Matsuoka, S.: An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. SC '10, IEEE Computer Society, New Orleans, LA, USA (2010)
9. Shimokawabe, T., Aoki, T., Onodera, N.: High-productivity framework on GPU-rich supercomputers for operational weather prediction code ASUCA. In: Proceedings of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. SC '14, IEEE Computer Society, New Orleans, LA, USA (2014)
10. Shimokawabe, T., Aoki, T., Onodera, N.: High-productivity framework for large-scale GPU/CPU stencil applications. Procedia Computer Science **80**, 1646 – 1657 (2016)
11. Shimokawabe, T., Aoki, T., Takaki, T., Yamanaka, A., Nukada, A., Endo, T., Maruyama, N., Matsuoka, S.: Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In: Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. SC '11, ACM, Seattle, WA, USA (2011)
12. Shimokawabe, T., Endo, T., Onodera, N., Aoki, T.: A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 525–529 (Sept 2017)
13. Unat, D., Cai, X., Baden, S.B.: Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of the international conference on Supercomputing. pp. 214–224. ICS '11, ACM, New York, NY, USA (2011)
14. Wahib, M., Maruyama, N., Aoki, T.: Daino: A high-level framework for parallel and efficient AMR on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 53:1–53:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016)
15. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. pp. 30–44. PLDI '91, ACM, New York, NY, USA (1991)

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22734-0_21