Procedia Computer Science

Volume 80, 2016, Pages 1646–1657

ICCS 2016. The International Conference on Computational Science

High-productivity Framework for Large-scale GPU/CPU Stencil Applications

Takashi Shimokawabe[1], Takayuki Aoki[1], and Naoyuki Onodera[2]

[1] Tokyo Institute of Technology, Meguro, Tokyo, Japan
shimokawabe@sim.gsic.titech.ac.jp
[2] National Maritime Research Institute, Mitaka, Tokyo, Japan

Abstract
A high-productivity framework for multi-GPU and multi-CPU computation of stencil applications is proposed. Our framework is implemented in C++ and CUDA languages. It automatically translates user-written stencil functions that update a grid point and generates both GPU and CPU codes. The programmers write user code just in the C++ language, and can execute the translated user code on either multiple multicore CPUs or multiple GPUs with optimization. The user code can be executed on multiple GPUs with the auto-tuning mechanism and the overlapping method to hide communication cost by computation. It can be also executed on multiple CPUs with OpenMP. The compressible flow code on GPU exploiting the optimizations provided by the framework has achieved 2.7 times faster than the non-optimized version.

Keywords: Stencil applications, Framework, GPU

1 Introduction

Grid-based physical simulations using stencil simulations such as weather prediction code and compressible flow are one of the important applications running on supercomputers. Thanks to high memory bandwidth of GPU, GPU computing has successfully accelerated various stencil applications [7, 6, 9, 3]. However, it forces the programmer to learn multiple distinctive programming models such as CUDA and introduce various complicated optimizations for multi-GPU computation especially to achieve high performance as expected. To improve programmer productivity and achieve high performance with low development cost, various types of high-level programming models were proposed [5, 10, 1, 2, 4]. STELLA was proposed as a domain-specific tool for stencil computations on structured grids in weather and climate models [4].

We are currently developing a high-productivity framework for multi-GPU and multi-CPU computation of stencil applications. The framework has been originally developed for the weather prediction code ASUCA running on multiple GPUs [8]. The framework is implemented in the C++ and CUDA languages and automatically translates user-written functions that

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.499

update a grid point and generates both GPU and CPU code. The programmer writes user's code just in C++ and can develop program code optimized for GPU supercomputers without introducing complicated optimizations for GPU computation and GPU-GPU communication.

In this paper, further optimizations for multi-GPU computation introduced into our framework are presented. The performance of the GPU computation of stencil applications is highly dependent on the run-time parameters such as the number of threads used in a calculation. A mechanism for automatically selecting the optimum parameters at run time is introduced into this framework. The framework provides overlapping technique with auto-tuning to hide communication overhead by computation. In order to improve productivity, the framework introduces the array-based data structure that supports element-wise computations. The user codes capable of GPU execution can be also accelerated on multiple CPUs with OpenMP by using the presented framework in this paper, resulting in reduced maintenance cost of applications. We show the performance results of diffusion computation and 3D compressible flow written by the proposed framework. TSUBAME2.5 at Tokyo Institute of Technology is used for these measurements.

2 Overview and Basic Design of Framework

This section reviews the basic design of the proposed framework. The original version of this framework was presented in our previous paper [8], which was developed for the weather prediction code ASUCA running on multiple GPUs.

The proposed framework is designed for stencil applications with explicit time integration running on regular structured grids. The framework is written in the C++ language and CUDA and can be used in the user code developed in C++, which improves portability of both framework and user code and facilitates cooperation with the existing codes. The framework is intended to execute a single user program on either multiple NVIDIA's GPUs or multiple x86 CPUs. To perform stencil computations on grids, the programmer only defines C++ functors that update a grid point, which are applied to entire grids by the framework. A user code is parallelized by MPI with OpenMP for CPU and by MPI with CUDA for GPU. The framework provides optimizations suitable for large-scale GPU computing such as overlapping methods and auto-tuning mechanism. The programmer can develop the program code optimized for both multi-CPU and multi-GPU computing without introducing complicated optimizations explicitly.

3 Programming Model and Implementation

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

3.1 Data structure for grid data

The framework originally exploits arrays of C/C++ languages as array data types. To further improve productivity, an unique data type `ETArray` and `Range` type, which represents a 1D/2D/3D rectangular ranges, are introduced into the framework. An object of `ETArray` holds an array data with one object of `Range` that represents its size and location. By using these types, the array data are allocated as follows:

```
unsigned int length[] = {nx+2*mgnx, ny+2*mgny, nz+2*mgnz};
int          begin [] = {-mgnx, -mgny, -mgnz};
```

```
Range3D whole(length, begin);
ETArray<float, Range3D> f_h(whole, MemoryType::HOST_MEMORY);
ETArray<float, Range3D> f_d(whole, MemoryType::DEVICE_MEMORY);
```

`ETArray` is initialized with parameters that specify a `Range` that represents the range of itself and a location of memory to allocate. Currently regular pageable host memory, page-locked host memory and device memory (i.e., GPU) are supported as the memory location.

3.2 Writing and executing stencil functions

In this framework, a stencil calculation must be defined as a C ++ functor called a *stencil function* with `ArrayIndex` provided by the framework. The stencil function for three-dimensional diffusion equation is defined as follows:

```
struct Diffusion3d {
__host__ __device__ float operator()(const float *f, const ArrayIndex &idx,
    float ce, float cw, float cn, float cs, float ct, float cb, float cc) {
    const float fn = 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)];
    return fn;
}};
```

`ArrayIndex` holds the size of given grid (n_x, n_y, n_z) and represents a certain grid point (i, j, k), which is the coordinate of the point where this function is applied. `ArrayIndex` with (n_x, n_y, n_z) is intended to be used for the `ETArray` having the same sizes. It provides a function for accessing to the (i, j, k) point and its neighboring points for the stencil access; `idx.ix()` and `idx.ix(-1, -2, 0)`, for example, returns the indexes representing (i, j, k) and $(i-1, j-2, k)$ points, respectively. Stencil functions can be defined as both host and device (i.e., GPU) functions by using the qualifiers `__host__` and `__device__` provided by CUDA.

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

```
Range3D inside; // 3D rectangular range where stencil functions are applied.
ETProperty *prop = new DeviceProperty; // Select GPU execution
view(fn, inside, prop)
    = funcf<float>(Diffusion3d(), ptr(f), idx(f), ce, cw, cn, cs, ct, cb, cc);
```

`f` and `fn` are ETArray data. The `funcf` function is defined as a C++ template function that takes an arbitrary number of different types of arguments. By using the C++ type inference, `funcf` calls an appropriate functor `Diffusion3d` and provides the given all the arguments to it. The `ptr` function provides the pointer pointing to (i, j, k) of the given `ETArray` to the stencil function. Texture cache is exploited to access these input pointers in the stencil functions on GPU. The `view` function executes a stencil function on the right-hand side of the assignment expression and applies the stencil function to the grid points of the given `ETArray fn` in the `inside` region. In typical stencil computations, the `inside` region is a region excluding the halo region from the whole computational domain. `ETProperty`, which is given to the `view` function, is also provided by the framework and is used to specify how to run the stencil function. Table 1 shows the family of `ETProperty` provided by the framework. The programmer can execute stencil functions on CPU with OpenMP and GPU with some optimizations such as auto-tuned kernel execution and an overlapping method by just specifying an appropriate property at `view`. Note that the programmer must call stencil functions by one of the host-execution `ETProperty`

Table 1: `ETProperty` family

ETProperty	Execution
HostProperty	Host (single thread) execution
OmpProperty	OpenMP execution on host
DeviceProperty	Device (GPU) execution
DeviceATProperty	Device execution with auto-tuning
DeviceOverlapProperty	Device execution with overlapping method
DeviceOverlapATProperty	Device execution with overlapping method and auto-tuning

with `ETArray` objects allocated on host, or by one of the device-execution `ETProperty` with those on device in order to perform them correctly. The details of overlapping technique and auto-tuning mechanism are described later.

3.3 Element-wise computation for arrays

Stencil applications often use element-wise computations as well as stencil computations. By using the C++ technique called expression templates, each expression related to `ETArray` generates GPU and CPU kernel codes corresponding to the expression itself, which allows us to write kernel codes as inline codes and achieve a well simple description. Examples of element-wise computations with `ETArray` are shown as follows:

```
// The following expressions are written in inline host codes
// instead of stencil functions.
f = 1.0 + 2 * 4.0; // ‘‘f’’ is ETArray. All elements of ‘‘f’’ are filled with 9.0.
g = 2.0 * sqrt(f); // ‘‘g’’ is ETArray. All elements of ‘‘g’’ are filled with 6.0.
h = sqrt((pos(axis0)-3)*(pos(axis0)-3) + (pos(axis1)-5) * (pos(axis1)-5));
```

`f`, `g` and `h` are `ETArray`. The framework provides element-wise mathematical functions. An element-wise square root computation for `f` is used above. The framework also provides the function `pos` for acquiring coordinates such as i, j and k, and objects `axis0` and `axis1` to specify x and y axes, which allows us to write and generate inline kernels using coordinates. For example, `h` will have the distance from the point $(3, 5)$ in two dimensions.

Using the operator overloading of C++, arithmetic operators corresponding to the classes provided by the framework such as `ETArray` and `pos` are provided. When an expression includes objects of these classes as the operands of the operators, a structure representing the expression is built at compile time by using the techniques of C++ expression template. When the right-hand side of the assignment operator of an `ETArray` allocated on the device memory receives this structure, this `ETArray` and the structure are transferred to GPU and then the expression represented by this structure is executed on GPU. The function `funcf` for writing stencil functions described above also supports this mechanism based on the expression template.

3.4 Data transfer of halo regions between subdomains

In stencil simulations using multiple GPUs or multiple CPUs, domain decomposition method is often used for parallelization of these. Figure 1 shows the domain decomposition method. Data exchanges of halo regions of subdomains are performed frequently since stencil computation that updates a point needs to access its neighboring points. This framework provides the `PBoundaryExchange` class to easily describe the communication to exchange these halo regions. `PBoundaryExchange` is typically used as follows:

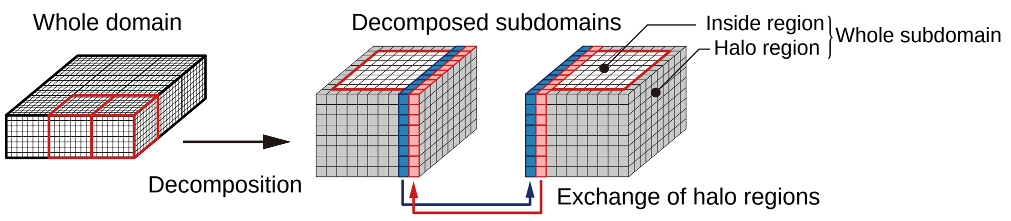

Figure 1: Multiple GPUs and CPUs computation for stencil applications.

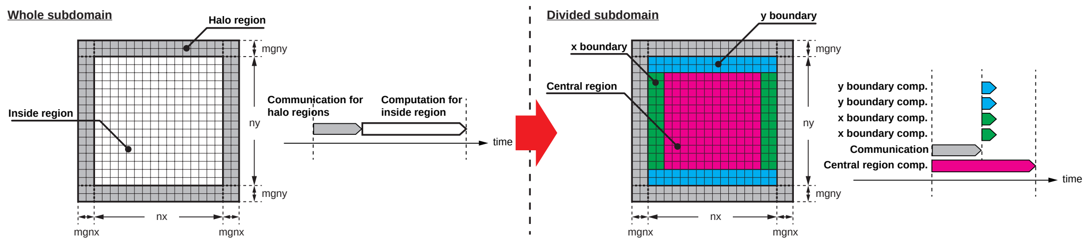

Figure 2: Scheme of the non-overlapping and the kernel-division overlapping method.

```
PBoundaryExchange exchange(inside, rank, neighbor_connect);
exchange.append(f1);
exchange.append(f2);
exchange.transfer();
```

`PBoundaryExchange` is initialized by `inside`, which is a `Range` object, and MPI rank information with neighbor subdomain ranks. Halo regions of `ETArray` objects appended by `PBoundaryExchange::append` are exchanged using MPI with CUDA Runtime API when `PBoundaryExchange::transfer` is executed. `PBoundaryExchange` may allocate temporary buffers on host memory when `ETArray` that `PBoundaryExchange` receives is allocated on GPU.

3.5 Overloading method for multi-GPU computation

In large-scale GPU computation, the communication time between GPUs cannot be ignored in comparison with the total execution time of applications. The overlapping technique to hide the communication costs by the calculation contributes to the performance improvement.

This framework provides *the kernel-division overlapping method* for multi-GPU computing reported in our previous work [7]. This method exploits the data independence in a single variable. Since each element of a variable can be calculated independently with the other elements, computations for the boundary regions that depend on halo data can be performed separately from other calculations for the remaining central region of the subdomain. Figure 2 shows the overlapping method provided by this framework. As shown in the figure, in the two-dimensional domain decomposition case, when each of computational subdomains requires two-element-thick mesh as halo regions, the framework divides one inside region into four boundary regions with two-element-thick mesh and the remaining central region. The framework divides a single execution of a stencil function on the whole inside region into five executions of it on these five divided regions. In the overlapping method on multiple GPUs, first, the framework updates the values in the central region by executing a user-written stencil function in a CUDA stream, while simultaneously the boundary exchanges between subdomains to update the values in the halo regions are performed in another stream. The boundary exchange consists of asynchronous memory copy from GPU to CPU executed by a CUDA memory operation, data exchanges between CPUs with MPI, and asynchronous memory copy from CPU to GPU. After this copy sequence finishes, the user-written stencil function is executed to update the values in the four boundary regions in four different streams, which contributes to exploiting computational resources effectively.

This framework provides the overlap technique described above that corresponds to the `ETArray`. The diffusion calculation using the overlapping method is written as follows:

```
ETProperty *prop = new DeviceOverlapProperty(Range3D::ndim);
vieweb(fn, &exchange, prop)
    = funcf<float>(Diffusion3d(), ptr(f), idx(f), ce, cw, cn, cs, ct, cb, cc);
```

This code provides the same result as the following reference code, which does not use the overlapping method:

```
ETProperty *prop = new DeviceProperty;
exchanges.transfer();
view(fn, exchanges.inside_region(), prop)
    = funcf<float>(Diffusion3d(), ptr(f), idx(f), ce, cw, cn, cs, ct, cb, cc);
```

The `vieweb` function is the `view` function receiving `PBoundaryExchange` object that handles the boundary exchanges. The `vieweb` function executes a stencil function and communication in parallel described above. The `prop` is used not only to specify how to run the stencil function, but also to provide information and working resources required for the execution; for example, `DeviceOverlapProperty` enables the overlapping method and provides CUDA streams for parallel execution.

3.6 Auto-tuning for single GPU computation

In this framework, three-dimensional stencil functions are invoked with a two-dimensional CUDA thread block that is arranged parallel to the xy plane. Each thread specifies a point (i, j) and calculates consecutive grid points marching in the z direction. Since the performance of GPU is often improved for large numbers of threads, the framework divides a subdomain assigned to each GPU into several pieces of subdomain in the z direction and assigns one CUDA thread block to each piece.

In the stencil calculation by GPU, the performance is largely dependent on the run-time parameters such as the number of CUDA threads and the number of divisions in the z direction. This framework provides an automatic tuning mechanism for these parameters; the framework selects the optimum parameters of the number of threads in x and y directions and the number of grid points marching in z direction. Table 2 shows the auto-tuned parameters in this mechanism. The programmer can apply this auto-tuning to user-written stencil functions by just specifying `DeviceATProperty` or `DeviceOverlapATProperty` shown in Table 1 at `view`. The programmer does not need to provide any other information at runtime. `DeviceATProperty` is typically used as follows:

```
ETProperty *prop = new DeviceATProperty;
view(fn, inside, prop)
    = funcf<float>(Diffusion3d(), ptr(f), idx(f), ce, cw, cn, cs, ct, cb, cc);
```

Table 2: `ETProperty` family

Number of threads in x direction	4, 8, 16, 32, 64, 128
Number of threads in y direction	1, 2, 4, 8, 16
Number of mesh in z-marching	1, 2, 4, 8, 16

When a stencil function by `view` with `DeviceATProperty` is executed, a new set of the parameters is specified at each execution and the execution time with this set is measured and recorded. After the measurement of execution time in all parameters is finished, the stencil function is performed with the optimal parameters to minimize the execution time. The auto-tuning is performed during early stage of time integration loop of applications. Although performance degradation is observed while the tuning is performed, the increase of the execution time due to this performance degradation can be ignored in comparison with the entire execution time of applications since each stencil function is typically performed more than tens of thousands of times in applications. `DeviceOverlapATProperty` provides the auto-tuning mechanism with the overlapping method.

4 Performance Evaluation and Discussion

In this section, we show the performance evaluation of the proposed framework. We apply the framework to a diffusion equation, which is a fundamental equation used in fluid simulations, and a 3D compressible flow application simulating the Rayleigh-Taylor instability.

We use the TSUBAME 2.5 supercomputer at the Tokyo Institute of Technology for our evaluation. The TSUBAME 2.5 supercomputer is equipped with 4224 NVIDIA Tesla K20X GPUs. The peak performance of each GPU in single precision is 3.95 TFlops. The on-board device memory (also called global memory in CUDA) provides 250 GB/s peak bandwidth in a Tesla K20X. Each node of TSUBAME 2.5 has three Tesla K20X attached to the PCI Express bus 2.0 $\times 16$ (8 GB/s), two QDR InfiniBand and two sockets of the Intel CPU Xeon X5670 (Westmere-EP) 2.93 GHz 6-core. All calculations in this section are done in single precision.

In order to measure the performance of an application, we count the number of floating-point operations in its simulation code by running it on a GPU with NVIDIA profiler. Using the obtained count and the elapsed time of the application, its performance is evaluated.

4.1 Diffusion equation

A diffusion equation is well used in physical simulations such as computational fluid dynamics and written as follows:

$$\frac{\partial f}{\partial t} = \kappa \nabla^2 f, \tag{1}$$

where f is a physical variable and κ is a diffusion coefficient. The physical variable on a certain point of a grid is updated by seven neighbor elements of the physical variable around that point in 3D. The boundary regions that have one-element-thick are needed for this computation.

Figure 3 shows the performance results of diffusion computation on a single GPU in the cases using typical fixed number of threads with typical fixed number of divisions in z direction and the case using the auto-tuning mechanism. As shown in the figure, the performance obtained by the auto-tuning mechanism achieves the fastest in these results for all mesh sizes; the performance of 215.3 GFlops is achieved by using the auto-tuning mechanism for a 512^3 mesh. Assuming that

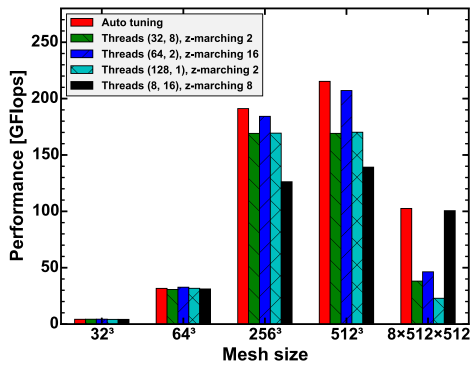

Figure 3: Performance results of diffusion computation with auto-tuning running on a Tesla K20X.

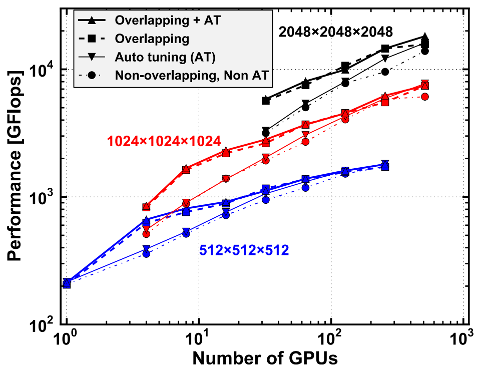

Figure 4: Strong scaling results of diffusion computation on multiple GPUs of TSUBAME2.5.

this computation is a memory-bound one and utilizes the peak bandwidth of GPU memory (i.e., 250 GB/s), the attainable performance is estimated to be 102.5 GFlops. Since the performance of 215.3 GFlops is achieved by using the auto-tuning, which is more than the estimated value, this auto-tuned stencil function is likely to be well optimized. Although the computation invoked with $(64, 2)$ threads and 16-element z marching, which is used for invoking non-tuned GPU kernels generated by the framework, achieves higher performance for almost all mesh sizes than other computations, that for a $8 \times 512 \times 512$ mesh remains at 46.4 GFlops since the number of threads in the x direction is far from the mesh size in that direction. The proposed auto-tuning mechanism overcomes this performance degradation and achieves 102.6 GFlops.

Figure 4 shows the strong scaling results of diffusion equation running on GPUs of TSUBAME2.5. The performance results of computations using the overlapping method or the auto-tuning mechanism or both are shown in this figure. The performance of non-optimized computation is also shown in the figure. We use three GPUs per each node for these calculations and vary the mesh size from 512^3 to 2048^3. We have chosen 2D decomposition for this computation since 3D decomposition, which looks better to reduce communication amount, tends to degrade GPU performance due to complicated memory access patterns for data exchanges between GPUs. As shown in this figure, the overlapping method contributes to the significant performance improvements by hiding communication overhead. The auto-tuning mechanism is considered to improve the single GPU performance, resulting that the overall multi-GPU performance is slightly improved in all mesh sizes. In the performance results using a 1024^3 mesh on 64 GPUs, for example, the overlapping method with the auto-tuning mechanism, the overlapping method only and the auto-tuning mechanism only achieve 3.78 TFlops, 3.70 TFlops and 3.06 TFlops, respectively while the non-optimized version reaches 2.70 TFlops.

4.2 Compressible Flow

We perform 3D compressible flow computation written by this framework and show computational results of the Rayleigh-Taylor instability, which is a more realistic simulation than

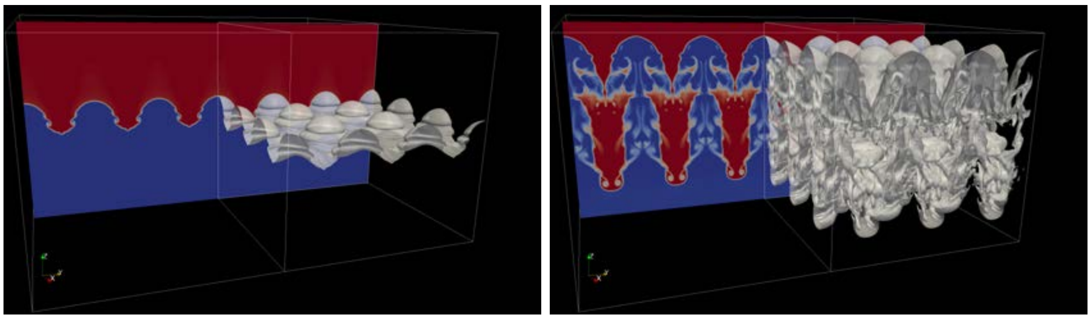

Figure 5: Simulation results of the Rayleigh-Taylor instability.

diffusion one. To simulate this, we solve 3D Euler equations described as follows:

$$\frac{\partial U}{\partial t}+\frac{\partial E}{\partial x}+\frac{\partial F}{\partial y}+\frac{\partial G}{\partial z}=S, \tag{2}$$

$$U=\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ \rho e\end{bmatrix}, E=\begin{bmatrix}\rho u\\ \rho uu+p\\ \rho vu\\ \rho wu\\ (\rho e+p)u\end{bmatrix}, F=\begin{bmatrix}\rho v\\ \rho uv\\ \rho vv+p\\ \rho wv\\ (\rho e+p)v\end{bmatrix}, G=\begin{bmatrix}\rho w\\ \rho uw\\ \rho vw\\ \rho ww+p\\ (\rho e+p)w\end{bmatrix}, S=\begin{bmatrix}0\\ 0\\ 0\\ \rho g\\ \rho wg\end{bmatrix},$$

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. In this simulation, time integration of five variables ρ, ρu, ρv, ρw, and ρe is solved. Each variable uses 4 `ETArray` objects to update itself in the time integration in our implementation. Updating values of the five variables on a center point needs to use 13 neighbor elements of these variables. The boundary regions that have two-element-thick are needed for this computation.

Figure 5 shows computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation on a $576 \times 576 \times 288$ mesh written by this framework. We use 64 GPUs of TSUBAME2.5 for this calculation. In this figure, red and blue region fill on the yz plane represent two fluids of different densities and white contour that is partly depicted represents the an interface between them.

Figure 6 shows the strong scaling results of compressible flow running on GPUs of TSUBAME2.5. Similar to Figure 4, the performance results of optimized and non-optimized versions are shown in this figure. We use three GPUs per each node for these calculations and vary the mesh size from 256^3 to 1024^3. Similar to diffusion computation, we have chosen 2D decomposition for this calculation. This compressible fluid computation needs five independent variables described above. Our implementation exploits five different stencil functions to update these variables. Since each of these stencil functions in this simulation often takes longer computation time than one in the diffusion calculation, the proportion of communication time to the entire execution time is considered to be smaller than one in the diffusion calculation. Thus the effect of the overlapping method on the performance improvements in this application is smaller than that in the diffusion computation as shown in this figure. On the other hand, since the number of threads $(64, 2)$ with 16-element z marching, which is used for non-optimized GPU kernels, is not suitable for these stencil functions unlike diffusion computation, the auto-tuning contributes to significant performance improvements shown in this figure. The auto-tuning mechanism, for

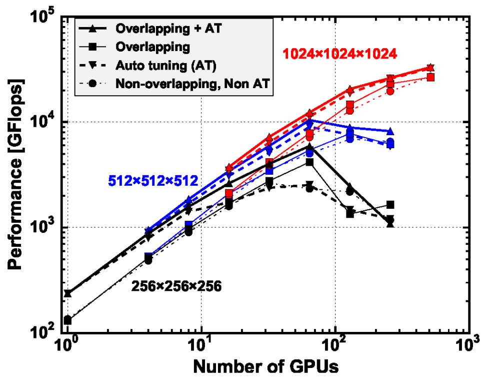

Figure 6: Strong scaling results of 3D compressible flow on multiple GPUs of TSUBAME2.5.

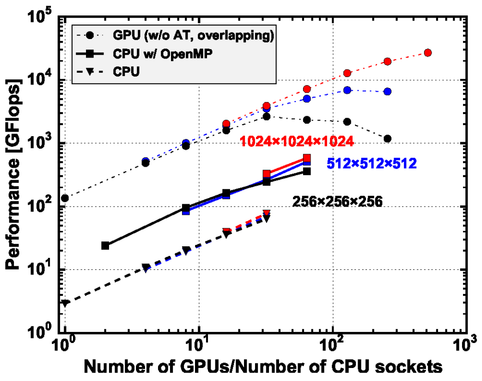

Figure 7: Strong scaling results on multiple CPUs of TSUBAME2.5. The performance of GPU computation without optimization is shown as a reference.

example, selects the number of threads (32, 2) with 2-element z marching for all five stencil functions in the case of using a 512^3 mesh on 64 GPUs.

The amount of computation on GPU is in proportion to the volume of the subdomain assigned to the GPU, while the amount of communication is in proportion to the surface area of this subdomain. When the number of GPUs used is larger, the volume that each GPU handles becomes smaller, resulting in an increase of the proportion of communication time to the entire execution time. Thus, the overlapping method becomes of benefit in the performance improvements when a large number of GPUs are used. For example, in the performance results using a 256^3 mesh on 8 GPUs, the overlapping method with the auto-tuning and the auto-tuning version achieve 1.57 TFlops and 1.41 TFlops, respectively while the overlapping method only version and the non-optimized version reach 0.971 TFlops and 0.902 TFlops, respectively. On the other hand, by using 64 GPUs for a 256^3 mesh, the overlapping method greatly contributes to the performance improvements, and the overlapping method with the auto-tuning and the overlapping method only achieve 5.93 TFlops and 4.17 TFlops, respectively while the auto-tuning version reaches 2.54 TFlops. The performance of 5.93 TFlops is 2.7 times faster than that of 2.33 TFlops obtained by the non-optimized version.

Figure 7 shows the performance results of compressible flow running on multiple CPUs. We use 1 CPU core per each node and 2 CPU sockets (i.e., 12 cores) per each node for these computations. In the latter case, OpenMP is used for parallelization on each node. The performance results of non-optimized GPU version are also shown in this figure as a reference. Note that all these calculations on GPU and CPU use the same single code. OpenMP parallelization provided in the framework works well, and the performance of 330 GFlops is achieved by 32 CPU sockets with OpenMP for a 1024^3 mesh, which result is 4.2 times faster than the result using 1 CPU core per each node.

5 Conclusion

We have developed the high-productivity framework for multi-GPU and multi-CPU computation of stencil applications. From the viewpoint of portability of both framework and user code and facilitating cooperation with the existing codes, the framework is implemented in the

C++ and CUDA languages and the user code can be written in the C++ language. Further optimizations for multi-GPU computation are introduced into our framework. A mechanism for automatically selecting the optimum parameters at run time is introduced into this framework. The framework provides optimizations for multi-GPU computing such as the overlapping technique with auto-tuning mechanism to hide communication overhead by computation, resulting in achieving optimal performance. The user code capable of GPU execution can be also executed on multiple CPUs with OpenMP without any change, resulting in reduced maintenance cost of applications. We have demonstrated the results of diffusion computation and compressible flow written by the framework. The optimized GPU version of compressible flow using the overlapping method with auto-tuning mechanism has achieved 2.7 times faster than the non-optimized GPU code.

6 Acknowledgments

This research was supported in part by KAKENHI, Grant-in-Aid for Young Scientists (B) 15K20995 and Grant-in-Aid for Scientific Research (S) 26220002 from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan, and in part by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" and "High Performance Computing Infrastructure" in Japan.

References

[1] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, 2011.

[2] Matthias Christen, Olaf Schenk, and Yifeng Cui. Patus for convenient high-performance stencils: Evaluation in earthquake simulations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 11:1–11:10, Salt Lake City, Utah, 2012. IEEE Computer Society Press.

[3] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, and G. Wellein. A Flexible Patch-Based Lattice Boltzmann Parallelization Approach for Heterogeneous GPU–CPU Clusters. *Parallel Computing*, 37(9):536–549, 2011.

[4] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. 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*, SC '15, pages 41:1–41:12, New York, NY, USA, 2015. ACM.

[5] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 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*, SC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[6] Takashi Shimokawabe, Takayuki Aoki, Junichi Ishida, Kohei Kawano, and Chiashi Muroi. 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.

[7] Takashi Shimokawabe, Takayuki Aoki, Chiashi Muroi, Junichi Ishida, Kohei Kawano, Toshio Endo, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. 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*, SC '10, pages 1–11, New Orleans, LA, USA, 2010. IEEE Computer Society.

[8] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. 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*, SC '14, pages 1–11, New Orleans, LA, USA, 2014. IEEE Computer Society.

[9] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Akinori Yamanaka, Akira Nukada, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka. 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*, SC '11, pages 1–11, Seattle, WA, USA, 2011. ACM.

[10] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.