High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA

Takashi Shimokawabe
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan
Email: shimokawabe@sim.gsic.titech.ac.jp

Takayuki Aoki
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan

Naoyuki Onodera
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan

***Abstract*—The weather prediction code demands large computational performance to achieve fast and high-resolution simulations. Skillful programming techniques are required for obtaining good parallel efficiency on GPU supercomputers. Our framework-based weather prediction code ASUCA has achieved good scalability with hiding complicated implementation and optimizations required for distributed GPUs, contributing to increasing the maintainability; ASUCA is a next-generation high-resolution meso-scale atmospheric model being developed by the Japan Meteorological Agency. Our framework automatically translates user-written stencil functions that update grid points and generates both GPU and CPU codes. User-written codes are parallelized by MPI with intra-node GPU peer-to-peer direct access. These codes can easily utilize optimizations such as overlapping technique to hide communication overhead by computation. Our simulations on the GPU-rich supercomputer TSUBAME 2.5 at the Tokyo Institute of Technology have demonstrated good strong and weak scalability achieving 209.6 TFlops in single precision for our largest model using 4,108 NVIDIA K20X GPUs.**

I. Introduction

Weather forecasting is essential for our lives and businesses. In recent years, numerical weather prediction demands fast and high-precision simulation over fine-grained grid more and more especially from the perspective of natural disaster prevention.

Large-scale weather simulation has been an active area of research for many years. In 2002, researchers at the Japan Marine Science and Technology Center (JAMSTEC) achieved the sustained performance 26.58 TFlops for AFES, which is a spectral atmospheric general circulation model, by using Earth Simulator [1]. This work was awarded the Gordon Bell Prize in the same year. In 2009, a benchmark test of the Weather Research Forecasting (WRF) model [2] achieved $50\,\mathrm{TFlops}$ by using the entire Jaguar Cray XT5 system at the Oak Ridge National Laboratory [3].

Recently, exploiting accelerators, including GPUs, along with conventional CPUs on supercomputers has emerged as an effective way to achieve high performance with relatively-low power consumption. Supercomputers such as Titan at the Oak Ridge National Laboratory and TSUBAME 2.5 at the Tokyo Institute of Technology are equipped with large number of GPUs. It is well known that the advantages of GPU in both computation power and wide memory bandwidth allow various scientific simulations, such as computational fluid dynamics[4], [5], and astrophysical N-body simulations[6], and phase-field simulations[7], to achieve high performance. In the field of numerical weather prediction, a computationally expensive physics module of the WRF model was accelerated by using a GPU [8], [9]. In the case of the WRF Single Moment 5-tracer (WSM5) microphysics, twenty-fold speedup was achieved. By using large-scale of GPUs, the weather prediction code ASUCA was accelerated by using multiple GPUs and achieved $145.0\,\mathrm{TFlops}$ on TSUBAME 2.0 in our previous research [10], [11]. WRF model achieved $285\,\mathrm{TFlops}$ by using NVIDIA Kepler GPUs of the Cray XE6 "Blue Waters" at NCSA at the University of Illinois [12].

Although various applications are accelerated by GPUs, programming on different types of devices by using low level platform-specific programming languages such as CUDA [13] that is specific to NVIDIA GPUs forces the programmer to learn multiple distinctive programming models especially to achieve high performance as expected. To solve this problem and improve programmer productivity, various types of high-level programming models were proposed. OpenACC is a directives-based parallel-programming standard designed to enable programmers to accelerate their codes on heterogeneous CPU/GPU computing systems [14]. Mint was proposed as a high-level framework specialized for stencil computations on CUDA-enabled GPUs [15]. As another example, Physis was proposed as a high-level programming framework based on a domain-specific language (DSL) for large-scale GPU computation specialized to stencil computations with regular multidimensional Cartesian grids [16]. PATUS was proposed as a code generation and auto-tuning DSL for stencil computations targeted at multi- and many-core processors [17], [18].

We are currently working on full GPU implementation for ASUCA [19] – a next-generation high resolution meso-scale atmospheric model being developed by the Japan Meteorological Agency. In previous work [10], [11], we have successfully implemented its dynamical core and a portion of physics processes as a full GPU application. Through implementing this on GPU, multi-GPU computation of mesh-based applications, including weather prediction codes, has the potential to achieve high performance. However, programming for large-scale parallel computing on GPU-rich supercomput-

ers is a more difficult task than programming for a single device since these applications need to introduce optimizations for communication along with single-GPU optimizations, such as the overlapping methods to hide communication overhead by computation reported in our previous research [10], [7]. To apply these complicated optimizations to various mesh-based applications including ASUCA easily, we have developed high-productivity and high-portability framework for multi-GPU computation of mesh-based applications, and implemented ASUCA based on this proposed framework from scratch. Since part of existing codes and external existing libraries are often used for development of real applications, the framework should be designed to have the capability of the cooperation with these existing codes. In addition, in order to enhance extensibility and portability of user codes with the framework, they should be written in standard languages without using non-standard programming models and language extension. Thus, unlike previous research, the proposed framework can be used in the user code developed in the C++ language. The framework itself is written in the C++ language with CUDA. The framework provides C++ classes that support the programmer to write stencil functions that update a grid point, execute these functions and describe efficient GPU-GPU communication. By using these classes, the programmer can write user code just in the C++ language and develop program code optimized for multiple GPU systems including GPU-rich supercomputers without introducing complicated optimizations. Since the programmer can write the stencil functions without depending on platform-specific programming languages, the framework is possible to translate these user-written functions to several platforms; the proposed framework currently generates CPU code and GPU code.

This paper reports that the programming model and our implementation strategies of the proposed framework, and the performance results of the dynamical core and a portion of physics processes in the framework-based ASUCA. To develop the framework-based ASUCA, all code are re-written in C++, ported from Fortran, using the all functions provided the proposed framework described later. Although original ASUCA adopts kij-ordering as the element order of three-dimensional arrays, we use ijk-ordering for the framework-based version to increases coalesced memory access on GPUs. We demonstrate the performance of the multi-GPU computation on the TSUBAME 2.5 supercomputer at Tokyo Institute of Technology. As a result, our code combines distributed GPUs over InfiniBand-connected nodes with MPI achieves very high performance of 209.6 TFlops in single precision using 4,108 GPUs for a mesh size of $19968 \times 20224 \times 60$.

II. Weather prediction code ASUCA

In this section, we review ASUCA (Asuca is a System based on a Unified Concept for Atmosphere), a next-generation high resolution mesoscale atmospheric model being developed by the Japan Meteorological Agency (JMA) [19], [10], [11]. The ASUCA is going to succeed the Japan Meteorological Agency Non-Hydrostatic Model (JMA-NHM) [20] as an operational non-hydrostatic regional model.

This model introduces non-hydrostatic equations that conserve mass and some highly efficient numerical methods in fluid dynamics that are increasingly popular in numerical weather prediction models.

ASUCA utilizes a generalized coordinate $(\hat{x}^1, \hat{x}^2, \hat{x}^3)$. Employing the Einstein summation convention, its flux-form non-hydrostatic balanced equations for the dynamical core are written as follows:

$$\frac{\partial}{\partial t}\left(\frac{\rho u^i}{J}\right) + \frac{\partial}{\partial \hat{x}^j}\left(\frac{\rho u^i \hat{u}^j}{J}\right) + \frac{\partial}{\partial \hat{x}^n}\left(\frac{1}{J}\frac{\partial \hat{x}^n}{\partial x^i}p\right) - \frac{\rho g^i}{J} = \frac{F^i}{J}, \tag{1}$$

$$\frac{\partial}{\partial t}\left(\frac{\rho}{J}\right) + \frac{\partial}{\partial \hat{x}^i}\left(\frac{\rho \hat{u}^i}{J}\right) = \frac{F_\rho}{J}, \tag{2}$$

$$\frac{\partial}{\partial t}\left(\frac{\rho \theta_m}{J}\right) + \frac{\partial}{\partial \hat{x}^i}\left(\frac{\rho \theta_m \hat{u}^i}{J}\right) = \frac{F_{\rho\theta_m}}{J}, \tag{3}$$

$$\frac{\partial}{\partial t}\left(\frac{\rho q_\alpha}{J}\right) + \frac{\partial}{\partial \hat{x}^i}\left(\frac{\rho q_\alpha \hat{u}^i_\alpha}{J}\right) = \frac{F_{\rho q_\alpha}}{J}\ (\alpha = v, c, r, i, s, g, h), \tag{4}$$

$$p = R_d \pi(\rho\theta_m), \tag{5}$$

where u^i $(i = 1, 2, 3)$ and $\hat{u}^i$ $(i = 1, 2, 3)$ represent the velocity components in the Cartesian coordinate and the generalized coordinate, respectively. Here, J is the Jacobian of coordinate transformation, π is the Exner function, ρ is a total mass density, and q_α represents a ratio of the density of water substance α to the total mass density. ASUCA can simulate water vapor, cloud water, rain, cloud ice, snow, graupel, and hail, each of which is represented by q_v, q_c, q_r, q_i, q_s, q_g, and q_h, respectively. The velocity $\hat{u}^i_\alpha$ in the equation of water substances denotes $\hat{u}^i + \hat{u}^i_{t\alpha}$, where $\hat{u}^i_{t\alpha}$ is terminal fall velocity of water substance α. Let θ_m be $\theta(\rho_d/\rho + \epsilon\rho_v/\rho)$, where θ is the potential temperature, $\rho_v = q_v\rho$, $\rho_d = \rho(1 - q_v - q_c - q_r - q_i - q_s - q_g - q_h)$ and ϵ is the ratio of the gas constant for water vapor R_v to that for dry air R_d. The notation F^i contains the Coriolis force, diffusion, diabatic effects and turbulent process. F^i, F_ρ and $F_{\rho\theta_m}$ involve terms arising from the density change due to precipitation. The term $F_{\rho q_\alpha}$ represents interactions between water substances calculated in cloud microphysics processes.

In ASUCA, the Lorenz coordinate is used on the Arakawa C grid. The equations are discretized using the finite volume method (FVM). The flux limiter function proposed by Koren [21] is adopted for monotonicity to avoid numerical oscillations. In a weather prediction code, since the vertical grid spacing is much smaller than the horizontal one, the sound speed in the vertical direction determines a time step. To avoid this, ASUCA introduces the horizontally explicit and vertically implicit (HE-VI) scheme with a time-splitting method [22]. In this method, each time integration step consists of several short sub-steps and a long time step. The short time steps, which employ the third-order Runge-Kutta scheme, are used for horizontal propagation of sound waves and gravity waves with implicit treatment for vertical propagation. The long time step is used for the advection of the momentum, the density, the potential temperature and the water substances, the Coriolis force, the diffusion and other effects by physical processes with the third-order Runge-Kutta method proposed by Wicker et al [23]. Although the physical processes of ASUCA have been developed separately from the dynamical core of ASUCA, a

Kessler-type warm-rain scheme for cloud-microphysics parameterization is embedded in the original ASUCA dynamical core code, which scheme is also implemented as an option in the JMA-NHM [24]. In this research, we use this scheme for the physical processes.

III. Overview of Framework

This section describes the proposed framework. The proposed framework is designed to provide highly-productive programming environment for stencil applications with explicit time integration running on regular structured grids, including weather prediction codes. The framework updates physical variables defined on grid points and stored in arrays in user programs. The framework is intended to execute user programs on NVIDIA's GPUs; the C/C++ language and CUDA are used for the implementation of CPU code and GPU code, respectively. The framework also supports multi-GPU computation.

Our major design goals of the framework are described as follows.

- The user code with the framework should be written in a standard language without using non-standard programming models and language extension, especially considering cooperation with external existing libraries. As array data types, the framework exploits arrays of C/C++ language without introducing any unique data types for arrays. These full-compatible data types allow us to call the external library freely in the user code.
- The framework should provide unified interfaces for both inter-node and intra-node communications while each of these communications is performed using the most appropriate method.
- The framework allows us to write multi-GPU code without considering handling multiple GPUs on a single process, which often requires careful programming techniques.
- To perform stencil computations on grids, the programmer only defines C++ functions that update a grid point, which is applied to entire grids by the framework. Our framework automatically translates these functions and generates both GPU and CPU code. The framework allows us to write the user code just in the C++ language and we can develop program code optimized for GPU computing without introducing complicated optimizations.
- The programmer should be able to write stencil functions without relying on element order of three-dimensional arrays. Stencil functions only depend on a physical coordinate.

IV. Framework Implementation and Programming Model

This section describes the implementation of the proposed framework and programming model of this. First we describe the structure of the entire framework and the execution of the user-written functions that update the physical variables on grids. Then, we describe the implementation and programming model of GPU-GPU communication and the overlapping method to hide this communication cost.

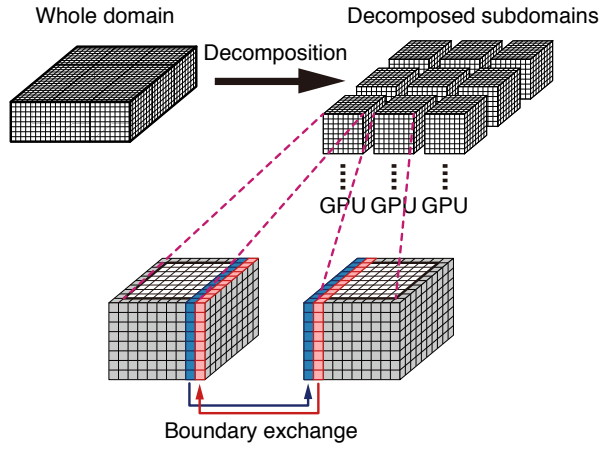

Fig. 1. Multi-GPU computing of mesh-based computation.

A. Structure of Framework

In the multi-GPU computation of mesh-based applications, domain decomposition is often used for these parallelization. The fundamental structure of this framework is based on this strategy. Figure 1 shows the domain decomposition of computational grid. Since stencil computation that updates to a point of grid needs to access its neighbor points, the data exchanges of boundary regions between subdomains are performed frequently.

In order to utilize peer-to-peer communication between GPUs on the same node for boundary exchanges, OpenMP and MPI library are used for intra-node and inter-node parallelization, respectively. Each GPU is assigned to a single OpenMP thread. Figure 2 shows overall structure of this framework. This framework parallelizes not parts of GPU computation in the user code but the entire user code from beginning to end including memory allocation and time integration loop. Thus, the programmer can focus on a single GPU as programming with only MPI (a red frame shown in Figure 2) and can write the user code without considering handling multiple GPUs with both MPI and OpenMP. Note that since computing on CPU using the framework is parallelized by OpenMP in the same way as multi-GPU computation, the computing on CPU needs boundary data exchanges as well as the GPU case even when computation is performed on a single process without using MPI.

B. Parallelizing User Code

The user program must first create a computational domain in each MPI process by using `DomainGroup`, `DomainManager` and `DomainSize`, which are C++ classes provided by the framework.

```
DomainManager manager(px, py, pz);
DomainSize domsize(nx, ny, nz,
                   mgnx, mgny, mgnz);
manager->
   init_domain_size_by_local(domsize);
```

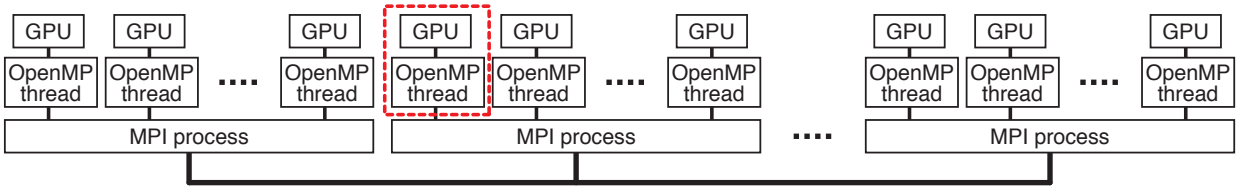

Fig. 2. Multi-GPU computing by using both MPI and OpenMP in the framework.

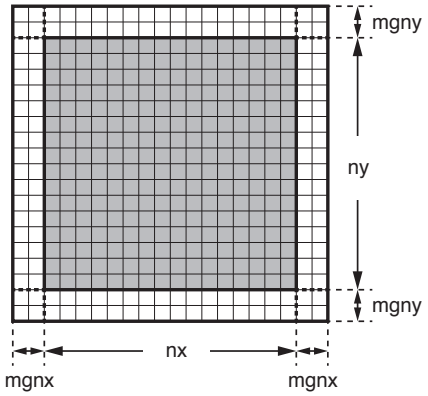

Fig. 3. X-Y plane of a computational subdomain that assigned to a GPU. The gray cells represent the computational region. The white cells represent the boundary regions that store the data sent by neighbor subdomains.

```
manager->set_thread_assignment(nthreads);
DomainGroup domain_group(rank, &manager);
domain_group.run(main_run);
```

`DomainManager` manages the 3D domain decomposition in the user program. The three parameters of this specify the division numbers of each of the three dimensions. Currently all decomposed subdomains must have the same size, which is specified by `init_domain_size_by_local` with `DomainSize`. The first three parameters of `DomainSize` are size of each of the three dimensions of the subdomain. The second three parameters are the boundary thickness of each of the three dimensions. Figure 3 shows X-Y plane of a decomposed subdomain. The boundary regions (i.e., the white cells in the figure) are used to store the data sent by neighbor subdomains. Since each subdomain is assigned to an OpenMP thread, it is computed by a single GPU. `DomainGroup` is initialized with the MPI rank `rank` and `DomainManager`, which has `nthreads` parameter that represents the number of OpenMP threads created in each MPI process. `DomainGroup` actually creates `nthreads` of OpenMP threads by using an OpenMP parallel directive. Each of threads executes a user-written function specified by `DomainManager::run`, i.e., `main_run` in the above code.

In order to utilize the intra-node communication as possible, the framework assigns neighboring subdomains to a single node. Since the communication between GPUs for the x boundary tends to degrade the performance due to complicated memory access pattern, the framework assigns this communication as intra-node communication and utilizes peer-to-peer communication for this in order to minimize this performance degradation.

The user-written function executed in the multiple threads (i.e., `main_run`) has to run simulation code from beginning to end including memory allocation and time integration loop as follows:

```
int main_run(const Domain &domain) {
  const DomainSize &domsize
      = domain.local_domain_size();
  const int ln = domsize.ln();
  float *f, *fn;
  cudaMalloc(&f , ln*sizeof(float));
  cudaMalloc(&fn, ln*sizeof(float));
  initialize_diffusion(domsize, f);
  ...
}
```

The function specified by `DomainManager::run` must receive a `Domain` object as the first parameter. `Domain` holds the information of a computational subdomain assigned to each OpenMP thread, including its size and the connection relation with neighbor subdomains. The size of computational subdomain can be retrieved by `Domain::local_domain_size()`, which may be used for allocation memory and initialization in the user code.

Utilizing multiple GPUs in a single process often requires the programmer to allocate the several numbers of arrays for a just single physical variable. However, thanks to using OpenMP for intra-node parallelization in this framework, the programmer allocates just one array for each physical variable in the user code in the same way as MPI code without OpenMP (i.e., flat-MPI code) even when multiple GPUs are handled by a single process, which contributes to simplifying the user code.

C. Stencil Computation on Grids

Since stencil computation is based on discretization of partial differential equations, stencil computation can be essentially divided into two parts; one is user-written stencil function that updates a grid point and the other is loop range that represents where user-written functions are executed. To use this framework, the programmer just writes stencil functions and executes them through C++ classes provided by this framework.

1) Writing Stencil Functions: In this framework, stencils must be defined as C++ functors called *stencil functions*. The stencil function for three-dimensional diffusion equation is defined as follows:

```
struct Diffusion3d {
__host__ __device__
void operator()(const ArrayIndex3D &idx,
  float ce, float cw, float cn, float cs,
  float ct, 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>()];
}
};
```

Stencil access patterns on three-dimensional grids are described by using `ArrayIndex3D`, which is provided by the framework. Similarly, classes for writing 1D and 2D access patterns are provided.

`ArrayIndex3D` holds the size of each dimension of a grid (n_x, n_y, n_z) and index parameters (i, j, k). `ArrayIndex3D` can be used for an array `f` that has $n_x n_y n_z$ elements. When `idx` is an object of `ArrayIndex3D`, `f[idx.ix()]` returns an element on the (i, j, k) point of the grid. `ArrayIndex3D` has C++ template member functions that provide indices of points around the (i, j, k) point of the grid; `idx.ix<1,0,0>()` and `idx.ix<-1,-2,0>()`, for example, return indices of $(i + 1, j, k)$ and $(i - 1, j - 2, k)$ points, respectively. Using template functions for writing stencil accesses allows us to assume that data dependencies between stencil points can be statically identified and compiler optimizations for index calculations can be expected.

Using `ArrayIndex3D` for writing stencil functions contributes to not only simplifying writing stencil accesses and enforcing regular neighbor data accesses patterns in stencil functions, but writing stencil functions without dependence on element order of three-dimensional arrays. `ArrayIndex3D` returns indices that can be used for arrays which of variables stored sequentially in the order of the x, y, z (ijk-ordering). To support to access variables stored in other orderings, such as kij-ordering, which is used in original ASUCA code, this framework provides other similar types of `ArrayIndex3D`. By these classes, we can easily change element order of arrays to an appropriate one that is suitable for target processor without modifying the user code, resulting in increasing maintainability of application codes.

The function parameters of stencil functions must begin with `ArrayIndex3D`, which represents the coordinate of the point where this function is applied. This is followed by any number of additional parameters, including scalar values and pointers of arrays, which typically have $n_x n_y n_z$ elements. The return type of stencil functions must be void. In stencil functions, any dependency among different stencil points must not be assumed, since stencil functions may be executed in parallel with an arbitrary order.

2) Run Stencil Functions on Grids: In order to apply user-written stencil functions to grids, the framework provides the `Loop3D` class, which is used to invoke the diffusion equation on the three-dimensional grid as follows:

```
Loop3D loop3d(nx+2*mgnx, mgnx, mgnx,
              ny+2*mgny, mgny, mgny,
              nz+2*mgnz, mgnz, mgnz);
loop3d.run(Diffusion3d(), ce, cw, cn, cs,
           ct, cb, cc, f, fn);
```

`Loop3D` is initialized with parameters that specify a 3D rectangular range where stencil functions are applied. Similarly, `Loop1D` and `Loop2D` for 1D and 2D grids are provided. The first three parameters of the constructor of `Loop3D` specify the range in the x direction. When the first three parameters are n_x, i_0 and i_1, stencil functions must be applied from index i_0 to index $n_x - i_1 - 1$ with assuming the size of grid in the x direction is n_x.

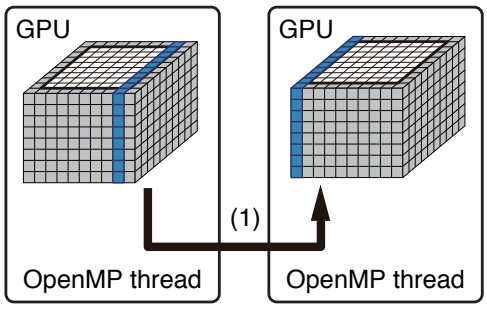

Fig. 4. Intra-node GPU-GPU communication by the OpenMP threads. This process is composed of (1) GPUDirect peer-to-peer access.

The parameters of `Loop3D::run` must begin with a stencil function defined as a functor, followed by any number of additional parameters that are provided to this functor. We use C++ type inference and call an appropriate functor at `Loop3D::run`. The programmer can define stencil functions as both host and device (i.e., GPU) functions using the qualifiers `__host__` and `__device__` provided by CUDA.

`Loop3D` executes stencil functions on grids sequentially for CPU while it executes the stencil functions in parallel for GPU using CUDA's global kernel functions. `Loop3D` determines whether a pointer given by `Loop3D::run` as a parameter points to host memory or device memory, and call appropriate internal functions within `Loop3D`. On GPU, arrays that store read-only data in global memory are loaded through read-only data cache by the framework to improve performance of global loads. To calculate the user-written functions for a given grid size (n_x, n_y, n_z), the kernel functions are configured for execution with $(64, 2, n_z/16)$ threads in each CUDA block. Each thread performs calculations consecutive 16 elements marching in the z direction, resulting in performance improvement.

D. GPU-GPU Communication

In this framework, multi-GPU calculations within a same node are performed by an MPI process with several OpenMP threads, each of which is assigned to a single GPU. Since pointers that point to arrays holding the data transferred between GPUs are registered in the memory space that are shared among all threads by using the framework functions prior to data transfer, each thread in the process is able to access memory allocated in others directly. Based on this, the intra-node GPU-GPU communication is performed by just a copy between the memories of two different GPUs using `cudaMemcpy`. When two GPUs support GPUDirect peer-to-peer access, communication between these two GPUs no longer needs to be staged through the host and is therefore faster. Figure 4 illustrates intra-node GPU-GPU communication based on peer-to-peer access.

On the other hand, inter-node GPU-GPU communication is performed by using the MPI library. Figure 5 illustrates this

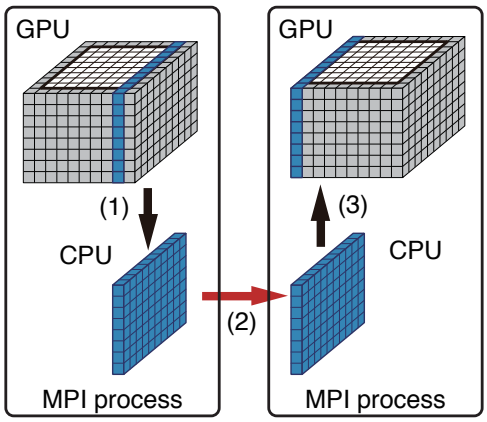

Fig. 5. Inter-node GPU-GPU communication by MPI. This process is composed of (1) memory copy from GPU to host, (2) data exchange by MPI communications and (3) memory copy from host to GPU.

communication. Since GPUs cannot directly access data stored on device memory of other GPUs on other nodes, the host CPUs are used as bridges to exchange boundary data between neighbor GPUs. This process is composed of the following three steps: (1) the data transfer from GPU to CPU using the CUDA runtime library, (2) the data exchange between nodes with the MPI library, and (3) the data transfer back from CPU to GPU with the CUDA runtime library. In order to improve performance of MPI communication and maintain the portability of the framework, the master thread executes all MPI communications required by all threads in the same process. Buffers on host memory used for MPI are allocated automatically by the framework. Note that GPUDirect RDMA is not used for this communication since TSUBAME2.5 does not currently support it.

The framework provides the `BoundaryExchange` class to write GPU-GPU communication. The `BoundaryExchange` class utilizes appropriate GPU-GPU communication described above; this class performs peer-to-peer communication between GPUs on a same node when possible while it performs inter-node GPU-GPU communication using the MPI library through host memory. `BoundaryExchange` is typically used as follows:

```
BoundaryExchange *exchange
            = domain.exchange();
exchange->append(array1);
exchange->append(array2);
exchange->append(array3);
exchange->transfer();
```

`BoundaryExchange` is initialized by `domain`, which is a `Domain` object, and holds the connection relation with neighbor subdomains and the size of data exchanged with them. When `BoundaryExchange::transfer` is called, boundary regions of arrays specified by `BoundaryExchange::append` are exchanged. In order to improve communication performance, the framework combines boundary data of several arrays into one array and transfers it to neighbor GPUs at one time. `BoundaryExchange` may allocate buffers on host memory when it uses MPI communication.

E. Overlapping Method

The data communication time between GPUs is not ignored in the total execution time in the case of large-scale computation. The overlapping technique to hide communication overhead with computation can contributes to performance improvement.

This framework provides kernel-division overlapping method reported in our previous work [10]. This method exploits data independency within a single variable. Since each element of a variable can be computed independently for one calculation, computations for the boundary regions can be executed separately from other calculations for the rest of the domain. By dividing a single kernel into several kernels for x, y and z boundaries, and an inside domain, we can overlap communication for the boundary data exchange with the computation for the inside domain.

Figure 6 illustrates the flow of the overlapping method named *one-divided-kernel overlapping method* in two-dimensional computation case. As shown in the figure, when each of computational subdomains requires two-element-thick mesh as halo regions, the framework assigns two-element-thick mesh to boundaries and executes five kernels for four boundaries and one inside region. First, the values in the inside region are computed in a CUDA stream named *stream1*, while simultaneously the boundary exchange between GPUs is executed in another stream named *stream2*. The boundary exchange consists of asynchronous memory copies from GPUs to CPUs executed by CUDA memory operations, data exchanges between CPUs with MPI, and asynchronous memory copies from CPUs to GPUs for inter-node communication case. When this copy sequence is completed, the computations for the four boundaries are executed in four different streams, which contributes to utilizing computational resources effectively. Note that since the performance of the GPU is often improved for large numbers of threads, dividing kernels to smaller domains degrades the kernel performance itself. However, this overlapping method by using several divided kernels have a potential to hide communication and result in improving the overall performance. Similar to the two-dimensional computational case, the kernel-division overlapping method for the three-dimensional computations exploits seven kernels in seven different streams for computations of boundaries and an inside region.

In mesh-based applications, communication time spent by transferring boundaries of a variable is often longer than time of a computation based on this variable, which degrades applications performance even when one-divided-kernel overlapping method is used. To avoid this, we exploit an operation that is often used in the real applications such as ASUCA. This operation consists of the following. First, a computation updates some variables. After that, boundaries of these variables are exchanged between GPUs. When it has finished, another computation is executed using these transferred variables. Exploiting a sequence of this operation, our framework provides another overlapping method to hide one communication cost by two computations, which is named *two-divided-kernel overlapping method*. Figure 6 also illustrates this overlapping method. Similar to previous one, each of two kernels is divided to five kernels in two-dimensional computations and seven kernels in three-dimensional computations. First, the values

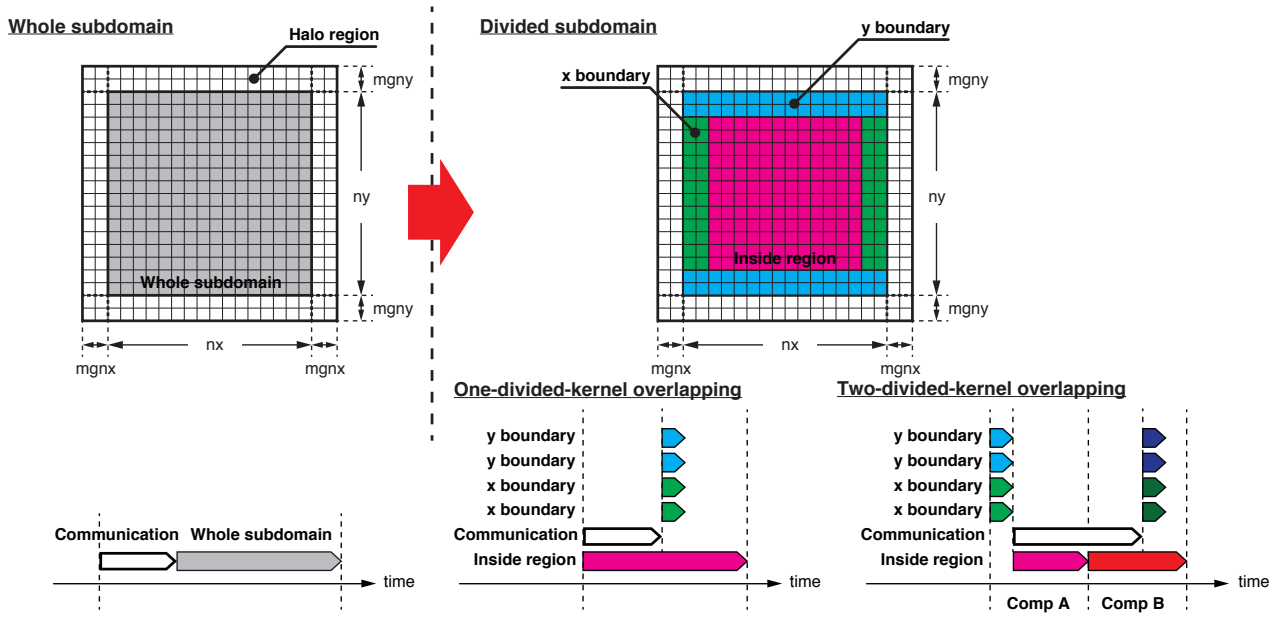

Fig. 6. Scheme of the non-overlapping method and the overlapping method based on kernel division.

in the boundary regions are updated by a computation named *Comp A*. After that, the boundary exchange between GPUs is executed while simultaneously the values in the inside regions are updated by Comp A. When all computation by Comp A have finished, another computation named *Comp B* for the inside region starts. After boundary exchange is completed, Comp B for the boundaries is executed.

In order to apply these two kernel-division overlapping methods to user programs, the framework provides the `CompCommBinder` class, which is used to execute diffusion computations along with boundary exchange as follows:

```
BoundaryExchange *exchange
            = domain.exchange();
exchange->append(f);
CompCommBinder<Loop3D> ccbinder(exchange);
ccbinder.set_post_func(&loop,
    create_funcholder<Loop3D>(
        Diffusion3d(), ce, cw, cn, cs,
        ct, cb, cc, f, fn));
ccbinder.set_use_overlapping();
ccbinder.run();
```

`CompCommBinder` is initialized with a `BoundaryExchange` object. To apply the one-divided-kernel overlapping method to a user-written function, by using `CompCommBinder::set_post_func`, the programmer specifies a loop range and the stencil function with additional parameters that are provided to this function. `CompCommBinder::run` divides the specific loop range into several loop ranges that correspond to boundaries and the inside region, and executes this stencil function on these loop ranges in several streams described above. Holding the parameters provided to the stencil function in `CompCommBinder` enables multiple executions of the stencil function. When the programmer specifies another stencil function by using `CompCommBinder::set_pre_func`, the framework automatically uses the two-divided-kernel overlapping method. Although the loop range for computation of the inside region is automatically determined by grid sizes of halo regions as default, the programmer can also specify loop range of the inside region freely by providing some other parameters to `CompCommBinder::set_pre_func` and `set_post_func`. `CompCommBinder::run` executes the boundary exchange and the stencil function sequentially when `CompCommBinder::set_use_overlapping` is not invoked.

V. Performance Analysis and Discussion

In this section, we present the strong and weak scaling results obtained by the weather prediction code ASUCA on the TSUBAME 2.5 supercomputer. 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.

This code is written with exploiting the all functions of our framework described in the previous section. First, we compare the strong scaling obtained with the non-overlapping method and the overlapping method. Next, we describe the weak scalability of ASUCA by adopting the overlapping method.

A. Strong Scaling

We show the strong scaling results of our framework-based weather prediction code using multiple GPUs of TSUBAME

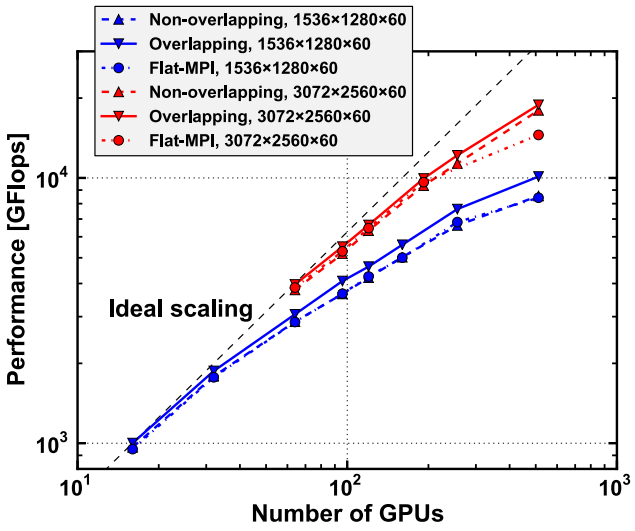

Fig. 7. Strong scaling results of the non-overlapping method, the overlapping method and flat-MPI implementation.

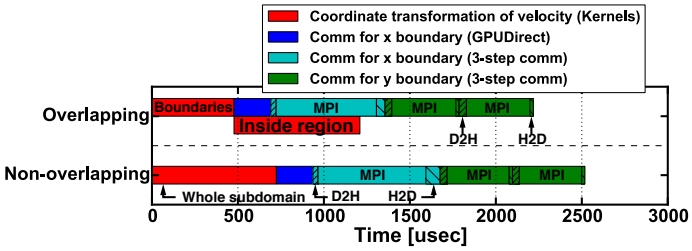

Fig. 8. Breakdown of communication times and computational times of the non-overlapping and one-divided-kernel overlapping methods. In the overlapping method, the computation is divided into the computation for the inside region (*inside region* in the graph) and the computation for boundaries (*boundaries*). Each three-step communication consists of GPU to host communication (*D2H*), MPI communication (*MPI*) and host to GPU communication (*H2D*). Note that some labels are abbreviated due to space limitations.

2.5.

In order to evaluate the performance of GPU computation, we count the number of floating-point operations in the framework-based weather prediction code by running it on a CPU with a performance counter provided by the Performance API (PAPI) [25]. The performance of the GPU computation is evaluated by using this obtained count and the observed GPU elapsed time. Note that since the framework supports both GPU and CPU executions, we use the same code for both CPU and GPU computation.

Figure 7 shows the performance of ASUCA running on multiple GPUs in single precision and compares the strong scalability of the non-overlapping version and the overlapping version of ASUCA. In this graph, Flat-MPI version, in which each MPI process handles a single GPU, is also shown as reference. The mountain wave test is used as a benchmark [26]. Since two of three GPUs on each TSUBAME node can utilize GPUDirect peer-to-peer access, we use these two GPUs per each node for these calculations. These two GPUs on each node are handled by two OpenMP threads on a single MPI process in both non-overlapping and overlapping methods while those are handled by two MPI process in flat-MPI version. Varying the number of GPUs used for the calculations, we perform ASUCA on two different mesh sizes: $1536 \times 1280 \times 60$ and $3072 \times 2560 \times 60$. We decompose the given grid in both the x and y directions (2D decomposition) and allocate each subdomain to a single GPU. Each GPU is responsible for all the elements in the z direction since the z dimension is relatively small in our weather prediction simulation. Since one of two x direction communications is performed within a node, peer-to-peer access is utilized for this communication.

As shown in Figure 7, we observe that the overlapping method works effectively to hide communication cost for both mesh sizes we expected, resulting in performance improvement. In the results using a $3072 \times 2560 \times 60$ mesh on 512 GPUs, for example, the overlapping method and the non-overlapping method achieve $18.9\,\mathrm{TFlops}$ and $18.0\,\mathrm{TFlops}$, respectively while flat-MPI version reaches $14.5\,\mathrm{TFlops}$.

In order to further analyze the benefits by exploiting the overlapping method, we go into the analysis of the breakdown of the computational times and communication times related to two key computations in both the overlapping and non-overlapping methods. Figure 8 shows the computation for coordinate transformation of velocity and related communication. The non-overlapping method, which result is depicted in the bottom of this graph, needs to perform computation first and communication next. The communication consists of the x direction communication using GPUDirect peer-to-peer access, the x direction communication using the three steps described above, and two of the y direction communication using the three steps. As shown in the graph, peer-to-peer access can successfully improves the communication performance. As the overlapping method, one-divided-kernel overlapping method is used for this computation. Since the performance of the GPU is often improved for large numbers of threads, dividing kernels to smaller domains degrades the kernel performance itself. While the computation by the single kernel takes $723.9\,\mu\mathrm{sec}$, the computation by the divided kernels takes $1209.3\,\mu\mathrm{sec}$ in total. Although computation time itself in the overlapping method is longer than that in the non-overlapping method, computation for the inside region is performed in parallel with a part of communication (i.e., $732.0\,\mu\mathrm{sec}$), resulting that performance improvement is observed in the overlapping method.

Figure 9 shows the computation for the pressure gradient force in the y direction with the computation for coordinate transformation of velocity and related communication. Since this communication is wedged between these two computations, two-divided-kernel overlapping method is utilized. Similar to Figure 8, although the divided kernels in the overlapping method takes longer time than the single kernel used in the non-overlapping method, a part of communication (i.e., $2916.7\,\mu\mathrm{sec}$) can be hidden by computation, which contributes to performance improvement.

B. Weak Scaling

We show the weak scaling results of ASUCA by multi-GPU computing on TSUBAME 2.5. Bigger domain has better performance and we choose that each GPU handles the domain

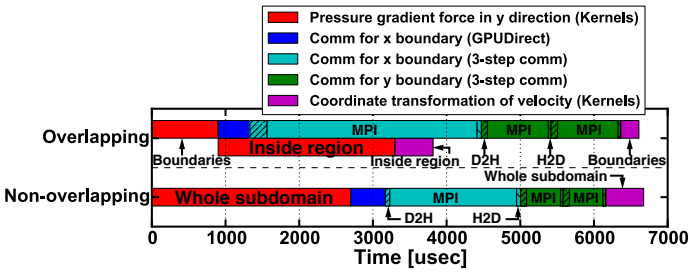

Fig. 9. Breakdown of communication times and computational times of the non-overlapping and two-divided-kernel overlapping methods. In the overlapping method, each computation is divided into the computation for the inside region (*inside region* in the graph) and the computation for boundaries (*boundaries*). Each three-step communication consists of GPU to host communication (*D2H*), MPI communication (*MPI*) and host to GPU communication (*H2D*). Note that some labels are abbreviated due to space limitations.

of $768 \times 128 \times 60$. Similar to the strong scaling study, the mountain wave test is used as a benchmark. Unlike the strong scaling study, we use three GPUs per each node of TSUBAME 2.5 in all cases of calculations in order to maximize attainable performance by using TSUBAME 2.5.

Figure 10 shows the performance of the framework-based ASUCA code running on multiple GPUs. The numbers of GPUs and the mesh sizes used for multi-GPU computing are shown in Table I. We measure the performance of ASUCA using both the overlapping method and non-overlapping methods in single precision. While computation and communication are performed sequentially in the non-overlapping methods, the overlapping version to hide communications overhead, resulting in performance improvement. As shown in the graph, we achieve an extremely high performance of 209.6 TFlops using 4,108 GPUs with the overlapping method in single precision. Comparing with the performance of 206.5 TFlops obtained by the non-overlapping method using 4,108 GPUs, the effects of the overlapping on performance improvement for the computation of the whole domain is approximately 2%. For the overlapping version, the weak scaling efficiency is above 95% for $19968 \times 20224 \times 60$ on 4108 GPUs with respect to the 24-GPU performance.

TABLE I. NUMBERS OF GPUS AND MESH SIZES FOR WEAK SCALING COMPUTATION.

Number of GPUs ($P_x \times P_y$)	Mesh size ($nx \times ny \times nz$)
24 (2×12)	$1536 \times 1536 \times 60$
864 (12×72)	$9216 \times 9216 \times 60$
1536 (16×96)	$12288 \times 12288 \times 60$
1944 (18×108)	$13824 \times 13824 \times 60$
2400 (20×120)	$15360 \times 15360 \times 60$
2904 (22×132)	$16896 \times 16896 \times 60$
3456 (24×144)	$18432 \times 18432 \times 60$
4056 (26×156)	$19968 \times 19968 \times 60$
4108 (26×158)	$19968 \times 20224 \times 60$

VI. REAL CASE SIMULATION WITH ASUCA

This section demonstrates that the framework-based ASUCA, which includes the full dynamical core and warm rains, can successfully simulate a basic set of real weather cases used in the JMA. Figure 11 shows a simulation result produced by the framework-based ASUCA describing a real

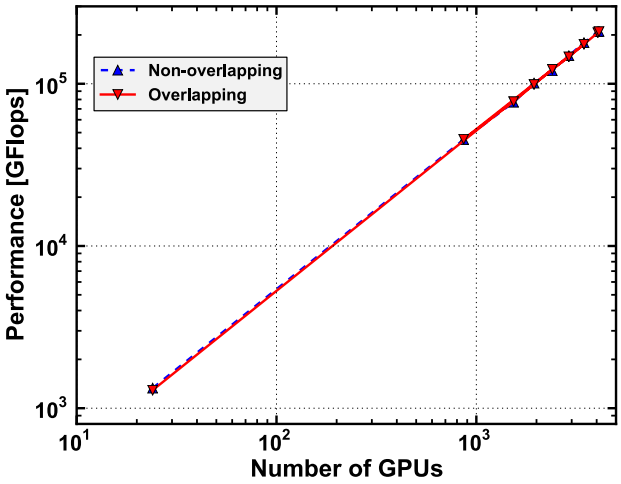

Fig. 10. Weak scaling results of the non-overlapping method and the overlapping method.

typhoon over Japan made with the real initial data and boundary data used for the current weather forecast at the JMA. This simulation is performed on a $5376 \times 4800 \times 57$ mesh with horizontal mesh resolution of 500 meters using 672 GPUs of TSUBAME 2.5.

VII. CONCLUSION

This paper has presented the programming model and implementation of our framework that is developed for multi-GPU computation of stencil applications, and evaluation of the weather prediction code ASUCA based on the proposed framework running on a supercomputer equipped with multiple GPUs. The design of framework focuses on the portability of both framework and user code and cooperation with the existing codes. Unlike previous research, the proposed framework itself is written in the C++ language with CUDA and can be used in the user code developed in the C++ language. The programmer can write user code just in the C++ language and develop program code optimized for multiple GPU systems without introducing complicated optimizations explicitly.

Our code can effectively utilize intra-node GPU peer-to-peer direct accesses with optimizations to hide communication overhead by overlapping of computation and communication. For stencil computation, the programmer writes only the platform-independent stencil functions that update a grid point using its neighbor points, which are applied to over grids on a specific device by the framework. Introducing CUDA-aware MPI to improve the performance will be a subject of our future work.

With our proposed framework, we have conducted performance studies of ASUCA using thousands of GPUs on the TSUBAME 2.5 supercomputer at Tokyo Institute of Technology. The performance evaluation has successfully demonstrated that strong scalability is improved by the overlapping method provided by the framework; with 4,108 GPUs, the performance reaches 209.6 TFlops in single precision for a mesh of $19968 \times 20224 \times 60$. We have also showed an actual

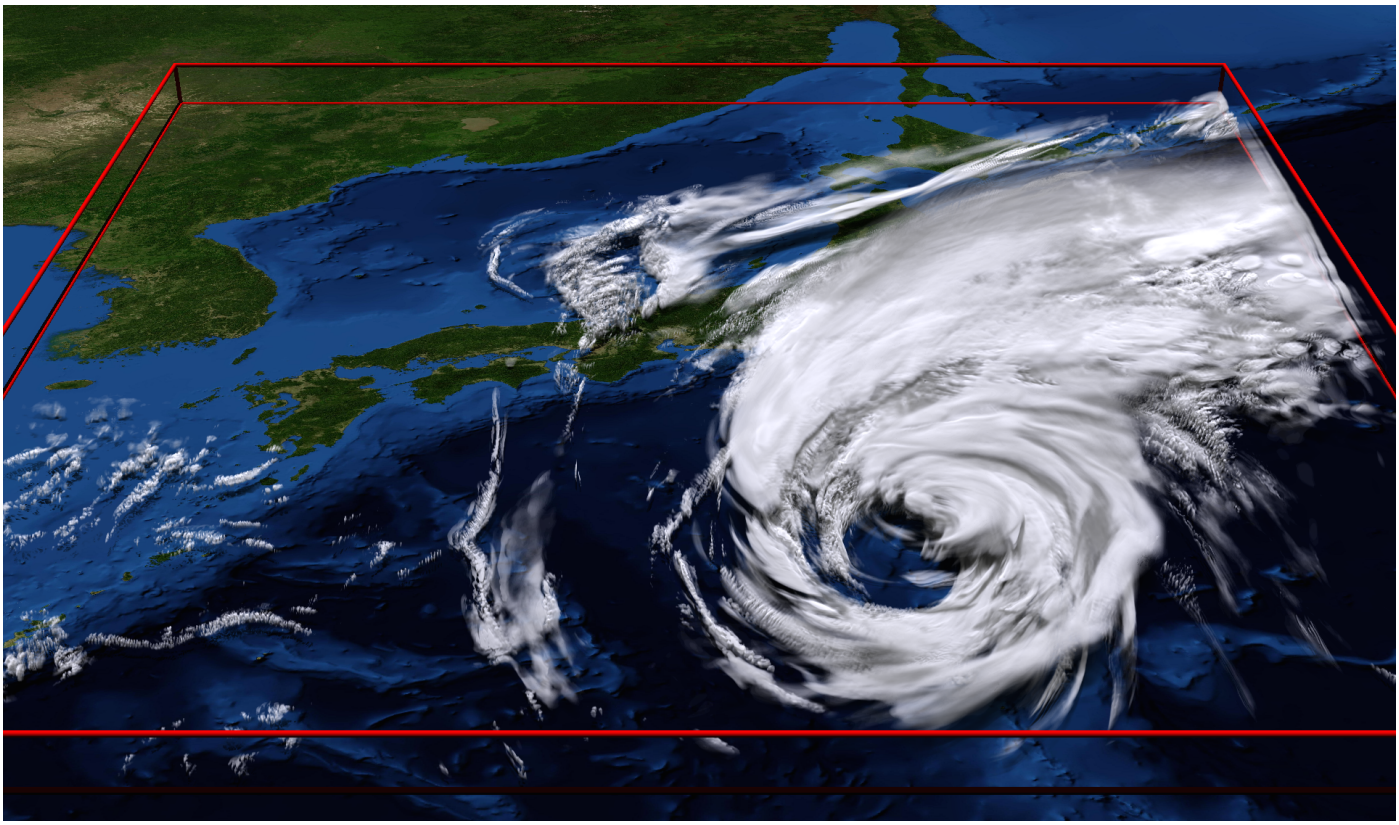

Fig. 11. An ASUCA simulation result describing a typhoon over Japan with $5376 \times 4800 \times 57$ mesh using 672 GPUs of the TSUBAME 2.5.

simulation result produced by ASUCA using the proposed framework.

ACKNOWLEDGMENT

This research was supported in part by KAKENHI, Grant-in-Aid for Young Scientists (B) 25870223, Grant-in-Aid for Scientific Research (B) 23360046 Grant-in-Aid for Scientific Research (S) 26220002 and Grant-in-Aid for Young Scientists (B) 25870226 from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan, in part by the Japan Science and Technology Agency (JST) Core Research of Evolutional Science and Technology (CREST) research program "Highly Productive, High Performance Application Frameworks for Post Petascale Computing," and in part by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" and "High Performance Computing Infrastructure" in Japan. The authors thank the Global Scientific Information and Computing Center, the Tokyo Institute of Technology for use of the resources of the TSUBAME2.5 supercomputer. The authors would like to thank Dr. Chiashi Muroi, Dr. Junichi Ishida, Dr. Kohei Kawano and Dr. Tabito Hara at the Japan Meteorological Agency the for providing the original ASUCA code, the real initial and boundary data.

REFERENCES

[1] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S.-i. Habata, M. Yokokawa, H. Itoh, and K. Otsuka, "A 26.58 Tflops global atmospheric simulation with the spectral transform method on the Earth Simulator," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–19.

[2] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G. Duda, X.-Y. Huang, W. Wang, and J. G. Powers, "A Description of the Advanced Research WRF Version 3," National Center for Atmospheric Research, 2008.

[3] A. S. Bland, R. A. Kendall, D. B. Kothe, J. H. Rogers, and G. M. Shipman, "Jaguar: The world's most powerful computer," in *2009 CUG Meeting*, 2009, pp. 1–7.

[4] J. C. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, no. AIAA 2009-758, jan 2009.

[5] T. Brandvik and G. Pullan, "Acceleration of a 3D Euler Solver using commodity graphics hardware," in *46th AIAA Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, January 2008.

[6] T. Hamada and K. Nitadori, "190 TFlops astrophysical N-body simulation on a cluster of GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. New Orleans, LA, USA: IEEE Computer Society, 2010, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.1

[7] T. Shimokawabe, T. Aoki, T. Takaki, A. Yamanaka, A. Nukada, T. Endo, N. Maruyama, and S. 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*, ser. SC '11. Seattle, WA, USA: ACM, 2011, pp. 1–11.

[8] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction." in *IPDPS*. IEEE, 2008, pp. 1–7.

[9] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *SC '09: Proceedings of the Conference on High Performance*

Computing Networking, Storage and Analysis. New York, NY, USA: ACM, 2009, pp. 1–11.

[10] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. 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*, ser. SC '10. New Orleans, LA, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.9

[11] T. Shimokawabe, T. Aoki, J. Ishida, K. Kawano, and C. Muroi, "145 TFlops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction," *Procedia Computer Science*, vol. 4, pp. 1535 – 1544, 2011, proceedings of the International Conference on Computational Science, ICCS 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911002249

[12] P. Johnsen, M. Straka, M. Shapiro, A. Norton, and T. Galarneau, "Petascale WRF simulation of hurricane sandy deployment of NCSA's Cray XE6 Blue Waters," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 63:1–63:7. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503231

[13] NVIDIA, "CUDA C Programming Guide 5.5," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, NVIDIA, 2013.

[14] OpenACC-Standard.org, "The OpenACC application programming interface," http://www.openacc-standard.org/, November 2011.

[15] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995932

[16] N. Maruyama, T. Nomura, K. Sato, and S. 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*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 11:1–11:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063398

[17] 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*, 2011, pp. 676–687.

[18] M. Christen, O. Schenk, and Y. 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*, ser. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, pp. 11:1–11:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389011

[19] J. Ishida, C. Muroi, K. Kawano, and Y. Kitamura, "Development of a new nonhydrostatic model "ASUCA" at JMA," *CAS/JSC WGNE Reserch Activities in Atomospheric and Oceanic Modelling*, 2010.

[20] K. Saito, T. Fujita, Y. Yamada, J.-i. Ishida, Y. Kumagai, K. Aranami, S. Ohmori, R. Nagasawa, S. Kumagai, C. Muroi, T. Kato, H. Eito, and Y. Yamazaki, "The operational JMA nonhydrostatic mesoscale model," *Monthly Weather Review*, vol. 134, pp. 1266–1298, 2006.

[21] B. Koren, "A robust upwind discretization method for advection, diffusion and source terms," *CWI Report NM-R9308*, 1993.

[22] W. C. Skamarock and J. B. Klemp, "Efficiency and accuracy of the Klemp-Wilhelmson Time-Splitting technique," *Monthly Weather Review*, vol. 122, pp. 2623–+, 1994.

[23] L. J. Wicker and W. C. Skamarock, "Time-splitting methods for elastic models using forward time schemes," *Monthly Weather Review*, vol. 130, pp. 2088–2097, 2002.

[24] M. Ikawa and K. Saito, "Description of a non-hydrostatic model developed at the Forecast Research Department of the MRI," *Technical Reports of the Meteorological Research Institute*, vol. 28, pp. 238–, 1991.

[25] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.

[26] T. Satomura, T. Iwasaki, K. Saito, C. Muroi, and K. Tsuboki, "Accuracy of terrain following coordinates over isolated mountain: Steep mountain model intercomparison project (st-MIP)," *Annuals of the Disaster Prevention Research Institute, Kyoto University*, vol. 46 B, pp. 337–346, 2003.