A High-productivity Framework for Multi-GPU computation of Mesh-based applications

Takashi Shimokawabe
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku,
Tokyo, Japan
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 paper proposes a high-productivity framework for multi-GPU computation of mesh-based applications. In order to achieve high performance on these applications, we have to introduce complicated optimized techniques for GPU computing, which requires relatively-high cost of implementation. Our framework automatically translates user-written functions that update a grid point and generates both GPU and CPU code. In order to execute user code on multiple GPUs, the framework parallelizes this code by using MPI and OpenMP. The framework also provides C++ classes to write GPU-GPU communication effectively. The programmers write user code just in the C++ language and can develop program code optimized for GPU supercomputers without introducing complicated optimizations for GPU computation and GPU-GPU communication. As an experiment evaluation, we have implemented multi-GPU computation of a diffusion equation by using this framework and achieved good weak scaling results. By using peer-to-peer access between GPUs in this framework, the framework-based diffusion computation using two NVIDIA Tesla K20X GPUs is 1.4 times faster than manual implementation code. We also show computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework.

Categories and Subject Descriptors
D.3.3 [Software]: Language Constructs and Features—Frameworks

General Terms
Languages

Keywords
Application Framework, Multi-GPU Computing, High Performance Computing

1. INTRODUCTION
Mesh-based physical simulations are important applications in the field of high-performance computing. Because of extremely memory-bottlenecked computation, these simulations are difficult problems on conventional supercomputers. Exploiting GPU for general-purpose computing has emerged as a high-performance computing technique to accelerate mesh-based physical applications [9, 8, 10, 6, 4, 3]. Although various applications are accelerated by GPUs, programming on different types of devices by using low level platform-specific programming languages such as CUDA [7] 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. Mint was proposed as a high-level framework specialized for stencil computations on CUDA-enabled GPUs [11]. 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 [5]. PATUS was proposed as a code generation and auto-tuning DSL for stencil computations targeted at multi- and many core processors [1, 2].

Recently, 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. Multi-GPU computation of mesh-based applications has the potential to achieve high performance. Introducing optimizations allow us to achieve optimal performance. However, programming for large-scale parallel computing on GPU-rich supercomputers is a more difficult task than programming for a single device since introducing optimizations for communication along with single-GPU optimizations is required.

In this paper, high-productivity framework for multi-GPU computing of mesh-based applications is proposed. 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 the standard language without using the non-standard programming model and language extension. Thus, unlike previous research, the proposed framework can be used in the user code developed in the C++ languages. 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.

2. OVERVIEW OF FRAMEWORK

The proposed framework is designed to provide highly-productive programming environment for stencil applications with explicit time integration running on regular structured grids. The framework updates the 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 proposed framework is written in the C++ language and CUDA and can be used in the user code developed in the C++ language. It is important that the user code with the framework can be written in the standard language without using the non-standard programming model and language extension, especially considering the 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.

- In the proposed framework, multiple GPUs on a same node are handled by a process with inter-node MPI communication. Since multiple GPUs on a same node are handled by a single process instead of several MPI processes, we may use peer-to-peer access between these GPUs, resulting in performance improvement.

- 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 framework provides a single function that supports efficient both inter-node and intra-node GPU communications. By using this, we do not have to call MPI functions for communications explicitly.

3. FRAMEWORK IMPLEMENTATION

This section describes the implementation of the proposed framework. 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 of GPU-GPU communication that is required in multi-GPU computing.

3.1 Structure of Framework

The proposed framework supports multiple GPU computing. In the multi-GPU computation of mesh-based applications, the domain decomposition is often used for these parallelization. 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.

Figure 2 shows the multi-GPU computing in this framework using both MPI and OpenMP. In order to utilize peer-to-peer communication between GPUs on the same node, OpenMP and MPI library are used for intra-node and inter-node parallelization, respectively. Each GPU is assigned to a single OpenMP thread. 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. Note that although overlapping communication with computation is an effective optimization for multi-GPU computation, this framework currently does not introduce it.

3.2 Stencil Computation on Grids

In order to execute stencil computation on grids, the pro-
grammer must describe functions that update a grid point. The framework provides C++ classes that apply user-written functions to grids. While the user-written functions are executed on grids sequentially for CPU within these classes, these are executed on grids in parallel for GPU using CUDA’s global kernel functions. On GPU, read-only data in global memory is 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 on consecutive 16 elements marching in the \(z\) direction, resulting in performance improvement.

### 3.3 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 3 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 4 illustrates this 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, 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.

### 4. PROGRAMMING MODEL

The framework is written in the C++ language and CUDA. The programmer is required to use the C++ template classes and functions provided by this framework to express stencil-based computations, which are executed with optimizations on GPU. In this section, we describe the programming model of this framework by taking a diffusion computation as an example.

#### 4.1 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.

```cpp
DomainManager manager(px, py, pz);
DomainSize domsize(nx, ny, nz, mgnx, mgny, mgnz);
manager->init_domain_size_by_local_domain_size(domsize);
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_domain_size` 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 5 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 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.

The user-written function executed in the multiple threads has to run simulation code from beginning to end including memory allocation and time integration loop as follows:

```c
int main_run(const Domain &domain) {
   const DomainSize &domain_size;
   float *f, *fn;
   cudaMalloc(&f, domain.size.ln() * sizeof(float));
   cudaMalloc(&fn, domain.size.ln() * sizeof(float));
   initialize_diffusion(domain_size, 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 same number 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.

### 4.2 Stencil Computation

#### 4.2.1 Writing Stencil Functions

In this framework, stencils must be defined as C++ functions called stencil functions. The stencil function for three-dimensional diffusion equation is defined as follows:

```c
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.i() + ce*f[idx.i<1,0,0>()] + cw*f[idx.i<0,0,1>()] + cn*f[idx.i<0,0,-1>()] + cs*f[idx.i<0,1,0>()] + ct*f[idx.i<0,1,-1>()] + cb*f[idx.i<0,-1,0>()] + cc*f[idx.i<0,-1,1>()];
   }
};
```

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. These classes contribute to simplifying writing stencil accesses and enforcing regular neighbor data accesses patterns in stencil functions.

ArrayIndex3D holds the size of each dimension of a grid (nx, ny, nz) and index parameters (i, j, k). ArrayIndex3D can be used for an array f that has nx*ny*nz elements. When idx is an object of ArrayIndex3D, f[idx.i()] will return 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; ArrayIndex3D has C++ template member functions that provide indices of points around the (i, j, k) point of the grid; index.i<1,0,0>() and index.i<-1,0,0>() will, for example, return indices of (i+1, j, k) and (i-1, j, 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.

The function parameter of stencil functions must begin with ArrayIndex3D, which represent 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 nx*ny*nz 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.

ArrayIndex3D returns indices that can be used for arrays which of variables stored sequentially in the order of the x, y, z (ijk-ordering). In stencil applications, other orderings are used to expect optimizations, for example, to increase the cache hit rate and to enable coalesced memory access. In order to change ordering without modifying user-written stencil functions, similar classes to access arrays with other ordering are also provided.

#### 4.2.2 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:

\[
\frac{\partial f}{\partial t} = \kappa \nabla^2 f,
\]

(1)

where \( f \) is a physical variable and \( \kappa \) is a diffusion coefficient.

Similar to `BoundaryExchange`, `BoundaryCondition` is initialized by a `Domain` object. After the initialization, a function that applies a boundary condition can be called. This class currently supports the Neumann and Dirichlet boundary conditions. Periodic boundary conditions can be applied to computation by using `BoundaryExchange` since these conditions need data transfer between subdomains. To apply more complicated boundary conditions to computation, the programmer needs to write the stencil functions that apply boundary conditions.

5. EXPERIMENTAL EVALUATION

To evaluate the proposed framework, we used a diffusion equation as a benchmark test. We also apply the proposed framework to a real stencil application; we show computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework.

5.1 Diffusion computation

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

\[
\frac{\partial f}{\partial t} = \kappa \nabla^2 f,
\]

where \( f \) is a physical variable and \( \kappa \) is a diffusion coefficient.

This computation needs three-stencil access in each direction; seven neighbor elements of the physical variable are used to update it on a center point of the grid. The boundary regions that have one-element-thick are needed for this computation.

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.

Figure 6 shows performance of diffusion computation written with the proposed framework and one of manual implementation as reference. The diffusion computation is performed using either one or two GPUs on a same node varying the mesh size from 64\(^3\) to 512\(^3\) in single precision. In the manual implementation using two GPUs, the MPI library is used for communication between them. In the framework case using two GPUs, performance results of diffusion computation with and without peer-to-peer access are shown in this figure. Handling multiple GPUs in a single process is required by peer-to-peer access, which tends to complicate the user code. From this reason, hand-written multi-GPU codes often adopt flat-MPI as their parallelization. Thus we do not use peer-to-peer access in the manual implementation in these evaluations. Since applications using this framework can run on CPU as well as GPU, we also show the performance obtained by using 1 CPU core and 4 CPU cores in this figure as reference.

As shown in this figure, in the one-GPU computation, the performance obtained by using the framework is higher than that by manual implementation; the performance of 194.2 GFlops is achieved by using the framework for a 512\(^3\)
mesh, which result is 1.04 times faster than the manual implementation code. Since this computation executes 13 floating-point operations (flop) and needs to read seven elements from the memory and write back one updated value to the memory, which is 32-byte access, to update each point of the grid in one time step, flop per byte ratio of this stencil computation is 0.41. Assuming that 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 194.2 GFlops is achieved by using the framework for a 512³ mesh, which is more than the estimated value, the stencil function implemented using the framework is likely to be well optimized. In the case of two GPUs, the framework improves the performance drastically compared with the manual implementation code, since GPUs can communicate to each other using peer-to-peer access in the framework instead of MPI communication. The performance of 353.7 GFlops is achieved using the framework, which is 1.4 times faster than the manual implementation code using two GPUs.

Figure 7 shows the weak scaling results of diffusion equation using TSUBAME 2.5. The performance of the manual implementation code and that of the code using the framework are shown in this figure. The calculations are performed in single precision. For the manual implementation, multi-GPU computation is parallelized by MPI without OpenMP (i.e., flat-MPI computation). Both codes use three GPUs per node. Note that while the framework handles three GPUs each node using OpenMP, the manual implementation code handles these three GPUs using three MPI processes. In this weak scaling measurements, each GPU computes a 1024 × 256 × 256 mesh. As shown in this figure, the performance of the code using the framework reaches that by the manual implementation. The weak scaling efficiency of the implementation using the framework is above 85% on 400 GPUs with respect to the 16-GPU performance.

5.2 3D Compressible Flow

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

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

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix}, \quad E = \begin{bmatrix} \rho u + p \\ \rho u v \\ \rho u w \\ \rho u e \end{bmatrix}, \quad F = \begin{bmatrix} \rho w \\ \rho w v \\ \rho w w + p \\ (p + e) w \end{bmatrix},$$

$$G = \begin{bmatrix} \rho w \\ \rho u w \\ \rho v w \\ \rho w w + p \\ (p + e) w \end{bmatrix}, \quad S = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

where ρ is density, (u, v, w) are velocity, p is pressure, and ε 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, which requires 13 neighbor elements of the each variable are used to update them on a center point of the grid in 3D computation. Since the above five variables are used for this simulation, GPU-GPU communications for them can be described as follows:

```c
BoundaryExchange *exchange = domain.exchange();
exchange->append(r);
exchange->append(ru);
exchange->append(rv);
exchange->append(rw);
exchange->append(re);
exchange->transfer();
```

Figure 8 shows computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework. We use 12 GPUs of TSUBAME 2.5 for this calculation. As shown in this figure, the real mesh-based applications can be written using the framework.
6. CONCLUSIONS

In order to improve the productivity of writing mesh-based applications optimized for multi-GPU systems and GPU-rich supercomputers, we have proposed a multi-GPU framework for these applications. From the viewpoint of 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++ languages. 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.

In order to perform intra-node GPU-GPU communications effectively, the proposed framework handles multiple GPUs on a same node by OpenMP threads with inter-node parallelization by the MPI library. The programmer can apply MPI with OpenMP parallelization to their user code without considering handling multiple GPUs on a single process, which often requires careful programming techniques. For stencil computation, the programmer writes only the stencil functions that update a grid point using its neighbor points, which are executed over grids by the framework. For multi-GPU computation, the framework provides C++ classes to write GPU-GPU communication effectively. The framework exploits peer-to-peer access between GPUs on a same node to improve performance of intra-node communication. Introducing overlapping communication with computation as an optimization and CUDA-aware MPI to improve the performance will be a subject of our future work.

For our evaluation, we performed a diffusion equation written by the framework on the TSUBAME2.5 supercomputer at the Tokyo Institute of Technology. The performance of the code using the framework reaches that by the manual implementation. In computation using two GPUs on a same node, the user code using the framework is 1.4 times faster than the manual implementation code thanks to utilizing peer-to-peer access. We also showed computational results of the Rayleigh-Taylor instability simulated by 3D compressible flow computation written by this framework.

7. ACKNOWLEDGMENTS

This research was supported in part by KAKENHI, Grant-in-Aid for Young Scientists (B) 25870223, Grant-in-Aid for Scientific Research (B) 23360046 and Grant-in-Aid for Young Scientists (B) 25870226 from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan, and 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”. 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.

8. REFERENCES


