Advanced High-productivity Framework for Large-scale GPU/CPU Stencil Computations

Takashi Shimokawabe *(Tokyo Institute of Technology)*
shimokawabe@sim.gsic.titech.ac.jp
http://www.sim.gsic.titech.ac.jp/Japanese/Member/shimokawabe

Grid-based physical simulations are important applications in the field of HPC. We are developing a high-productivity framework for multi-GPU and multi-CPU computation of stencil applications. The framework is implemented in the C++ language and automatically translates user-written functions that update a grid point and generates both GPU and CPU code. The programmers write user code just in C++ and can execute it on multiple GPUs with auto-tuning mechanism and the overlapping method to hide communication cost by computation. The same code can also executed on multiple CPUs with OpenMP without any change. The framework provides a data structure that supports element-wise computation, which allows us to write inline kernel code. This poster presents our proposed framework and its performance evaluation of diffusion computations running on multiple GPUs with auto-tuned overlapping method and on multiple CPUs with OpenMP.

1 Introduction

Grid-based physical simulations using stencil computations are one of the important applications running on supercomputers. We are 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 [1]. We are currently introducing further optimizations into this 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 auto-tuned overlapping technique to hide communication overhead by computation.

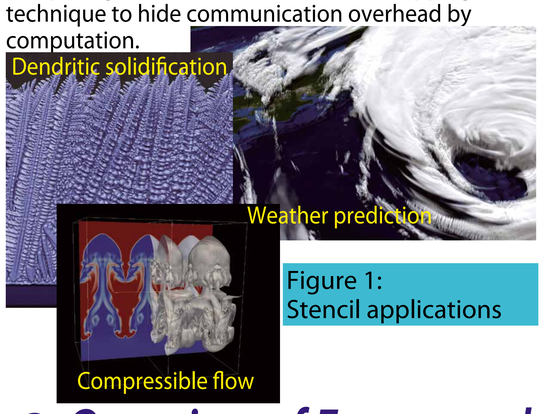


Figure 1: Stencil applications

2 Overview of Framework

- The proposed framework is designed for stencil applications with explicit time integration running on regular structured grids.
- The framework is intended to execute user programs on NVIDIA's GPUs and CPU.
- The framework is written in the C++ language and CUDA and can be used in the user code developed in the C++ language.
  --> Improving portability of both framework and user code and cooperation with the existing codes.
- The framework allows us to run code on multi-CPU with OpenMP and multiple GPU.
- The framework provides optimizations such as overlapping methods and auto-tuning.
- 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.

Reference

[1] T. Shimokawabe, T. Aoki and N. Onodera **"High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA,"** *in Proceedings of the 2014 ACM/IEEE conference on Supercomputing (SC'14)* , New Orleans, LA, USA, Nov 2014.

3 Programming Model

Data structure for grid data

- The framework introduces the following classes to improve productivity.
- **Range** class represents a 1D/2D/3D rectangular range.
- **ETArray** class has an array with its size and the position of its begin points.
- **ETArray** can be allocated on both CPU and GPU.

```
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);
```

Element-wise computation for arrays

- Stencil applications often use element-wise computations as well as stencil computations.
- Each expression related to **ETArray** generates a GPU/CPU kernel code based on itself by C++ expression template technique, which allows us to write kernel codes as inline codes.

```
int main() {
    ...
    // Each expression based on expression template generates an individual kernel.
    f = 1.0 + 2 * 4.0; // All elements of "f" are filled with 9.0
    g = 2.0 * sqrt(f); // All elements of "g" are filled with 6.0
    ...
```

Stencil Computation

- User-written stencil function (C++ funtor) that updates a grid point
- **ArrayIndex** represents the coordinate of the point where this function is applied.

```
struct Diffusion3d { // User-written stencil function
    __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;
    }};
```

- The **view** executes the stencil function with the appropriate way specified by **ETProperty**.
- The inside region of fn specified by **view** is updated by the stencil function.

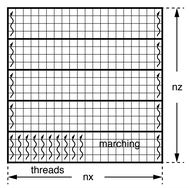
```
Range3D inside; // 3D rectangular range where stencil functions are applied.
ETProperty *prop = new DeviceProperty; // Select GPU execution
view(fn, inside, prop) = funcf<float>(Diffusion3d(), ptr(f), idx(f),
                                      ce, cw, cn, cs, ct, cb, cc); // Kernel
```

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

Auto-tuning for single GPU computation

- Since GPU performance often depends on the number of threads, execution of stencil functions with auto-tuning is effective to maximize performance.
- Tuning is done in early stage of time integration loop.

Tuning parameters	
# of threads in x direction	4, 8, 16, 32, 64, 128
# of threads in y direction	1, 2, 4, 8, 16
# of cell in z-marching	1, 2, 4, 8, 16




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

GPU-GPU communication

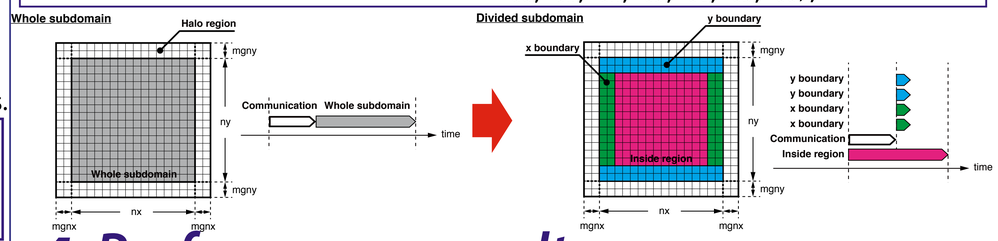
- Boundary regions of arrays appended by PBoundaryExchange::append are exchanged when PBoundaryExchange::transfer is executed.

```
PBoundaryExchange exchange(inside, rank, neighbor_connect);
exchange.append(f); // Boundary is decided by comparing inside with range of f.
exchange.transfer();
```

Overlapping method

- The **vieweb** function executes a stencil function and communication in parallel.
- The stencil function is divided into five regions in 2D, seven regions in 3D.
- The **prop** provides information and working resources required for this execution; for example, **DeviceOverlapProperty** provides CUDA streams for parallel execution.

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



4 Performance results

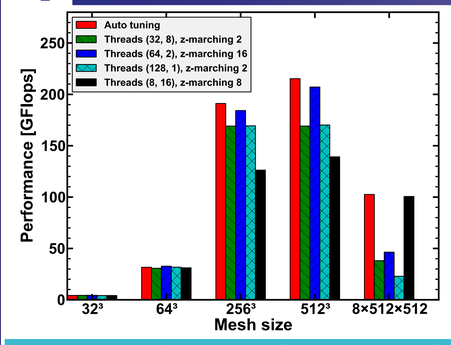


Figure 2: Single GPU Performance results of diffusion computation with auto-tuning running on Tesla K20X.

Performance results of diffusion computation, which is a fundamental equation used in fluid simulations, are shown. TSUBAME2.5 is used for these measurements.

TSUBAME 2.5 supercomputer at the Tokyo Institute of Technology

- Total 4224 NVIDIA Tesla K20X GPUs
- Each node of TSUBAME 2.5
  - 3 Tesla K20X GPUs attached to the PCI Express bus 2.0 × 16 (8 GB/s)
  - 2 sockets of the Intel CPU Xeon X5670(Westmere-EP) 2.93 GHz 6-core
  - 2 QDR InfiniBand

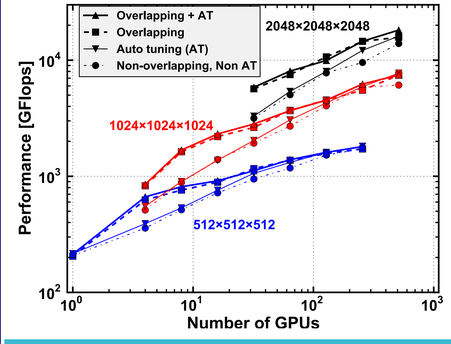


Figure 3: Strong scaling results of diffusion computation on TSUBAME2.5. Overlapping method or/and auto-tuning are used.

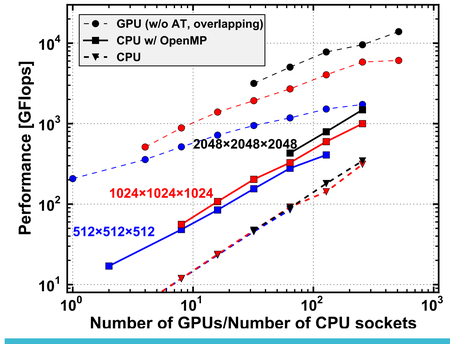


Figure 4: Comparison of strong scaling results of CPU (1core), OpenMP (CPU) and GPU (without optimization) of diffusion computation on TSUBAME2.5.