A porting and optimization of search for neighbour-particle in MPS method for GPU by using OpenACC

Takaaki Miyajima, Kenichi Kubota and Naoyuki Fujita
Numerical Simulation Research Unit, Aeronautical Technology Directorate,
Japan Aerospace Exploration Agency (JAXA)
7-44-1 Jindaiji-higashimachi Chofu Tokyo, Japan
miyajima.takaaki@jaxa.jp

ABSTRACT

Moving Particle Semi-implicit (MPS) method is a particle method used in fields such as computational fluid dynamics. It is classified as a particle method. Target fluids and objects are divided up into particles, and each particle interacts with its neighbour-particle. The search for neighbour-particle is the main bottleneck of the MPS method. In this paper, we port and optimize "search for neighbour-particle" part in MPS method for GPU by using OpenACC. It accounted for 56% of all the processing time. We present three different optimizations and evaluated them with three different data sets; 25,704, 224,910 and 2,247,750 particles. We also use four different GPUs; NVIDIA K20c, GTX1080, P100(PCIe) and P100(NVlink). As a result, P100(NVlink) GPU achieves 41.5 times speed-up compared with 24 MPI process CPU version when the number of particles is 2,247,750.

Keywords

OpenACC; GPU; Performance optimization; Moving Particle Semi-implicit; MPS

1. INTRODUCTION

The MPS method is developed for simulating fluid phenomena such as fragmentation of in-compressible fluids[2]. It is classified as a particle method and does not contain a stencil computation. An example of a simulation result is shown in Figure 1. The motion of each particle is calculated through interactions with neighbour-particle. The computational characteristics of MPS resemble those of smoothed particle hydrodynamics (SPH) or the N-body problem. The search for neighbour- particles is the main bottleneck since it requires a number of memory transactions. Although, there are some GPU implementations of MPS that are written in CUDA[6] [5] [7], this paper is among the first to adopt OpenACC for MPS method.

OpenACC is an emerging application programming interface (API) that supports GPU[1]. It provides compiler directives, runtime library routines, and environment variables. The users add directives and clauses to the existing code, and then the compiler automatically generates the CUDA code. The usage model of OpenACC is similar to that of OpenMP. OpenACC is the quickest, efficient and most portable way to port and optimize the existing code for GPUs. But it still requires knowledge of parallel computing and target GPU architecture.

In this paper, we present a porting and optimization of our in-house MPS program which is called "NSRU-MPS". NSRU-MPS is written in Fortran 95, and it was parallelized by MPI. Further optimizations such as exploiting thread-level parallelism were not done. Most time consuming subroutines in NSRU-MPS; the search for neighbour-particle is chosen to port. We give three different optimizations by using OpenACC. We evaluate them on four different GPUs; NVIDIA K20c, GTX1080, P100(PCIe version) and P100(NVlink version) and three different data sets. Additionally, we evaluate multicore implementation by using OpenACC.

This paper organized as follows. We first give a brief overview of NSRU-MPS in Section 2. Then we overview porting and optimization of existing program to GPU by using OpenACC. Section 4 describe the subroutines of NSRU-MPS are ported and optimized to GPU. The details of the optimization and evaluation are given in this section as well. The final section concludes the paper.

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2017) Bochum, DE, June 7-9, 2017.

2. MOVING PARTICLE SEMI-IMPLICIT METHOD

MPS is originally developed for simulating fluid dynamics such as fragmentation of in-compressible fluid[2]. An example of a simulation result is shown in Figure 1. A collapse of water column is a standard benchmark for MPS method. The motion of each particle is calculated through interactions with neighbour-particle. Accuracy of the calculation depends on the number of particles. For example, 4.0km × 3.5km tsunami analysis used 260 million particles[4].

2.1 Governing equations

MPS (Explicit MPS method) adopts a two-stage fractional step scheme, and each computational step calculates the following equations. Note that the search for neighbour-particle expressed in $\boldsymbol{r}_j - \boldsymbol{r}_i$ calculates the distance between target particle i and neighbour particle j. The following describes the calculation steps of MPS.

Proc 0) Set initial values of simulation

Proc 1) Calculate external forces

Proc 2) Move particles

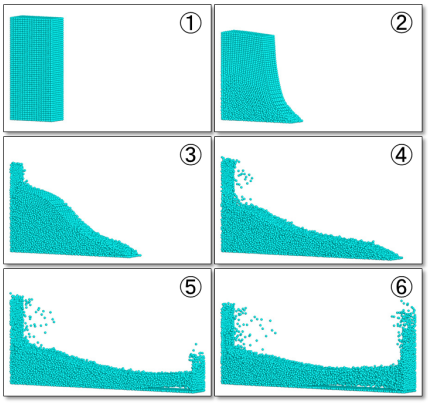

Figure 1: MPS simulation: A collapse of water column

Proc 3) Calculate pressure

Proc 4) Calculate gradient of pressure

Proc 5) Move particles

Proc 6) Increment the time step, and repeat Proc 0∼6

Proc 0) Set initial values of simulation

MPS first calculates the initial values of the simulation. λ^0 is an initial value of the weighted average of the distance to the neighbour-particle in the interaction area, n^0 is the initial particle number density, ρ is the initial density.

$$\lambda^0 = \frac{\sum_{j\neq i'}(|\boldsymbol{r}_j^0 - \boldsymbol{r}_{i'}^0|)^2\omega(|\boldsymbol{r}_j^0 - \boldsymbol{r}_{i'}^0|)}{\sum_{j\neq i'}\omega(|\boldsymbol{r}_j^0 - \boldsymbol{r}_{i'}^0|)} \tag{1}$$

$$n^0 = \sum_{j\neq i'}\omega(|\boldsymbol{r}_j^0 - \boldsymbol{r}_{i'}^0|) \tag{2}$$

where ω is function which calculates a weight of each neighbour-particle.

Proc 1) Calculate external forces

This step calculates the intermediate velocity of each particle. The second and third terms of the right hand side of the following equation are viscosity and gravity, respectively.

$$\boldsymbol{u}_i^* = \boldsymbol{u}_i^k + \Delta t\left(\nu\frac{2d}{\lambda^0 n^0}\sum_{j\neq i}(\boldsymbol{u}_j^k - \boldsymbol{u}_i^k)\omega(|\boldsymbol{r}_j - \boldsymbol{r}_i|) + g\right) \tag{3}$$

where k is the time step, t is the actual time, i and j are the particle numbers, ν is the kinematic viscosity, d is the number of space dimensions, g is the acceleration of gravity, $\boldsymbol{u}_i^k$ is velocity of particle i at time step k and $\boldsymbol{u}_i^*$ is intermediate velocity of particle i at time step k.

Proc 2) Move particles

This step calculates the intermediate position of each particle by using the result of Proc 1.

$$\boldsymbol{r}_i^* = \boldsymbol{r}_i^k + \Delta t\boldsymbol{u}_i^* \tag{4}$$

where $\boldsymbol{r}_i^k$ is the position of particle i at time step k.

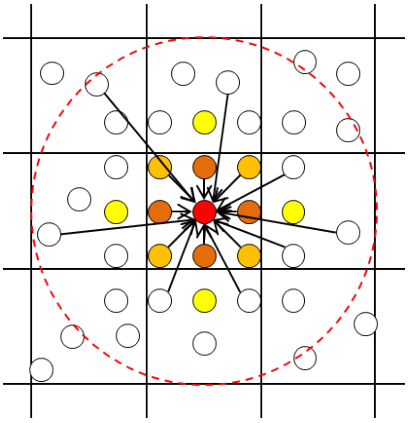

Figure 2: Search for neighbour-particle. The computational area is divided into multiple "buckets" (black boxes). The neighbor-particle are those within the interaction area (red circle).

Proc 3) Calculate pressure

This step calculates the pressure of each particle at the next time step.

$$n_i^* = \sum_{j\neq i}\omega(|\boldsymbol{r}_j^* - \boldsymbol{r}_i^*|) \tag{5}$$

$$P_i^{k+1} = c^2\frac{\rho^0}{n^0}(n_i^* - n^0) \tag{6}$$

where P_i^{k+1} is the pressure of particle i at time step $k+1$, c is the speed of sound, n_i^* is the intermediate particle number density of particle i at time step k.

Proc 4) Calculate pressure gradient

This step calculates the gradient of pressure in order to correct the intermediate velocity and position in Proc 5.

$$\langle\nabla P\rangle_i^{k+1} = \frac{d}{n^0}\sum_{j\neq i}\left(\frac{(P_i^{k+1} + P_j^{k+1})(\boldsymbol{r}_j - \boldsymbol{r}_i)}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^2}\omega_{grad}(|\boldsymbol{r}_j - \boldsymbol{r}_i|)\right) \tag{7}$$

where P_i^{k+1} is pressure of particle i at time step $k+1$.

Proc 5) Move particles

This step calculates correct velocity and position at the next time step.

$$\boldsymbol{u}_i^{k+1} = \boldsymbol{u}_i^* - \Delta t\left(\frac{1}{\rho}\nabla P\right)_i^{k+1} \tag{8}$$

$$\boldsymbol{r}_i^{k+1} = \boldsymbol{r}_i^* - \Delta t\left(\Delta t\left(\frac{1}{\rho}\nabla P\right)_i^{k+1}\right) \tag{9}$$

where $\boldsymbol{u}_i^{k+1}$ and $\boldsymbol{r}_i^k$ are the velocity and position of particle i at time step $k+1$.

Proc 6) Forward time step

This step calculates the actual time step (Δt) and increments the time. Δt is defined by the maximum value of $\boldsymbol{u}$.

2.2 Search for neighbour-particle

The search for neighbour-particle is one of the most time-consuming parts. Proc 1, 3 and 4 perform it in each time step. Equation 5 is a typical case. It computes the particle number density by searching for neighbour-particle and aggregates the physical value of the particles. Each particle drifts as the timesteps

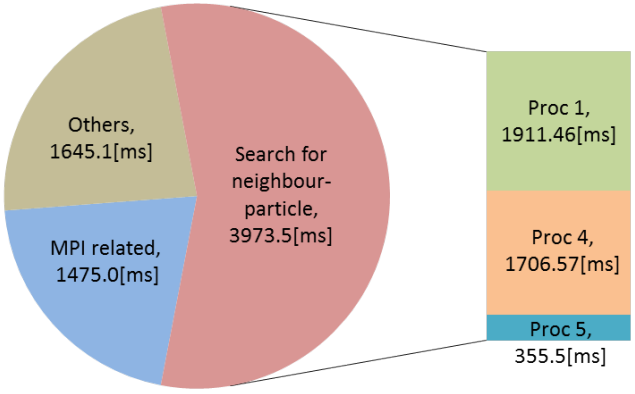

Figure 3: Profile of NSRU-MPS when the number of particles and MPI processes are 2,247,750 and 24, respectively.

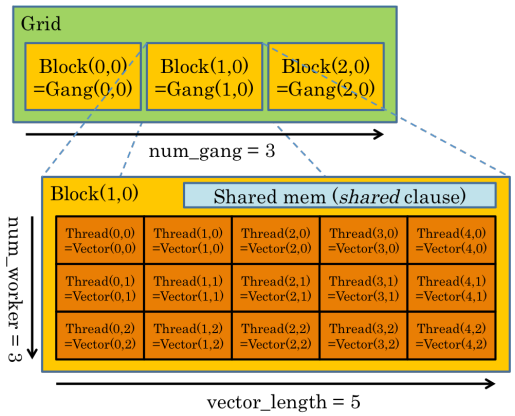

Figure 4: OpenACC's three level of parallelism. For NVIDIA's GPU, gang, worker, and vector correspond to block, warp, and thread of a warp, respectively.

progress and the neighbour-particle changes momentarily. Many studies have been devoted to reduce the time spent on search for neighbour-particles [6] [8] [5]. The bucket, linked-list, hash and book-keeping method have been proposed. In particular, NSRU-MPS adopts bucket method. Figure 2 illustrates the relationship between a bucket and search for neighbour-particle. The red particle is the target and the red circle depicts the interaction area. Pressure or particle number density of the centre particle is calculated using the distances with neighbour-particle. Weight of neighbour-particle only depends on the distance. A computational area is divided into multiple buckets and search for neighbour-particle is done only in adjacent buckets. The distances to each neighbour-particle in adjacent buckets are used to compute the value of the target particle (in red). Then, the weight of each neighbour-particle is calculated from the distance. Finally, all the weights are assigned to the centre particle. The above computation is done on all the particles in the simulation area. Thus, "search for neighbour-particle" consists of memory transactions to search, calculation of weight of a neighbour-particle and aggregation of all weights.

2.3 NSRU-MPS

Our three-dimensional MPS code, called "NSRU-MPS", is written in Fortran 95 and parallelized with Flat-MPI. The simulation domain is decomposed using MPI, but each MPI process is not parallelized using OpenMP or Pthread. Each particle has nine different physical values; position r_i, velocity u_i, pressure P_i, intermediate position r_i^*, intermediate velocity u_i^*, gradient of pressure $\langle \bigtriangledown P \rangle_i$, particle number density n_i, and particle number i. These values are single-precision floating-point and defined in an Structure of Array (SoA) style data structure. Each value is stored in an independent array whose length is the total number of particles. Additionally, a bucket number array, halo array, and a number of buffer arrays for MPI are prepared. All computations are performed in single precision floating point.

We conducted a preliminary experiment on NSRU-MPS in a single node environment consisting of two Intel Xeon E5-2697 v2 @ 2.7GHz CPUs and, 128GB of DDR3-12800 memory. There is 48 logical threads in total (2 CPUs × 12 cores × 2 hyper-threading). We use PGI Fortran compiler 16.10 and OpenMPI 1.10.5. Compile option is "-O3 -fast" and MPI option is "-bind-to socket -npersocket 12 -n 24". The time steps amounted to 200. Target simulation is a collapse of water column. Simulation area is 40[cm]×40[cm]×8[cm]. We prepare three different data sets which have different number of particles (and size of buckets). Simulation target is the same Figure 1. A small has 25,704 particles (35×35×7 buckets). A medium has 224,910 particles (70×70×14 buckets). A large has 2,247,750 particles (150×150×30 buckets). Each process computes one 24th of total particles in the preliminary experiment. For example, one process computes 1,123,875 particles when large data set.

Figure 3 shows a profile of NSRU-MPS when large data set. Search for neighbour-particle accounted for 56% of the total processing time. MPI data preparation and communication accounted for 21%. The rest of the functions consume less than 23%. We also evaluate a performance impact of the number of MPI processes as shown in Table 1. As the number of MPI processes are increased, a time spent on search for neighbour-particle (Search) becomes smaller. But, MPI related part becomes larger rapidly. In Section 4, we focus on Proc 1 which accounted for 27% of the total processing time.

Table 1: Relationship between processing time and processes

# of processes	2	4	8	12	24
Search	41122.1	22086.9	11468.3	4226.0	3973.5
MPI related	98.3	901.8	1068.2	1416.9	1475.1
Others	9757.1	5562.7	3139.9	2562.3	1645.2
Proc 1	19827.5	10655.7	5524.1	4239.6	1911.4

3. OPENACC

OpenACC can be applied to standard C, C++, and Fortran to specify regions of code for offloading from CPU to GPU. A directive is, in C, a #pragma, or, in Fortran, a specially formatted comment statement that is interpreted by a compiler to augment information about or specify the behavior of the program. that is interpreted by a compiler to augment information about or specify the behavior of the program.

3.1 Programming model

Programming model of OpenACC provides three levels of parallelism; Gangs, Workers and Vectors as shown in Figure 4. Gangs can have one or more workers that share resources (such as cache, streaming multiprocessor, etc.) Workers can have multiple vectors. Vectors work as SIMT threads. Gangs and vectors are mapped to particular grid or threads by using the *gang* and *vector* clauses. Knowledge of the target architecture (e.g. warp size) is required so as to achieve the best speedup. In NVIDIA's CUDA, gang, worker, and vector correspond to block, warp, and thread of a warp, respectively. The size of gang is equal to the product of the number of workers and the vector length [3]. OpenACC provides a way of thread mapping; the number of gang, workers and vector can be configured by *num_gang*, *num_workers*, and *vector_length*, respectively.

3.2 Directives and clauses

The *kernels* and *parallel* directives are provided for parallel computing. The *data* directive is used for data management between CPU and GPU. All the directives can have clauses such as *async*, *gang* or *copy* so as to optimize the kernel. The *loop*

directive is used in the parallel region which is specified using the *kernels* and *parallel* directives.

kernels directive. Parallel regions which is specified using the *kernels* directive are automatically analyzed loop structures and the necessary data by OpenACC compiler. The compiler identifies loops that can be parallelized, and maps abstract parallelism in the loops to hardware parallelism, and then generates actual kernels for GPU. The compiler also identifies the data used in the parallel regions and automatically transfer them from CPU to GPU. It often requires additional information to utilize more parallelism in the loops.

parallel directive. The *parallel* directive plays a key role in optimizing loops. It is used for coarse grained (e.g. a large nested loop) optimization. Unlike the *kernels* directive, they provide flexibility for users and often achieve higher performance. On the other hand, users have a responsibility to express dependencies or ensure the correctness of the result. It can have *loop* directive as a clause. By using these clauses, users are able to optimize the parallel region. Without *loop* directive, each loop iteration does not run in parallel.

loop directive and clauses. The *loop* directives tells the compiler to map each iteration of the loop to the thread. It deals with fine grained (e.g. a single loop in a nested loop) optimization. It can have clauses shown in Table 2. The users can arbitrary map the loop iterations to the target device by using *gang(N)*, *worker(N)* and *vector(N)* clauses. The *collapse* clause is used to obtain larger amounts of parallelism from nested loop. The *seq* clause tells the compiler that the following loop cannot be run in parallel. The *atomic* directive is required to obtain the correct result when the users add the *independent* clauses since all the operations in the loop run in parallel.

Table 2: OpenACC's clauses and functions.

clause	Function
gang(N)	map the loop to the N thread block
worker(N)	map the loop to the N warp
vector(N)	map the loop to the N thread
seq	run the loop sequentially
collapse(N)	make a N-nested loop to one large loop
independent	run each iteration independently
atomic	perform atomic operation

data directive. For optimization of data movements, OpenACC provides the *acc data* directive, which transfers data between the host and the device memory at an arbitrary timing. Reducing the number and amount of data transferred between host and the device is a key to success. It is important to keep the data as much as possible in the device's memory and transfer them only necessary.

4. PORTING AND OPTIMIZATION

In this section, we describe optimizations of search for neighbour-particle in Proc 1. It has a quintuple nested loop that calculates external forces. We use four different computation nodes consisting of two CPUs and two GPUs. Nodes have different GPU; Tesla K20c, GeForce GTX1080, Tesla P100(NV-link) and P100(PCIe). The details of each GPU are shown in Table 3. The compiler use for K20c, GTX1080 and P100(PCIe) is PGI Fortran Compiler for x86-64 16.10. The compiler used for P100(NVlink) is PGI Fortran Compiler for linuxpower 16.10 since host CPU is IBM POWER8. A compile option is "-acc -ta=nvidia, cuda8.0, fastmath, cc60". In the case of Tesla K20c, "cc35" is used instead of "cc60" since the compute capability is 3.5. OpenMPI 1.10.2 is used with "mpiexec -n 2" option. One process is assigned to each CPU in a single node. (Note that Proc 1 doesn't contain MPI communication.) Processing time is an average for the first 200 time steps and measured using the *MPI_Wtime* function. The simulation setup is the same as described in Section 2.

Table 3: A spec of GPUs used in evaluation.

GPU	Single FP [TFLOPS]	Proc. Freq. [MHz]	CUDA Cores	Memory BW [Gbps]
K20c	3.5	706	2,496	208
GTX1080	8.8	1,733	2,560	320
P100 (PCIe)	9.3	1,303	3,584	732
P100 (NVlink)	10.6	1,406	3,584	732

4.1 Neighbour-particle search in Laplacian_u subroutine

As described in Section 2.2, the search for neighbour-particle in Proc 1 accounted for 27% and took 1911.4[ms] to compute. Search for neighbour-particle computes distance and aggregate all the adjacent particles in the neighbor buckets. Each particle can be computed independently; thus, this is massively data parallel. On the other hand, aggregation pose a read-after-write (RAW) hazard when this part is parallelized. In this section, we describe three optimizations of the loops and thread mapping.

Listing 3 shows the pseudo code of the original implementation. It consists of a quintuple nested loop. The first loop designates the target particles. The second, third, and fourth loops are for traversing 3×3×3 (=27) adjacent buckets. The fifth loop searching for neighbour-particle and aggregates the physical value of the particles. Except for the aggregation in the fifth loop, all the loops can be computed independently. But the number of iterations of fifth loop differs from particle to particle. This is because the number of neighbour-particle changes each timestep. A calculation of distance and weight on line 17 is the most computational intensive part in the subroutine. The memory access pattern cannot be coalesced. Line 13 is a typical case.

Listing 1: Pseudo code of search for neighbour-particle

```
1  ! for all the particles
2  do−loop1: target_ptcl = 1,all_ptcl
3    ib = bucket_num[m]
4    ! traverse adjacent buckets (3−dim: 3x3x3=27)
5    do−loop2: x=x1,x2
6      do−loop3: y=y1,y2
7        do−loop4: z=z1,z2
8          ibb = get_adj_bucket_num(x,y,z)
9          num_of_ptcl = get_num_of_ptcl_in_bucket(ibb)
10         ! accumulate all the neighbour−particle
11         do−loop5: np = 1,num_of_ptcl ! indefinite loop
12           if (ptcl_is_in_halo)
13             lcr = ptcl_halo[np] ! random access
14           else
15             lcr = ptcl[np] ! random access
16           end if
17           dist = sqrt(dot_product(m, lcr)) ! get distance
18           weight = get_weight(dist)
19           accum = accum + phys(weight) ! aggregation
20     m_phys[m] = m_phys[m] + accum ! in−place add
```

Naive optimization. "Naive" optimization is simple since each particle is assigned to each CUDA thread. Inner-loops (do-loop2, 3, 4 and 5) run in sequentially. We add *acc kernels* and *acc loop gang vector* to the first loop so as to assign CUDA thread per particle. Figure 5 shows a thread assignment of this optimization. The total number of CUDA threads is the same as the number of particles. When the number of particles is 14,688, the grid size becomes 115 (= 14,688 ÷ 128). Theoretical occupancy also becomes 100% and computational resource of GPU is fully used. *acc loop collapse(3) seq* is added to do-loop2 and consecutive thee loops (do-loop2, 3 and 4) are unrolled. For the do-loop5, we add *acc loop seq* so as to run the loop sequentially. We also evaluate a performance impact of the vector size. 64, 128, 256 and 512 vector are evaluated, but there is no difference.

Listing 2: Pseudo code of "Naive" optimization

```
1  !$acc kernels
```

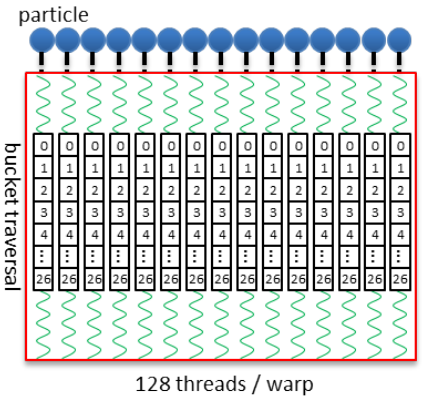

Figure 5: "Naive" optimization.

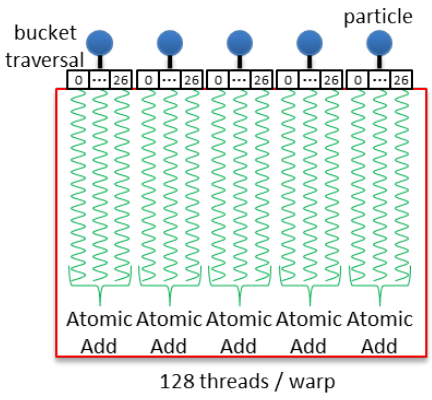

Figure 6: "Atomic" optimization.

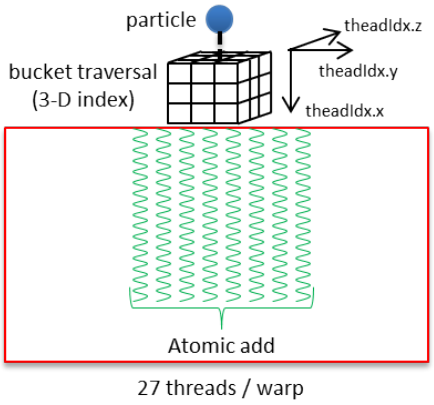

Figure 7: "3-D" thread optimization.

```
!$acc loop gang vector(128)
do−loop1: target_ptcl = 1,all_ptcl
  ...
  !$acc loop collapse(3) seq
  do−loop2: x=x1,x2
    do−loop3: y=y1,y2
      do−loop4: z=z1,z2
        ...
        !$acc loop seq
        do−loop5: np = 1,num_of_ptcl
          ...
```

Atomic Optimization. "Atomic" optimization increase the number of available threads and uses atomic operation. A bucket traversal of each particle is assigned to each CUDA thread as shown in Figure 6. The total number of CUDA threads is 27 times larger than "Naive". When the number of particles is 14,688, the grid size becomes 3,099 (= 14,688 × 27 ÷ 128). Theoretical occupancy becomes 100% and computational resource of GPU is fully used. Additionally, an atomic operation is used for in-place add. We add *acc parallel* and *acc loop collapse(4) gang vector* to the first loop so as to assign CUDA thread per bucket traverse. In-place add and bucket index calculation are moved from do-loop1 to do-loop4.

Listing 3: Pseudo code of "Atomic" optimization

```
!$acc parallel
!$acc loop collapse(4) independent gang vector(128)
do−loop1: target_ptcl = 1,all_ptcl
  ...
  do−loop2: x=x1,x2
    do−loop3: y=y1,y2
      do−loop4: z=z1,z2
        ...
        ! moved here from do−loop1
        ib = bucket_num[m]
          !$acc loop seq
          do−loop5: np = 1,num_of_ptcl
            ...
        ! moved here from do−loop1
        !$acc atomic update
        m_phys[m] = m_phys[m] + accum ! in−place add
        !$acc end atomic
```

3-D Thread Optimization. "3-D thread" optimization uses 3-dimensional thread index and atomic operation. Index of adjacent buckets becomes "(x,y,z)" as shown in Figure 7. *threadIdx.x*, *threadIdx.y* and *threadIdx.z* are used for each dimension. Although the total number of CUDA threads is 27 times larger than "Naive", block size decreases to 27. Occupancy becomes lower and computational resource of GPU cannot be fully used. When the number of particles is 14,688, the grid size becomes 14,688 (= 14,688 × 27 ÷ 27).

Listing 4: Pseudo code of "3-D thread" optimization

```
!$acc kernels
!$acc loop independent
do−loop1: target_ptcl = 1,all_ptcl
  ...
  !$acc loop vector(3)
  do−loop2: x=x1,x2
    !$acc loop vector(3)
    do−loop3: y=y1,y2
      !$acc loop vector(3)
      do−loop4: z=z1,z2
        ...
        ! moved here from do−loop1
        ib = bucket_num[m]
          !$acc loop seq
          do−loop5: np = 1,num_of_ptcl
            ...
        ! moved here from do−loop1
        !$acc atomic update
        m_phys[m] = m_phys[m] + accum ! in−place add
        !$acc end atomic
```

Multicore Implementation. "Multicore" implementation is compiled with the *-ta=multicore* option. The added directives and clauses are completely the same as those in the "Naive" optimization. We use "mpiexec -bind-to socket -n 2" for multicore implementation on Xeon CPU. The number of thread per process is 12, thus 24 threads are totally launched. The processing time is 3407.4[ms]. It is 0.56 times speed-up compared with the original (single-thread 24 MPI processes). Note that the processing time increases as the number of process increased.

4.2 Evaluation

Figure 8 and Table 4 show the result of optimizations. A combination of P100(NVlink) and "Naive" is the fastest except for small data set. In the case of large data set (2,247,750 particles), it achieves ×45.1 speed-up compared with 24 MPI processes on Xeon CPU. In case of P100(PCIe) and P100(NVlink), "3-D Thread" is faster than "Naive" for the small data set. The difference between "Naive" and "Atomic" is an intensity of computation. "Naive" doesn't have atomic operation, has more computations and has less in-flight memory transactions compared with "Atomic". "3-D Thread" has less atomic operation, less computations and less in-flight memory transactions because of the low occupancy.

Figure 9 shows a stall reason of the large data set. "Data

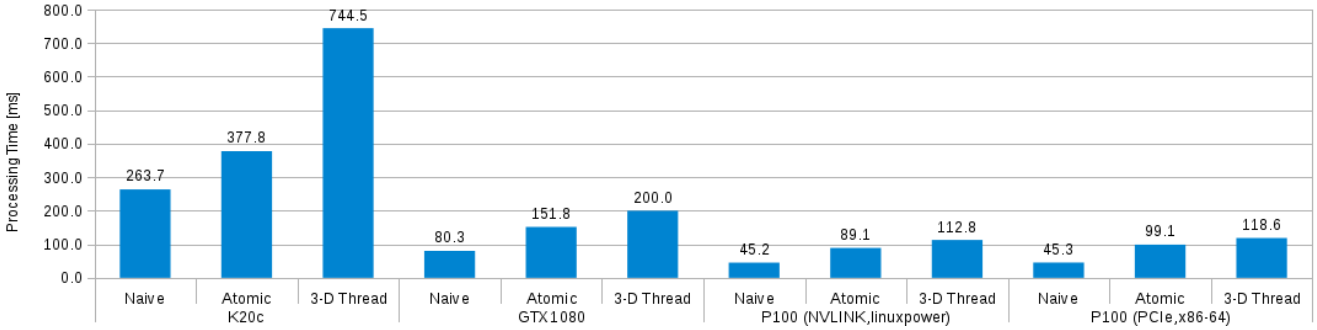

Figure 8: Processing time on large data set (2,247,750 particles). "Naive" is the fastest for all the GPUs.

Table 4: Processing time on small and medium data sets. "Naive" is the fastest for the middle, but the fastest optimization is different for the small data set.

GPU	K20c			GTX1080			P100 (NVlink)			P100 (PCIe)		
Optimization	Naive	Atomic	3-D	Naive	Atomic	3-D	Naive	Atomic	3-D	Naive	Atomic	3-D
Small (25,704 particles) [ms]	8.6	6.2	15.9	1.4	1.9	2.5	1.7	1.7	1.5	1.6	1.3	1.2
Medium (224,910 particles) [ms]	26.7	36.0	69.4	8.7	15.1	20.6	4.3	4.3	11.1	4.7	4.8	11.6

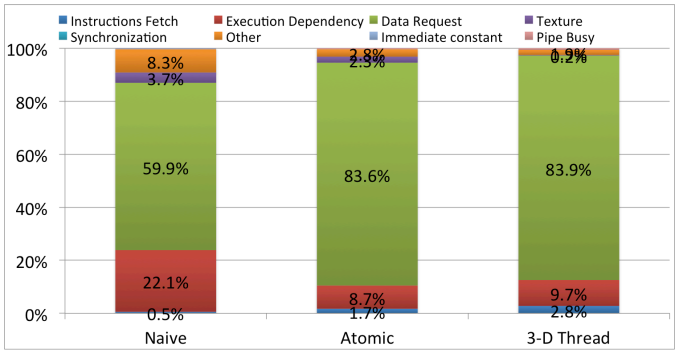

Figure 9: Stall reason of three optimizations of P100(PCIe)

request" [1] is responsible for 83% of stall reason in the case of "Atomic" and "3-D Thread". The number of in-flight memory transactions of "Atomic" and "3-D Thread" is too much high for current GPUs. Achieved DRAM read throughput of "Naive", "Atomic" and "3-D Thread" is 2.90GB/s, 1.26GB/s and 2.39GB/s respectively. Search for neighbour-particle is memory bandwidth bound and hard to coalesce the memory access.

Presented optimization can be applied to Proc 3 and 4. If Proc 3 and 4 would be speed-ed up 45.1 times as well, time spent on search for neighbour-particle decreased to 88.1[ms]. Additionally, time spent on MPI related part is 45[ms] since two MPI processes are used in evaluation. Overall processing time will be 1831.5 [ms](88.1 + 98.3 + 1645.1) and achieve 3.87 times speed-up compared with 24 MPI processes on Xeon CPU.

5. CONCLUSION

In this paper, we present a porting and optimization of search of neighbour-particle in our MPS program. Search for neighbour-particle in Proc 1 which accounted for 26% of the total processing time is chosen. We show three optimization; "Naive", "Atomic" and "3-D thread", and evaluate with three different data sets. Evaluation are conducted on four different GPUs; NVIDIA K20c, GTX1080, P100(PCIe) and P100(NVlink). When the number of particles are 2,247,750, a combination of "Naive" and P100(NVlink) achieves 41.5 times speed-up compared with the original version (a single-thread 24 MPI processes) on Xeon CPU. "Naive" is the fastest for all the GPUs when the data set is large enough (224,910 and 2,247,750 particles). In the case of data set is small (25,704 particles), fastest optimization is different.

We will conduct a detailed analysis of our optimizations and optimize the rest of our MPS program. We also compare our OpenACC optimizations with another CUDA implementations.

[1] Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding.

Acknowledgment

The authors would like to thank Akira Naruse at NVIDIA for his helpful comments and discussion on our work.

6. REFERENCES

[1] Openacc home — www.openacc.org. http://www.openacc.org/.

[2] S. Koshizuka and Y. Oka. Moving particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear Science and Engineering*, 123:421–434, 1996.

[3] J. Larkin. *OpenACC Programming & Best Practices Guide*, July 2015.

[4] K. Murotani, S. Koshizuka, T. Tamai, K. Shibata, N. Mitsume, S. Yoshimura, S. Tanaka, K. Hasegawa, E. Nagai, and T. Fujisawa. Development of hierarchical domain decomposition explicit mps method and application to large-scale tsunami analysis with floating objects. *Journal of Advanced Simulation in Science and Engineering*, 1(1):16–35, 2014.

[5] K. Murotani, I. Masaie, T. Matsunaga, S. Koshizuka, R. Shioya, M. Ogino, and T. Fujisawa. Performance improvements of differential operators code for mps method on gpu. *Computational Particle Mechanics*, 2(3):261–272, 2015.

[6] W. Seiya, A. Takayuki, T. Satori, and S. Takashi. Neighbor-particle Searching Method for Particle Simulation Based on Contact Interaction Model for GPU Computing. *IPSJ Transactions on Advanced Computing Systems*, 8(4):50–60, 2015.

[7] Y. Sota, A. Watanabe, and T. Kojima. Accerelation of the moving paricle semi-implicit method through multi-gpu parallel computing with dynamic domain decomposition. *Journal of Japan Society of Civil Engineers, Ser. A2 (Applied Mechanics (AM))*, 69(2), 2013.

[8] H. Sun, Y. Tian, Y. Zhang, J. Wu, S. Wang, Q. Yang, and Q. Zhou. A special sorting method for neighbor search procedure in smoothed particle hydrodynamics on gpus. In *Parallel Processing Workshops (ICPPW), 2015 44th International Conference on*, pages 81–85, Sept 2015.