An optimization of search for neighbour-particle in MPS method for Xeon, Xeon Phi and GPU by using directives

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

ABSTRACT

Moving Particle Semi-implicit (MPS) method is a particle-base simulation used in fields such as computational fluid dynamics. Target fluids and objects are divided up into particles, and each particle interacts with its neighbour-particle. This process is called “search for neighbour-particle" and the most time consuming part in the MPS method. In this paper, we port and optimize search for neighbour-particle for NVIDIA GPU, Intel Xeon CPU and Intel Xeon Phi by adding directives. We present two different optimizations and evaluated them with a standard particle-base simulation benchmark. particles. As a result, NVIDIA Tesla P100 (NVlink) GPU achieves 5.2 times speed-up compared with Intel Xeon Gold 6150 CPU when the number of particles is 224,910.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms*; • **Applied computing** → *Aerospace*;

KEYWORDS

Moving Particle Semi-Implicit, GPU, Xeon Phi, Search for neighbour-particle

ACM Reference Format:
Takaaki Miyajima, Kenichi Kubota and Naoyuki Fujita. 2018. An optimization of search for neighbour-particle in MPS method for Xeon, Xeon Phi and GPU by using directives. In *Proceedings of International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia).* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/

1 INTRODUCTION

The MPS method is developed for simulating fluid phenomena such as fragmentation of in-compressible fluids [1]. An example of MPS simulation 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. There are some implementations for GPU and many core [3] [5] [6]. Accuracy of the calculation depends on the number of particles. For example, 4.0km × 3.5km tsunami analysis used 260 million particles[2].

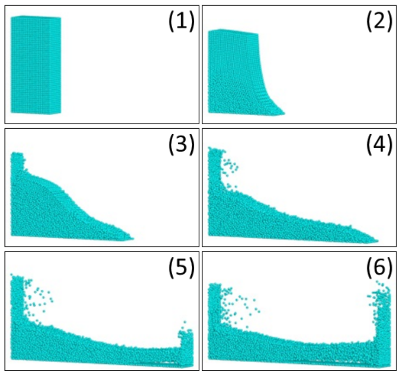

Figure 1: MPS simulation: A collapse of water column.

In this paper, we present a porting and optimization of our in-house MPS program; “p-flow". p-flow is written in Fortran 95, and it was parallelized by MPI. Further optimizations such as exploiting thread-level parallelism were not done. The search for neighbour-particle is ported and parallelized for GPU, Xeon and Xeon Phi by using OpenACC and OpenMP. We give two different optimizations for GPU and one optimization for Xeon/Xeon Phi. We evaluate them on dual NVIDIA Tesla P100 (NVlink) GPU, dual Intel Xeon Gold 6150 and single Xeon Phi 7210. When the number of particles is 224,910, processing time of each system is 6.7[ms], 35.1[ms] and 104.1[ms], respectively.

2 P-FLOW: IN-HOUSE MPS CODE

p-flow is our in-house MPS program. It adopts explicit MPS method for large scale simulation. It consists of two-stage fractional step scheme, and each computational step performs the following processing steps. Proc 1: Calculate external forces. Proc 2: Move particles. Proc 3: Calculate pressure. Proc 4: Calculate gradient of pressure. Proc 5: Move particles. Proc 6: Increment the time step, and repeat Proc 1~6. Each particle has nine different physical quantities; position, velocity, pressure, intermediate position, intermediate velocity, gradient of pressure, particle number density, and particle number. These quantities are single-precision floating-point and defined in an Structure of Array (SoA) style data structure.

2.1 Search for neighbour-particle

The search for neighbour-particle is the most time-consuming parts in p-flow. It is performed several times in each time step. The following equation to calculate density which is performed in Proc 3 is a typical case. $n_i^* = \sum_{j \neq i} \omega(|\boldsymbol{r}_j^* - \boldsymbol{r}_i^*|)$, where n_i^* is the intermediate particle number density of particle i, ω is weight function

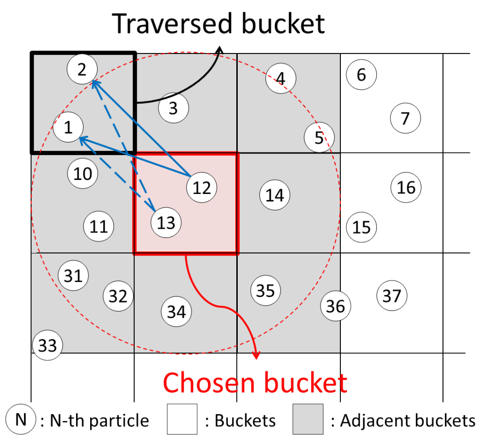

Figure 2: Search for neighbour-particle with bucket.

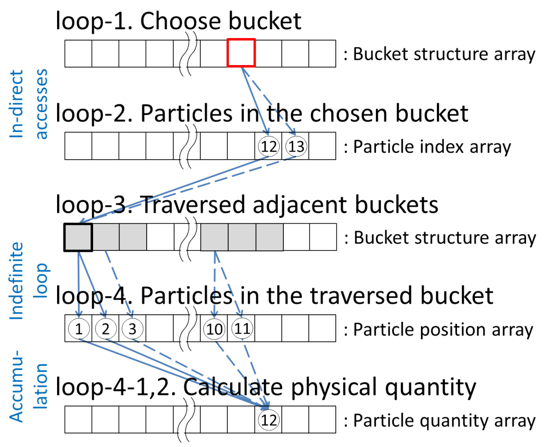
loop-1. Choose bucket
: Bucket structure array
In-direct accesses
loop-2. Particles in the chosen bucket
: Particle index array
loop-3. Traversed adjacent buckets
: Bucket structure array
Indefinite loop
loop-4. Particles in the traversed bucket
: Particle position array
Accumu-lation
loop-4-1,2. Calculate physical quantity
: Particle quantity array

Figure 3: Memory access pattern of search for neighbour-particle.

and $\boldsymbol{r}_i^*$ is the intermediate position of particle i. Each particle drifts as the timesteps progress and the neighbour-particle changes momentarily. Some studies have been done to reduce the time spent on search for neighbour-particles [3] [5] [7]. The bucket, linked-list, hash and book-keeping method have been proposed.

p-flow adopts bucket with linked-list method as shown in Figure 2. It illustrates a two-dimensional for brevity. The simulation domain is decomposed by multiple regular hexahedrons. It is called “buckets" (black boxes). Interactions are considered only in adjacent buckets. The neighbor-particle are those within the interaction area (red circle). Actual calculation of search for neighbour-particle is as follows. First, the distances to each neighbour-particle are calculated. Then, the weight of each neighbour-particle is calculated from the distance. Finally, all the weighted physical quantities are accumulated to the centre particle. The above computation is done on all the particles in the simulation area. The size of bucket is 3.1 times large than that of particle and the maximum number of particles in bucket is 33. The number of particles in bucket changes every time step.

In the case of p-flow, search for neighbour-particle for density calculation consists of quintuple nested loop as follows. Distance, weight and physical quantity are calculated in loop-4.

(1) loop-1: Choose a target bucket. (red bucket in Figure 2)
(2) loop-2: Pickup a target particle (particle 12) in the target bucket.
(3) loop-3: Traverse 3×3×3 adjacent buckets. (3×3 in the Figure 2)
(4) loop-4: Access particles (particle 1∼16) in adjacent buckets.

Figure 3 illustrates a memory access pattern of the loop. Particles in the same chosen bucket traverse the same buckets and access the same particles. In the Figure 2, particle 12 and 13 first access the same traversed bucket in the upper left of figure. Then they access the particles 1 and 2 to calculate distance, weight and physical quantity. And then they access the upper middle bucket in figure as a next iteration of loop-1. The nested loop is not easy to vectorize or improve cache utilization. There are four large reasons. First, all the loops contains indirect accesses. Second, loop-4 is indefinite loop since the number of particles in a traversed bucket is uncertain. Third, vectorization is not easy since each target particle can traverse different bucket and access different particle. Fourth, not easy to utilize cache since adjacent particles changes time by time. On the other hand, each particle can be compute independently and there are no particular order to traverse and accumulate quantity at loop-3 and 4. Thus, a number of parallelism can be utilized.

2.2 Preliminary evaluation

We conducted a preliminary evaluation on p-flow in a single node environment consisting of two Intel Xeon Gold 6150 @ 2.7GHz CPUs and, 192GB of DDR4-2400 memory. There is 72 logical threads in total (2 CPUs × 18 cores × 2 hyper-threading). We use Intel Fortran compiler 17.0.4 and Intel MPI. 72 processes are used and it achieves highest performance when Hyper-Threading is enabled. Compile option is “-O3 -fpp -xHOST -fp-model fast=2" and MPI option is “-bind-to socket -npersocket 36 -n 72". Processing time is an average for the first 200 time steps and measured using the MPI_Wtime function. Target simulation is a collapse of water column as shown in Figure 1. Simulation area is 40[cm]×40[cm]×8[cm] (70×70×14 buckets). The number of particles is 224,910. In the preliminary evaluation, search for neighbour-particle accounted for 35.8% of the total processing time. MPI data preparation and communication accounted for 27.3%. The rest of functions including dynamic domain decomposition accounted for 39.3%. Density calculation takes 47.8[ms].

3 OPTIMIZATION

In this section, we gives two optimizations; “Bucket per thread" and “Bucket per thread block". Target subroutine is search for neighbour-particle in density calculation. OpenMP and OpenACC are used for Xeon/Xeon Phi and GPU, respectively.

3.1 Optimization 1: Bucket per thread

In this optimization, each bucket (loop-1) is assigned to each thread in CPU/GPU as shown in Figure 4. loop-2×4 are processed sequentially by each thread. Although all the memory accesses (bucket

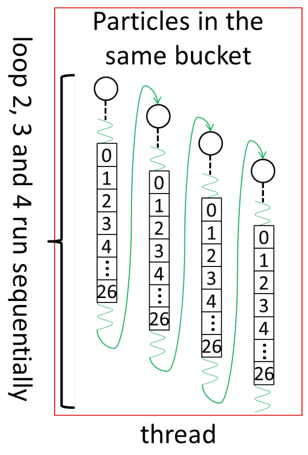
Particles in the same bucket
loop 2, 3 and 4 run sequentially
0
1
2
3
4
26
thread

Figure 4: Bucket per thread.

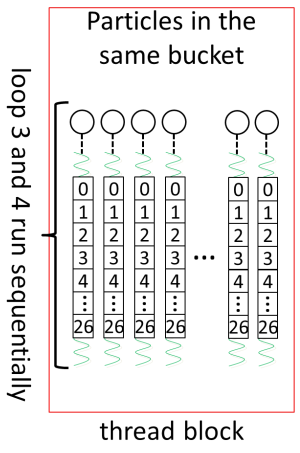

Figure 5: Bucket per thread block.

traversal and particles in the traversed buckets) of first particle (particle 8 in the Figure 2) become cache miss, memory accesses of second particle become cache hit if the cache for one thread is large enough. Thus this optimization requires larger cache compared with Optimization 2. We calculate the maximum memory size of each physical quantity. The number of traversed buckets in three-dimensional simulation is 27 (= 3×3×3) and the maximum number of particles in the 27 buckets is 891 (= 33×27). The maximum memory size of each physical quantity of neighbour-particle is 3,564 bytes (= 891×4 bytes). In the case of density calculation, four physical quantities (x, y, z coordinate and density) are required. 14,256 bytes (= 4×3,564 bytes) is needed to cache all the physical quantities in each loop-1 iteration. Additionally, bucket management data structure are required as well. The L1 data cache size of Xeon and Xeon Phi node are larger than 14,256 bytes as shown in Table 1. Hence, all the physical quantities in each loop-1 iteration can be stored in L1 data cache. In the case of Tesla P100, L1 data cache is shared among 32 CUDA threads in warp. It means that each CUDA thread can only use 445 bytes as a cache and too small to cache the physical quantities.

For Xeon and Xeon Phi, "$omp parallel do" directive is added on the loop-1. "schedule" clause is added for load balancing since the number of particles in chosen bucket is different. Three scheduling policy (static, dynamic and guided) are evaluated. For GPU, "$acc parallel loop gang vector" directive is added on the loop-1.

3.2 Optimization 2: Bucket per thread block

In this optimization, each bucket (loop-1) is assigned to thread block of GPU as shown in Figure 5. This optimization is applied only to GPU. We tried to perform similar optimization by adding "$omp simd" directive on loop-4 for Xeon/Xeon Phi. But, Intel compiler doesn't generate vector instructions. We will try to vectorize the loop-4 by using intrinsics in future work. Each particle in the same bucket (loop-2) is assigned to threads in the same thread block. loop-3 and 4 are processed sequentially by each thread. This optimization makes memory access of threads simple. All the threads in a chosen bucket access the same address since each particle in the same chosen bucket traverses the same adjacent buckets and particles in loop-3 and 4. It means that, CUDA threads in the same warp can share the physical quantities. Tesla P100 has 24 KB L1 cache and it is enough to store all the physical quantities for one loop-1 iteration. In terms of cache utilization, optimization 2 is much more efficient than that of optimization 1. If the particles in traversed bucket is aligned in global memory, eight particles are stored in cache at once since NVIDIA Tesla P100 fetches 256bit data by single memory transaction. On the other hand, memory access is not coalesced and bandwidth utilization is still low. When the number of particles in bucket is 32, utilization of CUDA thread is 100%. But average utilization of CUDA thread is low since the number of particle in bucket often becomes less than 32.

For GPU, "$acc parallel vector_length(N)" and "$acc loop gang" directive is added on the loop-1 as well. $acc loop vector" is added on loop-2. Additionally, "$acc loop seq" is added on loop-4. *vector_length(N)* clause determines the number of threads in thread block. 16, 32, 64 and 128 are evaluated.

4 EVALUATION

We use four different computation nodes for evaluation; Intel Xeon Gold 6150, Xeon Phi KNL 7210, NVIDIA Tesla P100 (PCIe) and P100 (NVlink). Except for Xeon Phi, computation node has two CPUs and two GPUs. The details of each node are shown in Table 1. For Xeon and Xeon Phi, Intel Fortran compiler ver.17.0.4 is used. A compile option is "-qopenmp -O3 -xCOMMON-AVX512 -fp-model fast=2". One process is assigned to each CPU in a single node. Thread affinity for Xeon and Xeon Phi is "granularity=fine,compact". For Xeon Phi, Flat-Quadrant mode and MCDRAM is used by "numactl −membind=1". For P100 (NVlink) and P100 (PCIe), PGI Fortran compiler ver.17.7 is used. A compile option is "-acc -ta=nvidia:cc60, cuda8.0, fastmath". OpenMPI 1.10.5 is used with "-bind-to socket -npersocket 1 -n 2" option. Processing time is measured in the same way as the preliminary evaluation. Note that elapse time of data transfer between CPU and GPU is ignored. The simulation setup is the same as described in Section 2.

Table 1: Specification of computation nodes.

Computation Node	Performance [TFLOPS]	Clock [GHz]	The # of threads	Mem BW [GB/s]	L1 cache [KB]
Xeon Gold 6150	N/A	2.7	36	256	512
KNL 7210	5.3	1.3	256	480	32
P100 (PCIe)	9.3	1.1	3,584	732	24
P100 (NVlink)	10.6	1.3	3,584	732	24

4.1 Xeon and Xeon Phi

Table 2 shows evaluation of optimization 1 and scheduling policies. "KNL 7210 (Single)" shows that actual measured processing time on one KNL 7210. Best configuration for both processor is "dynamic" (chunk size 1). In the case of Xeon Gold 6150, "dynamic" is 14.8% faster than "guided". In the case of KNL 7210, "dynamic" is 36.1% faster than "guided". This result implies that load balancing is more important for KNL 7210. "static" is approximately 3% faster than "guided" for Xeon and KNL. Comparing "Hybrid+dynamic" and "Flat", both are almost the same processing time in the case of KNL 7210. In the case of Xeon Gold 6150, “Hybrid+dynamic" is 36% faster than “Flat". Additionally, “Hybrid+static" is only 3.9% faster than “Flat". We suppose that fine-grain domain decomposition (72 domains) in single node works equal to static scheduling. One other thing to note here is that processing time of domain decomposition subroutine which is not a target of this paper increases in direct proportion to the number of domain. Projected number of processing time of two KNL 7210 is shown in “KNL 7210 (Dual†)". If there are two KNL 7210, the performance will be twice. The reason why is that the density calculation doesn't have inter-process communication and the number of particles in each process is reduced in half. If we compare “KNL 7210 (Dual†)" with Xeon Gold 6150 (Dual), Xeon Gold 6150 is 1.48 times faster than KNL 7210.

Table 2: Scheduling policy and processing time[ms].

Parallelization	Hybrid	Hybrid	Hybrid	Flat
Scheduling Policy	dynamic	static	guided	N/A
# of processes	2	2	2	72 or 256
# of threads	36	36	36	N/A
KNL 7210 (Single)	104.1	117.2	121.1	105.4
KNL 7210 (Dual†)	52.1	58.6	60.6	52.7
Xeon Gold 6150 (Dual)	35.1	46.0	47.8	47.8

4.2 Tesla P100

Table 3 shows evaluation of optimization 1, 2 and block size. Optimization 1 (“Opt.1" in the table) is about 7 times slower than optimization 2 (“Opt.2" in the table). As we expected, optimization 2 utilizes cache much more efficient than that of optimization 1. For optimization 2, best block size for both P100 is 32 or 64. We expected that block size 32 is better than that of 64 since the most of chosen buckets contains less than 33 particles. When block size is 32 and a chosen bucket contains less than 33 particles, the thread block is executed once. On the other hand, when a chosen bucket contains 33 particles, the thread block is executed twice. In future work, we will investigate the impact on the overall performance of the bucket which contains 33 particles.

Table 3: Optimization, block size and processing time[ms].

Optimization	Opt.1	Opt.2			
block size	64	16	32	64	128
P100 (PCIe, Dual)	59.3	13.4	7.8	7.8	8.3
P100 (NVlink, Dual)	52.8	11.6	6.7	6.7	7.2

4.3 Xeon, Xeon Phi and Tesla P100

Figure 6 shows comparison of processing time of each computation node. “P100 (NVlink, Dual) + block size 32" is the fastest among four computational nodes. It is 5.2 times faster than dual Xeon Gold 6150 and 15.5 times faster than single KNL 7210. Both optimizations doesn't utilize vector unit in Xeon/Xeon Phi. Even if loop-4 is vectorized, indefinite loop and gather/scatter will degrade performance. Different algorithm (e.g. Verlet list in molecular dynamics [4]) might be required to vectorize the loop.

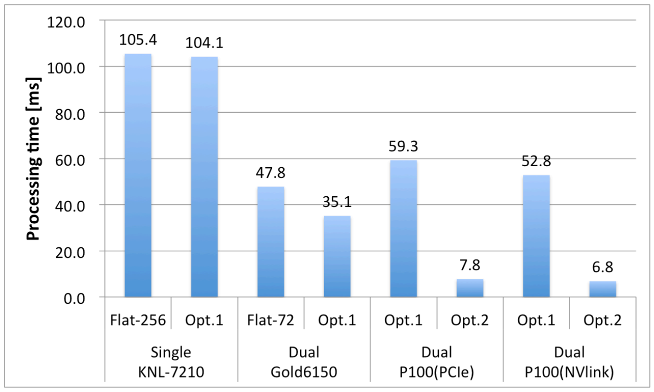

Figure 6: Processing time of each optimization.

5 CONCLUSION

In this paper, we present a porting and optimization of search of neighbour-particle in our in-house MPS program. We give two optimizations; “Bucket per thread" and “Bucket per thread block". Evaluation are conducted on four different computational nodes; dual NVIDIA Tesla P100 (PCIe), dual Tesla P100 (NVlink), dual Intel Xeon Gold 6150 and single Xeon Phi 7210. When the number of particles are 224,910. Processing time of each computational node is 7.8, 6.8, 35.1 and 104.1, respectively. In future work, we will conduct a detailed analysis of proposed optimizations and find out the way to vectorize search for neighbour-particle. We also compare our OpenACC optimizations with another CUDA implementations.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 16K16064.

REFERENCES

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

[2] Kohei Murotani, Seiichi Koshizuka, Tasuku Tamai, Kazuya Shibata, Naoto Mitsume, Shinobu Yoshimura, Satoshi Tanaka, Kyoko Hasegawa, Eiichi Nagai, and Toshimitsu Fujisawa. 2014. 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 (2014), 16–35. https://doi.org/10.15748/jasse.1.16

[3] Kohei Murotani, Issei Masaie, Takuya Matsunaga, Seiichi Koshizuka, Ryuji Shioya, Masao Ogino, and Toshimitsu Fujisawa. 2015. Performance improvements of differential operators code for MPS method on GPU. *Computational Particle Mechanics* 2, 3 (2015), 261–272. https://doi.org/10.1007/s40571-015-0059-2

[4] Simon J. Pennycook, Chris J. Hughes, M. Smelyanskiy, and S.A. Jarvis. 2013. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), 1085–1097. https://doi.org/10.1109/IPDPS.2013.44

[5] Watanabe Seiya, Aoki Takayuki, Tsuzuki Satori, and Shimokawabe Takashi. 2015. Neighbor-particle Searching Method for Particle Simulation Based on Contact Interaction Model for GPU Computing. *IPSJ Transactions on Advanced Computing Systems* 8, 4 (2015), 50–60.

[6] Yasuhide Sota, Akihide Watanabe, and Takashi Kojima. 2013. 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).

[7] H. Sun, Y. Tian, Y. Zhang, J. Wu, S. Wang, Q. Yang, and Q. Zhou. 2015. A Special Sorting Method for Neighbor Search Procedure in Smoothed Particle Hydrodynamics on GPUs. In *Parallel Processing Workshops (ICPPW), 2015 44th International Conference on.* 81–85. https://doi.org/10.1109/ICPPW.2015.46