GPU-acceleration for Moving Particle Semi-implicit Method

Chiemi Hori[1], Hitoshi Gotoh[2*], Hiroyuki Ikari[3] and Abbas Khayyer[4]

[1] Graduate student, Department of Civil and Earth Resources Engineering, Kyoto University
Katsura Campus, Nishikyo-ku, Kyoto, 615-8540, Japan, e-mail: c.hori@kt2.ecs.kyoto-u.ac.jp

[2] Professor, Department of Civil and Earth Resources Engineering, Kyoto University
Katsura Campus, Nishikyo-ku, Kyoto, 615-8540, Japan, e-mail: gotoh@particle.kuciv.kyoto-u.ac.jp

[3] Assistant Professor, Department of Civil and Earth Resources Engineering, Kyoto University
Katsura Campus, Nishikyo-ku, Kyoto, 615-8540, Japan, e-mail: ikari@particle.kuciv.kyoto-u.ac.jp

[4] Lecturer, Department of Civil and Earth Resources Engineering, Kyoto University
Katsura Campus, Nishikyo-ku, Kyoto, 615-8540, Japan, e-mail: khayyer@particle.kuciv.kyoto-u.ac.jp

Abstract:

The MPS (Moving Particle Semi-implicit) method has been proven useful in computation free-surface hydrodynamic flows. Despite its applicability, one of its drawbacks in practical application is the high computational load. On the other hand, Graphics Processing Unit (GPU), which was originally developed for acceleration of computer graphics, now provides unprecedented capability for scientific computations.

The main objective of this study is to develop a GPU-accelerated MPS code using CUDA (Compute Unified Device Architecture) language. Several techniques have been shown to optimize calculations in CUDA. In order to promote the acceleration by GPU, particular attentions are given to both the search of neighboring particles and the iterative solution of simultaneous linear equations in the Poisson Pressure Equation.

In this paper, 2-dimensional calculations of elliptical drop evolution and dam break flow have been carried out by the GPU-accelerated MPS method, and the accuracy and performance of GPU-based code are investigated by comparing the results with those by CPU. It is shown that results of GPU-based calculations can be obtained much faster with the same reliability as the CPU-based ones.

Key words: Particle method, MPS method, Semi-Implicit Algorithm, GPU, CUDA

1. Introduction:

A free-surface flow analysis is indispensable to solve a lot of hydrodynamic issues. Especially, in flood control, it is a key to conduct accurate analysis of a violent flow. In a design of a coastal structure, a numerical wave flume has attracted considerable attention as a tool to predict

* Corresponding Author, TEL +81-75-383-3309 , FAX +81-75-383-3311

hydrodynamic forces corresponding to an extreme wave precisely. In an accurate prediction of wave overtopping discharge in addition to wave force, it is important to treat a fragmentation of fluid like a splash flexibly. The particle method is one of the methods to meet such a requirement because it has a high reproducibility of a complicated behavior of water surface change with a fragmentation and coalescence of water (e.g. Gotoh, 2009).

The most challenging problem in practical use of the particle method is related to a high computational load. Accurate calculation using a large number of particles over 1 million is necessary in a practical calculation, and such a calculation is not possible without incorporating high performance CPU or parallel computing technique, however, a hardware including its maintenance is too expensive to be widely provided to engineers as a desktop design tool.

GPU (Graphics Processing Unit) is a multi-processor executing a large number of three-dimensional geometry processing in a high speed. Because this kind of graphics processing is independent of others, it is suitable for parallelization. Hence, GPU has been developed as a hardware with parallelizing many processors which have a simpler structure than CPU.

In recent years, a floating-point arithmetic by GPU became possible, and the General-Purpose computing on GPU (GPGPU) was tried. In 2006, CUDA (Compute Unified Device Architecture) was opened to public, and development of a general-purpose computing code by C language has become possible without special knowledge in graphics coding. In 2007, the Tesla, which is a GPU without any output terminals for graphics, was released, and a full-scale entry to high performance computing was started.

GPU computing has been also applied in CFD (Computational Fluid Dynamics) simulations. In early times when the GPU had started to be used generally, its acceleration performance in calculations and its visual realism of fluids motion were mainly focused. However, through the past couple of years, its reliability for an accurate prediction has been also studied (e.g. Rossinell and Koumoutsakos, 2008).

As for a particle method, in case of the SPH (Smoothed Particle Hydrodynamics) method (Gingold and Monaghan, 1982), there are a number of previous studies (e.g. McCabe et al., 2009), because it is easy to apply GPU calculation to the explicit algorithm of the SPH method. However, due to a fully explicit nature, the use of a numerical stabilizer (e.g. an artificial viscosity term with empirical constants) or application of a Reimann solver tends to be indispensable in a Weakly Compressible SPS (WCSPH) calculation.

On the other hand, semi-implicit particle-based methods such as the Moving Particle Semi-implicit (MPS; Koshizuka and Oka, 1996) or the Incompressible SPH (ISPH; Shao and Lo, 2003) methods appear to be more stable than the fully explicit ones allowing relatively stable/accurate calculations (e.g. Gotoh and Sakai, 2006; Khayyer et al., 2008) without an artificial numerical stabilizer or specific solvers. However, it is difficult to implement a GPU-based MPS or ISPH calculation due to their semi-implicit algorithm. To the best of authors' knowledge, there has not been any study on GPU application to the MPS method. Therefore, in this paper, key items in the development of a GPU-based MPS code are clarified, and their efficiency on acceleration in computation is investigated.

2. Calculation Technique:

2.1. MPS method overflow

Fig. 1 shows a computational flow chart of the MPS method which is used in this paper (Ikari and Gotoh, 2008). The time integration is mainly composed of two steps. The first step corresponds to an explicit calculation considering the gravity and viscosity terms. The second step is an implicit calculation accounting for the pressure term. Pressure values are obtained by solving a Poisson equation, which is discretized into a simultaneous linear equation. As the solver of this equation, the ICCG (Incomplete Cholesky Conjugate Gradient) method is implemented in the original MPS method. In addition, a measure to prevent particles from encountering collision is conducted. Just before each step, searches of neighboring particles are executed.

In this paper, the SCG (Scaled CG) method is applied for pressure calculation in both GPU-based code and CPU-based code, since implementation of the SCG method in GPU-based calculations is quite easier than that of the ICCG.

2.2 Development environment

In this section, the structure of a GPU and an overview of the CUDA programming model are presented. Fig. 2 shows the important details of Tesla C1060, or the GPU used in this study. Tesla C1060 is an assembly of 30 multiprocessors (MP), with 8 streaming processors (SP) in each. Each MP possesses its own *shared memory* (16KB) commonly used by all the 8 SPs in it. Communications between the MPs are performed through the device memory (*global memory*; 4GB), which is accessible to all the SPs of the MPs (Harish and Narayanan, 2007).

The CUDA programming model is an assembly of many *threads* running in parallel. An assembly of threads, which is called a *block*, runs on a MP. An assembly of all blocks in a single execution is called a *grid* (Harish and Narayanan, 2007). The maximal number of threads that each MP can handle at a time is 1,024. But in practice, the number of threads is limited by the amount of shared memory and registers, so it is application-dependent (Liu et al, 2008). Resisters, which are possessed by each MP as a set, are on-chip memories used to store temporal values.

The CUDA programming model assumes that a *host* (CPU) code, written in C language, calls *kernels* which are executed on the *device* (GPU). The host code also includes instructions for setting parallelism and communicating data between host and device (NVIDIA, 2008).

2.3. CUDA implementation of MPS method

2.3.1. Key points for the overall implementation

(a) In principle, one thread processes one particle. For example, when a thousand particles are calculated, a thousand threads are launched. Fig. 3 briefly shows the difference in coding between the sequential process by single thread and the parallel process by many threads.

(b) The performance improves if one uses the shared memory which can be accessed much faster than the device memory. But as mentioned above, the shared memory's scope is limited to blocks. In this paper, as an essential tuning at a minimum cost, the best usage of the shared memory is made in the following manner.

If a variable is accessed more than twice from threads of the same block in the kernel, the variable is instructed to be loaded from the device memory into the shared memory, and then the shared memory is accessed during the calculation.

In Fig. 4, a shared memory (*Variational_Velocity_s[I]*) is used to store the temporal summation of the acceleration. Each acceleration is obtained from the interaction between the target particle and its neighboring particle. Then at the end of the kernel, the data is stored to the device memory (global memory).

(c) The global memory accesses are performed on segments. In order to optimize the global memory transactions, memory accesses should be coalesced whenever possible. To achieve this, arrays of physical quantities, pressures, velocities and locations of particles are stored to be aligned in the global memory. Moreover, a list of neighboring particles and the coefficient matrix of the Poisson Pressure Equation are aligned as shown in Fig. 5 so that coalesced accesses are executed as many times as possible.

In the case of single thread of CPU, a coding in Fig. 6(a) implements the same arrangement of the data (*NeighboringIdList[I][L]*) as that of Fig.5(a). If this code (Fig. 6(a)) was naively translated into a code for GPU, the schematic diagram would be Fig. 6(b). Nevertheless, the array (*NeighboringIdList[I][L]*) in the code of Fig. 6(b) provides inefficiency to threads on GPU. In order to align the data in the same way of Fig. 5(b), the array should be transposed as it is shown in Fig. 6(c).

(d) The double precision floating point computation unit is only one in each MP of the GPU used in this paper. Consequently, the performance is around 10 times lower than for 32 bit arithmetic treating integer and single precision floating point. But in this paper, our CUDA code supports the double precision for floating point computations which are expected to have the double precision in the original code for the sequential process by single CPU.

(e) The threads of a block are executed in groups of 32 threads, called *warps*. A warp executes a single instruction at a time across all its threads. When conditional branching occurs in one warp, a serialization happens and the performance gets lower. This deterioration is called "warp-divergence". However, the branch instruction is not considered here and the code is not changed especially for that.

(f) Although synchronization of all threads within a block can be achieved, synchronization of all threads in a grid is not guaranteed until all threads have executed the kernel. The barrier between blocks is regarded as a priority, so as to ensure the variable update on the global memory. Therefore our MPS-CUDA program consists of the collection of several lightweight kernels.

2.3.2. The search for neighboring particles

Firstly, the framework of the search for neighboring particles is explained briefly. The framework is basically the same as that in our conventional code for a single thread (Gotoh et al, 2005). The framework consists of these operations (Fig. 7).

I. Listing particles with the cell-scale resolutions: Each particle is stored in a specific cell according to the particle's position. The length of the cell is as long as the radius of influence.

II. Listing neighboring particles of each particle: Neighboring particles of a target particle are

searched by visiting its associated and surrounding cells possessing the list of particles.

Operations I and II have been proven to improve the efficiency in the conventional sequential calculation (Gotoh et al, 2005). These operations are also applied in our CUDA-MPS code to ensure the fairness of the comparison of the calculation time. In this section, the way to implement operation I on GPU is stated, while implementation of operation II is described in 2.3.4.

In the operation I, particles on each cell should be listed sequentially. It is not efficient for thousands of parallel threads to conduct such a sequential process. Therefore, one of the relatively more efficient ways is that one block processes one cell, and that threads of the block search particles in the cell and synchronously make the list of its candidates. In the sequential computation, the screening cell is arranged to cover the rectangular calculated domain. This approach is relatively efficient in the conventional sequential computation (Gotoh et al, 2005), although a lot of amount of memory must be consumed. However, in the GPU computing, when one block is assigned to one cell, any idle blocks may be set. The more the number of the cells in which no particles are located, the more the number of idle blocks. When the fluid particles distribute nonuniformly, the number of idle blocks increases.

In this paper, a function which allocates as few cells as possible are added, in order to save computation cost. This preparatory function is named P2 (preparation 2) in Fig. 7. The function is implemented by applying a sliced data structure proposed by Harada et al (2008). The method is briefly explained as follows. A space of two-dimensions is assumed. The computational region is sliced parallel to the y-axis and is divided into several stripes. The breath of a stripe equals to the length of the cell. On each slice, a maximum and minimum of y-coordinate among particles' positions are searched. Then on each slice, cells are allocated only between the minimum and maximum. A parallel reduction (Harris, 2008) is applied to the search of a maximum or minimum value.

After allocating cells, operation I is conducted. To reduce particles searched by threads of one block processing the cell, two preparations are added in our GPU-based calculation. The first preparation (P1 in Fig. 7) is changing particle numbers so that the particles' x-coordinate positions are arranged in ascending order. A block transition sort, which is provided by Harada et al. (2008), is applied for this sorting. The second preparation (P3 in Fig. 7) is memorizing one arbitrary number of particles existing in each slice. Thanks to those preparations, in the operation I, threads only have to check limited number of particles in a few slices.

Fig. 7 shows the difference between the computational flow chart of the search for the neighboring particles by GPU-based calculation and that by CPU's sequential calculation.

2.3.3. The Poisson Pressure Equation

A GPU is capable of adding and subtracting vectors work in the CG method, since every thread can perfectly process one particle (one element). Inner product calculations are split into two kernels. In one kernel, each thread prepares one product by multiplying elements, and then a parallel reduction process is conducted in order to sum all those products in the other kernel.

On the other hand, the iterative solver includes a matrix-vector multiplication. A coefficient matrix of the PPE is a sparse matrix. A sparse matrix-vector (SpMV) multiplication almost

monopolizes the computation time in one iterative operation on a CPU, as well as on a GPU. To make matters worse, the calculation of a parallel process by many threads on GPU would not get much faster than that by a sequential process of CPU, if one translated the code naively. Memory latency greatly affects performance of the SpMV multiplication (e.g. Bell and Garland, 2008). The coefficient matrix is stored into global memory, using a format similar to the ELLPACK-R one (Vazquez et al, 2009). Therefore, the coalesced global memory access can be achieved on loading elements of the coefficient matrix.

However, it is impossible to do coalesced access to following two arrays of N dimension (N is the number of total particles). One of those arrays is double precision float, storing the vector. The other is integer, storing flags used to judge whether the particle is a non-free-surface water particle. Since each thread accesses those arrays by using indices which are the neighboring particles' numbers of each particle, those accesses are executed randomly. So those arrays are allocated in read-only cached texture memory described below, in order to reduce the effect of the irregular and random access (Takai and Nagai, 2009).

2.3.4. Utilization of texture memory

A GPU has a specific unit that accelerates address references to textures of 3D-graphics. CUDA also supports this unit, as *texture memory* (NVIDIA, 2008). The texture memory is a read-only memory. Since it is cached on-chip, it can be accessed relatively faster. On the contrary, the global memory space is not cached. Therefore, reading global memory through texture fetching bring about some benefits that can hide the latency of addressing calculations better, even in the situation where uncoalesced loads from global memory happen. The effective utilization of texture memory results in the improvement of the performance for applications with frequent random access to the data, such as particle method. Moreover, if there is locality in the texture fetches, the texture memory exhibits higher bandwidth.

Although the texture memory is useful for acceleration, the coding to use the texture memory is more complicated than that to use the global memory. Hence in this paper, the texture memory is used only in the two kernels. One is the kernel for a local search of neighboring particles (operation Ⅱ in 2.3.2), and the other is the kernel for the SpMV multiplication in the iterative solver for PPE. The reason why the latter is treated in this way is already mentioned in 2.3.3.

The former kernel provides lots of warp-divergences which are mentioned in 2.3.1(e). In this kernel, each thread for one particle visits its surrounding cells. Each cell contains different number of candidate particles. Every thread in one warp must wait until every thread of the warp finishes accessing their cells. A cell sometimes possesses much more particles than near cells do. Therefore, a thread accessing the cell which has more particles keeps any other threads in the same warp waiting, even if the other threads access only the cells possessing a few particles. This would result in a non-optimized function of threads.

Consequently, in this kernel, it takes by far the longest computation time in the whole time of the search for neighboring particles. Therefore, the speed of the calculation of the search for neighboring particles can be increased, if that of this kernel is improved by using the texture memory.

3. Verification:

3.1. Purpose

In this chapter, the results of simulations by using both the CUDA-MPS calculation code and the conventional sequential calculation code are discussed. From now on, for the sake of convenience, the conventional sequential calculations are called 'only CPU', and the CUDA calculations, which have been implemented through this paper, are called 'GPU+CPU'. Table 1 shows the computational environment corresponding to the calculations of this paper.

The algorithm and implementations are optimized differently in each code so as to shorten each calculation time. However, the methodology of the numerical solution to get an answer for a problem, that is the flow chart shown in Fig. 1, is the same. Therefore, it is anticipated that results by both calculations will be exactly the same. However, the results by GPU+CPU and those by only CPU are not exactly the same. Such differences are most likely caused by the following factors.

1) Tesla C1060, or any graphic card provided by NVIDIA as of January 2010, does not have an ECC (Error-Correcting Codes) in graphics memory systems. Therefore, wrong values may be written into the memory accidentally (Ogawa and Aoki, 2009; Maruyama, 2009).

2) The GPU has multiply-add calculation units. FMAD (Floating-point Multiple ADd) operations are bundled and optimized by the CUDA Compiler. Since multiply-add is used in many calculations, GPU can achieve the high arithmetic capacity by possessing the special calculation units additionally. But as truncated intermediate results of multiplication are used, there is a possibility that the multiply-add results by GPU will be different from that by general X86 CPU (Fixstars Corporation, 2009; NVIDIA, 2008). However, this optimization can be avoided manually in order to get the same results by GPU as those by CPU. In this paper, this avoidance is carried out as far as possible.

3) Other factors are as follows: the reliability of the compiler, the way of summation (sequential addition by CPU or reduction by GPU), human errors, etc.

Although differences in results by only CPU and GPU+CPU after only one time integration are small enough to ignore, in calculations which have many time integrations, significant differences can become obvious. Furthermore, in the original MPS method, numerical errors that are obtained by solving the PPE are prone to be accumulated (Khayyer and Gotoh, 2009; 2011). Therefore, these errors gradually generate differences in movements of particles, so that simulations by only CPU and GPU+CPU do not become exactly the same concerning water surface profiles or fluid velocity.

From the next section, equal importance is placed on the comparison of calculation speed up and the accuracy of results, similar to Rossinelli et al. (2010) that have investigated those issues carefully. Through this discussion, it is aimed to demonstrate that GPU+CPU calculations are as reliable as only CPU ones and meet practical calculations.

3.2. 2-Dimensional calculation results

2-D simulations are conducted for verification of CUDA-MPS code. The radius of influence is chosen to be 4.0d (d : diameter of particle) for the Laplacian model and 2.1d for the gradient model (Koshizuka and Oka, 1996).

3.2.1. Comparison of calculation time

The calculation time per one time step is compared. The calculated domain is shown in Fig. 8. Simulations under five cases are conducted. Total numbers of particles are 7,080, 11,676, 24,360, 89,520 and 109,211 in each case by changing diameter of particle to 4.0, 3.0, 2.0, 1.0 and 0.9 mm, respectively.

All calculations in this paper apply a variable time increment, which is set to satisfy the Courant's stability condition. So in the case of different diameters of particles, even if the physical times are the same, their total numbers of time steps will become different. That is the reason why the calculation time per one time step in this section is calculated by dividing a net running time by its total number of time steps.

Fig. 9 shows the time fraction in the five cases simulated by only CPU or GPU+CPU. Each segment shows, beginning, search for neighboring particles (neighboring), generation of matrix (matrix), preconditioning by diagonal scaling (scaling), iterative computation (pcg solver), external calculation (gravity and viscosity terms; external), writing output into text files (output) and others. It must be noted that the part of output includes time dedicated to the transfer of memory between device and host in the case of GPU+CPU. From this figure, it can be stated that the iterative calculation takes up most of the calculation time both on only CPU and on GPU+CPU.

Fig. 10 shows the improvement factor of using GPU+CPU in comparison to only CPU results. Regarding relatively many particles, it can be observed that GPU+CPU is more than 10 times faster than only CPU in generation of matrix, preconditioning and external calculation. In the kernels handling these tasks, one thread can process one particle more easily than in the other kernels.

On the other hand, the calculation time of the search of neighboring particles or the iterative computation is not so efficiently shorter than expected. The acceleration of the whole calculation time seems to depend on that of the iterative computation time, and then the performance is improved by a factor of 7 plus at the most.

3.2.2. Accuracy

(1) Evolution of an elliptical drop

The simulation of an evolution of elliptical water drop (Monaghan, 1994; Khayyer and Gotoh, 2008; 2009) is carried out by using each code. The initial fluid configuration is a circle of radius 1.0 m subjected to no external forces but an initial velocity field as (-100x, 100y) m/s. The domain is represented by a total number of 7,845 particles being 2.0cm in diameter.

The snapshots showing the particle configuration with the horizontal velocity distribution at t = 0.008s are depicted in Fig. 11. In this calculation, both results gained by the two computational architectures seem almost the same from a qualitative aspect.

Values of semi-minor axis, semi-major axis and their product are obtained by calculating with

only CPU, GPU+CPU and an analytical solution (Khayyer and Gotoh, 2009). The time histories of those values are shown in Fig. 12. Differences between the simulated results and the analytical ones are intensified as the drop continues to evolve. But both simulations seem to result in almost the same results.

For further verification of two simulations, numerical errors of GPU+CPU results compared with values obtained by only CPU are plotted in Fig. 13. The vertical axis's value equals to $|(a_{\text{GPU+CPU}}-a_{\text{only CPU}})/a_{\text{only CPU}}|\times 100[\%]$, for example in semi-minor axis ($a_{\text{only CPU}}$: values obtained by only CPU, $a_{\text{GPU+CPU}}$: values obtained by GPU+CPU). This figure indicates that the maximal value of GPU-based calculation's errors is less than $5.0\times 10^{-5}\%$, in this specific simulation.

To simulate 0.010 seconds in physical time, the net running times required for the only CPU and GPU+CPU calculations are 104.5s and 25.4s (103.7s and 24.4s if outputs and memory transfers are excluded), respectively. For this relatively small number of particles, the performance is higher than that shown in Fig. 10. Since this calculation treats only water particles, branch instruction depending on particle types occurs relatively a few times. Accordingly, warp-divergences under this condition are fewer than those in a dam break simulation. Therefore, the speed up is slightly improved.

(2) A dam break with impact

The results of a dam break simulation illustrated in Fig. 8 are compared. The particle diameter is 4.0mm, and the number of total particles is 7,080.

It can be stated that both calculations present the same results prior to the water impact on the wall as shown in Fig. 14. However, after the impact, distributions of scattering particles appear to become slightly different. Fig. 15 shows the snapshots of water particles expressing a violent plunging jet impact at t = 0.750s. In this figure, GPU+CPU depicts a few particles in the air chamber beneath the plunging jet and reveals a visible difference from only CPU.

Nevertheless, these differences are expected to be improved by a GPU released in the near future. Above all, global profiles of water surface and velocity distributions seem almost the same, even in such a case where a violent flow and long time integration are included.

4. Conclusive Remarks:

In this paper, in order to accelerate the MPS-based calculations which requires comparatively higher computational load, basic examinations on GPU application are carried out. Indispensable items for an acceleration of arithmetic code by CUDA are pointed out, and the items which can bottleneck the acceleration in the standard MPS code are considered. In the neighboring particle search, an efficient calculating process to generate neighbors' list on GPU is shown. In the SCG method for pressure calculation, which is easy to be translated but difficult to be accelerated, the calculating procedure is examined in detail. It is clarified that the memory latency in a sparse matrix-vector multiplication is bottleneck of the acceleration, and its appropriate remedy is examined.

In some previous studies on GPU application to the SPH method, the achievements of several ten times of acceleration ratio were reported. The acceleration ratio in this paper is about 3 to 7

times. The MPS calculation by using only CPU without GPU is generally faster than the (fully explicit) SPH calculation due to its independency on speed of sound and hence incorporation of relatively larger calculation time steps. Therefore, it can be said that the GPU-accelerated MPS calculation shows comparable or rather better performance.

In this paper, the standard MPS code for two-dimensions is accelerated by GPU under a few relatively simple conditions. However, in order to predict violent fluid motion more precisely, GPU accelerations should be applied to 3D-MPS code, as the parallelization on cluster PC (e.g. Ikari and Gotoh, 2008). Moreover, development of the GPU code of the refined particle methods (Shao and Gotoh, 2005; Gotoh and Sakai, 2006; Khayyer and Gotoh, 2009; 2010; 2011) is also included in our future works.

References:

Bell, N. and Garland, M. 2008. Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, p32.

Fixstars Corporation. 2009. NVIDIA CUDA Information Site. http://gpu.fixstars.com/. (in Japanese)

Gingold, R. A. and Monaghan, J. J. 1982. Kernel Estimates as a Basis for General Particle Methods in Hydrodynamics, *J. Comp. Phys.*, 46, 429-453.

Gotoh, H. 2009. Lagrangian Particle Method as Advanced Technology for Numerical Wave Flume. *International Journal of Offshore and Polar Engineering*, 19, 3, 161-167.

Gotoh, H., Ikari, H., Memita, T. and Sakai, T. 2005. Lagrangian particle method for simulation of wave overtopping on a vertical seawall. *Coast. Eng. J.* 47(2 & 3), 157-181.

Gotoh, H. and Sakai, T. 2006. Key issues in the particle method for computation of wave breaking. *Coastal Engineering*, 53, 171-179.

Harada, T., Masaie, I., Koshizuka, S., Kawaguchi, Y. 2008. Accelerating Particle-based Simulations Utilizing Spatial Locality on the GPU. Transactions of JSCES, Paper No.20080016. (in Japanese)

Harish, P. and Nrayanan, P. J. 2007. Accelerating Large Graph Algorithms on the GPU using CUDA, HiPC 2007, LNCS 4873, 197-208.

Harris, M. 2008. Optimizing Parallel Reduction in CUDA. NVIDIA CUDA SDK 2.0.

Hu, C.H., Kashiwagi, M. 2004. A CIP Method for Numerical Simulations of Violent Free Surface Flows, *Journal of Marine Science and Technology*, 9(4), 143-157.

Ikari, H. and Gotoh, H. 2008. Parallelization of MPS method for 3D wave analysis, *Advances in Hydro-science and Engineering, 8th International Conference on Hydro-science and Engineering (ICHE)*, Nagoya, Japan.

Khayyer, A. and Gotoh, H. 2008. Development of CMPS method for Accurate Water-Surface Tracking in Breaking Waves. *Coast. Eng. J.*, 20, 2, 179-207.

Khayyer, A. and Gotoh, H. 2009. Modified Moving Particle Semi-implicit methods for the prediction of 2D wave impact pressure. *Coastal Engineering*, 56, 4, 419-440.

Khayyer, A. and Gotoh, H. 2010. A higher order Laplacian model for enhancement and stabilization of pressure calculation by the MPS method. *Applied Ocean Research*, 32, 1,

124-131.

Khayyer, A. and Gotoh, H. 2011. Enhancement of stability and accuracy of the moving particle semi-implicit method. *J. Comp. Phys.*, 230, 8, 3093-3118.

Khayyer, A., Gotoh, H. and Shao, S. D. 2008. Corrected Incompressible SPH method for accurate water-surface tracking in breaking waves. *Coastal Engineering*, 55, 3, 236-250.

Koshizuka, S. and Oka, Y. 1996. Moving-particle semi-implicit method for fragmentation of incompressible fluid, *Nucl. Sci. and Eng.*, 123, 421-434.

Liu, W., Schmidt, B., Voss, G. and Muller-Wittig, W. 2008. Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA, *Computer Physics Communications*, 179, 634-641.

Maruyama, N., Nukada, A. and Matsuoka, S. 2009. Software-Based ECC for GPUs. *Presented at Symposium on Application Accelerators in High Performance Computing, Urbana, Illinois, US, July 27-31, 2009*.

McCabe, C., Causon, D. M. and Mingham, C. G. 2009. Graphics Processing Unit Accelerated Calculations of Free Surface Flows using Smoothed Particle Hydrodynamics, *4th international SPHERIC workshop, Nantes, France, May 27-29, 2009*, 384-391.

Monaghan, J. J. 1994. Simulating free surface flows with SPH. *J. Comput. Phys.* 110, 399-406.

NVIDIA. 2008. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0.

NVIDIA. Compute unified device architecture, http://www.nvidia.com/object/cuda_home.html.

Ogawa, S. and Aoki, T. 2009. GPU Computing for 2-dimensional incompressible-flow Simulation based on Multigrid method. Transactions of JSCES, Paper No.20090021. (in Japanese)

Rossinelli, D., Bergdorf, M., Cottet, G.-H. and Koumoutsakos, P. 2010. GPU accelerated simulations of bluff body flows using vortex particle methods. *J. Comp. Phys.*, 229, 3316-3333.

Rossinelli, D. and Koumoutsakos, P. 2008. Vortex methods for incompressible flow simulations on the GPU. *Visual Comput*, 24, 699-708.

Shao, S. D. and Gotoh, H. 2005. Turbulence particle models for tracking free surfaces. *Journal of Hydraulic Research*, 43, 3, 276-289.

Shao, S. D. and Lo, E. Y. M. 2003. Incompressible SPH method for simulating Newtonian and non-Newtonian flows with a free surface. *Adv. Water Resour.*, 26, 7, 787–800.

Takai, Y. and Nagai, G. 2009. Fast calculation of conjugate gradient method by GPU, *Proceedings of the Conference on Computational Engineering and Science, Tokyo, Japan, May 12-18, 2009*, 14, 283-284. (in Japanese)

Vazques, F., Garzon, E. M., Martinez, J. A. and Fernandez, J. J. 2009. The sparse matrix vector product on GPUs. http://www.ace.ual.es/TR/SpMV.GPU.pdf.

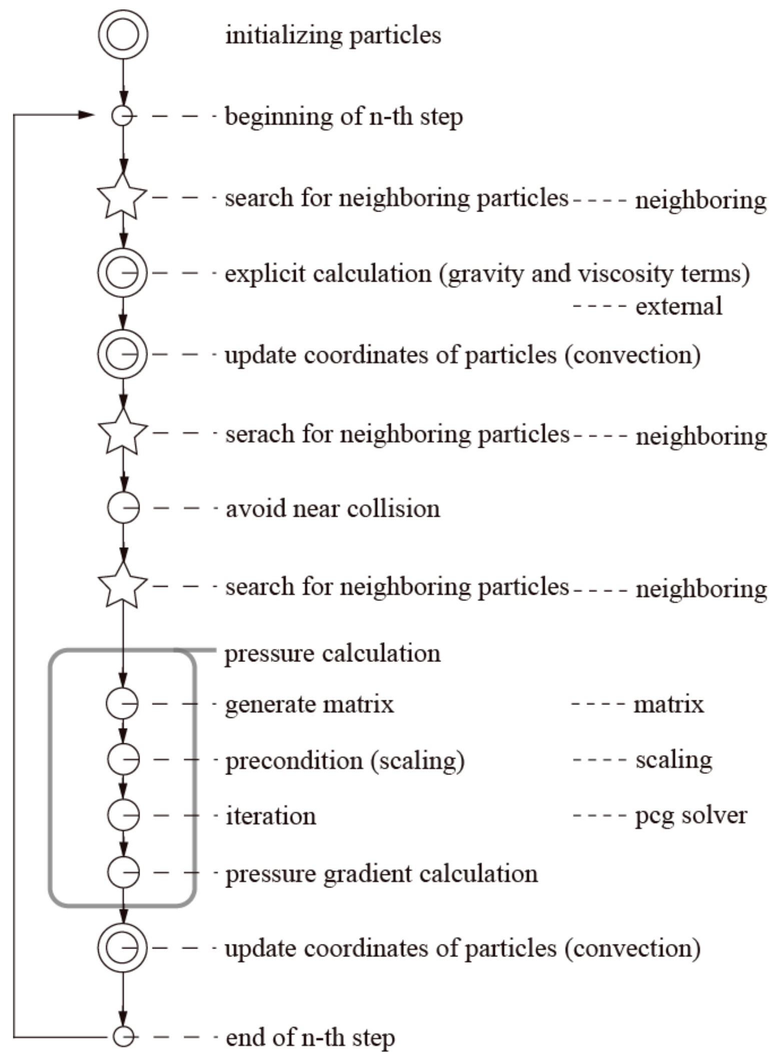


Fig. 1. Computational flow chart of the MPS method

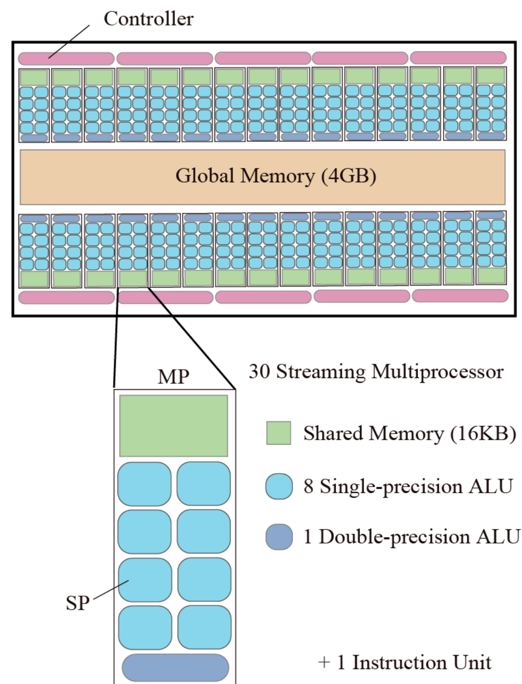


Fig. 2. Tesla C1060 structure

```
void Gravity(double* Variational_Velocity)
{
  for(int I=0;I<N;I++) // I:particle ID number
  {
    Variational_Velocity[I]
    += gravitational_acceleration * dt
  }
}
```

(a) Loop iteration by single thread of CPU (*N* is the number of particles.)

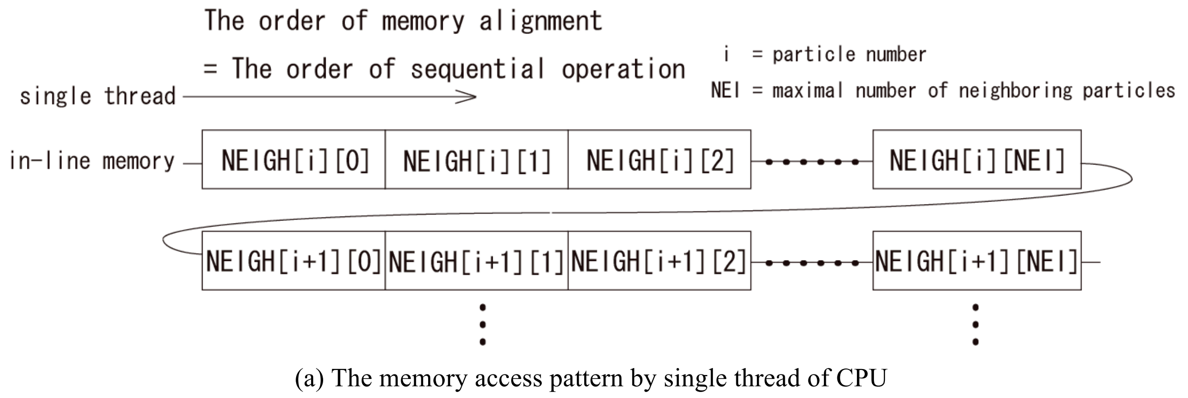
```
__global__
void Gravity(double* Variational_Velocity_g)
{
  int I=threadIdx.x; // I:particle ID number
                     // threadIdx.x:thread ID
    Variational_Velocity_g[I]
    += gravitational_acceleration * dt
}
```

(b) Light and simple task by each thread on GPU
(The case that the number of threads is less than or equal to the number of particles)

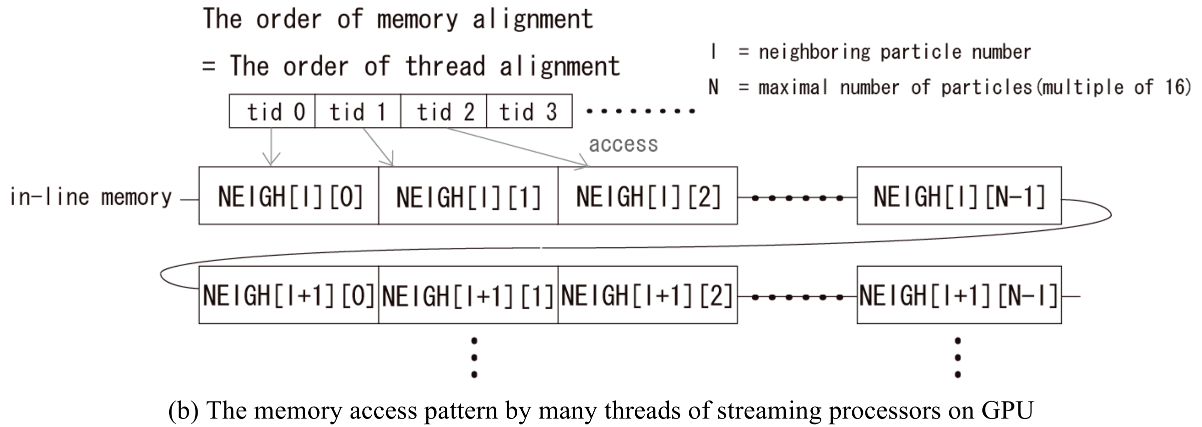
Fig. 3. The schematic diagram of coding of processing particles

```
__global__
void Viscosity(double* Variational_Velocity_g)
{
  int I=threadIdx.x;
  extern __shared__ double
       Variational_Velocity_s[];
        // set a data in shared memory

  for(int L=1; NeighboringMax[I]; L++)
  {
    int J=NeighboringIdList[L][I];
       // J:neigboring particle ID number

    /* --- Here are calculations of
     interaction force by accessing
     the position/velocity data in global memory
     of particle I and those of particle J --- */

    Variational_Velocity_s[I]
    += viscosity_interaction_IandJ * dt
        // the data in shared memory is
        // frequently accessed
  }
  Variational_Velocity_g[I] // update of the data
                            // in global memory
  += Variational_Velocity_s[I]
}
```

Fig. 4. The schematic diagram of coding of shared memory usage on GPU



(a) The memory access pattern by single thread of CPU



(b) The memory access pattern by many threads of streaming processors on GPU

Fig. 5. The alignment of neighboring particle list

```
void Viscosity(double* Variational_Velocity)
{
  for(int I=0;I<N;I++)
  {
    for(int L=1; NeighboringMax[I]; L++)
    {
      int J=NeighboringIdList[I][L];

      /* --- Here are calculations of
       interaction force between
       particle I and those of particle J --- */

       Variational_Velocity[I]
       += viscosity_interaction_IandJ * dt
  }
```

(a) Two-dimensional array that fits the sequential process by single threads of CPU

Fig. 6. The schematic diagram of coding of two-dimensional array

```
__global__
void Viscosity(double* Variational_Velocity_g)
{
  /* -- omission -- */

  for(int L=1; NeighboringMax[I]; L++)
  {
    int J=NeighboringIdList[I][L];
       // J:neigboring particle ID number

    /* -- omission -- */
  }
  /* -- omission -- */
`
```

(b) Two-dimensional array that does not fit the parallel process by many threads on GPU

```
__global__
void Viscosity(double* Variational_Velocity_g)
{
  /* -- omission -- */

  for(int L=1; NeighboringMax[I]; L++)
  {
    int J=NeighboringIdList[L][I];
       // J:neigboring particle ID number

    /* -- omission -- */
  }
  /* -- omission -- */
}
```

(c) Two-dimensional array that fits the parallel process by many threads on GPU

Fig. 6. *(continued)*

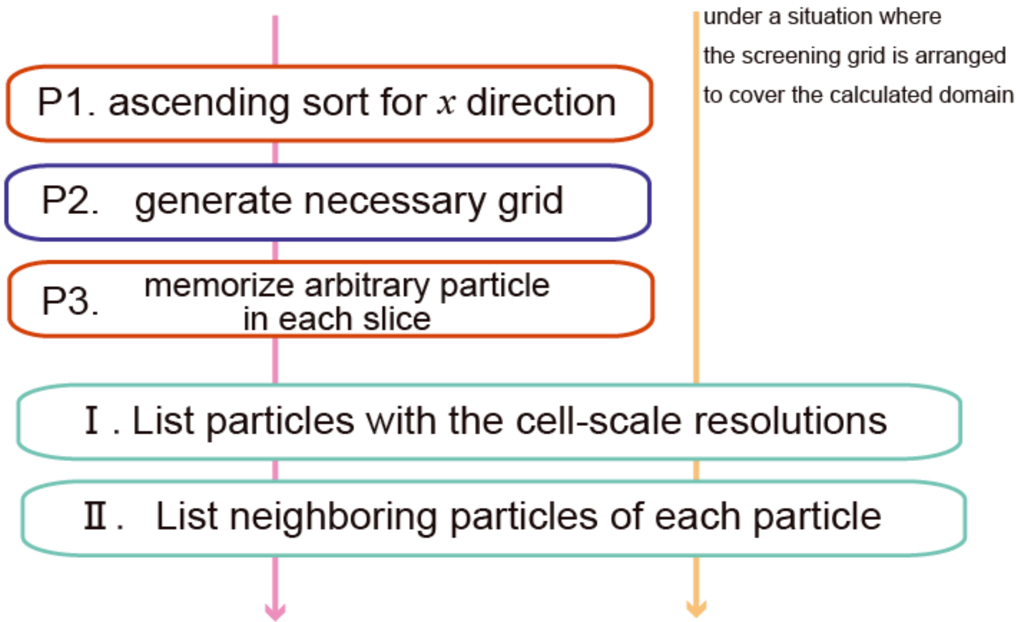


Fig. 7. Computational flow chart in the search for neighboring particles

Table 1. The computation environment in this paper

	only CPU	GPU+CPU
CPU	Intel® Core™ i7 920 2.67GHz (1 core is used in this paper)	
Main memory	2.49GB	
Graphics Card		NVIDIA® Tesla™ C 1060 4GB
Driver Version		185.85
OS	Microsoft Windows XP Professional Version 2002 SP3	
Programming Language	Fortran	CUDA, C
Compiler	Intel® Fortran Compiler 9.1	CUDA 2.1, Microsoft Visual Studio 2005

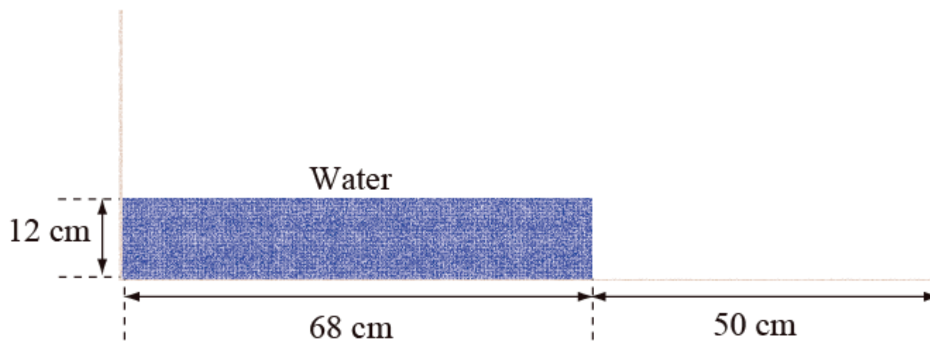


Fig. 8. Calculated domain of dam break (Hu and Kashiwagi, 2004)

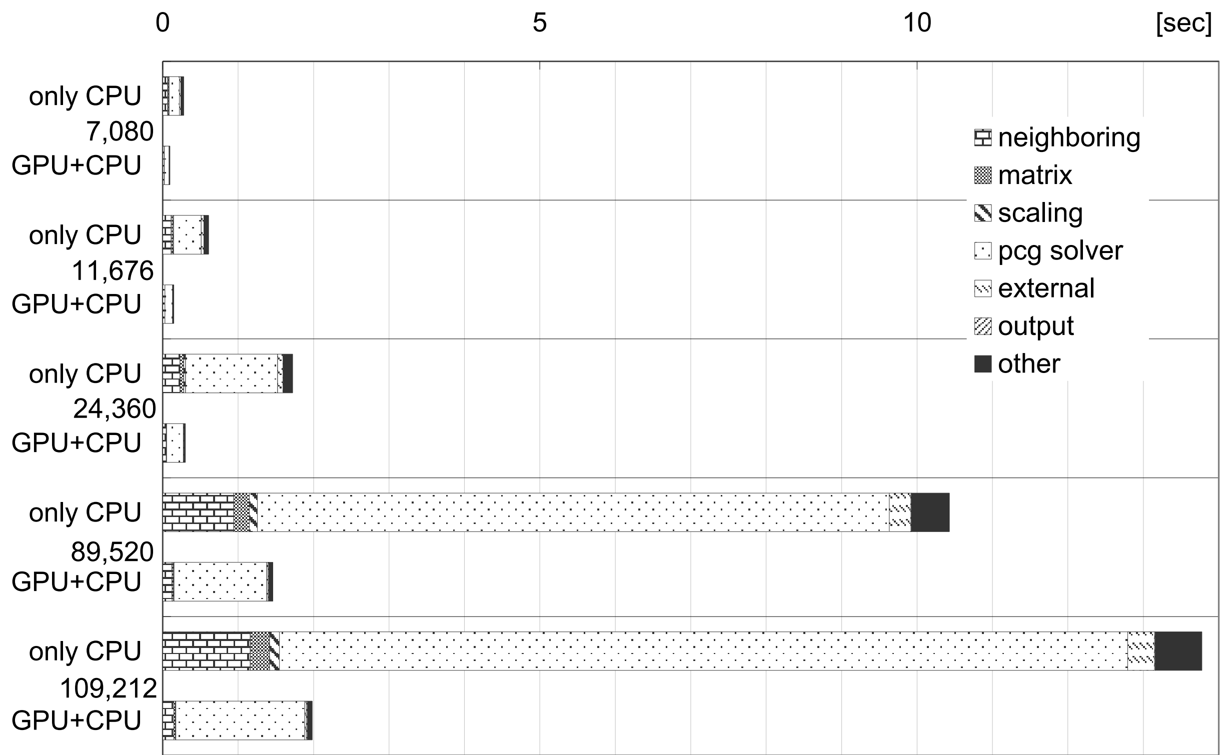


Fig. 9. Calculation time fraction

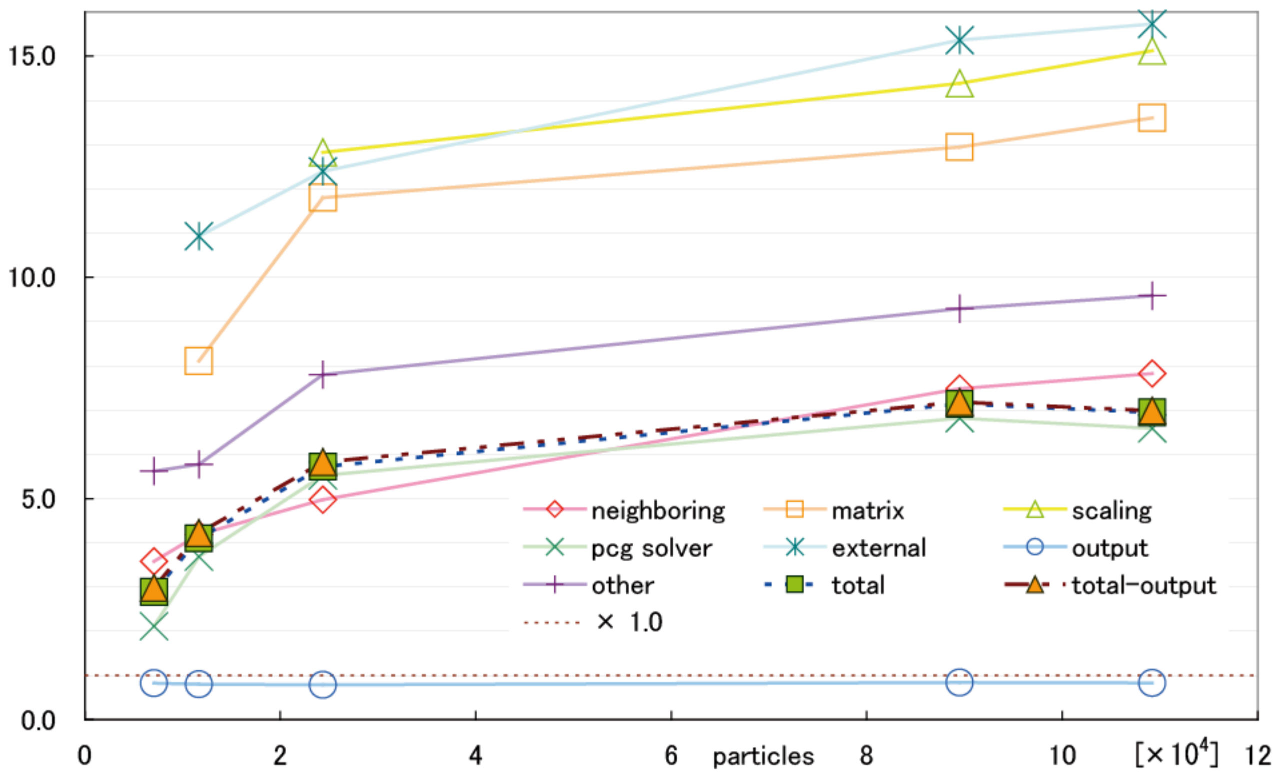


Fig. 10. Speedups by GPU+CPU

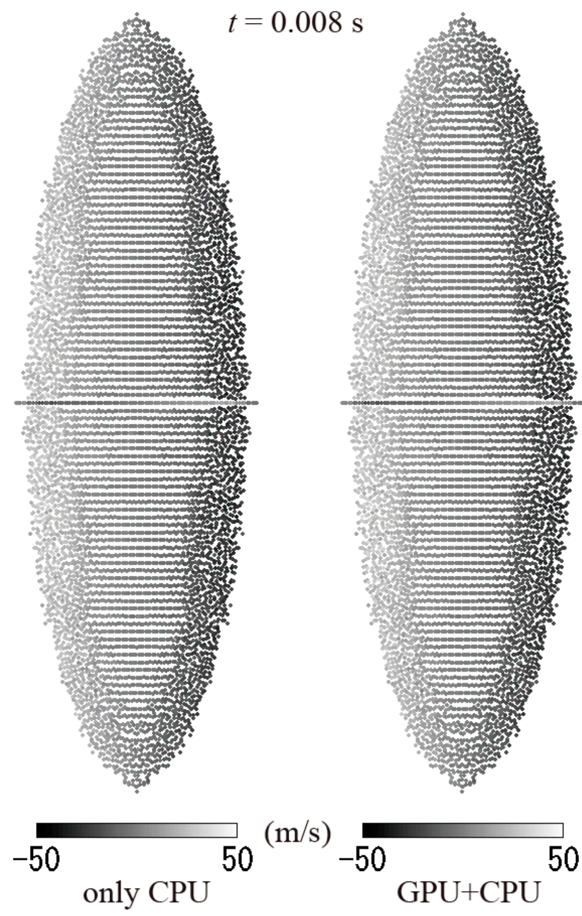


Fig. 11. Snapshots of water drops as they evolve to narrow ellipses at t = 0.008s

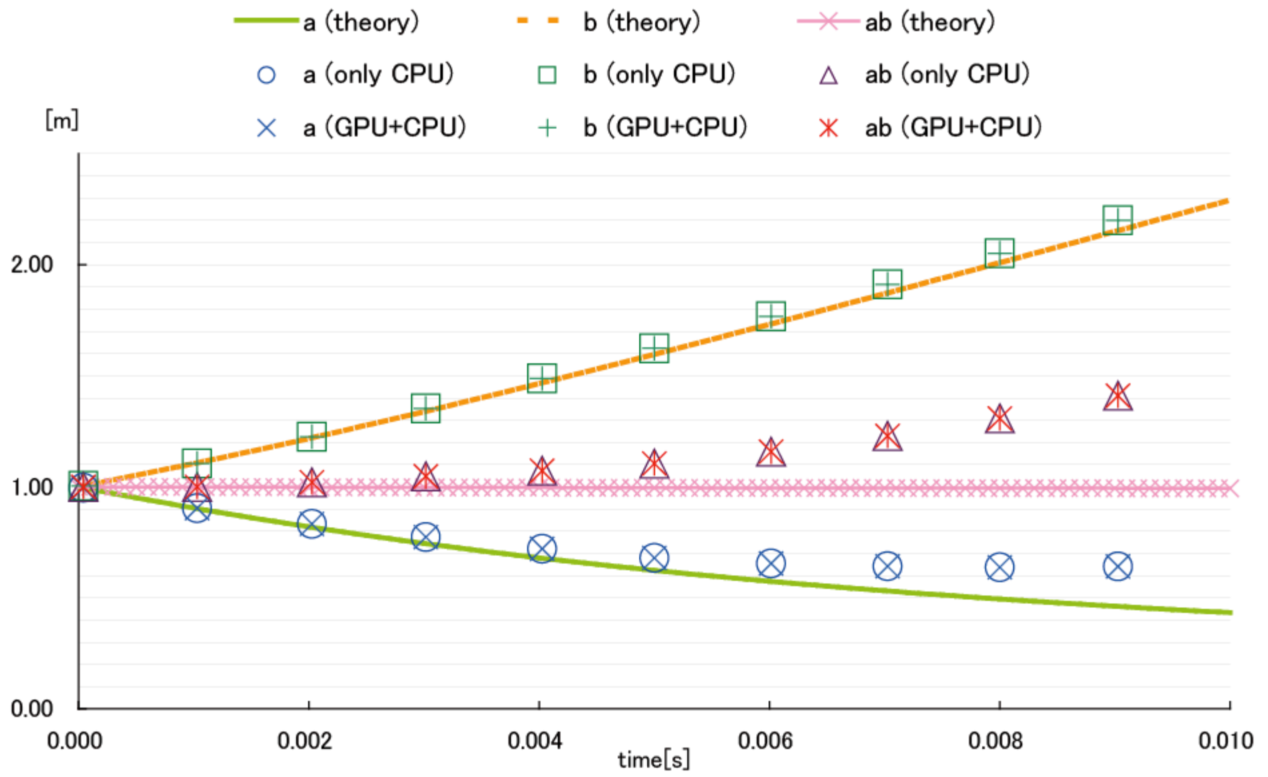


Fig. 12. Computational and theoretical variation of the elliptical drops
Semi-minor (a), semi-major (b) axes, and axes production (ab)

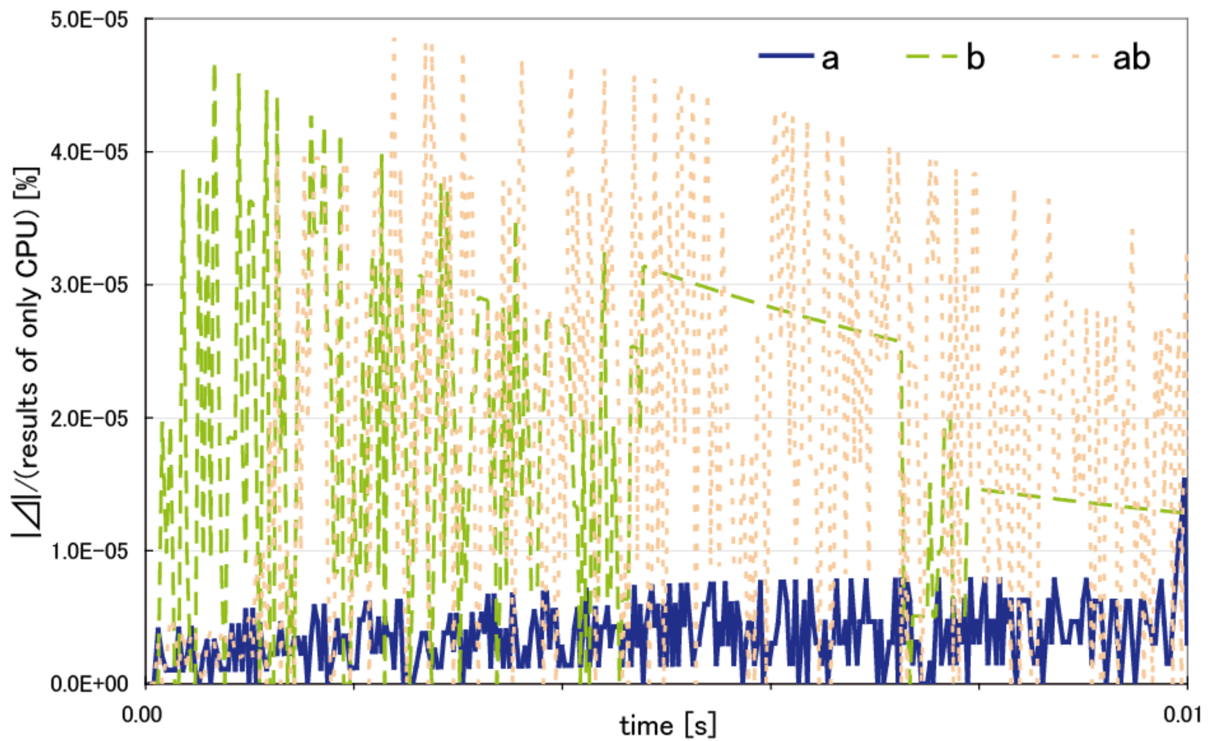


Fig. 13. Differences between results obtained from the GPU+CPU and the only CPU
Semi-minor (a), semi-major (b) axes, and axes production (ab)

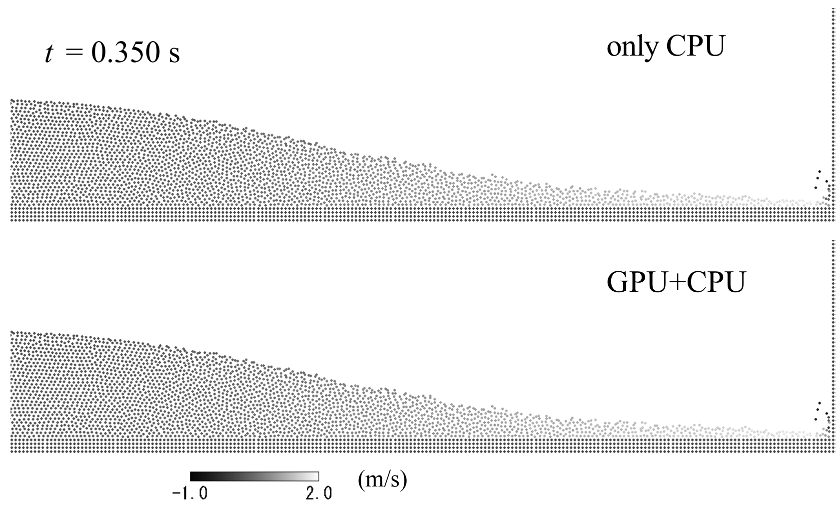


Fig. 14. Snapshots of water particles in dam break simulations at t = 0.350s

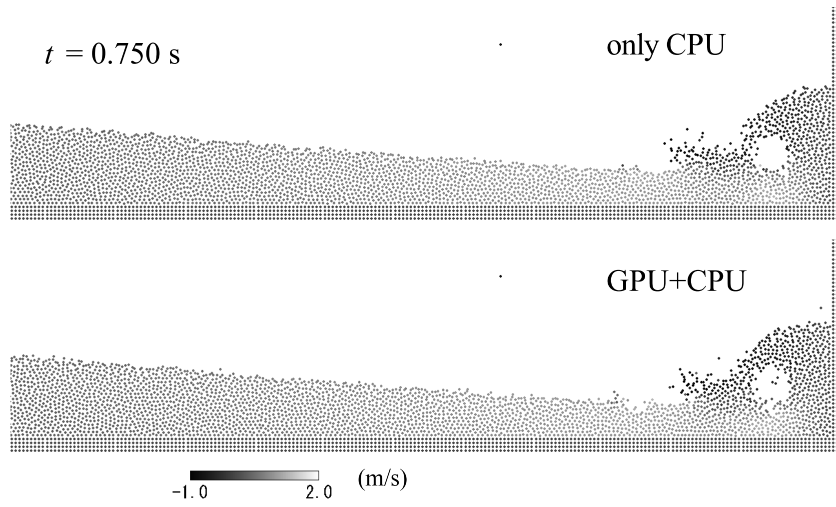


Fig. 15. Snapshots of water particles in dam break simulations at t = 0.750s