Computer Physics Communications 184 (2013) 2159–2168

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Highly scalable implementation of an N-body code on a GPU cluster

Yohei Miki [a,b,*], Daisuke Takahashi [c,e], Masao Mori [d,e]

[a] Graduate School of Pure and Applied Sciences, University of Tsukuba, Tsukuba, Ibaraki 305-8571, Japan
[b] Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
[c] Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
[d] Faculty of Pure and Applied Sciences, University of Tsukuba, Tsukuba, Ibaraki 305-8571, Japan
[e] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305-8577, Japan

ARTICLE INFO

Article history:
Received 3 August 2012
Received in revised form
28 February 2013
Accepted 19 April 2013
Available online 2 May 2013

Keywords:
N-body calculation
GPGPU
Performance tuning
Performance modeling
CUDA
OpenMP/MPI

ABSTRACT

We have developed a highly optimized code for collisionless N-body calculations based on direct summation. Our new optimization hides the global memory access latency, and the resulting CUDA code has a peak performance of 1006.7 GFlop/s in single precision (assuming 26 floating-point operations per interaction) with a single NVIDIA Tesla M2090 board. To improve the scalability of the OpenMP/MPI hybrid parallelized code, we have reduced the number of communications among multiple GPUs and have overlapped communications with computations to hide communication time. The code's performance was measured on the HA-PACS (Highly Accelerated Parallel Advanced system for Computational Sciences), a recently installed GPGPU cluster at University of Tsukuba. The results show excellent scalability with superlinear scaling when the number of N-body particles per GPU is less than 10^4 and parallel efficiency approaching unity when the number of N-body particles per GPU is greater than 10^4. The CUDA/OpenMP/MPI code has a peak performance of 255.5 TFlop/s when 256 NVIDIA Tesla M2090 boards are used, which is 75.0% of the theoretical peak performance.

1. Introduction

In astrophysics, collisionless N-body simulations are one of the most powerful tools for investigating structure formation of large scale structure, formation and evolution history of stellar systems such as galaxies. The fundamental equation of N-body simulations is Newton's equation of motion expressed as

$$\boldsymbol{a}_i = \sum_{j=0, j\neq i}^{N-1} \frac{G m_j \left(\boldsymbol{x}_j - \boldsymbol{x}_i\right)}{\left(\left|\boldsymbol{x}_j - \boldsymbol{x}_i\right|^2 + \epsilon^2\right)^{3/2}}, \tag{1}$$

where G is the gravitational constant, N is the number of particles, and m_i, $\boldsymbol{x}_i$ and $\boldsymbol{a}_i$ are the mass, position, and acceleration of the i-th particle, respectively. The gravitational softening parameter ϵ, introduced to avoid divergence due to division by zero, eliminates self-interaction when calculating gravitational force. The amount of computation for this equation is proportional to the number of *i-particles*, N_i, the particles on which gravitational force is exerted, and the number of *j-particles*, N_j, the particles that cause gravitational force.

Because a large number of N-body particles are necessary to investigate astrophysical phenomena in detail, many studies have been devoted to achieving fast computation for N-body simulations. Some of the proposed algorithms for reducing the amount of computation include the particle-mesh method and the tree method [1,2]. The computational complexity of the tree method is $O(N \log N)$ because the multipole expansion technique reduces the contribution of N_j.

In astrophysics, there are cases that require direct N-body simulations. For example, direct summation is employed to investigate the long-term evolution of globular clusters because their lifespan is much longer than dynamical time. Inaccurate gravitational force calculations incorrectly characterize the orbital evolution of the stars in such systems because numerous orbital integration steps are necessary for computing the time evolution of these systems. Indeed, a fourth-order Hermite scheme with double precision is often employed to investigate the dynamical evolution of globular clusters. Although the present study is not directly aimed at investigating the dynamical evolution of globular clusters, our optimized implementation of N-body calculations provides a useful tool for research in this field.

Investigations that achieve high performance and scalability for simple and characteristic algorithms constitute an important

* Corresponding author at: Graduate School of Pure and Applied Sciences, University of Tsukuba, Tsukuba, Ibaraki 305-8571, Japan.
E-mail address: ymiki@ccs.tsukuba.ac.jp (Y. Miki).

0010-4655/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cpc.2013.04.011

area of computer science. These investigations propose ways to optimize various complicated applications. An N-body simulation using direct summation is a well-known characteristic problem owing to its computation-intensive nature. Because a computational complexity of $O(N^2)$ severely limits the tractable problem size, computer science makes an essential contribution to the development of other sciences when it proposes performance improvements using recent architectures that can increase the tractable problem size.

One way to reduce the computation time for direct N-body calculations is to use an accelerator. The most famous and among the most successful accelerators for gravitational many-body systems is the GRAPE ("Gravity PipE") series [3,4]. Its high performance results from the pipelined and massively parallel architecture design, which enables massive parallelization of gravitational force calculations.

Recently, Graphics Processing Units (GPU) have become one of the most attractive accelerators owing to the development of General Purpose computing on GPU (GPGPU). Furthermore, many GPU clusters, including Titan, Tianhe-1A, Nebulae, TSUBAME 2.0, and HA-PACS, appear on the TOP 500 list [5], which indicates the popularity of GPU clusters. Therefore, improving the performance on GPU clusters is an important problem, particularly because the rapid increase in GPU performance and the development of GPU clusters enable the acceleration of numerical simulations, and thus the effectiveness of accelerator device.

Detailed investigations of the effectiveness of accelerating N-body simulations using GPUs are very important. Many previous studies have addressed the use of GPU to accelerate N-body simulations in various research fields, including direct summation for collisionless systems [6–10], the tree method for collisionless systems [8,9,11–13], and direct summation for collisional systems [14–17].

In this study, we develop a highly optimized hybrid parallel N-body code based on direct summation for collisionless systems. The remainder of this paper is organized as follows. Section 2 introduces optimization techniques for single GPU computations using CUDA. Sections 3 and 4 describe our parallelization strategy and its implementation using OpenMP and MPI, respectively. Section 5 presents our performance measurements, and Section 6 analyzes them. Finally, Section 7 summarizes this work.

2. Implementation and optimization of code for a single GPU

Many earlier studies have reported that massive parallelization about i-particles can achieve high levels of performance [6–10]. The implementation and optimization technique in the present study are based on Nyland et al. [7], whose source code is included in CUDA SDK for CUDA 3.x and 4.x and samples for CUDA 5.0, and on our previous work [10]. The implementations in [7,10] achieve high levels of performance–930 and 991 GFlop/s in single precision, respectively—on an NVIDIA Tesla M2090 board.

In both implementations, a block contains 256 threads and the shared memory stores the locations of 256 j-particles to minimize global memory access time within the innermost loop. The implementations in [7,10] differ in are the number of unrolls in the innermost loop, cache configurations, and number of operations required to calculate gravitational interactions. The number of unrolls in our innermost loop in [10] is 128 compared to 32 in [7]. We set "L1 cache preferred" because in most cases, this produced a slight performance increase over "shared memory preferred" in our experiments.

The most important difference is in the calculation of ${r_{ji}}^2 + \epsilon^2$. Both implementations use a `float3` variable `rji`, a `float` variable `eps2`, and a `float` variable `r2` to store the displacement vector $r_{ji} \equiv x_j - x_i$, ϵ^2, and the calculated value of ${r_{ji}}^2 + \epsilon^2$, respectively, as shown in Listing 1. The source codes for the two implementations in Listing 1 appear to be almost identical; however, the generated sets of instructions are quite different. The implementation in [7] first performs one multiplication and two fused multiply-add (FMA) operations, followed by one addition. According to the CUDA C Programming Guide [18], this code's computational cost is four clock cycles. On the other hand, our implementation in [10] performs only three FMA operations, which use three clock cycles, and is therefore faster than the implementation in [7]. The resulting optimization is primarily due to the fact that `r2` is calculated in the innermost loop; this small detail directly enhances performance.

The present study implements an additional optimization to achieve even better performance. Earlier studies have frequently emphasized the importance of utilizing shared memory because reducing the number of global memory accesses is an effective way to improve performance. In both [7,10], two `__syncthreads()` instructions are performed immediately before and after an instruction that loads from the global memory and stores to the shared memory, as shown in Listing 2. Here the shared memory and global memory each contain a `float4` array – `body[]` and `jpos[]`, respectively – to store the positions of j-particles. In both studies, the load from `jpos[]` and the store to `body[]` are performed in the same instruction. The two `__syncthreads()` instructions are necessary to maintain consistency while updating the set of j-particles shared by entire threads within a block.

Listing 1: Calculations of ${r_{ji}}^2 + \epsilon^2$ in Nyland et al. [7] and Miki et al. [10]

```
/* Implementation of Nyland et al. */
r2  = rji.x * rji.x
    + rji.y * rji.y
    + rji.z * rji.z;
r2 += eps2;

/* Our previous work */
r2  = eps2 + rji.x * rji.x
           + rji.y * rji.y
           + rji.z * rji.z;
```

Listing 2: Difference between previous implementations and this studies calculation of gravitational force

```
/* Previous works */
for(jh = 0; jh < Nj; jh += blockDim.x){
  __syncthreads();
  body[ii] = jpos[jh + threadIdx.x];
  __syncthreads();
  for(j = 0; j < blockDim.x; j++)
    calc_gravity();
}

/* This work */
for(jh = 0; jh < Nj; jh += blockDim.x){
  float4 pj = jpos[jh + threadIdx.x];
  __syncthreads();
  body[ii] = pj;
  __syncthreads();
  for(j = 0; j < blockDim.x; j++)
    calc_gravity();
}
```

Listing 3: Source code for gravitational attraction calculation for a single GPU

```
__global__ void calc_gravity(int Ni,
    float4 *ipos, float4 *acc, int Nj,
    float4 *jpos)
{
  __shared__ float4 body[NTHREADS];
  int  i =  blockIdx.x * blockDim.x
        + threadIdx.x;
  int ii = threadIdx.x;

  /* set i-particles */
  /* x, y, z, and mass */
  float4 pi = ipos[i];
  /* ax, ay, az, and potential */
  float4 ai = {0.0f, 0.0f, 0.0f, 0.0f};

  int nj = blockDim.x;
  for(int jh = 0; jh < Nj; jh += nj){
    /* set j-particles */
    float4 pj = jpos[jh + ii];
    __syncthreads();
    body[ii] = pj;
    __syncthreads();

#pragma unroll NUNROLL
    for(int j = 0; j < nj; j++){
      pj = body[j];

      float3 rji;
      rji.x = pj.x - pi.x;
      rji.y = pj.y - pi.y;
      rji.z = pj.z - pi.z;
      float rinv = rsqrtf(eps2 + rji.x *
          rji.x + rji.y * rji.y + rji.z
          * rji.z);
      /* ai.w += rinv * pj.w; */
      rinv = rinv * rinv * rinv * pj.w;
      ai.x += rji.x * rinv;
      ai.y += rji.y * rinv;
      ai.z += rji.z * rinv;
    }
  }
  atomicAdd(&(acc[i].x), ai.x);
  atomicAdd(&(acc[i].y), ai.y);
  atomicAdd(&(acc[i].z), ai.z);
  atomicAdd(&(acc[i].w), ai.w);
}
```

As far as a streaming multiprocessor (SM) contains multiple blocks, the memory access time for a block can be hidden by overlapping the access with the calculations of other blocks. The number of blocks per SM is a few in many cases owing to the limitations of the shared memory's capacity and the number of registers, and a block typically contains several warps. When an SM containing two blocks is used for an N-body calculation, there is only one possibility for overlapping instructions: one block calculates particle–particle interactions and the other block executes the load and store instructions. This is the only possibility because two `__syncthreads()` instructions separate the global memory accesses and particle–particle interaction calculations. Because the `__syncthreads()` instruction synchronizes entire threads within a block, the separate execution allows block-level overlapping but rules out warp-level overlapping of instructions in the implementations shown in Listing 2.

Table 1
Measurement environment.

Number of nodes	268
CPU	Intel Xeon E5-2670 16 cores per node, 2.6 GHz
RAM	128 GB (DDR 3, 1600 MHz)
GPU	NVIDIA Tesla M2090 512 CUDA cores, 1.3 GHz 4 boards per node
Video RAM	6 GB (GDDR 5, ECC on) per GPU
C compiler	icc 13.0.1.117 (gcc 4.4.5 compatibility)
MPI library	Intel MPI 4.1.0.024
CUDA toolkit	4.2.9
CUDA driver	304.54
Interconnection	Infiniband QDR ×2 rails
Network topology	Fat-tree

Therefore, the present study augments the possibilities of overlapping instructions through a careful modification of the CUDA code in Listing 2. We separate the instructions that load from the global memory and store to the shared memory. This allows CUDA schedulers to perform warp-level overlapped execution of these instructions since the execution of the load instruction and calculation is not separated. Because the number of blocks per SM cannot exceed the number of warps per SM, the new implementation provides more opportunities to hide slow global memory access time.

We also eliminate `if`-statements to avoid additional computational costs when dealing with arbitrary numbers of particles. Instead, we add massless particles to fill the i-particle and j-particle arrays when the number of particles is not an integral multiple of the number of threads.

At the end of Listing 3, we used atomic instructions to update information about i-particles' acceleration in the global memory. These atomic instructions are necessary to ensure consistency when the computations are performed on multiple GPUs.

To calculate gravitational potential defined as $-\sum_{j=0,j\neq i}^{N-1} Gm_j$ $(|\boldsymbol{x}_j - \boldsymbol{x}_i|^2 + \epsilon^2)^{-1/2}$, calculation of the commented-out line `ai.w += rinv * pj.w` in Listing 3 is executed. To avoid using `if`-statements in the innermost kernel, we subtract the term $-Gm_i/\epsilon$ at the end of the calculation to eliminate self-interaction. This simple method is useful as long as $\epsilon \neq 0$, which is common in collisionless N-body simulations.

The crucial parameters for enhancing performance in Listing 3 are the number of threads per block (`NTHREADS`), the number of unrolls for the innermost loop (`NUNROLL`), and the cache configuration ("shared memory preferred" or "L1 cache preferred"). In [7], it was claimed that the key to high performance is to have a large number of threads per block and a large number of unrolls. However, the complicated relation between these parameters prompts us to determine the optimal parameter settings for achieving the highest performance.

We performed a parameter study to examine the best configurations on an NVIDIA Tesla M2090 board when the number of N-body particles is 1048 576 (Table 1 provides more detailed information). Fig. 1 shows the performance measured in single precision as a function of the number of unrolls: the plotted lines show the results for different values of `NTHREADS` (T_{tot} in the legend of Fig. 1) and the left/right panels show the results for "shared memory preferred"/"L1 cache preferred" configurations. The top panels show that the optimal number of threads per block is 512. The peak performance for the parameter study is 1006.7 GFlop/s in single precision, which is reached in the case of 256 unrolls and the "L1 cache preferred" configuration. The peak performance in excess of 1 TFlop/s in single precision is due to the optimization introduced in this study. The effects of the

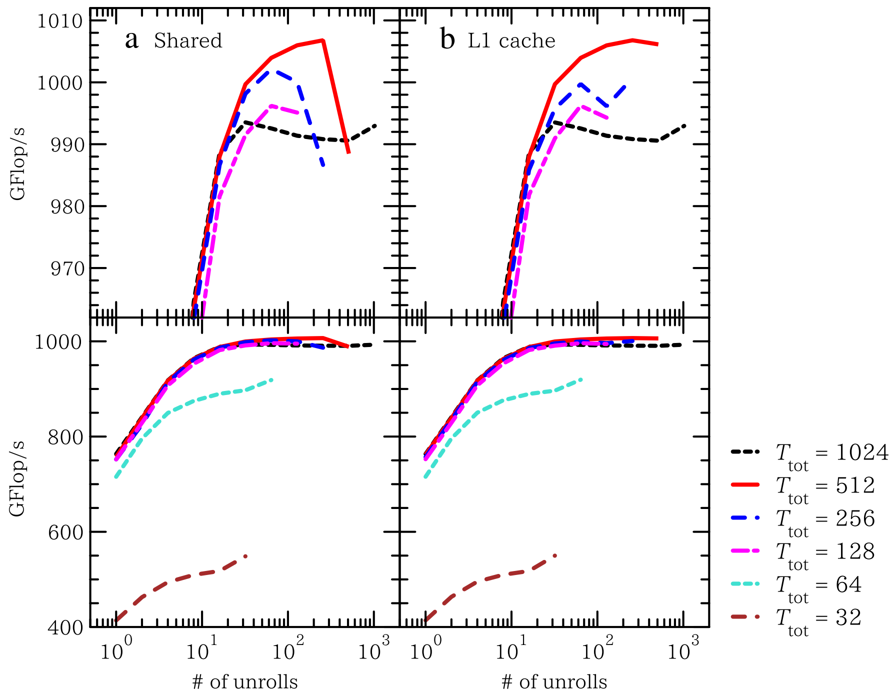

Fig. 1. Results of parameter study to determine the optimal parameter sets. Measured performance in single precision is plotted as a function of the number of unrolls. The brown, cyan, magenta, blue, red, and black lines indicate 32, 64, 128, …, and 1024 threads per block, respectively. The left and right panels show the results for the "shared memory preferred" and "L1 cache preferred" configurations, respectively. The top panels are enlargements of the bottom panels.

```
1: Update position data for $N/2$ $i$-particles on each device.
2: Calculate gravitational interactions among $N/2$ $i$-particles on each device.
3: Copy the updated position data for $N/2$ particles from the peer device as a new set of $j$-particles.
4: Calculate gravitational interactions between $N/2$ $i$-particles and $N/2$ $j$-particles.
5: Update the velocity data for $N/2$ $i$-particles on each device
```

Fig. 2. Algorithm for OpenMP parallelization.

optimization itself are not so great, but it is the tipping point for the performance exceeding 1 TFlop/s per single NVIDIA Tesla M2090 board.

3. Parallelization based on OpenMP

When the position data for N-body particles are divided between two GPUs, communication between the two devices is necessary for calculating gravitational interaction. Here peer-to-peer memory access between the two devices is a good choice to reduce communication time because peer-to-peer memory access does not require accessing memories through the CPU. Because peer-to-peer memory access requires sharing a pointer to global memory on each device within a process, we have parallelized the code using OpenMP.

An OpenMP thread controls a GPU, and each GPU stores half of the N-body particles: device 0 has position data for particles 0 through $N/2 - 1$ and device 1 has position data for the remaining N particles. The algorithm for calculating gravitational interactions and orbit integration using a leap-frog integrator is shown in Fig. 2.

Communication between GPUs (step 3 in Fig. 2) reduces the parallel efficiency by interrupting the kernel function's execution (step 4 in Fig. 2). Therefore, additional changes are necessary to realize high scalability when many GPUs are used. In the algorithm

```
1: Calculate gravitational interactions among $N/2$ $i$-particles on each device (stream ID sid).
2: Flip the stream ID sid through an exclusive or with unity.
3: Send the position data for $j$-particles to the peer device (stream ID sid).
4: Synchronize instructions related to stream ID sid.
5: Calculate gravitational interactions between $N/2$ $i$-particles and $N/2$ $j$-particles (stream ID sid).
```

Fig. 3. Implementation of "SendSync" mode.

```
1: Calculate gravitational interactions among $N/2$ $i$-particles on each device (stream ID sid).
2: Flip the stream ID sid through exclusive or with unity.
3: Synchronize OpenMP threads to confirm that both devices have already set their own $j$-particle data.
4: Receive the position data for $j$-particles from the peer device (stream ID sid).
5: Calculate gravitational interactions between $N/2$ $i$-particles and $N/2$ $j$-particles (stream ID sid).
```

Fig. 4. Implementation of "SyncRecv" mode.

in Fig. 2, the second and third steps can be performed concurrently. This enables overlapped execution of the gravitational interaction calculations and communications between the two GPUs, thus hiding the communication time. We use two CUDA streams to overlap the two instructions: one CUDA stream performs the first and second steps, while the other executes the remaining steps after the first has been executed.

Each of the two implementations below (SendSync in Fig. 3 and SyncRecv in Fig. 4) can be used to realize the algorithm in Fig. 2 (steps 2–4). SendSync is named after the data send and synchronize before the calculation, and SyncRecv is named after the synchronize and data receive before the calculation. It is not

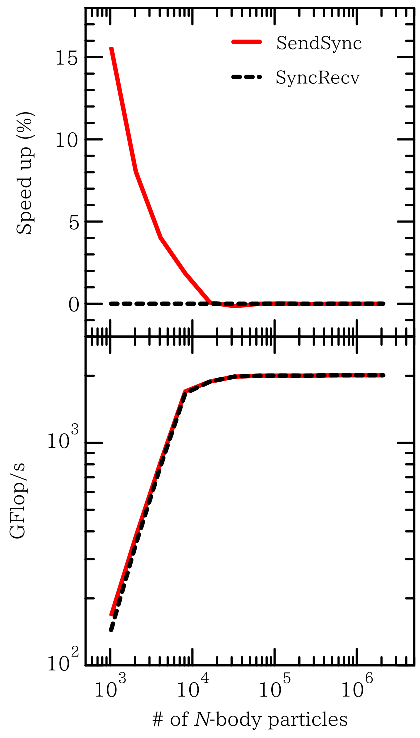

Fig. 5. Comparison between the SendSync (solid red lines) and SyncRecv (dotted black lines) modes. The bottom panel shows the measured performance of both modes as a function of the number of N-body particles. The top panel shows the speedup rate for the SendSync mode compared with the SyncRecv mode.

obvious which implementations is better; however, it is crucial to determine this for deciding how to implement the code. To this end, we must examine factors such as time needed to synchronize the CUDA streams and OpenMP threads, which implementation is easier to optimize for the CUDA/C compiler, and how the single GPU code performs.

Fig. 5 compares the measured performance of the algorithm using SendSync ("SendSync mode") and SyncRecv ("SyncRecv mode"). The red (black) line in the bottom panel shows the measured performance of the SendSync (SyncRecv) mode as a function of the number of N-body particles. The lines in the top panel represent the speedup rates from the SyncRecv mode. The experiment shows that SendSync mode is suitable in the test environment (listed in Table 1) for the low-N region.

4. Parallelization using the message passing interface

To handle data distribution on multiple GPUs attached to a distributed computational node, we must use the Message Passing Interface (MPI). It is straightforward and simplest to implement this case as a natural extension of the algorithm presented in Section 3. However, we have developed a more sophisticated algorithm to achieve a high degree of scalability by reducing the number of communications between the MPI processes. The overall algorithm is roughly divided into two phases: transfer and accumulation phases.

The transfer phase, shown in Fig. 6, is a natural extension of the algorithm in Section 3 for a multinode environment. In this phase, each MPI process transfers the position data for j-particles to the next MPI process (line 2 of Fig. 6) until all MPI processes complete calculation of gravitational interactions for all j-particles. If n_t MPI processes are involved in the transfer phase, the total number of communications is $n_t - 1$ owing to the ring-like communication pattern.

```
1: for $s = 0$ to $N_{\rm proc} - 2$ do
2:     Send data stored in the array 0 to array 1 on the next process connected
       as ring structure.
3:     Copy position data on array 1 as a new set of $j$-particles from host
       process to each device.
4:     Calculate gravitational interactions between $i$-particles and $j$-particles.
5:     Add the data on the array 0 to the array 1.
6:     Swap the array 0 and the array 1.
7: end for
```

Fig. 6. Algorithm of the transfer phase.

```
1: while data size of $j$-particles does not exceed the capacity of arrays 0
   and 1, or the number of $j$-particles becomes the half of the total number
   of $N$-body particles do
2:     Determine the pair process to exchange position data for $N$-body par-
       ticles.
3:     Send data stored in array 0 to array 1 on the pair process.
4:     Copy position data on array 1 as a new set of $j$-particles from host
       process to each device.
5:     Calculate gravitational interactions between $i$-particles and $j$-particles.
6:     Add the data on array 0 to array 1.
7:     Swap arrays 0 and 1.
8: end while
```

Fig. 7. Accumulation phase algorithm.

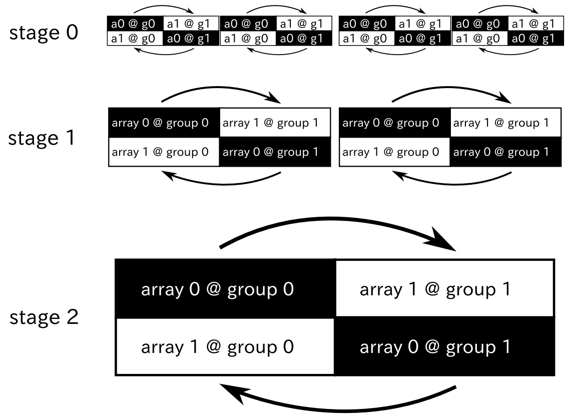

Fig. 8. Schematic view of data set growth at each stage of the accumulation phase. The black and white blocks are the source and destination arrays of the MPI communication, respectively.

To realize overlapped executions of the memory copy instructions from CPU to GPU and calculations on the GPU, we again use two CUDA streams, as described in Section 3. In addition, we utilize nonblocking communication functions such as `MPI_Isend` and `MPI_Irecv` to minimize the impacts of communications on the parallel efficiency by overlapping communication among MPI processes with other instructions.

Fig. 7 shows the accumulation phase that we introduced to reduce the number of communications. The fundamental idea behind the accumulation phase is quite simple: an MPI process sends position data for all known j-particles, namely the MPI process's initial j-particle data and those it has received in earlier stages, to other MPI processes.

Fig. 8 presents a schematic view of the accumulation phase for eight MPI processes. Each MPI process has two arrays (arrays 0 and

1 in Figs. 7 and 8). In the initial stage (stage 0), two MPI processes constitute a fundamental unit represented by two black boxes and two white boxes in Fig. 8. The black boxes (corresponding to array 0 in Fig. 7) contain position data for *j*-particles while the white boxes do not contain any data. In stage 0, every MPI process sends the data in its array 0 to array 1 of the other MPI process in the same unit, as indicated by the arrows (step 3 in Fig. 7). Before moving to the next stage, we merge the position data for the *j*-particles in arrays 0 and 1 in each unit (steps 6 and 7). At the end of this stage, both MPI processes have the same data for *j*-particles. At the beginning of stage 1, the MPI processes constitute new groups by merging each pair of groups that already share a dataset (groups connected by arrows in Fig. 8). After pairs of groups from the previous stage have been unified, the same steps are repeated until the accumulation phase completes.

The transferred data size doubles in each stage, as shown in Fig. 8. The number of communications is $\log_2 n_a$, where n_a is the maximum number of MPI processes constituting a unit in the accumulation phase. This is significantly smaller than $n_a - 1$ owing to its implementation as a binary tree through the exchange of datasets. The process rank of the target process of the exchanging is given by exclusive or of MPI rank with unity `<<(stage ID)`. It should be noted that the accumulation phase communication pattern is suitable for tree network topologies but not for mesh-torus topologies because MPI processes must also access distant processes.

The accumulation phase can be repeated as long as the size of the dataset for *j*-particles does not exceed the capacity of the *j*-particles arrays. When the size reaches the array's capacity, the communication algorithm proceeds to the transfer phase to complete calculation of the gravitational interactions of all *j*-particles.

5. Performance measurements

We measured the performance of our code on the HA-PACS (Highly Accelerated Parallel Advanced system for Computational Sciences), a newly installed GPU cluster at University of Tsukuba [19]. The HA-PACS is equipped with high-end GPUs and CPUs connected by PCI-express Generation 3.0. Each HA-PACS node consists of two Intel Sandy Bridge-EP sockets and four NVIDIA Tesla M2090 boards, and the CPUs support full-bandwidth connection of the GPUs without any performance bottlenecks. The interconnection network employs a dual-rail Infiniband QDR with a full bisection-bandwidth fat-tree configuration. The HA-PACS's peak performance is 1604 TFlop/s in single precision owing to the GPU's high performance of 1427 TFlop/s in single precision. Table 1 provides other details about the HA-PACS. Because two GPUs on every HA-PACS socket share the PCI lane, the HA-PACS is a suitable testbed for our implementation using peer-to-peer memory access. The fat-tree network topology, as opposed to mesh-torus interconnections, is suitable for the MPI communication's accumulation phase.

For measurement accuracy, we repeatedly measured the kernel function's total execution time. Since we use two CUDA streams to overlap the calculations and communications, as discussed in Sections 3 and 4, the streams must be synchronized at the end of the kernel function's sequential execution. This is because without the additional synchronization, a kernel function running the previous step could execute concurrently with a kernel function (related to another CUDA stream) running the current step, which would lead to an overestimation of the performance. Therefore, we must add either the `cudaDeviceSynchronize()` function or the `cudaStreamSynchronize()` function to the appropriate CUDA stream.

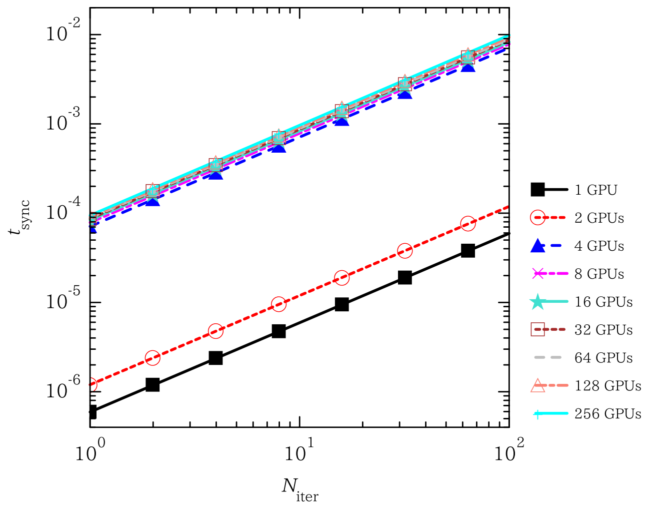

Fig. 9. Execution time of the synchronization instruction $t_{\rm sync}$ as a function of the number $N_{\rm iter}$ of synchronization instructions executed. Each symbol shows the measured $t_{\rm sync}$: black filled squares for a single GPU (CUDA), red open circles for two GPUs (CUDA with OpenMP), and other symbols for 4, 8, ..., 256 GPU boards (CUDA with OpenMP/MPI). The plotted lines show the fitted results based on the least-square method for each number of GPUs.

Table 2
Synchronization cost.

Number of GPU(s)	$t_{\rm sync}/N_{\rm iter}$ (s)
1	5.93×10^{-7}
2	1.19×10^{-6}
4	7.11×10^{-5}
8	7.84×10^{-5}
16	8.26×10^{-5}
32	8.68×10^{-5}
64	8.88×10^{-5}
128	9.37×10^{-5}
256	9.86×10^{-5}

The execution times for the synchronizing instructions themselves do not allow precise measurements, especially in the low-*N* region. Thus, we must evaluate their execution times independently and subtract these from the measured execution times. We measured the execution time of the synchronization instruction $t_{\rm sync}$ by measuring the execution time of $N_{\rm iter}$ synchronization instructions (`cudaStreamSynchronize()` for a single GPU, `cudaStreamSynchronize()` plus OpenMP barrier for two GPUs, `cudaStreamSynchronize()` plus OpenMP barrier with `MPI_Barrier()` for the CUDA/OpenMP/MPI implementation). Fig. 9 shows the resulting measurements: the horizontal axis represents the number of iterations, and the vertical axis represents the total execution time for the repeated execution of synchronize instructions. The black filled squares, red open circles, and other symbols show the measured $t_{\rm sync}$ for 1, 2, and 4–256 GPUs, respectively. The plotted lines in Fig. 9 are the fitted results based on the least square method. Table 2 lists the corresponding values of $t_{\rm sync}/N_{\rm iter}$.

We also measured the launch times for a dummy kernel function on the basis of the same strategy. The results in Fig. 10 show no significant difference between asynchronous and synchronous launches. The fitted launch times are 2.69×10^{-6} and 2.72×10^{-6} s for asynchronous and synchronous launches, respectively.

The maximum number of GPUs used in the performance measurements is 256. The number of *N*-body particles per GPU ranges from 512 to 1048 576, which corresponds to a maximum of $256 \times 1048\,576 = 2^{28} = 268\,435\,456$ particles. Fig. 11 shows the performance measurements. The horizontal axis represents the

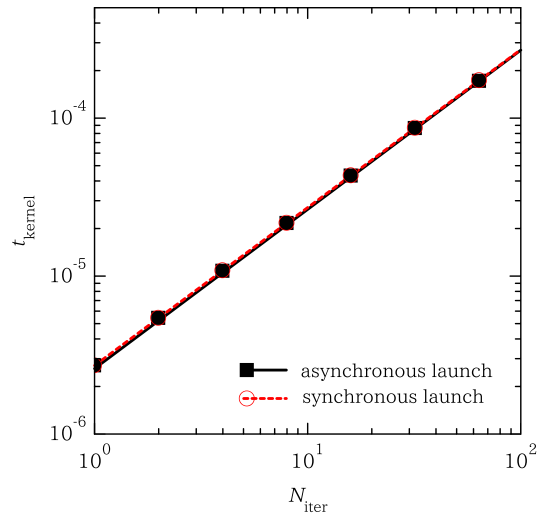

Fig. 10. Launch time for a kernel function, $t_{\rm kernel}$, as a function of the number $N_{\rm iter}$ of launches of a dummy kernel function. Black filled squares and red open circles show $t_{\rm kernel}$ for asynchronous and synchronous launches, respectively. The plotted lines show the fitted results based on the least-square method.

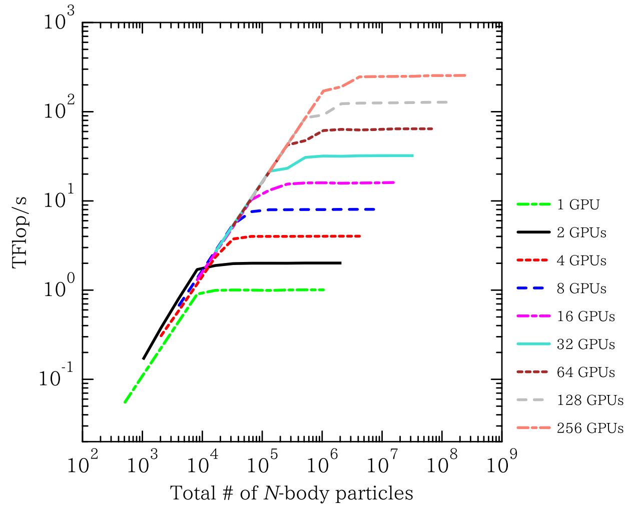

Fig. 11. Measured performance in single precision as a function of the total number of N-body particles, under the assumption that one interaction corresponds to 26 floating-point operations. The dot-dashed green line shows the performance for execution by a single GPU, and the solid black line shows the performance for execution by two GPUs employing OpenMP. The remaining lines show the performance for hybrid parallelization based on OpenMP and MPI; the lines from top to bottom show the performance of 256, 128, …, 4 GPUs.

total number of N-body particles, and the vertical axis represents the measured performance in single precision, assuming 26 floating-point operations for one interaction, which is the most plausible estimate for GPUs with compute capability 2.0 [10]. The lines from the bottom to the top of Fig. 11 represent the measured performance in single precision for 1, 2,…, 256 GPUs. The peak performance of 255.5 TFlop/s in single precision is reached when $N = 268\,435\,456$ and 256 GPUs are used.

6. Performance analysis

In this section, we present a performance model for the measured performance (Fig. 11). First, we focus on the measured performance of the CUDA code in Section 6.1. Then we move to the parallel efficiency of the OpenMP/MPI hybrid parallelized CUDA code in Section 6.2.

6.1. Performance analysis for single GPU calculation

We evaluate the measured performance for the single GPU calculation in this subsection. Because our implementation overlaps multiple instructions as much as possible to improve performance, analytic modeling of the performance is quite difficult. Thus, we have constructed a model equation with several unknown parameters and have determined that these parameters fit the results of the performance measurements, as discussed in [10]. In what follows, we estimate the total number of clock cycles needed to complete calculation of the gravitational interactions, $C_{\rm all}$.

There is a certain cost $C_{\rm kernel}$ for launching the kernel function to start the computation. In addition, each thread must load position data for i-particles from the global memory before calculating the interactions with j-particles. At this stage, a high latency L (one of the unknown parameters) of approximately 400–800 clock cycles occurs, according to the NVIDIA CUDA C Programming Guide [18]. The data transfer time from the global memory is negligibly smaller than L owing to the GPU's wide memory bandwidth of 177.6 GB/s.

Once each thread has stored the position data for i-particles in the registers, the gravitational force calculation begins. To achieve a high level of performance, we have divided the force calculation loop into two steps: the first step copies position data for $T_{\rm tot}$ j-particles from the global memory to the shared memory, and in the second step, threads calculate gravitational interactions among $T_{\rm tot}$ i-particles and $T_{\rm tot}$ j-particles, where $T_{\rm tot}$ is the number of threads per block. The calculation loop is performed $N_j/T_{\rm tot}$ times to calculate gravitational interactions from with all j-particles. Thus, the number of clock cycles needed to execute the gravitational force calculation in each block is

$$C_{\rm int} = \frac{N_j}{T_{\rm tot}} \times (L + T_{\rm tot}C_{\rm calc}) \times \frac{T_{\rm tot}}{N_{\rm core}} = \frac{N_j}{N_{\rm core}} (L + T_{\rm tot}C_{\rm calc}) \,. \quad (2)$$

The most important unknown parameter in Eq. (2) is the number of clock cycles for calculating a single interaction, $C_{\rm calc}$. $N_{\rm core}$ is the number of CUDA cores per SM, which is 32 for GPUs with compute capability 2.0. The term $T_{\rm tot}/N_{\rm core}$ represents the warp schedulers automatically dividing the computations into $T_{\rm tot}/N_{\rm core}$ groups owing to the limited number of CUDA cores per SM. Furthermore, if one SM contains multiple blocks, then the latency L is sometimes hidden by overlapping global memory data transfers and calculations of interactions among i-particles and j-particles. In such cases, $C_{\rm int}$ becomes

$$C_{\rm hide} = \frac{N_j}{N_{\rm core}} \max (L, T_{\rm tot}C_{\rm calc}) \,. \quad (3)$$

At the end of the computations, the resultant data must be transferred to the global memory in L clock cycles. To summarize this analysis, the number of clock cycles to complete computation of a block, $C_{\rm block}$, is $C_{\rm int}+C_{\rm kernel}+2L$ or $C_{\rm hide}+(C_{\rm kernel}+2L)/B_{\rm SM}$ when $B_{\rm SM}$ blocks are simultaneously assigned to an SM, is either one or greater than two. In the latter case, the factor $1/B_{\rm SM}$ represents the effect of overlapped memory transfer time due to the multiple blocks.

To evaluate $C_{\rm all}$ using $C_{\rm block}$, we need to determine the number of times the blocks must repeat the computation loop. The total number of blocks, $B_{\rm tot}$, is expressed as $N_i/T_{\rm tot}$; therefore, the number of loop iterations, N_{tot}, is represented in terms of the number of SMs within a GPU, $N_{\rm SM}$, as follows:

$$N_{\rm tot} = {\rm ceil}\left(\frac{B_{\rm tot}}{N_{\rm SM}}\right) = {\rm ceil}\left(\frac{N_i}{T_{\rm tot}N_{\rm SM}}\right) . \quad (4)$$

If $B_{SM} \geq 2$, then gravitational interaction calculations and transfers from the global memory are overlapped in some of the N_{tot} loop iterations. There are $N_{hide} = \text{floor}(N_{tot}/B_{SM})$ such iterations, and the number of remaining loops is $N_{rem} = N_{tot} - B_{SM}N_{hid}$. As the result, C_{all} is expressed as

$$C_{all} = N_{hide}\left[\frac{B_{SM}N_j}{N_{core}}\max\left(L, T_{tot}C_{calc}\right) + C_{kernel} + 2L\right] + N_{rem}\left[\frac{N_j}{N_{core}}\left(L + T_{tot}C_{calc}\right) + C_{kernel} + 2L\right]. \quad (5)$$

We estimate the unknown parameters C_{calc}, L, and C_{kernel}. We already have the estimates for L and C_{kernel}. The latency L is about 400–800 clock cycles, and the launch time for a kernel function C_{kernel} is the time measured for a dummy kernel launch, calculated as in Section 5. The resultant time of 2.7 ms corresponds to 3500 clock cycles. Therefore, we focus on the remaining unknown parameter C_{calc}.

First, we evaluate the maximum value of B_{SM} for our implementation. B_{SM} is determined by the number of available registers and the capacity of shared memory per SM, which are 32 768 and 16 kB, respectively, for an NVIDIA Tesla M2090 board with the "L1 cache preferred" option. All threads use 23 registers; therefore, $B_{SM} \leq 32768/(23 \times 512) \cong 2.8$ when T_{tot} is 512. Each block uses 8 kB to store the position data for 512 particles, since 16 bytes (four single-precision floating-point numbers) are needed to store each position. Therefore, the maximum value of B_{SM} is two for our implementation owing to the limitations of shared memory capacity and the number of registers per SM.

We estimate C_{calc} using Eq. (5). Since all quantities related to N_{tot} or N_{hide} are powers of two, N_{rem} becomes zero and the term related to N_{rem} vanishes. Therefore, overlapped gravitational interaction calculations and transfers from the global memory always occur when N_i is a power of two and is greater than $T_{tot}N_{SM} = 8192$. Furthermore, if $N_j \gg N_{core}/B_{SM}$, then the execution time for the inside interaction loop becomes much greater than that for the outside loop, and thus the contribution of the term $C_{kernel} + 2L$ becomes negligibly small. The measured execution time for the case of $N_i = N_j = 1048\,576$ is 28.4 s, corresponding to 36.9 billion clock cycles. Thus, according to Eq. (5), C_{calc} can be estimated to be $C_{all}/(N_iN_j/512) \cong 17.2$ clock cycles.

We must explain why we consider the execution time of 28.4 s to be due to C_{calc} rather than L. The arithmetic intensity $T_{tot}C_{calc}/L$ is a useful parameter for determining whether the dominant term is $T_{tot}C_{calc}$ or L. As we have noted, the latency L corresponds to 400–800 clock cycles. If L is dominant compared to the term $T_{tot}C_{calc}$ so that the arithmetic intensity is less than unity, then C_{calc} must be smaller than $L/T_{tot} \lesssim 3$. However, this is not the case because calculation of gravitational interaction involves at least three subtractions, six FMA operations, three multiplications, and calculation of an inverse square root. Therefore, the execution time of 28.4 s must represent the contribution of $T_{tot}C_{calc}$.

Eq. (5) reproduces the trends shown in Fig. 11 well. The characteristic point, which corresponds to a change in the performance trend, is $N = 16\,384$. The performance increases in proportion to N when N is less than 16 384, while a sustained performance is achieved when N is greater than 16 384.

The behavior in the low-N region is due to wasted SMs. N_{tot} in Eq. (5) is $\text{ceil}(N_i/8192)$, which is one or two when N_i is less than 16 384. This means that C_{all} is $(N_j/N_{core})(L + T_{tot}C_{calc}) + C_{kernel} + 2L$ or $(B_{SM}N_j/N_{core}) \times \max(L,\ T_{tot}C_{calc}) + C_{kernel} + 2L$ when N_{tot} is one or two. In both cases, the dependence of C_{all} on N is proportional to N_j only when the amount of computation is proportional to N_iN_j. This is why the measured performance increases proportionally to N.

Once all SMs have performed their computations, C_{all}'s dependence on N changes to be proportional to N_iN_j. Therefore, the behavior of the measured performance becomes stable when N is greater than 16 384.

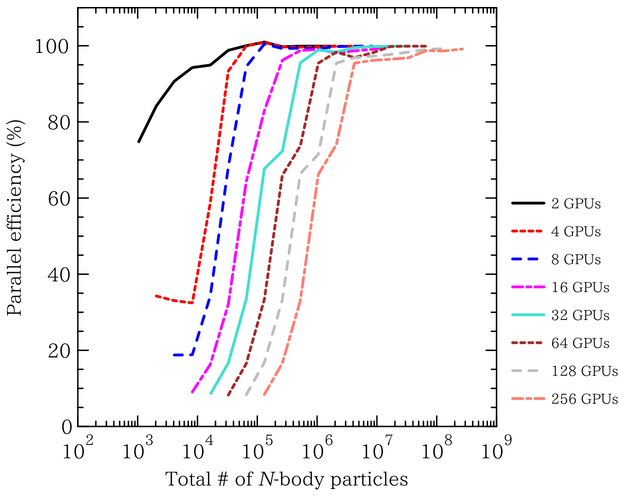

Fig. 12. Parallel efficiency $P(N;\ N_{GPU})/(P(N;\ 1) \times N_{GPU})$ plotted as a function of the total number of N-body particles shows strong scaling. The solid black line represent the efficiency of using two GPUs based on OpenMP, and the other plots show the parallel efficiency for the hybrid parallelized cases using OpenMP/MPI: from left to right, these represent 4, 8, ..., 256 GPUs.

6.2. Performance analysis for multiple GPU calculation

In this subsection, we analyze the performance when multiple GPUs are used. We denote the performance for N particles using N_{GPU} GPUs as $P(N;\ N_{GPU})$. Fig. 12 plots the parallel efficiency defined as $P(N;\ N_{GPU})/(P(N;\ 1) \times N_{GPU})$. Because we have measured the performance of a single GPU only for the region $512 \leq N \leq 1048\,576$, we assume that $P(N \geq 1048\,576;\ 1)$ equals the sustained value $P(1048\,576;\ 1)$. The horizontal axis represents the number of N-body particles, and the vertical axis represents the parallel efficiency $P(N;\ N_{GPU})/(P(N;\ 1) \times N_{GPU})$. Fig. 12 shows the parallel efficiencies for 2, 4, ..., 256 GPUs from left to right.

Fig. 12 clearly shows that the parallel efficiency of two GPUs based on OpenMP is greater than the parallel efficiency of the OpenMP/MPI hybrid cases in the low-N region. This is natural because the time needed to synchronize and/or transfer data for OpenMP is significantly shorter than that for OpenMP/MPI (e.g., Fig. 9). The saturation point for the increase in parallel efficiency roughly corresponds to the converging point of the SendSync and SyncRecv modes in Fig. 5. This suggests that the convergence of the measured performance is the result of the communication time completely being hidden by overlapping execution time.

On the other hand, the behavior of the OpenMP/MPI hybrid parallelized code shown in Fig. 12 is not the same as that of the CUDA/OpenMP code. To understand this, we have plotted the parallel efficiency with weak scaling, $P(N;\ N_{GPU})/(P(N/N_{GPU};\ 1) \times N_{GPU})$, in Fig. 13. The horizontal axis represents the number of particles per GPU, and the vertical axis represents the parallel efficiency. Fig. 13 shows the parallel efficiency for 2, 4, ..., 256 GPUs from left to right, the same as in Fig. 12. Hence, the performance of the kernel function in the measurement of $P(N;\ N_{GPU})$ is the same as that in the measurement of $P(N/N_{GPU};\ 1)$. However, Fig. 13 is more suitable than Fig. 12 for discussing the performance trends.

The parallel efficiency converges to unity when the number of particles per GPU is greater than 10^4. This is not surprising because we have overlapped communications among the multiple GPUs and calculations of gravitational interaction to hide the communication time. Owing to the overlap, communication and synchronization among the multiple processes do not impact the measured performance per GPU, so that the performance of the single GPU performance is retained. This is also supported by the similarity of the measured performance curves in Fig. 11.

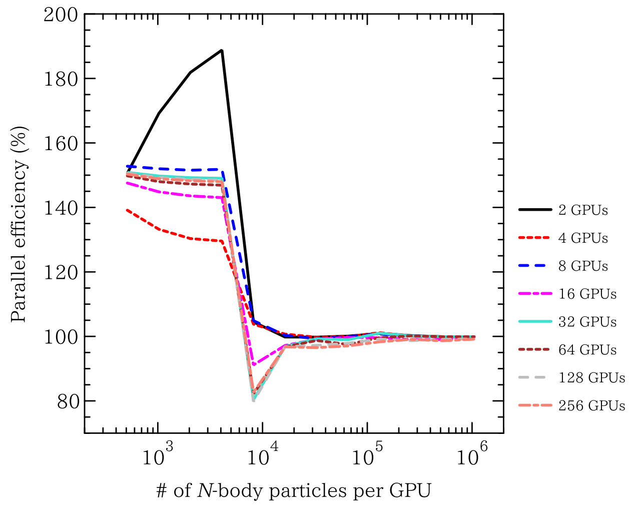

Fig. 13. Parallel efficiency $P(N;\, N_{\mathrm{GPU}})/(P(N/N_{\mathrm{GPU}};\, 1) \times N_{\mathrm{GPU}})$ plotted as a function of the number of particles per GPU shows weak scaling. The lines are same as those for the strong scaling in Fig. 12.

The behavior of the measured performance per GPU for the OpenMP/MPI parallelized CUDA code is roughly similar to that for the single GPU code.

The region where the number of N-body particles per GPU is less than 10^4 is quite interesting. The measured parallel efficiency reaches 1.5 when more than four GPUs are used. We consider this superlinear scaling to be a by-product of the overlapped communications and calculations. To overlap calculations within a GPU and communications between the CPU and GPU, the instructions should belong to different CUDA streams. Therefore, we utilize two CUDA streams.

The advantage of using multiple CUDA streams is that this opens the possibility of overlapping when the number of N-body particles per GPU is greater than 10^4. However, there is an additional benefit when the number of N-body particles per GPU is less than 10^4. As discussed in the previous section, some of the SMs do not calculate gravitational interaction when this number is less than 16 384. This means that the remaining SMs simultaneously perform the kernel function related to the second CUDA stream. This study uses two CUDA streams; therefore, the maximum expected speedup ratio is two. This is the reason for the superlinear scaling that appears when the number of N-body particles per GPU is less than 10^4. The measured parallel efficiency fails to reach two owing to the existence of additional instructions such as communications between the host and device and synchronization of related devices.

To assess the speedup ratio from the execution of two CUDA streams, we have constructed a simple model of kernel execution in the accumulation phase. The accumulation phase is the dominant phase of the MPI communication layer when the number of N-body particles per GPU is small. Fig. 14 shows an outline of the model. The executed instructions for each CUDA stream are shown as a function of time. The white boxes labeled $t_{\mathrm{calc}}(N';\, 1)$ denote the kernel execution time for $N' \equiv N/N_{\mathrm{GPU}}$ particles on a single GPU, and the dark boxes labeled $t_{\mathrm{sync}}(N_{\mathrm{GPU}})$ represent the time needed to synchronize N_{GPU} GPU boards. The necessary time for calculation depends on N_i and N_j, as discussed in Section 6.1. Because the number of j-particles doubles in each communication, the kernel execution time also doubles after each communication. The single exception is the initial step for stream 0. This difference represents the parallelization using OpenMP introduced in Section 3. Table 3 lists the measured execution times of the kernel function for a single GPU. The time for synchronization has already been estimated, as shown in Table 2. We ignore the time needed to transfer data between the host and a device or among multiple MPI processes because it is small compared with kernel execution time and synchronization time.

Table 3
Execution times for the kernel function.

N	Time (s)
512	1.23×10^{-4}
1 024	2.44×10^{-4}
2 048	4.85×10^{-4}
4 096	9.68×10^{-4}
8 192	1.93×10^{-3}
16 384	7.03×10^{-3}

The total execution time for the model shown in Fig. 14 can be expressed as

$$t_{\mathrm{calc}}(N;\, 1)\left(1 + \sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i\right) + t_{\mathrm{sync}}(N_{\mathrm{GPU}})\log_4(N_{\mathrm{GPU}}) \tag{6}$$

for cases where N_{GPU} is not a power of four corresponding to stream 0 in the figure, and as

$$2t_{\mathrm{calc}}(N;\, 1)\sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i + t_{\mathrm{sync}}(N_{\mathrm{GPU}})\log_4(N_{\mathrm{GPU}}) \tag{7}$$

for cases where N_{GPU} is a power of four corresponding to stream 1. Therefore, the theoretical parallel efficiency is

$$\left[\left(1 + \sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i\right)\Big/ N_{\mathrm{GPU}} + \frac{t_{\mathrm{sync}}(N_{\mathrm{GPU}})\log_4(N_{\mathrm{GPU}})}{t_{\mathrm{calc}}(N/N_{\mathrm{GPU}};\, 1)N_{\mathrm{GPU}}}\right]^{-1} \tag{8}$$

and

$$\left[\frac{2}{N_{\mathrm{GPU}}}\sum_{i=0}^{\log_4(N_{\mathrm{GPU}}/2)} 4^i + \frac{t_{\mathrm{sync}}(N_{\mathrm{GPU}})\log_4(N_{\mathrm{GPU}})}{t_{\mathrm{calc}}(N/N_{\mathrm{GPU}};\, 1)N_{\mathrm{GPU}}}\right]^{-1} \tag{9}$$

for streams 0 and 1, respectively.

Because Eqs. (8) and (9) are too complicated for discussing the fundamental properties of the parallel efficiency, we examine two complementary limits. First, we assume $t_{\mathrm{sync}}(N_{\mathrm{GPU}})$ is zero to evaluate the parallel efficiency's upper limit. In this case, the estimated theoretical parallel efficiency is 2.0, 1.33, 1.6, 1.45, 1.52, 1.49, and 1.51 when 4, 8, 16, 32, 64, 128, and 256 GPUs are used, respectively. Next, we assume that $t_{\mathrm{sync}}(N_{\mathrm{GPU}})$ is twice $t_{\mathrm{calc}}(N;\, 1)$ for $N \le 16\,384$ and $N_{\mathrm{GPU}} \ge 4$ in order to evaluate the lower limit. In this case, the theoretical parallel efficiency is 1.0, 1.0, 1.14, 1.23, 1.33, 1.39, and 1.44 for 4, 8, 16, 32, and 64, 128, and 256 GPUs are used, respectively. Although these estimates do not explain all the data in detail, they provide a rough explanation of the trends in Fig. 13. Furthermore, these theoretical estimates suggest that the parallel efficiency approaches 1.5 as the number of GPUs increases.

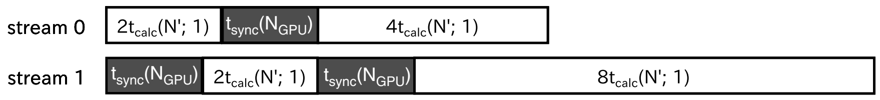

Fig. 14. Schematic view of the overlapped execution model for different CUDA streams. The horizontal axis represents time, and each block shows the instructions executed: white blocks represent calculations and dark blocks represent synchronization.

7. Summary

We have developed a highly optimized code for collisionless N-body calculations based on direct summation.

Our new optimization, aimed at hiding the global memory access latency, enables a peak performance in excess of 1 TFlop/s per single NVIDIA Tesla M2090 board. The performance of the CUDA code peaks at 1006.7 GFlop/s in single precision when $N = 1048\,576$ (assuming 26 floating-point operations per interaction), which is 75.7% of the theoretical peak performance.

To improve the scalability of the OpenMP/MPI hybrid parallelized CUDA code, we have reduced the number of communications among the multiple GPUs and have overlapped communications with computations to hide communication time. The performance of the code peaks at 255.5 TFlop/s in single precision when $N = 268\,435\,456$ and 256 NVIDIA Tesla M2090 boards are used, which is 75.0% of the theoretical peak performance and 99.1% of the parallel efficiency.

The code has excellent scalability with a superlinear scaling 1.5 when the number of N-body particles per GPU is less than 16 384, and the parallel efficiency approaches unity when the number of N-body particles per GPU is greater than 16 384. Finally, we have presented a performance model that explains these trends well.

Acknowledgments

We thank Go Ogiya for fruitful detailed discussion and Taisuke Boku for his useful comments. We are also grateful to an anonymous referee for constructive comments that have improved our manuscript. The numerical simulations were performed on the HA-PACS at the Center for Computational Sciences, University of Tsukuba. We are grateful to the HA-PACS project team for the opportunity to use a supercomputer under construction and also for their generous technical assistances. This work was partially supported by the program of the Pre-Strategic Initiatives, University of Tsukuba and by the Grant-in-Aid for Scientific Research (A) (21244013), (C) (24500029), and (C) (25400222).

References

[1] R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles, 1988.
[2] J. Barnes, P. Hut, A hierarchical $O(N \log N)$ force-calculation algorithm, Nature 324 (1986) 446–449. http://dx.doi.org/10.1038/324446a0.
[3] S.K. Okumura, J. Makino, T. Ebisuzaki, T. Fukushige, T. Ito, D. Sugimoto, E. Hashimoto, K. Tomida, N. Miyakawa, Highly parallelized special-purpose computer, GRAPE-3, Publications of the Astronomical Society of Japan 45 (1993) 329–338.
[4] A. Kawai, T. Fukushige, J. Makino, M. Taiji, GRAPE-5: a special-purpose computer for N-body simulations, Publications of the Astronomical Society of Japan 52 (2000) 659–676.
[5] Top500 list. URL http://www.top500.org/.
[6] T. Hamada, T. Iitaka, The chamomile scheme: an optimized algorithm for N-body simulations on programmable graphics processing units, ArXiv Astrophysics e-prints. arXiv:arXiv:astro-ph/0703100.
[7] L. Nyland, M. Harris, J. Prins, Fast N-Body Simulation with CUDA, 2007.
[8] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, M. Taiji, 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, ACM, New York, NY, USA, 2009, pp. 62:1–62:12. http://doi.acm.org/10.1145/1654059.1654123.
[9] T. Hamada, K. Nitadori, 190 TFlops astrophysical N-body simulation on a cluster of GPUs, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–9. http://dx.doi.org/10.1109/SC.2010.1.
[10] Y. Miki, D. Takahashi, M. Mori, A fast implementation and performance analysis of collisionless N-body code based on GPGPU, in: Proceedings of the International Conference on Computational Science, ICCS 2012, Procedia Computer Science 9 (0) (2012) 96–105. http://dx.doi.org/10.1016/j.procs.2012.04.011. URL http://www.sciencedirect.com/science/article/pii/S1877050912001329.
[11] E. Gaburov, J. Bédorf, S. Portegies Zwart, Gravitational tree-code on graphics processing units: implementation in CUDA, Procedia Computer Science 1 (2010) 1119–1127. http://dx.doi.org/10.1016/j.procs.2010.04.124.
[12] J. Bédorf, E. Gaburov, S. Portegies Zwart, A sparse octree gravitational N-body code that runs entirely on the GPU processor, Journal of Computational Physics 231 (2012) 2825–2839. http://dx.doi.org/10.1016/j.jcp.2011.12.024.
[13] N. Nakasato, Implementation of a parallel tree method on a GPU, in: Scientific Computation Methods and Applications, Journal of Computational Science 3 (3) (2012) 132–141. http://dx.doi.org/10.1016/j.jocs.2011.01.006. URL http://www.sciencedirect.com/science/article/pii/S1877750311000135.
[14] S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. Portegies Zwart, P. Berczik, Performance analysis of direct N-body algorithms on special-purpose supercomputers, New Astronomy 12 (2007) 357–377. http://dx.doi.org/10.1016/j.newast.2006.11.003.
[15] S.F. Portegies Zwart, R.G. Belleman, P.M. Geldof, High-performance direct gravitational N-body simulations on graphics processing units, New Astronomy 12 (2007) 641–650. http://dx.doi.org/10.1016/j.newast.2007.05.004.
[16] R.G. Belleman, J. Bédorf, S.F. Portegies Zwart, High performance direct gravitational N-body simulations on graphics processing units II: an implementation in CUDA, New Astronomy 13 (2008) 103–112. http://dx.doi.org/10.1016/j.newast.2007.07.004.
[17] E. Gaburov, S. Harfst, S. Portegies Zwart, SAPPORO: a way to turn your graphics cards into a GRAPE-6, New Astronomy 14 (2009) 630–637. http://dx.doi.org/10.1016/j.newast.2009.03.002.
[18] Nvidia, NVIDIA CUDA C Programming Guide Version 4.2, 2012.
[19] HA-PACS Project. URL http://www.ccs.tsukuba.ac.jp/CCS/eng/research-activities/projects/ha-pacs.