42 TFlops Hierarchical *N*-body Simulations on GPUs with Applications in both Astrophysics and Turbulence

Tsuyoshi Hamada
Department of Computer and Information Sciences
Nagasaki University
Nagasaki, Japan
hamada@cis.nagasaki-u.ac.jp

Rio Yokota
Department of Mathematics
University of Bristol
Bristol, United Kingdom
rio.yokota@bristol.ac.uk

Keigo Nitadori
High-Performance Molecular Simulation Team
RIKEN Advanced Science Institute
Wako, Japan
keigo@riken.jp

Tetsu Narumi
Department of Computer Science
University of Electro-Communications
Tokyo, Japan
narumi@cs.uec.ac.jp

Kenji Yasuoka
Department of Mechanical Engineering
Keio University
Yokohama, Japan
yasuoka@mech.keio.ac.jp

Makoto Taiji
High-Performance Molecular Simulation Team
RIKEN Advanced Science Institute
Wako, Japan
taiji@riken.jp

ABSTRACT

As an entry for the 2009 Gordon Bell price/performance prize, we present the results of two different hierarchical N-body simulations on a cluster of 256 graphics processing units (GPUs). Unlike many previous N-body simulations on GPUs that scale as $\mathcal{O}(N^2)$, the present method calculates the $\mathcal{O}(N \log N)$ treecode and $\mathcal{O}(N)$ fast multipole method (FMM) on the GPUs with unprecedented efficiency. We demonstrate the performance of our method by choosing one standard application –a gravitational N-body simulation– and one non-standard application –simulation of turbulence using vortex particles. The gravitational simulation using the treecode with 1,608,044,129 particles showed a sustained performance of 42.15 TFlops. The vortex particle simulation of homogeneous isotropic turbulence using the periodic FMM with 16,777,216 particles showed a sustained performance of 20.2 TFlops. The overall cost of the hardware was 228,912 dollars. The maximum corrected performance is 28.1TFlops for the gravitational simulation, which results in a cost performance of **124 MFlops/$**. This correction is performed by counting the Flops based on the most efficient CPU algorithm. Any extra Flops that arise from the GPU implementation and parameter differences are not included in the 124 MFlops/$.

1. INTRODUCTION

Groundbreaking N-body simulations have won the Gordon Bell prize in 1992 [29],1995-2003 [21, 5, 30, 28, 14, 18, 20, 19], and 2006 [22], in many cases with the aid of special-purpose computers, *i.e.* GRAPE and MDGRAPE. However, in the field of N-body simulations during the past few years, graphics processing units (GPUs) have been preferred, rather than the expensive special-purpose computers.

The direct $O(N^2)$ N-body simulations using NVIDIA's CUDA (Compute Unified Device Architecture) programming environment have achieved a performance of over 200 GFlops on a single GPU. Hamada *et al.* reported a performance of 256 GFlops for a 131,072-body simulation on an NVIDIA GeForce 8800 GTX [10]. Nyland *et al.* reported a performance of 342 GFlops for a 16,384-body simulation on an NVIDIA GeForce 8800 GTX [24]. Belleman *et al.* also reported a GPU performance of 340 GFlops for a 131,072-body simulation on an NVIDIA GeForce 8800 GTX with their code `Kirin` [4]. More recently, Hamada reported a performance of 653 GFlops for a 131,072-body simulation on an NVIDIA GeForce 8800 GTS/512 [11][1]. Gaburov *et al.* report a performance of 800 GFlops for a 10^6 particle simulation on two GeForce 9800GX2s with their code `Sapporo` [6].

The use of such data-parallel processors are a necessary but not sufficient condition for executing large scale N-body simulations within a reasonable amount of time. Hierarchical algorithms such as the tree algorithm [2] and fast multipole method (FMM) [8] are also indispensable requisites, because they bring the order of the operation count from $\mathcal{O}(N^2)$ down to $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$. Stock & Gharakhani [27] implemented the treecode on the GPU to accelerate their vortex method calculation. Similarly, Gumerov & Duraiswami [9] calculated the Coulomb interaction using the FMM on GPUs.

One of the most common problems in previous GPU implementations of hierarchical algorithms was the inefficient use of parallel pipelines when the number of target particles per cell was small. In order to solve this problem, we developed a method that could pack "multiple walks" that are

[1] `http://progrape.jp/cs/`

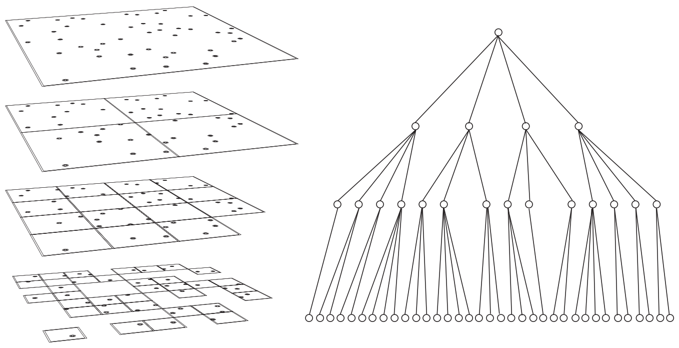


Figure 1: Hierarchical division of the calculation domain and the corresponding tree structure

evaluated by the GPU simultaneously. In order to implement this novel approach, we redesigned the tree algorithm and FMM. As a result, a drastic gain in performance was achieved as compared to previous implementations of these hierarchical algorithms on GPUs.

We demonstrate the performance of our hierarchical N-body methods by choosing one standard application –a gravitational N-body simulation– and one non-standard application –simulation of turbulence using vortex particles. First, We performed a cosmological N-body simulation of a sphere of radius 136.28 Mpc (mega parsec) with 1,608,044,129 particles for 327 timesteps using our GPU-tree algorithm. Subsequently, we applied our GPU-FMM to the calculation of a homogeneous isotropic turbulence using $256^3 = 16,777,216$ vortex particles.

The remaining sections of this paper is organized as follows. In section 2, we present a novel GPU implementation of the tree algorithm, and discuss the problems of the previous approaches, while explaining why the present method performs better. In section 3, the results of a cosmological N-body simulation using our GPU-treecode are shown. In section 4, we present a novel GPU implementation of the FMM, and the results of its performance validation tests. In section 5, the results of a vortex particle simulation of homogeneous isotropic turbulence using our GPU-FMM are shown. In section 6 we calculate the overall cost performance of our simulations. Finally, the conclusions are drawn in section 7.

2. GPU TREE CODE

2.1 The Tree code

The treecode by Barnes and Hut [2] represents a system of N particles in a hierarchical manner by the use of a spatial tree data structure, as shown in Figure 1. Whenever the force on a particle is required, the tree is traversed, starting at the root. At each level, a gravity center of particles in a cell is added to an "interaction list" if it is distant enough for the truncated series approximation to maintain sufficient accuracy in the force. If the cell is too close, the subcells are used for the force evaluation or opened further. This tree traversal procedure is called a "walk". The "interaction list" contains particles themselves or gravity centers of cells.

The original BH algorithm performs the walk for each particle separately. The parallel code [17][2] that we have developed is based on Barnes' modified tree algorithm [3], where the neighboring particles are grouped together to share an "interaction list". Figure 2 is an illustration of the shared "interaction list" for the particles shown in red. The modified tree algorithm terminates the tree traversal when the cell contains less than N_{crit} particles. This results in an average number of particles per cell N_g in between $N_{crit}/8$ and N_{crit}, depending on the non-uniformity. The modified tree algorithm reduces the calculation cost of the host computer by roughly a factor of N_g. On the other hand, the amount of work on the force pipelines increases as we increase N_g, since the interaction list becomes longer. There is, therefore, an optimal N_g at which the total computation time is minimum. On CPUs, the optimal value is typically $N_g \approx 32$ [3]. On GPUs, N_{crit} is around 1000. Note that the performance (Flops) we are reporting is the corrected value after the difference in the N_g between the CPU and GPU is taken into account. This correction is performed by counting the Flops based on the most efficient CPU algorithm. Any extra Flops that arise from the difference in N_{crit} are not counted.

Load balancing in our parallel code was achieved by space decomposition using an orthogonal recursive bisection (ORB) [29]. In an ORB, the box of particles is partitioned into two boxes with an equal number of particles using a separating hyperplane oriented to lie along the smallest dimension. This partitioning process is repeated recursively until the number of divided boxes becomes the same as the number of processors. When the number of processors is not a power of two, it is a trivial matter to adjust the division at each

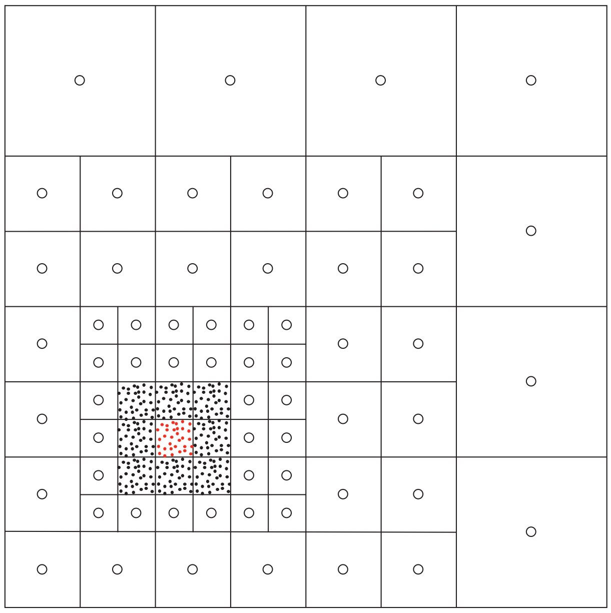


Figure 2: Interaction list in Barnes' modified tree algorithm. The small black dots represent the interacting particles. The white circles represent the truncated series expansions for each cell. Both the dots and circles are included in an interaction list.

[2]`http://www.artcompsci.org/~makino/softwares/pC++tree/index.html`

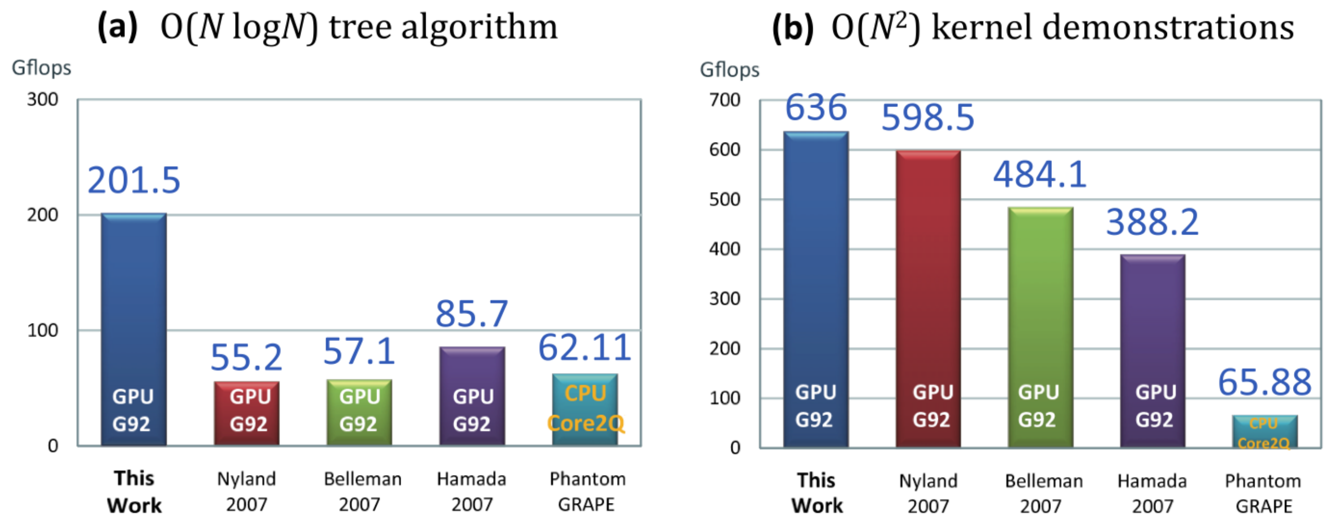


Figure 3: Comparison of the performances of the N-body kernel using (a) the $\mathcal{O}(N \log N)$ tree algorithm and (b) the $\mathcal{O}(N^2)$ direct-summation approach with shared timestep scheme on a single GeForce 8800 GTS. For the Flops count, we used a unified operation count of 38 for the calculation between a pair of particles.

step accordingly. It is expensive to recompute the ORB at each time step, but the cost of incremental load-balancing is negligible.

2.2 Comparison of GPU tree codes

Figure 3 shows the performance of the $\mathcal{O}(N^2)$ direct summation, and the $\mathcal{O}(N \log N)$ tree algorithm with the shared time-step scheme. We compared four different GPU implementations of the direct/tree N-body algorithm, using the same hardware –a single NVIDIA GeForce 8800 GTS GPU connected to an Intel Core 2 Quad Q6600 CPU based PC. GPU codes include (1) CUNBODY-1.0 by Hamada *et al.* (Hereafter Hamada, 2007 [10]), (2) `Kirin` by Belleman *et al.* (Belleman,2007 [4]) , (3) the algorithm proposed by Nyland *et al.* for the GPU Gems 3 (Nyland, 2007 [24]), and (4) the modified algorithm proposed in this work. Since we did not have the original source code for `Kirin`'s implementation, the corresponding results were measured through our own implementation of their method as they describe it in their paper. We also considered the results of the tree algorithm executed on a CPU with the highly-optimized code "Phantom-GRAPE (GRAvity PipE)" developed by Nitadori *et al.* [23] [3]. The "Phantom-GRAPE" was highly optimized using Intel's streaming SIMD extensions (SSE) in the assembly language, and is known to be one of the fastest variations of the tree algorithm for general-purpose processors. Plummer spheres with 65,536 and 4,194,304 particles were used for the direct algorithm and the tree algorithm, respectively. We set a softening parameter $\varepsilon = 0.025$.

For the direct $\mathcal{O}(N^2)$ calculations, all four variations of the GPU implementation showed a better performance than the highly optimized CPU implementation "Phantom-GRAPE". The difference in the performance of the four implementations are as follows. The approach of (Belleman, 2007 [4]) and (Nyland, 2007 [24]) are essentially the same, except for the number of threads. Since the approach by (Nyland, 2007 [24]) uses a larger number of threads (32,768 against 2,048), their implementation exhibited a better performance. The performance of CUNBOODY-1.0/(Hamada, 2007 [10]) was poor due to an overhead of the force reduction since it was designed to operate efficiently with the tree algorithm. This problem and the details of our proposed approach will be explained later.

As clearly shown in Figure 3 (a), our new code is much faster than other codes for the tree algorithm. It has a speed performance that is 2.4 times that of the previous GPU implementations and 3.2 times faster than that of optimized CPU codes. For the Flops count, we used a unified operation count of 38 for the calculation between a pair of particles (see section 3.1 for details). On the other hand, for the previous tree algorithms, the performance gains by the GPU were almost negligible (Belleman, 2007 [4]) and (Nyland, 2007 [24]) or were fairly small (Hamada, 2007 [10]) compared to CPUs. In order to explain why the increase in performance is small, we first describe the main concepts of the past implementations.

Historically, the task of extracting fine-grained parallelism from tree algorithms started with Barnes' modified tree algorithm on vector processors[3]. This was followed by a few attempts to run a similar method on the GRAPE [16], GRAPE-3 [1], GRAPE-5 [13], and on parallel GRAPE-6 hardware [17]. They were able to achieve a speed roughly 10 times faster than that of the fastest parallel code on general-purpose parallel computers. [17]

More recently, there has been a large amount of interest in porting these algorithms to GPUs. In order to clearly describe the differences between the different GPU implementations of the tree algorithm, we will introduce a conceptual diagram of the basic components of the GPU calculation in Figure 4. Each row corresponds to a thread in the GPU, and each group of rows corresponds to a SIMD block (thread block). The horizontal axis indicates the time of processing, and the blue circles indicate threads.

[3] `http://progrape.jp/phantom/`

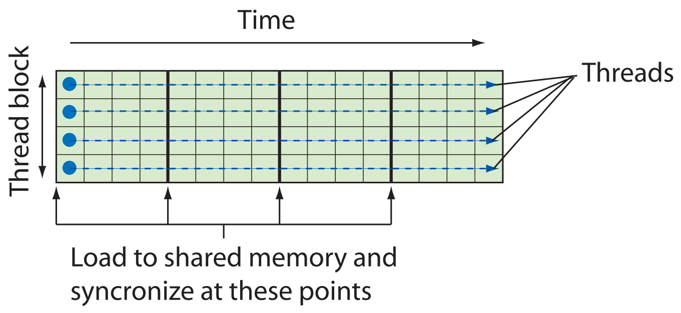


Figure 4: Definition of the conceptual diagram used in the following three figures

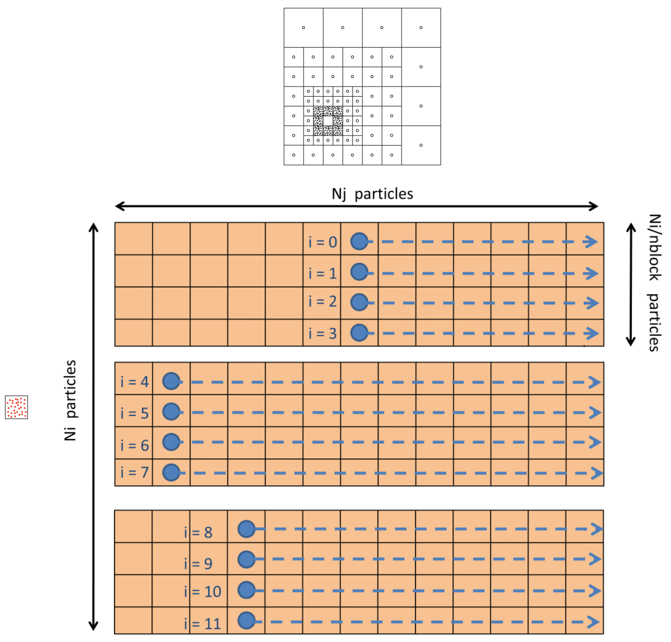


Figure 5: Implementation of the GPU code reported by Nyland *et al.*[24] and Belleman *et al.*[4]

There were two approaches in the previous GPU implementation of the tree algorithm. Figure 5 shows the first approach taken by Nyland *et al.* [24] and Belleman *et al.* [4]. Each orange block represents a thread-block described in Figure 4.

The pictures from Figure 2 are also inserted to clearly depict which part is being parallelized, and which part is being processed sequentially. For brevity we shall call the target particles "i-particles" and the source particles "j-particles". N_i is the number of i-particles in the terminal cell, and N_j is the number of j-particles that interact with it. In this GPU implementation, each thread calculates the force on a different i-particle– we call this the "i-parallel" approach. In this method, when the number of i-particles in a walk N_g (which shares the same interaction list as the j-particles) is smaller than the number of physical processor units, the rest of the processors remain idle. Since the optimal N_g for GPUs is typically of the order of several hundreds, the

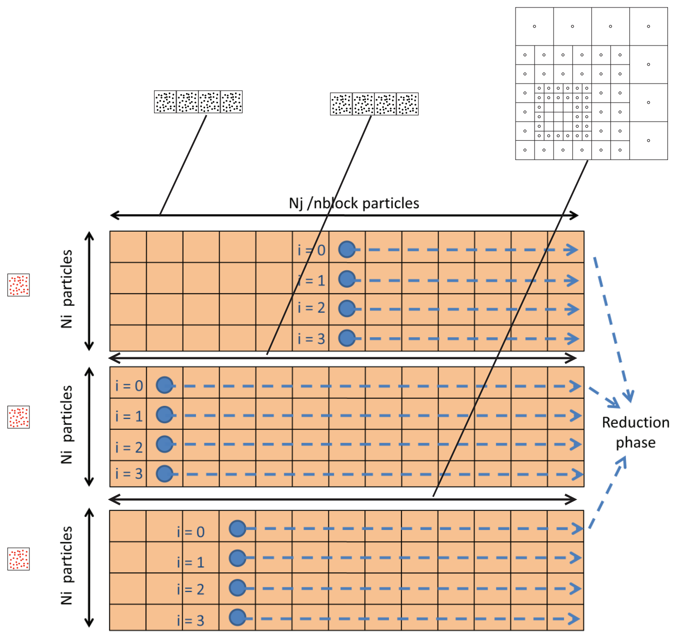


Figure 6: Implementation of GPU code reported by Hamada et al[10]

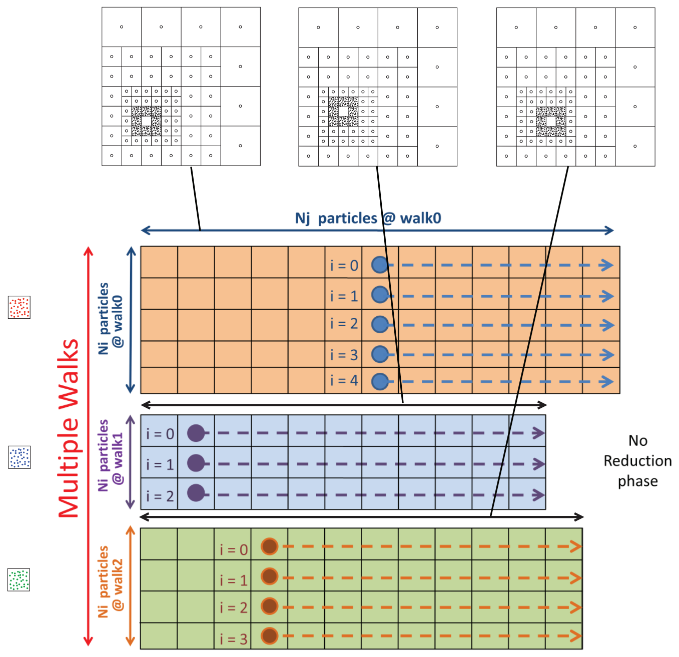


Figure 7: Implementation of the novel GPU code reported by this paper –the multiple walks method

efficiency of processor usage tends to be low in this implementation. Belleman *et al.* [4] use a large N_g –up to 32,768– to obscure this inefficiency.

An alternative approach proposed by Hamada *et al.* [10] is shown in Figure 6. In this case, the so-called "j-parallel" approach is used in addition to the i-parallel approach. The j-particles are divided into several groups, and the partial forces on the i-particles are calculated by different blocks. Multiple threads on each block evaluate different i-particles. The number of parallel i-particles is equal to the number of physical processors divided by the number of thread-blocks (N_{blocks}), which is usually smaller than N_g. Thus, the performance is better than that of the first approach. However, the results of the kernel benchmarks (see Figure 3) show only a marginal increase in performance over the Phantom-GRAPE. This is due to the overhead of the reduction operations required for the partial forces. Indeed, in this approach, partial forces on an i-particle are calculated by different blocks. Thus, they need to be summed. Unfortunately, the reduction operations on a GPU are slow due to the large overhead in the thread synchronization. There is a report available on the reduction operations that run on GPUs [26]. However, it is only valid for reductions of large arrays with thousands of elements. Therefore, the fastest method for the reduction of small arrays is to use the host CPU. Hence, the amount of communication between the GPU and the host CPU increases by N_{blocks}.

2.3 Details of proposed code

In this section, we explain the details of our novel approach and present its advantages over previous approaches. The remaining problems and their possible solutions will also be discussed.

Figure 7 shows our new effective implementation of the tree algorithm on GPUs. The main idea comes from viewing each thread block as one GRAPE, and mapping the previous tree-GRAPE algorithm accordingly. In our method, multiple walks are evaluated by different GPU blocks in parallel. In previous implementations, only a single walk was calculated at a time. In Figure 7, the operations of three blocks are depicted. The threads in each block evaluate different i-particles of the same walk. The sequence of the algorithm is as follows:

1. First, the data of the multiple walks are prepared on the host CPU, *i.e.* the lists of i-particles and j-particles of each walk are prepared. The number of walks in each calculation is determined by the size of the GPU global memory.

2. Multiple walks are then sent to the GPU.

3. Calculations are performed. The GPU is partitioned so that each GPU block evaluates a single walk.

4. The forces calculated by the different blocks in the GPU are received in a bundle.

5. The orbital integration and other caculations are performed on the host CPU.

6. The process is repeated.

A pseudo C++ code (emulator) for one GPU call from the host is as follows:

```
void force_nwalk(
    float4 xi[],
    float4 xj[],
    float4 accp[],
    int ioff[],
    int joff[],
    int nwalk)
{
  // block level parallelism
  for(int iw=0; iw<nwalk; iw++){
    int ni = ioff[iw+1] - ioff[iw];
    int nj = joff[iw+1] - joff[iw];
    // thread level parallelism
    for(int i=0; i<ni; i++){
      int ii = ioff[iw] + i;
      float x = xi[ii].x;
      float y = xi[ii].y;
      float z = xi[ii].z;
      float eps2 = xi[ii].w;
      float ax = 0;
      float ay = 0;
      float az = 0;
      float pot = 0;
      for(int j=0; j<nj; j++){
        int jj = joff[iw] + j;
        float dx = xj[jj].x - x;
        float dy = xj[jj].y - y;
        float dz = xj[jj].z - z;
        float r2 = eps2 + dx*dx + dy*dy + dz*dz;
        float r2inv = 1.f / r2;
        float rinv = xj[jj].w * sqrtf(r2inv);
        pot += rinv;
        float r3inv = rinv * r2inv;
        ax += dx * r3inv;
        ay += dy * r3inv;
        az += dz * r3inv;
      }
      accp[ii].x = ax;
      accp[ii].y = ay;
      accp[ii].z = az;
      accp[ii].w = -pot;
    }
  }
}
```

In all the previous approaches, the outer most loop (walk-loop) has been performed serially, and either the next i-loop or the inner most j-loop has been mapped to the multiple blocks of GPU. In our new approach, the outer most walk-loop is mapped to the multiple blocks, hence we can fully exploit the parallel nature of the GPU. Our approach has several advantages. First, the reduction of partial forces is no longer necessary since each walk is calculated in a block and no j-parallelization is used. This solves the problem in the previous approach proposed by Hamada *et al.* Second, multiple walks are sent to a GPU simultaneously, and the forces of the multiple walks are also retrieved from the GPU at the same time. This enables a more efficient communication between the CPU and GPU, since the number of DMA requests decrease, and the length of each DMA transfer becomes longer. Third, it makes parallelization in the host computation easier. Our approach involves the calculation of multiple walks at the same time. Therefore, each thread in the CPU can process a group of different walks in parallel.

Thus, we have successfully developed a novel tree algorithm that can be implemented efficiently on GPUs. Implementation results (see Figure 3) show that our new algorithm has a speed performance that is 2.4 times that of the previous GPU implementations and 3.2 times faster than that of optimized CPU codes.

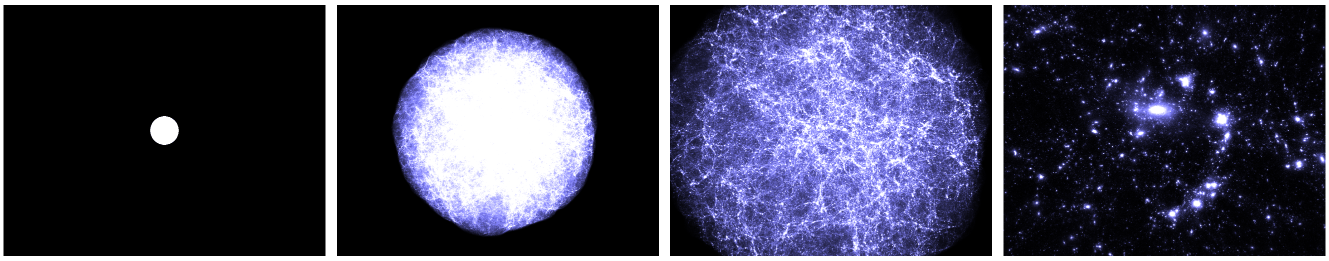


Figure 10: Snapshots of the simulation with 64M particles at z =18.8 (left), z = 4.5 (middle left), z = 2.6 (middle right), and z = 0 (right).

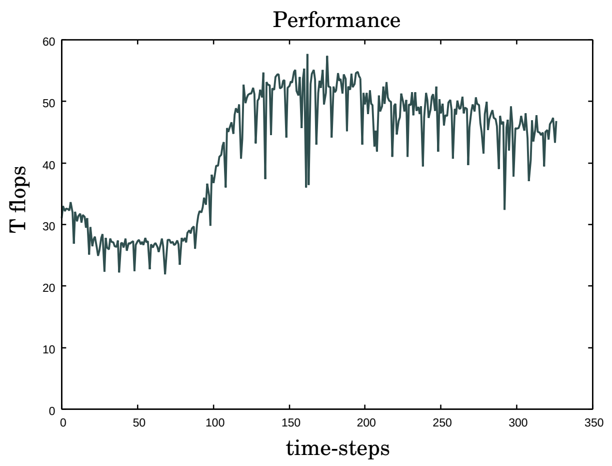


Figure 8: Measured performance of each timestep

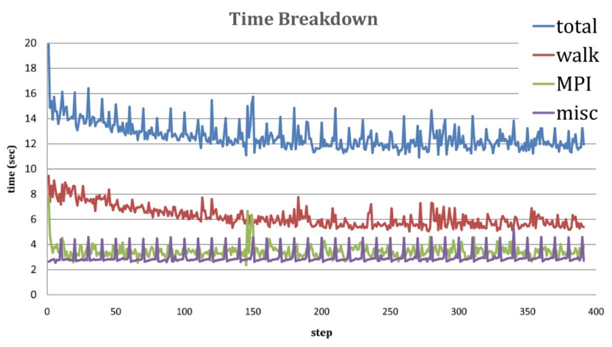


Figure 9: Breakdown of the calculation time of each timestep

3. GPU TREE APPLICATION

In this section, we present a cosmological N-body simulation that uses our novel algorithm, on a GPU cluster. We used CentOS for the machines that used x86_64 linux (version 5.1) as the operating system, and OpenMPI (version 1.3.1) for the implementation of the message passing interface. The CPU code was written in C++ and compiled with the GNU compiler collection (version 4.1.2). The GPU code was also written in C++ and complied with the NVIDIA CUDA compilation tool (version 2.0), respectively.

3.1 Performance validation

We performed a cosmological N-body simulation of a sphere of radius 136.28 Mpc (mega parsec) with 1,608,044,129 particles for 327 timesteps. We assigned initial positions and velocities to particles of a spherical region selected from a discrete realization of a density contrast field based on a standard cold dark matter scenario. A particle represents 8.20×10^4 solar masses. We performed the simulation from $z = 21.1$, where z is red-shifted, to $z = 18.5$. Figure 10 shows images of the simulation with 64M particles.

The change in the performance with the evolution of the system is shown in Figure 8, along with its breakdown in FIgure 9. Here, we use an operation count of 38 operations per interaction. The floating point operation count used by previous Gordon Bell winners for hierarchical N-body simulations has always been 38 both on general and special purpose computers [30, 28, 14, 13]. The 38 comes from the method of calculating gravity using the Newton-Raphson method with an initial guess by Chebyshev interpolation [12]. It is close to 36 when a division and a square root are counted as ten floating point operations, respectively. Although this operation count strictly holds for only CPUs and GPUs can perform division or square root operations with lower cost than those for 10 multiplications or additions, we adapt this operation count once again in our calculations for the sake of comparability as an "equivalent CPU Flops" rather than a "GPU Flops". Note that the operation count of the other GPU implementations we compared with (Nyland *et al.* [24], Belleman *et al.* [4]) were corrected to 38 when we compared the Flops with theirs in this paper. Due to the evolution of a large-scale structure in the universe, the number of interactions increased, and as a result, the performance also increased. After 150 steps the performance reached a constant state.

In total, the number of particle-particle interactions was 6.19×10^{15}. This implies that the average length of the interaction list was 11,765. The whole simulation took 5,604

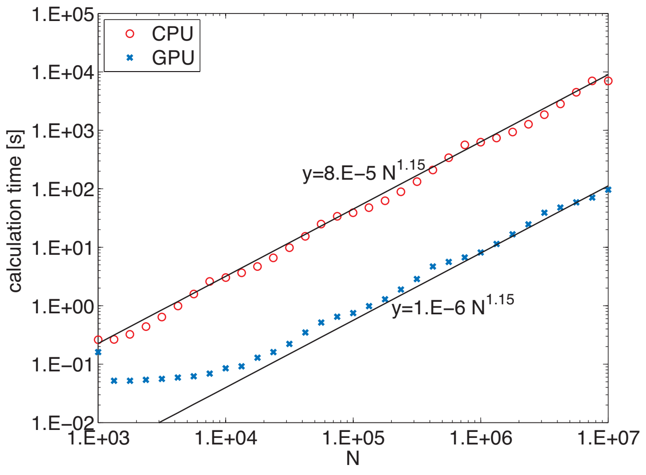


Figure 11: Cputime of the FMM on a serial CPU and GPU ($p = 10$). Dotted lines are functional fits as $y = f(N)$, where y is the time, and N is the number of particles.

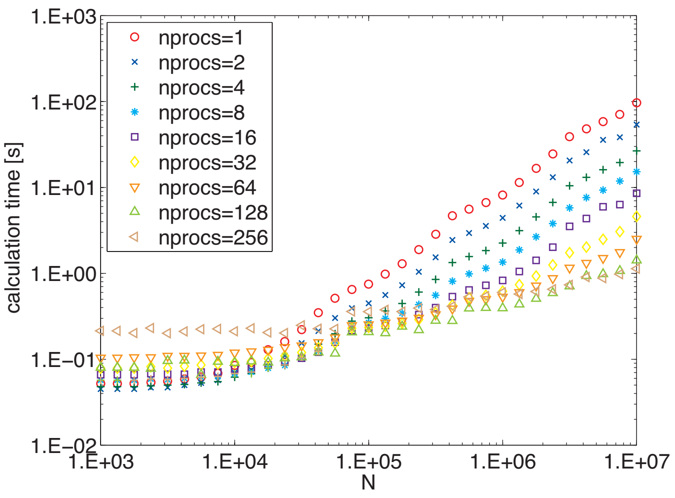


Figure 12: Cputime of the FMM on parallel GPUs

s, which included I/O operations, and the resulting average computing speed was 42.15 TFlops.

It should be noted that our modified tree algorithm performs a larger number of operations than the conventional tree algorithm that runs on a general purpose microprocessor. In order to rectify this, we estimated the operation count of the original tree algorithm for the same simulation based on the data of a simulation image (at $z = 19.66$) and the same accuracy parameter. The number of interaction counts was then found to be 4.12×10^{15}. The effective sustained speed is hence 28.1 TFlops and the resulting price/performance ratio is thus equal to a mere \$8.15/GFlops.

Finally, it should also be noted that our modified tree algorithm is more accurate than the original tree algorithm for the same accuracy parameter, as shown in [3][14].

4. GPU FMM CODE

4.1 FMM on GPUs

The FMM by Greengard and Rokhlin [8] also uses the tree structure but calculates cell-cell interactions, instead of particle-cell interactions. In order to execute the FMM efficiently on the GPU, the following modifications were made to the FMM. First, the complex spherical harmonics were transformed to real basis functions, in order to avoid complex arithmetic on the GPU. Second, in order to minimize the memory usage per interaction, all of the translation matrices were generated on-the-fly. This extra work load is not included in the Flops count. Third, the box structure and interaction list of the FMM are restructured and renumbered to match the number of threads per block, so that no threads remain idle, while the data can be continuously transferred to the shared memory in a coalesced manner. Finally, the same multipole walks technique described in section 2.3 is implemented for all stages of the FMM algorithm.

The FMM is composed of the particle interaction (P2P), multipole expansion from particle (P2M), multipole to multipole translation (M2M), multipole to local translation (M2L), local to local translation (L2L), and local expansion to particle (L2P). We will describe the details of the GPU implementation of each component by explaining what variables are passed to the GPU, which of them are stored in the shared memory, and how they are used in the GPU kernel.

For the P2M calculation, the coordinates and particle strengths are stored in the shared memory. The expansion coefficients are also stored in the shared memory before they are copied to the global memory in a coalesced manner. Each thread block handles one FMM box, and each thread handles one expansion coefficient. Using the coordinates in the shared memory and the center of expansion in the register, the recurrence relation is calculated, and the resulting expansion coefficients are then copied to the global memory.

For the M2M, M2L, and L2L, the expansion coefficients, the order of expansion, box size, and interaction list of FMM boxes are passed on to the GPU. The shared memory in this case is occupied by the expansion coefficients. Again, each thread block handles one FMM box while each thread handles one target expansion coefficient. The calculation loops through the interacting boxes, and a unique index describing the relative position of the target box and source box is calculated. This index is used to extract a precalculated Wigner D matrix from the global memory. The expansion coefficients are rotated using this matrix. Then, the recurrence relation for the translation is calculated without any global memory access. Finally, the expansion coefficients are rotated back.

For the L2P, the coordinates and expansion coefficients are stored in the shared memory. For the calculation of Eq. ((1) the velocity $\mathbf{u}_i$ is also stored, and for Eq. (3) the change rate of vortex strength $D\boldsymbol{\alpha}_i/Dt$ is also stored in the shared memory before it is written to the global memory in a coalesced manner. Each thread block handles a portion of the particles in the FMM box, while each thread handles one particle. The calculation loops through all expansion coefficients and calculates Eq. (1) or (3) without any global memory access. Finally, the results are copied to the global memory in a coalesced manner.

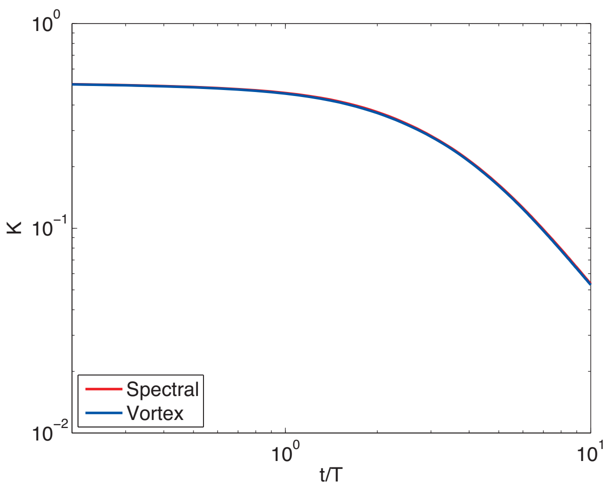


Figure 13: Decay of kinetic energy

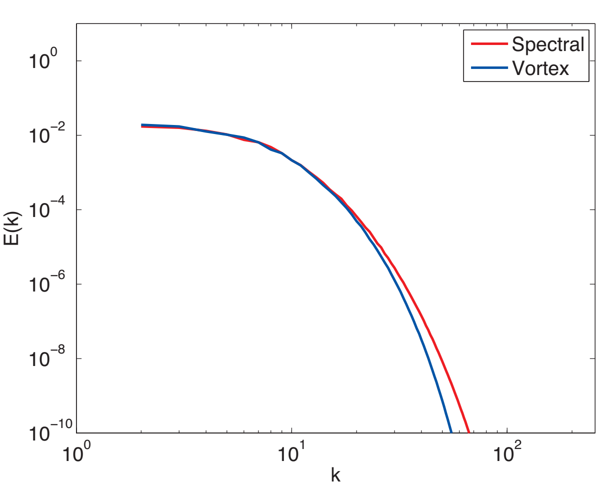


Figure 14: Energy spectra at $t/T = 10$

4.2 Vortex method

The vortex method [15] is a particle based method for fluid dynamics simulations. The particle based discretization allows the continuum physics to be solved as a N-body problem. Therefore, the hierarchical N-body methods that extract the full potential of GPUs can be used for the simulation of turbulence. Unlike other particle based fluid dynamics solvers *e.g.* SPH[7], the vortex method is especially well suitable for solving turbulence, because the vortex interactions seen in turbulent flows are exactly what it calculates using the interacting vortex particles.

Since the vortex method is neither a standard method for simulating turbulence or a standard application for FMMs, we will give a brief explanation of the method itself. In the vortex method, the Navier-Stokes equation is solved in the velocity-vorticity form, and discretized with vortex particles. The velocity is calculated by

$$\mathbf{u}_i = \sum_{j=1}^{N} \boldsymbol{\alpha}_j g_\sigma \times \nabla G \tag{1}$$

where α is the strength of vortex particles. $G = 1/4\pi r_{ij}$ is the Green's function for the Laplace equation and

$$g_\sigma = \operatorname{erf}\left(\sqrt{\frac{r_{ij}^2}{2\sigma_j^2}}\right) - \sqrt{\frac{4}{\pi}}\sqrt{\frac{r_{ij}^2}{2\sigma_j^2}} \exp\left(-\frac{r_{ij}^2}{2\sigma_j^2}\right), \tag{2}$$

is the cutoff function, where r is the distance between the interacting particles, and σ is the standard deviation of the Gaussian function. The vorticity equation is solved in a fractional step manner by calculating the stretching

$$\frac{D\boldsymbol{\alpha}_i}{Dt} = \sum_{j=1}^{n} \boldsymbol{\alpha}_j \nabla(g_\sigma \times \nabla G) \cdot \boldsymbol{\alpha}_i. \tag{3}$$

and the diffusion

$$\sigma^2 = 2\nu t \tag{4}$$

separately. We perform a radial basis function interpolation for reinitialized Gaussian distributions [31] every 100 steps to ensure the convergence of the diffusion calculation. The second order Adams-Bashforth method is used for all time integration calculations. Eqs. (1) and (3) involve far field interactions and can be solved using hierarchical N-body algorithms, such as the FMM.

4.3 Performance validation

In order to evaluate the performance of the present GPU calculation, we first measured the performance of a serial GPU. The particles are randomly positioned in a $[-\pi, \pi]^3$ domain, and given a random vortex strength between 0 and $1/N$. The core radius is set to $\sigma = 2\pi N^{-1/3}$, which results in an average overlap of $\sigma/\Delta x = 1$.

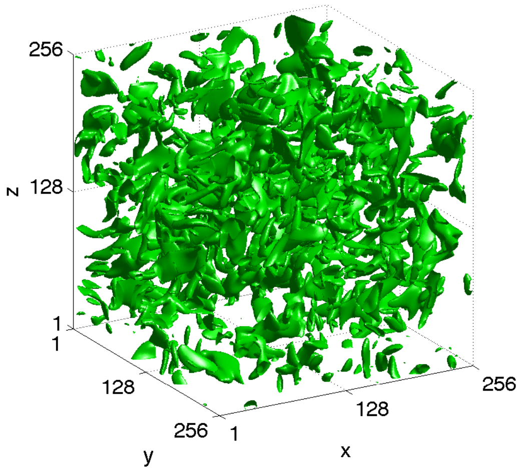
We now present the results of the velocity calculation in Eq. (1) using the FMM on a serial CPU and GPU. The order of multipole expansions is set to $p = 10$ unless otherwise noted. Figure 11 shows the calculation time of the FMM on a serial CPU and GPU. Our FMM does not scale exactly as $\mathcal{O}(N)$, but rather shows a scaling close to $\mathcal{O}(N^{1.15})$. This is observed from the results of both the CPU and GPU. Judging from the asymptotic constants of the two lines, the FMM on the GPU is approximately 80 times faster than the FMM on the CPU.

The optimum level of the oct-tree differs between the CPU and GPU calculations. The number of particles per terminal cell N_g ranges from 100 to 800 for the present FMM on GPUs. The optimal N_g for the FMM on the CPU ranges from 20 to 160. Thus, the optimum level of the FMM on GPUs is lower then the one on CPUs for small N, but seems to shift more frequently as N becomes larger. This is also reflected in Figure 11, where the calculation time of the GPU seems to have a better scaling than the dotted line of $\mathcal{O}(N^{1.15})$

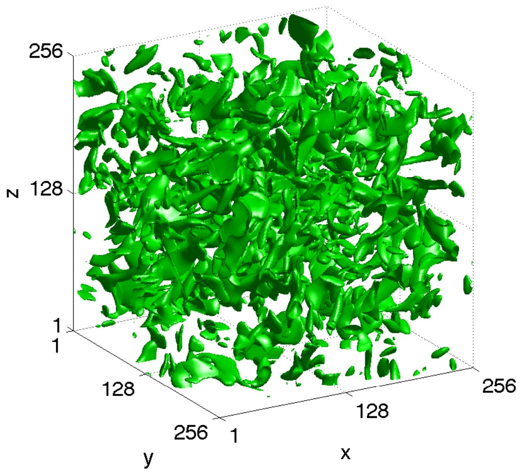
Figure 12 shows the calculation time of the parallel FMM using different number of GPUs. The low parallel efficiency for small N is a direct consequence of the low performance of GPUs for small N. Especially, when the FMM is parallelized the number of target particles handled by each GPU becomes $N/nprocs$, and high parallel efficiency can only be achieved when $N/nprocs$ is large. It is also seen in Figure 12 that the calculation time for $N < 10^4$ increases for

Table 1: Number of floating point operations for the vortex method calculation

Description	Equation	Value
Total number of particles	N	16,777,216
Number of level of cell subdivisions	N_{level}	5
Average number of particles in a cell	$N_{cell} = N/(8^{N_{level}})$	512
Average number of source particles per target particle	$N_j = 27N_{cell}$	13,824
Operations per pair wise velocity (Eq. (1))+ stretching (Eq. (3)) interaction	K	133
Operations per time step	KNN_j	3.08×10^{13}



(a) *Spectral method*



(b) *Vortex method*

Figure 15: Isosurface of the second invariant (II) of the velocity derivative tensor

nprocs $>= 16$. This is because the time spent on communication becomes large compared to the calculation time. The communication time increases monotonically as *nprocs* increases, and dominates the calculation at *nprocs* $= 256$.

5. GPU FMM APPLICATION

5.1 Calculation Conditions

The flow field of interest is a decaying isotropic turbulence with an initial Reynolds number of $Re_\lambda \approx 100$. The calculation domain is $[-\pi, \pi]^3$ and has periodic boundary conditions in all directions. Details of the periodic FMM are shown in our previous publication [31]. The number of calculation points was $N = 256^3 = 16,777,216$ for both the vortex method and spectral method. The order of multipole expansion was set to $p = 10$, and the number of periodic images was $2^5 \times 2^5 \times 2^5$ for the present calculations. We used a total of 256 GPUs for the calculation of the isotropic turbulence.

The spectral Galerkin method with primitive variable formulation [25] is used in the present study as reference. A pseudo-spectral method was used to compute the convolution sums, and the aliasing error was removed by the 3/2-rule. The time integration was performed using the fourth order Runge-Kutta method for all terms. No forcing was applied to the calculation, since it would be difficult to do so with vortex methods. The spectral method was calculated on the same number of processors without using the GPUs.

The initial condition was generated in Fourier space as a solenoidal isotropic velocity field with random phases and a prescribed energy spectrum, and transformed to physical space. The spectral method calculation used this initial condition directly. The core radius of the vortex elements were set to $2\pi/N^{1/3}$ so that the overlap ratio was 1.

5.2 Calculation Results

Figure 13 shows the decay of kinetic energy, which is defined as

$$K = \frac{1}{2}\sum_{i=1}^{N} u_i^2 + v_i^2 + w_i^2. \quad (5)$$

Spectral is the spectral method and Vortex is the vortex method calculation, respectively. The time is normalized by the eddy turnover time T. The integral scale and eddy

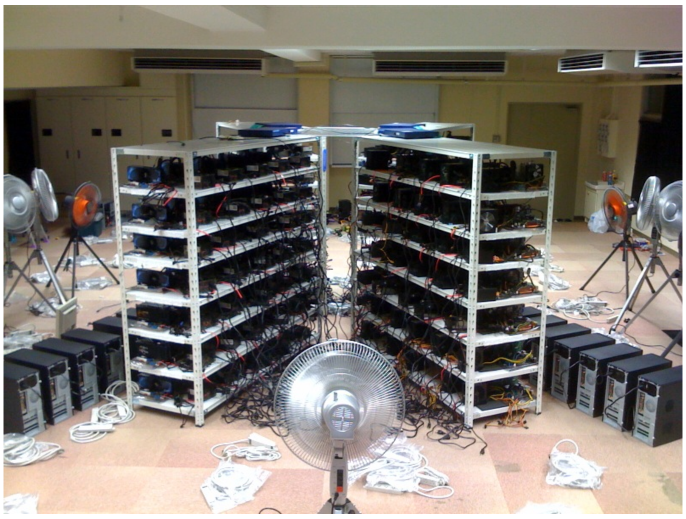


Figure 16: Photographs of the GPU cluster (left) and the GPU (right)

turnover time have the following relation.

$$L = \frac{\pi}{2u'^2} \int k^{-1} E(k) dk \qquad (6)$$

$$T = L/u'. \qquad (7)$$

where $u' = \frac{2}{3}K$. The homogeneous isotropic turbulence does not have any production of turbulence, and thus the kinetic energy decays monotonically with time. This decay rate is known to show a self-similar behavior at the finial period of decay. This is confirmed by the straight drop of K that appears at the end of this log-log plot. The results of the two methods agree perfectly until $t/T = 10$, where the kinetic energy drops an order of magnitude from the initial value.

Figure 14 shows the energy spectrum at $t/T = 10$. k is the wave number, and E(k) is the kinetic energy contained in the wave number k. At this Reynolds number it is difficult to observe an inertial subrange of $k^{-5/3}$, nor a k^4 behavior at low wave numbers. The results of the two methods are in good agreement, except for the fact that the vortex method slightly underestimates the energy at higher wave numbers.

The isosurface of the second invariant of the velocity derivative tensor $II = u_{i,j}u_{j,i}$ at time $t/T = 10$ is shown in Figure 15. Figure 15(a) is the isosurface of the spectral method, Figure 15(b) shows the isosurface from the vortex method calculation. Although, the larger structures match between the two methods, the smaller structures behave differently. The difference in the small structures can also be observed in Figure 14, where the kinetic energy at higher wave numbers do not match. However, since the energy spectrum of the vortex method matches with the reference calculation up to the dissipation scale ($k \approx 30$), it is sufficient for most turbulent flow applications.

The total floating point operation count of our entire simulation is estimated to be around 3.08×10^{16} for 1000 time steps. The details of the estimation are shown in Table 1. Only the direct summation part of the FMM is considered in this estimation. The operation count of the kernel $K = 133$ was obtained by hand-counting the operators in the CUDA code. The end-to-end measurement of the wall-clock time of our vortex method simulation was $1,520$ seconds. Thus, the average performance of our simulation was approximately 20.2 TFlops.

Table 2: Price of the GPU cluster

Elements	Quantity	Price (JPY)	Price ($)
GPUs	256	12,160,000	$ 118,345
Host PCs	128	10,716,032	$ 104,292
Network switch	4	644,800	$ 6,275
Total		**23,520,832**	**$ 228,912**

Table 3: Comparison with SC06 GB finalist

	'06 finalist	This work	Ratio
number of particles	2,159,038	1,608,044,129	745
price	$ 2,384	$ 228,912	96
GFlops (uncorrected)	36.31	42,150	1,161
GFlops (corrected)	15.39	28,100	1,826
$/GFlops	158	8.5	19

6. COST PERFORMANCE

We used 128 host PCs with 256 GPUs for the simulation. Each PC had a CoreTM 2-Quad 2.4-GHz Q6600 processor on an X38 chipset motherboard with a single Gigabit Ethernet (GbE) port. A photograph of the GPU cluster is shown in Figure 16.

The costs of the constituent elements of our GPU cluster are summarized in Table 2. The prices of the GPU, the host PC, and the three GbE network switches were 47,500 JPY, 83,719 JPY, and 161,200 JPY, respectively. The total price of 128 GPUs, 128 PCs, and 4 switches was 23,520,832 JPY, which is equivalent to $228,912 (102.75 JPY = $1 on April 4th 2008). All prices are inclusive of a sales tax of 5%. The sustained performance of 42.15 TFlops. The corrected performance of the gravitational simulation in section 3 is 28.1TFlops, which results in a cost performance of 124 MFlops/$. The comparison with previous Gordon Bell win-

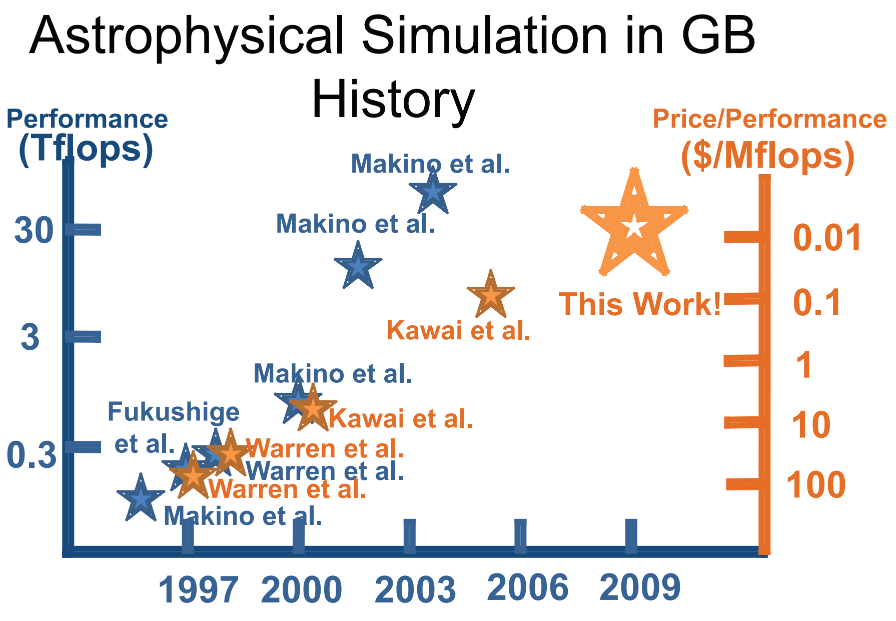


Figure 17: Comparison with previous Gordon Bell winners

ners is shown in Figure 17.

Compared to the Gordon Bell finalist of 2006 [13], the number of particles used in the present gravitational simulation is 768 times larger. Furthermore, the performance is 1870 times higher and the price/performance is 18.59 times better as shown in Table 3. The correction factor of our calculation is larger than that of theirs. This is due to the fact that our calculation can use a relatively small N_g, because the data transfer was faster (We used a PCI express gen2 x16 as to where they used a PCI-X). Another reason for the different correction factor is the use of SSE and multicore tuning in our host routines.

7. CONCLUSION

A hierarchical N-body simulation has been performed on a cluster of 256 graphics processing units (GPUs). Using this fast N-body solver, A gravitational N-body simulation using 1,608,044,129 particles was performed as a standard benchmark. In addition, the vortex particle simulation of homogeneous isotropic turbulence using 16,777,216 particles was performed. The treecode was used for the gravitational simulation, while the FMM was used for the vortex particle simulation.

With our approach, both the tree algorithm and FMM show a significant performance gain when executed on GPUs as compared to the performances of the hierarchical algorithms running on CPUs. The previous implementations of the hierarchical algorithms made it difficult to achieve an effective GPU performance, especially compared to their performances on conventional PC clusters. Using our novel approach, however, a GPU cluster can outperform a PC cluster from the viewpoints of cost/performance, power/performance, and physical dimensions/performance. The gravitational N-body simulation showed a sustained performance of 42.15 TFlops, while the vortex particle simulation showed a sustained performance of 20.2 TFlops. The overall cost of the hardware was 228,912 dollars. The maximum corrected performance is 28.1TFlops of the gravitational simulation, which results in a cost performance of **124 MFlops/$**. The correction is performed by counting the Flops based on the most efficient CPU algorithm. Any extra Flops that arise from the GPU implementation and parameter differences are not counted.

The present acceleration technique enabled the calculation of a homogeneous isotropic turbulence using a moderate number of vortex elements in a very short time. The kinetic energy decay and energy spectrum of the well resolved vortex method calculation agreed quantitatively with that of the reference calculation using a spectral method. This opens an interesting possibility for fluid dynamics simulations to extract a large amount of computational power from future many-core architectures at a minimum price/performance by using hierarchical N-body methods.

8. ACKNOWLEDGMENTS

This study was supported by a budget called “Technology Integration Using Real-time Information Processing” from the Special Coordination Funds for Promoting Science and Technology, Ministry of Education, Culture, Sports, Science and Technology, Japan (MEXT). This study was also, partially supported by the Core Research for Evolution Science and Technology (CREST) of the Japan Science and Technology Corporation (JST).

9. ADDITIONAL AUTHORS

Additional authors: Kiyoshi Oguri (Department of Computer and Information Sciences, Nagasaki University, email: `oguri@cis.nagasaki-u.ac.jp`).

10. REFERENCES

[1] E. Athanassoula, A. Bosma, J.-C. Lambert, and J. Makino. Performance and accuracy of a grape-3

system for collisionless n-body simulations. *Monthly Notices of the Royal Astronomical Society*, 293:369–380, 1998.

[2] J. Barnes and P. Hut. O(nlogn) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] J. E. Barnes. A modified tree code: Don't laugh; it runs. *Jouranl of Computational Physics*, 87:161–170, 1990.

[4] R. G. Belleman, J. Bedorf, and S. F. Portegies Zwart. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, 13:103–112, 2008.

[5] T. Fukushige and J. Makino. N-body simulation of galaxy formation on grape-4 special-purpose computer. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, pages 1–9, 1996.

[6] E. Gaburov, S. Harfst, and S. Portegies Zwart. Sapporo: A way to turn your graphics cards into a grape-6. *New Astronomy*, 14:630–637, 2009.

[7] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: Theory and application to non-sperical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[8] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

[9] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227:8290–8313, 2008.

[10] T. Hamada and T. Iitaka. The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units. *arXiv:astro-ph/0703100v1*, 2007.

[11] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji. A novel multiple-walk parallel algorithm for the barnes-hut treecode on gpus- towards cost effective, high performence n-body simulation. *Computer Science - Research and Development*, Special Issue Paper:1–11, 2009.

[12] A. H. Karp. Speeding up n-body calculations on machines without hardware square root. *Scientific Programming*, 1:133–141, 1992.

[13] A. Kawai and T. Fukushige. $158/gflops astrophysical n-body simulation with reconfigurable add-in card and hierarchical tree algorithm. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 1–8, 2006.

[14] A. Kawai, T. Fukushige, and J. Makino. $ 7.0/mflops astrophysical n-body simulation with treecode on grape-5. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 1–6, 1999.

[15] P. D. Koumoutsakos and G.-H. Cottet. *Vortex Methods -Theory and Practice-*. Cambridge University Press, 2000.

[16] J. Makino. Treecode with a special-purpose processor. *Publication of the Astronomical Society of Japan*, 43:621–638, 1991.

[17] J. Makino. A fast parallel treecode with grape. *Publications of the Astronomical Society of Japan*, 56:521–531, 2004.

[18] J. Makino, T. Fukushige, and M. Koga. A 1.349 tflops simulation of black holes in a galactic center on grape-6. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pages 1–18, 2000.

[19] J. Makino, E. Kokubo, and T. Fukushige. Performance evaluation and tuning of grape-6 –towards 40 "real" tflops. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 1–11, 2003.

[20] J. Makino, E. Kokubo, T. Fukushige, and H. Daisaka. A 29.5 tflops simulation of planetesimals in uranus-neptune region on grape-6. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, 2002.

[21] J. Makino and M. Taiji. Astrophysical n-body simulations on grape-4 special-purpose computer. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 1–10, 1995.

[22] T. Narumi, Y. Ohno, N. Okimoto, T. Koishi, A. Suenaga, N. Futatsugi, R. Yanai, R. Himeno, S. Fujikawa, M. Taiji, and M. Ikei. A 55 tflops simulation of amyloid-forming peptides from yeast prion sup35 with the special-purpose computer system mdgrape-3. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 1–16, Tampa, Florida, 2006.

[23] K. Nitadori, J. Makino, and P. Hut. Performance tuning of n-body codes on modern microprocessors: I. direct integration with a hermite scheme on x86_64 architecture. *New Astronomy*, 12:169–181, 2006.

[24] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with cuda. *GPU Gems*, 3:677–695, 2007.

[25] R. S. Rogallo. Numerical experiments in homogeneous turbulence. NASA Technical Memorandum 81315, NASA Ames Research Center, 1981.

[26] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 1–11, 2007.

[27] M. J. Stock and A. Gharakhani. Toward efficient gpu-accelerated n-body simulations. *AIAA Paper*, 2008-608, 46th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, Jan. 7 - 10:1–13, 2008.

[28] M. S. Warren, T. C. Germann, P. S. Lomdahl, D. M. Beazley, and J. K. Salmon. Avalon: An alpha/linux cluster achieves 10 gflops for $150k. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–10, 1998.

[29] M. S. Warren and J. K. Salmon. Astrophysical n-body simulation using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 570–576, 1992.

[30] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, and T. Sterling. Pentium pro inside: I. a treecode at 430 gigaflops on asci red, ii. price/performance of $ 50/mflop on loki and hyglac. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–16, 1997.

[31] R. Yokota, T. K. Sheel, and S. Obi. Calculation of isotropic turbulence using a pure lagrangian vortex method. *Journal of Computational Physics*, 226:1589–1606, 2007.