T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

Title	Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence
Authors	Rio Yokota, Tetsu Narumi, Ryuji Sakamaki, Shun Kameoka, Shinnosuke Obi, Kenji Yasuoka
Citation	Computer Physics Communications, Vol. 180, No. 11, pp. 2066–2078
Pub. date	2009, 6
DOI	http://dx.doi.org/10.1016/j.cpc.2009.06.009
Note	このファイルは著者（最終）版です。 This file is author (final) version.
Creative Commons	See next page.

Powered by T2R2 (Science Tokyo Research Repository)

License



Creative Commons : CC BY-NC-ND

Fast Multipole Methods on a Cluster of GPUs for the Meshless Simulation of Turbulence

R. Yokota[a,*], T. Narumi[a], R. Sakamaki[a], S. Kameoka[a], S. Obi[a], K. Yasuoka[a]

[a] *Department of Mechanical Engineering, Keio University, 3-14-1 Hiyoshi Yokohama, Japan*

Abstract

Recent advances in the parallelizability of fast N-body algorithms, and the programmability of graphics processing units (GPUs) have opened a new path for particle based simulations. For the simulation of turbulence, vortex methods can now be considered as an interesting alternative to finite difference and spectral methods. The present study focuses on the efficient implementation of the fast multipole method and pseudo-particle method on a cluster of NVIDIA GeForce 8800 GT GPUs, and applies this to a vortex method calculation of homogeneous isotropic turbulence. The results of the present vortex method agree quantitatively with that of the reference calculation using a spectral method. We achieved a maximum speed of 7.48 TFlops using 64 GPUs, and the cost performance was near \$9.4/GFlops. The calculation of the present vortex method on 64 GPUs took $4,120$s, while the spectral method on 32 CPUs took $4,910$s.

Key words: Fast multipole method, Pseudo-particle method, Graphics processing unit, Particle method

1. Introduction

Particle-based simulations are a natural method for solving discrete systems such as in astrophysics, and molecular dynamics (MD). They also provide an interesting alternative to grid-based methods for solving continuum systems, as seen in smooth particle hydrodynamics (SPH) and vortex meth-

*Corresponding author.
Email addresses: rio.yokota@bristol.ac.uk (R. Yokota)

ods (VM). While much of the numerical and implementation issues have been successfully addressed,[1] the computational cost of the N-body interaction still remains a problem.

The use of hierarchical algorithms such as the treecode [2] and fast multipole method (FMM) [3] are a necessary but not sufficient condition for executing particle-based simulations in a reasonable amount of time. Parallelization of hierarchical N-body algorithms is also essential and there have been many attempts to parallelize the treecode [4] and FMM [5, 6, 7]. Another way to extract the intrinsic parallelism of N-body problems is to design a fully pipelined, hard-wired processor dedicated to the calculation of gravitational interaction (GRAPE) [8] or arbitrary central force (MDGRAPE) [9, 10, 11, 12]. These special-purpose computers can also be parallelized [13] and used simultaneously with fast algorithms [14, 15].

A more recent trend in the hardware industry is the development of programmable graphics processing units (GPU). The 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 for both gravitational [16, 17] and MD simulations [18]. Furthermore, Schive *et al.* [19] used 32 GPUs to accelerate the direct calculation of gravitational interactions, and Stone *et al.* [20] used 6 GPUs for the calculation of the particle mesh Ewald (PME) method in their MD simulation. Stock & Gharakhani [21] implemented the treecode on the GPU to accelerate their vortex method calculation. Their treecode was approximately 17 times faster, whereas their direct calculation was 127 times faster on the GPU for $N = 500,000$ particles. Similarly, Gumerov & Duraiswami [22] calculated the Coulomb interaction using the FMM on GPUs, and achieved a 72-fold speed-up for the FMM, while their direct calculation was 855 times faster on the GPU for $N = 1,048,576$ particles. The difference between the absolute acceleration ratio of Stock & Gharakhani and Gumerov & Duraiswami is mainly caused by the difference in the speed of the CPU calculation. The relative acceleration ratio between the fast algorithm and direct calculation was $17/127 \approx 0.134$ for Stock & Gharakhani and $72/855 \approx 0.084$ for Gumerov & Duraiswami. The difference between the relative acceleration ratio is probably due to the difference in the data-parallelism of the treecode and FMM. What remains a challenge is the fact that there still remains a large gap between the acceleration ratio of the fast algorithms and the direct calculation.

The large gap between the acceleration ratio of the fast algorithms and the direct calculation is due to the slightly low data-parallelism of the mul-

tipole moments in the hierarchical box structure. In our present study we investigate the possibility of using pseudo-particles instead of multipole moments. The use of pseudo-particle methods (PPM) have been proposed by Anderson [23] and improved by Makino [24]. These methods are slower than the standard FMM on ordinary CPUs because they require so many pseudo-particles to achieve the same accuracy as the standard FMM. However, on GPUs, their data-parallelism may become an advantage.

For vortex methods, fast N-body algorithms on data-parallel processors may become an interesting alternative to grid based fast Poisson solvers. The equations that govern the pairwise interaction of vortex elements are more complex than those of the gravitational and molecular dynamics calculations. This complicates the implementation on special-purpose computers (MDGRAPE) [25, 26], but allows a more efficient implementation on GPUs because it is easier to hide the memory latency.

We apply the fast multipole method on a cluster of GPUs to the calculation of a homogeneous isotropic turbulence using vortex methods. The homogeneous isotropic turbulence has been a traditional benchmark for evaluating the ability of supercomputers.[27, 28] It has also served as the test case for which the vortex methods were directly compared with a spectral method under the same calculation conditions, and showed quantitative agreement.[29, 30] We wish to reduce the calculation cost of vortex methods, by making use of the inherent parallelism of the N-body calculation, and also the new GPU technology.

2. Vortex Method

The present vortex method solves the following set of equations.

$$\nabla^2 \mathbf{u} = -\nabla \times \boldsymbol{\omega} \tag{1}$$

$$\frac{D\boldsymbol{\omega}}{Dt} = \boldsymbol{\omega} \cdot \nabla \mathbf{u} + \nu \nabla^2 \boldsymbol{\omega} \tag{2}$$

where $\mathbf{u}$ is the velocity vector, $\boldsymbol{\omega}$ is the vorticity vector, and ν is the kinematic viscosity. When the velocity poisson equation (1) is formulated as an integral equation using Green's functions, it yields the generalized Biot-Savart equation. In the present vortex method the vorticity equation (2) is solved in a fractional step manner by solving the first and second term on the right hand side, and the left hand side separately.

First, the vorticity field is discretized by using a superposition of Gaussian distributions.

$$\boldsymbol{\omega}_i = \sum_{j=1}^{N} \boldsymbol{\alpha}_j \zeta_\sigma, \tag{3}$$

where $\boldsymbol{\alpha}$ is the vortex strength, and

$$\zeta_\sigma = \frac{1}{(2\pi\sigma_j^2)^{3/2}} \exp\left(-\frac{r_{ij}^2}{2\sigma_j^2}\right) \tag{4}$$

is the Gaussian smoothing function. σ is often referred to as the core radius, and represents the physical length scale of vortex elements. r_{ij} is the distance between point i and point j. In N-body calculations, i is referred to as the target, and j is referred to as the source. The generalized Biot-Savart equation can be written as

$$\mathbf{u}_i = \sum_{j=1}^{N} \boldsymbol{\alpha}_j g_\sigma \times \nabla G \tag{5}$$

where G is the Green's function for the Laplace equation and

$$g_\sigma = \operatorname{erf}\left(\sqrt{\frac{r_{ij}^2}{2\sigma_j^2}}\right) - \sqrt{\frac{4}{\pi}}\sqrt{\frac{r_{ij}^2}{2\sigma_j^2}} \exp\left(-\frac{r_{ij}^2}{2\sigma_j^2}\right), \tag{6}$$

is the cutoff function corresponding to the Gaussian distribution in Eq. (4).

The left hand side (convection term) of Eq. (2) is accounted for by advancing the coordinates according to the local velocity.

$$\frac{D\mathbf{x}_i}{Dt} = \mathbf{u}_i \tag{7}$$

The first term on the right hand side (stretching term) of Eq. (2)

$$\frac{D\boldsymbol{\omega}}{Dt} = \boldsymbol{\omega} \cdot \nabla \mathbf{u} \tag{8}$$

is solved by substituting Eq. (3) and Eq. (5), which yields

$$\frac{D\boldsymbol{\alpha}_i}{Dt} = \sum_{j=1}^{n} \boldsymbol{\alpha}_j \nabla(g_\sigma \times \nabla G) \cdot \boldsymbol{\alpha}_i. \tag{9}$$

In the present calculations, we use the core spreading method (CSM) to solve the second term on the right hand side (diffusion term)

$$\frac{D\boldsymbol{\omega}}{Dt} = \nu\nabla^2\boldsymbol{\omega}. \tag{10}$$

by changing the variance of the Gaussian distribution

$$\sigma^2 = 2\nu t \tag{11}$$

to account for the diffusion. We also perform radial basis function interpolation for smaller Gaussian distributions[30] to ensure the convergence of the CSM.

In summary, the present vortex method solves Eqs. (5), (7), (9), and (11). Among these, Eqs. (5) and (9) involve far field interactions and must be solved using fast N-body algorithms. The second order Adams-Bashforth method is used for all time integration calculations.

There have been numerous attempts to accelerate vortex methods using fast N-body algorithms and specialized hardware. Winckelmans *et al.* [31] used the treecode to accelerate the vortex method calculation and also the boundary element method calculation. Marzouk & Ghoniem [32] used K-means clustering for load balancing their parallel treecode, and applied the code to the simulation of a transverse jet. Cocle *et al.* [33] use a hybrid of the Vortex-In-Cell method and FMM to utilize the benefit of both methods. Sbalzarini *et al.* [34] developed a particle-mesh library that calculates one vortex method iteration for 268 million particles in 85 seconds using 128 processors. Using this library, Chatelain *et al.* [35] have performed a calculation of wing tip vortices using 10 billion vortex elements. Sheel *et al.* [25] used a special purpose computer originally designed for molecular dynamics (MDGRAPE-2) and showed a 100-fold acceleration of the direct interaction calculation. Later, they applied the FMM on the MDGRAPE-3 and observed a 16 times speed-up of the FMM [26].

In the present work, we will implement the FMM and pseudo-particle method on a cluster of 64 GPUs. The details of the FMM, pseudo-particle method, their parallelization and implementation on the GPU are given in the following section.

3. FMM

3.1. FMM for Vortex Methods

In order to keep this paper self-contained, the minimum requirements for formulating the present fast multipole method are shown first. The Green's function for the Laplace kernel can be approximated by the multipole expansion

$$\sum_{j=1}^{N} G \approx \frac{1}{4\pi} \sum_{n=0}^{p} \sum_{m=-n}^{n} \underbrace{r_i^{-n-1} Y_n^m(\theta_i, \phi_i)}_{S_i} \left\{ \sum_{j=1}^{N} \underbrace{\rho_j^n Y_n^{-m}(\alpha_j, \beta_j)}_{M_j} \right\}, \tag{12}$$

and also the local expansion

$$\sum_{j=1}^{N} G \approx \frac{1}{4\pi} \sum_{n=0}^{p} \sum_{m=-n}^{n} \underbrace{r_i^{n} Y_n^m(\theta_i, \phi_i)}_{R_i} \left\{ \sum_{j=1}^{N} \underbrace{\rho_j^{-n-1} Y_n^{-m}(\alpha_j, \beta_j)}_{L_j} \right\}, \tag{13}$$

where p is the order of expansion. In these equations, the location of particle i relative to the center of expansion is expressed in spherical coordinates using (r_i, θ_i, ϕ_i). Similarly, the relative location of j is noted as $(\rho_j, \alpha_j, \beta_j)$. $Y_n^m(\theta, \phi)$ represents the spherical harmonics of the form

$$Y_n^m(\theta, \phi) = \sqrt{\frac{(n - |m|)!}{(n + |m|)!}} P_n^{|m|}(\cos\theta) e^{\iota m \phi}. \tag{14}$$

and $P_n^{|m|}(\mu)$ are the associated Legendre functions, which are obtained from the following recurrence relations.

$$(n-m)P_n^m(x) = x(2n-1)P_{n-1}^m(x) - (n+m-1)P_{n-2}^m(x) \tag{15}$$

$$P_m^m(x) = (-1)^m(2m-1)!(1-x^2)^{m/2} \tag{16}$$

$$P_{m+1}^m = x(2m+1)P_m^m(x). \tag{17}$$

We define the operators S_i, M_j, R_i, L_j in Eqs. (12) and (13) to simplify the equations in the following steps. Using these operators, Eq. (5) can be

written as

$$\mathbf{u}_i \approx \frac{1}{4\pi} \sum_{n=0}^{p} \sum_{m=-n}^{n} \left\{ \sum_{j=1}^{N} \boldsymbol{\alpha}_j M_j \right\} \times \nabla S_i, \tag{18}$$

$$\mathbf{u}_i \approx \frac{1}{4\pi} \sum_{n=0}^{p} \sum_{m=-n}^{n} \left\{ \sum_{j=1}^{N} \boldsymbol{\alpha}_j L_j \right\} \times \nabla R_i. \tag{19}$$

Similarly, Eq. (9) can be written as

$$\frac{D\boldsymbol{\alpha}_i}{Dt} \approx \frac{1}{4\pi} \sum_{n=0}^{p} \sum_{m=-n}^{n} \left\{ \sum_{j=1}^{N} \boldsymbol{\alpha}_j \times \nabla M_j \right\} (\boldsymbol{\alpha}_i \cdot \nabla S_i), \tag{20}$$

$$\frac{D\boldsymbol{\alpha}_i}{Dt} \approx \frac{1}{4\pi} \sum_{n=0}^{p} \sum_{m=-n}^{n} \left\{ \sum_{j=1}^{N} \boldsymbol{\alpha}_j \times \nabla L_j \right\} (\boldsymbol{\alpha}_i \cdot \nabla R_i). \tag{21}$$

The cutoff function does not appear in these equations since they are used to calculate the effect of the far field, for which it would have negligible effect. Unlike the potential equation, these equations require the calculation of gradients of the spherical harmonic. For example, in Eqs. (19) and (21) ∇R must be calculated. If we take the gradient of R in spherical coordinates, this results in

$$\nabla R = \left(n\rho^{n-1} Y, \rho^n \frac{\partial Y}{\partial \theta}, \iota m \rho^n Y \right). \tag{22}$$

The calculation of these values is straightforward except for the derivative of Y_n^m, which from Eq. (14) becomes

$$\frac{\partial Y_n^m(\theta, \phi)}{\partial \theta} = \sqrt{\frac{(n - |m|)!}{(n + |m|)!}} \frac{\partial P_n^{|m|}(cos\theta)}{\partial \theta} e^{\iota m \phi}. \tag{23}$$

The derrivative of P_n^m can be calculated from

$$\frac{\partial P_n^m}{\partial \theta} = \frac{(n - m + 1) P_{n+1}^m - (n + 1) \cos\theta P_n^m}{\sin\theta}. \tag{24}$$

The ∇M in Eq. (20) can be calculated in a similar fashion.

3.2. Rotation Based Translation

The translation operators of the standard FMM have a complexity of $O(p^4)$. For problems that require high accuracy, the order of expansion p must be increased. There are a few methods which can reduce the complexity of the translation operator. The rotation based translation [36] has a complexity of $O(p^3)$, and the plane wave based translation [37] has a complexity of $O(p^2 \log p)$. However, the asymptotic constant is smaller for the methods with higher complexity, so for calculations that do not require large p, the effectiveness of these methods is questionable. For our case, the minimum accuracy requirements of the vortex method suggest that the order of expansion should be at least $p = 10$. For this case, the rotation based translation has the best performance.

The rotation based translation makes use of a certain property of the spherical harmonic. Assuming that $\theta = 0$, and $\phi = 0$ in Eq. (14) yields

$$Y_n^m(0,0) = \sqrt{\frac{(n-|m|)!}{(n+|m|)!}} P_n^{|m|}. \tag{25}$$

Following this, the translation operators can be written as

$$M_j^k = \sum_{n=0}^{j} \frac{\hat{M}_{j-n}^k A_n^0 A_{j-n}^k \rho^n Y_n^0(0,0)}{(-1)^n A_j^k} \tag{26}$$

$$L_j^k = \sum_{n=0}^{p-1} \frac{\hat{M}_n^k A_n^k A_j^k Y_{j+n}^0(0,0)}{(-1)^j A_{j+n}^0 \rho^{j+n+1}} \tag{27}$$

$$L_j^k = \sum_{n=j}^{p-1} \frac{\hat{L}_n^k A_{n-j}^0 A_j^k \rho^{n-j} Y_{n-j}^0(0,0)}{A_n^k}. \tag{28}$$

where

$$A_n^m = \frac{(-1)^n}{(n-m)!(n+m)!}. \tag{29}$$

The $\hat{M}$ and $\hat{L}$ represent the expansion coefficients before each translation, while the unmarked M and L represent the expansion coefficients after the translation. From the definition of M and L in Eqs. (12) and (13), we can see that M and L are the spherical harmonic Y multiplied by values which are independent of the angle. Thus, the rotation of M and L directly follow

that of Y.

$$M_n^m(\alpha+\theta,\beta+\phi) = \sum_{k=-n}^{n} D_n^{km}(\theta,\phi) M_n^k(\alpha,\beta) \tag{30}$$

$$L_n^m(\alpha+\theta,\beta+\phi) = \sum_{k=-n}^{n} D_n^{km}(\theta,\phi) L_n^k(\alpha,\beta). \tag{31}$$

The Wigner D matrix can be decomposed into

$$D_n^{km}(\theta,\phi) = d_n^{km}(\theta) e^{\iota(k+m)\phi}. \tag{32}$$

There exist a variety of recurrence relations for d_n^{km}. It is true that some of these recurrence relations [38] accumulate too much round off error for calculations with large expansion order. The recurrence relations by White & Head-Gordon [36] are sufficient for large expansion order $p > 10$. The recurrence relations can be written as

$$d_n^{n,n} = \cos\left(\frac{\theta}{2}\right)^{2n} \tag{33}$$

$$d_n^{n,k-1} = \sqrt{\frac{n+k}{n-k+1}} \tan\left(\frac{\theta}{2}\right) d_n^{n,k} \tag{34}$$

$$\begin{aligned} d_n^{m-1,k} = & \sqrt{\frac{n(n+1)-k(k+1)}{n(n+1)-m(m-1)}} d_n^{m,k+1} \\ & - \frac{k+m}{\sqrt{n(n+1)-m(m-1)}} \frac{\sin\theta}{1+\cos\theta} d_n^{m,k}. \end{aligned} \tag{35}$$

Since the translation stencil of the multipole-to-local translation is fixed, we precompute the Wigner D matrix and also the translation matrix in Eq. (27).

3.3. Pseudo-Particle Method (PPM)

Instead of calculating the multipole and local moments at the center of the boxes, PPMs calculate the physical properties of interest at quadrature points placed on a spherical shell surrounding the boxes.[24] The multipole expansion and translation is expressed as

$$q_i = \sum_{j=1}^{N} q_j \sum_{n=0}^{p} \frac{2n+1}{K} \left(\frac{\rho_j}{r_s}\right)^n P_n(\cos\gamma_{ij}) \tag{36}$$

Table 1: Number of quadrature points K corresponding to p

p	K
1	4
2	14
3	26
4	36
5	60
6	84
7	108
8	144
9	180
10	216

and the local expansion and translation becomes

$$q_i = \sum_{j=1}^{N} q_j \sum_{n=0}^{p} \frac{2n+1}{K} \left(\frac{r_s}{\rho_j}\right)^{n+1} P_n(\cos\gamma_{ij}). \tag{37}$$

q is the physical property of interest, K is the number of quadrature points on the sphere circumscribing the box, and r_s is the radius of this sphere. $P_n(\mu)$ is the Legendre polynomial, and γ_{ij} is the angle between the position vector of source and target particles. The quadrature points are positioned according to the spherical-t design [24]. The number of quadrature points K that are required to achieve the same accuracy as a multipole expansion of order p is shown in Table 1. The multipole-to-local translation requires the calculation of K^2 interactions, and may become quite expensive for large K.

Given that $\mathbf{x}_i = (r_i, \theta_i, \phi_i)$ and $\mathbf{x}_j = (\rho_j, \alpha_j, \beta_j)$, $\cos\gamma_{ij}$ can be written as

$$\cos\gamma_{ij} = \frac{\mathbf{x}_i \cdot \mathbf{x}_j}{r_i \rho_j} \tag{38}$$

The physical property of interest q in Eqs. (36) and (37) can be the source strength $\boldsymbol{\alpha}$ during the upward pass and velocity $\mathbf{u}$ during the downward pass. In this case the multipole to local translation can be calculated by Eq. (5), and becomes similar to the direct summation, but for pseudo-particles instead of particles.

3.4. Parallelization of FMM

The present study involves the parallelization of the FMM on distributed memory architectures using MPI. Balancing the computational workload and amount of data transfer between the processes is a challenging task for adaptive FMMs. However, for the present calculation of the homogeneous turbulence, the particle density remains constant throughout the entire domain and varies little over time. Furthermore, the periodic boundary condition prevents the load of the boarder cells from becoming small. Therefore, the workload between different processes can be balanced by simply partitioning the parallel computation domain according to the oct-tree structure of the FMM. In order to minimize memory requirements, each process holds only the particle information within the partitioned domain, throughout most of the vortex method calculation. Thus, the particle information and multipole moments near the borders of the partitioned domain must be communicated. The flow of calculation is as follows.

1. Communicate particle data (P2Pcomm)
2. Particle to particle interaction (P2P) : Calculated on GPUs
3. Multipole expansion from particles (P2M)
4. Multipole to multipole translation (M2M)
5. Communicate multipole data (M2Lcomm)
6. Multipole to local translation (M2L) : Calculated on GPUs for PPM
7. Local to local translation (L2L)
8. Local expansion to particles (L2P)

At step 1. the information necessary for calculating the direct summation of all particles within the partitioned domain is communicated. At step 2. the direct summation is performed. Steps 3 and 4 do not require any communication, and are performed locally. At step 5. all the multipole moments that are necessary for the multipole to local translation at that particular level, are transmitted. Then, the multipole to local translation is calculated at step 6. Steps 7 and 8 also do not require any communication, and are performed locally.

4. Graphics Processing Units

4.1. Hardware Specifications

In the present calculations, we use NVIDIA's GeForce 8800 GT which has 112 streaming processors, where eight streaming processors are grouped into

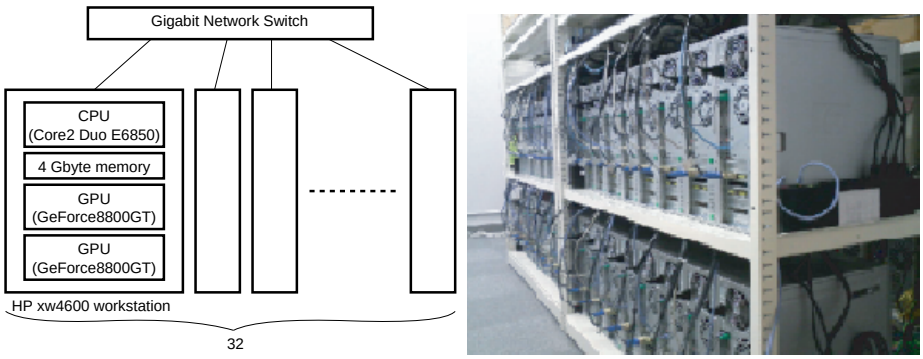


Figure 1: Block diagram of the computing system (left) and its photo (right)

a multiprocessor. It performs 32 SIMD (Single Instruction Multiple Data) operations in four clock cycles. Each multiprocessor has only 16 Kbyte of shared memory, which can be accessed at high speed. Larger data must be stored in the global memory (the memory chip outside of the GPU processor), which then requires a very large latency (400-600 clock cycles) when loading to the multiprocessor. Therefore, the key to achieving high efficiency on GPUs, is to run as many threads as possible in order to obscure the memory latency. The clock frequency of the multiprocessor of our XFX's GPU is 1,562 MHz, which is slightly higher than the NVIDIA's reference board ($1,500$ MHz). The peak performance of the GeForce 8800 GT is 350 GFlops ($= 1,562 \times 2 \times 112$).

Fig. 1 shows a schematic of our GPU cluster system. We used 32 nodes of PCs (HP xw4600), each of them having a dual-core CPU (Intel Core2 Duo E6850), 4Gbyte of memory, and two GPUs (XFX PV-T88P-YDQ4 which has NVIDIA's GeForce 8800 GT). A GPU is plugged in via 16x PCI Express 2.0 slots. We added a cooling fan next to each GPU slot to enhance the reliability of the system. The PCs are connected by a gigabit ethernet network through a 48-port HUB (NETGEAR GS748TS-100JPS).

4.2. Direct Summation on GPU

Our vortex method code is written in Fortran, and the subroutines for the calculation of Eqs. (5) and (9) are modified to call a library routine that uses the GPU. This library routine is developed using the NVIDIA's CUDA programming environment. We used 128 threads per multiprocessor, where each thread handles one target particle. We use a one dimensional grid of one

dimensional blocks of threads. All the arithmetic operations on the GPU are performed in single precision. The flow of the library routine is as follows.

1. The coordinates $\mathbf{x}_j$, strengths $\boldsymbol{\alpha}_j$, volume V_j, and core radius σ_j of the source particles are stored in a single array of size $8N$ in the global GPU memory. The coordinates $\mathbf{x}_i$, strengths $\boldsymbol{\alpha}_i$, volume V_i of the target particles are also stored in a single array of size $7N$ in the global GPU memory. If the number of vortex elements N is too large for the global GPU memory to handle at once, the library automatically divides the arrays to manageable sizes and passes them to the GPU sequentially.
2. Each thread handles one target particle. The information of target particles is copied from the global memory to the register.
3. Each block handles a group of source particles, which is composed of chunks of 128. The information of source particles is copied from the global memory to the shared memory in chunks of 128. Each thread reads a different part of the chunk in a coalesced manner before the thread is synchronized.
4. A loop is executed to run through all 128 particles of the chunk and sums the effect to the target variable in the local memory. The threads are synchronized after each loop is finished.
5. After the summation loop is completed for all chunks of the source particles, the information of the target particles is copied to the global memory. Once the information is copied to the global memory, the next thread block is executed.

In order to evaluate the performance of the present GPU calculation, we first measured the performance of a serial GPU direct summation routine. 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 $\sigma/\Delta x = 1$.

The calculation time of the direct summation on a serial CPU and GPU is shown in Fig. 2 for different number of particles N. Calculation of the stretching is slightly more complex than the velocity calculation, thus requires a slightly longer time. Both the CPU and GPU calculation scale as $O(N^2)$, but the GPU is approximately 160 times faster. The CPU is a Intel Core2 Duo E6850 and the code is written in Fortran 90 and compiled on a Intel Fortran compiler 10.0, which performs vectorization of the code by default. Both cores are used by explicitly defining openmp routines and compiling

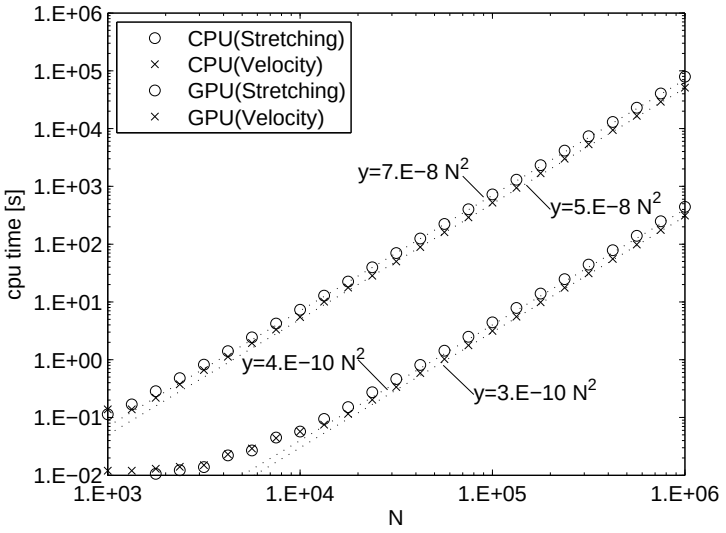


Figure 2: Cputime of the direct summation on a serial CPU and GPU. Dotted lines are functional fits as $y = f(N)$, where y is the time, and N is the number of particles.

with the "-openmp" option. The GPU is a NVIDIA GeForce 8800 GT and the Fortran code calls a CUDA library, which was compiled with NVCC using the "-use_fast_math" option.

In order to compare the performance of our code with the ones by Stock & Gharakhani [21] and Gumerov & Duraiswami [22] more directly, we have performed "potential+force" and "velocity+stretching" calculations on the CPU and GPU. The calculation time is shown in Fig. 3. First, by comparing Fig. 2 with Fig. 3, we see that the "velocity+stretching" takes only about 1.1 times longer than just the stretching on the CPU, while it takes around 1.5 times longer on the GPU. The GPU calculations slow down because of the extra data transfer that is necessary for writing both the velocity and stretching results to the global memory. The calculation for "stretching+velocity" adds only 18 floating point operations to the stretching only, because the cutoff function used for the first term of the stretching can also be used for the velocity calculation. Second, it can be seen from Fig. 3 that the calculation of "potential+force" takes only 1/5 of the execution time of "velocity+stretching". This ratio can be used to compare the results of Stock & Gharakhani [21] and Gumerov & Duraiswami [22]. Third, Stock

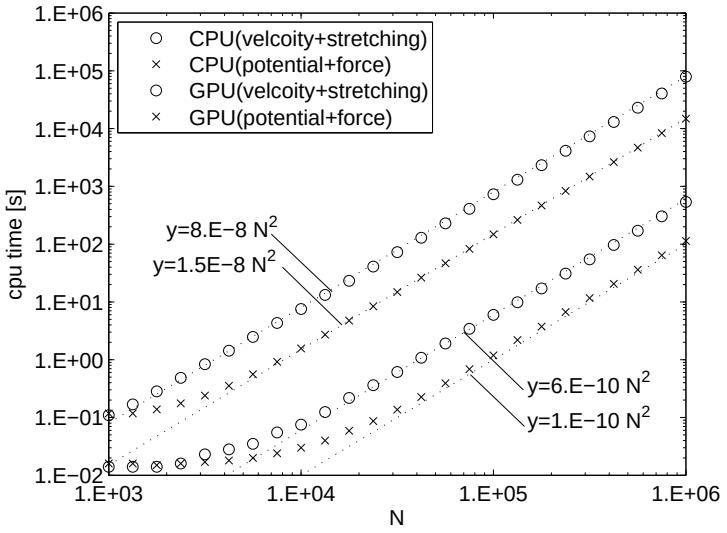


Figure 3: Cputime of the direct summation on a serial CPU and GPU. Dotted lines are functional fits as $y = f(N)$, where y is the time, and N is the number of particles.

& Gharakhani [21] report that their direct N^2 calculation for the "velocity+stretching" for $N = 500,000$ takes $11,242$s on the CPU and 88.7s on the GPU. From Fig. 3 we see that our code would take $20,000$s on the CPU and 150s on the GPU. The speed-up ratio by Stock & Gharakhani [21] $11,242/88.7 \approx 127$ is similar to ours $20,000/150 \approx 133$. The difference in the absolute speed is likely to be caused by the difference in hardware, and also the difference in the way the velocity gradient is handled in our stretching calculation. Fourth, Gumerov & Duraiswami [22] report that their direct N^2 calculation for the "potential+force" for $N = 1,048,576$ takes around 107,100s on the CPU and 125.36s on the GPU. From Fig. 3 we see that our calculation for the "potential+force" takes approximately $16,500$s on the CPU and 110s on the GPU. The speed-up ratio of Gumerov & Duraiswami [22] $107,100/125.36 \approx 855$ is quite different from ours $16,500/110 \approx 150$. Considering the fact that both Stock & Gharakhani [21] and Gumerov & Duraiswami [22] use the GeForce 8800GTX, which is 128/112 times faster than our GeForce 8800GT, the large acceleration rate of Gumerov & Duraiswami [22] suggests that their CPU is nearly 6 times slower than the dual-core CPU used by Stock & Gharakhani [21] and the present authors.

Table 2: Flops count of the velocity and stretching kernel

	velocity			stretching		
Operation	Count	Flo	Total	Count	Flo	Total
$+ - *$	33	1	33	64	1	64
fdividef	1	5 (1)	5 (1)	1	5 (1)	5 (1)
sqrtf	1	4 (1)	4 (1)	1	4 (1)	4 (1)
expf	1	8 (1)	8 (1)	1	8 (1)	8 (1)
erff	1	8 (1)	8 (1)	1	8 (1)	8 (1)
powf	1	13 (1)	13 (1)	2	13 (1)	26 (2)
Total			70 (38)			115 (70)

The CPU/GPU speed-up ratio is only a relative indicator for speed. We will look at the speed of the present GPU calculation in terms of Flops, in order to evaluate the absolute speed. When measuring the Flops, we used the operation count shown in Table 2. While the standard Flops count (shown in parentheses), treats the **sqrt, exp, pow** as one floating point operation, we adopt a measure that takes into account the the number of clock cycles it takes relative to a single-precision floating-point add. For N-body problems, which often involve **sqrt, exp, pow**, and even **erf** operations, the standard Flops count equates complex kernels to bad performance. Therefore, we believe the standard Flops count alone is insufficient for evaluating the computational efficiency of our complex kernel and wish to provide an alternative measure along with the standard Flops count. *Operation* denotes the type of operation performed on the GPU, *Count* is the number of times these operations appear during a pairwise interaction, *Flo* is the number of clock cycles relative to a single-precision floating-point add.. These numbers were taken from the NVIDIA CUDA programing guide [41], except for the **erff**, which we assumed to be equal to **expf**.

Fig. 4 shows the Flops for the direct interaction on a serial GPU. The stretching term calculation reaches 260 GFlops, while the velocity reaches 225 GFlops. When the standard Flops count is used, the stretching term calculation achieves $260/115 \times 70 \approx 158$ GFlops, while the velocity calculation reaches $225/70 \times 38 \approx 122$ GFlops. The calculations of Hamada & Iitaka [18] and Nyland et al. [16] suggest that there is still some room for improvement

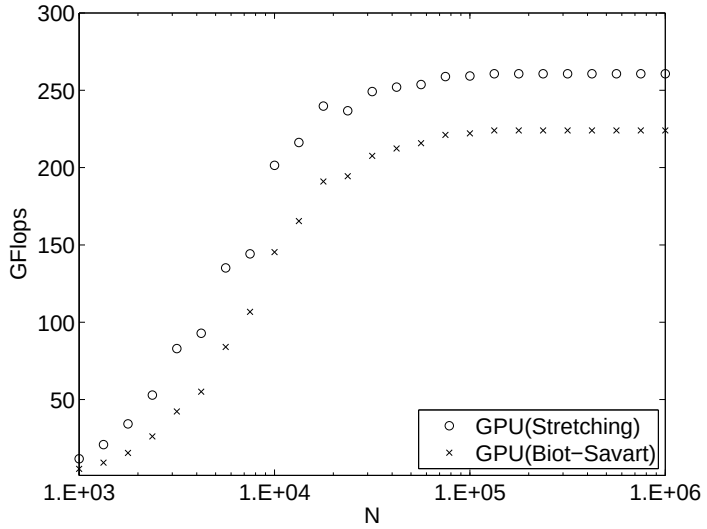


Figure 4: Flops of the direct summation on a serial GPU. Dotted lines are functional fits as $y = f(N)$, where y is the time, and N is the number of particles.

in the $10^3 < N < 10^4$ region.

4.3. FMM on GPU

There have been few previous attempts to efficiently implement fast algorithms on GPUs. Stock & Gharakhani [21] ported the entire treecode onto a NVIDIA GeForce 8800 GTX. They fixed the number of particles per box to 64 by excluding the particles far from the pole axis of the spherical harmonics, and calculating them directly. As a result, they could use one block of threads per box, while using 64 threads per block and having one thread per target particle without wasting any threads. Their treecode on the GPU was approximately 17 times faster than their treecode on the CPU.

Gumerov & Duraiswami [22] implemented the entire FMM on a NVIDIA GeForce 8800 GTX. Their FMM used many effective techniques, such as real basis functions, the RCR decomposition, a translation stencil that reduces the multipole-to-local translations from 189 to 119, and a variable truncation number. On the GPU, they used one thread block per box for the particle-to-particle interaction, one thread per box for the particle-to-multipole, local-to-particle, and multipole-to-local translation. They achieved a 72-fold speed-up

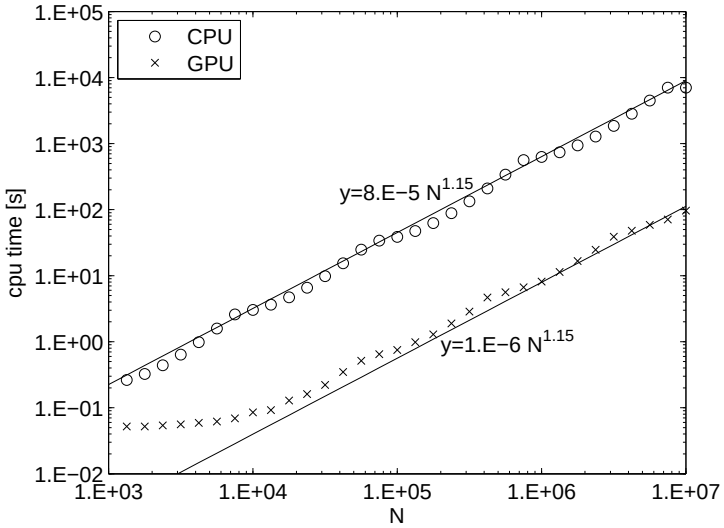


Figure 5: Cputime of the FMM on a serial CPU and GPU ($p = 10$). Dotted lines are functional fits as $y = f(N)$, where y is the time, and N is the number of particles.

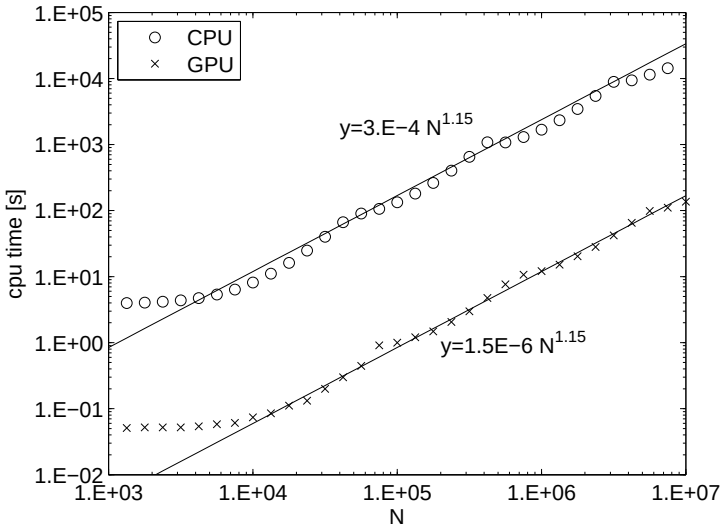


Figure 6: Cputime of the PPM on a serial CPU and GPU ($p = 10$). Dotted lines are functional fits as $y = f(N)$, where y is the time, and N is the number of particles.

for the FMM, while their direct calculation was 855 times faster on the GPU for $N = 1,048,576$ particles. However, they report that the multipole-to-local translation itself achieved only speed-ups in the range of 2-5. This is less than 1% of the speed-up of their direct interaction.

In the present calculation we have adopted all of the techniques used by Gumerov & Duraiswami [22], except for the reduced translation stencil. All parts of the FMM algorithm are ported to the GPU. The basic approach of our GPU implementation is also similar to Gumerov & Duraiswami [22], in the sense that we both use a one dimensional block of threads to handle each FMM box, and how each thread writes one expansion coefcient to the shared memory. However, the details of the actual CUDA kernel may differ significantly since this information is not given in Gumerov & Duraiswami [22]. In our CUDA kernel, the translation coefficients are generated on-th-fly but recurrence relations are calculated only up to the necessary order. For example, the thread that calculates L_0^0 in Eq. (27) requires only the translation coefficients for j = 0 and k = 0. Even though this creates an imbalance in the work load among different threads, we have observed a significant speed-up by using this technique. Furthermore, we associate the target expansion coefficients/particles to the threads and sum the effect of the different source expansion coefficients/particles as they are calculated. Therefore, it is not necessary to perform any kind of reduction calculation on the GPU. The loops are run from higher order expansions to lower order expansions so that smaller coefficients are summed first. This allows us to minimize the round-off error in the single precision calculations on the GPU.

We first present the results of the velocity calculation using the FMM and PPM on a serial CPU and GPU. The observations made for the velocity calculation are directly applicable to the stretching or velocity+stretching calculation. Thus, we will show only the results for the velocity calculations hereafter. The order of multipole expansions is set to $p = 10$ unless otherwise noted.

Fig. 5 shows the calculation time of the FMM on a serial CPU and GPU. Our FMM does not scale exactly as $O(N)$, but rather shows a scaling close to $O(N^{1.15})$. 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. For the FMM on the CPU the box level switches from 2 to 3 at $N = 10^4$, 3 to 4 at $N = 10^5$, 4 to 5 at $N = 10^6$, and 5 to 6 at $N = 10^7$. On the other hand,

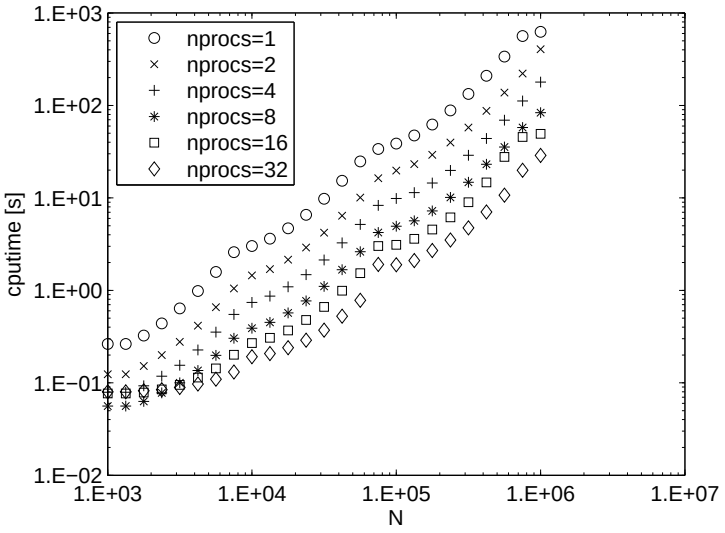


Figure 7: Cputime of the FMM on parallel CPUs

the box level of the FMM on the GPU switches from 2 to 3 at $N = 7 \times 10^4$ and 3 to 4 at $N = 5 \times 10^5$, and 4 to 5 at $N = 3 \times 10^6$. In other words, the FMM box contains more particles for the GPU calculations.

The calculation time of the PPM on a serial CPU and GPU are shown in Fig. 6. Our PPM also scales as $O(N^{1.15})$. By comparing Figs. 5 and 6, it can be seen that the calculation time of the FMM on the GPU is 1.5 times faster than the PPM, while the FMM on the CPU is around 3.75 times faster than the PPM on the CPU. The multipole-to-local translation of the rotation based FMM requires the calculation of Eq. (27) for $p = 10$, while the multipole-to-local translation of the PPM requires the calculation of Eq. (5) for $K = 216$ to achieve the same accuracy. Thus, the calculation load of the multipole-to-local translation of the PPM is much larger than the FMM, so when it is processed on the CPU it takes much longer. However, when it is processed on the GPU it hides the memory latency due to its larger calculation load, and the inefficiency of the PPM is canceled out.

Before showing the results of the FMM on parallel GPUs, we will present the results of the FMM on parallel CPUs for reference. Fig. 7 shows the calculation time of the FMM for different number of CPUs. *nprocs* is the number of MPI processes (number of CPUs/GPUs). The parallel efficiency

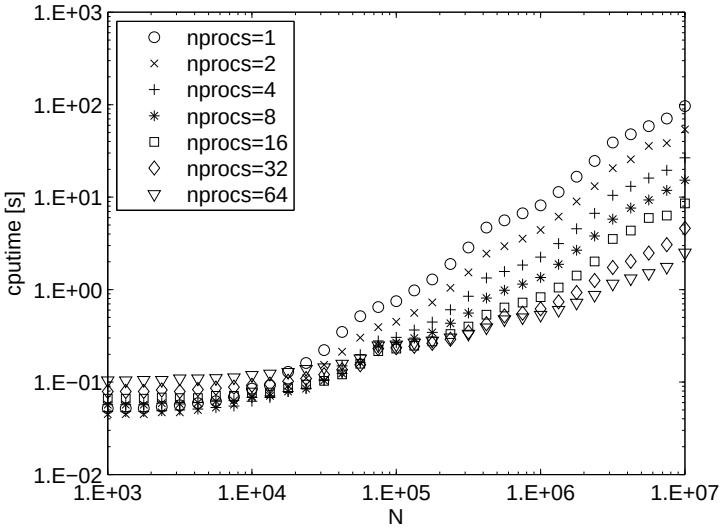


Figure 8: Cputime of the FMM on parallel GPUs

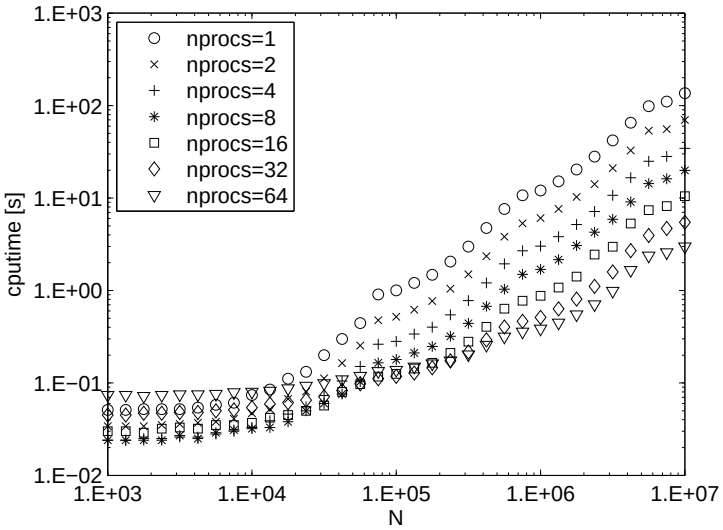


Figure 9: Cputime of the PPM on parallel GPUs

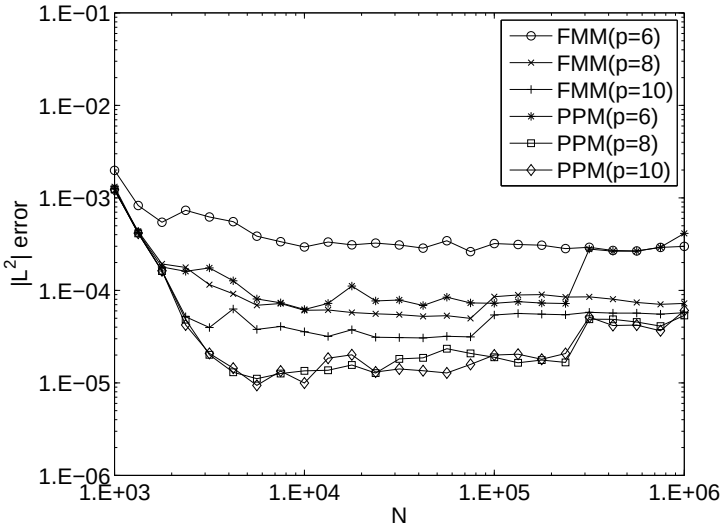


Figure 10: L^2 norm error of the velocity calculation

for 32 CPUs is 49% at $N = 10^4$, 63% at $N = 10^5$, and 68% at $N = 10^6$.

Fig. 8 shows the calculation time of the FMM for different number of GPUs. Since there are two GPUs per workstation, the parallel calculation on GPUs has results for until $nprocs = 64$. For fairness, we will first compare the results of 32 GPUs with the results of 32 CPUs shown above. The parallel efficiency for 32 GPUs is 3% at $N = 10^4$, 10% at $N = 10^5$, 41% at $N = 10^6$, and 66% at $N = 10^7$. The low parallel efficiency for small N is a direct consequence of the low performance of GPUs for small N, as shown in Fig. 4. 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. The parallel efficiency for 64 GPUs is 1% at $N = 10^4$, 5% at $N = 10^5$, 24% at $N = 10^6$, and 60% at $N = 10^7$. It can be seen from Fig. 8 that the calculation time for $N < 10^4$ increases for $nprocs \geq 16$. This is because the time spent on communication becomes large compared to the calculation time. The communication time increases monotonically as $nprocs$ increases.

Fig. 9 shows the calculation time of the PPM for different number of GPUs. In this case, The parallel efficiency for 32 GPUs is 4% at $N = 10^4$, 26% at $N = 10^5$, 73% at $N = 10^6$, and 78% at $N = 10^7$. Also, for 64 GPUs

the parallel efficiency is 1% at $N = 10^4$, 11% at $N = 10^5$, 49% at $N = 10^6$, and 71% at $N = 10^7$. The PPM seems to have higher parallel efficiency compared to the FMM. Though, the FMM for $N = 10^7$ on 64 GPUs takes approximately 2.5s, and is slightly faster than the PPM for the same N on 64 GPUs, which takes approximately 3.0s.

Finally, the error of the present FMM and PPM are investigated by comparing them with the results of the direct interaction on the CPU. The $|L^2|$ error is shown in Fig. 10 for different p. The large error for small N is due to the following reason. The Biot-Savart law has a cutoff function as shown in Eq. (6), but the FMM and PPM calculate for a singular kernel as shown in Eqs. (18)-(21), (36), and (37). This has negligible effect if the neighbor region of the FMM is significantly large compared to σ, because the effect of the cutoff function decays rapidly. However, for the present calculations we use $\sigma = 2\pi N^{-1/3}$ and the FMM box level is always larger than 2. Thus, for small N, the neighbor region is relatively small compared to σ, so the FMM would be assuming a singular function at a location where the cutoff function is non-negligeble.

For the FMM the error decreases as p increases. For $p = 10$ the error remains below 10^{-4} for all N. On the other hand, the PPM error is large for $p = 6$, but is almost the same for $p = 8$ and $p = 10$. Furthermore, in contrast to the FMM, the PPM error increases significantly when the box level is changed from 2 to 3. This is caused by the accuracy of the particle-to-multipole calculation being significantly higher than the multipole-to-multipole translations in the pseudo-particle method. This allows us to achieve high accuracy when the box level is 2, where multipole-to-multipole translations are not necessary. We have also confirmed that the $|L^2|$ error of the FMM and PPM are the same for the CPU and GPU calculation and also the parallel calculations using CPUs and GPUs.

5. Calculation of Isotropic Turbulence

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 $[-\pi, \pi]$ and has periodic boundary conditions in all directions. In the present vortex method calculation, the periodic boundary condition is approximated by the use of periodic images. The calculation of the periodic images is done by the FMM by extending the oct-tree outwards, thus adding only a few percent of

additional calculation time to the non-periodic FMM. Details of the periodic FMM are shown in our previous publication [30]. The number of calculation points was $N = 256^3$ 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 64 GPUs for the calculation of the isotropic turbulence.

The spectral Galerkin method with primitive variable formulation 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 a single processor without using any 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 strength of the vortex elements was calculated from the vorticity field on the grid by solving a system of equations for (3). The core radius of the vortex elements were set to $2\pi/N$ so that the overlap ratio was 1.

5.2. Calculation Results

Fig. 11 shows the decay of kinetic energy, which is defined as

$$K = \frac{1}{2} \sum_{i=1}^{N} u_i^2 + v_i^2 + w_i^2. \tag{39}$$

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 turnover time have the following relation.

$$L = \frac{\pi}{2u'^2} \int k^{-1} E(k) dk \tag{40}$$

$$T = L/u'. \tag{41}$$

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

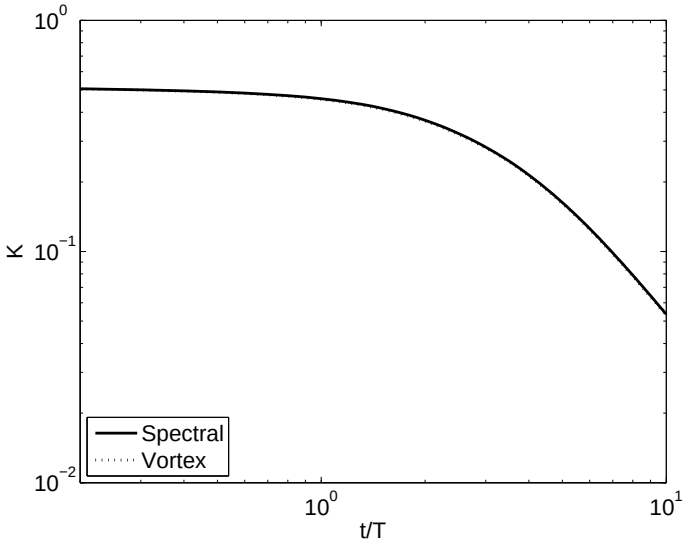


Figure 11: Decay of kinetic energy

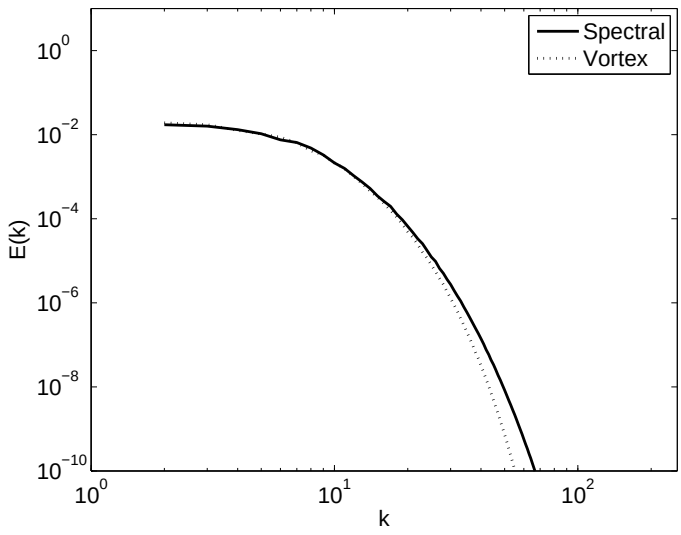
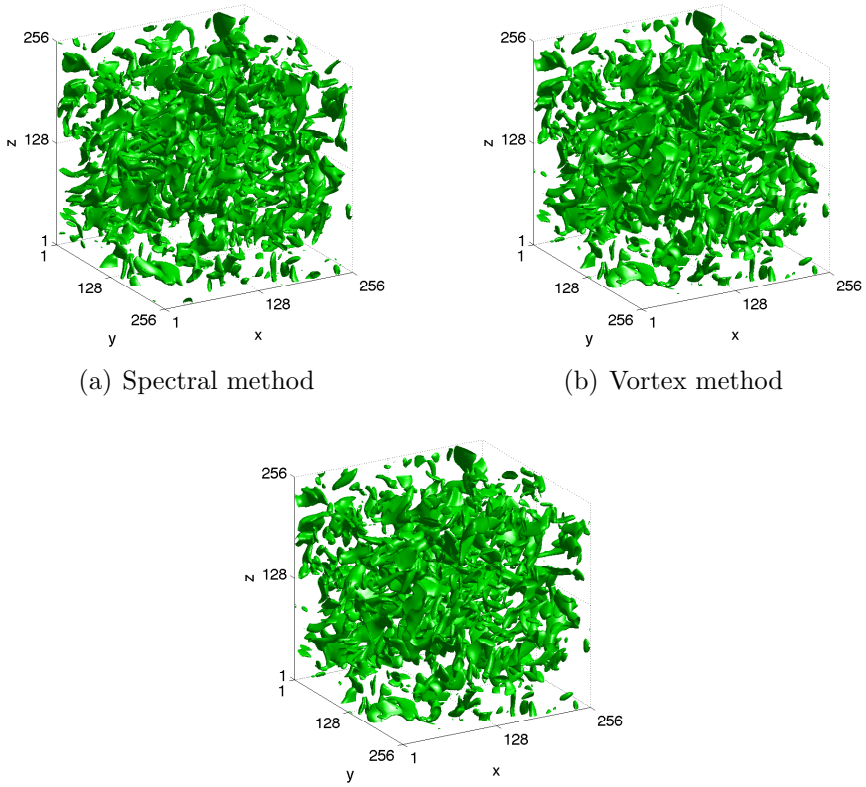


Figure 12: Energy spectra at $t/T = 10$



(a) Spectral method

(b) Vortex method

(c) $II_{spec} - II_{vor} = 2 \times 10^{-4}$

Figure 13: Isosurface of the second invariant (II) of the velcoity derivative tensor

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.

Fig. 12 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 Fig. 13. Fig. 13(a) is the isosurface of the spectral method, Fig. 13(b) shows the isosurface from the vortex method calculation, while the isosurface of the difference of II between the two methods is shown in Fig. 13(c). 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 Fig. 12, where the kinetic energy at higher wave numbers do not match.

These results indicate the soundness of the present vortex method calculation using GPUs. It is fair to say that the single precision calculation of the velocity does not have any detrimental effect on the final accuracy of our turbulence simulations. Furthermore, the calculation of the present vortex method on 64 GPUs took $4,120$s, while the spectral method on 32 CPUs took $4,910$s.

5.3. Cost Performance of the Present Simulation

Table 3 shows the number of floating point operations involved in the present calculation. The operation count is only for the direct summation (P2P) part, because the number of operations of other parts are relatively small and also difficult to calculate exactly. The simulation domain is divided into $32 \times 32 \times 32$ sub-cells. The number of floating point operations required for a time step is counted as 3.08×10^{13} based on the relative clock cycle count, and 2.04×10^{13} using the standard Flops count.

Table 4 shows the cost of our computing hardware. We used 32 HP workstations for the host computer. Two GPUs are installed in a PC. Additional 4 Gbyte of memory and two cooling fans are also installed on each of the host computers. The conversion rate between US dollar and Japanese yen (JPY) is \$1=107 JPY, which is the rate at Jan. 31, 2008, when the hardware was installed. The total price amounts to \$70,323.

Finally, Table 5 shows the performance and the cost performance of our simulation. We have calculated the isotropic turbulence for 1000 steps, which took $4,120$ seconds. The performance is 7.48 TFlops based on the relative clock cycle count and 4.95 TFlops based on the standard Flops count. The cost performance is \$9.4/GFlops for the former and \$14.2/GFlops for the latter.

Table 3: Number of floating point operations for Run2

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
Flos per pairwise velocity+ stretching interaction	K	133 (88)
Flos per time step	KNN_j	3.08×10^{13} (2.04×10^{13})

Table 4: Price of the computing system (JPY)

Parts	Price per unit	Number of units	Sum
Host PC (HP xw4600)	131,300	32	4,201,600
2 Gbyte DIMM	10,000	64	640,000
GPU (GeForce8800GT)	38,200	64	2,444,800
Cooling fan	1,000	64	64,000
HUB (48-port gigabit)	164,200	1	164,200
Cables	10,000	1	10,000
Total (1$=107JPY)			7,524,600 $70,323

Table 5: Performance and cost performance

Description	Equation	Value
Flos per time step	$N_{fp,step}$	3.08×10^{13} (2.04×10^{13})
Number of time steps	N_{step}	1000
Total of Flos in the simulation	$N_{fp} = N_{fp,step} N_{step}$	3.08×10^{16} (2.04×10^{16})
Total time for the simulation (sec)	T_{total}	4,120
Performance (TFlops)	$S = N_{fp}/T_{total}$	**7.48 (4.95)**
Price (US dollar)	P	70,323
Price per performance ($/GFlops)	P/S	**9.40 (14.2)**

6. Conclusion

The fast multipole method (FMM) and pseudo-particle method (PPM) have been calculated on a cluster of NVIDIA GeForce 8800 GT GPUs. All parts of the algorithm for both the FMM and PPM have been ported to the GPU. We applied our acceleration method to the vortex method calculation of a homogeneous isotropic turbulence using 256^3 vortex elements and compared the results with a spectral method calculation using 256^3 grid points. The following conclusions were obtained from the results of our calculation.

The direct summation on a single GPU (GeForce 8800GT) for velocity, stretching, velocity+stretching, and potential+force show an acceleration rate between 133 and 160 over the CPU (Intel Core2Duo E6850, using both cores). We have calculated the Flops using the standard Flops count, and also an alternative Flops count that takes into account the different number of clock cycles required for different operations. The standard Flops count reaches 158 GFlops, while the alternative Flops count reaches 260 GFlops using a single GPU for the direct summation of the stretching calculation.

Both our FMM and PPM scale as $O(N^{1.15})$, and the speed of the calculation on the GPU are similar, while the calculation on the CPU is 3.75 times faster when using the FMM. The parallel calculation of the FMM using 32 CPUs yields a parallel efficiency of 68% for $N = 10^6$ particles. The FMM on 32 GPUs has only a 41% parallel efficiency for $N = 10^6$ particles, though it rises to 66% for $N = 10^7$ particles. On the other hand, the PPM on 32

GPUs has a 73% parallel efficiency even at $N = 10^6$.

The $|L^2|$ error of the PPM is lower than that of the FMM for the equivalent order of expansion p. Furthermore, there still remains some room for acceleration for the GPU calculation for small N. Accelerating for small N may lead to the further acceleration of the PPM calculation on the GPU. Thus, the PPM is an interesting alternative to standard FMMs for the calculation on parallel GPUs.

The present acceleration technique enabled the calculation of a homogeneous isotropic turbulence using a relatively large number of vortex elements. 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. Such accuracy for completely meshless turbulence simulations have not been reported previously. We also achieved a maximum speed of 7.48 TFlops, and a cost performance near \$9.4/GFlops. The calculation of the present vortex method on 64 GPUs took $4,120$s, while the spectral method on 32 CPUs took $4,910$s to calculate 1000 time steps.

Acknowledgement

This study was partially supported by the Core Research for Evolution Science and Technology (CREST) of the Japan Science and Technology Corporation (JST). We thank Dr. Hamada and Dr. Taiji for the fruitful discussions on GPU computing. The authors also thank the reviewers for their suggestive comments.

References

[1] P. Koumoutsakos, Annu. Rev. Fluid Mech. 37 (2005) 457.

[2] J. E. Barnes, P. Hut, Nature 324 (1986) 446.

[3] L. Greengard, V. Rokhlin, J. Comput. Phys. 73 (1987) 325.

[4] J. K. Salmon, M. S. Warren, G. S. Winckelmans, Int. J. Supercomputing Applications 8 (1994) 129.

[5] J. P. Singh, C. Holt, J. L. Hennessy, A. Gupta, in: Proc. of the Supercomputing Conference (1993) p. 54.

[6] C. H. Choi, K. Ruedenberg, M. S. Gordon, J. Comput. Chem. 22 (2001) 1484.

[7] J. Kurzak, B. M. Pettitt, J. Parallel Distrib. Comput. 65 (2005) 870.

[8] D. Sugimoto, Y. Chikada, J. Makino, T. Ito, T. Ebisuzaki, M. Umemura, Nature 345 (1990) 33.

[9] T. Fukushige, M. Taiji, J. Makino, T. Ebisuzaki, D. Sugimoto, Astrophys. J. 468 (1996) 51.

[10] R. Susukita, T. Ebisuzaki, B. G. Elmegreen, H. Furusawa, K. Kato, A. Kawai, Y. Kobayashi, T. Koishi, G. D. McNiven, T. Narumi, K. Yasuoka, Comput. Phys. Commun. 155 (2003) 115.

[11] T. Narumi, Y. Ohno, N. Futatsugi, N. Okimoto, A. Suenaga, R. Yanai, M. Taiji, in: Proc. of NIC Workshop 34 (2006) p. 29.

[12] T. Narumi, Y. Ohno, N. Okimoto, T. Koishi, A. Suenaga, N. Futatsugi, R. Yanai, R. Himeno, S. Fujikawa, M. Ikei, M. Taiji, Proc. of the SC06 (2006)

[13] E. Athanassoula, A. Bosma, J. C. Lambert, J. Makino, Mon. Not. R. Astron. Soc. 293 (1998) 369.

[14] J. Makino, Publ. of the Astronomical Society of Japan 43 (1991) 621.

[15] N. H. Chau, A. Kawai, T. Ebisuzaki, in:Proc. of 6th SCI 2002, Orlando, Colorado (2002) 477.

[16] L. Nyland, M. Harris, J. Prins, GPU Gems 3, ed. H. Nguyen, (Addison-Wesley, 2007) p. 677.

[17] R. G. Belleman, J. Bedorf, S. F. Portegies Zwart, New Astronomy 13 (2008) 103.

[18] T. Hamada, T. Iitaka, ArXiv Astrophysics e-prints, astro-ph/073100 (2007).

[19] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, T. Chiueh, ArXiv Astrophysics e-prints astro-ph:0707.2991v1 (2007).

[20] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, J. Comput. Chem. 28 (2007) 2618.

[21] M. J. Stock, A. Gharakhani, in: Proc. of 46th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada (2008).

[22] N. A. Gumerov, R. Duraiswami, J. Comput. Phys. 227 (2008) 8290.

[23] C. R. Anderson, SIAM J. ScI. Stat. Comput. 13 (1992) 923.

[24] J. Makino, J. Comput. Phys. 151 (1999) 910.

[25] T. K. Sheel, K. Yasuoka, S. Obi, Comput. Fluids 36 (2007) 1319.

[26] T. K. Sheel, R. Yokota, K. Yasuoka, S. Obi, Trans. Japan Soc. Comput. Eng. Sci., (2008) 0003.

[27] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, Y. Kaneda, in: Proc. of SC2002, Baltimore, Maryland (2002)

[28] R. T. Fisher, L. P. Kadanoff, D. Q. Lamb, A. Dubey, T. Plewa, A. Calder, F. Cattaneo, P. Constantin, I. Foster, M. E. Papka, S. I. Abarzhi, S. M. Asida, P. M. Rich, C. C. Glendenin, K. Antypas, D. J. Sheeler, L. B. Reid, B. Gallagher, S. G. Needham, IBM J. Res. & Dev. 52 (2008) 127.

[29] G.-H. Cottet, B. Michaux, S. Ossia, G. VanderLinden, J. Comput. Phys. 175 (2002) 702.

[30] R. Yokota, T. K. Sheel, S. Obi, J. Comput. Phys. 226 (2007) 1589.

[31] G. S. Winckelmans, J. K. Salmon, M. S. Warren, A. Leonard, B. Jodoin, ESAIM Proc. 1 (1996) 225.

[32] Y. M. Marzouk, A. F. Ghoniem, J. Comput. Phys. 207 (2005) 493.

[33] R. Cocle, G. Winckelmans, G. Daeninck, J. Comput. Phys. 227 (2008) 2263.

[34] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Heiber, E. M. Kostalis, P. Koumoutsakos, J. Comput. Phys. 215 (2006) 566.

[35] P. Chatelain, A. Curioni, M. Bergdorf, D. Rossinelli, W. Anderoni, P. Koumoutsakos, Comput. Methods Appl. Mech. Engrg. 197 (2008) 1296.

[36] C. A. White, M. Head-Gordon, J. Chem. Phys. 105 (1996) 5061.

[37] H. Cheng, L. Greengard, V. Rokhlin, J. Comput. Phys. 155 (1999) 468.

[38] M. A. Blanco, M. Flórez, M. Bermejo, J. Mol. Struct. 419 (1997) 19.

[39] C. G. Lambert, T. A. Darden, J. A. Board, J. Comp. Phys. 126 (1996) 274.

[40] M. Challacombe, C. White, M. Head-Gordon, J. Chem. Phys. 107 (1997) 10131.

[41] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, v. 2.0, (2008).
`http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`