Fast Multipole Methods on Graphics Processors

Nail A. Gumerov and Ramani Duraiswami
Perceptual Interfaces and Reality Lab.
Computer Science and UMIACS
University of Maryland, College Park
{gumerov,ramani}@umiacs.umd.edu
Fantalgo, LLC, Elkridge MD
info@fantalgo.com www.fantalgo.com

21 Nov 2007

Abstract

The Fast Multipole Method allows the rapid evaluation of sums of radial basis functions centered at points distributed inside a computational domain at a large number of evaluation points to a specified accuracy ϵ. The method scales as $O(N)$ compared to the direct method with complexity $O(N^2)$, which allows one to solve larger scale problems. Graphical processing units (GPU) are now increasingly viewed as data parallel compute coprocessors that can provide significant computational performance at low price. We describe acceleration of the FMM using the data parallel GPU architecture.

The FMM has a complex hierarchical (adaptive) structure, which is not easily implemented on data-parallel processors. We described strategies for parallelization of all components of the FMM, develop a model to explain the performance of the algorithm on the GPU architectures, and determined optimal settings for the FMM on the GPU, which are different from those on usual CPUs. Some innovations in the FMM algorithm, including the use of modified stencils, real polynomial basis functions for the Laplace kernel, and decompositions of the translation operators, are also described.

We obtained accelerations of the Laplace kernel FMM on a single NVIDIA GeForce 8800 GTX GPU in the range 30-60 compared to a serial CPU implementation for benchmark cases of up to million size. For a problem with a million sources, the summations involved are performed in approximately one second. This performance is equivalent to solving of the same problem at 24-43 Teraflop rate if we use straightforward summation.

1 Introduction

Fast Multipole Methods (FMM) were invented in the late 1980s [6], and have since been identified as one of the ten most important algorithmic contributions of the 20th century [4]. The FMM is widely used in many areas because of its ability to achieve linear time and memory dense matrix vector products to a fixed prescribed accuracy ϵ for problems arising in diverse areas (molecular dynamics, astrophysics, acoustics, fluid mechanics, electromagnetics, scattered data interpolation etc.).

Graphics Processing Units: A recent hardware trend is for the development of highly capable data-parallel processors, with their origin in the gaming and graphics industries. In 2007, while the fastest Intel CPU can only achieve ~20 Gflops speeds on benchmarks, the GPUs have speeds that are more than an order of magnitude higher. Of course, the GPU performs highly specialized tasks, while the CPU is a general purpose processor. Fig. 1 shows the relative abilities of GPUs and CPUs (on separate benchmarks) in early 2006. The trends reported are expected to continue for the next few years.

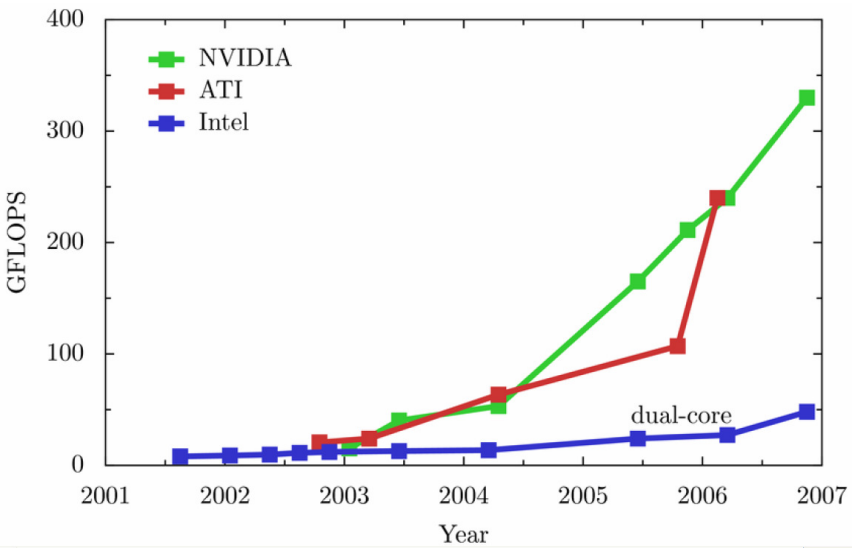

Figure 1: GPU and CPU growth in speed over the last 6 years.

While GPUs were originally intended for specialized graphics operations, it is often possible to perform general purpose computations on them, and this has been a vigorous area of research over the last few years. One of the problems that a number of researchers have tackled for speed up is the so called N body problem, in which the Coulombic (or gravitational) potentials and/or forces exerted by all charges (particles) in a system on each other are evaluated. This force calculation is part of a time stepping procedure in which the accelerations are evaluated, while the potential is necessary for energy computations. Recently several researchers have reported the use of GPUs, either in isolation or in a cluster to speed up these computations using direct algorithms, in which the interaction of every pair of particles is considered (see e.g., [20, 19, 16]). While impressive speedups are reported, these algorithms suffer from the $O\left(N^2\right)$ nature of the algorithm.

Efforts in Japan, under the auspices of the Grape project, have sought to develop special hardware, similar to the GPU, for performing N-body calculations as well (see e.g., [22, 23]). The substantial investments required to create special purpose hardware have meant that the progress in new generations has been slow (about 7 years since the previous Grape version was introduced). In contrast, the market driven development of the GPUs has lead to yearly introduction of new and improved hardware.

Present contribution: We map the complex FMM algorithm on to the GPU architecture. The idea to parallelize the FMM is not new and there are publications which address some basic issues [7, 13, 14, 15, 17]. However these publications are related to development of more coarse-grained parallel algorithms. We developed algorithms that work with the nonuniform memory access costs for the GPU global and local memory, and a data parallel architecture. We used the hierarchical nature of the FMM to balance cost trees, and achieve optimal performance. This property of the FMM is very important as the different parts of the FMM can be accelerated with different degrees of efficiency, and at optimal settings, the overall performance is controlled not only by the slowest step of algorithm. Finally, our implementation on the GPU also suggested several improvements to the CPU FMM algorithm, which are also described.

We implemented all the parts of the FMM algorithm on the GPU and obtained a properly scaled algorithm that requires times of the order of 1 s to compute the matrix vector product for a million sized problem on a current NVIDIA GeForce 8800GTX GPU. This corresponds to effective flop rates of 24-43 Tflops, following the formula given in [21, 19].

2 Fast multipole method (FMM)

While several papers and books describing the FMM are available, to keep the paper self contained and establish notation we provide a brief introduction here. The basic task that the FMM performs is computation of potentials and its gradients (forces) generated by a set of sources and evaluated at a set of evaluation points (receivers)

$$\phi_j = \phi\left(\mathbf{y}_j\right) = \sum_{i=1}^{N} \Phi\left(\mathbf{y}_j, \mathbf{x}_i\right) q_i, \quad j = 1, ..., M, \quad \mathbf{x}_i, \mathbf{y}_j \in \mathbb{R}^d, \tag{1}$$

where $\{\mathbf{x}_i\}$ are the sources, $\{\mathbf{y}_j\}$ the receivers, q_i strengths, and d is the dimensionality of the problem. It can also be used to compute the gradient

$$\nabla\phi_j = \left.\nabla_{\mathbf{y}}\phi\left(\mathbf{y}\right)\right|_{\mathbf{y}=\mathbf{y}_j} = \sum_{i=1}^{N} \left.\nabla_{\mathbf{y}}\Phi\left(\mathbf{y}, \mathbf{x}_i\right)\right|_{\mathbf{y}=\mathbf{y}_j} q_i. \tag{2}$$

Obvious examples of such problems include particle interaction, star motion, and molecular dynamics. At the same time this problem may be recognized as a matrix vector product, where the matrix is derived from a particular kind of function/potential or radial basis function. Discretization of integral equation formulations of common PDE's lead to linear systems with the same problem of summation of contribution of many sources and evaluation of them at the required locations. Iterative methods provide solution of these large systems.

The kernel, or elementary potential, is given by function $\Phi\left(\mathbf{y}_j, \mathbf{x}_i\right)$, and the FMM has been developed for several kernels. In this paper, in the sequel we will consider the Laplace kernel in 3D, for which

$$\Phi\left(\mathbf{y}, \mathbf{x}\right) = \frac{1}{|\mathbf{x} - \mathbf{y}|}, \quad \mathbf{x} \neq \mathbf{y}, \tag{3}$$

and zero for $\mathbf{x} = \mathbf{y}$. The theory of the FMM for this kernel is well developed starting from the original works of the Rokhlin and Greengard [6].

2.1 Outline of the FMM

The main idea of the FMM is based on performing the sum (1) via the decomposition

$$\phi_j = \sum_{i=1}^{N} \Phi_{ji} q_i = \sum_{\mathbf{x}_i \notin \Omega(\mathbf{y}_j)} \Phi_{ji} q_i + \sum_{\mathbf{x}_i \in \Omega(\mathbf{y}_j)} \Phi_{ji} q_i, \quad j = 1, ..., M, \tag{4}$$

where $\Omega\left(\mathbf{y}_j\right)$ is some domain (neighborhood) of the point $\mathbf{y}_j$. This also can be written in the matrix vector form as

$$\phi = \mathbf{\Phi}\mathbf{q} = \mathbf{\Phi}^{(dense)}\mathbf{q} + \mathbf{\Phi}^{(sparse)}\mathbf{q}. \tag{5}$$

The latter sparse matrix vector multiplication is performed directly. All other steps of the FMM are dedicated to computation of $\mathbf{\Phi}^{(dense)}\mathbf{q}$ and involve data structures, multipole and local expansions, and translation theory. For computation of gradient we have similar decomposition.

Despite the fact there are several modifications of the basic algorithm, it can be summarized as follows. Assume that the computational domain including all sources and receivers is included in a large cube (which by way the scaling can be made to provide the unit cube). This cube then is subdivided by an octree by dividing this box into 8 sub-cubes, and going down to level $l_{\max}$, so that at each level we have 8^l boxes

($l = 0, ..., l_{\max}$). Each box containing sources can be referred as a source box and each box containing receivers can be referred as a receiver box. Obviously, depending on the distribution the $l_{\max}$ boxes can contain different number of points, and some boxes may even be empty. The empty boxes are skipped at all levels, which provides basic adaptivity of the FMM for nonuniform distributions. The source boxes constitute the source hierarchy (parent-child relationships) and the receiver boxes form the receiver box hierarchy. The structuring of the initial source and receiver boxes can be done using spatial ordering (bit-interleaving technique, e.g. see [9]). This technique also allows one to determine neighbors for each box, which is important for the FMM. Also some necessary computations can be done before the run of the FMM (for example, precomputation of the elements of the translation matrices).

The FMM itself computes ϕ_j and $\nabla\phi_j$ for an arbitrary input vector $\{q_i\}$. This part can be repeated for the same source and receiver locations many times (e.g. for iterative solution of linear system (1), when the solution $\{q_i\}$ must be found for a given $\{\phi_j\}$). It consists of three basic steps: Upward pass, Downward Pass, and Final Summation.

2.1.1 Upward Pass

Step 1 For each box at level $l_{\max}$ generate multipole, or S expansion for each source at the box center. This expansion is a finite vector of expansion coefficients $\{C_n^m\}$ over some basis (usually over spherical basis functions). As function representation in this basis are infinite the series are truncated with truncation number p, so each box is represented by p^2 coefficients. The choice of p is dictated by the required accuracy and is at most determined by the length of floating point number representation (single or double precision). The expansion from all sources in the box are consolidated in to a single expansion.

Step 2 For levels from $l_{\max}$ to 2 generate S expansion for each box. This is achieved by application of the multipole-to-multipole (or $S|S$) translation operator to each box. The procedure goes in a hierarchical way from level to level, and any expansion for the parent source box is obtained from the S expansions of its children.

2.1.2 Downward Pass

Starting from level 2 down to level $l_{\max}$ for each receiver box generate its local, or R expansion. This is achieved by the following two step procedure:

Step 1 Translate S expansions from the source boxes of the same level belonging to the receiver box parent neighborhood, but not belonging to the neighborhood of the receiver box itself, to R expansions centered at the center of the receiver box (multipole-to-local, or $S|R$ translations). Consolidate the expansions.

Step 2 Translate the R expansion from the parent receiver box center to child box centers and consolidate them with the $S|R$ translated expansions at this level. This step starts from level 3 (or from level 2, if the R expansion for the parent box is initialized to zero).

2.1.3 Final Summation

Step 1 Evaluate the R expansion of each receiver box at level $l_{\max}$ at all receiver locations inside this box.

Step 1' If gradient computations are needed apply differential operators expressed in terms of expansion coefficients and efficiently obtain vector expansion coefficients for the gradient.

Step 2 For each receiver perform direct summation of the sources belonging to its box neighborhood and sum this up with the result of Step 1) of the Final summation. This step is nothing but sparse matrix vector multiplication. Note that the sparse matrix vector product is independent of the rest of the FMM and can be performed anytime.

This also can be represented as a flow chart 2.

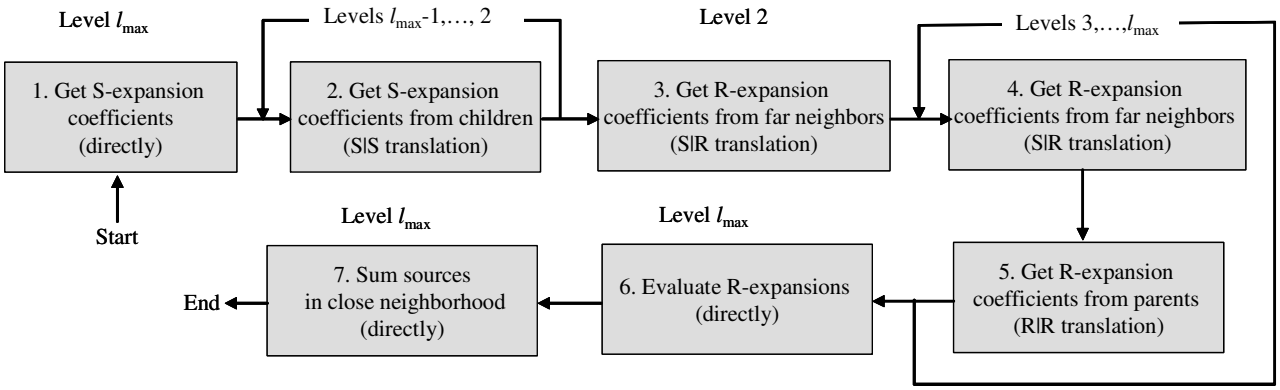

Figure 2: A flow chart of the standard FMM.

2.2 Characteristics of our FMM implementation

Different authors use different bases, translation, methods and schemes for the FMM for the Laplace equation. These methods have their own advantages and disadvantages and a particular method of choice depends on the problem to be solved, particularly the accuracy. Here we consider the FMM consistent with single precision float arithmetic on the GPU. While some software for double precision exist on the GPU, their use show a decrease of the computational speed up to 10 times, and while we will still see an acceleration relative to the CPU, this will not be as dramatic. GPU manufacturers envision in the closest future the release of GPUs with double precision hardware (both ATI and NVIDIA have announced this feature by early 2008). In this case the single precision algorithms can be modified accordingly and the fastest methods for high precision computations can be implemented and tested, without writing artificial libraries.

Thus, for the current state of the GPU computations with 3,4, or 5 digit accuracy are appropriate, which cover a broad class of practical needs. Our tests showed that computations with a truncation number $p = 16$ and higher using the 4 byte floats produce heavy loss of accuracy, overflows/underflows (due to summation of numbers of very different magnitude) and cannot be used for large scale problems. On the other hand, computations with relatively small truncation numbers, like $p = 4, 8$, and 12 are stable, and produced the required 3, 4 or 5 digits for million size problems, and we focused on this range of truncation numbers.

Our comparative study of various translation methods on the CPU [11] show that for this range standard multipole expansions and rotation-coaxial translation-rotation (RCR) decomposition of the translation operators provide the best results. The RCR decomposition introduced for the FMM in Ref. [18] has $O(p^3)$ complexity with low asymptotic constant for any translation operator and for p from the indicated range is as fast or faster than $O(p^2)$ methods with larger asymptotic constants and the size of the representing vectors (e.g. diagonal forms of the translation operators, [8]).

We also revised and modified the computationally expensive original $S|R$ translation scheme as described below and used normalized real basis functions that can be evaluated quickly.

2.2.1 Basis functions

Elementary solutions of the Laplace equation in three dimensions in spherical coordinates (r, θ, φ)

$$x = r\sin\theta\cos\varphi, \quad x = r\sin\theta\sin\varphi, \quad z = r\cos\theta, \tag{6}$$

can be represented as

$$R_n^m(\mathbf{r}) = \alpha_n^m r^n Y_n^m(\theta,\varphi), \quad S_n^m(\mathbf{r}) = \beta_n^m r^{-n-1} Y_n^m(\theta,\varphi), \quad n = 0,1,..., \quad m = -n,...,n. \tag{7}$$

Here $R_n^m(\mathbf{r})$ are respectively the regular (local) and $S_n^m(\mathbf{r})$ the singular (far field, or multipole) spherical basis functions, α_n^m and β_n^m are normalization constants selected by convenience, and $Y_n^m(\theta,\varphi)$ are the orthonormal spherical harmonics:

$$\begin{aligned} Y_n^m(\theta,\varphi) &= N_n^m P_n^{|m|}(\mu) e^{im\varphi}, \quad \mu = \cos\theta, \\ N_n^m &= (-1)^m \sqrt{\frac{2n+1}{4\pi}\frac{(n-|m|)!}{(n+|m|)!}}, \quad n = 0,1,2,..., \quad m = -n,...,n, \end{aligned} \tag{8}$$

where $P_n^{|m|}(\mu)$ are the associated Legendre functions [2]. We will use the definition of the associated Legendre function $P_n^m(\mu)$ that is consistent with the value on the cut $(-1,1)$ of the hypergeometric function $P_n^m(z)$ (see Abramowitz & Stegun, [2]). These functions can be obtained from the Legendre polynomials $P_n(\mu)$ via the Rodrigues' formula

$$P_n^m(\mu) = (-1)^m \left(1-\mu^2\right)^{m/2} \frac{d^m}{d\mu^m} P_n(\mu), \quad P_n(\mu) = \frac{1}{2^n n!}\frac{d^n}{d\mu^n}\left(\mu^2-1\right)^n. \tag{9}$$

Straightforward computation of these basis functions involves several relatively costly operations, such as conversion back and forth to spherical coordinates, and also the basis is complex, which is not needed for real function computation. One of the most convenient bases for translation was proposed by Epton and Dembart [5], which is basis (7) with

$$\begin{aligned} \alpha_n^m &= (-1)^n i^{-|m|} \sqrt{\frac{4\pi}{(2n+1)(n-m)!(n+m)!}}, \\ \beta_n^m &= i^{|m|} \sqrt{\frac{4\pi(n-m)!(n+m)!}{2n+1}}, \quad n = 0,1,..., \quad m = -n,...,n. \end{aligned} \tag{10}$$

Perhaps these functions are the best for complex computations, while for real functions we instead set

$$\begin{aligned} &\alpha_n^m = (-1)^n \sqrt{\frac{4\pi}{(2n+1)(n-m)!(n+m)!}}, \quad \beta_n^m = \sqrt{\frac{4\pi(n-m)!(n+m)!}{2n+1}}, \\ &n = 0,1,...., \quad m = -n,...,n. \end{aligned} \tag{11}$$

and defined real basis functions as

$$\widetilde{R}_n^m = \begin{cases} \mathrm{Re}\{R_n^m\}, & m \geqslant 0 \\ \mathrm{Im}\{R_n^m\}, & m < 0 \end{cases}, \quad \widetilde{S}_n^m = \begin{cases} \mathrm{Re}\{S_n^m\}, & m \geqslant 0 \\ \mathrm{Im}\{S_n^m\}, & m < 0 \end{cases}. \tag{12}$$

In these bases the regular (local) and singular (multipole) expansions of harmonic functions take form

$$\phi(\mathbf{r}) = \sum_{n=0}^{p-1}\sum_{m=-n}^{n} D_n^m \widetilde{R}_n^m(\mathbf{r}), \text{ or } \phi(\mathbf{r}) = \sum_{n=0}^{p-1}\sum_{m=-n}^{n} C_n^m \widetilde{S}_n^m(\mathbf{r}), \tag{13}$$

where coefficients are real and we truncated the infinite sums to work only with p^2 terms.

Note also that in an actual FMM implementation evaluation of the singular basis functions can be avoided, since the evaluation stage requires only the regular basis function, while the coefficients of the multipole expansion over the singular basis are also respectively normalized regular function,

$$G(\mathbf{r},\mathbf{r}_0)=\frac{1}{|\mathbf{r}-\mathbf{r}_0|}=\sum_{n=0}^{\infty}\sum_{m=-n}^{n}(-1)^n R_n^{-m}(\mathbf{r}_0)\,S_n^m(\mathbf{r})\,. \tag{14}$$

Dropping details of derivation, one can compute $\widetilde{R}_n^m(\mathbf{r})$ using the following recursions

$$\begin{aligned}
&\widetilde{R}_0^0=1,\quad \widetilde{R}_1^1=-\frac{1}{2}x,\quad \widetilde{R}_1^{-1}=\frac{1}{2}y, \\
&\widetilde{R}_{|m|}^{|m|}=-\frac{\left(x\widetilde{R}_{|m|-1}^{|m|-1}+y\widetilde{R}_{|m|-1}^{-|m|+1}\right)}{2\,|m|},\quad \widetilde{R}_{|m|}^{-|m|}=\frac{\left(y\widetilde{R}_{|m|-1}^{|m|-1}-x\widetilde{R}_{|m|-1}^{-|m|+1}\right)}{2\,|m|},\quad |m|=2,3,.... \\
&\widetilde{R}_{|m|+1}^{m}=-z\widetilde{R}_{|m|}^{m},\quad m=0,\pm1,...., \\
&\widetilde{R}_n^m=-\frac{(2n-1)z\widetilde{R}_{n-1}^m+r^2\widetilde{R}_{n-2}^m}{(n-|m|)\,(n+|m|)},\quad n=|m|+2,....,\quad m=-n,...,n.
\end{aligned} \tag{15}$$

These recursions show that functions $\widetilde{R}_n^m$ are real polynomials of degree n of Cartesian coordinates of $\mathbf{r}=(x,y,z)$ and to compute them there is no need to convert to spherical coordinates and back. Further their use avoids the need to use special functions with limited accuracy and longer clock cycles, a consideration that is important on the GPU, and to an extent on the CPU. Further, the algorithm does not need complex arithmetic.

2.2.2 RCR decomposition

Elementary regular and multipole (singular) solutions of the Laplace equation centered at a point can be expanded in series over elementary solutions centered about some other spatial point. This can be written as the following addition theorems

$$\begin{aligned}
R_n^m(\mathbf{r}+\mathbf{t})&=\sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(R|R)_{n'n}^{m'm}(\mathbf{t})R_{n'}^{m'}(\mathbf{r})\,, \\
S_n^m(\mathbf{r}+\mathbf{t})&=\sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(S|R)_{n'n}^{m'm}(\mathbf{t})R_{n'}^{m'}(\mathbf{r})\,,\quad |\mathbf{r}|<|\mathbf{t}|\,, \\
S_n^m(\mathbf{r}+\mathbf{t})&=\sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(S|S)_{n'n}^{m'm}(\mathbf{t})S_{n'}^{m'}(\mathbf{r})\,,\quad |\mathbf{r}|>|\mathbf{t}|\,,
\end{aligned} \tag{16}$$

where $\mathbf{t}$ is the translation vector, and $(R|R)_{n'n}^{m'm}$, $(S|R)_{n'n}^{m'm}$, and $(S|S)_{n'n}^{m'm}$ are the four index local-to-local, multipole-to-local, and multipole-to-multipole reexpansion coefficients. Explicit expressions for these coefficients can be found elsewhere (see e.g. [5, 3]). It is not difficult to show then that reexpansion of the basis produces matrix translation operator, which acts on the coefficients of function expansion and relates expansion coefficients in different bases with different centers. For example for multipole-to-local translation we have

$$D_n^m=\sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(S|R)_{nn'}^{mm'}(\mathbf{t})C_{n'}^{m'}, \tag{17}$$

where $C_{n'}^{m'}$ are coefficients of multipole expansion of function in some reference frame, and D_n^m are coefficients of expansion of the same function over the local basis in the reference frame with the origin shifted by vector $\mathbf{t}$.

Despite the fact that the coefficients of the translation matrix can be easily computed the translation operation converting p^2 to p^2 coefficients requires p^4 multiplications if applied directly. This number in the real basis can be reduced to $2p^3 + O(p^2)$ for the $S|R$ translation using the RCR decomposition (see [18] for an initial treatment, and [9, 11] for extensions). In this decomposition we first perform rotation of the reference frame to point axis z to the target, then perform coaxial translation along the z axis, and then rotate the reference frame back. Each operation has $O(p^3)$ complexity due to acts only on certain subspaces of the coefficient vector.

For coaxial translations we have

$$(S|R)_{nn'}^{mm'}(\mathbf{i}_z t) = \delta_{mm'}(S|R)_{nn'}^{m}(t),$$

where $\delta_{mm'}$ is the Kronecker symbol and expressions for the $S|S$ and $R|R$ translations are similar. In the basis we use the entries of the coaxial translation matrices can be easily computed as:

$$\begin{aligned}
(R|R)_{nn'}^{m}(t) &= r_{n'-n}(t), \quad n' \geqslant |m|, \\
(S|R)_{nn'}^{m}(t) &= s_{n+n'}(t), \quad n, n' \geqslant |m|, \\
(S|S)_{nn'}^{m}(t) &= r_{n-n'}(t), \quad n \geqslant |m|,
\end{aligned} \tag{18}$$

where the functions $r_n(t)$ and $s_n(t)$ are

$$r_n(t) = \frac{(-t)^n}{n!}, \quad s_n(t) = \frac{n!}{t^{n+1}} \quad n = 0, 1, ..., \quad t \geqslant 0, \tag{19}$$

and zero for $n < 0$.

The rotation transform is a bit more tricky, since in the normalized basis the rotation operators are different in the S and R bases. However multiplication of the rotation operators by diagonal matrices makes them the same in all bases, which can be used for more compact storage of the rotation data. The diagonals of these diagonal matrices consist of coefficients α_n^m for the R basis and β_n^m for the S basis (which is equivalent to the use of bases (7) with $\alpha_n^m = 1$ and $\beta_n^m = 1$. Note then that in this case the rotation operators are related to rotation of spherical harmonics, which is common for many equations, including the Laplace and Helmholtz equation. A detailed theory of fast recursive computation of the rotation coefficients and rotation can be found e.g. in our book [9] or technical report [11]. In the present study we use the methods described there with a small modification as the rotation operators act on real basis functions, as described above.

2.2.3 Translation stencil

One of the reasons for the high cost of the FMM multipole-to-local translation stage in the original algorithm was that the stencil used in the FMM required 189 such translations in 3D. In [3] not only were more efficient translation operators based on an exponential representation introduced, but a translation stencil that took advantage of symmetries and reduced the number of translations to ~40 were used.

As discussed earlier, for the accuracies that can be achieved on the GPU, the usual FMM representation in terms of multipoles and regular functions, along with the $O\left(p^3\right)$ translation operators are preferred. It turns out that we found some symmetries that allow the translations in the regular FMM stencil to be reduced. Consider now the neighborhood of the parent receiver box. There are 216 boxes on the level of the receiver box itself. S expansions from boxes which are not in the neighborhood of the receiver box should be translated to each box center (one can count that in this case we have 1512 translations, or 189 translations per receiver box, as the parent receiver box has 8 children). The remark is then that there exist 80 boxes (see

the 3D parent neighborhood in Fig. 3) from which the $S|R$ translations can be made not to the children, but to the parent box itself, while satisfying the specified error bound. Such translations can be consolidated with the R expansion at the parent box and Step 2 of the downward pass can be performed without any additional constraint. The saving here is that instead of $80\times 8 = 640$ translations (each source box to each receiver box) we have only 80 translations, which saves 560 translations out of 1512 for this domain, or effectively reduces 189 translations per box to 119 translations.

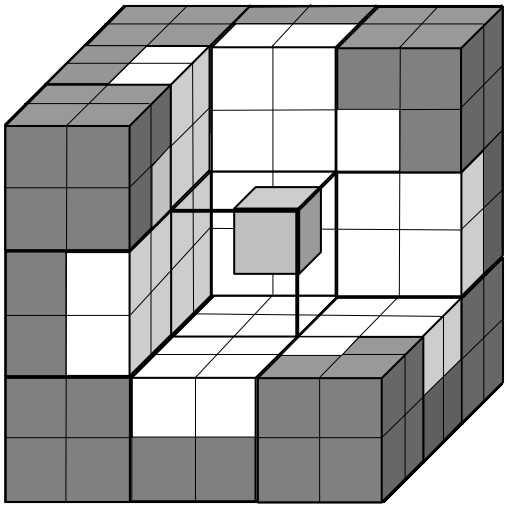

Figure 3: A section of the translation stencil used in the computations. Instead of translation to the central receiver box (the light gray shade) the source boxes shown in the dark gray can be translated to the center of the parent receiver box (the center of the stencil). Then this is information can be passed to each child box.

This is an example of a simple translation stencil, which amortizes the translation cost for boxes having the same parent. On the CPU we implemented and tested an even more advanced translation stencil which reduces almost twice even this number of translations (to 61 $S|R$ translations per box). However that stencil requires a more complicated internal data structure and the GPU code runs a bit slower than the one for the stencil described above, due to more extensive global memory access. So in this implementation we limited ourselves with the above simplification.

A theoretical justification of this scheme follows from the error bounds, which we provide below. Here we mention that introduction of the stencil modifies the 2-step downward pass of the FMM as follows (the white source boxes in Fig. 3 are referred as boxes of type 1, while the shaded source boxes are boxes of type 2).

Modified Downward Pass Starting from level 2 down to level $l_{\max}$ for each receiver box generate its local, or R expansion. This is achieved by the following procedure:

Step 1. Translate S expansions from the source boxes of type 1 belonging to the receiver stencil to the R expansions centered at the center of the receiver box (multipole-to-local, or $S|R$ translations). Consolidate the expansions.

Step 2. Translate S expansions from the source boxes of type 2 belonging to the receiver stencil to the R expansions centered at the center of the parent receiver box. Consolidate the expansions with the existing parent R expansion (zero for level 2).

Step 3. Translate the R expansion from the parent receiver box and consolidate with the $S|R$ translated expansions.

2.2.4 Variable truncation number

In the standard FMM for the Laplace equation the truncation number p is fixed for all translations/expansions. The choice of p is dictated by the error bound evaluated for the "worst" case, which corresponds to the $S|R$ translation from the source box of size 1 to the receiver box of size 1 which centers are separated by distance 2 (the Laplace equation is scale independent). Let us provide a bit more general look at the problem.

Assume that a set of sources is located within a d dimensional sphere of radius R_1, which we denote as Ω_1 and the set of evaluation points is located within a d dimensional sphere of radius R_2, which we denote as Ω_2. We assume that these domains do not intersect and denote the distance between the centers of these spheres as $D > R_1 + R_2$. Assume that the multipole expansions about the center of $\Omega_1$converges absolutely and uniformly and the same holds for the local expansions about the center of Ω_2. For the Laplace equation these series can be majorated by geometric progressions. More precisely, if the series are p truncated, then the error, ϵ_p, is bounded as

$$\epsilon_p < C\eta^p, \quad \eta\left(\Omega_1, \Omega_2\right) = \frac{\max(R_1, R_2)}{D - \min(R_1, R_2)}, \tag{20}$$

where C is some constant. Note that $\eta\left(\Omega_1, \Omega_2\right)$ is a purely geometrical quantity, which characterizes the multipole-to-local translation. If $\eta \geqslant 1$ this means that the intersection of domains Ω_1 and Ω_2 is not empty and the expansions diverge due to singularity in the domain of local expansion. So the error due to single source will be a function of η and given $\eta < 1$ determines the number of terms to guarantee the accuracy. Larger η, more terms are needed for a specified error.

The FMM errors for the kernels under consideration are determined by the errors of the multipole-to-local translations. So for the kernels considered, when the same length series (truncation numbers) can be used for all types of translations for all levels, the only criteria, which can be checked to stay within the specified error bounds is compliance with the criterion

$$\eta\left(\Omega_1, \Omega_2\right) \leqslant \eta_*\left(d\right). \tag{21}$$

The quantity η_* depends on the dimension d. In the standard translation scheme, the distance between the centers of the closest domains (or centers of the cubic boxes of unit size separated by one box) is $D = 2$, while the radii of the minimal spheres which include the source and receiver boxes are $R_1 = R_2 = \frac{1}{2}\sqrt{d}$. So the standard η_* which determines the length of expansion for a specified error is

$$\eta_*\left(d\right) = \frac{d^{1/2}}{4 - d^{1/2}}, \quad \eta_*\left(3\right) \approx 0.7637. \tag{22}$$

The translation stencil scheme suggested above then can be justified, as in this case we have $R_2 = \sqrt{3}$, $R_1 = \frac{1}{2}\sqrt{3}$ and one can check directly that any gray box in Fig. 3 has a distance from the stencil center greater than

$$D_* = \min(R_1, R_2) + \frac{\max(R_1, R_2)}{\eta_*\left(3\right)}. \tag{23}$$

Consider now general case which includes both the white and gray boxes in Fig. 3. To provide the same error as for the closest boxes, it is sufficient to use truncation number

$$p' = \frac{\ln \epsilon_p - \ln C}{\ln \eta},$$

which follows from the first Eq. (20). On the other hand for given p we have

$$p = \frac{\ln \epsilon_p - \ln C}{\ln \eta_*}.$$

Therefore, the truncation number providing the same error can be determined from

$$p' = p\frac{\ln \eta_*}{\ln \eta} = p \ln \eta_* / \ln \frac{\max(R_1, R_2)}{D - \min(R_1, R_2)}. \tag{24}$$

As parameters p, η_*, and R_1 are fixed we can for any $S|R$ translation determine the effective truncation number p', which does not exceed p and, therefore the function representation of length p'^2 can be obtained from the similar representation of length p^2 by a simple additional truncation. We found that such additional truncations bring substantial computational savings without substantial loss of accuracy.

In fact this algorithm is to be preferred in the sense that the errors achieved while remaining smaller than the user specified ϵ,are closer to it, and in this sense wasteful computations are not performed.

2.3 Computation of gradient

As soon as expansion coefficients D_n^m for a given box are found, we have for the gradient of the far field (see the first Eq. (13)):

$$\nabla\phi(\mathbf{r}) = \sum_{n=0}^{p-1}\sum_{m=-n}^{n} D_n^m \nabla \widetilde{R}_n^m(\mathbf{r}) + error(p) = \sum_{n=0}^{p-1}\sum_{m=-n}^{n} \mathbf{D}_n^m \widetilde{R}_n^m(\mathbf{r}) + error(p), \tag{25}$$

where $\mathbf{D}_n^m$ is a real three-dimensional vector, which x, y, and z components correspond to the components of the gradient. It is clear that such representation should hold since the derivatives of the harmonic functions are also harmonic functions expandable over the same basis. Explicit expressions for $\mathbf{D}_n^m$ can be obtained from the recursions for local basis functions R_n^m – a method described in [9]. Such recursions can be found elsewhere (e.g. [5, 10]). Note however, that here one should convert these formulae to real basis according to definitions (7), (11), and (12).

2.4 Computational complexity

The relative costs of the different steps of the FMM depend on the source and receiver distributions, truncation number p (or accuracy of computations), and on $l_{\max}$. To simplify the analysis and provide benchmark results we constrain ourselves to cases where N sources and M receivers are randomly (uniformly) distributed inside a unit cube (usually we selected M to be of the same order of magnitude as N, so in the complexity estimates one can assume $M \approx N$).

The main optimization of the FMM is based on proper selection of the depth of the octree, or $l_{\max}$. For a given size of the problem, N, it is convenient to introduce the clustering parameter, s, which is the number of sources (and for our case receivers) in the smallest box. In this case the computational costs for the sparse and dense matrices are

$$\text{Cost}^{(dense)} = N\left(A_1 + \frac{A_2}{s}\right), \quad \text{Cost}^{(sparse)} = B_1 N s, \quad s = N \cdot 8^{-l_{\max}}, \tag{26}$$

where A_1 and A_2 are some constants depending on the translation scheme and prescribed accuracy, B_1 is a constant determined by the size of the neighborhood and the computational complexity of function $\Phi(\mathbf{y}, \mathbf{x})$. The cluster size s must be chosen to minimize the overall algorithm cost.

If the algorithm is implemented on a single processor the CPU time required for execution of the task can be evaluated as

$$\text{Time}^{(sngl)} = C \cdot \left(\text{Cost}^{(dense)} + \text{Cost}^{(sparse)} \right) = CN \left(A_1 + \frac{A_2}{s} + B_1 s \right), \tag{27}$$

where C is some constant, and the optimal clustering parameter and maximum level of space subdivision can be found easily by differentiation of this function with respect to s:

$$s_{opt} = \left(\frac{A_2}{B_1} \right)^{1/2}, \quad l_{\max}^{(opt)} = \log_8 \frac{N}{s_{opt}}. \tag{28}$$

This provides optimal time

$$\text{Time}_{opt}^{(sngl)} = CN \left[A_1 + 2 \left(A_2 B_1 \right)^{1/2} \right], \tag{29}$$

which also shows that in the case when constants A, B, and C do not depend on N, the optimal algorithm scales as $O(N)$.

Note that normally $A_1 \ll (A_2 B_1)^{1/2}$ (A_1 is determined by the Step 1) of the Upward Pass and Step 1) of Final Summation, while the most computational work in the FMM is related to the $S|R$ translations). In this case the optimal settings and the total optimal time depend on A_2, B_1 and the processor (C).

3 NVIDIA G80 series GPU Architecture

The NVIDIA G80 GPU, the one on which we developed our software, is the current generation of NVIDIA GPU, and has also been released as the Tesla compute coprocessor. It consists of a set of multiprocessors (16 on our GeForce 8800GTX), each composed of 8 processors. All multiprocessors talk to a global device memory, which in the case of our GPU is 768MB, but can be as large as 1.5 GB for more recently released GPUs/coprocessors. The 8 processors in each multiprocessor share 16 kB local read-write "shared" memory, a local set of 8192 registers, and a constant memory of 64kB over all multiprocessors, of which 8kB can be cached locally at one multiprocessor.

A programming model (Compute Unified Device Architecture or CUDA) and a C compiler (with language extensions) that compiles code to run on the GPU are provided. Under CUDA the GPU is a compute device that is a highly multithreaded coprocessor, composed of several multiprocessors, each composed of several processors. A thread block is a batch of threads that executes on a multiprocessor that have access to its local memory. They perform their computations and become idle when they reach a synchronization point, waiting for other threads in the block to reach the synchronization point. Each thread is identified by its thread ID (the number within the block). The thread index is used to map the different pieces of data to the thread. The programmer writes data parallel code, which executes the same instructions on different data, though some customization of each thread is possible based on different behaviors depending on the value of the thread index.

To achieve efficiency on the GPU, algorithm designers must account for the substantially higher cost (two orders of magnitude higher) to access fresh data from the GPU main memory. This penalty is paid for the first data access, though additional contiguous data in the main memory can be accessed cheaply after this first penalty is paid. An application that achieves such efficient reads and writes to contiguous memory is said to be *coalesced*. Thus programming on the nonuniform memory architecture of the GPU requires that each of the operations be defined in a way that ensures that main memory access (reads and writes) are minimized, and coalesced as far as possible when they occur.

We have developed an extensible middleware library [12] that works with NVIDIA's CUDA to ease the burden of programming on the GPU, and allows writing code in Fortran 9x. The FMM was programmed using this library and a few functions written in CU (Nvidia's extended C).

4 FMM on GPU

Peculiarities of the architecture, memory constraints and bandwidth, and relative costs of the memory access and various arithmetic operations drastically change the complexity model and cost balance of the FMM on GPU compared with that on CPU. Our study shows that on the GPU the performance is limited mostly by some intrinsic balance for efficient sparse matrix vector multiplication, and in lesser degree by the balance of the $S|R$ translations and direct summations, which in contrast is the sole criterion for optimal performance on the CPU.

4.1 Direct summation baseline

First, let us consider a direct method for solution of problem (1), i.e. computation of the sum without the FMM as a baseline for comparison. In examples of efficient use of CUDA the NVIDIA provides a routine for dot product, which can be used as a part of matrix vector product. However, our experiments with this routine showed that it is actually designed for a given, or precomputed matrix. This is not the case of the FMM, where the matrix entries are computed "on the fly" as needed, which does not require large storage or additional memory access.

Our algorithm is designed in the following way.

1. All source data, including coordinates and source intensities are stored in a single array in the global GPU memory (the size of the array is $4N$ for potential evaluation and $7N$ for potential/force evaluation). Two more large arrays reside in the global memory, one with the receiver coordinates (size $3M$), and the other is allocated for the result of the matrix vector product (M for potential computations and $4M$ for potential and force computations). In case the sources and receivers are the same, as in particle dynamics computations, these can be reduced.

2. The routine is executed in CUDA on a one dimensional grid of one dimensional blocks of threads. By default each block contains 256 threads, which was obtained via an empirical study of optimal thread-block size. This study also showed that for good performance the thread block grid should contain not less than 32 blocks for a 16 multiprocessor configuration. If the number of receivers is not a divisor of the block size, only the residual number of threads is employed for computations of the last block.

3. Each thread in the block handles computation of one element of the output vector (or one receiver). For this purpose a register for potential calculation (and three registers for calculation of forces) are allocated and initialized to zero. The thread can take care on another receiver only after the entire block of threads is executed, threads are synchronized, and the shared memory can be overwritten.

4. Each thread reads the respective receiver coordinates directly from the global memory and puts them into the thread stack.

5. For a given block the source data are executed in a loop by batches of size B floats, where B depends on the type of the kernel (4 or 7 floats per source for monopole or monopole/dipole summations, respectively) and the size of the shared memory available. Currently we use 8 kB, or 2048 floats of the shared memory (so e.g. for monopole summation this determines $B = 512$). If the number of sources is not a divisor of B the last batch takes care on the residual number of sources.

6. All threads of the block are employed to read source data for the given batch from global memory and put them into the shared memory (consequent reading: one thread per one float, until the entire batch is loaded followed by thread synchronization).

7. Each thread computes product of one matrix element (kernel) with the source intensity (or in the case of dipoles the dipole moment with the kernel) and sums it up with the content of the respective summation register.

8. After summation of contributions of all sources (all batches) each thread writes the sum from its register into the global memory.

Table 1. Direct sum (1) on CPU and GPU			
N	Serial CPU(s)	GPU(s)	Ratio
2048	2.02E-01	6.19E-04	326
8192	3.23E+00	6.87E-03	471
32768	5.13E+01	8.98E-02	572
131072	8.17E+02	1.36E+00	603

Table 1 shows some comparative performance results for solution of the same benchmark problem (N random sources of random intensities $q \in (0, 1)$, and $M = N + 1$ random receivers) using a serial CPU programming and the GPU (everything in single precision). It is important to note that we used no optimizations afforded by our Intel CPU (SSE extensions and parallelization), and the results for the CPU are not meant to indicate the best CPU performance achievable. Rather, our goal was to compare with the simplest baseline case on the CPU. The same applies to all other comparisons with CPU results in the subsequent sections. While in a sense this is unfair to the CPU implementations, our goal is not to perform a "competition" between the two architectures, but rather use the CPU implementation to guide GPU code development.

The table and the graph plotting the table data (see 4) show that the timing of the CPU is consistent with the theoretical dependence, Time $= AN^2$, while the GPU timing approaches quadratic dependence in an asymptotic way and only large enough N are consistent with this dependence. This is also clear from the time ratio between the CPU and GPU processes shown in the table, which stabilizes at relatively large values of N.

This result is explainable, based on the algorithm we used. Indeed, a full GPU load is achieved only when the number of blocks exceeds 32. Taking into account that in our case each block contains 256 receivers this results in the size of the receiver set $M \geqslant 8192$ for full load. Second, as we mentioned, there exists an overhead in computational cost related to the read/write operations. This overhead is proportional to the number of sources and receivers. In other words, the read/write time is scaled linearly with N. Third, some time, which is much larger, than the time of a single arithmetic operation is needed to access the first pointer in the array of receivers, sources, and the output vector. Assuming that a coalesced reading/writing is realized, we should simply add some constant to the execution time. With this reasoning the model of GPU timing for direct summation can be written as

$$\text{Time} = AN^2 + BN + C, \tag{30}$$

where A, B, and C are some constants. Assuming that this model applies for $N \geqslant 8192$ we took three values from the table at $N = 8192, 32768$, and 131072 and determined these constants. The resulting fit line is shown in Fig. 4. It is seen that this fit can be continued even below 8192 and is close enough to the observed time.

4.2 Sparse matrix vector multiplication

The sparse matrix vector multiplication is one of the most time consuming parts of the FMM. Moreover, the performance achieved for this part can be applied to acceleration of the entire FMM algorithm by proper selection of the maximum level of the octree space subdivision. The algorithm which we used is in many

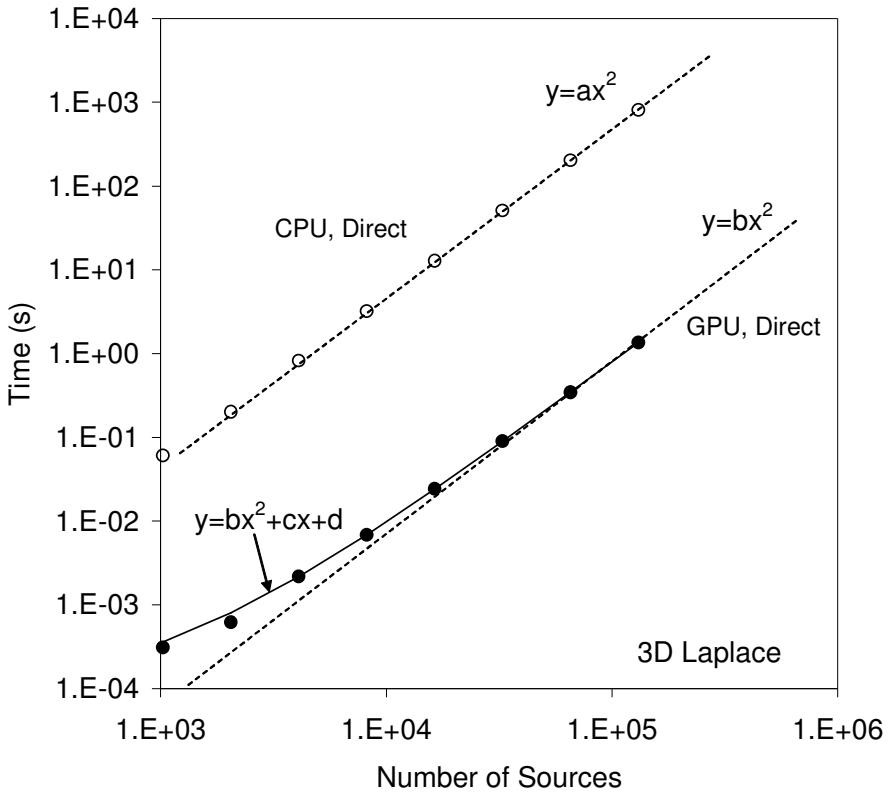

Figure 4: Comparison of the time required for solution of the same problem (direct summation (1)) using a serial CPU code and the GPU. N sources and $M = N + 1$ receivers are unstructured sets of random data.

aspects similar to the total direct summation algorithm described above, but with several necessary modifications, which are dictated by the FMM structure. These modifications are important enough, to consider their impact on the algorithm performance.

Consider the direct summation step in the FMM. Let us take some receiver box b_r at level $l_{\max}$ containing r receivers. The neighborhood of this box, if it is located not on the domain boundary consists of at most 27 non-empty source boxes, b_s. Assume for simplicity that each of these boxes contains approximately s sources. Therefore to obtain the required r sums for box b_r we need to perform multiplication of matrix of size at most $r \times 27s$ by a vector of length at most $27s$.

The hierarchical spatial ordering based on bit-interleaving, which we use provides that all receivers and sources in a given box are allocated in the memory contiguously. In other words, if we have a list of sources, then to access data for a given box b_s we just need to pass a pointer corresponding to box b_s and the amount of data in the box. Instead of this we can also use a bookmark index which says that the source indices for a given box are located between corresponding elements in a long array. The same relates to the receiver boxes.

Since the neighbor relation is not hierarchical (like child-parent) an additional data structure which lists all the source box neighbors for a given receiver box (our algorithm is adaptive, and, in fact the number of occupied source boxes in the neighborhood can be an arbitrary number from 0 to 27). Data for up to $27s$ sources required for one receiver box execution are allocated in the memory not in subsequent way, but by pieces, each of which needs a pointer.

Main modifications of the execution model on the GPU then appear to be as follows.

1. Each block of threads executes one receiver box (Block-per-box parallelization).
2. Each block of threads accesses data structure.
3. Loop over all sources contributing to the result requires partially noncoalesced read of the source data (box by box).

These features substantially slow down the performance. Indeed due to these modifications not all the threads in the block are employed, or they may be employed unequally (the number of receivers varies from box to box). The requirement that the receiver box is executed by a block is dictated by the necessity to put source data required for processing of this block into the shared memory and employ as many threads as possible for data read operation. Also only receivers of the same box share the same set of sources. As we mentioned due to non-hierarchical nature of the neighbor relation access to any box should be considered as random, and these reads each pay a penalty of initial access per box.

4.2.1 Model for run times of the sparse matrix vector product on the GPU

To develop a theoretical performance model, let us look at actual run times for the sparse matrix vector multiplier for problem (1). First, let us take clustering parameter s, or $l_{\max}$ which are optimal for the CPU codes ($p = 8$ and $p = 12$ for which we set data structures with the requirement $s \leqslant s_{\max} = 60 - 80$) and are results of the sparse and dense matrix vector cost balance (since the level changes discretely over integers, in practice the balance is approximate). Fig. 5 shows the performance for this case.

Two things can be noticed immediately. First that the GPU acceleration of the FMM compared to the CPU is very nonuniform. It varies by an order of magnitude for the same $l_{\max}$. On the other hand it is not difficult to see that the GPU timing for a given $l_{\max}$ changes slowly with increase of data size. In theory the complexity of the matrix vector product for fixed $l_{\max}$ should grow quadratically with N, which is perfectly reflected in the CPU timing. The "nonuniform" acceleration in this case is because on the CPU performance is determined by the data size, while on the GPU it is determined by the depth of the octree ($l_{\max}$).

Eq. (26, right) provides the time for the CPU implementation, where the cost of data access is asumed negligible. This dependence shows that at fixed $l_{\max}$ the time is proportional to N^2, which is in a very good agreement with the experimental results. However for the GPU realization the time appears to be a different function of N^2. It is more or less clear that this is due to the substantial costs of the data access operations. Since the algorithm is executed box by box, where the number of boxes is a function of $l_{\max}$ the number of random access operations is also is a function of $l_{\max}$. The number of boxes is ~$8^{l_{\max}}$, or N/s In this case we can evaluate the GPU time as follows.

$$\text{Time} = ANs + B\frac{N}{s} + CN, \quad s = N \cdot 8^{-l_{\max}}. \tag{31}$$

The first term in this formula represents the number of arithmetic operations and similar to (26, right). The second term represents the number of boxes and characterizes the time for random memory access. The last term here is the cost of reading/writing of data sets of characteristic length N in a sequential manner or fast access to the shared memory or other coalesced read/write. Model constants can be evaluated from three experimental data points and the fit is plotted in figure Fig. 5. Note that for this fit we determined the constants only from three data points for level $l_{\max} = 5$. However, the fit works nicely for the maximum subdivision levels 4 and 3. Some deviation is observed for level 2, which has an explanation that for these levels the data size becomes too small to provide a good GPU load.

The nonnegligible magnitude of the second term in the right hand side of Eq. (31) has far-reaching consequences. Indeed, if Eq. (26, right) shows that to reduce the time for this FMM step for fixed N one

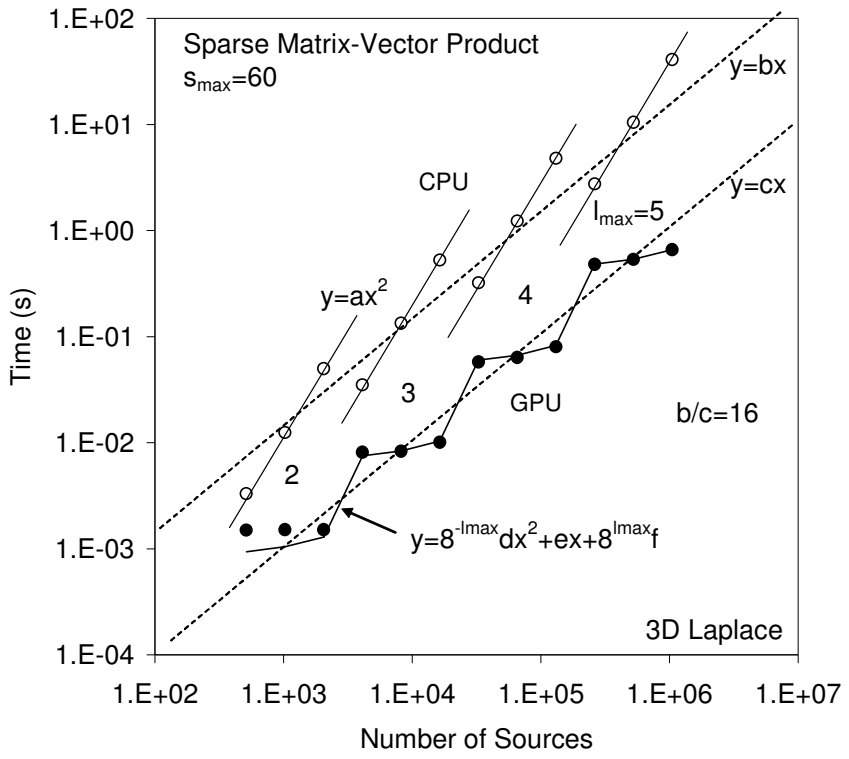

Figure 5: Performance of sparse matrix-vector multiplier used in the FMM for maximum cluster parameter $s_{\max} = 60$. The numbers 2,...5 near the curves show the maximum level of octree space subdivision $l_{\max}$. Solid and dashed lines show different data fits.

should decrease s (or increase $l_{\max}$) as much as possible, then Eq. (31) claims that substantial decreasing or increasing of s will increase the computation time, and there exists a unique minimum of function Time(s). This minimum is realized at $s = s_{opt}^{(sparse)}$ and the optimum time is $\text{Time}_{opt}^{(sparse)} = \text{Time}(s_{opt}^{(sparse)})$, which can be expressed in terms of the hardware and algorithm related constants A, B, and C as

$$s_{opt}^{(sparse)} = \left(\frac{B}{A}\right)^{1/2}, \quad \text{Time}_{opt}^{(sparse)} = \left[2\,(AB)^{1/2} + C\right] N. \tag{32}$$

The latter equation also shows that the optimal time for the algorithm is linear in N.

The estimate of fitting constants, which we obtained shows that $s_{opt}^{(sparse)} = 91$, which is larger than the size of the maximum cluster used for the above computations. We note then that the octree is a discrete object and the number of levels is an integer and if we build any continuous model approximation and find the optimum, this optimum cannot be achieved exactly. So the only way to change the performance for a given N is to change $l_{\max}$ (or sufficiently increase $s_{\max}$). To check these predictions we performed a run for larger $s_{\max}$ $(= 320)$ to see how the performance changes.

Figure 6 demonstrates one order of magnitude of increase of the relative GPU performance. Indeed the CPU timing is proportional to the number of arithmetic operations. For large clusters, of course, this number increases. However the GPU time decreases (compare with 5). Despite this decrease is not huge, but the density of arithmetic operations increases and the Gflops count increases at least by one order of magnitude. One can treat this as an amortization of the random global memory access cost, or simply look at formula (31).

It is seen that the qualitative behavior of the CPU time remains the same and is described by Eq. (26, right) well. The qualitative behavior of the GPU time however changes, and the step function of Fig. 5 turns into almost linear dependence on N. (It is interesting to note that the cost model for the GPU sparse matrix product is qualitatively similar to the cost model for the entire FMM – compare Eqs (27) and (31)). We also plotted the fit (31) with the same A, B, and C as for Fig. 5. It is clear that it is consistent with the large cluster case as well. Also the closeness of the GPU time to the linear dependence in this case witnesses that the selection of the cluster size is close to optimal (for optimal size the dependence is linear, see Eq. (32)). Further increase of the cluster size ($l_{\max}$ reduction) is not beneficial for performance. The GPU dependence qualitatively becomes similar to the CPU dependence and the execution time increases (see Eq. (31)). It is also interesting that cluster sizes, which we consider as optimal (~256) are also optimal for the thread block size of the GPU. Also the maximum number threads in the block on the processor we used is limited by 768, for which both performance of the GPU downgrades and imbalance between the random global memory access costs and arithmetic complexity becomes substantial.

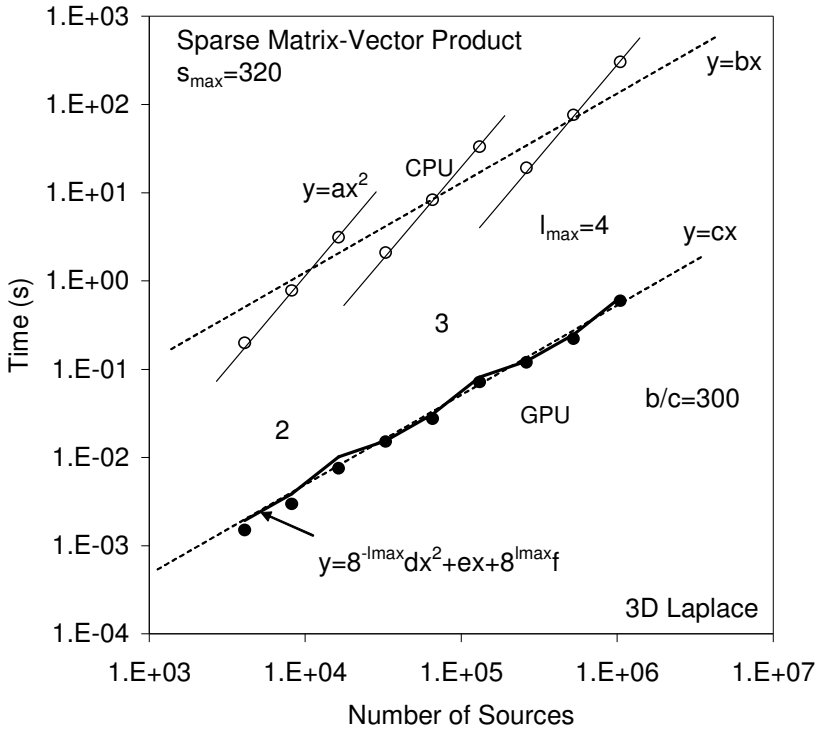

Figure 6: The same as Fig. 5, but for $s_{\max} = 320$.

We note then that for a given algorithm for sparse matrix vector multiplication and hardware, which determine the constants A, B, and C in Eq. (32) we have a minimum possible time for the entire FMM. Indeed any other FMM procedure can only increase the total time, and in any case it will be larger than $\text{Time}_{opt}^{(sparse)}$ given by Eq. (32).

Also due to different complexity dependences for the CPU and GPU to be fair we should rather compare the best performing cases for each type of architecture. Table 2 shows the time ratio for the best performing cases ($p = 8$ and 12). Again, as mentioned previously, the CPU times are for a serial implementation that does not take advantage of architecture specific vectorization instructions available, such as SSE.

Table 2. Sparse m/v mult. at best settings on CPU/GPU			
N	CPU(s)	GPU(s)	Ratio
32768	3.22E-01	1.51E-02	21
65536	1.23E+00	2.75E-02	45
131072	4.81E+00	7.13E-02	68
262144	2.75E+00	1.20E-01	23
524288	1.05E+01	2.21E-01	47
1048576	4.10E+01	5.96E-01	69

4.3 Other FMM subroutines

Below we briefly describe each routine performing the other steps of the FMM and provide some data on its performance.

4.3.1 Multipole expansion generator

The purpose of this subroutine is to take as input data on sources and generate S expansion for each source box at the finest level, which is the Step 1) of the FMM Upward Pass. As we mentioned above this procedure is equivalent to generation of the R basis for each source in the given box followed by consolidation of all expansions. The R basis is generated by the recursions (15) and can be performed for each source independently. So one solution for parallelization is to assign each thread to handle one source expansion.

Some drawback of this method is that after generation of expansions they need to be consolidated, which will cause data transfer. For efficiency, they should form a block of threads handled by one processor. The block size for execution of any subroutine in GPU can be defined by user, but it is fixed. In the FMM we face the situation where each box may have different number of sources. So if a source box is handled by a block of threads then some threads should be idled, which of course reduces the work load. We tried this approach and found that GPU accelerations compared to the serial CPU code in this case are in range 3-7, which appear to be rather low compared with performance of other routines.

The efficiency of the S expansion generator substantially increases if we use a different parallelization model: one thread per box. In this case one thread performs expansion for each of s sources in the box and consolidates these expansion. So one thread produces full S expansion for the entire box. The advantage of this approach is that in this case the work of each thread is completely independent and so there is no need for shared memory. This perfectly fits the situation when each box may have different number of sources, as the thread which finishes work for a given box simply takes care on the other box, without waiting or synchronization with other threads. The disadvantage of this approach is that to realize the full GPU load the number of boxes should be sufficiently large. Indeed, if an optimal thread block size is 256 and there are 16 multiprocessors (so we need at least 32 blocks of threads to realize an optimal GPU load), then the number of boxes should be at least 8192 for a good performance. Note that $l_{\max} = 4$ we have at most 8^4 =4096 boxes, and for $l_{\max} = 5$ this number becomes 8^5 =32768 boxes. So the method can work in full power only for large enough problems.

Table 3. Performance of S expansion generator ($p = 12$)						
	$s_{\max} = 60$			$s_{\max} = 320$		
N	Ser CPU(s)	GPU(s)	Time	Ser. CPU(s)	GPU(s)	Ratio
16384	2.81E-02	4.22E-03	6.7	2.78E-02	2.34E-02	1.2
131072	2.29E-01	5.93E-03	38.6	2.23E-01	2.52E-02	8.9
262144	4.96E-01	1.73E-02	28.6	4.53E-01	1.05E-02	43.3
1048576	1.92E+00	4.61E-02	41.6	1.81E+00	3.81E-02	47.4

We implemented and tested this subroutine and results for $p = 12$ are presented in Table 3. The performance on the CPU does not depend on the number of boxes and is a linear function of N. On the other hand the performance of GPU in this range of N and cluster sizes is heavily influenced by the number of boxes, or $l_{\max}$. This fact is due to the algorithm we used, and as we explained above a full GPU load can be achieved by this algorithm only at levels $l_{\max} \geqslant 4$. For $s_{\max} = 320$ criterion $l_{\max} \geqslant 4$ is achieved only for $N \geqslant 262144$ and after that one can expect more or less linear dependence of the GPU time on N.

Of course, there are opportunities to improve performance for $l_{\max} < 4$. However we note that the time spent for the S expansion generation usually does not exceed a couple of percent of the overall FMM run time and also the FMM is efficient for relatively large problems.

4.3.2 Local expansion evaluator

This subroutine realizes the Step 1) of the final summation. The reason why we describe it here out of order is that it is very similar to the S expansion generator discussed above. For parallelization of this routine we use the one thread per box strategy. The performance of this subroutine is approximately the same as of the S expansion generator. Finally we mention that both S and R subroutines described above use fast (diagonal) renormalization of the expansion, which provides more compact storage of the rotation operators.

4.3.3 *LevelUp*

The subroutine *LevelUp* is the core of the Step 2 in the FMM Upward Pass. It takes as input S expansions for all source boxes at level l and produces expansions for all source boxes at level $l-1$. Being called in a loop for $l = l_{\max}, ..., 3$ it completes the full Upward Pass of the FMM. The kernel of routine *LevelUp* consists of the $S|S$ translator, which takes as input some S expansion and produces a new (translated) S expansion. To produce the S expansion for the parent box all $S|S$ translated expansions from its children boxes should be consolidated.

We developed several parallel versions of this basic subroutine, which have some advantages and disadvantages, and the version which is described below showed slightly better performance compared to other versions we currently have. This version is based on the fact that the resulting S expansions for the parent boxes can be generated independently. So each thread can take care on one parent box. However, the work load of the GPU in this case becomes very small for low $l_{\max}$ and more or less reasonable speedup can be achieved only if several threads are allocated to process a parent box. Since each parent box in the octree has at most 8 children and for each child $S|S$ translation can be performed independently, we used two dimensional 32× 8 blocks of threads and one dimensional grid of blocks. In this setting each parent box was served by 8 threads, with the thread id in y varying from 0 to 7 for identification of the child boxes.

As we mentioned above we use the RCR decomposition of translation operators. In fact each rotation is decomposed further into α and β rotations, corresponding to the respective Euler angles. So the translation matrix $(\mathbf{S}|\mathbf{S})(\mathbf{t})$ in our case is decomposed as

$$(\mathbf{S}|\mathbf{S})(\mathbf{t}) = \mathbf{A}^{-1}(\alpha)\,\mathbf{B}(\beta)\,\underline{(\mathbf{S}|\mathbf{S})}(t)\,\mathbf{B}(\beta)\,\mathbf{A}(\alpha), \tag{33}$$

where (t, β, α) are the spherical coordinates of the translation vector $\mathbf{t}$, while $\mathbf{A}$ and $\mathbf{B}$ are the rotation matrices and $\underline{(\mathbf{S}|\mathbf{S})}$ is the coaxial translation matrix (see also Fig. 7). Peculiarity of the $S|S$ translation is that all translations for a given level have the same length t, there are only two different angles β, and four different α (in Fig. 7 only one translation is shown, while 7 other translations have symmetric properties). In addition there is a central symmetry of the translation operator:

$$(\mathbf{S}|\mathbf{S})(-\mathbf{t}) = \mathbf{A}^{-1}(\alpha)\,\mathbf{FB}(\pi-\beta)\,\underline{(\mathbf{S}|\mathbf{S})}(t)\,\mathbf{B}(\pi-\beta)\,\mathbf{FA}(\alpha), \tag{34}$$

where $\mathbf{F}$ is a very simple constant diagonal flip operator. This allows to use only one constant matrix $\mathbf{B}\,(\pi/4)$ for all $S|S$ translations. We use these facts for data compression, which allows us to put dense constant matrices $\underline{(\mathbf{S}|\mathbf{S})}\,(t)$ and $\mathbf{B}\,(\pi/4)$ into the fast (constant or shared) memory. Also operator $\mathbf{A}\,(\alpha)$ is diagonal and for its computation only angle α is needed, so there is no cost to read or write this operator to the global memory.

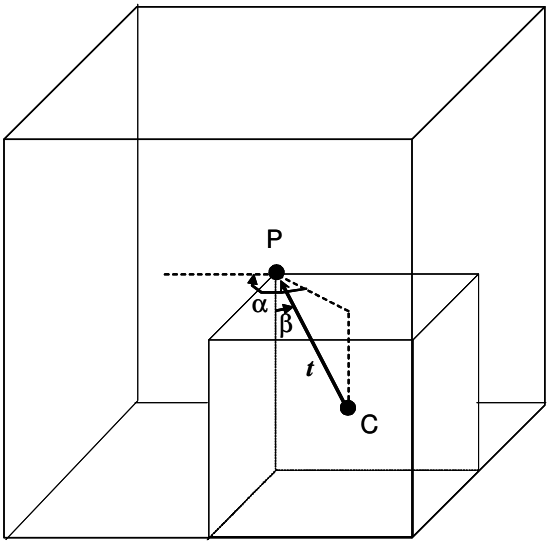

Figure 7: Translation from the center of a child box (C) to its parent (P). α and β are Euler rotation angles and $\mathbf{t}$ is the translation vector.

Operations $\mathbf{A}\,(\alpha)\,\phi$, $\mathbf{B}\,(\beta)\,\phi$, $\mathbf{F}\phi$, and $\underline{(\mathbf{S}|\mathbf{S})}\,(t)\,\phi$ are implemented as static inline function in the CU language and compiled with the kernel library. Each thread calls these inline functions after it reads the input data from the global GPU memory. After performing the $S|S$ translation the threads are synchronized, and a reduction operation is used to consolidate the obtained S expansions and write them back to the global memory.

Table 4. Performance for Step 2 of the FMM Upward Pass

	$p=4$			$p=8$			$p=12$		
$l_{\max}$	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio
3	3.07E-04	1.44E-04	2.1	1.87E-03	9.26E-04	2.0	4.68E-03	2.49E-03	1.9
4	2.97E-03	4.57E-04	6.5	1.61E-02	2.18E-03	7.4	4.26E-02	5.62E-03	7.6
5	2.39E-02	2.34E-03	10.2	1.29E-01	1.16E-02	11.1	3.45E-01	3.09E-02	11.2
6	1.86E-01	1.89E-02	9.9	1.02E+00	9.39E-02	10.9	2.72E+00	2.55E-01	10.6

Table 4 shows performance of the CPU and GPU subroutines to perform entire Step 2 of the Upward Pass (so the *LevelUp* is called $l_{\max}-2$ times). The execution time here is presented as a function of $l_{\max}$. Indeed, the complexity of this step depends only on the number of source boxes, not on N. This number for uniform distributions, which we use as a benchmark is approximately 8^l for level l. Of course it may be not exactly 8^l due to some empty boxes appear (especially at low $s_{\max}$) at random realizations.

The obtained speedups are in range 2-11, where 2 corresponds to $l_{\max}=3$ which is only 512 children and 64 parent boxes at most. So the efficiency of translation/per thread parallelization is low (as we mentioned for the current architecture sizes involving 8192 parallel processes or more can be run at full efficiency. In fact, even $l_{\max}=4$ appears to be a bit low for a full load. For levels $l_{\max}\geqslant 5$, the GPU speed up over the serial code is approximately one order of magnitude, which includes computations not only for $l_{\max}$, but for

all levels from $l_{\max}$ to 3. We also note that speedups are uniform for all p considered.

Despite the speedups for this step appear to be lower than for other steps presented above, two observations make the obtained results satisfactory for the entire FMM.

The first observation is that, as will be seen below, the FMM on the GPU is optimal for a smaller $l_{\max}$ than on the CPU. The optimization study for the sparse matrix vector multiplier clearly indicates that for the benchmark case $l_{\max}$ does not exceed 4 for problem sizes $N \leqslant 2^{20}$. In most cases $l_{\max}$ for the GPU is equal to $l_{\max} - 1$ for CPU, while for case $p = 4$ it may be even smaller (4 instead of 6). Since the optimal settings for the CPU or GPU are not optimal for each other we can create a table for the best performing cases on each architecture and compare speedups as we deed before for the sparse matrix vector multiplier (Table 3). The value ∞ here appears due to Step 2 of the Upward Pass is not needed for $l_{\max} = 2$ (or its execution time is 0). We also put N/A for the cases which are not optimal. The speedups then appear to be in the same range as for other routines.

Table 5. Effective Performance (the best settings for CPU and GPU) for Step 2 of the FMM Upward Pass

$l_{\max}$		$p = 4$			$p = 8$			$p = 12$		
CPU	GPU	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio
3	2	3.07E-04	0.00E+00	∞	1.87E-03	0.00E+00	∞	4.68E-03	0.00E+00	∞
4	3	2.97E-03	1.44E-04	21	1.61E-02	9.26E-04	17	4.26E-02	2.49E-03	17
5	4	2.39E-02	4.57E-04	52	1.29E-01	2.18E-03	59	3.45E-01	5.62E-03	61
6	4	1.86E-01	4.57E-04	408	N/A	N/A	N/A	N/A	N/A	N/A

The second observation is that the Upward Pass is a very cheap step of the FMM and normally takes not more than 1% of the total time. This also diminishes the value of high speedups for this step.

4.3.4 *LevelDown*

Perhaps this is the most challenging subroutine for efficient parallelization. It generates R-expansions for all receiver boxes at level l using as input data (S expansions) from source boxes of the same level and R expansions from level $l - 1$. Despite this subroutine is wrapped as a single entity, it call three global device functions, *LevelDown*1, *LevelDown*2, and *LevelDown*3, which corresponds to the stencil translation scheme that we employ both for CPU and GPU. *LevelDown*1 performs all required multipole-to-local ($S|R$) translations from the source boxes to the receiver boxes of the same level (white boxes in Fig. 3), followed by consolidation. *LevelDown*2 performs all necessary $S|R$ translations from the other subset of the source boxes to the parent receiver box (gray boxes in Fig. 3), followed by consolidation. *LevelDown*3 performs the local-to-local ($R|R$) translations from the parent box to its children and adds the result to that computed by *LevelDown*1.

The *LevelDown*3 subroutine is very similar to *LevelUp*, since it performs $R|R$ translations from parent box to its 8 children. So we applied similar tricks to reduce the amount of translation data for this part and use fast access memory to store these data. One can even use the tables and data for the *LevelUp* to estimate the costs and speedups for *LevelDown*3.

Most problems for efficient parallelization appear for *LevelDown*1 and *LevelDown*2. We tried several translation schemes which demonstrated for these routines speedups in range 2-5 and we still are looking for better solutions. Current versions of these subroutines are built on the following principles.

*LevelDown*1 assigns a single receiver box at level l to a single thread. In this case thread synchronization is not needed and the nonuniformity of the receiver box neighborhoods is not very important. This thread performs up to 109 $S|R$ translations according to the stencil structure, consolidates all expansions, and writes the result to the global memory. *LevelDown*2 parallelized differently, due to the number of parent boxes can be substantially small. In this subroutine a block of threads performs task to perform all operations for a single parent box. Each thread is responsible for one $S|R$ translation. The maximum number of the $S|R$

translations to the parent receiver box in the stencil is 80. So up to 80 threads are employed at a time. This number is a bit low for a good performance of the GPU, but allows to all threads perform read/write operations efficiently and use the shared memory.

For the $S|R$ translation we use decomposition (33) (one should change there $S|S$ to $S|R$). The difference is that for the $S|R$ translations we need much more translation data, which cannot fit neither shared nor constant memory available with the scheme we use (this is an indication to search more compact translation operator decompositions of the same or better complexity; such schemes exist and even if they are not the best for CPU in our future research we are going to try to implement them on GPU). Indeed in the stencil there are 48 different α, 49 different β, and 14 different translation distances. To reduce the amount of translation data to read from the global memory, we put all αs to the constant memory as well as all translation distances. We load only one unitary ($S|R$) coaxial translation matrix, which we rescale each time to produce actual coaxial operator. This is based on decomposition

$$\underline{(\mathbf{S}|\mathbf{R})}\,(t) = \frac{1}{t}\Lambda\left(\frac{1}{t}\right)\underline{(\mathbf{S}|\mathbf{R})}\,(1)\,\Lambda\left(\frac{1}{t}\right), \tag{35}$$

where Λ is a diagonal matrix which nth subspaces has $1/t^n$ normalization. In this case $\underline{(\mathbf{S}|\mathbf{R})}\,(1)$ easily fits shared or constant memory.

More storage related problems are due to operator $\mathbf{B}\,(\beta)$. In principle it also can be decomposed into product of two constant axis flip matrices and diagonal matrix. But our tests on CPU for this method showed substantial reduction of the CPU performance and we did not proceed in this way. This operator is a block dense matrix (dense for each rotation subspace) and its size for $p = 12$ is 1152 floats. So to store 49 different operators we need about 226 KB of memory. An ideal solution would be to put these data to the constant memory, but it is limited to about 60 kB for the GPU we use. If a hardware with sufficient fast access memory will be available the algorithm can be modified, and we expect sensible acceleration in this case.

Another peculiarity of the $S|R$ translations is a heavy access to the data structure (neighborhood of the parent box at children level). Since the algorithm is adaptive we need for every translation obtain data on the source box and translation parameters. Despite we are using partially ordered lists, the access pattern is rather random, which builds substantial overhead. Our translation data structure represents translations as a graph and each entry to the list corresponds to the edge of this graph. Based on translation id it allows to retrieve source and target indices and encapsulated translation index. Being unpacked the translation index provides pointers to the α, β, and t-indices to access arrays storing respective rotation and translation data. This scheme is efficient on CPU, but maybe not the best for GPU, where the cost of one random access to global memory is equal up to 150 float operations, and instead of reading precomputed data GPU may rather produce them at higher rate.

In any case, we have implemented and tested all these subroutines and the test results are provided in the tables below.

Table 6. Performance for the FMM Downward Pass

	$p = 4$			$p = 8$			$p = 12$		
$l_{\max}$	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio
3	3.08E-02	1.46E-02	2.1	9.28E-02	7.71E-02	1.2	2.18E-01	2.33E-01	0.9
4	2.03E-01	6.14E-02	3.3	9.36E-01	2.65E-01	3.5	2.39E+00	7.18E-01	3.3
5	1.86E+00	3.59E-01	5.2	8.55E+00	1.80E+00	4.8	2.19E+01	4.84E+00	4.5

Table 6 similarly to Table 4 shows relatively low accelerations of this part of the FMM. Moreover for low $l_{\max}$ the GPU subroutine may even run slower than the serial CPU (!). However even this case we do not advise to switch between the CPU and GPU, since such a switch involves the slowest memory copying process (CPU-GPU), and if possible all data should stay on the GPU global memory. Performance improves as the size of the problem and, respectively, the maximum level of the octree increases and for $l_{\max} = 5$ the

time ratio can reach 5 or so. It is clear that even though such accelerations are not so impressive as for other subroutines, they show that it is preferable to run even the most heavy (in terms of operations with complex enough data structure, and extensive random access to the global memory) part of the FMM on the GPU rather than on the CPU. The speedup ratios look much better, if we plot the effective accelerations for the Downward Pass. Reduction of the $l_{\max}$ as it is seen is very beneficial for this part of the algorithm, and the effective accelerations for problems of size $N \sim 2^{18} - 2^{20}$ are in the range 30 (we have one anomalous case of 250 times speedup). Effective accelerations of order 30 are more or less consistent with the effective accelerations for the other steps of the algorithm, particularly with that for the sparse matrix vector multiplier (see Table 2). As this applies to all FMM steps this allows us to expect that the entire FMM can be speeded up 30 times or so.

Table 7. Effective Performance (the best settings for CPU and GPU) for the FMM Downward Pass

$l_{\max}$		$p = 4$			$p = 8$			$p = 12$		
CPU	GPU	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio
3	2	3.08E-02	4.83E-03	6	9.28E-02	2.31E-02	4	2.18E-01	6.22E-02	4
4	3	2.03E-01	1.46E-02	14	9.36E-01	7.71E-02	12	2.39E+00	2.33E-01	10
5	4	1.86E+00	6.14E-02	30	8.55E+00	2.65E-01	32	2.19E+01	7.18E-01	30
6	4	1.54E+01	6.14E-02	250	N/A	N/A	N/A	N/A	N/A	N/A

4.4 Overall performance

To demonstrate a typical profile of the FMM we present Table 8, which shows the execution time for different components of the FMM for problem (1). Optimal settings for the CPU and GPU are used for fixed truncation numbers. The error here is measured as

$$\epsilon_2 = \frac{\epsilon_2^{(abs)}}{\|\phi_{exact}(\mathbf{y})\|_2}, \tag{36}$$

where

$$\epsilon_2^{(abs)} = \left[\frac{1}{M}\sum_{j=1}^{M}|\phi_{exact}(\mathbf{y}_j) - \phi_{approx}(\mathbf{y}_j)|^2\right]^{1/2}, \quad \|\phi_{exact}(\mathbf{y})\|_2 = \left[\frac{1}{M}\sum_{j=1}^{M}|\phi_{exact}(\mathbf{y}_j)|^2\right]^{1/2},$$

where $\phi_{exact}(\mathbf{r})$ and $\phi_{approx}(\mathbf{r})$ are the exact (computed directly on CPU with double precision) and approximate (FMM and/or use of GPU) solutions of the problem. As shown in our report [11] ϵ_2 is a statistically stable measure of the error (for $M \geqslant 100$ it almost does not depend on M for a fixed distribution of N sources). Concerning the errors, we can see that the GPU code for the cases displayed showed the error ϵ_2 which does not exceed that for the CPU code. This error was even slightly smaller for larger p, which can be explained by the fact that the GPU code was executed with a smaller octree depth than on the CPU. This was beneficial not only for a good overall timing, but for the error, since more source-receiver interaction were taken into account with higher (machine) precision as they moved from the approximate dense matrix vector product to "exact" sparse matrix vector product.

Table 8. Comparison of optimal CPU and GPU FMM: $N = 1,048,576$.						
	S exp	Up trans	Down trans	R eval	Sparse	Total
$p = 4$, CPU: $l_{\max} = 6, \epsilon_2 = 2.3 \times 10^{-4}$, GPU: $l_{\max} = 4, \epsilon_2 = 2.3 \times 10^{-4}$.						
Serial CPU(s)	0.34	0.19	15.06	0.34	6.31	22.25
GPU(s)	0.011	0.00046	0.061	0.011	0.60	0.6835
Speedup (times)	31	413	247	31	11	32.6
$p = 8$, CPU: $l_{\max} = 5, \epsilon_2 = 8.8 \times 10^{-6}$, GPU: $l_{\max} = 4, \epsilon_2 = 8.3 \times 10^{-6}$.						
Serial CPU(s)	0.83	0.12	8.42	0.85	40.95	51.17
GPU(s)	0.020	0.00218	0.265	0.021	0.60	0.9082
Speedup (times)	42	55	32	40	68	56.3
$p = 12$, CPU: $l_{\max} = 5, \epsilon_2 = 1.3 \times 10^{-6}$, GPU: $l_{\max} = 4, \epsilon_2 = 9.5 \times 10^{-7}$						
Serial CPU(s)	1.91	0.33	21.30	1.93	40.95	66.56
GPU(s)	0.040	0.00562	0.72	0.030	0.59	1.395
Speedup (times)	48	59	30	67	69	47.7

This table demonstrates overall speedups of the entire FMMrun in the range 30-60 for the million size case, which is consistent with the efficient speedups for all components. It is well seen from data for $p = 8$ and $p = 12$. The case $p = 4$ shows higher imbalance of component accelerations, which is due to the larger difference in optimal $l_{\max}$ for the CPU and GPU realizations.

It may be noticed that for all cases considered the sparse matrix multiplier takes a substantial part of the total execution time. That puts a lighter weight on optimization of the translation subroutines. Indeed, assuming that the entire dense matrix vector product (the first four components in Table 8) are executed with zero time, we can see that the total time will change at most by a factor of 2.3 for $p = 12$ and only by a factor of 1.14 for $p = 4$. Since the sparse matrix vector is optimized, we can see that even for the best possible translation methods we cannot expect increase of the overall performance by orders of magnitude, and at most by a factor of approximately 2. This is totally different with the model of the FMM on CPU, where efficiency of the translation has a heavy impact on total time as it controls the depth of the octree, and one can therefore expect orders of magnitude accelerations by balancing the costs of the dense and sparse parts of the FMM.

4.4.1 FMM and roundoff errors

There are two types of errors associated with the use of the FMM. First, we have algorithmic errors due to truncation of infinite series, which is controlled by the truncation number p, and, second, there are machine errors due to roundoff in various operations and intrinsic functions. To study the errors we created a benchmark case, where all sources have intensity $q = 1$ and distributed in a regular grid over the unit cube. 729 evaluation points were also put in a grid covering the domain. As a measure of the error we took the partial relative L_2 norm error ϵ_2 (see Eq. (36)). "Exact" solution was computed on the CPU using double and quadruple-precision and direct summation (double precision and quadruple-precision provided consistent results which are far more accurate than the single precision computations). Single precision results obtained on the CPU and GPU using the FMM and direct summation on GPU were compared with the "exact" solution. The use of grids and constant source intensity for error check was dictated by the desire to eliminate any possible error in input (positions and intensities). The results are shown in Fig. 8.

First, we can note that the CPU and GPU implementations of the FMM produce approximately the same results, which clearly show that the error is controlled by p for the cases studied. A small difference here in the FMM errors is explainable by the fact that lower $l_{\max}$ is used for the GPU. Second, it is seen that the direct sums on the GPU are computed with error which increases with N. We relate this to accumulation of roundoff errors in direct summation when computing large sums.

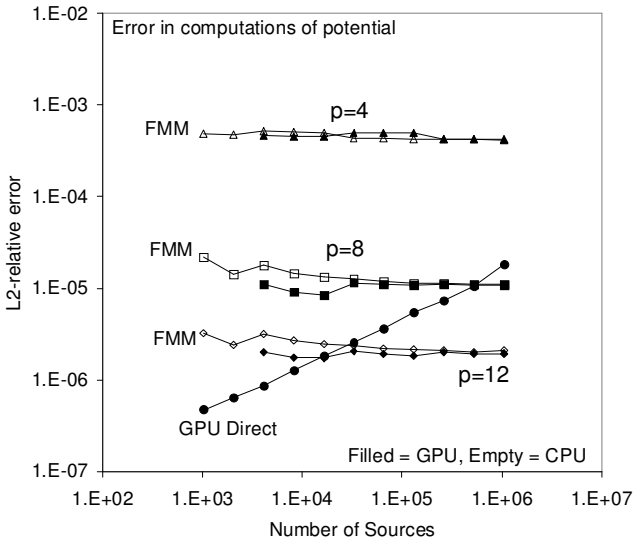

Figure 8: ϵ_2 error (see (36)) for evaluation of sum (1) using the FMM on CPU (filled markers) and on GPU (empty markers). Also the error for direct summation on the GPU is shown. The errors are obtained by comparisons with the double precision direct computations on CPU for a grid of N sources of intensity 1 and 729 receivers.

On the other hand, for large sums the FMM demonstrates more stability to roundoff errors. Indeed, only a limited number of source receiver interactions is taken into account directly (sparse matrix) and computations of the far field via expansions involve very limited number of operations (limited number of expansion terms and a few stable translation operators). So if we speak about summation with machine precision (i.e. precision available through the use of standard hardware on the GPU) it appears that the use of the FMM with moderate truncation numbers ($p = 8, 12$ in our case) is sufficient to reach this level of accuracy. Moreover, the use of the FMM may be even preferable (Fig. 8 shows that FMM on GPU is more accurate than direct summation for $N \gtrsim 50,000$ and $p = 12$ and for $N \gtrsim 1,000,000$ and $p = 8$). We also performed more error tests with random distributions, different number of evaluation points, and random source intensities, which confirm this finding.

4.4.2 FMM time

Finally we present the performance (timing) results for different truncation numbers. All results in this section are obtained for computations of both the potential and its gradient as given by Eqs (1) and (2). Tests were performed on our benchmark problem (N random sources of random intensities $q \in (0, 1)$, and independent set of $M = N + 1$ random receivers). Table 11 shows some numbers obtained. Note that these numbers are somehow random, as we used random realizations, while we did each computation at least 10 times to be sure that the results are stable. One also can compare these data with that given in Table 10 to see the cost of computations of gradient.

Table 11. Performance of the FMM implementations on CPU and GPU

$p = 4$			$p = 8$			$p = 12$		
CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio	CPU(s)	GPU(s)	Ratio
28.37	0.979	29	88.09	1.227	72	116.1	1.761	66

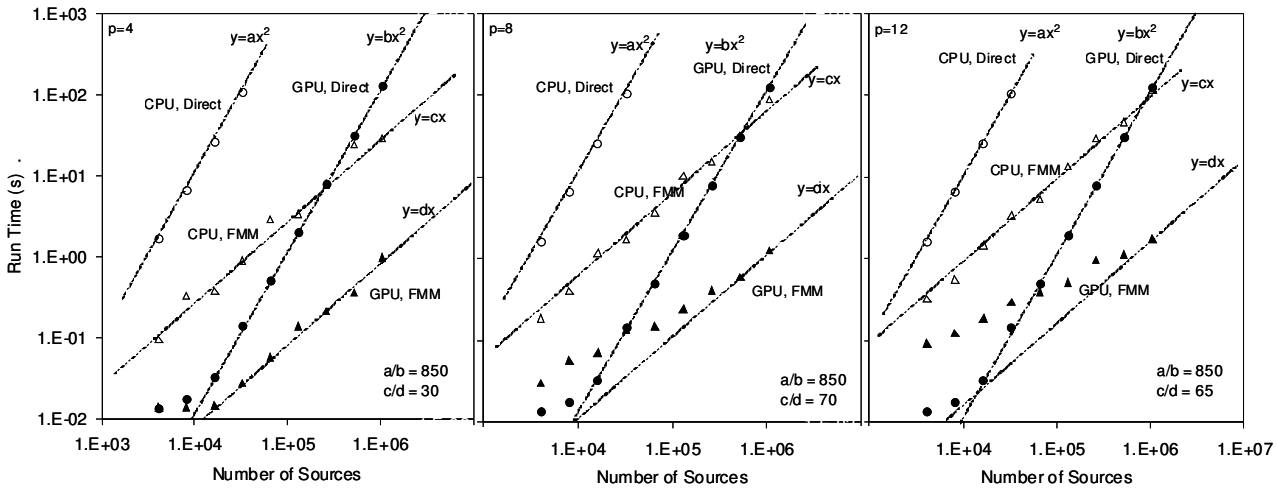

Figure 9: FMM wall clock time (in seconds) for serial CPU code (one core of 2.67 GHz Intel Core 2 extreme QX is employed) and for GPU (NVIDIA GeForce 8800 GTX) for different truncation numbers p (potential+gradient). Also direct summation timing is displayed for both architectures. No SSE optimizations for the CPU were used.

Fig. 9 shows wall clock time required to run the FMM and direct summation on a single CPU and GPU. We note a huge acceleration for direct summation, which should be reduced if one uses more proper CPU memory management, SSE instructions, and high performance libraries (we compared with a naive straightforward implementation, when the direct summator simply executes the nested loop).

For $p = 4$ at large enough N both the CPU and GPU times depend on N linearly, which is consistent with the theory of the FMM. It interesting also to note that the so-called break-even point, i.e. the point at which the time of the direct summation and the FMM is equal for the FMM running on the GPU is somewhere around $N \sim 10^4$, while for the FMM on CPU this point is below $N = 10^3$. So the use of the FMM is efficient on the GPU for larger N. It is also interesting to note the second break-even point, where the direct summator for the GPU crosses the dependence for serial CPU. This point is located $N \sim 2 \times 10^5$, which also shows that it is maybe more efficient (do not forget about the error!) to compute problems with smaller N on the GPU directly than use rather complex FMM strategy on the CPU. In any way due to linear scaling any FMM at some N beats any direct method scaled quadratically and for million size problems the FMM is more efficient method. We also note that it is possible to compute even million size problem using the direct method on GPU, which takes a reasonable ~100 s time. But the FMM on GPU performs this computations for times less than for 1 s (0.98 s). It is accepted by several researchers in the N-body community (e.g. [21]) that the number of float operations for a single potential/gradient evaluation is 38, and this is the figure used for comparisons in e.g., [19]. In this case the effective rate due to algorithmic, FMM, and, hardware, GPU, acceleration for the $N = 2^{20}$ case reaches number about 43 Tflops (!).

Case $p = 8$ has some similarities and differences with the case $p = 4$. The similarity is in fact that the FMM is greatly accelerated by the use of the GPU. The difference is that the time dependence of the FMM on GPU on N seems not linear. Since p influences only the dense matrix vector product this is manifestation of the fact that our parallelization approach based on parallelization of operations on boxes does not reach asymptotic saturation in the range of N, for which the current study is conducted. Indeed for large p the downward pass costs as the sparse matrix vector multiplication, while at levels $l_{\max} = 3$ and 4 it still does not reach its asymptotic dependence on $l_{\max}$, or N. However our tests for larger $l_{\max}$, which we did not

present in Table 8 show that, in fact such saturation occurs and accelerations for the CPU analog running at the same level stabilize around 5. So it is anticipated that at larger N the dependence of the GPU time on N will be linear if we keep $s_{\max}$ to be a constant ($s_{\max} = 320$). We also can note that the break even point for the CPU is still below $N = 10^3$ while this point for the GPU shifts towards larger $N \sim 3 \times 10^4$. Our estimates show that the effective rate of computations in this case 34 Tflops for $N = 2^{20}$.

Case $p = 12$ is qualitatively similar to case $p = 8$ and the same explanation for the nonlinear behavior of the FMM run time on the GPU applies to it. The dependence of the run time on $l_{\max}$ rather than on N is expressed here even more than in the graph for $p = 8$. The break-even point shifts further towards to larger N and is around $N \sim 7 \times 10^4$ and the effective rate in this case is about 24 Tflops for $N = 2^{20}$.

5 Conclusions

We have accelerated the run time of the FMM algorithm on the GPU in the range 30-70 (depending on the accuracy) which is equivalent to the computing rates in the 24-43 Teraflop equivalent rates [21] on a single GPU and showed a feasibility of putting the FMM on the GPU. This should permit the use of these architectures in several scientific computation problems where the FMM or hierarchical algorithms can be profitably used. Our analysis showed the following conclusions:

Optimal cluster size on a GPU: On the CPU the local direct sum cost is proportional to the cluster size and the cost of the far-field evaluation decreases with increasing cluster size, and it is possible to find an optimal setting. In contrast, on the GPU architecture access to global memory is expensive. As there are more boxes, and more clusters, for each cluster we need to perform random access to the global memory for its neighbor list. Because of this there is an optimal size for the cluster (independent of the FMM far field). For optimality, the cluster size should be larger than certain value determined by the hardware. This optimal size is larger than that for the CPU, and correspondingly the FMM is faster than the direct product on the GPU only for larger problem sizes and the FMM optimization should be performed with these optimal settings. The increase of the cluster size results in more effective acceleration of all other components of the FMM as well and improves the accuracy of the computations.

Parallelization Strategies for the FMM: The FMM operations require a different amount of local memory. For example, each translation requires the use of a relatively high amount of constant translation data, which is difficult to fit into the small amount of constant memory available. Accordingly we used two different parallelization strategies of TpB ("thread per box") and BpB ("block (of threads) per box"). The upward pass (which requires creating multipole expansion coefficients, and multipole-to-multipole translation could be done via the TpB strategy, while in the downward pass, the multipole-to-local translation and local summation required a BpB strategy, while the local expansion evaluation could be done via a TpB strategy.

Error in the Computed Solution: Current GPUs are single precision architectures, and the behavior of the error in the solution is of interest. As a general conclusion, we can say that the GPU behaves as expected. An interesting feature of the GPU sum is that at some point the accumulated arithmetic errors due to computing large sums of (reciprocal) square roots becomes much larger, and the FMM becomes more accurate than direct summation.

6 Acknowledgements

We would like to acknowledge partial funding of this work via a grant from NASA to Fantalgo. In addition NG was also partially supported by the Center for Multiscale Plasma Dynamics (CMPD), a Department of Energy Fusion Science Center.

References

[1] NVIDIA CUDA *Compute Unified Device Architecture Programming Guide, V. 1.0*, 06/01/2007. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf)

[2] M. Abramowitz, and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965.

[3] H. Cheng, L. Greengard, and V. Rokhlin, A fast adaptive multipole algorithm in three dimensions, *J. Comput. Phys.*, 155, 1999, 468-498.

[4] J.J. Dongarra and F. Sullivan, The top 10 algorithms, *Computing in Science & Engineering,* 2, 2000, 22-23.

[5] M.A. Epton and B. Dembart, Multipole translation theory for the three-dimensional Laplace and Helmholtz equations, *SIAM J. Sci. Comput.*, 16(4), 1995, 865-897.

[6] L. Greengard and V. Rokhlin, A fast algorithm for particle simulations, *J. Comput. Phys.*, 73, 1987, 325-348.

[7] L. Greengard and W.D. Gropp, A parallel version of the fast multipole method, *Comput. Math. App.* 20, 1990, 63-71.

[8] L. Greengard and V. Rokhlin, A new version of the fast multipole method for the Laplace equation in three dimensions, *Acta Numerica*, 6, 1997, 229-269.

[9] N.A. Gumerov and R. Duraiswami, *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*, Elsevier, Oxford, UK, 2005.

[10] N.A. Gumerov and R. Duraiswami, Fast multipole method for the biharmonic equation in three dimensions, *J. Comput. Physics*, 215 (1), 2006, 363-383.

[11] N.A. Gumerov and R. Duraiswami, Comparison of the efficiency of translation operators used in the fast multipole method for the 3D Laplace equation, *University of Maryland Technical Report* UMIACS-TR-#2003-28, (http://www.cs.umd.edu/Library/TRs/CS-TR-4701/CS-TR-4701.pdf).

[12] N.A. Gumerov, R. Duraiswami, and W. Dorland, Middleware for programming NVIDIA GPUs from Fortran 9X, (poster presentation at Supercomputing 2007: online at www.umiacs.umd.edu/~ramani/pubs/Gumerov_SC07_poster.pdf)

[13] J.P. Singh, C. Holt, J.L. Hennessy, and A. Gupta, A parallel adaptive fast multipole method, in: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, December 1993.

[14] J.P. Singh, J.L. Hennessy, and A. Gupta, Implications of hierarchical N-body methods for multiprocessor architectures, *ACM Trans. Comput. Systems* 13 (2), 1995, 141–202.

[15] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, and J.L. Hennessy, Load balancing and data locality in adaptive hierarchical N-body methods: B-H, FMM, and radiosity, *J. Parallel Distrib. Comput.* 27, 1995, 118–141.

[16] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten. Accelerating Molecular Modeling Applications with Graphics Processors, *J. Comput. Chem.*, in press, 2007.

[17] S.-H. Teng, Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation, *SIAM J. Sci. Comput.* 19(2), 1998, 635–656.

[18] C.A. White and M. Head-Gordon, Rotation around the quartic angular momentum barrier in fast multipole method calculations, *J. Chem. Phys.*, 105(12), 1996, 5061-5067.

[19] Tsuyoshi Hamada, Toshiaki Iitaka, The Chamomile scheme: an optimized algorithm for N-body simulations on programmable graphics processing units (posted on arXiv: astro-ph/0703100v1, 6 Mar 2007).

[20] L. Nyland, M. Harris, and J. Prins, N-body simulations on a GPU, in *Proc of the ACM Workshop on General-Purpose Computation on Graphics Processors*, 2004.

[21] S. Warren, M, K. Salmon, J, J. Becker, D, P. Goda, M, 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 *Proc. Supercomputing 97*, in CD-ROM. IEEE, Los Alamitos, CA, 1997.

[22] J. Makino, T. Fukushige, M. Koga, and K. Namura, GRAPE-6: Massively-parallel special-purpose computer for astrophysical particle simulations, *Publication of Astronomical Society of Japan*, 55, 2001, 1163–1187.

[23] T. Fukushige, J. Makino, and A. Kawai, GRAPE-6A: A single-card GRAPE-6 for parallel PC-GRAPE cluster systems, PASJ: Publ. Astron. Soc. Japan 57, 2005, 1009-1021.