Journal of Computational Physics 227 (2008) 8290–8313

Contents lists available at ScienceDirect

Journal of Computational Physics

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

Fast multipole methods on graphics processors

Nail A. Gumerov *, Ramani Duraiswami

Perceptual Interfaces and Reality Laboratory, Computer Science and UMIACS, University of Maryland, College Park, United States
Fantalgo, LLC, Elkridge, MD, United States

ARTICLE INFO

Article history:
Received 24 November 2007
Received in revised form 21 May 2008
Accepted 22 May 2008
Available online 10 June 2008

ABSTRACT

The fast multipole method allows the rapid approximate evaluation of sums of radial basis functions. For a specified accuracy, ϵ, the method scales as $O(N)$ in both time and memory compared to the direct method with complexity $O(N^2)$, which allows the solution of larger problems with given resources. 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 architecture; and determined optimal settings for the FMM on the GPU. These optimal settings 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 of 30–60 compared to a serial CPU FMM implementation. 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 a 43 Teraflop rate if we use straightforward summation.

1. Introduction

Fast multipole methods (FMM) were invented in the late 1980s [8], and have since been identified as one of the ten most important algorithmic contributions in the 20th century [5]. The FMM is widely used for problems arising in diverse areas (molecular dynamics, astrophysics, acoustics, fluid mechanics, electromagnetics, scattered data interpolation, etc.) because of its ability to achieve linear time and memory dense matrix vector products to a fixed prescribed accuracy ϵ.

Graphics processing units: A recent hardware trend, with origin in the gaming and graphics industries, is the development of highly capable data-parallel processors. In 2007, while the fastest Intel CPU can only achieve ~20 Gflops speeds on benchmarks, 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 2006. The trends reported are expected to continue for the next few 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

* Corresponding author. Address: Perceptual Interfaces and Reality Laboratory, Computer Science and UMIACS, University of Maryland, College Park, United States. Tel.: +1 301 405 8210.
E-mail addresses: gumerov@umiacs.umd.edu, info@fantalgo.com (N.A. Gumerov), ramani@umiacs.umd.edu, info@fantalgo.com (R. Duraiswami).
URLs: http://www.umiacs.umd.edu/~gumerov, http://www.fantalgo.com (N.A. Gumerov), http://www.umiacs.umd.edu/~ramani, http://www.fantalgo.com (R. Duraiswami).

0021-9991/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2008.05.023

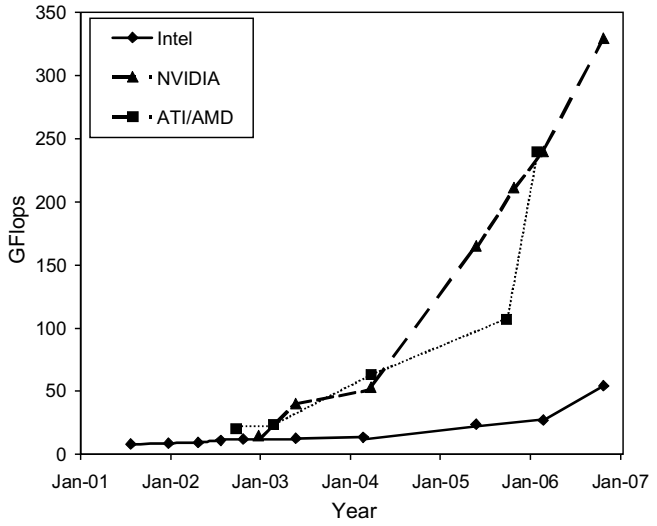

Fig. 1. GPU and CPU growth in speed over the last six years (adapted from a figure in [24]).

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. [19,16,23,29]). While impressive speedups are reported, these algorithms suffer from their $O(N^2)$ nature for large *N*. Some preliminary steps towards achieving a hybrid CPU-GPU treecode are presented in work concurrent to this paper [28].

Researchers in Japan, under the auspices of the Grape project, have also sought to develop special hardware, similar to the GPU, for performing *N* body calculations (see e.g. [18,15]). However, 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 the 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 that address some basic issues [9,17,20–22,25]. However, these publications are mostly related to the development of more coarse-grained parallel algorithms, and often focus on tree codes. In contrast, in this paper, we work with the full FMM algorithm and optimize performance by accounting for the nonuniform memory access costs associated with the global and local memory in the GPU, and the data parallel architecture. We use the hierarchical nature of the FMM to balance cost trees, and achieve optimal performance. This 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 an algorithm that is able to compute a matrix vector product for a matrix of size $10^6 \times 10^6$ in a time of the order of one second on a NVIDIA GeForce 8800GTX GPU. This corresponds to effective flop rates of 43 Tflops, following the formula given in [26,16].

2. Fast multipole method (FMM)

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

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

where $\{\mathbf{x}_i\}$ are the sources, $\{\mathbf{y}_j\}$ the receivers, q_i the strengths and d is the dimensionality of the problem. The function $\Phi(\mathbf{y}_j, \mathbf{x}_i)$ is called the kernel, and the FMM has been developed for several kernels. The FMM can also be used to compute the gradient of the potential at the receiver locations as

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

Examples of problems where such computations arise include particle interaction, star motion, and molecular dynamics. This problem may also be recognized as a matrix vector product, where the matrix elements are derived from particular radial basis functions/potentials. Such matrices arise via the discretization of integral equation formulations of common PDE's, and iterative techniques for the solutions of the associated linear systems require repeated matrix vector products that can be accelerated via the FMM.

In this paper we will consider the Coulomb (or Laplace) kernel in 3D, for which

$$\Phi(\mathbf{y},\mathbf{x}) = \begin{array}{ll} |\mathbf{x}-\mathbf{y}|^{-1}, & \mathbf{x}\neq\mathbf{y} \\ 0, & \mathbf{x}=\mathbf{y} \end{array} \tag{3}$$

The theory of the FMM for this kernel is well developed starting from the original works of Rokhlin and Greengard [8].

2.1. Outline of the FMM

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

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

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

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

The latter sparse matrix vector multiplication is performed directly. All other steps of the FMM are dedicated to approximate computation of $\mathbf{\Phi}^{(\text{dense})}\mathbf{q}$ to a specified error ϵ via the use of data structures, multipole and local expansions and translation theory.

The algorithm can be summarized as follows. Assume that the computational domain including all sources and receivers is contained in a large cube (which can be scaled to the unit cube). This cube then is subdivided via an octree into 8 subcubes, and recursively going down to level $l_{\max}$, so that at each level we have 8^l boxes ($l = 0,\ldots,l_{\max}$). 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 numbers of points, and some boxes may even be empty. The empty boxes are skipped at all levels, which provides a basic adaptivity to 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 [11]). This technique also allows one to determine neighbors for each box, which is important for the FMM. In addition, 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 $\{\phi_j\}$). The FMM consists of three basic steps: upward pass, downward pass and final summation, which are described in detail below.

2.1.1. Upward pass

Step 1. For each box at the finest level, $l_{\max}$, generate a multipole, or S expansion for each source at the box center. This expansion is vector of expansion coefficients $\{C_n^m\}$ over some basis (usually the multipole solutions of Laplace's equation). As function representation in this basis is infinite, the series are truncated with truncation number p, so that there are p^2 coefficients associated with each box center. The choice of p is dictated by the required accuracy. The maximum achievable accuracy on any computer architecture is related to the machine precision (single or double precision), and this provides a limit on the maximum p that may be used. Expansions from all sources in the box are summed in a single expansion by summing the coefficients corresponding to each source.

Step 2. Moving up the source hierarchy, for levels from $l_{\max}$ to 2, we generate S expansion coefficients for each box. At each level, we apply the multipole-to-multipole (or $S|S$) translation operator on the coefficients of each child box, and consolidate (sum) the results. The procedure is repeated hierarchically from level to level up the tree.

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 via the following two step procedure:

Step 1. Translate the multipole S expansions from the source boxes at the same level belonging to the neighborhood of the receiver box's parent, but not belonging to the neighborhood of the receiver box itself, to R expansions centered at the center of the receiver box using multipole-to-local, or $S|R$, translations. All the expansions are consolidated (summed) into one coefficient vector.

Step 2. Translate the R expansion coefficients from the parent receiver box center to child box centers and consolidate them with the $S|R$ translated expansions at this level available from Step 1. This step starts from level 3 (or formally from level 2, if the R expansion for the parent box is considered initialized to zero).

2.1.3. Final summation

Step 1. Evaluate local R expansions for each receiver box at level $l_{\max}$ at all receiver locations in the box.

Step 1′. If gradient computations are needed, apply the sparse matrices corresponding to differentiation to the vectors of expansion coefficients, and efficiently obtain the expansion coefficients for the gradient. Evaluate the gradient expansion at each receiver location in the box.

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

This also can be represented as a flow chart 2 (See Fig. 2).

2.2. Characteristics of our FMM implementation

There are several bases, translation, methods and schemes for the FMM for the Laplace kernel. Each of these have their own advantages and disadvantages. The choice of a particular method depends on the problem to be solved and the accuracy desired. We consider here an FMM algorithm consistent with the single precision floating point arithmetic on the GPU available to us. While some software for double precision exists on the GPU, their use is reported to show a decrease of the computational speed by up to 10 times (see the discussion e.g. in [7]), 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 that this feature will be released in 2008). In this case, 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 4 byte floats produce a 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 problems with $N \simeq 10^6$, and we focus on this range of truncation numbers in this paper.

Our comparative study of various translation methods on the CPU [13] show that for this range, the standard multipole expansions and rotation-coaxial translation-rotation (RCR) decomposition of the translation operators provide the best results. The RCR decomposition, introduced first in Ref. [27], has $O(p^3)$ complexity with a low asymptotic constant for any translation operator. For the values of p considered here, they are as fast or faster than the $O(p^2)$ methods in the literature, which have much larger asymptotic constants and result in increased sizes of the representing vectors [10].

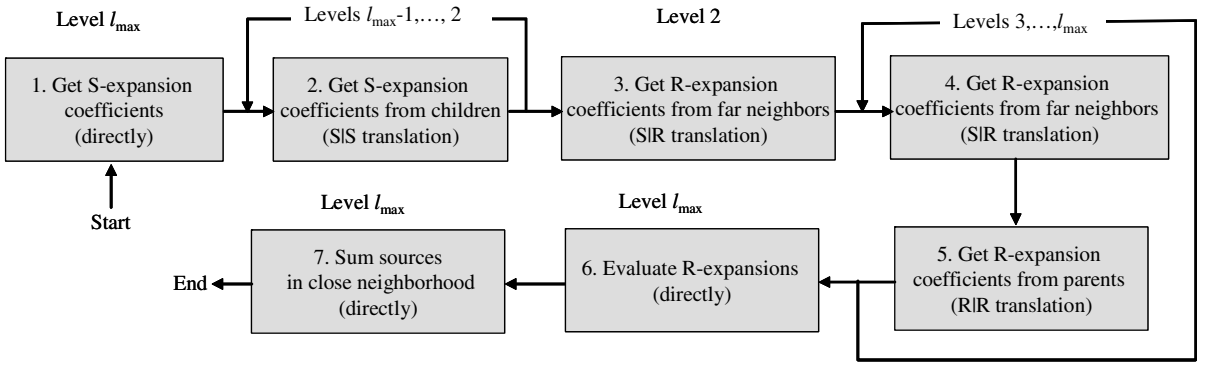

Fig. 2. A flow chart of the standard FMM.

We also revised and modified the computationally expensive original $S|R$ translation scheme associated with the conventional FMM, and used normalized real basis functions that can be evaluated quickly on the GPU.

2.2.1. Basis functions

In spherical coordinates (r,θ,φ), related to the Cartesian coordinates (x,y,z) via

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

elementary solutions of the Laplace equation in 3D can be represented as

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

Here $R_n^m(\mathbf{r})$ are the regular (local) spherical basis functions 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 that can be selected by convenience, and $Y_n^m(\theta,\varphi)$ are the orthonormal spherical harmonics:

$$Y_n^m(\theta,\varphi) = N_n^m P_n^{|m|}(\mu) e^{\mathrm{i}m\varphi}, \quad \mu = \cos\theta, \tag{8}$$

$$N_n^m = (-1)^m \sqrt{\frac{2n+1}{4\pi}\frac{(n-|m|)!}{(n+|m|)!}}, \quad n = 0,1,2,\ldots, \quad m = -n,\ldots,n,$$

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

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

Straightforward computation of these basis functions involves several relatively costly operations, such as conversion back and forth to spherical coordinates. Further, as defined above, these functions are complex. This is an unnecessary expense when the terms in (1) are all real.

One of the most convenient bases for translation was proposed by Epton and Dembart [6], which is basis (7) with

$$\alpha_n^m = (-1)^n i^{-|m|} \sqrt{\frac{4\pi}{(2n+1)(n-m)!(n+m)!}}, \tag{10}$$

$$\beta_n^m = i^{|m|} \sqrt{\frac{4\pi(n-m)!(n+m)!}{2n+1}}, \quad n = 0,1,\ldots, \quad m = -n,\ldots,n.$$

However, these functions are also complex. To obtain a real basis we set

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

$$n = 0,1,\ldots, \quad m = -n,\ldots,n.$$

and define 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 the form

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

where the C_n^m and D_n^m coefficients are real, and the infinite sums are truncated and p^2 terms retained.

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

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

Dropping details of derivation, one can compute each $\widetilde{R}_n^m(\mathbf{r})$ recursively in a few operations using the following

$$\widetilde{R}_0^0 = 1, \quad \widetilde{R}_1^1 = -\frac{1}{2}x, \quad \widetilde{R}_1^{-1} = \frac{1}{2}y, \tag{15}$$

$$\widetilde{R}^{|m|}_{|m|} = -\frac{\left(x\widetilde{R}^{|m|-1}_{|m|-1} + y\widetilde{R}^{-|m|+1}_{|m|-1}\right)}{2|m|}, \quad \widetilde{R}^{-|m|}_{|m|} = \frac{(y\widetilde{R}^{|m|-1}_{|m|-1} - x\widetilde{R}^{-|m|+1}_{|m|-1})}{2|m|}, \quad |m| = 2,3,\ldots$$

$$\widetilde{R}^{m}_{|m|+1} = -z\widetilde{R}^{m}_{|m|}, \quad m = 0,\pm 1,\ldots,$$

$$\widetilde{R}^{m}_{n} = -\frac{(2n-1)z\widetilde{R}^{m}_{n-1} + r^2\widetilde{R}^{m}_{n-2}}{(n-|m|)(n+|m|)}, \quad n = |m|+2,\ldots, \quad m = -n,\ldots,n.$$

These recursions show that functions $\widetilde{R}^{m}_{n}$ are real polynomials of degree n of Cartesian coordinates of $\mathbf{r} = (x,y,z)$. To compute them there is no need to convert to spherical coordinates and back. Further, their use avoids the need to use special functions, which might be implemented with limited accuracy and longer clock cycles, a consideration that is important on both the GPU and the CPU. Finally, 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

$$R^{m}_{n}(\mathbf{r}+\mathbf{t}) = \sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(R|R)^{m'm}_{n'n}(\mathbf{t})R^{m'}_{n'}(\mathbf{r}), \tag{16}$$

$$S^{m}_{n}(\mathbf{r}+\mathbf{t}) = \sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(S|R)^{m'm}_{n'n}(\mathbf{t})R^{m'}_{n'}(\mathbf{r}), \quad |\mathbf{r}| < |\mathbf{t}|,$$

$$S^{m}_{n}(\mathbf{r}+\mathbf{t}) = \sum_{n'=0}^{\infty}\sum_{m'=-n'}^{n'}(S|S)^{m'm}_{n'n}(\mathbf{t})S^{m'}_{n'}(\mathbf{r}), \quad |\mathbf{r}| > |\mathbf{t}|,$$

where $\mathbf{t}$ is the translation vector, and $(R|R)^{m'm}_{n'n}$, $(S|R)^{m'm}_{n'n}$, and $(S|S)^{m'm}_{n'n}$ 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. [6,4]). It is not difficult to show then that reexpansion of the basis functions produces the entries of the matrix translation operator, which allows conversions of expansion coefficients in one basis to expansion coefficients in the same or different basis centered at a different point. For example, for multipole-to-local translation we have

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

where $C^{m'}_{n'}$ are coefficients of multipole expansion in some reference frame, and D^{m}_{n} are coefficients of expansion of the same function over the local basis in the reference frame with the origin shifted by the translation 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, which is expensive. This can be reduced to $2p^3 + O(p^2)$ for the $S|R$ translation using the RCR decomposition (see [27] for an initial treatment, and [11,13] for extensions). In this decomposition, we first perform rotation of the reference frame to point the z axis towards the target, then perform coaxial translation along the rotated z axis, and then rotate the reference frame back (see Fig. 3). Each operation has $O(p^3)$ complexity since it can be decomposed in to a set of $O(p)$ translations each of which has $O(p^2)$ complexity.

For coaxial translations we have

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

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 are

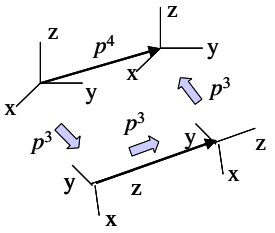

Fig. 3. Illustration of the rotation coaxial-translation rotation (RCR) technique for $O(p^3)$ translation (see [13]).

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

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

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

and zero for $n < 0$.

The rotation transform is a bit more involved, since the rotation operators when expressed in the normalized basis functions used here are different in the S and R bases. However, multiplication of the rotation operators by diagonal matrices makes them the same in the two bases, and this can be used for more compact storage of the rotation operator data. The diagonals of these diagonal matrices consist of coefficients α^m_n for the R basis and β^m_n for the S basis (which is equivalent to the use of bases (7) with $\alpha^m_n = 1$ and $\beta^m_n = 1$). Note that in this case the rotation operators are related to rotation of spherical harmonics. A detailed theory of fast recursive computation of the rotation coefficients and rotation can be found e.g. in our book [11] or in our technical report [13]. We use the methods described there with small modifications to account for the fact that we deal with the purely real basis functions 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 is that there are 189 translations per non boundary receiver box in 3D. In [4] more efficient translation operators for large p based on an exponential representation and a translation stencil that took advantage of symmetries and reduced the number of translations to ~40. As discussed earlier, for the accuracy that can be achieved on the GPU, the usual FMM representation in terms of multipoles and regular functions, along with the $O(p^3)$ translation operators are used in our implementation. We found some symmetries that allow the number of translations in the regular FMM stencil and the $O(p^3)$ translation operators to be reduced.

Consider the neighborhood of the parent receiver box. There are 216 boxes on the level corresponding to that of the receiver box. S expansions from boxes that are not in the neighborhood of the receiver box should be translated to R expansions at each box center. Considering the eight children boxes together, we have $8 \times 189 = 1512$ translations. We observe that there exist 80 boxes (see the 3D parent neighborhood in Fig. 4) from which the $S|R$ translations can be made not to the children, but to the parent box itself, while still satisfying the specified error bound. The coefficients resulting from such translations can be summed with the R expansion at the parent box and Step 2 of the downward pass can be performed without any additional constraints. 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.

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 that reduces this number of translations

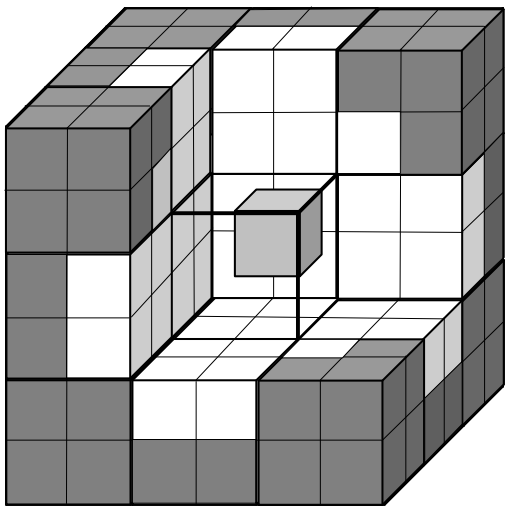

Fig. 4. A section of the translation stencil used in the computations. Instead of translation to the central receiver box (the light gray shade) the source boxes shown in the dark gray can be translated to the center of the parent receiver box (the center of the stencil). Then this is information can be passed to each child box.

by another factor of two (more precisely, to 61 $S|R$ translations per receiver box). However, that stencil requires a more complicated internal data structure (domains composed of two and four children boxes) and the GPU code runs a bit slower than the one for the stencil described above, due to more extensive global memory access. In the implementation described in this paper we limited ourselves with the 119 translation implementation.

A theoretical justification of this scheme follows from the error bounds provided below. Introduction of the stencil modifies the 2-step downward pass of the FMM as follows (the white source boxes in Fig. 4 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 whose centers are separated by a distance of 2 box units (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 Ω_1 converge 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

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

where C is some constant. Note that $\eta(\Omega_1, \Omega_2)$ is a purely geometrical quantity (see Fig. 5), 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. The error due to translation of the expansion of a single source will be a function of η and for a given $\eta < 1$ we can determine the number of terms to guarantee the required accuracy. A larger η, means that 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. 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 needs to be checked to stay within the specified error bounds is

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

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 that include the source and receiver boxes are $R_1 = R_2 = d^{1/2}/2$. Therefore, the standard η_* that determines the length of expansion for a specified error is

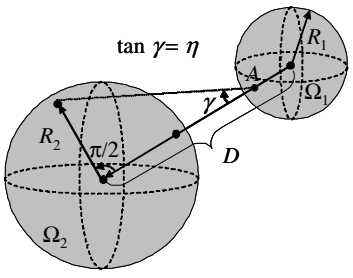

Fig. 5. Illustration of the S/R translation from source domain Ω_1 of radius R_1 to receiver domain Ω_2 of radius R_2 which centers are separated by distance D. The parameter η can be interpreted as the tangent of angle γ shown in the figure, where A is the point of Ω_1 closest to Ω_2.

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

The translation stencil scheme suggested above then can be justified, as in this case we have $R_2 = 3^{1/2}$, $R_1 = 3^{1/2}/2$ and one can check directly that any gray box in Fig. 4 has a distance from the stencil center greater than

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

Consider now the general case that includes both the white and gray boxes in Fig. 4. To provide the same error as for the closest boxes, it is sufficient to use the 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.

Remark 1. In fact this algorithm is to be preferred in the sense that the errors actually achieved are closer to the specified error ϵ (while not exceeding it); and wasteful extra precision computations are not performed.

Remark 2. Translation with reduced p means use of rectangularly truncated translation matrices $p^2 \times p'^2$ that act on vectors of size p'^2 and produce vectors of size p^2.

2.3. Computation of gradient

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

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

where $\mathbf{D}_n^m$ is a real three-dimensional vector, whose x, y, and z components correspond to the components of the gradient. It is clear that such a representation should hold since derivatives of a 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 [11]. Such recursions can be found elsewhere (e.g. [6,12]). Note however, that one should convert these formulae to the real basis according to the 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 the 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 that 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

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

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

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

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:

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

This provides an optimal time

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

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 most computational work in the FMM is related to the $S|R$ translations. 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, was the current generation of NVIDIA GPU at the time this paper was submitted, 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 768 MB, 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 64 kB over all multiprocessors, of which 8 kB 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 by NVIDIA [1]. This model is supposed to be extended over the next few generations of processors, making investment of effort on programming it worthwhile. Under CUDA the GPU is a compute device that is a highly multithreaded coprocessor. A thread block is a batch of threads that each execute on a multiprocessor and 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 (one, two or three indices). The choice of 1,2 or 3D index layout 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 indices.

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 [14] 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 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. Routines for the computation of dot products and matrix vector products are available in the NVIDIA SDK, but require storage of the whole matrix. This is not the case in this problem where the matrix entries are computed "on the fly" as needed, and matrix elements are not stored.

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 determined 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 remaining 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 of 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 shared memory.
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 of the remaining 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 the 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 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). Large times given in italic are estimates of the CPU time based on N^2 scaling. 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 straightforward CPU implementation to guide GPU code development.

The table and the associated graph for the potential calculation (see 6) 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. 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 may be explained by analyzing the algorithm. 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 a full load. Second, as mentioned earlier, there exists an overhead in the computations related to

Table 1
Comparison of direct computations on the CPU & GPU Exponents in parentheses; italics indicate estimates rather than computations

N	Potential			Potential + force		
	CPU (s)	GPU (s)	Ratio	CPU (s)	GPU (s)	Ratio
1024	0.0502	3.246(−4)	154	0.102	4.86(−4)	209
2048	0.2017	7.959(−4)	253	0.4095	9.53(−4)	429.5
4096	0.799	2.201(−3)	362	1.633	2.99(−3)	546
8192	3.185	6.851(−3)	465	6.541	9.86(−3)	663
16,384	12.76	2.424(−2)	527	26.155	3.49(−2)	750
32,768	51.06	7.947(−2)	642	104.6	0.138	758
65,536	204	0.3452	592	*418.5*	0.502	834
131,072	817	1.361	600	*1674*	1.986	843
262,144	*3268*	5.361	609.5	*6696*	7.892	848
524,288	*1.307(4)*	21.345	612	*2.678(4)*	31.41	853
1,048,576	*5.228(4)*	85.205	614	*1.071(5)*	125.36	855

read/write operations from main memory. This overhead is proportional to the number of sources and receivers. In other words, the read/write time scales linearly with N. Third, some time that is much larger than the time needed for 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. 6. 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 step that computes the local interactions 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, which controls the amount of the interactions computed directly and via the FMM approximation. Our local interaction algorithm is in many aspects similar to the total direct summation algorithm described above, but with several necessary modifications, dictated by the FMM structure. The impact of these modifications on the algorithm performance is considered below.

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

Our data structures for organizing the source and receiver data provide hierarchical spatial ordering based on bit-interleaving and ensures that all data 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 list that provides the minimum and maximum source indices for a given box, where all the source indices are stored in long array in global memory. The same remark applies to the receiver boxes.

Since the neighbor relation is not hierarchical (like the child-parent relationship), an additional data structure that 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.

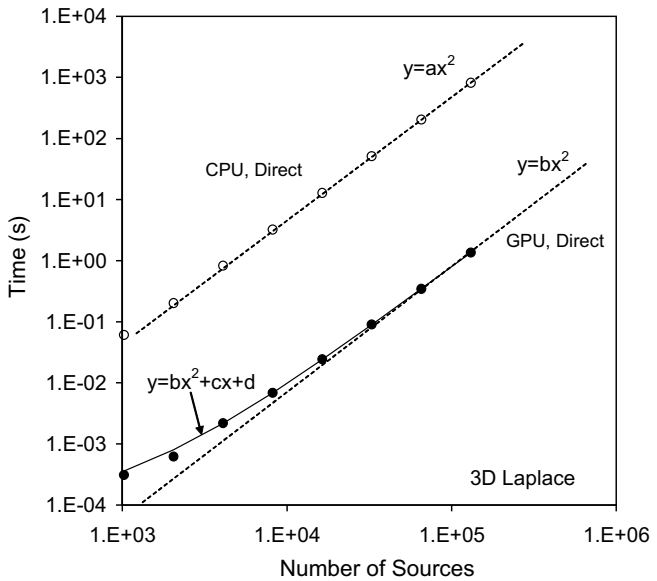

Fig. 6. Comparison of the time required for solution of the same problem (direct summation (1)) using a serial CPU code and the GPU. N sources and $M = N + 1$ receivers are unstructured sets of random data. In this figure, and in subsequent figures, we present functional fits as $y=f(x)$, where y is the time, and x is the number of particles (N).

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

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

These modifications substantially slow down the performance. Indeed due to the 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 source data for all receivers in a box are the same, and we choose to use a block of threads to deal with each receiver box, since we can then put the source data into shared memory and employ as many threads as possible. As mentioned earlier, due to the non-hierarchical nature of the neighbor relation, initial access to any box should be considered as a random data access, and these reads each pay the corresponding initial access penalty.

4.2.1. Model for run times of the sparse matrix vector product

We use actual run times for the sparse matrix vector multiplier FMM evaluation of (1) to develop a theoretical performance model that can then be used to determine optimal settings. We first consider the performance for the values of the clustering parameter s, or $l_{\max}$ that are *optimal for the CPU* ($p = 8$ and $p = 12$ for which the CPU optimal values of s are $s \leqslant s_{\max} = 60 - 80$). At optimality the sparse and dense matrix vector costs balance, though since the level changes discretely over integers, in practice the balance is approximate. Fig. 7 shows the performance for this case.

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 GPU acceleration of the FMM, at this choice of the CPU optimal s, is very nonuniform. It varies by an order of magnitude for the same $l_{\max}$. Of course, if we had not decided to fix $s_{\max}$ the value of $l_{\max}$ would not change. The GPU timing for a given $l_{\max}$ changes slowly with increase of data size. The "nonuniform" acceleration in this case is because on the CPU (at its optimal setting) the performance is determined by the data size, while on the GPU (at a nonoptimal setting) it is determined by the depth of the octree ($l_{\max}$).

Eq. (26) provides the time for the CPU implementation, where the cost of data access is assumed negligible. This dependence shows that at fixed $l_{\max}$ the relative cost of the local sum increases and the overall algorithm time is proportional to N^2, which is in very good agreement with the experimental results. However, in 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 data access. 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 $\sim 8^{l_{\max}}$, 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 represents the number of arithmetic operations and similar to (26). The second term represents the number of boxes and characterizes the time for random memory access. The last term is the cost of reading/writing of data sets of

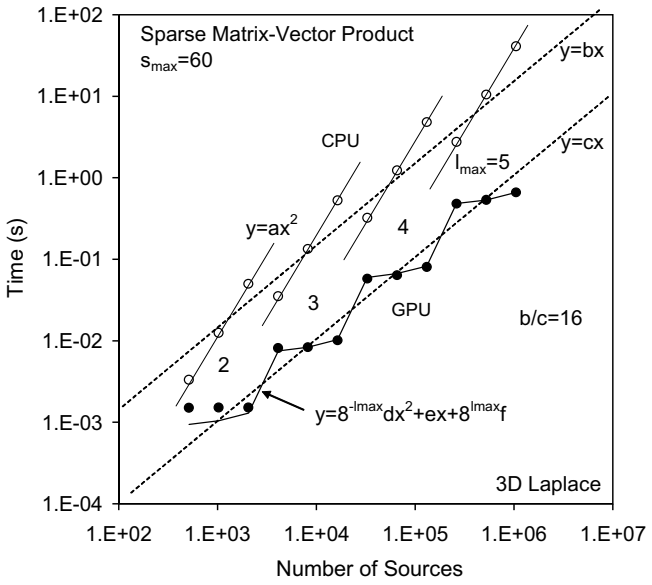

Fig. 7. Performance of sparse matrix-vector multiplier used in the FMM for maximum cluster parameter $s_{\max} = 60$. The numbers 2,...5 near the curves show the maximum level of octree space subdivision $l_{\max}$. Solid and dashed lines show different data fits.

length N in a sequential manner or for fast access to the shared memory or other coalesced read/write. Model constants can be evaluated from experimental data and the fit is plotted in figure Fig. 7. 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 as well. Some deviation is observed for level 2, which may be explained by the fact that for these levels the data size becomes too small to provide a good GPU load.

The non-negligible magnitude of the second term in the right hand side of Eq. (31) changes the optimal settings for this step of the FMM on the GPU. On the CPU Eq. (26) shows that to reduce the time for this FMM step for fixed N one should decrease s (or increase $l_{\max}$) as much as possible. On the GPU Eq. (31) claims that substantial decreasing or increasing of s will increase the computation time, and there exists a *unique minimum value* of the function Time(s) with respect to s. This minimum is realized at $s = s_{\text{opt}}^{(\text{sparse})}$ and the optimum time is $\text{Time}_{\text{opt}}^{(\text{sparse})} = \text{Time}(s_{\text{opt}}^{(\text{sparse})})$, which can be expressed in terms of the hardware and algorithm related constants A, B and C as

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

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 $s_{\text{opt}}^{(\text{sparse})} = 91$, which is larger than the size of the CPU determined maximum cluster size used in the above computations. Of course, the octree is a discrete object and the number of levels is an integer. Any optimum based on a continuous model approximation cannot be achieved exactly. Therefore, the only way to change the performance for a given N is to empirically 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.

Fig. 8 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 Fig. 7). 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, as shown in (31).

It is seen that the qualitative behavior of the CPU time remains the same and is well described by Eq. (26). The qualitative behavior of the GPU time however changes, and the step function of Fig. 7 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. 7. It is clear that it is consistent with the large cluster case as well. In addition, the closeness of the GPU time to the linear dependence in this case shows 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 becomes qualitatively 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. In addition, the maximum number threads in the block on the processor we used is limited by 512, for which the performance of the GPU downgrades, and the imbalance between the random global memory access costs and arithmetic complexity becomes substantial.

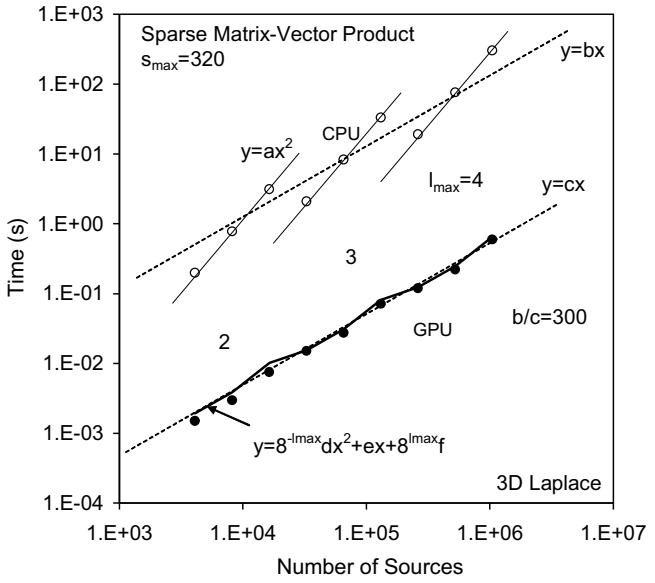

Fig. 8. The same as Fig. 7, but for $s_{\max} = 320$.

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 $\mathrm{Time}_{\mathrm{opt}}^{(\mathrm{sparse})}$ given by Eq. (32).

Because of the 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, potential computations only). 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.

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

This subroutine generates S expansions for each source box at the finest level, which is Step 1 of the FMM Upward Pass. As 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. Therefore, one solution for parallelization is to assign each thread to handle one source expansion.

Pseudocode 1 Multipole expansion generator:

{Executed in a 1-D grid of threads; each thread assigned to a source box b; sources are ordered in an array in global memory on a per box basis; `bound` contains pointers to data for each box}
`C`$(0 : p^2 - 1) =$ `0` {C is the array of multipole expansion coefficients}
for `i=bound(b) to bound(b+1)-1` **do**
`x=src(i), y=src(i+1), z=src(i+2), q=src(i+3)` {coordinates relative to center of box b}
  `GenSexp(C)` {generates coeffs for source i from 0 to $p^2 - 1$ recursively and adds to C}
end for
`scoefs`$(bp^2 : (b+1)p^2 - 1) =$ `C`$(0 : p^2 - 1)$ {write to global mem}

A drawback of this method is that after generation of expansions they need to be consolidated, which will necessitate data transfer to GPU global memory, unless they form a block of threads handled by one processor. The block size for execution of any subroutine in GPU can be defined by the user, but it is fixed during execution. In the FMM each box may have different number of sources. Thus if a source box is handled by a block of threads then threads could be idle, which, of course, reduces the utilization efficiency. GPU accelerations compared to the serial CPU code in this case are in ranges 3–7, which appear to be rather low compared with performance of other routines.

The efficiency of the S expansion generator substantially increases with a different parallelization model: one thread per box. In this case one thread performs expansion for each of s sources in the box and sums the expansion coefficients in to one vector. Therefore, one thread produces the full S expansion for the entire box. The advantage of this approach is that 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 that finishes work for a given box simply takes care of another box, without waiting or need for 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. Therefore, the method can work efficiently only for large enough problems.

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 is due to the parallelization strategy used, and as explained above a full GPU load can be achieved by this algorithm only at levels $l_{\max} \geqslant 4$. For $s_{\max} = 320$, the criterion $l_{\max} \geqslant 4$ is achieved only for $N \geqslant 2^{18} = 262,144$ and after that one can expect more or less linear dependence of the GPU time on N.

Table 2
Sparse m/v mult. at best settings

N	CPU (s)	GPU (s)	Ratio
32,768	0.322	0.0151	21
65,536	1.23	0.0275	45
131,072	4.81	0.0713	68
262,144	2.75	0.120	23
524,288	10.5	0.221	47
1,048,576	41.0	0.596	69

Table 3
Performance of S expansion generator ($p = 12$)

N	$s_{max} = 60$			$s_{max} = 320$		
	CPU (s)	GPU (s)	Ratio	CPU (s)	GPU (s)	Ratio
16,384	0.0281	0.0042	6.7	0.028	0.0234	1.2
131,072	0.229	0.0059	38.6	0.223	0.0252	8.9
262,144	0.496	0.0173	28.6	0.453	0.0105	43.3
1,048,576	1.92	0.0461	41.6	1.810	0.0381	47.4

There are further opportunities to improve performance for $l_{max} < 4$. However, the time spent for the S expansion generation usually does not exceed 2 percent of the overall FMM run time. Further the FMM on the GPU is efficient only for relatively large problems.

4.3.2. Local expansion evaluator

Pseudocode 2 Local expansion evaluator

```
{Executed in a 1-D grid of threads; each thread assigned to a receiver box b; receivers in box b are ordered in global mem}
D(0 : p^2 − 1) = rcoefs(bp^2 : (b + 1)p^2 − 1) {read expansion coefficients from global mem}
For gradient computation the grad operator matrix in terms of coefficients is applied to D here
for i=bound(b) to bound(b+1)-1 do
  x=rec[i], y=rec[i+1], z=rec[i+2] {Read coordinates, relative to box center, from global memory.}
  sum=0
  Reval(D) {evaluate basis functions from 0 to p^2 − 1 recursively
  adds basis functions times expansion coefficients to sum
  output[i]=sum
end for
```

This subroutine is part of Step 1 of the final summation. The reason why it is described out of order is that it is very similar to the S expansion generator discussed above. For parallelization of this routine the one thread per box strategy is used. 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

Pseudocode 3 LevelUp

```
{Executed on a 2D grid of threads; a thread deals with a pair of child (b_ch) \ & parent (b_par) source boxes; idx corresponds to parent; idy=0-7 to child;
threads idle for empty children}
texpan=SStrans(thread.idy,scale,expan,texpan) {S|S-translates via RCR-decomposition; RCR angles correspond to child box location; scale is
the scaling for a given level}
synchronize threads
for n=0 to p^2 − 1 do
  Consolidate coefficients 0,...,p^2 − 1 for each block of threads using array variable "shared"
  shared(threadId)=texpan(n) {each thread writes one coefficient to the shared memory based on idx,idy}
  synchronize threads
  Sum up all data with the same idx and store in sum
  if thread.idy==0 then
    scoefs(b_par p^2 + n) = sum {threads with idy=0 write result to global memory to the parent box related pointer}
  end if
end for
```

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 $l = l_{max}, \ldots, 3$ it completes the full Upward Pass of the FMM. The kernel of routine *LevelUp* consists of the $S|S$ translator, which takes as input an S expansion from one box (usually the child) and produces a new (translated) S expansion at the center of another box (the parent). To produce the S expansion for the parent box all $S|S$ translated expansion coefficients from its children boxes should be summed.

We have not yet come upon an optimal strategy for this subroutine. The current version is based on the fact that the resulting S expansions for the parent boxes can be generated independently. Therefore, 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 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. Therefore, 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. 9). A 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. 9 only one translation is shown, while 7 other translations have symmetric properties). In addition there is a central symmetry of the translation operator:

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

where $\mathbf{F}$ is a very simple constant diagonal flip operator. This means that all we need for all $S|S$ translations is one constant matrix $\mathbf{B}(\pi/4)$. This 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. The operator $\mathbf{A}(\alpha)$ is diagonal and the only input needed for its computation is the angle α, so there is no cost to read or write this operator to global memory.

The operations $\mathbf{A}(\alpha)\boldsymbol{\phi}$, $\mathbf{B}(\beta)\boldsymbol{\phi}$, $\mathbf{F}\boldsymbol{\phi}$ and $\underline{(\mathbf{S}|\mathbf{S})}(t)\phi$ are implemented as static inline functions in CUDA extended C 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 shows performance of the CPU and GPU subroutines to perform the entire Step 2 of the Upward Pass (so the *LevelUp* is called $l_{\max} - 2$ times). The execution time here is presented here 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 is approximately 8^l for level l,and this value is used a benchmark.

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. Therefore, 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. Note that the speedups are uniform for all p considered.

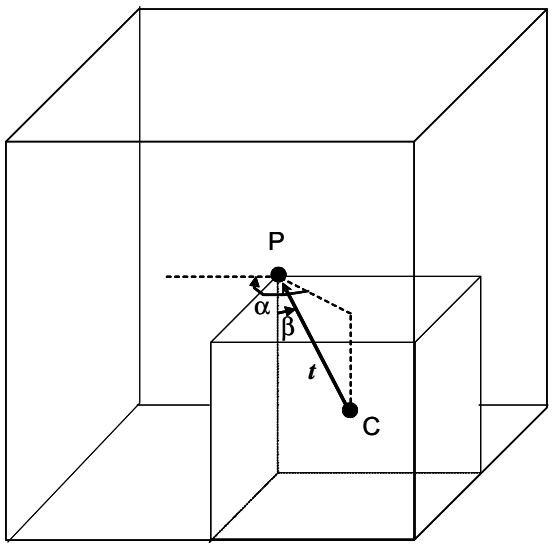

Fig. 9. Translation from the center of a child box (C) to its parent (P). α and β are Euler rotation angles and $\mathbf{t}$ is the translation vector.

Table 4
Performance for Step 2 of the FMM upward pass

l_{max}	$p=4$			$p=8$			$p=12$		
	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 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

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 did before for the sparse matrix vector multiplier (Table 5). 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 that are not optimal. The speedups then appear to be in the same range as for other routines.

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 investing substantial resources in achieving high speedups for this step.

Pseudocode 4 *LevelDown*1

{Each thread handles one receiver box b_r; alpha rotation data in constant memory `Arot`}
put normalized coaxial translation coefficients to the shared memory ("`shared`")
`rexpan`$(0:p^2-1)$=0
for `itrans=tbound(rbox) to tbound(rbox+1)-1` **do**
  {loop is over all translations performed directly to a given receiver box}
  Using `itrans` determine pointers to rotation (`Rot`), source box (b_s), and truncation number ($\mathtt{pvar}=p'$)
  `expan`$(0:p'^2-1)=$`scoefs`$(b_s p^2 : b_s p^2+p'^2-1)$ {read S-expansion coefficients from global mem}
  `SRtrans(scale,pvar,ARot,Rot,shared,expan,rexpan)` {SR-translates via the RCR-decomposition; `scale` is the scaling for given level; generated R-expansion added to `rexpan`}
end for
`rcoef`$(b_r p^2 : (b_r+1)p^2-1) =$ `rexpan`$(0:p^2-1)$

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 the fact that this subroutine is thought of as a single entity, in our code it is actually comprised of three global device functions, *LevelDown*1, *LevelDown*2 and *LevelDown*3, which correspond to the stencil translation scheme. *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. 4), 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. 4), 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.

Pseudocode 5 *LevelDown*2

{Each block of threads handles one parent receiver box b_r, each thread handles one translation; alpha rotation data in constant memory `Arot`}
put normalized coaxial translation coefficients to the shared memory ("`shared`")
`rexpan`$(0:p^2-1)=0$
if `threadID` < `ntrans2(rbox)` **then**
  {`ntrans2` is the number of translations to the parent box}
  get pointers to rotation data (`Rot`), source box (b_s) and truncation number (`pvar` $= p'$)
  `expan`$(0:p'^2-1)=$ `scoefs`$(b_s p^2 : b_s p^2 + p'^2 - 1)$ {read S-expansion coefficients from global mem}
  `rexpan+=SRtrans(scale,pvar,ARot,Rot,shared,expan)` {$S|R$-translates via the RCR-decomposition; `scale` is the scaling for given level; generated R-expansion added to `rexpan`}
  synchronize threads
  for n=0:p^2-1 **do**
    {consolidate coefficients}
    `shared(threadId)=texpan(n)` {each thread writes one coefficient to shared mem based on thread id}
    synchronize threads
    sum up coefficients for all threads {reduction}
    if threadId==0
      `rcoef`$(b_r p^2 : (b_r+1)p^2 - 1) +=$ `rexpan`$(0:p^2-1)$
    end if
    {threads with `id=0` add the result to the parent box data in the global memory}
  end for
end if

The *LevelDown*3 subroutine is very similar to *LevelUp*, since it performs $R|R$ translations from parent box to its 8 children. Therefore, 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.

Efficient parallelization is more difficult for *LevelDown*1 and *LevelDown*2. We tried several translation schemes that demonstrated that the speedups achieved on these routines is only in the range 2-5 and we still are looking for better solutions. Our current version of these subroutines work as follows.

*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 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 a decomposition similar to (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 can fit neither in shared nor in constant memory. (This fact is an indication that research on more compact translation operator decompositions of the same or better complexity is necessary. Ideas for some such schemes are discussed in [11] and, even if they are not the best for CPU implementations, in future research we are going to try to implement them on the GPU). Indeed, in the stencil there are 48 different values of the angle α, 49 different values of the angle β and 14 different translation distances. To reduce the amount of translation data read from the global memory, all α's and translation distances are put in constant memory. Only one $(S|R)$ coaxial translation matrix needs to be loaded in addition, and it is rescaled each time to produce actual coaxial operator. This is based on use of the 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 whose nth subspace 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 presented by the operator $\mathbf{B}(\beta)$. In principle, it can also be decomposed into the product of two constant matrices that cause an axis flip, and a diagonal matrix. However, tests on the CPU showed that this method is not efficient on the CPU and we did not proceed in this way. This operator has a block dense matrix representation (dense for each rotation subspace) and its size for $p = 12$ is 1152 floats. 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 NVIDIA 8800 GTX GPU we use. If hardware with sufficient fast access memory becomes available, the algorithm can be modified and we expect sensible and more usual GPU accelerations.

Pseudocode 6 *LevelDown3*

{Executed on a 2D grid of threads; idx is assigned to parent receiver box b_{par}, while idy=0-7 to a child b_{ch}}
`expan`$(0:p^2-1) =$ `rcoefs`$(b_{par}p^2:(b_{par}+1)p^2-1)$ {read parent expansion coefficients from global mem}
`RRtrans(thread.idy, scale,expan, texpan)` {RR-translates via RCR-decomposition; RCR angles correspond to child box location; `scale` is the scaling for a given level; generates `texpan`}
`rcoefs`$(b_{ch}p^2:(b_{ch}+1)p^2-1) +=$ `texpan`$(0:p^2-1)$ {add the result to the child box data in the global memory}.

Another peculiarity of the $S|R$ translations is the number of input data reads needed for data corresponding to boxes in the data structure "neighborhood of the parent box at children level". Since the algorithm is adaptive, for every translation we need data on the source box and its translation parameters. Despite the fact that we are using partially ordered lists, the access pattern is rather random, which adds to the cost. Our translation data structure represents translations as a graph and each entry to the list corresponds to an edge of this graph. Based on the entry it allows retrieval of source and target indices and an encapsulated "translation index." This index can be unpacked to provide pointers to the α, β and t-indices of global arrays storing respective rotation and translation data. The test results are provided in the tables below.

Table 6 similarly to Table 4 shows relatively low accelerations of this part of the FMM. 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 compute them at higher rate. Moreover for low $l_{\max}$ the GPU subroutine may even run slower than the serial CPU (!). However, even in this case we do not advise to switch between the CPU and GPU, since such a switch involves use of the slowest memory copying process (CPU-GPU), which should be avoided 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 as impressive as for other subroutines, they show that it is preferable to run even the most expensive (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 $l_{\max}$ 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 (See Table 7).

4.4. Overall performance

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

Table 6
Performance for the FMM downward pass

$l_{\max}$	$p=4$			$p=8$			$p=12$		
	CPU (s)	GPU (s)	Ratio	CPU (s)	GPU (s)	Ratio	CPU (s)	GPU (s)	Ratio
3	0.031	0.0146	2.1	0.093	7.71E−02	1.2	0.218	0.233	0.9
4	0.203	0.0614	3.3	0.936	2.65E−01	3.5	2.39	0.718	3.3
5	1.86	0.359	5.2	8.55	1.80	4.8	21.9	4.84	4.5

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	0.031	4.83E−03	6	0.0928	0.0231	4	0.218	0.0622	4
4	3	0.203	0.0146	14	0.936	0.0771	12	2.39	0.233	10
5	4	1.86	0.0614	30	8.55	0.265	32	21.9	0.718	30
6	4	1.54	0.0614	250	N/A	N/A	N/A	N/A	N/A	N/A

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

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

where

$$\epsilon_2^{(\mathrm{abs})} = \left[\frac{1}{M}\sum_{j=1}^{M} |\phi_{\mathrm{exact}}(\mathbf{y}_j) - \phi_{\mathrm{approx}}(\mathbf{y}_j)|^2\right]^{1/2}, \quad |\phi_{\mathrm{exact}}(\mathbf{y})|_2 = \left[\frac{1}{M}\sum_{j=1}^{M} |\phi_{\mathrm{exact}}(\mathbf{y}_j)|^2\right]^{1/2}, \tag{37}$$

where $\phi_{\mathrm{exact}}(\mathbf{r})$ and $\phi_{\mathrm{approx}}(\mathbf{r})$ 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 [13] 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.

This table demonstrates overall speedups of the entire FMM run 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, 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, even for the best possible translation methods we cannot expect an increase of the overall performance by orders of magnitude, and at most by a factor of approximately 2. This is totally different from 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, the errors due to truncation of infinite series, which are controlled by specification of the truncation number p, and, second, the 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. The error measure used is ϵ_2 in Eq. (36). The "exact" solution was computed on the CPU using double and quadruple-precision and direct summation (double precision and quadruple-precision provided consistent results that 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. 10.

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 in the FMM errors is explained 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 that increases with N. We relate this to accumulation of roundoff errors in direct summation when computing large sums.

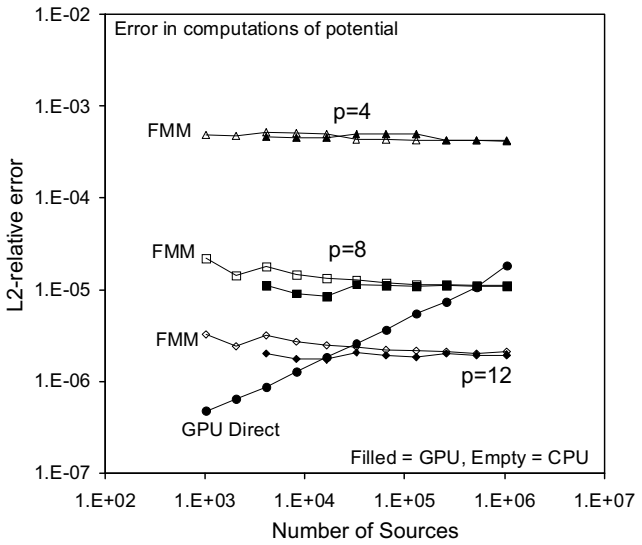

Fig. 10. ϵ_2 error (see (36)) for evaluation of sum (1) using the FMM on CPU (filled markers) and on GPU (empty markers). Also the error for direct summation on the GPU is shown. The errors are obtained by comparisons with the double precision direct computations on CPU for a grid of N sources of intensity 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 are 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). Therefore, 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. 10 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 9 shows some numbers obtained for $N = 2^{20}$. 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 9 to see the cost of computations of gradient (See Table 9).

Fig. 11 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 cache memory management and SSE instructions (we compared with a naive straightforward implementation, when the direct sum routine simply executes a memory aligned 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 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 $N = 10^3$. Therefore, 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 sum evaluator for the GPU crosses the dependence for serial CPU. This point is at $N \sim 2 \times 10^5$, 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

Table 9
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

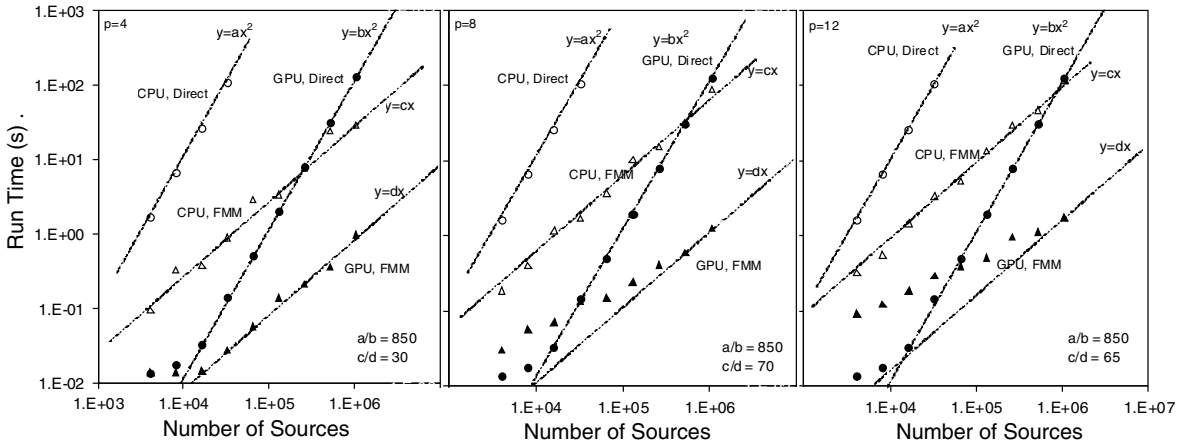

Fig. 11. FMM wall clock time (in seconds) for serial CPU code (one core of 2.67 GHz Intel Core 2 extreme QX is employed) and for GPU (NVIDIA GeForce 8800 GTX) for different truncation numbers p (potential+gradient). Also direct summation timing is displayed for both architectures. No SSE optimizations for the CPU were used.

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 second time. But the FMM on GPU performs this computations for times less than for 1 second (0.98 seconds). It is accepted by several researchers in the N-body community (e.g. [26]) 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. [16]. 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 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 much 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. Therefore, 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 [26] on a single GPU and showed that it is feasible to program a complicated algorithm such as 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, while the cost of the far-field evaluation decreases with increasing cluster size. Accordingly, 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 evaluation). For optimality, the cluster size should be larger than certain value determined by the hardware. This optimal size is larger than that on the CPU, and, correspondingly, the FMM is faster than the direct product on the GPU only for larger problem sizes. The FMM should be run on the architectures at these optimal settings. The increase of the cluster size on the GPU also results in more effective acceleration of all other components of the FMM and improves the accuracy of the computations, since a larger fraction of points are computed directly.

Parallelization Strategies for the FMM: Each of the FMM operations require differing amounts 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, and more complex parallelization is needed. 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.

Acknowledgments

We would like to acknowledge funding of this work via award NNX07CA22P from NASA to Fantalgo, LLC. In addition NG was also partially supported by the Center for Multiscale Plasma Dynamics (CMPD), a Department of Energy Fusion Science Center, during the writing of this paper.

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, I.A. Stegun, Handbook of Mathematical Functions, Dover Publications, 1965.
[4] H. Cheng, L. Greengard, V. Rokhlin, A fast adaptive multipole algorithm in three dimensions, J. Comput. Phys. 155 (1999) 468–498.
[5] J.J. Dongarra, F. Sullivan, The top 10 algorithms, Comput. Sci. Eng. 2 (2000) 22–23.
[6] M.A. Epton, B. Dembart, Multipole translation theory for the three-dimensional Laplace and Helmholtz equations, SIAM J. Sci. Comput. 16 (4) (1995) 865–897.
[7] D. Goeddeke, R. Strzodka, S. Turek, Accelerating double precision FEM simulations with GPUs, in: F. Hülsemann, M. Kowarschik, U. Rüde (Eds.), Frontiers in Simulation (Proceedings of ASIM 2005), SCS Publishing House, 2005, pp. 139–144.
[8] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, J. Comput. Phys. 73 (1987) 325–348.
[9] L. Greengard, W.D. Gropp, A parallel version of the fast multipole method, Comput. Math. App. 20 (1990).
[10] L. Greengard, V. Rokhlin, A new version of the fast multipole method for the Laplace equation in three dimensions, Acta Numer. 6 (1997) 229–269.
[11] N.A. Gumerov, R. Duraiswami, Fast Multipole Methods for the Helmholtz Equation in Three Dimensions, Elsevier, Oxford, UK, 2005.
[12] N.A. Gumerov, R. Duraiswami, Fast multipole method for the biharmonic equation in three dimensions, J. Comput. Phys. 215 (1) (2006) 363–383.
[13] N.A. Gumerov, 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>).
[14] N.A. Gumerov, R. Duraiswami, W. Dorland, Middleware for programming NVIDIA GPUs from Fortran 9X. (poster presentation at Supercomputing 2007. <www.umiacs.umd.edu/~ramani/pubs/Gumerov_SC07_poster.pdf>).
[15] T. Fukushige, J. Makino, A. Kawai, GRAPE-6A: A single-card GRAPE-6 for parallel PC-GRAPE cluster systems, PASJ: Publ. Astron. Soc. Jpn. 57 (2005) 1009–1021.
[16] T. Hamada. T. Iitaka, The Chamomile scheme: an optimized algorithm for N-body simulations on programmable graphics processing units (posted on arXiv: astro-ph/0703100v1, 6 March 2007).
[17] J.F. Leathrum, J.A. Board, The parallel fast multipole algorithm in three dimensions, Technical report, Duke University, April 1992.
[18] J. Makino, T. Fukushige, M. Koga, K. Namura, GRAPE-6: Massively-parallel special-purpose computer for astrophysical particle simulations, Publ. Astronom. Soc. Jpn. 55 (2001) 1163–1187.
[19] L. Nyland, M. Harris, J. Prins, Fast N-body Simulations with CUDA, in: Hubert Nguyen (Ed.), GPU Gems 3, 31, 677–695. Addison Wesley, August 2007.
[20] J.P. Singh, C. Holt, J.L. Hennessy, A. Gupta, A parallel adaptive fast multipole method, in: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, December 1993.
[21] J.P. Singh, J.L. Hennessy, A. Gupta, Implications of hierarchical N-body methods for multiprocessor architectures, ACM Trans. Comput. Syst. 13 (2) (1995) 141–202.
[22] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, 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.
[23] 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. 28 (2007) 2618–2640.
[24] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, Timothy J. Purcell, A survey of general-purpose computation on graphics hardware, Comput. Graphics Forum 26 (2007) 80–113.
[25] 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.
[26] S. Warren, M, K. Salmon, J, J. Becker, D, P. Goda, M, T. Sterling, Pentium Pro inside: I.A treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac,” in: Proceedings of the Supercomputing 97, in CD-ROM. IEEE, Los Alamitos, CA, 1997.
[27] C.A. White, M. Head-Gordon, Rotation around the quartic angular momentum barrier in fast multipole method calculations, J. Chem. Phys. 105 (12) (1996) 5061–5067.
[28] M.J. Stock, A. Gharakhani. Toward efficient GPU-accelerated N-body simulations, in: 46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-608, January 2008, Reno, Nevada.
[29] S.F. Portegies Zwart, R.G. Belleman, P.M. Geldof, High-performance direct gravitational N-body simulations on graphics processing units, New Astron. 12 (8) (2007) 641–650.