INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2012; **89**:105–133
Published online 3 August 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nme.3240

Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units

Toru Takahashi[1,*,†], Cris Cecka[2], William Fong[3] and Eric Darve[4]

[1] *Department of Mechanical Science and Engineering, Nagoya University, Nagoya 464-8603, Japan*
[2] *Institute for Computational and Mathematical Engineering, Stanford University, Stanford, CA 94305-4040, U.S.A.*
[3] *Air and Missile Defense Department, Johns Hopkins University, Laurel, MD 20723-6099, U.S.A.*
[4] *Department of Mechanical Engineering, Stanford University, Stanford, CA 94305-4040, U.S.A.*

SUMMARY

This paper presents a number of algorithms to run the fast multipole method (FMM) on NVIDIA CUDA-capable graphical processing units (GPUs) (Nvidia Corporation, Sta. Clara, CA, USA). The FMM is a class of methods to compute pairwise interactions between N particles for a given error tolerance and with computational cost of $\mathcal{O}(N)$. The methods described in the paper are applicable to any FMMs in which the multipole-to-local (M2L) operator is a dense matrix and the matrix is precomputed. This is the case for example in the black-box fast multipole method (bbFMM), which is a variant of the FMM that can handle large class of kernels. This example will be used in our benchmarks.

In the FMM, two operators represent most of the computational cost, and an optimal implementation typically tries to balance those two operators. One is the nearby interaction calculation (direct sum calculation, line 29 in Listing 1), and the other is the M2L operation. We focus on the M2L. By combining multiple M2L operations and reordering the primitive loops of the M2L so that CUDA threads can reuse or share common data, these approaches reduce the movement of data in the GPU. Because memory bandwidth is the primary bottleneck of these methods, significant performance improvements are realized. Four M2L schemes are detailed and analyzed in the case of a uniform tree.

The four schemes are tested and compared with an optimized, OpenMP parallelized, multi-core CPU code. We consider high and low precision calculations by varying the number of Chebyshev nodes used in the bbFMM. The accuracy of the GPU codes is found to be satisfactory and achieved performance over 200 Gflop/s on one NVIDIA Tesla C1060 GPU (Nvidia Corporation, Sta. Clara, CA, USA). This was compared against two quad-core Intel Xeon E5345 processors (Intel Corporation, Sta. Clara, CA, USA) running at 2.33 GHz, for a combined peak performance of 149 Gflop/s for single precision. For the low FMM accuracy case, the observed performance of the CPU code was 37 Gflop/s, whereas for the high FMM accuracy case, the performance was about 8.5 Gflop/s, most likely because of a higher frequency of cache misses. We also present benchmarks on an NVIDIA C2050 GPU (a Fermi processor)(Nvidia Corporation, Sta. Clara, CA, USA) in single and double precision. Copyright © 2011 John Wiley & Sons, Ltd.

Received 6 November 2010; Revised 24 April 2011; Accepted 8 May 2011

KEY WORDS: fast multipole method (FMM); graphical processing units (GPUs); Nvidia CUDA; high performance computing (HPC)

1. INTRODUCTION

1.1. Background of fast multipole method on graphical processing units

The fast multipole method (FMM), introduced by Rokhlin and Greengard in [1,2], is a fast algorithm to sum the pairwise interactions between many bodies. It has been successfully applied to classical

*Correspondence to: Toru Takahashi, Department of Mechanical Science and Engineering, Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan.

†E-mail: ttaka@nuem.nagoya-u.ac.jp

particle simulations and solving integral equations [3]. A currently active area of study in the FMM is its generalization [4–7]. The purpose is to construct an FMM that can handle an entire class of kernels in a single FMM-like formulation and implementation. Such an FMM is often called a kernel-independent FMM.

Similar to other variants proposed so far, the bbFMM [4] can handle non-oscillatory kernels and requires $\mathcal{O}(N)$ operations. The advantages of the bbFMM are that it is applicable to a wide range of kernels, only requires a user-supplied program sub-routine that evaluates numerically the kernel (no analytical expansion is required), requires a small precomputation time, and can be proved to minimize the number of coefficients for the far field representation (for the L^2 norm).

Another active area of research is parallelization of the FMM. Since early works [8, 9], a variety of parallel versions of the FMM have been proposed, implemented, and tested in various computing environments (see [10], for example). Recently, commodity graphical processing units (GPUs) are being adopted as a compute resource for the FMM as well as other scientific applications.

The emergence of GPUs as scientific computing platforms is due to its high computational power and cheap cost. As of August 2010, the most powerful graphics board is AMD's Radeon HD5970 (AMD, Sunnyvale, CA, USA) (with dual GPUs) [11]: the peak floating-point arithmetic performance sustains 4.64 Tflop/s and 928 Gflop/s in the single and double precision, respectively; the memory bandwidth is 256 GB/s (GDDR5); the price is about $650 with 2 GB RAM. These numbers are much better than high-end CPUs. For example, AMD's twelve core CPU Opteron 6176 SE 2.3 GHz (Magny-cours) sustains 220.8 Gflop/s for single-precision floating-point performance, and 43 GB/s of bandwidth for $1386 (as of June 23, 2010 [12]). Also, Intel's six core CPU Xeon X5680 (Intel Corporation, Sta. Clara, CA, USA) 3.33 GHz (Westmere-EP) sustains 159.84 Gflop/s for single-precision floating-point performance, and 32 GB/s of bandwidth for $1663 (as of July 18, 2010 [13]). Although, we must bear in mind that GPUs are coprocessors to CPUs, not replacements. From a programming perspective, NVIDIA's GPUs are preferable to AMD's GPUs for computational physics applications. We can program NVIDIA's GPUs using the C/C++ based computing language CUDA [14]. With the development of OpenCL [15], a language for parallel programming for heterogeneous systems, this GPU specific programming model may not be an issue in the future.

1.2. Related works for fast multipole method on graphical processing units

In 2008, Gumerov *et al*. pioneered an FMM code running on GPU [16]. The three dimensional Laplace kernel FMM achieved a speed up of 30 to 70 times (depending on the accuracy) on a single NVIDIA GPU. For the *multipole-to-local* (*M2L*) *operation*, they applied the rotation coaxial-translation rotation decomposition of White and Head-Gordon [17], a technique that rotates the spherical harmonics to lower the computational cost. In addition, the maximum number of M2L operations per receiver box (target box in our terminology) was reduced from 189 to 119 by means of a stencil structure. Their elaborate M2L improves on the original M2L on CPU, but it was challenging to obtain an efficient implementation on GPU. Their M2L implementation showed relatively low acceleration compared with other parts of the FMM.

Yokota *et al*. implemented the FMM and a variant (pseudo particle multipole method) for the Laplace kernel on a PC cluster with 256 GPUs [18]. They basically followed the FMM in Gumerov and Duraiswami [16], but independently developed an FMM code on their machine. Their FMM on a single GPU was approximately 80 times faster than using two CPU cores. In addition, the FMM on 32 GPUs had 66% parallel efficiency with 10^7 particles. This work was investigated by Hamada *et al*. in conjunction with their high-performance GPU implementation of a tree code [19].

The paper by Lashuk *et al*. [20] has some similarities with this paper. The kernel-independent FMM proposed by co-authors Ying *et al*. [6, 21] was implemented on multiple GPUs. However, their M2L operation differs from ours. The GPU was applied to the diagonal translation step of their M2L, which is a vector–vector multiplication. Thus, as mentioned in the paper, the calculation was less efficient on the GPU because the ratio between arithmetic computation and memory traffic is small. In a work related to Lashuk *et al*. [20], Chandramowlishwaran *et al*. [22] studied an optimization of the M2L operation (V-list in their terminology) and the near interaction calculation (U-list)

DOI: 10.1002/nme

for the kernel-independent FMM of Ying *et al*. [6] for multicore systems, and compared their results with [20] using GPUs.

Cruz *et al*. in [10] discussed developing computations on the GPU in their parallelized FMM library, but the details, as far as we know, have not been unveiled.

1.3. Present work

In this paper, we investigate implementations of FMMs in which the M2L operator is a dense matrix, such as in bbFMM [4], on an NVIDIA GPU. We will focus on the case of a uniform tree, in which the leaf boxes have all the same size. We will refer the reader to Greengard and Lashuk *et al*. [20, 23, 24] for a discussion of adaptive FMMs and their parallel implementation.

We will focus on strategies for performing the expensive M2L operation on a GPU. Thus, this work is influenced by a paper written by Coulaud *et al*. [25] in which they proposed efficient algorithms and implementations of the M2L operation (for the Laplace kernel) using the Basic Linear Algebra Subprograms (BLAS) library [26] on a single CPU. Their basic idea is to concatenate a number of M2L operations, each of which can be expressed as a matrix–vector product, into a single matrix–matrix product and then to perform the product with a level 3 BLAS routine (GEMM). This allows overlapping memory access with computations because the matrix–matrix products require $\mathcal{O}(n^3)$ operations, whereas the memory transfer requires only $\mathcal{O}(n^2)$ operations (where n is the size of matrices). The authors made it possible to aggregate many more M2L operations by considering a number of schemes to layout the multipole moments and local coefficients in memory. Also, they proposed implementing their FMM code on shared and distributed architectures, including GPU architectures with the help of BLAS library, but results have not yet been reported to our knowledge. In [27], Couland *et al*. extended their formulation to an adaptive version of the FMM.

Similar to [25], we design methods that allow many M2L operations to be performed at once in the form of a matrix–matrix product. However, in contrast to [25], we must keep in mind that a GPU has many concurrent processing cores and that the memory storage per core is much smaller than that of a typical CPU.

The key difficulty on a GPU is that using the efficient BLAS routines now available on these platforms does not lead to significant speed-ups. The reason is that the matrices involved in the M2L operations are too small to efficiently use the hardware, and matrix vector products are typically slow because they involve few flops per word read from memory. Consequently, we investigate different blocking techniques that leverage the specific pattern of the M2L interaction list to increase the flop to word read from memory ratio.

It is important to mention that the techniques described in this paper can be easily extended because the M2L operation of the bbFMM has a structure common to many FMMs. Similar to many other methods, the M2L operation of the bbFMM is based on the addition of matrix–vector products, where the matrices (corresponding to the M2L operators) are dense, usually small ($\lesssim 1000$), and precomputable.

The rest of this paper is organized as follows. Section 2 gives an outline of the FMM and the CPU implementation. Section 3 introduces the compute unified device architecture (CUDA). In Section 4 through 8, we propose four methods for performing the M2L operation using CUDA. In Section 9 and 10, we investigate the performance and accuracy of our GPU-accelerated bbFMM.

2. OUTLINE OF THE FAST MULTIPOLE METHOD FORMULATION AND CPU IMPLEMENTATION

2.1. Outline of the mathematical calculation and terminology

We wish to compute the matrix vector product:

$$f(\mathbf{x}_i) := \sum_{j=1}^{N} K(\mathbf{x}_i, \mathbf{y}_j)\ \sigma_j \qquad i \in \{1, \ldots, N\}, \tag{1}$$

DOI: 10.1002/nme

where $\mathbf{x}_i$ and $\mathbf{y}_j$ are N target and source points (particles), respectively, in a cubic domain $[-\frac{\ell}{2}, \frac{\ell}{2}]^3 \subset \mathbb{R}^3$ where ℓ is the prescribed edge length of the domain and σ_j are source strengths. In practice, we typically have $\mathbf{x}_i = \mathbf{y}_i$ and the sum in (1) taken with $j \neq i$. The *kernel*, K, is any smooth, real-valued, non-oscillatory function. In a direct computation, (1) requires $\mathcal{O}(N^2)$ operations. The bbFMM presented in [4] provides an $\mathcal{O}(N)$ approximation method to compute (1) by means of the Chebyshev interpolation and the singular value decomposition (SVD) for the far field evaluation associated with the FMM tree structure. We will not repeat the details of the method in the body of this paper. A short summary is provided as Appendix A.

We first clarify some of the terminology related to the FMM tree structure that will be used throughout this paper. Using a uniform octree, the cubic domain that contains all the N points is called the *box* of *level 0* or *root box*. We recursively obtain boxes of levels 1 to κ by dividing a box (*parent box*) at level k into eight cubes (*child boxes*) at level $k+1$. We call such eight boxes *siblings*. The boxes of level κ are *leaves* (see Table I).

The bbFMM follows the usual FMM pattern of computation with an upward and downward pass. From the standpoint of the GPU algorithms that we develop, the goal is to calculate products and sums of the type:

$$\mathbf{L}(T) := \sum_{S \in \mathcal{I}(T)} \mathbf{D}^{(T,S)}\, \mathbf{M}(S) \tag{2}$$

This corresponds to the M2L operator in the FMM. It will be referred to as the *M2L formula* hereafter. The notations are as follows:

$\mathbf{M}(S)$	array of multipole coefficients associated with box S
$\mathbf{L}(T)$	array of local expansion coefficients associated with box T
$\mathbf{D}^{(T,S)}$	M2L operator associated with the pair of boxes (T, S)
$\mathcal{I}(T)$	list of boxes in the interaction list of T

The interaction list $\mathcal{I}(T)$ is the set of boxes at the same level as T that of the following: 1) are not adjacent to T; and 2) whose parent box is adjacent to the parent box of T (Figure 1). There are at most 189 $(= 6^3 - 3^3)$ source boxes for each T.

A key assumption in terms of the GPU implementation is that the matrix $\mathbf{D}^{(T,S)}$ is dense, and no special mathematical relation (such as recurrence relation or other mathematical transformation) is used to accelerate this product. In that sense, the work in this paper extends to any FMM that requires such dense matrix vector products.

2.2. Implementation on CPU

We now describe a simple implementation of a bbFMM code for shared memory machines. In this paper, we mainly focus on using single-precision floating-point arithmetic because when this work

Table I. Table of common notations.

Notation	Description
$\mathbf{x}, \mathbf{x}_i$ / $\mathbf{y}, \mathbf{y}_j$	Target/source point ($1 \leqslant i, j \leqslant N$).
σ_j	Coefficient of jth source point ($1 \leqslant j \leqslant N$).
ℓ	Edge length of the computational domain.
κ	Maximum (finest) level of octree.
n	Number of Chebyshev nodes in each dimension.
r	Chebyshev truncation (terms in expansion), equal to $n^3/2$ [4].
T / S	Target/source box with center **t/s**.
$\mathcal{I}(T)$ / $\mathcal{N}(T)$	Interaction/near-neighbor list of target box T.
$\mathbf{D}^{(T,S)}$, $\mathbf{D}^{(i)}$	Multipole-to-local (M2L) operator from box S to T. $\mathbb{R}^{r\times r}$, called D-matrix or D-data ($0 \leqslant i < 316$).
$\mathbf{M}$	Multipole coefficients, $\mathbb{R}^r$, called M-vector or M-data.
$\mathbf{L}$	Local coefficients, $\mathbb{R}^r$, called L-vector or L-data.

DOI: 10.1002/nme

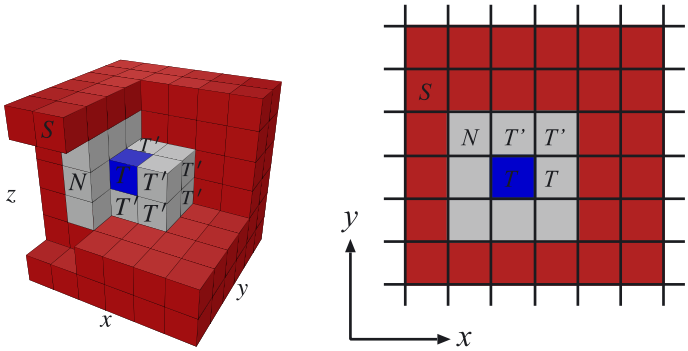


Figure 1. Example of source boxes ($\mathcal{I}(T)$; red) and near-neighbor boxes ($\mathcal{N}(T)$; gray) for a given target box (T; blue) (left: 3D, right: 2D for illustration purposes). Note that boxes denoted by T' are the siblings of T, and that T itself is included in its near-neighbor list.

was done, GPUs were an order of magnitude faster when computing in single precision rather than double precision. For example, the peak performance of NVIDIA C1060 [28], which is used in Section 9, is 933 Gflop/s in single precision, and only 78 Gflop/s in double precision. Nevertheless, double precision is also examined using an NVIDIA C2050 [29] in Section 10.

In the case of single-precision floating-point arithmetic, the accuracy of the sum in (1) saturates around $n = 8$, as investigated in [4]. In this paper, we consider $n = 8$ and $n = 4$ for comparison. The corresponding numbers of multipole coefficients or cut-off numbers r are then 256 and 32, respectively.

Listing 1 shows a pseudocode for the upward, interaction, and downward passes. The CPU implementation was not optimized beyond basic optimizations steps that include: openMP pragmas to run on multiple cores, and optimization compiler options. Details are given in Section 9.1.3.

3. COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

We briefly overview NVIDIA's CUDA-capable GPUs [14].

3.1. Hardware configuration

An NVIDIA's CUDA-capable GPU (*device*) is a programmable graphics card which acts as a coprocessor to a *host* CPU. GPUs typically have a large amount of DRAM memory (*device memory*) which interacts with the host's memory. A GPU contains multiple *streaming multiprocessors* (SMs), which are run as single instruction multiple data (SIMD) processors and consist of eight *scalar processors* (SPs) with 32-bit *registers*, *shared memory*, and an instruction unit. See Figure 2.

3.2. Programming

A GPU is a massively multithreaded coprocessor for parallel computing. Each thread executes a program called a *kernel*. Similar to an ordinary C-function, a kernel is invoked by its *host code* from the CPU, but a kernel is executed by an arbitrary number of threads on the GPU. Before invoking a kernel and after terminating it, data may be transferred between the CPU and GPU via the device memory. A kernel is written in the CUDA programming language, which is an extension of the C/C++ language.

Threads running a kernel are grouped hierarchically. At the lowest level, each thread has a register space available to it on the SM.

Then, a predefined number of threads are aggregated in a *thread-block*. Any thread within a thread-block can be identified with its thread ID, which can be used to assign different data and/or instructions to different threads. If two or more threads belong to the same thread-block, the threads can share data via shared memory on the SM. For example, a thread may write data into the shared

DOI: 10.1002/nme

```
1  // Upward Pass (create multipole moment and M2M).
2  for( int level = κ; level ⩾ 2; level-- ) {              // Loop over levels.
3    for( box B in this level ) {                           // Loop over boxes.
4      if( B is a leaf ) {
5        Compute B's multipole coefficients using sources in B.
6      } else {
7        for( child box C of B ) {                          // Loop over children.
8          Accumulate C's multipole coefficients into B's via M2M formula.
9        }
10     }
11   }
12 }
13 // Interaction Pass (M2L and post-M2L).
14 for( int level = 2; level ⩽ κ; level++ ) {              // Loop over levels.
15   for( box T in this level ) {                          // Loop over target boxes.
16     for( box S in interaction list of T ) {             // Loop over source boxes.
17       Apply M2L to S's multipole coefficients.
18       Accumulate result into T's local coefficients.
19     }
20     Post-process T's local coefficients (see [4] p. 8719, step 3c).
21   }
22 }
23 // Downward Pass (L2L and evaluate fields).
24 for( int level = 2; level ⩽ κ; level++ ) {              // Loop over levels.
25   for( box B in this level ) {                          // Loop over boxes.
26     if( B is a leaf ) {
27       Evaluate contributions to all target points in B using B's local
         coefficients.
28       for( box N in neighbor list of B ) {              // Loop over neighbors.
29         Evaluate contributions from sources in N to all target points in B
           .
30       }
31     } else {
32       for( child box C of B ) {                         // Loop over children.
33         Add contribution from B to C's local coefficients via L2L formula.
34       }
35     }
36   }
37 }
```

Listing 1. Pseudo black-box fast multipole method code.

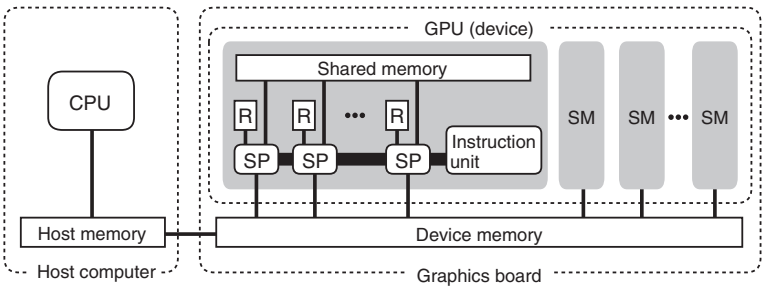


Figure 2. CPU-GPU system: R stands for register.

memory and other threads may read it (although we have to make sure that the writing precedes the reading). Cooperation of this sort is accomplished by being able to synchronize threads of a thread-block at any time.

Finally, a prescribed number of thread-blocks are aggregated in a *grid*, which is invoked when the kernel is executed. Any thread-block within a grid can be identified with its thread-block ID. Although all threads of a grid have access to device memory, it cannot be used to communicate between the threads because there is no synchronization between thread-blocks.

DOI: 10.1002/nme

3.3. Performance guidelines

We review some important and common guidelines [14], which will be used and referenced in this paper. The description that is given is for devices of compute capability 1.x (e.g., Tesla C1060). See [14] for devices of compute capability 2.x (e.g., Tesla C2050).

First is the choice of the number of threads per thread-block. A GPU can handle thousands of threads concurrently and hundreds of threads per thread-block. In hardware, an SM decomposes a thread-block into groups of 32 parallel threads called *warps*. If 32 threads in a warp execute one common instruction, the execution time is the minimum (e.g., 32 single-precision floating-point add, multiply, or multiply–add (MAD) instructions can be executed in 4 clock cycles). Otherwise, the 32 threads *diverge* (i.e., instructions are executed serially or *serialized* by the relevant threads), which results in lower performance. Therefore, it is important to choose the size of the thread-block as a multiple of warp-size (32) and ensure that all threads can execute the same operations.

Second, the memory bandwidth is probably the single most important resource on the device. An SM takes 400–600 clock cycles to read or write to the device memory. We therefore need to minimize the number of accesses to the device memory. On the other hand, the memory latency of shared memory and registers are much lower (e.g., 26 and 24 clock cycles, respectively, for G80 series [30]).

Third, we need to consider the memory access pattern. Roughly speaking, threads in a warp should read from and write to contiguous addresses in memory for *coalesced access*. If the access is not contiguous (or properly aligned within the memory), an SM serializes multiple memory transactions to serve the request, which significantly degrades performance. Similarly, threads in a *half-warp* (either the first or second half of a warp) can efficiently access data in shared memory, provided there are no *bank conflicts*, which typically occur when any two threads in the half-warp access the same shared-memory module.

Last, we may increase the number of *active thread-blocks* per SM. If two or more thread-blocks can be assigned to a single SM, the SM has a chance to switch from one thread-block (that has just begun doing memory transaction, for example) to another thread-block (that is ready to do arithmetic operations, for example) so that the use of resources on the SM is maximized. The number of active thread-blocks per SM can increase as the amount of hardware resources (shared memory, registers, counters for threads, thread-blocks, warps, etc) requested per thread-block decreases. As an example, we therefore need to write kernels that do not use excessive amounts of local variables.

4. DEVELOPMENT OF MULTIPOLE-TO-LOCAL SCHEMES FOR THE GRAPHICAL PROCESSING UNIT

Along with the nearby interaction computation, which is performed by directly evaluating the kernel, the M2L operation is the most time-consuming part of the bbFMM. Recall that all boxes of all levels of the octree perform the M2L operation, and each box computes a summation of at most (typically) 189 dense matrix–vector products, where the dimensions of the matrices are $r \times r$. Thus, we primarily explore efficient schemes to execute the M2L operation [Equation (2)] using CUDA.

Recall that the goal is to calculate the sums and products in Equation (2). This requires the four primitive loops shown in Listing 2. At the highest level, the schemes differ in the ordering and blocking of these required operations.

```
1  for all target boxes T in this level
2    for all source boxes S ∈ I(T)
3      for all i, 0 ⩽ i < r
4        for all j, 0 ⩽ j < r
5          L_i(T) += D_ij^(T,S) M_j(S)
```

Listing 2. Multipole-to-local operation for a single level.

DOI: 10.1002/nme

In this paper, we propose and review four schemes. As we will show, the bottleneck of these schemes is reading the D-matrix, which becomes central to the development of a highly efficient method. From Listing 2, we see that optimizing these reads to maximize sharing and reuse of the D-matrix requires taking advantage of the structure of $\mathcal{I}(T)$. To this end, each method incrementally introduces notations and structures that can be used to take advantage of the structure of the mentioned loops and improve on previous schemes.

The first scheme is based on the simplest approach, in which the M2L operation of each box is computed by one thread-block independent of other boxes. Although this is not a competitive scheme, we present it because of its simplicity and its potential usefulness to readers not familiar with CUDA codes.

In the second scheme, we block the loop over T and S by grouping eight sibling boxes into *clusters*. Considering now a pair of clusters, many pairs (T, S) share the same D-matrix. We therefore read a D-matrix and perform as many (T, S) interactions as possible using this D-matrix.

In the third scheme, we block computations one step further by grouping clusters into *chunks* and then performing the same optimization. That is, we load a D-matrix and perform as many (T, S) interactions as possible within a chunk.

In contrast to the two previous schemes, which put the loops over i and j as the innermost loops, the fourth scheme moves those loops to the outside, and the loops over T and S become the inner loops. By reducing the storage requirements, this allows a more aggressive blocking of the (T, S) interaction pairs.

The differences between these methods will become clearer as they are presented, but the main point is that when the ij-loops are the innermost loops (scheme 1–3), we are able to perform the computations as a series of straightforward matrix-vector products which can be computed efficiently on GPU hardware. The drawback is that the blocking is less efficient. With the fourth scheme, the blocking is most efficient with nearly maximum data reuse. However, the innermost loop of the fourth scheme is over the (T, S) pairs, which is slightly more difficult to perform on GPU hardware because of the irregularities in the interaction list. This trade-off is part of what is explored in this paper.

We now detail each scheme, presenting the technical optimizations required to achieve a nearly optimal efficiency for each. The numerical benchmarks will show that with the current generation of hardware, the fourth scheme outperforms the second and third ones, but are in a virtual tie with one another.

To compare the efficiency of these four schemes, we compare the ratio of arithmetic operations to data traffic (flop-to-word ratio) with the GPU-specific flop-to-word ratio (ratio of the peak performance in flops to the peak bandwidth of the device memory). The GPU-specific flop-to-word ratio provides an approximate target number of arithmetic operations per data that allows for the GPU to run at its peak performance [30]. If a given scheme's ratio is much less than the GPU's flop-to-word ratio, the primary bottleneck of the scheme is memory transfers.

Table II shows the GPU-specific flop-to-word ratios for a number of recent NVIDIA GPUs, in which the peak performances are for MAD (multiply–add) operations, in consideration of our matrix–vector products comprising mostly MAD operations.

Table II. Flop-to-word ratios of recent NVIDIA graphical processing units: peak performances are for the multiply–add (MAD) operation in single-precision floating-point arithmetic.

GPU model		GeForce 8800GTX [30, 31]	GeForce 9800GTX+ [32]	GeForce GTX285 [33]	Tesla C1060 [28]	Tesla C2050 [29]
Peak performance	[Gflop/s]	346	470	708	624	1030
Bandwidth	[GB/s]	86.4	70.4	159	102	115†
Flop-to-word ratio	[flop/word]	16	27	18	24	36

†Error correcting code (ECC) is enabled.

DOI: 10.1002/nme

5. BASIC (NON-BLOCKING) SCHEME

This is the simplest method. We describe it mostly for readers who are not familiar with GPUs to provide a simple algorithm that has reasonable performance. This approach will be outperformed by revisions of this algorithm to follow in subsequent sections.

For a given level $k \in \{2, \ldots, \kappa\}$, we assign one thread-block to one target box T and use r threads for each thread-block. In each thread-block, the ith thread computes the ith row of $\mathbf{L}(T)$. That is, thread i of thread-block T computes

$$L_i(T) = \sum_{S \in \mathcal{I}(T)} \sum_{j=0}^{r-1} D_{ij}^{(T,S)} M_j(S). \tag{3}$$

This scheme has a two-level parallelism: the coarse parallelism over target boxes (T), and the fine parallelism over rows (i). With this approach, we can expect a good load balance among both thread-blocks and threads.

Listing 3 shows the loop ordering for this scheme. At level k, each kernel uses 8^k thread-blocks, each of which is assigned to a target box. In the case that 8^k exceeds the maximum number of thread-blocks per grid (e.g., 65535 for the Tesla C1060), we split the boxes into smaller groups and launch the kernel multiple times until all groups have been processed.

```
1  for all target boxes T in level k          // Parallelized over 8^k thread blocks
2    for all i, 0 ⩽ i < r                     // Parallelized over r threads
3      for all source boxes S ∈ I(T)
4        Load in shared memory M_i(S).
5        for all j, 0 ⩽ j < r
6          L_i(T) += D_ij^(T,S) M_j(S)
```

Listing 3. Rough pseudocode for the first scheme single-level multipole-to-local operation.

Equation (3) implies that the M-data is common to all threads in a thread-block. Thus, we share M-data within the thread-block. To this end, we let r threads read r elements of the M-data concurrently from the device memory and write them into shared memory. See listing 7 in Appendix B for a more detailed GPU pseudocode.

Table III shows the data traffic for the device memory, the floating-point operation counts, and the method's flop-to-word ratio, which is calculated as the ratio of the operation counts to the total data traffic. Note that one matrix–vector product requires exactly $r(2r-1)$ floating-point operations (counting multiplications and additions). Note that this approach's flop-to-word ratio is much less than any GPU-specific flop-to-word ratio in Table II. Hence, the bottleneck of this scheme is the data transfer. Reading the D-data is the most problematic and motivates us to investigate other schemes.

Table III. Statistics of the first scheme (basic scheme).

		Per target box
Read M-data	[word]	$189r$
Read D-data	[word]	$189r^2$
Write L-data	[word]	r
Operation counts	[flop]	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	1.9 for $r = 32$ ($n = 4$) 2.0 for $r = 256$ ($n = 8$)

DOI: 10.1002/nme

6. SIBLING-BLOCKING SCHEME: REUSE D-DATA IN SIBLINGS

6.1. Design

To reduce the number of D-data reads, we perform the M2L operation at the grain of *clusters* instead of boxes, where we define a group of eight sibling boxes as a cluster. A pair of adjacent clusters generally includes 8×8 interactions to be computed. Among such 64 interactions, we observe that two or more interactions can typically be associated with a common M2L-transfer vector or D-matrix. Thus, we can combine the corresponding matrix–vector products into a single matrix–matrix product in which the D-data is reused.

To explain how to combine matrix–vector products, we introduce some terms. First, we define a *sibling index* (0 to 7) for every cluster as in Figure 3. Next, for any given cluster (called a *target cluster*), we define the *source clusters* as the clusters that are adjacent to the target cluster — no other clusters interact with the target cluster. There are at most 26 ($= 3^3 - 1$) source clusters for every target cluster. We identify such source clusters by a *source-cluster index* (0 to 26) defined as in Figure 4. The source cluster with the index 13 can be ignored, because it is identical to the target cluster so there is no interaction to be computed.

Table IV shows the sibling and cluster indices in the 2D case for illustration. The *interaction-kinds* listed in Table IV (left) are sets of pairs (T_i, S_j) which share the same D-data. Thus, 16 matrix–vector products are reduced to 8 matrix–matrix products. Also, note that interaction-kinds, which correspond to nearby interactions must be removed. We can specify such unallowable interaction-kinds for each source-cluster index, as shown in Table IV (right). In three dimensions, we can similarly reduce the 64 matrix–vector products to 27 matrix-matrix products. Table V (left) shows the interaction-kinds and (right) the unallowable interaction-kinds for each source-cluster index corresponding to the nearby interactions. Note that the interactions of kind 0 to 7 remain matrix–vector products because their D-matrices are not common to any other interactions.

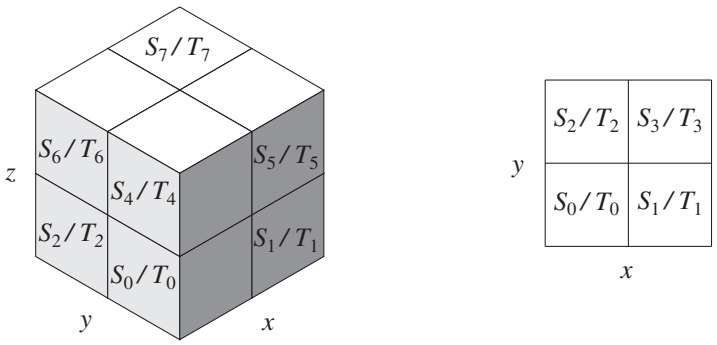


Figure 3. Definition of sibling indices in a cluster (left: 3D, right: 2D for explanation).

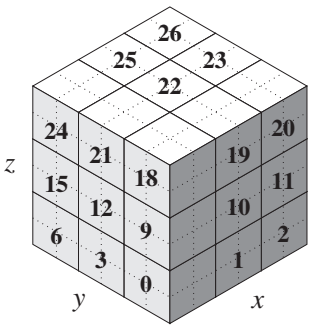
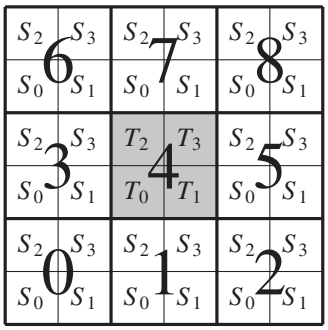


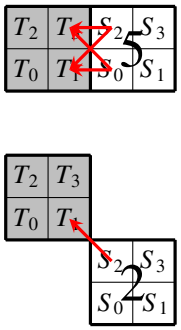
Figure 4. Definition of source-cluster indices (0 to 26): the cluster with index 13 at the center (not visible) is the target cluster.

DOI: 10.1002/nme

Table IV. To illustrate Table V, we depict the interaction-kinds in two dimensions.

Interaction-kind	Interactions (A pair 'i-j' denotes the interaction between T_i and S_j)	Source-cluster index	Unallowable interaction-kinds
0	0-3	0	0
1	1-2	1	0, 1, 5
2	2-1	2	1
3	3-0	3	0, 2, 4
4	0-1, 2-3	4	0-8 (all)
5	0-2, 1-3	5	1, 3, 6
6	1-0, 3-2	6	2
7	2-0, 3-1	7	2, 3, 7
8	0-0, 1-1, 2-2, 3-3	8	3





(Left) List of 9 2D interaction-kinds in pairs of target clusters (consisting of T_0 to T_3) and source clusters (consisting of S_0 to S_3).
(Right) List of unallowable interaction-kinds.
(Lower left) The sibling index and source cluster index of each box and cluster.
(Lower right) Examples of the unallowable interaction-kinds when the source cluster index is 5 and 2. (When the source cluster index is 5, interaction-kinds 1:1-2, 3:3-0, and 6:1-0,3-2 are not allowed. When the source cluster index is 2, only the interaction-kind 1:1-2 is not allowed.)

For example, Figure 5 shows three pairs of clusters, where T_0–T_7 and S_0–S_7 denote the target and source siblings in a target cluster (TC in the code) and source cluster (SC in the code), respectively. The source-cluster indices are 23, 26, and 14 for the cases left, center, and right, respectively (see Figure 4). In each case, observe that the eight interactions of interaction-kind 26 (between T_0 and S_0, …, and T_7 and S_7) share the same transfer vector. Also, this interaction-kind is allowable for any source-cluster index (i.e., the interaction-kind of 26 is not found in Table V (right)). The eight interactions can therefore be represented as the single matrix–matrix product

$$[\mathbf{L}(T_0),\ \mathbf{L}(T_1),\ldots,\ \mathbf{L}(T_7)] = \mathbf{D}^{(T_0,S_0)}\,[\mathbf{M}(S_0),\ \mathbf{M}(S_1),\ldots,\ \mathbf{M}(S_7)]\,,$$

where the dimension of the two matrices in the square brackets is $r \times 8$.

6.2. *Code and statistics*

To implement this scheme in CUDA, we assign one thread-block to one target cluster, and use r threads per thread-block. The ith thread computes the ith element of each of the eight L-vectors for the target cluster. Namely, using pseudocode notations:

$$\{L_i(T_a),\ L_i(T_b),\ L_i(T_c),\ldots\} += \sum_{j=0}^{r-1} D_{ij}^{(T_a,S_d)}\ \{M_j(S_d),\ M_j(S_e),\ M_j(S_f),\ldots\} \qquad (4)$$

for all source clusters SC (at most 26) and all allowable interaction-kinds (at most 27), where the pairs of boxes (*i.e.*, T_a and S_d, T_b and S_e, T_c and S_f, …) are determined by the source-cluster index and interaction-kind.

DOI: 10.1002/nme

Table V. List of 27 3D interaction-kinds in a pair of target cluster and unallowable interaction-kinds.

Interaction-kind	Interactions (A pair 'i-j' denotes the interaction between T_i and S_j)	Source–cluster index	Unallowable interaction-kinds
0	0-7	0	0
1	1-6	1	0, 1, 8
2	2-5	2	1
3	3-4	3	0, 2, 9
4	4-3	4	0, 1, 2, 3, 8, 9, 11, 13, 20
5	5-2	5	1, 3, 11
6	6-1	6	2
7	7-0	7	2, 3, 13
8	0-6, 1-7	8	3
9	0-5, 2-7	9	0, 4, 10
10	0-3, 4-7	10	0, 1, 4, 5, 8, 10, 12, 16, 21
11	1-4, 3-6	11	1, 5, 12
12	1-2, 5-6	12	0, 2, 4, 6, 9, 10, 14, 17, 22
13	2-4, 3-5	13	0 – 26 (all)
14	2-1, 6-5	14	1, 3, 5, 7, 11, 12, 15, 18, 23
15	3-0, 7-4	15	2, 6, 14
16	4-2, 5-3	16	2, 3, 6, 7, 13, 14, 15, 19, 24
17	4-1, 6-3	17	3, 7, 15
18	5-0, 7-2	18	4
19	6-0, 7-1	19	4, 5, 16
20	0-4, 1-5, 2-6, 3-7	20	5
21	0-2, 1-3, 4-6, 5-7	21	4, 6, 17
22	0-1, 2-3, 4-5, 6-7	22	4, 5, 6, 7, 16, 17, 18, 19, 25
23	1-0, 3-2, 5-4, 7-6	23	5, 7, 18
24	2-0, 3-1, 6-4, 7-5	24	6
25	4-0, 5-1, 6-2, 7-3	25	6, 7, 19
26	0-0, 1-1, 2-2, 3-3, 4-4, 5-5, 6-6, 7-7	26	7

(Left) List of 27 3D interaction-kinds in a pair of target cluster (consisting of T_0 to T_7) and source cluster (consisting of S_0 to S_7).
(Right) List of unallowable interaction-kinds. Note that any source clusters with index 13 are ignored because they coincide with their field clusters.

Listing 4 shows the loops ordering for this scheme. At level k, each kernel uses 8^{k-1} thread-blocks, each of which is assigned to a target cluster. See Listing 8 in Appendix B for more detailed GPU pseudocode. Additional optimizations, that are too long to describe in this manuscript were implemented to account for some of the memory limitations and special characteristics of the device. The actual code can be found online in [34].

Table VI shows the statistics of our implementation of the second scheme. The traffic of D-data is successfully reduced to 46% of the first scheme. Also, the traffic of M-data decreases from $189r$ to $26r$. The flop-to-word ratio is about twice as large than the first scheme, but still significantly less than the GPU-specific ratio given in Table II.

7. CLUSTER-BLOCKING SCHEME: SHARE D-DATA IN CLUSTERS

7.1. Design

The D-matrix traffic is further reduced by sharing it within groups of clusters. We call a group of $B \times B \times B$ clusters a *chunk* of size B. Figure 6 shows chunks of size 2.

In this scheme, the order of cluster interaction is chosen such that all B^3 clusters share the same D-data. A difficulty at this point is that the blocking is in effect so large that we cannot store the entire D-matrix in memory. As a result, we split the D-matrix into tiles and denote such tiles by $\mathbf{D}^{pq}$ ($0 \leqslant p < P$ and $0 \leqslant q < Q$), where the dimension of every tile is $\frac{r}{P} \times \frac{r}{Q}$ (see Figure 7).

DOI: 10.1002/nme

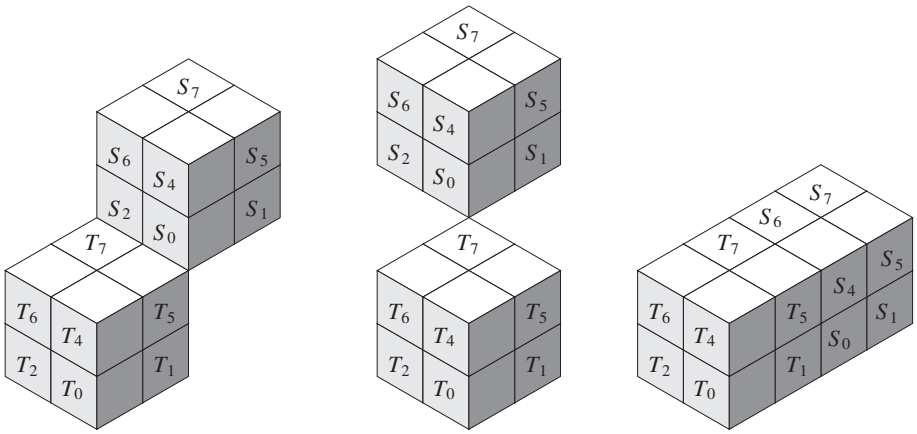


Figure 5. Example of three pairs of target and source clusters. The source-cluster indices are 23, 26, and 14 for the cases left, center, and right, respectively.

```
1  for all target clusters TC in level k          // Parallelized over 8^{k-1} thread blocks
2    for all i, 0 ⩽ i < r                         // Parallelized over r threads
3      for all 26 source clusters SC of TC
4        Load in shared memory M_i(SC).
5        for all allowable interaction-kinds I
6          for all j, 0 ⩽ j < r
7            L_i(TC) += D_ij^(I) M_j(SC)
```

Listing 4. Rough pseudocode for the second scheme single-level multipole-to-local operation.

Table VI. Statistics of the second scheme (sibling-blocking scheme).

		Per target cluster	Per target box
Read M-data	[word]	$8 \cdot 26 \cdot r$	$26r$
Read D-data	[word]	$26 \cdot 27 \cdot r^2$	$\frac{26 \cdot 27}{8} r^2$
Read/Write L-data	[word]	$8 \cdot 26 \cdot r$	$26r$
Operation counts	[flop]	$8 \cdot 189 \cdot r(2r-1)$	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	4.2 for $r = 32$ 4.3 for $r = 256$	

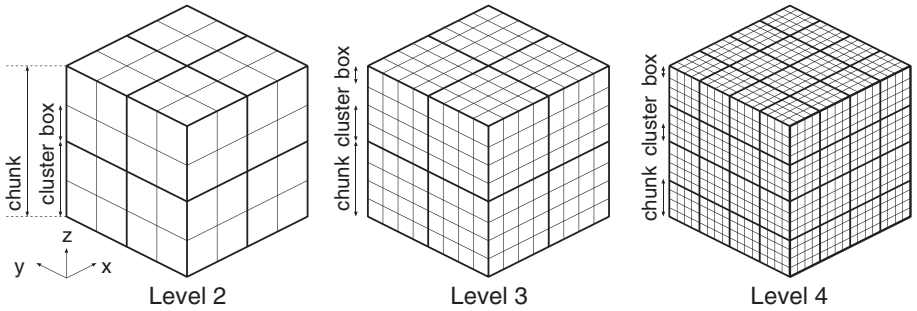


Figure 6. Chunks of size 2 for levels 2, 3 and 4.

Correspondingly, the vectors $\mathbf{L}$ and $\mathbf{M}$ are split into P and Q parts, respectively. They are denoted by $\mathbf{L}^p$ and $\mathbf{M}^q$. One limitation is that the M-vectors now need to be read more often compared with the second scheme, because we need to read it once for each row tile.

DOI: 10.1002/nme

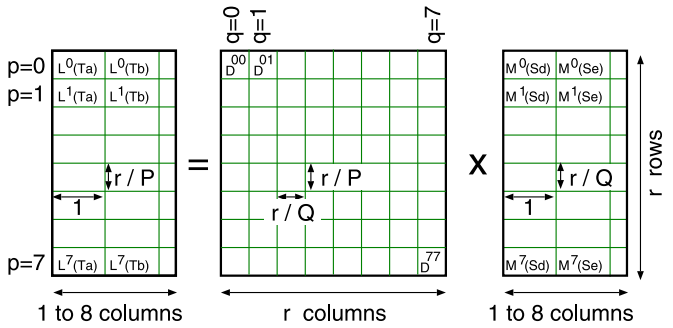


Figure 7. Splitting L-vector(s), M-vector(s), and D-matrix into tiles: this shows the case of $P = Q = 8$. The sets of $\{T_a, T_b, T_c, \ldots\}$ and $\{S_d, S_e, S_f, \ldots\}$ depend on the underlying source-cluster index and interaction-kind (see Table V).

We use two-dimensional thread-blocks of size $\frac{r}{P} \times B^3$, and assign one thread-block to one chunk. In each thread-block, the (i, TC)-th thread (where $0 \leqslant i < \frac{r}{P}$ and $0 \leqslant TC < B^3$) manages the ith row of all the sub-vectors $\mathbf{L}^p$ for all the eight boxes $T_0, \ldots, T_7$ in the TC-th cluster.

7.2. *Code and statistics*

Listing 5 shows the loop ordering for this scheme. At level k, each kernel uses $8^{k-1}/B^3$ thread-blocks, each of which is assigned to a chunk of boxes. See Listing 9 in Appendix B for more detailed GPU pseudocode. The actual code can be found online in [34].

```
1  for all chunks G in level k                    // Parallelized over 8^{k-1}/B^3 thread blocks
2    for all target clusters TC of G              // Parallelized over B^3 threads
3      for all i, 0 ⩽ i < r/P                     // Parallelized over r/P threads
4        for all row tiles p, 0 ⩽ p < P
5          for all column tiles q, 0 ⩽ q < Q
6            for all 26 source clusters SC of TC
7              Load in shared memory M_i^q(SC).
8              for all allowable interaction-kinds I
9                Load in shared memory D_ij^{pq(I)} for all j.
10               for all j, 0 ⩽ j < r/Q
11                 L_i(TC) += D_ij^{pq(I)} M_j^q(SC)
```

Listing 5. Rough pseudocode for the third scheme single-level multipole-to-local operation.

Table VII shows the statistics of this scheme. The key improvement is that the traffic of D-data now scales like $1/B^3$ whereas the traffic for the M-data scales like P. We present the results for $B = 2$ because of shared memory constraints for storing the M-data. In the case of $r = 32$, we use $P = Q = 1$; that is, it is unnecessary to split the D-matrix into tiles. In the case of $r = 256$, we choose $P = Q = 8$. In both cases, the dimension of the thread-blocks is 32×8.

With the previously mentioned parameters, we can obtain flop-to-word ratios about 8 $(= B^3)$ times higher than the second scheme. These flop-to-word ratios are comparable to the GPU-specific ratio of 24 for the Tesla C1060 in Table II. Note that the performance of the third scheme may be compute limited rather than bandwidth limited if the scheme is run on the 8800GTX or GTX285.

8. IJ-BLOCKING SCHEME: YET ANOTHER SHARING OF D-DATA

This scheme attempts to obtain a higher flop-to-word ratio by considering a large-scale blocking of boxes but in a manner different from the third scheme. Essentially we chose as inner loop the interaction list loop over the (T, S) pairs, the outer loop becoming the loop over i and j.

DOI: 10.1002/nme

Table VII. Statistics of the third scheme (cluster-blocking scheme).

		Per chunk of size B	Per target box
Read M-data	[word]	$8B^3 \cdot P \cdot Q \cdot 26 \cdot \frac{r}{Q}$	$26Pr$
Read D-data	[word]	$P \cdot Q \cdot 26 \cdot 27 \cdot \frac{r}{P} \cdot \frac{r}{Q}$	$\frac{26 \cdot 27}{8B^3} r^2$
Write L-data	[word]	$8B^3 \cdot P \cdot \frac{r}{P}$	r
Operation counts	[flop]	$8B^3 \cdot 189 \cdot r(2r-1)$	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	21 for $r = 32$, $B = 2$, $P = Q = 1$ 32 for $r = 256$, $B = 2$, $P = Q = 8$	

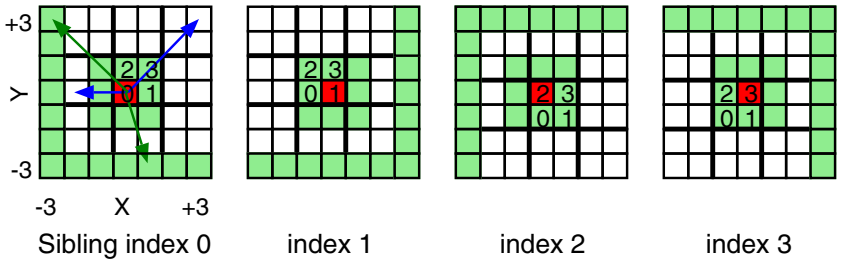


Figure 8. Example of allowable multipole-to-local-transfer vectors (white boxes, blue arrows) and unallowable transfer vectors (colored boxes, green arrows) in 2D (for illustration purposes). These sets depend on the sibling index of the box.

8.1. Design

A thread-block is assigned to one chunk of size B. Each thread is assigned to a box in a chunk. Thus, we use $8B^3$ threads per thread-block. We compute all $8B^3$ L-vectors for every chunk as follows. First, for a given column index j ($0 \leqslant j < r$), we let all the $8B^3$ threads share M_j for the $(2B+4) \times (2B+4) \times (2B+4)$ boxes that enclose the relevant chunk. To deal with the boundary of the domain, we appropriately place *ghost boxes* outside of the domain, and let $M_j = 0$ in each ghost box. Next, for a given row index i, we let all the threads share $D_{ij}^{(I)}$ for all 316 indices I. Finally, using the shared M-data and D-data, each thread computes

$$L_{ij}(T) := \sum_{S \in \mathcal{I}(T)} D_{ij}^{(T,S)} M_j(S), \tag{5}$$

where T denotes the target box that the relevant thread is assigned to. Last, the result is accumulated into $L_i(T)$. This process is repeated for all i and j.

This scheme can share D-data between $8B^3$ threads, whereas the previous scheme shares between B^3 threads. Because the storage requirements for the fourth scheme are different from the third scheme, we are able to use $B = 4$ in the fourth scheme, but only $B = 2$ in the third scheme.

However, the present scheme has a special difficulty originating from the asymmetric treatment of the eight siblings. Namely, the total number of M2L-transfer vectors is 316 ($= 7^3 - 3^3$) but the target boxes only request a subset of 189 ($= 6^3 - 3^3$) vectors, which depends on their sibling-index. This is illustrated by Figure 8 in the two-dimensional case. This can perturb the access pattern for the shared M-data in the computation of (5) and degenerate the performance because of the resulting warp-serializations and/or bank conflicts for shared-memory.

This issue is resolved optimally for GPUs with compute capability 1.x, for which the bank conflict can be avoided. For GPUs with compute capability 2.x, because of the way bank conflicts can happen, the optimal ordering of operations results in a two-way bank conflict (that is, for each bank, two threads are accessing the bank). This cannot be avoided. First, we store $(2B+4)^3$ M-data so that data for the same sibling-index is stored together into a three-dimensional shared-memory array (Figure 9). Next, we assign a group of B^3 threads to B^3 target boxes with the same sibling-index.

DOI: 10.1002/nme

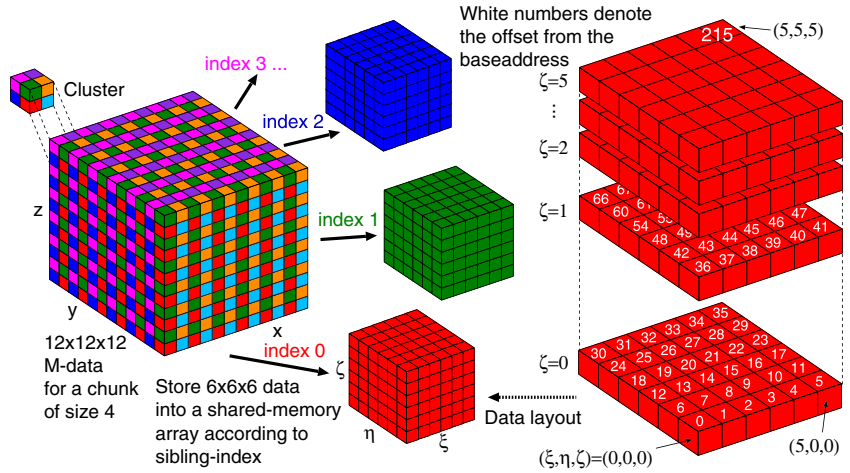


Figure 9. Layout of M-data in three-dimensional shared-memory arrays, in which we assume that $B = 4$.

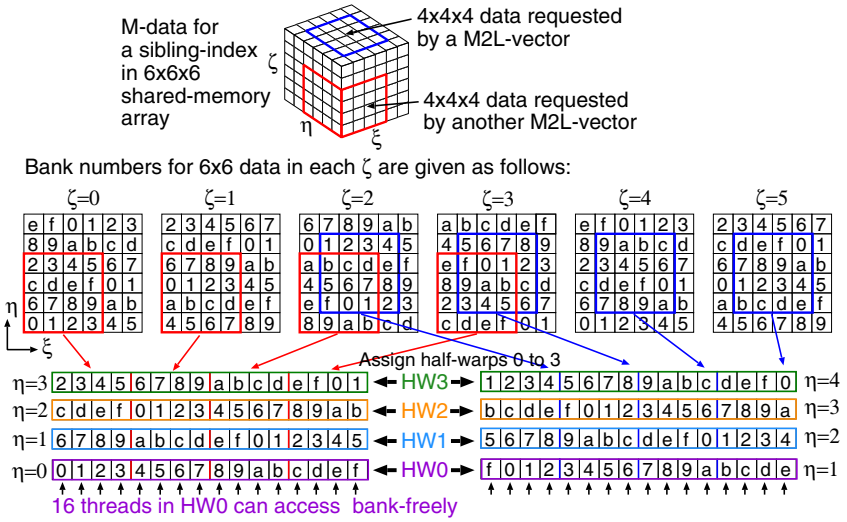


Figure 10. Suppose $B = 4$. (Top): We illustrate M-data stored in a $6 \times 6 \times 6$ shared-memory array for a certain sibling-index (Figure 9) and the $4 \times 4 \times 4$ data (either red or blue cube) requested by $4 \times 4 \times 4$ target boxes with the same sibling-index. (Middle): We illustrate the shared-memory bank IDs (0 to f; there are 16 shared-memory banks per SM for the GPUs with compute capability 1.x) given to the $6 \times 6 \times 6$ shared-memory array. (Bottom): We notice that, if one half-warp (HW) is assigned to a slice of 4×4 data for a fixed η, the half-warp can read the 16 data without bank-conflicts because their bank IDs are different from one another.

Then, for a given M2L-vector, all the B^3 threads can read the requested B^3 data from the same shared-memory array, which is necessary to prevent the warps for the B^3 threads from diverging (we assume that B^3 is a multiple of the warp-size). To avoid bank-conflicts for the shared-memory arrays, we have to make each half-warp in every B^3 threads read their 16 words from different banks. This is readily possible when $B = 4$ as illustrated in Figure 10. For greater Bs, if we assume $B = 2^k$ with $k \geq 2$, we can also avoid such bank-conflicts in reading the requested B^3 M-data because we can read every 4^3 words in the same way as the case of $B = 4$.

Note that the present scheme cannot work for coarse levels because chunks of size B cannot be defined for such levels. In addition, when one thread-block is assigned to one chunk, the number of thread-blocks issued is small. To resolve these problems, another scheme is applied for levels ranging from 2 to a prescribed level $k_{\text{switch}} (\geq 2)$, whereas the present scheme is applied to levels

DOI: 10.1002/nme

$k_{\text{switch}}+1$ to κ. Moreover, r rows of L-vector are split into P tiles, and thus, one chunk is processed by P thread-blocks. Then, the (G, p)th thread-block is assigned to the Gth chunk for pth row tile, where $0 \leqslant G < 8^k/(8B^3)$ for level k and $0 \leqslant p < P$. In the tests presented in Section 9, we applied the sibling-blocking (second) scheme to level 2 (that is, we choose $k_{\text{switch}} = 2$) and $P = 8$ for the low precision case (i.e., $r = 32$) and $P = 16$ for the high precision case (i.e., $r = 256$).

8.2. Code and statistics

Listing 6 shows the loop ordering for this scheme. At level k, each kernel uses $8^{k-1}/B^3$ thread-blocks, each of which is assigned to a chunk of boxes. See Listing 10 in Appendix B for more detailed GPU pseudocode. The actual code can be found online in [34].

```
1  for all chunks G in level k                  // Parallelized over 8^{k-1}/B^3 thread blocks
2    for all row tiles p, 0 ⩽ p < P             // Parallelized over P thread blocks
3      for all target boxes T of G              // Parallelized over 8B^3 threads
4        for all j, 0 ⩽ j < r
5          Load in shared memory M_j(G).
6          for all i, pr/P ⩽ i < (p+1)r/P
7            Load in shared memory D_ij^(⋆).
8            for all 189 source boxes S ∈ I(T)
9              L_i(T) += D_ij^(T,S) M_j(S)
```

Listing 6. Rough pseudocode for the fourth scheme single-level multipole-to-local operation.

Table VIII shows the statistics. We can obtain a very large flop-to-word ratio over 100 with $B = 4$. Here, $P = 8$ and 16 are assumed (and they were actually used in our numerical benchmarks in Section 9), but P is less sensitive to the flop-to-word ratio than B.

9. NUMERICAL RESULTS

In the following sections, we investigate the performance and accuracy of the GPU-accelerated bbFMM program in comparison with an optimized multi-core bbFMM program run on the CPU.

9.1. Setup

9.1.1. N-body system. We consider systems of particles uniformly distributed in a cubic domain with edge length of one ($\ell = 1$) with $N = 10^3, \ldots, 10^7$ particles and octree depths of $\kappa = 2, 3, 4, 5$, and 6, respectively, according to $\kappa := \log N/\log 8 - 1$ as in [4]. The source strengths $\{\sigma_j\}$ are either $+1$ or -1, with net zero strength in total.

The test kernel is the Laplace kernel $K(\mathbf{x}, \mathbf{y}) = 1/|\mathbf{x} - \mathbf{y}|$. Note that the choice of kernel in the bbFMM does not affect the performance of the M2L operation.

Table VIII. Statistics of the fourth scheme (ij-blocking scheme).

		Per chunk of size B	Per target box
Read M-data	[word]	$(2B+4)^3 \cdot r \cdot P$	$\frac{(B+2)^3 P}{B^3} r$
Read D-data	[word]	$316 \cdot r \cdot \frac{r}{P} \cdot P$	$\frac{316}{8B^3} r^2$
Read/write L-data	[word]	$2 \cdot 8B^3 \cdot r \cdot \frac{r}{P} \cdot P$	$2r^2$
Operation counts	[flop]	$8B^3 \cdot 189 \cdot \frac{r}{P}(2r-1) \cdot P$	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	108 for $r = 32$, $B = 4$, $P = 8$ 133 for $r = 256$, $B = 4$, $P = 16$	

DOI: 10.1002/nme

9.1.2. Computer. In the first series of tests, we used a DELL Poweredge 1950 with two quad-core Intel Xeon E5345 CPUs running at 2.33 GHz (4 MB shared L2 cache per two cores) with 16 GB of 667 MHz DDR2 SDRAM (5.333 GB/s) [35] on Intel 5000X chipset [36]. Each core has the SSE3 (streaming SIMD instructions 3) unit. All eight cores can share 16 GB of memory. The peak performance in single-precision floating-point is 18.64 Gflop/s per core and 149.12 Gflop/s in total.

For the GPU, we used one of four NVIDIA Tesla C1060 graphics cards [28] in a Tesla S1070 connected to this machine via PCIe 8x. A C1060 has 30 SMs (1.30 GHz of clock rate; 16 KB of shared memory per SM; 16384 32-bit registers per SM) and 4 GB of device memory. When three single-precision floating-point operations are performed per clock cycle, the peak speed is 933 Gflop/s. When a multiply–add (add or multiply) operation is processed per clock cycle, it is 624 and 312 Gflop/s, respectively.

We used Intel's C/C++ compiler 10.1 (with optimizing options `-O3` and `-xT`) for the C codes, and NVIDIA's CUDA SDK 2.2 for the CUDA codes.

9.1.3. Codes. Following Section 2, we developed a single-precision bbFMM program optimized for the earlier mentioned shared memory machine. This code is referred to as the CPU code hereafter.

The M2L operation of the CPU code is implemented with a scheme similar to the sibling-blocking scheme (Section 6). Accordingly, the loop over boxes (line 15 in Listing 1) was replaced with a loop over clusters. We then parallelized the loop using the eight cores by prepending the OpenMP directive '`#pragma omp parallel for`.' In addition, we vectorized the loop over rows (the summation over j in Equation (4)) with the SSE3 units.

From the CPU code, we developed a GPU-accelerated bbFMM program by altering the M2L computation according to Sections 5 to 8. This program can choose one of the four M2L schemes, which we refer to as the GPU1, 2, 3, and 4 codes appropriately. Note that the GPU4 code uses the second scheme for the lower levels, whereas the fourth scheme is used for higher levels (Section 8.1).

For comparison, we also implemented the direct code on CPU, which is an $\mathcal{O}(N^2)$ implementation of the sum in Equation (1) using single-precision floating-point arithmetic.

9.2. Accuracy results

We checked the accuracy of the four GPU codes in comparison with the CPU code and the direct code by measuring the relative L^2-error, ε, given by

$$\varepsilon := \left[\frac{\sum_{i=1}^{N} (f^{\text{target}}(\mathbf{x}_i) - f^{\text{reference}}(\mathbf{x}_i))^2}{\sum_{i=1}^{N} (f^{\text{reference}}(\mathbf{x}_i))^2} \right]^{\frac{1}{2}},$$

where 'target' and 'reference' stand for a target code and its reference, respectively.

Table IX shows the errors of the CPU and four GPU codes versus the direct code run in single precision. The direct code took too much time to execute the $N = 10^7$ case. The errors of the GPU codes are of the same order as those of the CPU code in both the low ($r = 32$) and high ($r = 256$) precision cases. Indeed, Table X shows that the error of the GPU codes versus the CPU code is at the level of single-precision round-off error. Furthermore, we note that the dependency of error on n in Table IX agrees with Figure 6(a) in [4].

In the next section, we discuss the M2L kernels in more detail as they are the primary focus of this study.

9.3. Multipole-to-local results

Table XI shows the timing and performance of the M2L kernels. Here, we measured the time with the Read Time Stamp Counter (RDTSC) for C codes and the event management functions for CUDA codes [14]. Each time is the average of 10 executions. The performance is computed as '(the number of the M2L operations in the tree) × (floating-point operations per M2L operation, that is, $r(2r-1)$)/(elapsed time of the M2L stage)'. The M2L operations were indeed accelerated by the GPU. The performances of GPU3 and GPU4 sustained a hundred Gflop/s or more for large N.

DOI: 10.1002/nme

Table IX. Error versus direct code (top: $r = 32$, bottom: $r = 256$): The direct code is implemented in single-precision floating-point arithmetic.

N	CPU	GPU1	GPU2	GPU3	GPU4
10^3	3.25091e-04	3.25130e-04	3.25108e-04	3.25119e-04	3.25108e-04
10^4	7.33478e-04	7.33501e-04	7.33459e-04	7.33455e-04	7.33462e-04
10^5	6.66763e-04	6.66775e-04	6.66768e-04	6.66741e-04	6.66764e-04
10^6	7.83211e-04	7.83177e-04	7.83209e-04	7.83210e-04	7.83209e-04

N	CPU	GPU1	GPU2	GPU3	GPU4
10^3	1.46095e-06	1.78251e-06	1.49480e-06	1.69537e-06	1.49480e-06
10^4	3.27401e-06	3.64197e-06	3.19091e-06	3.69513e-06	3.19252e-06
10^5	7.11934e-06	7.45730e-06	7.13091e-06	7.57058e-06	7.12101e-06
10^6	1.84872e-05	1.86662e-05	1.84916e-05	1.87624e-05	1.84984e-05

Table X. Error of GPU codes versus CPU code (top: $r = 32$, bottom: $r = 256$).

N	GPU1	GPU2	GPU3	GPU4
10^3	6.86350e-07	2.89722e-07	5.02913e-07	2.89722e-07
10^4	1.10693e-06	4.46567e-07	9.60799e-07	4.58189e-07
10^5	1.44904e-06	4.22898e-07	1.29688e-06	4.60100e-07
10^6	1.81868e-06	5.08796e-07	1.74660e-06	5.68528e-07
10^7	1.52659e-06	4.47121e-07	1.51480e-06	4.89664e-07

N	GPU1	GPU2	GPU3	GPU4
10^3	1.11132e-06	3.63981e-07	1.03731e-06	3.63981e-07
10^4	1.95096e-06	6.60109e-07	2.03845e-06	6.48679e-07
10^5	2.53376e-06	6.29465e-07	2.77803e-06	6.75485e-07
10^6	2.96203e-06	7.61351e-07	3.52526e-06	8.57488e-07
10^7	2.63074e-06	6.26272e-07	2.95823e-06	7.44576e-07

For small N, the performance of GPU3 is worse than the other methods. When κ is small, GPU3 does not issue enough thread-blocks to make full use of all SMs of the GPU.

The performance of GPU1 and GPU2 saturate as N increases. The reason is that the performances were bounded by data transfer between SMs and the device memory of GPU, as we can expect from the comparison of the scheme's flop-to-word ratios with the GPU-specific ratio. To see this more quantitatively, we determine an upper bound on the performance because of the GPU's peak bandwidth. This corresponds to the lower bound on the method's runtime if it were to only read and write the required data at the peak bandwidth of the GPU. This performance (called the *bandwidth peak performance* [37]) of a scheme is given by

$$\begin{aligned}\text{Bandwidth peak performance [flop/s]} &= \frac{\text{Total arithmetic count [flop]}}{\frac{\text{Total data traffic [B]}}{\text{Peak bandwidth [B/s]}}} \\ &= \frac{\text{Flop-to-word ratio [flop/word]}}{\frac{4\ \text{[B/word]}}{\text{Peak bandwidth [B/s]}}},\end{aligned} \tag{6}$$

where the peak bandwidth is 102 GB/s as in Table II, and the scheme's flop-to-word ratios are given in Tables III, VI, VII, and VIII. The bandwidth peak performances are shown in Table XII. These results are plotted in Figure 11 together with the measured performance shown in Table XI.

DOI: 10.1002/nme

Table XI. Performance of multipole-to-local kernels using a Tesla C1060 (top: $r = 32$, bottom: $r = 256$), measured in Gflop/s.

Code / κ	2	3	4	5	6
CPU	11.4	32.0	37.0	37.6	37.6
	(5.51e–4)	(3.56e–3)	(3.49e–2)	(3.24e–1)	(2.81e+0)
GPU1	16.2	32.4	39.4	40.9	41.1
	(3.84e–4)	(3.51e–3)	(3.28e–2)	(2.98e–1)	(2.57e+0)
GPU2	6.4	24.6	57.7	69.6	71.4
	(9.75e–4)	(4.62e–3)	(2.24e–2)	(1.75e–1)	(1.48e+0)
GPU3	1.5	13.8	62.7	128.9	156.8
	(4.15e–3)	(8.26e–3)	(2.06e–2)	(9.45e–2)	(6.73e–1)
GPU4	6.5	33.8	120.1	210.3	220.0
	(9.65e–4)	(3.37e–3)	(1.08e–2)	(5.80e–2)	(4.80e–1)

Code / κ	2	3	4	5	6
CPU	8.6	8.8	8.6	8.6	8.6
	(4.70e–2)	(8.35e–1)	(9.72e+0)	(9.18e+1)	(7.98e+2)
GPU1	41.9	41.9	42.1	41.8	41.7
	(9.67e–3)	(1.76e–1)	(1.99e+0)	(1.89e+1)	(1.64e+2)
GPU2	51.7	91.2	100.7	101.7	101.3
	(7.83e–3)	(8.10e–2)	(8.32e–1)	(7.77e+0)	(6.76e+1)
GPU3	1.6	14.4	65.2	133.1	161.6
	(2.56e–1)	(5.12e–1)	(1.29e+0)	(5.93e+0)	(4.24e+1)
GPU4	51.8	88.7	177.9	241.9	269.5
	(7.81e–3)	(8.32e–2)	(4.71e–1)	(3.27e+0)	(2.54e+1)

The numbers in parenthesis are the timing in second. Note that GPU4 is equivalent to GPU2 for $\kappa = 2$.

Table XII. Bandwidth peak performance calculated from Equation (6): Unit is Gflop/s.

r	GPU1	GPU2	GPU3	GPU4
32	49	93	542	2742
256	51	107	816	3167

The bandwidth peak performances approximately agree with their upper bounds of measured performances. Therefore, the performances of GPU1 and GPU2 are indeed bounded by data transfer.

On the other hand, GPU3 and GPU4 seem to overcome the bottleneck of data transfers in view of their flop-to-word ratios. Nevertheless, the measured performances in Table XI were less than the arithmetic peak performance of 624 Gflop/s for MAD operation, although they were close to 312 Gflop/s for add or multiply operation per clock cycle. This inefficiency might be a consequence of insufficient number of active thread-blocks per SM. Indeed, the item (v) of Table XIII shows that the number of active thread-blocks per SM in GPU3 and GPU4 is only one, which makes it impossible for an SM to hide the memory latency by alternating thread-blocks.

In Table XIII, the items (v) and (vi) show the utilizations of registers and shared memory in the present implementations, respectively. If we utilize much more resources on GPU, for example, registers to prefetch certain data [30], we may be able to obtain faster codes.

Finally, we discuss the performance of the CPU code, whose scheme is similar to GPU2. As shown in Table XI, the sustained performance of the CPU code clearly saturates as N increases, similar to GPU1 and GPU2. The bandwidth peak performance calculated from Equation (6) is 22.4 and 22.9 Gflop/s for $r = 32$ and 256, respectively, where we used the GPU2's flop-to-word ratio and the read bandwidth of 21.3 GB/s [36]. The result for $r = 256$ is roughly close to the upper bounds of the bandwidth peak performance. In the case $r = 32$, it is likely that computations were mostly performed with data in cache, rather than main memory.

DOI: 10.1002/nme

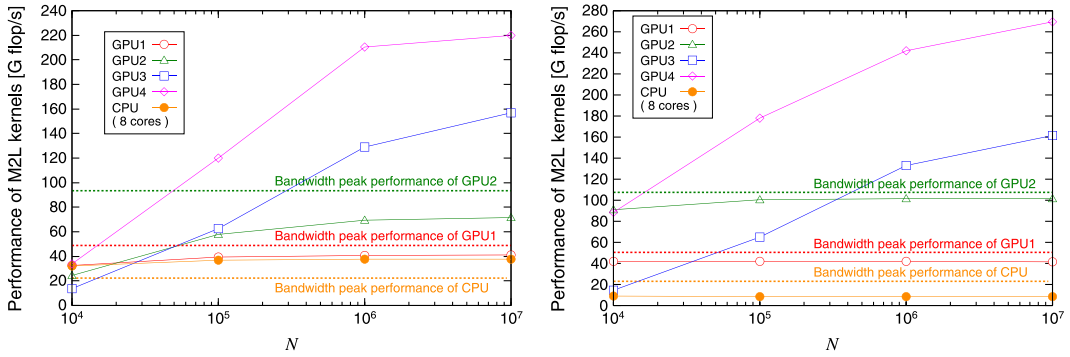


Figure 11. Performance of multipole-to-local kernels and bandwidth peak performance (left: $r = 32$, right: $r = 256$).

Table XIII. Profile of the GPU kernels ($r = 256$; Tesla C1060 GPU).

Item		GPU1	GPU2	GPU3$^{\diamondsuit}$	GPU4$^{\natural}$	Maximum
(i)	# of threads per TB	256	256	256	512	512
(ii)	# of REGs per thread	11	16	30	26	16384/(i)
(iii)	SMEM per TB [KB]†	2.5	4.1	14.7	11.7	16
(vi)	# of active TBs per SM§	4	3	1	1	8
(v)	Utilization of REGs [%]$^{\flat}$	69	75	47	81	100
(vi)	Utilization of SMEM [%]$^{\sharp}$	62	77	92	73	100

TB, thread-block; REG, register; SMEM, shared-memory.
†We counted this number from the kernels directly.
§This number is given by the minimum of (maximum # of REGs per SM; 16384)/(ii) and (maximum amount of SMEM per SM; 16 KB)/(iii), whereas neither the number of active warps nor threads or TBs per SM can exceed their maximums.
$^{\flat}$This is given by (i) × (ii) × (vi)/(maximum # of REGs per SM).
$^{\sharp}$This is given by (iii) × (ii) × (vi)/(maximum amount of SMEM per SM).
$^{\diamondsuit}$ $B = 2$ is used.
$^{\natural}$ $B = 4$ and $P = 16$ are used and numbers are those for $k > k_{\text{switch}}$.

10. BENCHMARKS USING FERMI PROCESSORS

The GPU-accelerated programs presented in Section 9 correspond to GPUs of compute capability 1.x. Our methods also apply to newer architectures, such as Fermi [38], or GPUs of compute capability 2.x, some of which are designed to offer high performance in double precision.

We made a few modifications to the GPU4 code, which obtained the best performance on Tesla C1060 as seen in Table XI, so that the hardware resources on a Fermi processor can be better utilized. To this end, we re-examined the loop unrolling for the columns j (line 9, Listing 10, Appendix B), and tile rows i (line 13), letting $P = 4$ for any r. The best value for unrolling was determined through numerical tests. As a result, the utilizations of registers and shared memory are kept at the same level as those in Table XIII for both single precision and double precision. On the other hand, the shared-memory bank conflict cannot be avoided, as mentioned in Section 8.1. This is because Fermi processors have 32 banks for their shared memory, compared with 16 banks for GPUs of capability 1.x. For the latter, bank conflicts between two half warps cannot occur [14]. As a result, a two-way bank conflict occurs on Fermi processors (the single and double precision cases present the same issues).

We used a DELL Precision T7500 with two six-core Intel Xeon X5680 CPUs (3.33 GHz) on Intel X5520 chipset with 1333 MHz DDR3 RDIMM (32 GB/s of bandwidth per CPU) [13]. This is not the same CPU that was used in the previous tests. In particular this CPU was found to outperform the CPU on the DELL Poweredge 1950. The peak performance of the Intel Xeon X5680 for

DOI: 10.1002/nme

12 cores is 319.68 and 159.84 Gflop/s for single precision and double precision, respectively. The workstation is equipped with an NVIDIA Tesla C2050 graphics card [29], which is based on the Fermi architecture. The peak performance for MAD operation is 1.03 Tflop/s and 515 Gflop/s for single precision and double precision, respectively. The memory bandwidth (ECC on) is 115 GB/s. Thus, the flop-to-word ratio is 36 for both single and double precision. This number is less than the flop-to-word ratio of GPU4 in Table VIII. In the benchmark here, we used the Intel C/C++ 12.0 compiler (with optimizing options `-O3` and `-xSSE4.2`) for the C codes, and NVIDIA's CUDA SDK 3.2 for the CUDA codes.

Table XIV shows the errors of the CPU and four GPU codes versus the direct code in the case of double precision (the result for single precision is omitted because it is similar to the result in Table IX), where the direct code was run at the same precision as the other codes. In all codes, the accuracy for $r = 32$ was not improved by using double precision instead of single precision because the error due to truncating r to 32 is greater than the single-precision round-off error. Meanwhile, an improvement was observed for $r = 256$ (the error was reduced from $\sim 10^{-6}$ to $\sim 10^{-7}$).

Table XV compares the error of the GPU codes with the CPU code in the double-precision case. The errors are comparable to the machine round-off error.

Tables XVI and XVII show the performance of the M2L kernels. The CPU used in this test outperforms the CPU used in the previous results. For the CPU code, with $r = 256$ in single precision, a 4.3 ($= 41.3/8.6$) times speedup is observed relative to the CPU performance in Table XI; it results

Table XIV. Error versus direct code for double precision using Tesla C2050 (top: $r = 32$, bottom: $r = 256$): The direct code is implemented in double precision. There is no difference between any of the codes for each N in the present numeric format.

N	CPU	GPU1	GPU2	GPU3	GPU4
10^3	3.22720e-04	3.22720e-04	3.22720e-04	3.22720e-04	3.22720e-04
10^4	7.32120e-04	7.32120e-04	7.32120e-04	7.32120e-04	7.32120e-04
10^5	6.66505e-04	6.66505e-04	6.66505e-04	6.66505e-04	6.66505e-04
10^6	7.80298e-04	7.80298e-04	7.80298e-04	7.80298e-04	7.80298e-04

N	CPU	GPU1	GPU2	GPU3	GPU4
10^3	1.30735e-07	1.30735e-07	1.30735e-07	1.30735e-07	1.30735e-07
10^4	2.80162e-07	2.80162e-07	2.80162e-07	2.80162e-07	2.80162e-07
10^5	2.52779e-07	2.52779e-07	2.52779e-07	2.52779e-07	2.52779e-07
10^6	2.99511e-07	2.99511e-07	2.99511e-07	2.99511e-07	2.99511e-07

Table XV. Error of GPU codes versus CPU code for double precision using Tesla C2050 (top: $r = 32$, bottom: $r = 256$).

N	GPU1	GPU2	GPU3	GPU4
10^3	1.30775e–15	5.67221e–16	1.02956e–15	5.67221e–16
10^4	2.33102e–15	8.45871e–16	1.81839e–15	9.06713e–16
10^5	2.78896e–15	8.15784e–16	2.46338e–15	9.11237e–16
10^6	3.24408e–15	9.69221e–16	3.24737e–15	1.09494e–15
10^7	2.76807e–15	8.43073e–16	2.68563e–15	9.19547e–16

N	GPU1	GPU2	GPU3	GPU4
10^3	2.71648e–15	6.80833e–16	2.40215e–15	6.80833e–16
10^4	6.29231e–15	1.26256e–15	6.78295e–15	1.34415e–15
10^5	7.62251e–15	1.13736e–15	8.70325e–15	1.41340e–15
10^6	8.65915e–15	1.53186e–15	1.17202e–14	1.78414e–15
10^7	8.15838e–15	1.23552e–15	9.69601e–15	1.49434e–15

DOI: 10.1002/nme

Table XVI. Performance of multipole-to-local kernels for single precision using Tesla C2050, measured in Gflop/s (top: $r = 32$, bottom: $r = 256$).

Code / κ	2	3	4	5	6
CPU	36.9	94.9	119.8	123.9	125.2
	(1.44e–3)	(1.20e–3)	(1.08e–2)	(9.83e–2)	(8.43e–1)
GPU1	30.5	58.9	70.3	72.1	72.0
	(2.05e–4)	(1.93e–3)	(1.84e–2)	(1.69e–1)	(1.47e+0)
GPU2	11.1	43.3	113.0	142.7	146.9
	(5.63e–4)	(2.63e–3)	(1.14e–2)	(8.53e–2)	(7.18e–1)
GPU3	3.6	32.6	133.4	230.0	257.3
	(1.75e–3)	(3.49e–3)	(9.68e–3)	(5.29e–2)	(4.10e–1)
GPU4	11.1	49.8	173.1	302.2	358.8
	(5.63e–4)	(2.28e–3)	(7.46e–3)	(4.03e–2)	(2.95e–1)

Code / κ	2	3	4	5	6
CPU	33.8	38.7	40.3	41.0	41.3
	(1.21e–2)	(1.91e–1)	(2.08e+0)	(1.93e+1)	(1.66e+2)
GPU1	44.6	46.0	46.5	46.5	46.4
	(9.08e–3)	(1.60e–1)	(1.80e+0)	(1.70e+1)	(1.48e+2)
GPU2	62.8	116.5	120.3	120.1	134.1
	(6.45e–3)	(6.34e–2)	(6.97e–1)	(6.58e+0)	(5.11e+1)
GPU3	3.7	33.6	135.9	227.2	258.5
	(1.10e–1)	(2.20e–1)	(6.17e–1)	(3.48e+0)	(2.65e+1)
GPU4	62.7	64.8	192.1	319.0	375.5
	(6.46e–3)	(1.13e–1)	(4.36e–1)	(2.48e+0)	(1.82e+1)

The numbers in parenthesis are the timing in second.
Note that GPU4 is equivalent to GPU2 for $\kappa = 2$, and the CPU code is parallelized with 12 cores on dual Xeon X5680 CPUs (3.33 GHz).

from the improvement of the bandwidth by a factor of $32 \times 2/21.3 = 3.00$ rather than the improvement of arithmetic performance by a factor of $(12/8) \times (2.33/3.33) = 2.14$, because the bottleneck of the CPU code, which is based on the sibling-blocking scheme (GPU2), is the bandwidth rather than computation. We also observe that the drop in performance on the CPU when going from single to double precision is less than 2 with $r = 32$ and very close to 2 with $r = 256$.

In the case of single precision, the trends for the C2050 are similar to that of C1060 in Table XI. The more recent Fermi hardware is approximately 1.5–2 times faster than the Tesla C1060. The drop in performance from single to double varies but was found to be close to 2.

11. CONCLUSION

We investigated techniques to accelerate the black-box FMM of [4] with CUDA-capable GPUs [14], in the case of a uniform FMM tree. This paper focused on the implementation of the time-consuming M2L operation, which consists of at most 189 dense matrix–vector products for every box in a given level. We presented the four schemes for the M2L operation with different memory bandwidth requirements (flop-to-word ratios), which usually become the bottleneck when one runs an application on a GPU. Our strategy to improve the flop-to-word ratio is to block boxes to reuse or share the common data, such as the M2L operators (D-matrices).

In numerical tests using 10^3 to 10^7 particles, the four GPU codes that correspond to the four M2L schemes were implemented for NVIDIA GPUs of compute capability of 1.x and tested on a Tesla C1060. High and low precisions were considered by running with two different Chebyshev expansion orders ($n = 4$ and 8, respectively); $n = 8$ is close to the smallest number that gives the best accuracy with single-precision floating-point arithmetic. All the GPU codes ran without

DOI: 10.1002/nme

Table XVII. Performance of multipole-to-local kernels for double precision using Tesla C2050, measured in Gflop/s (top: $r = 32$, bottom: $r = 256$).

Code / κ	2	3	4	5	6
CPU	23.6	57.4	72.9	75.0	77.8
	(2.65e–4)	(1.98e–3)	(1.77e–2)	(1.62e–1)	(1.36e+0)
GPU1	12.8	22.9	26.6	27.1	27.1
	(4.87e–4)	(4.96e–3)	(4.85e–2)	(4.49e–1)	(3.89e+0)
GPU2	9.5	36.2	77.1	92.0	95.5
	(6.56e–4)	(3.14e–3)	(1.68e–2)	(1.32e–1)	(1.11e+0)
GPU3	2.2	20.1	65.4	97.9	109.6
	(2.84e–3)	(5.65e–3)	(1.98e–2)	(1.24e–1)	(9.63e–1)
GPU4	9.5	26.2	83.4	140.3	165.1
	(6.54e–4)	(4.35e–3)	(1.55e–2)	(8.68e–2)	(6.39e–1)

Code / κ	2	3	4	5	6
CPU	17.0	17.9	19.0	19.2	19.4
	(2.39e–2)	(4.11e–1)	(4.42e+0)	(4.13e+1)	(3.52e+2)
GPU1	23.1	23.7	23.9	23.9	23.9
	(1.75e–2)	(3.12e–1)	(3.51e+0)	(3.30e+1)	(2.87e+2)
GPU2	41.4	52.5	56.3	57.1	57.2
	(9.79e–3)	(1.41e–1)	(1.49e+0)	(1.38e+1)	(1.20e+2)
GPU3	2.2	20.5	66.2	99.0	110.8
	(1.80e–1)	(3.61e–1)	(1.27e+0)	(7.98e+0)	(6.18e+1)
GPU4	41.3	30.3	88.4	145.9	171.1
	(9.81e–3)	(2.44e–1)	(9.48e–1)	(5.42e+0)	(4.00e+1)

The numbers in parenthesis are the timing in second.
Note that GPU4 is equivalent to GPU2 for $\kappa = 2$, and the CPU code is parallelized with 12 cores on dual Xeon X5680 CPUs (3.33 GHz).

significant degeneration of the results. The present GPU codes outperformed a parallelized CPU code using eight cores on dual Intel Xeon E5345 CPUs. Furthermore, we ran the GPU codes on a Tesla C2050, which is based on the Fermi architecture. In comparison with the CPU code using 12 cores of dual Intel Xeon X5680 CPUs, we confirmed the advantage of the GPU codes, including the double-precision case.

The proposed M2L schemes would be applicable to other FMM variants because our schemes are not dependent on the specifics of the bbFMM. In particular, if the M2L operator is dense, our schemes are immediately applicable. For example, the M2L operation of the low-frequency regime FMM [39] related to wave analyses (acoustics, electrodynamics, elastodynamics, etc) can be represented by such a dense matrix–vector product. As of now, we applied the basic and sibling-blocking schemes to the low-frequency fast-multipole accelerated boundary integral equation method [40].

APPENDIX A: DETAILS OF THE BLACK-BOX FAST MULTIPOLE METHOD [4] OF FONG AND DARVE

For simplicity, we will assume that K is a translation invariant and homogeneous function of degree λ. That is, $K(\mathbf{x}+\mathbf{c},\mathbf{y}+\mathbf{c}) = K(\mathbf{x},\mathbf{y})$ for any $\mathbf{c} \in \mathbb{R}^3$ and $K(\alpha\mathbf{x},\alpha\mathbf{y}) = \alpha^{\lambda}K(\mathbf{x},\mathbf{y})$ for any $\alpha \in \mathbb{R}\backslash\{0\}$. These assumptions are common to many particle simulations in electrostatics, elastostatics, stationary Stokes flow problems, etc.

The bbFMM separates the variables $\mathbf{x}$ and $\mathbf{y}$ in the kernel $K(\mathbf{x},\mathbf{y})$ using a Chebyshev interpolation. Such separation of variables is key to constructing an FMM. Let $\mathbf{x}$ be a point in a box T (*target box*) at level k with center $\mathbf{t}$ and let $\mathbf{y}$ be a point in a box S (*source box*) at level k with center $\mathbf{s}$. Let T and S be well-separated. By virtue of homogeneity, we can scale these boxes, with edge lengths

DOI: 10.1002/nme

of $\ell/2^k$, to boxes with unit length. Then, applying the $(n-1)$th order Chebyshev interpolation to $K(\mathbf{x},\mathbf{y})$ for every dimension of $\mathbf{x}$ and $\mathbf{y}$, we obtain

$$K(\mathbf{x},\mathbf{y}) \approx \left(\frac{\ell}{2^{k+1}}\right)^{\lambda} \sum_{\mathbf{l}} \sum_{\mathbf{m}} R_n(\bar{\mathbf{x}}_{\mathbf{l}}, \hat{\mathbf{x}}) K(\bar{\mathbf{x}}_{\mathbf{l}}, \bar{\mathbf{y}}_{\mathbf{m}}) R_n(\bar{\mathbf{y}}_{\mathbf{m}}, \hat{\mathbf{y}}), \tag{A.1}$$

where $\hat{\mathbf{x}} := (\mathbf{x}-\mathbf{t})/(\ell/2^{k+1})$ and $\hat{\mathbf{y}} := (\mathbf{y}-\mathbf{s})/(\ell/2^{k+1})$ are the scaled target and source points, respectively, and

$$\bar{\mathbf{x}}_{\mathbf{m}} = (\bar{x}_{m_1}, \bar{x}_{m_2}, \bar{x}_{m_3}) \in [-1,1]^3 \quad \text{and} \quad \bar{\mathbf{y}}_{\mathbf{m}} = (\bar{y}_{m_1}, \bar{y}_{m_2}, \bar{y}_{m_3}) \in [-1,1]^3$$

are 3-vectors of *Chebyshev nodes*. These are the roots of $T_n(x) := \cos(n \arccos x)$, the Chebyshev polynomial of degree n [41]. The numbers l_i and m_i are integers between 1 and n. Also,

$$R_n(\mathbf{x},\mathbf{y}) := S_n(x_1,y_1) S_n(x_2,y_2) S_n(x_3,y_3),$$

where S_n is the interpolation function defined as

$$S_n(x,y) := \frac{1}{n} + \frac{2}{n} \sum_{k=1}^{n-1} T_k(x) T_k(y) \qquad x,y \in [-1,1].$$

The prefactor $(\ell/2^{k+1})^{\lambda}$ in (A.1) is due to the homogeneity of K. We can rewrite (A.1) in the following matrix form:

$$K(\mathbf{x},\mathbf{y}) \approx \left(\frac{\ell}{2^{k+1}}\right)^{\lambda} \mathbf{S}_n^{\mathrm{T}}(\bar{\mathbf{x}}, \hat{\mathbf{x}}) \mathbf{K}(\bar{\mathbf{x}}, \bar{\mathbf{y}}) \mathbf{S}_n(\bar{\mathbf{y}}, \hat{\mathbf{y}}), \tag{A.2}$$

where $\mathbf{S}_n(\bar{\mathbf{x}}, \hat{\mathbf{x}})$ is the n^3-dimensional vector whose $(l_1 + (l_2-1)n + (l_3-1)n^2)$-th element is defined as $R_n(\bar{\mathbf{x}}_{\mathbf{l}}, \hat{\mathbf{x}})$ and $\mathbf{K}(\bar{\mathbf{x}}, \bar{\mathbf{y}})$ is the $n^3 \times n^3$ matrix whose $(l_1 + (l_2-1)n + (l_3-1)n^2, m_1 + (m_2-1)n + (m_3-1)n^2)$-th element is $K(\bar{\mathbf{x}}_{\mathbf{l}}, \bar{\mathbf{y}}_{\mathbf{m}})$. In general, $\mathbf{K}$ is dense.

The matrix $\mathbf{K}$ in (A.2) corresponds to the M2L operator. Because K is translation invariant, $\mathbf{K}$ can be represented by 316 $(= 7^3 - 3^3)$ matrices $\mathbf{K}^{(i)}$ $(0 \leqslant i < 316)$ that correspond to 316 M2L-transfer vectors indexed by (v_1, v_2, v_3), where

$$(v_1, v_2, v_3) \in [-3,3]^3 \setminus [-1,1]^3.$$

Finally, to improve the computational cost of the M2L operation (A.2), with $\mathbf{K}$ replaced with $\mathbf{K}^{(i)}$, we approximate the rank-n^3 matrix $\mathbf{K}^{(i)}$ with a rank-r matrix by means of the SVD. More precisely, we apply the SVD to the weighted kernel matrix $(\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(i)} (\mathbf{\Omega}^y)^{\frac{1}{2}}$ instead of $\mathbf{K}^{(i)}$ so that the error in the low-rank approximation can be minimized with respect to the L^2 norm in $\mathbf{x}$ and $\mathbf{y}$. The matrices $\mathbf{\Omega}^x$ and $\mathbf{\Omega}^y$ are $n^3 \times n^3$ diagonal matrices whose $(l_1 + (l_2-1)n + (l_3-1)n^2)$-th diagonal elements are given by $(\pi/n)^3 \sqrt{1-\bar{x}_{l_1}} \sqrt{1-\bar{x}_{l_2}} \sqrt{1-\bar{x}_{l_3}}$ and $(\pi/n)^3 \sqrt{1-\bar{y}_{l_1}} \sqrt{1-\bar{y}_{l_2}} \sqrt{1-\bar{y}_{l_3}}$, respectively. They correspond to quadrature weights (see [4]).

Now, let us consider the SVD of the following $n^3 \times 316n^3$ matrix $\mathbf{K}_{\mathrm{fat}}$ and $316n^3 \times n^3$ matrix $\mathbf{K}_{\mathrm{thin}}$:

$$\mathbf{K}_{\mathrm{fat}} := \left[(\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(0)} (\mathbf{\Omega}^y)^{\frac{1}{2}}, \quad (\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(1)} (\mathbf{\Omega}^y)^{\frac{1}{2}}, \quad \ldots, \quad (\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(315)} (\mathbf{\Omega}^y)^{\frac{1}{2}} \right],$$

$$\mathbf{K}_{\mathrm{thin}} := \left[(\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(0)} (\mathbf{\Omega}^y)^{\frac{1}{2}}; \quad (\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(1)} (\mathbf{\Omega}^y)^{\frac{1}{2}}; \quad \ldots; \quad (\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(315)} (\mathbf{\Omega}^y)^{\frac{1}{2}} \right].$$

We used the MATLAB [42] notations ',' and ';' for row and column vectors. We obtain the following 316 *SVD-compressed transfer matrices* (*operators*):

$$\mathbf{D}^{(i)} := \mathbf{U}_r^{\mathrm{T}} (\mathbf{\Omega}^x)^{\frac{1}{2}} \mathbf{K}^{(i)} (\mathbf{\Omega}^y)^{\frac{1}{2}} \mathbf{V}_r,$$

DOI: 10.1002/nme

where $\mathbf{U}_r$ (respectively $\mathbf{V}_r$) is the $n^3 \times r$ matrix whose ith column is the ith left (resp. right) singular vectors associated with the r largest singular values of $\mathbf{K}_{\text{fat}}$ (resp. $\mathbf{K}_{\text{thin}}$). The compressed transfer matrices $\mathbf{D}^{(i)}$ are $r \times r$-dimensional and dense. It was found that choosing the number r (*cut-off number*) equal to $n^3/2$ gives good results.

Using the fact that $\mathbf{\Omega}^x$ and $\mathbf{\Omega}^y$ are invertible and that $\mathbf{U}_r$ and $\mathbf{V}_r$ are unitary matrices, we can express the far field interaction between two well-separated boxes T and S at level k as follows:

$$f(\mathbf{x}_i) \approx \left(\frac{\ell}{2^{k+1}}\right)^{\lambda} \mathbf{S}_n^{\mathrm{T}}(\bar{\mathbf{x}}, \hat{\mathbf{x}}_i)(\mathbf{\Omega}^x)^{-\frac{1}{2}} \mathbf{U}_r \mathbf{D}^{(T,S)} \mathbf{V}_r^{\mathrm{T}} (\mathbf{\Omega}^y)^{-\frac{1}{2}} \sum_{\{j \mid \mathbf{y}_j \in S\}} \mathbf{S}_n(\bar{\mathbf{y}}, \hat{\mathbf{y}}_j)\, \sigma_j$$

for $\mathbf{x}_i \in T$, where (T, S) is a short hand notation indicating one of the 316 indices that corresponds to the M2L-transfer vector between T and S. Here, the product of $\mathbf{D}^{(T,S)}$ with its right-hand side vector is called as the M2L operation (see Equation (2)) and its product with the remaining part (sparse matrix) is called as the post-M2L operation, which is not expensive relative to the M2L operation.

APPENDIX B: GPU PSEUDOCODE

```
1  void M2L_first( int r, float D[316][r][r], float M[][r], float L[][r],
2                  int num_source_boxes[], int list_source_boxes[][189],
                   int list_D_indices[][189] )
3  {
4    // Current thread-block is assigned to a target box T in a given level k.
5    int T = blockIdx.x;        // 0 ⩽ T < 8^k.
6    // Current thread is assigned to the i-th element of T's L-vector.
7    int i = threadIdx.x;       // 0 ⩽ i < r.
8    // Initialize the i-th element of T's L-vector.
9    float Li = 0;
10
11   // Loop over at most 189 source boxes.
12   for( int a = 0; a < num_source_boxes[T]; a++ ) {
13     // Read the a-th source box S from the list of source boxes (interaction list) of T.
14     int S = list_source_boxes[T][a];        // 0 ⩽ S < 8^k.
15     // Read the index (T, S) to the D-matrix associated with T and S.
16     int dindex = list_D_indices[T][a];      // 0 ⩽ dindex < 316.
17     // Read the i-th element of S's M-vector and store it into shared memory.
18     __shared__ float sM[r];
19     sM[i] = M[S][i];
20     // Synchronize all the threads to ensure that sM[ ] is available from shared memory.
21     __syncthreads();
22     // Loop over columns.
23     for( int j = 0; j < r; j++ ) {
24       // Read (i, j)-th element of D^(T,S) into a register Dij.
25       float Dij = D[dindex][j][i];
26       // Perform Eq. (3) as a MAD operation.
27       Li += Dij * sM[j];
28     }
29     // Synchronize all the threads to ensure that other threads are done using sM[i].
30     __syncthreads();
31   }
32
33   // Write Li to the device memory.
34   L[T][i] = Li;
35 }
```

Listing 7. Pseudo multipole-to-local kernel of the firsts cheme (basic scheme).

DOI: 10.1002/nme

```
1  void M2L_second( ... )
2  {
3    // Assign current thread-block to cluster TC in a given level k.
4    int TC = blockIdx.x;          // 0 ⩽ TC < 8^{k-1}.
5    // Assign current thread to row i of the L-data.
6    int i = threadIdx.x;          // 0 ⩽ i < r.
7    // Initialize the ith rows of the eight L-vectors to be computed.
8    float Li[8] = {0,0,0,0,0,0,0,0};
9
10   for all 26 source-clusters SC of TC {
11     // Read the ith row of SC's eight M-vectors into shared memory.
12     __shared__ float sM[8][r];
13     sM[:][i] = M[SC][:][i];
14
15     for all 27 interaction-kinds kind {
16       if( this kind is allowable with SC ) { // Refer to Table V.
17         for all j, 0 ⩽ j < r {
18           // Read the (i,j)th element of the D-matrix.
19           float Dij = D[index for pair (SC,TC) and kind][j][i];
20           // Compute for all pairs of boxes (T_a,S_d),(T_b,S_e),... for this kind (Table V).
21           Li[T_a] += Dij * sM[S_d][j]; ...
22         }
23       }
24     }
25   }
26
27   // Write Li to the device memory.
28   L[TC][:][i] = Li[:];
29 }
```

Listing 8. Pseudo multipole-to-local kernel of the second scheme (sibling-blockingscheme).

```
1  void M2L_third( ... )
2  {
3    // Assign current thread-block to chunk G for a given level k.
4    int G = blockIdx.x;           // 0 ⩽ G < 8^{k-1}/B^3.
5    // Assign current thread to target cluster TC in G.
6    int TC = threadIdx.y;         // 0 ⩽ TC < B^3.
7    // Assign current thread to tile row i of the the eight tiled L-vectors L^p for TC.
8    int i = threadIdx.x;          // 0 ⩽ i < r/P.
9
10   for all row tiles p, 0 ⩽ p < P {
11     // Initialize the eight tiled L-vectors.
12     float Lp[8] = {0,0,0,0,0,0,0,0};
13     for all column tiles q, 0 ⩽ q < Q {
14       for all 26 source-clusters SC of TC in G {
15         // Read the M^q of SC's eight siblings into shared memory.
16         __shared__ float Mq[B^3][8][r/Q];
17         Mq[TC][:][i] = M[SC][:][q][i];
18         for all 27 interaction-kinds kind {
19           if ( this kind is allowable with SC ) { // Refer to Table V.
20             // Read D^pq and store it in shared memory.
21             __shared__ float Dpq[r/P][r/Q];
22             Dpq[:][i] = D[index for pair (SC,TC) and kind][p][q][:][i];
23             for all tile columns j, 0 ⩽ j < r/Q {
24               // Compute for all pairs of boxes (T_a,S_d),(T_b,S_e),... for this kind (Table V).
25               Lp[T_a] += Dpq[i][j] * Mq[TC][S_b][j]; ...
26             }
27           }
28         }
29       }
30     }
31
32     // Write Lp into the device memory.
33     L[TC][:][p][i] = Lp[:];
34   }
35 }
```

Listing 9. Pseudo multipole-to-local kernel of the third scheme (cluster-blocking scheme).

DOI: 10.1002/nme

```
1  void M2L_fourth( ... )
2  {
3    // Assign current thread-block to chunk G for a given level k and row tile p.
4    int G = blockIdx.x;     // 0 ⩽ G < 8^k/(8B^3).
5    int p = blockIdx.y;     // 0 ⩽ p < P.
6    // Assign curent thread to target box T in G.
7    int T = threadIdx.x;    // 0 ⩽ T < 8B^3.
8
9    for all columns j, 0 ⩽ j < r {
10     // Read M_j of (2B+4)^3 boxes around G.
11     __shared__ float Mj[(2B+4)^3];
12     Mj[:] = M[j][:];
13     for all tile rows i, pr/P ⩽ i < (p+1)r/P {
14       // Read 316 D_ij.
15       __shared__ float Dij[316];
16       Dij[:] = D[i][j][:];
17       // Initialize L_ij(T).
18       float Lij = 0;
19       for all 189 source boxes S in T's interaction-list {
20         Lij += Dij[index for pair (T,S)] * Mj[S];
21       }
22       // Write L_ij(T) to L_i(T).
23       L[T][i] += Lij;
24     }
25   }
26 }
```

Listing 10. Pseudo multipole-to-local kernel of the fourth scheme (*ij*-blocking scheme).

ACKNOWLEDGEMENTS

We thank the Army High Performance Computing Research Center at Stanford for letting us use some of their facilities. In addition, the first author is grateful to the financial support from MEXT KAKENHI (22760062) and the Hori Information Science Promotion Foundation.

REFERENCES

1. Rokhlin V. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics* 1985; **60**(2):187–207.
2. Greengard L, Rokhlin V. A fast algorithm for particle simulations. *Journal of Computational Physics* 1987; **73**(2):325–348.
3. Nishimura N. Fast multipole accelerated boundary integral equation methods. *Applied Mechanics Review* 2002; **55**:299–324.
4. Fong W, Darve E. The black-box fast multipole method. *Journal of Computational Physics* 2009; **228**(23): 8712–8725.
5. Gimbutas Z, Rokhlin V. A generalized fast multipole method for nonoscillatory kernels. *SIAM Journal of Scientific Computing* 2002; **24**(3):796–817.
6. Ying L, Biros G, Zorin D. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics* 2004; **196**(2):591–626.
7. Martinsson PG, Rokhlin V. An accelerated kernel-independent fast multipole method in one dimension. *SIAM Journal of Scientific Computing* 2007; **29**(3):1160–1178.
8. Greengard L, Gropp WD. A parallel version of the fast multipole method. *Computers and Mathematics Application* 1990; **20**(7):63–71.
9. Zhao F, Johnsson SL. The parallel multipole method on the connection machine. *SIAM Journal Scientific and Statistical Computing* 1991; **12**(6):1420–1437.
10. Cruz FA, Knepley MG, Barba LA. PetFMM – A dynamically load-balancing parallel fast multipole library. *International Journal for Numerical Methods in Engineering* 2011; **85**(4):403–428.
11. ATI Radeon HD 5970 Graphics Feature Summary. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5970/Pages/ati-radeon-hd-5970-overview.aspx#1 [18 April 2011].
12. AMD Opteron 6000 series platform. http://www.amd.com/US/PRODUCTS/SERVER/PROCESSORS/6000-SERIES-PLATFORM/Pages/6000-series-platform.aspx [18 April 2011]. The price is referenced from http://www.amd.com/us/products/pricing/Pages/server-opteron.aspx [18 April 2011].

DOI: 10.1002/nme

13. Intel Xeon Processor 5600 series. http://www.intel.com/itcenter/products/xeon/5600/?iid=gg_work+home_xeon5600, [18 April 2011]. The price is referenced from http://www.intc.com/priceList.cfm [18 April 2011].
14. *CUDA Programming Guide Version 3.2*. http://www.nvidia.com/object/cuda_develop.html [18 April 2011].
15. Khronos Group's OpenCL homepage. http://www.khronos.org/opencl/ [18 April 2011].
16. Gumerov N, Duraiswami R. Fast multipole methods on graphics processors. *Journal of Computational Physics* 2008; **227**(18):8290–8313.
17. White CA, Head-Gordon M. Rotating around the quartic momentum barrier in fast multipole method. *The Journal of Chemical Physics* 1996; **105**(12):5061–5067.
18. Yokota R, Narumi T, Sakamaki T, Kameoka S, Obi S, Yasuoka K. Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. *Computer Physics Communications* 2009; **180**(11):2066–2078.
19. Hamada T, Narumi T, Yokota R, Yasuoka K, Nitadori K, Taiji M. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, Oregon. Article No. 62, 2009.
20. Lashuk I, Chandramowlishwaran A, Langston H, Nguyen T-A, Sampath R, Shringarpure A, Vuduc R, Ying L, Zorin D, Biros G. A massively parallel adaptive fast-multipole method on heterogeneous architectures. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, Oregon. Article No. 58, 2009.
21. Ying L, Biros G, Zorin D, Langston H. A new parallel kernel-independent fast multipole algorithm. *Proceedings of SC03, The SCxy Conference series*, Phoenix, Arizona, 2003.
22. Chandramowlishwaran A, Madduri K, Vuduc R. Diagnosis, tuning, and redesign for multicore performance: a case study of the fast multipole method. *Proceedings of ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, 2010.
23. Carrier J, Greengard L, Rokhlin V. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal on Scientific and Statistical Computing* 1988; **9**:669–686.
24. Cheng H, Greengard L, Rokhlin V. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics* 1999; **155**(2):468–498.
25. Coulaud O, Fortin P, Roman J. High performance BLAS formulation of the multipole-to-local operator in the fast multipole method. *Journal of Computational Physics* 2008; **227**(3):1836–1862.
26. BLAS (Basic Linear Algebra Subprograms) homepage. http://www.netlib.org/blas/ [18 April 2011].
27. Coulaud O, Fortin P, Roman J. High performance BLAS formulation of the adaptive fast multipole method. *Mathematical and Computer Modelling* 2010; **51**:177–188.
28. NVIDIA's Tesla C1060 board specification. http://www.nvidia.com/docs/IO/43395/BD-04111-001_v05.pdf [18 April 2011].
29. NVIDIA's Tesla C2050 board specification. http://www.nvidia.com/object/personal-supercomputing.html [18 April 2011].
30. Volkov V, Demmel J. LU, QR and Cholesky factorizations using vector capabilities of GPUs. *LAPACK Working Note* 2008; **202**:UCB/EECS-2008-49. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html [18 April 2011].
31. NVIDIA's GeForce 8 series specification. http://www.nvidia.com/page/geforce8.html [18 April 2011].
32. NVIDIA's GeForce 9800GTX+ board specification. http://www.nvidia.com/object/product_geforce_9800_gtx_plus_us.html [18 April 2011].
33. NVIDIA's GeForce GTX 285 board specification. http://www.nvidia.com/object/product_geforce_gtx_285_us.html [18 April 2011].
34. The present CUDA kernels etc are available from http://sourceforge.net/projects/bbfmmgpu [18 April 2011].
35. Quad-Core Intel Xeon Processor 5300 Series Datasheet, September 2007. http://www.intel.com/Assets/en_US/PDF/datasheet/315569.pdf [18 April 2011].
36. Intel 5000X Chipset Memory Controller Hub (MCH) Datasheet. http://www.intel.com/products/server/chipsets/5000x/5000x-technicaldocuments.htm [18 April 2011].
37. Volkov V, Kazian B. Fitting FFT onto the G80 architecture, http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf [18 April 2011].
38. Glaskowsky P. NVIDIA's Fermi: The first complete GPU architecture, September 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf [18 April 2011].
39. Epton MA, Dembart B. Multipole translation theory for the 3-dimensional Laplace and Helmholtz equations. *SIAM Journal Scientific Computing* 1995; **16**(4):865–897.
40. Takahashi T, Cecka C, Darve E. An implementation of low-frequency fast multipole BIEM for Helmholtz' equation on GPU. *Proceedings of JSME 23rd Computational Mechanics Conference (CD-ROM)*, Hokkaido, Japan, 2010; 319–321.
41. Abramowitz M, Stegun IA (eds). *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables (eighth Dover printing)*. Dover: New York, 1972.
42. Website for MATLAB. http://www.mathworks.com/products/matlab/ [18 April 2011].

DOI: 10.1002/nme