C. R. Mecanique 339 (2011) 185–193

Contents lists available at ScienceDirect

Comptes Rendus Mecanique

www.sciencedirect.com

High Performance Computing / Le Calcul Intensif

The fast multipole method on parallel clusters, multicore processors, and graphics processing units

La méthode multipôle rapide sur des grappes d'ordinateurs en parallèle, des processeurs multi-cœurs, et des processeurs graphiques

Eric Darve [a,*], Cris Cecka [a], Toru Takahashi [b]

[a] *Mechanical Engineering Department, Institute for Computational and Mathematical Engineering, Stanford University, Durand 209, 496 Lomita Mall, 94305-3030 Stanford, CA, USA*
[b] *Department of Mechanical Science and Engineering, Nagoya University, Japan*

ARTICLE INFO

Article history:
Available online 11 January 2011

Keywords:
Computer science
Fast multipole method
Parallel computer

Mots-clés :
Informatique, algorithmique
Méthode multipôle rapide
Calculateur parallèle

ABSTRACT

In this article, we discuss how the fast multipole method (FMM) can be implemented on modern parallel computers, ranging from computer clusters to multicore processors and graphics cards (GPU). The FMM is a somewhat difficult application for parallel computing because of its tree structure and the fact that it requires many complex operations which are not regularly structured. Computational linear algebra with dense matrices for example allows many optimizations that leverage the regular computation pattern. FMM can be similarly optimized but we will see that the complexity of the optimization steps is greater. The discussion will start with a general presentation of FMMs. We briefly discuss parallel methods for the FMM, such as building the FMM tree in parallel, and reducing communication during the FMM procedure. Finally, we will focus on porting and optimizing the FMM on GPUs.

RÉSUMÉ

Dans cet article, nous présentons l'implémentation de la méthode multipôle rapide (FMM) sur des calculateurs parallèles modernes, depuis les grappes parallèles aux processeurs multi-cœurs et aux cartes graphiques (GPU). La FMM est une application difficile à paralléliser à cause de la structure d'arbre et le fait qu'elle demande des opérations complexes qui ne sont pas structurées de façon régulière. L'algèbre linéaire computationnelle avec des matrices denses par exemple permet des optimisations qui utilisent les motifs réguliers de calcul. La FMM peut être optimisée de façon similaire mais nous verrons que la complexité de l'optimisation est supérieure. La discussion débute par une présentation générale de la FMM. On discutera brièvement des méthodes parallèles pour la FMM, comme la construction de l'arbre, et la réduction des communications durant le calcul. Finalement, nous présenterons plus en détail le développement et l'optimisation de la FMM sur GPUs.

* Corresponding author.
E-mail address: darve@stanford.edu (E. Darve).

1631-0721/$ – see front matter
doi:10.1016/j.crme.2010.12.005

Fig. 1. Two spheres containing r_i and r_j illustrating the restricted domain of validity of (1).

1. Fast multipole method

The fast multipole method (FMM) is a technique to calculate sums like:

$$\phi(r_i) = \sum_{j=1}^{N} K(r_i, r_j)\sigma_j, \quad 1 \leqslant i \leqslant N$$

in $\mathcal{O}(N)$ or $\mathcal{O}(N \ln^{\alpha} N)$ arithmetic operations. There are many different FMM formulations. Most of them rely on low-rank approximations of the kernel function K in the form:

$$K(r, r') = \sum_{q=1}^{p} u_q(r) \sum_{s=1}^{p} T_{qs} v_s(r') + \varepsilon \tag{1}$$

A quick way to realize how this form can be used for fast summations consists in realizing that the sum $\sum_{j=1}^{N} K(r_i, r_j)\sigma_j$ can now be approximated in three steps:

1. Multipole coefficients: $M_s = \sum_{j=1}^{N} v_s(r_j)\sigma_j$
2. Local coefficients: $L_q = \sum_{s=1}^{p} T_{qs} M_s$
3. Potential: $\phi(r_i) = \sum_{q=1}^{p} u_q(r_i) L_q + \varepsilon$

Each step requires $\mathcal{O}(N)$ floating point operations. Later on, we will use the words interact or interaction to describe the application of the Multipole-to-Local (M2L) operator T_{qs}.

There are many ways to construct the functions u_q, v_s, and the matrix T_{qs} [1]. To give a simple example in one dimension, let us assume that we have an order p interpolation scheme, given by some functions $S_q(r)$ and $S'_s(r')$, and with interpolation points $\bar{r}_q$ and $\bar{r}'_s$, such that:

$$\begin{aligned} K(r, r') &= \sum_{q=1}^{p} K(\bar{r}_q, r') S_q(r) + \varepsilon_0 \\ &= \sum_{q=1}^{p} \sum_{s=1}^{p} K(\bar{r}_q, \bar{r}'_s) S_q(r) S'_s(r') + \varepsilon \end{aligned}$$

This example leads to the following definitions: $u_q(r) = S_q(r)$, $T_{qs} = K(\bar{r}_q, \bar{r}'_s)$, and $v_s(r') = S'_s(r')$. A popular interpolation scheme is the family of Chebyshev polynomials. Other constructions of fast multipole expansions include Taylor expansions, spherical harmonics, plane waves, ...

A technical difficulty in the method is that in most cases such a low-rank approximation does not exist when r and r' can assume any values. Instead, a common assumption is that r and r' are sufficiently well "separated". This is true for example for kernels that have a singularity when $r = r'$, such as $K(r, r') = 1/|r - r'|$. The condition is then typically given in the following form. Assume that we have two points C and C' and spheres S and S' of radius R centered at C and C' with $|C - C'| > 2R$, then an approximation of type (1) exists for all $r_i \in S$ and $r_j \in S'$. See Fig. 1. In that case we often have that the error decays exponentially fast with the order of the expansion:

$$p = \mathcal{O}(\ln 1/\varepsilon)$$

One of the main discoveries of the FMM is a way to construct a fast $\mathcal{O}(N)$ method in cases where approximation (1) must be satisfied. The approach is based on a well-known octree structure [2,3]. For example, consider a set of particles at locations r_j with charges σ_j. These particles can be enclosed in a cube. If one subdivides the cube into eight octants, no acceleration is possible since all octants are, by construction, neighbors of one another and consequently condition (1) is not satisfied. See Fig. 2(a).

Instead each octant is further decomposed into octants as shown in Fig. 2(b). This allows computing all interactions between pairs of clusters that are not nearest neighbors, e.g., the pair of gray clusters in Fig. 2(b). Interactions between particles in neighboring clusters are computed by further subdividing each octant into smaller octants. This process is

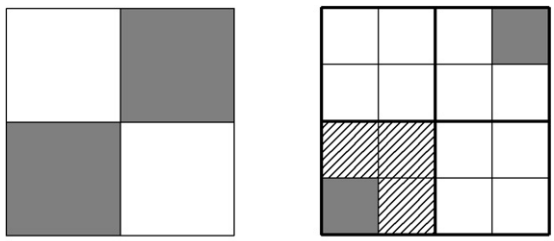

Fig. 2. 2D illustration: (a) A square is subdivided into four quadrants; approximation (1) (well-separated condition) does not apply to any pair of quadrants. (b) Each quadrant from (a) is further subdivided into 4 quadrants. The interaction between particles in the two gray clusters can now be computed using the low-rank approximation (1). The interaction between the gray cluster and its neighboring clusters (with the diagonal pattern) cannot be computed at this level of the tree. A further decomposition of each quadrant is required.

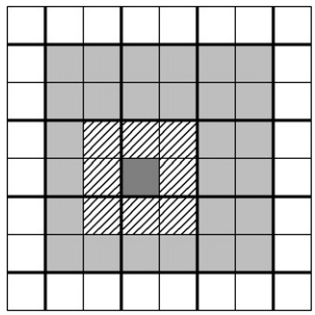

Fig. 3. 2D illustration: basic interaction list. In a typical situation, the gray cluster near the center has to interact with all the light gray clusters around it. In 3D there are 189 of them.

recursively repeated until the clusters are small enough that interactions between particles in neighboring clusters can be computed directly, that is by direct evaluation of $\sum_j K(r_i, r_j)\sigma_j$.

In a typical situation, a cluster is required to interact only with clusters that are neighbors (i.e., they share a vertex, edge, or face) of their parent cluster. In addition, since they cannot interact with clusters that are their own neighbors, each cluster is therefore only required to interact with a relatively small number of clusters, at most 189 ($= 6^3 - 3^3$) in 3D. This is shown in Fig. 3.

In most FMMs, the expansion (1) is not applied directly to each independent level of the tree. Indeed, computing M_s^C — the multipole coefficients for cluster C — for each cluster directly using the particle positions, r_i and r_j, as in (1), the total cost would be $\mathcal{O}(N \ln N)$ since there are N particles and $\mathcal{O}(\ln N)$ levels in the tree. Similarly, if we compute the potential $\phi(r_i)$ directly using each level's L_q^C, the total cost would also be $\mathcal{O}(N \ln N)$. Instead, in the FMM, one evaluates u_q^C and v_s^C only for leaf clusters C and determines operators to propagate this information up and down the tree. This is illustrated in Fig. 4. We start by computing M_s^C for all leaf clusters C. Then $M_s^{C'}$ is computed for all clusters C' in the tree by progressively propagating the information upward using Multipole-to-Multipole operators. Partial local coefficients L_q^C are computed for all clusters by applying the Multipole-to-Local operator $T_{qs}^{CC'}$ to $M_s^{C'}$. Finally, this information is propagated down the tree using Local-to-Local operators, until the final set of local coefficients L_q^C are obtained for all leaf clusters C. The potential $\phi(r_i)$ can finally be estimated after adding near neighbor interactions that are computed directly.

Mathematical analysis is required to determine the exact definition of u_q and v_s and the nature of the Multipole-to-Multipole (M2M), Multipole-to-Local (M2L), and Local-to-Local (L2L) operators. However, in most cases, these operators are dense matrices which need to multiply the vectors of coefficients M_s^C and L_q^C [1]. See Fig. 4.

An additional difficulty in the method arises when the particles r_j are distributed inhomogeneously in space. In that case a uniform octree cannot be used and instead an adaptive octree is constructed, in which the number of levels varies spatially. More levels are used in places where the concentration of particles is greater. See Fig. 5. This allows maintaining the optimal $\mathcal{O}(N)$ computational cost. A few additional operators are then required. See for example Lashuk et al. [4].

2. Parallelization on distributed clusters and multicore processors

The parallelization of the calculation has effectively two components: the construction of the appropriate distributed tree structure, and the application of the FMM itself.

Many difficulties arise when using adaptive trees [5]. The construction of the tree structure is fairly straightforward when the number of nodes is not too large, but special techniques must be applied beyond. Many implementations are based on the scheme of Warren and Salmon [6]. The basic difficulty is locating, in memory, a given cluster on another processor.

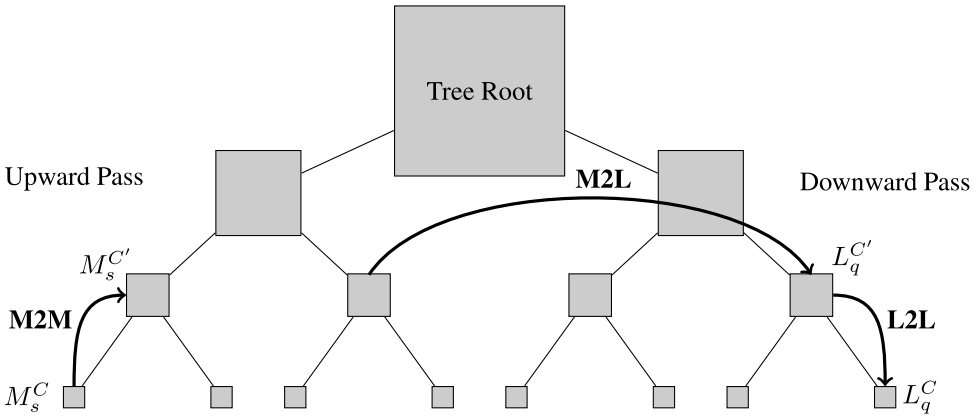

Fig. 4. Illustration of the three passes in the FMM.

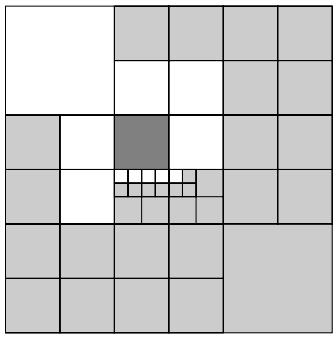

Fig. 5. Adaptive FMM tree. When the distribution of points is highly inhomogeneous, an adaptive tree is required. We consider the dark gray cluster near the center. All clusters that interact with the gray cluster through a multipole expansion are shown in light gray. This interaction is either of the type M2L or Particle-to-Local or Multipole-to-Particle [4]. The total computational cost remains $\mathcal{O}(N)$ with these changes.

Warren and Salmon [6] proposed to associate each cluster with a key. The mapping of this key to memory locations is then achieved via a hash table. This was shown to be an efficient approach.

The construction of the tree itself [4] follows two steps. In step 1, a distributed array with all the leaves of the tree is constructed and Morton-sorted.[1] In step 2, the local essential tree (LET) for each process is constructed. This tree is the union of the clusters in the interactions lists of all the leaf clusters and their ancestors owned by a process. This tree defines exactly all the information that will be necessary to evaluate the field values $\phi(r_i)$ at all r_i owned by a process. Typically this procedure is completed by a further load balancing step to try to balance the workload across processors and reduce the required communication between nodes [8].

The FMM itself can be parallelized in several ways. There are three basic classes of operations:

- **Independent operations.** The direct interactions, $\sum_j K(r_i, r_j)\sigma_j$, are independent of (and can therefore be computed concurrently with) the far away contributions obtained through multipole expansions. Additionally, the application of the M2L operator has a lot of independent operations when looping over all the clusters in the interaction list and over all clusters in the tree. In the M2L step, only a final reduction to get L_q^C requires synchronization.
- **Sequential operations.** The M2M operation for a cluster must be complete before the M2L operator can be applied with that cluster. Similarly the L2L operator can only be applied when the M2L operator and L2L operators for all ancestor clusters are complete.
- **Linear algebra operations.** The application of the M2M, L2L, and M2L operators, which are often formulated as matrix-vector products, can be parallelized.

The required communication steps are often more efficient when they follow explicitly the tree structure of the FMM itself. Such a procedure is suggested in Lashuk et al. [4]. Let us consider the general case of an adaptive tree structure as shown in Fig. 5. In that case, each process is assigned a subset of the leaf clusters. In the upward pass, each process goes

[1] Morton-order is a space filling curve. Points in multi-dimensions are ordered by interleaving the binary representation of their coordinate values [7]. It is also called Z-order because of the shape of the curve.

Algorithm 1 All-reduce-like operation using a tree structure for the FMM. The input are partial M_s^C computed by each process. The output is the set of all complete M_s^C data in the local essential tree (LET) for each process. We assume that we have 2^d processes that need to communicate. The current process is denoted by r.

```
1  r = process rank;
2  M_r = all partial M_s^C data owned by r;
3  For i = d − 1 to 0 {
4      // At the first step, i = d − 1 and only two groups of processes are considered.
5      // As i increases, groups are split in two. At the end, each group contains two processes only.
6      s = r XOR 2^i;                    // The partner process in the other group.
7      // Denote G_0 the group of processes containing r and G_1 the group of processes containing s.
8      u_s = s AND (2^d − 2^i);          // The starting index of the clusters in the group G_1.
9      u_e = s OR (2^i − 1);             // The final index of the clusters in the group G_1.
10     v_s = r AND (2^d − 2^i);          // The starting index of the clusters in the group G_0.
11     v_e = r OR (2^i − 1);             // The final index of the clusters in the group G_0.
12     Send to s all the clusters in M_r which belong to the interaction list
         of at least one cluster in G_1 (with index range from u_s to u_e);
13     Delete from M_r all clusters that are not in the interaction list of
       any cluster in G_0 (with index range from v_s to v_e);
14     Receive data from s and append to M_r;
15     If r has multiple partial M_s^C data for a given cluster C, reduce this
       data;
16 } // end for loop
```

up in the tree computing partial M^C data. This data is complete whenever a process owns all the children clusters of C, otherwise only partial M^C data is obtained at this step.

The procedure follows a recursive process. The processes are first split into two groups. A process r in group 0 is associated with a process s in group 1 in a unique fashion ("partner" process). Then r sends to s all information owned by r (partial M data) that is needed by at least one process in group 1. Similarly r receives information from s, that is required by at least one cluster in group 0. Then group 0 is split into two groups and the process is repeated. At the beginning each process owns partial M^C data. When a process receives multiple M^C data for the same C (this is the case when multiple processes computed partial M^C data for cluster C), it reduces the data before sending it to other processes. This is similar to an all reduce operation based on a tree structure, but it incorporates steps specific to the FMM. The algorithm is given above. The interaction list of a cluster has been explained before. See Fig. 3 for example.

For a detailed discussion of the performance of the FMM on a single node multi-core processor see [9,10]. In particular the authors report performance on multi-core processors on par with an optimized GPU implementation. They consider the FMM implementation of Ying et al. [11]. Using an Intel Nehalem-EX system (four sockets, 8 cores per socket, two threads per core = 64 threads total), they optimized their code to achieve speedups of 1.7× over the previous best multithreaded implementation, and of 19.3× over a sequential but highly tuned code [9]. In a recent work, which will be presented at the International Conference for High Performance Computing, Networking, Storage, and Analysis, November 2010 (ACM Gordon Bell Finalist Presentation), a Stokesian particulate flow solver was created and run on a hybrid CPU/GPU cluster. The physical system is a large number of red blood cells in a Stokesian fluid. 200 million deformable red blood cells were simulated. This simulation required integrating parallelism at all levels including inter-node distributed memory parallelism, intra-node shared memory parallelism, data parallelism (vectorization), and fine-grained multithreading for GPUs. On the Oak Ridge National Laboratory's Jaguar PF system, a performance of 0.7 Petaflops was achieved.

Other authors have looked at the parallel implementation of the fast multipole method in the context of uniform trees (with a fixed number of levels): see for example Kurzak and Pettitt [12] and Ogata et al. [13]. Sylvand [14] presents an implementation of the FMM in the European Aeronautic Defence and Space Company N.V. integral equations code, for electromagnetics applications. Wu et al. [15] presents a parallel implementation of the FMM for electromagnetic applications as well.

3. Parallelization on graphics processor units

We will focus the discussion in this section on the implementation of the FMM on Nvidia graphics processor units (GPU), such as processors based on the Fermi architecture and which are CUDA capable (compute unified device architecture).

In this paper we will not review the architecture of Nvidia GPUs. See the CUDA programming guide for details [16]. To summarize the main components, the GPU is a co-processor attached to a host CPU. The GPU is composed of several streaming multi-processors (SM). Each multi-processor is composed of several arithmetic units or cores, capable of processing data concurrently in a SIMD (single instruction multiple data) style. The computation has to be divided into *blocks* (the collection of blocks is called a *grid*), each of which gets processed by one SM. On a GPU, the calculation is performed by

Algorithm 2 Sequence of operations to perform in the M2L stage of the FMM.

```
For all cluster C in this level {
  For all cluster C' in the interaction list of C {
    For all q=1 to p {
      For all s=1 to p {
        L_q^C += T_qs^{CC'} M_s^{C'};
      } } } }
```

Table 1
An equivalence class is a set of (D_i, D'_j) pairs that share the same $T^{DD'}$ matrix data. The size of this class is the number of pairs in the class. The right column shows the number of classes that have a given size. This table gives an idea of the savings that can be expected using the GPU2 approach.

Size of equivalence class	Number of equivalence classes
1	8
2	12
4	6
8	1

large number of threads that all execute the same program, called a *kernel*, using different data. Within an SM, threads can be synchronized and can share data stored in the so-called *shared memory*. The shared memory is almost as fast as the register files; its size varies but is typically 16 kB on most devices. A key optimization consists in using this shared memory space to avoid reading/writing data from/to main memory.

Computing using GPUs is typically limited by the rate at which data can be moved either from the CPU memory to the GPU or from the GPU memory to GPU registers in each SM. In the case of the FMM, the time required to move data from the CPU and GPU is relatively small so that this is not a significant issue. In order to reduce the movement of data between the GPU memory and register files, one has to resort to different types of blocking strategies (see examples of such approaches in Cecka et al. [17,18]) and to using the shared memory on each SM as effectively as possible. One key measure of the efficiency is *arithmetic intensity* which measures the number of floating point operations in a kernel divided by the number of words that need to be read from memory. This is a fairly approximate measure of performance but is a good indicator of whether an algorithm will run well on a GPU or not. Typically, and arithmetic intensity of 16 to 24 flops per word are at least required to enable the device to reach peak computation efficiency.

The primitive loops in the M2L stage of the FMM are shown in Algorithm 2. The loops can be blocked in several ways leading to varying degrees of arithmetic intensity, as we discuss in the remainder of this section.

In the simplest implementation, one defines a kernel that reads in the matrix $T^{CC'}$ and the vector $M^{C'}$, performs a matrix-vector product, and accumulates the result in L_q^C. Most of the data traffic is generated by the matrix $T^{CC'}$ since $M^{C'}$ is only a vector. This approach results in poor efficiency since it requires (basically) reading one number $T_{qs}^{CC'}$ while only performing one multiplication and one addition. Thus, the arithmetic intensity is very low. For the following discussion, we denote this method GPU1.

The first approach to improve performance consists in attempting to reuse the matrix $T^{CC'}$ multiple times before discarding it. Since $T^{CC'}$ is a function only of the vector joining the centers of the clusters, it is the same for many pairs of clusters (C, C'). Consider two neighboring clusters C and C' and their children clusters (using any consistent numbering scheme) D_0 to D_7 and D'_0 to D'_7 respectively. Among all pairs of clusters (D_i, D'_j) that need to interact, many share the same T operator. We will use the symbol $\equiv$ to denote pairs that share the same T operator. For example, we have $(D_0, D'_0) \equiv (D_1, D'_1) \equiv (D_2, D'_2) \equiv \cdots$ since the T matrix is the same in each case. This optimization then consists of loading a $T^{DD'}$ operator, and performing the matrix-vector product $T^{D_k D'_l} M^{D'_l}$ for all pairs $(D_k, D'_l) \equiv (D, D')$. This method is called GPU2. This is summarized in Table 1.

The next optimization consists in blocking together the clusters C and C' so that a given operator T can be shared by even more pairs of clusters (D_i, D'_j). This method is called GPU3.

We performed some numerical tests using $p = 32$ and $p = 256$. The FMM algorithm that was used is described in Fong and Darve [1]. In the naive approach (GPU1), the flop to word ratio is about 1.9–2. With the first optimization above (GPU2) this number increases to 4.2–4.3, while the last optimization (GPU3) increases this number to 21 for $p = 32$ and 32 for $p = 256$.

An alternative approach (method GPU4) consists in moving the loops over q and s to the outside and using as inner loops the loops over C and C' (see Algorithm 2). This allows increasing the arithmetic intensity further since this allows sharing even more T data. In particular, for a given q and s, it is now possible to load all the possible values for $T_{qs}^{CC'}$ (there are 316 different values) and use them to calculate the interaction for a very large number of (D_i, D'_j) pairs simultaneously.

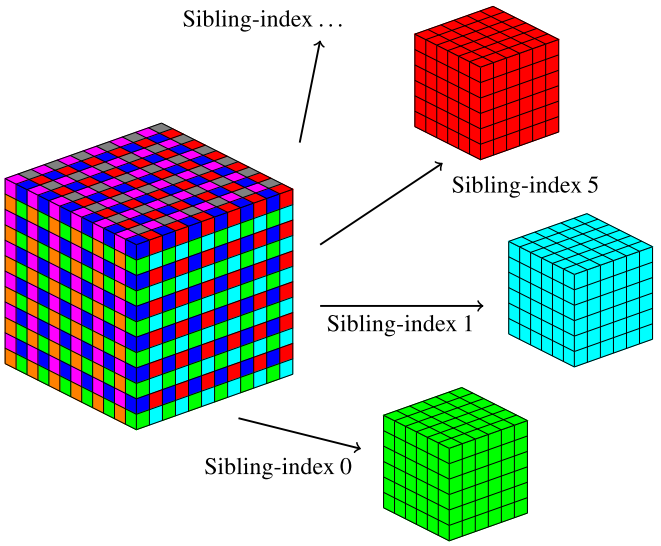

Fig. 6. Memory layout of M data. The data is laid out in memory so that clusters with the same sibling index are contiguous. This allows a fast coalesced access to memory when applying the M2L operator. For the figure, we chose $B = 4$.

By moving the q and s loops to the outside, the arithmetic intensity is further increased to 108 for $p = 32$ and 133 for $p = 256$. Note though that performance cannot be predicted on the basis of the arithmetic intensity only.

The downside of GPU4 is that in the previous versions of the algorithm above (GPU2 and GPU3) the inner loops perform matrix-vector or matrix–matrix products, which can be run efficiently on GPUs. In contrast, in GPU4 the inner loop (which is parallelized) is over the interaction list, which is slightly more irregular (see Fig. 3). The data layout in memory must be chosen carefully in order to allow for coalesced memory accesses (access to aligned contiguous memory addresses by contiguous threads) and avoid bank conflicts. Bank conflicts may arise when certain threads access data in the same memory bank in shared memory. See details in the Nvidia CUDA C Programming Guide [16]. A proper layout of the data in memory and the proper ordering of data access in shared memory can prevent these issues.

We briefly discuss how to optimally layout the data in memory to achieve coalescing and avoid bank conflicts. Let us associate to each cluster a sibling index based on its position with respect to the parent cluster. Although the interaction list pattern differs for clusters with different sibling indices, it is the same for all clusters with the same sibling index. By storing the M data in memory such that clusters with the same sibling index are stored contiguously, we can read the data in coalesced transactions into shared memory. See Fig. 6. In addition, this will allow us to devise a simple scheme to avoid bank conflicts when accessing the data.

The issue of bank conflicts is more subtle. When accessing data in the shared memory, 16 threads in a half-warp (a half-warp is a group of threads executing together in an SM) basically need to access data in different memory banks (see [16] for details). For a given $T^{DD'}$ operator, to calculate the partial local coefficients L for a group of $B \times B \times B$ clusters with the same sibling index, we must load all of the associated M data for a group of $(2+B) \times (2+B) \times (2+B)$ clusters. Note that those clusters will all have the same sibling index (hence, the storing of M data by sibling index). Because of the size of the M data, by default, bank conflicts are expected due to the "padding" with the two ghost clusters. See Fig. 7. However it turns out that a simple remedy exists. Let us assume that the data is laid out first along x, then y, and then z. Assume 16 threads in a half-warp are assigned to work on clusters with the following indices:

$$
\begin{array}{llll}
(i_0, j_0, z_0+0) & (i_0+1, j_0, z_0+0) & (i_0+2, j_0, z_0+0) & (i_0+3, j_0, z_0+0) \\
(i_0, j_0, z_0+1) & (i_0+1, j_0, z_0+1) & (i_0+2, j_0, z_0+1) & (i_0+3, j_0, z_0+1) \\
(i_0, j_0, z_0+2) & (i_0+1, j_0, z_0+2) & (i_0+2, j_0, z_0+2) & (i_0+3, j_0, z_0+2) \\
(i_0, j_0, z_0+3) & (i_0+1, j_0, z_0+3) & (i_0+2, j_0, z_0+3) & (i_0+3, j_0, z_0+3)
\end{array}
$$

The ordering is modified by keeping the y coordinate constant and moving along the z axis. Because of the way the octree is constructed, B is equal to 2^l for some $l \geqslant 2$. In that case the bank index for (i_0, j_0, z_0+1) is equal to the bank index of (i_0, j_0, z_0) plus

$$(2+B)^2 \bmod 16 = 4 + 2^{l+2} + 2^{2l} \bmod 16 = 4$$

Consequently, the banks are accessed conflict-free with this approach. See Fig. 7. These are some of the issues that need to be considered in order to have code that executes at a good fraction of peak efficiency of the card.

We performed some numerical benchmarks for systems with octree depths of 2, 3, 4, 5, and 6. Since we are only presenting results for the M2L kernel in this paper, the number and distribution of particles is irrelevant to our discussion.

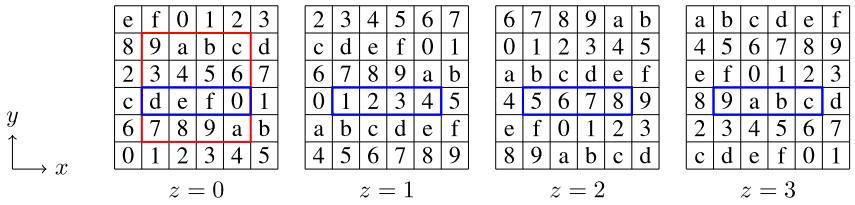

Fig. 7. Pattern of data access used to avoid bank conflicts. The default access pattern, which accesses data following the x, y, z ordering, is shown in the red square. Each bank is numbered from 0 to 9 and then a to f (hexadecimal base). We can see for example that the red square contains two 9s and two as, showing that bank conflicts occur. However, if a half-warp processes data in the blue rectangles, all bank conflicts are avoided. This scheme is simple to implement and yields improved performance.

Table 2
Numerical benchmarks of the GPU methods discussed above. The table indicates the number of Giga floating point operations per second (Gflop/s) sustained for $p = 32$ (top) and $p = 256$ (bottom). CPU was optimized to run on eight cores, and it uses SSE3 instructions (peak is 75 Gflop/s). GPU1 is the simplest implementation. GPU2 uses blocking at the level of octants, while GPU3 groups clusters together to achieve a higher level of blocking. GPU4 moves the q and s loops to the outside achieving an even higher degree of reuse of the T data. The peak performance of the GPU is somewhere between 312 and 624 Gflop/s depending on how many multiply-add merged operations can be performed.

Octree levels	2	3	4	5	6
CPU	11.4	32.0	37.0	37.6	37.6
GPU1	16.2	32.4	39.4	40.9	41.1
GPU2	6.4	24.6	57.7	69.6	71.4
GPU3	1.5	13.8	62.7	128.9	156.8
GPU4	6.5	33.8	120.1	210.3	220.0

Octree levels	2	3	4	5	6
CPU	8.6	8.8	8.6	8.6	8.6
GPU1	41.9	41.9	42.1	41.8	41.7
GPU2	51.7	91.2	100.7	101.7	101.3
GPU3	1.6	14.4	65.2	133.1	161.6
GPU4	51.8	88.7	177.9	241.9	269.5

As a point of reference, octree depths of 2, 3, 4, 5, and 6 correspond approximately to a number of particles equal to $N = 10^3, \ldots, 10^7$. The test kernel is the Laplace kernel $K(r, r') = 1/|r - r'|$. Note that the choice of kernel, in this implementation, does not affect the performance of the M2L operation.

We used a DELL Poweredge 1950 with two quad-core Intel Xeon E5345 CPUs running at 2.33 GHz (4MB shared L2 cache per two cores) with 16 GB of 667 Hz DDR2 SDRAM (5.333 GB/s). 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 9.32 Gflop/s per core and 74.56 Gflop/s in total. For the GPU, we used one of four NVIDIA Tesla C1060 graphics cards, in a Tesla S1070, connected to the above 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 (312, respectively) Gflop/s.

We used Intel's C/C++ compiler 10.1 (with optimizing options `-O3` and `-xT`) for the CPU codes, and NVIDIA's CUDA SDK 2.2 for the GPU codes. The CPU code was based on the same blocking method as GPU2 and parallelized using OpenMP; the code uses all eight cores. It was further optimized by use of SSE3 instructions (vectorization). We summarize the key result which is the number of Gflops achieved in the M2L kernels. We tested two cases with $p = 32$ and $p = 256$. The results are shown in Table 2.

The performance of the CPU in Table 2 depends in large part on the efficiency of the use of the cache. The difference in the size of the dataset for $p = 32$ and $p = 256$ explains in part the difference in performance. As usual those numbers should be taken with some care since further optimization of the CPU code is likely to improve performance. The performance of GPU1 is primarily limited by the memory bandwidth of the GPU and quickly saturates with the problem size. GPU3 and GPU4 offer very high arithmetic intensity (high number of flops per word). In principle they should be close to peak performance. However performance depends on many factors including coalesced memory access, bank conflicts, fused multiply-add operations, number of threads running concurrently in each SM, number of blocks in a CUDA grid, number of blocks running concurrently in each SM, etc. However the obtained numbers show that a good fraction of peak performance is achievable once sufficient optimization of the code has been completed.

We would like to conclude this section by mentioning the work of Yokota et al. who developed codes to run the FMM on clusters of GPUs [19–21]. In Yokota [21], results are presented for a vortex simulation (isotropic turbulence) run with the FMM on 64 GPUs, achieved a performance of 7.5 Tflops. The accuracy was validated with a comparison of the results with a spectral method code. In Hamada [20] (entry for the 2009 Gordon Bell prize and winner in the price per performance category), N body simulations were run on 256 GPUs with the FMM. With 1.6 billion particles, the performance was 42 Tflops.

Sheel et al. [22] has a discussion of FMM algorithms on the special purpose computer MDGRAPE-3. In addition, in recent work, the FMM for Helmholtz kernel (i.e., $K(r,r') = \exp(ik|r-r'|)/|r-r'|$) was investigated for high-frequency case by Xu et al. [23] and low-frequency case by Cwikla et al. [24] in the context of electromagnetic scattering problems. In Takahashi et al. [25], the schemes GPU1 and GPU2 were tentatively exported to the low-frequency FMM accelerated boundary integral equation method, using a non-uniform octree and double-precision floating point arithmetic.

4. Conclusion

This paper has reviewed the basic steps of the fast multipole method from a computer science standpoint. We did not describe the mathematics behind the method (see Fong and Darve [1] for more information). Efficient implementation on distributed memory machines can be become quite difficult when the number of cores is large. Special techniques to construct distributed adaptive octree are required, along with special communication schemes to reduce the communication bottlenecks during the application of the FMM.

Running code on GPU leads to specific optimizations, where one typically tries to increase the number of arithmetic operations performed per word read from memory. This leads to significant changes in the algorithm, primarily in terms of changes in the ordering of the loops and blocking of the operations. However once those optimizations are in place, a good fraction of the peak performance of the hardware can be achieved.

Acknowledgements

We thank the Army High Performance Computing Research Center at Stanford for letting us use some of their facilities.

References

[1] W. Fong, E. Darve, The black-box fast multipole method, Journal of Computational Physics 228 (23) (2009) 8712–8725.
[2] V. Rokhlin, Rapid solution of integral equations of classical potential theory, Journal of Computational Physics 60 (2) (1985) 187–207.
[3] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, Journal of Computational Physics 73 (2) (1987) 325–348.
[4] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, G. Biros, A massively parallel adaptive fast-multipole method on heterogeneous architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, 2009, pp. 1–12.
[5] F.A. Cruz, M.G. Knepley, L.A. Barba, PetFMM – A dynamically load-balancing parallel fast multipole library.
[6] M.S. Warren, J.K. Salmon, A parallel hashed oct-tree *n*-body algorithm, in: Proceedings of the 1993 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, 1993, p. 21.
[7] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, International Business Machines Co., 1966.
[8] H. Sundar, R.S. Sampath, G. Biros, Bottom-up construction and 2:1 balance refinement of linear octrees in parallel, SIAM Journal on Scientific Computing 30 (5) (2008) 2675–2708.
[9] K. Madduri, R. Vuduc, Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, 2010.
[10] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, R. Vuduc, Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures, in: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, 2010, pp. 1–12.
[11] L. Ying, G. Biros, D. Zorin, A kernel-independent adaptive fast multipole algorithm in two and three dimensions, Journal of Computational Physics 196 (2) (2004) 591–626.
[12] J. Kurzak, B.M. Pettitt, Massively parallel implementation of a fast multipole method for distributed memory machines, Journal of Parallel and Distributed Computing 65 (7) (2005) 881.
[13] S. Ogata, T.J. Campbell, R.K. Kalia, A. Nakano, P. Vashishta, S. Vemparala, Scalable and portable implementation of the fast multipole method on parallel computers 1, Computer Physics Communications 153 (3) (2003) 445–461.
[14] G. Sylvand, Performance of a parallel implementation of the FMM for electromagnetics applications, International Journal for Numerical Methods in Fluids 43 (8) (2003) 865–879.
[15] F. Wu, Y. Zhang, Z.Z. Oo, E. Li, Parallel multilevel fast multipole method for solving large-scale problems, IEEE Antennas and Propagation Magazine 47 (4) (2005) 111.
[16] NVIDIA Corporation, NVIDIA CUDA programming guide 3.0, online at www.nvidia.com, Feb 2010.
[17] C. Cecka, A.J. Lew, E. Darve, Assembly of finite element methods on graphics processors, Intl. J. Numerical Methods in Engineering (2009).
[18] C. Cecka, A. Lew, E. Darve, Introduction to assembly of finite element methods on graphics processors, in: IOP Conference Series: Materials Science and Engineering, vol. 10, IOP Publishing, 2010, p. 012009.
[19] R. Yokota, T. Hamada, J.P. Bardhan, M.G. Knepley, L.A. Barba, Biomolecular electrostatics simulation by an FMM-based BEM on 512 GPUs, Arxiv preprint arXiv:1007.4591, 2010.
[20] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, M. Taiji, 42 TFlops hierarchical *N*-body simulations on GPUs with applications in both astrophysics and turbulence, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM, 2009, pp. 1–12.
[21] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, K. Yasuoka, Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence, Computer Physics Communications 180 (11) (2009) 2066–2078.
[22] T.K. Sheel, R. Yokota, K. Yasuoka, S. Obi, The study of colliding vortex rings using a special-purpose computer and FMM, Transactions of the Japan Society for Computational Engineering and Science 3 (2008) 13.
[23] K. Xu, D.Z. Ding, Z.H. Fan, R.S. Chen, Multilevel fast multipole algorithm enhanced by GPU parallel technique for electromagnetic scattering problems, Microwave and Optical Technology Letters 52 (3) (Mar 2010) 502–507.
[24] M. Cwikla, J. Aronsson, V. Okhmatovski, Low-frequency MLFMA on graphics processors, IEEE Antennas and Wireless Propagation Letters 9 (2010) 8–11.
[25] T. Takahashi, C. Cecka, E. Darve, An implementation of low-frequency fast multipole BIEM for Helmholtz' equation on GPU, in: Proceedings of JSME 23rd Computational Mechanics Conference (CD-ROM), Kitami, Hokkaido, Japan, JSME, September 2010.