J Supercomput (2013) 63:897–918
DOI 10.1007/s11227-012-0836-0

Adaptive fast multipole methods on the GPU

Anders Goude · Stefan Engblom

Published online: 25 October 2012

Abstract We present a highly general implementation of fast multipole methods on graphics processing units (GPUs). Our two-dimensional double precision code features an asymmetric type of adaptive space discretization leading to a particularly elegant and flexible implementation. All steps of the multipole algorithm are efficiently performed on the GPU, including the initial phase, which assembles the topological information of the input data. Through careful timing experiments, we investigate the effects of the various peculiarities of the GPU architecture.

Keywords Adaptive fast multipole method · CUDA · Graphics processing units · Tesla C2075

1 Introduction

We discuss in this paper implementation and performance issues for adaptive fast multipole methods (FMMs). Our concerns are focused on using modern high-throughput graphics processing units (GPUs), which have seen an increased popularity in Scientific Computing in recent years. This is mainly thanks to their high peak floating point performance and memory bandwidth, implying a theoretical performance which is an order of magnitude better than for CPUs (or even more). However,

A. Goude
Division of Electricity, Department of Engineering Sciences, Uppsala University, 751 21 Uppsala, Sweden
e-mail: anders.goude@angstrom.uu.se

S. Engblom (✉)
Division of Scientific Computing, Department of Information Technology, Uppsala University, 751 05 Uppsala, Sweden
e-mail: stefane@it.uu.se
url: http://user.it.uu.se/~stefane

in practice for problems in Scientific Computing, the floating point peak performance can be difficult to realize since many such problems are bandwidth limited [24]. Although the GPU processor bandwidth is up to 4 times larger than that of the CPU, this is clearly not sufficient whenever the parallel speedup is (or could be) much larger than this. Moreover, according to the GPU computational model, the threads has to be run synchronously in a highly stringent manner. For these reasons, near optimal performance can generally only be expected for algorithms of predominantly data-parallel character.

Another difficulty with many algorithms in Scientific Computing is that the GPU off-chip bandwidth is comparably small such that the ability to mask this communication becomes very important [24]. Since the traditional form of many algorithms often involves intermediate steps for which the GPU architecture is suboptimal, a fair degree of rethinking is usually necessary to obtain an efficient implementation.

Fast multipole methods appeared first in [4, 9] and have remained important computational tools for evaluating pairwise interactions of the type

$$\Phi(x_i) = \sum_{j=1, j\neq i}^{N} G(x_i, x_j), \quad x_i \in \mathbf{R}^D, \ i = 1 \ldots N, \tag{1.1}$$

where $D \in \{2, 3\}$. More generally, one may consider to evaluate

$$\Phi(y_i) = \sum_{j=1, x_j\neq y_i}^{N} G(y_i, x_j), \quad i = 1 \ldots M, \tag{1.2}$$

where $\{y_i\}$ is a set of *evaluation points* and $\{x_j\}$ a set of *source points*. In this paper, we shall also conveniently use the terms *potentials* or simply *particles* to denote the set of sources $\{x_j\}$.

Although the direct evaluation of (1.1) has a complexity of $\mathcal{O}(N^2)$, the task is trivially parallelizable and can be performed much more efficiently using GPUs than CPUs. For sufficiently large N, however, tree-based codes in general and the FMM algorithm in particular become important alternatives. The practical complexity of FMMs scales linearly with the input data and, moreover, effective a priori error estimates are available. Parallel implementations are, however, often highly complicated and balancing efficiency with software complexity is not so straightforward [20, 23].

In this paper, we present a double precision GPU implementation of the FMM algorithm which is fully *adaptive*. Although adaptivity implies a more complex algorithm, this feature is critical in many important applications. Moreover, in our approach, all steps of the FMM algorithm are performed on the GPU, thereby reducing memory traffic to the host CPU.

Successful implementations of the FMM algorithm for GPUs have been reported previously [12, 26, 27] under certain limitations. Specifically, with GPUs the performance of single precision algorithms is a factor of *at least* 2 times better than double precision [16, p. 11]. In fact, for computationally intensive applications, this factor can reach as high as 8 times [25], which implies that single precision speedups vis-à-vis CPU implementations can well be > 2. It should be noted, however, that simpler tree-based methods than the FMM exist that offer a better performance at low tolerance (cf. [3], [10, Chap. 8.7]), and that the FMM is of interest mainly for higher accuracy demands.

In Sect. 2, we give an overview of our version of the adaptive FMM. The details of the GPU implementation are found in Sect. 3, and in a separate Sect. 4, we highlight the algorithmic changes that were made to the original serial code described in [7]. In Sect. 5, we examine in detail the speed-ups obtained when moving the various phases of the algorithm from the CPU to the GPU. We also reason about the results such that our findings may benefit others who try to port their codes to the GPU. Since the FMM has been judged to be one of the top 10 most important algorithms of the twentieth century [5], it is our hope that insights obtained here is of general value. A final concluding discussion around these matters is found in Sect. 6.

Availability of software The code discussed in the paper is publicly available and the performance experiments reported here can be repeated through the Matlab-scripts we distribute. Refer to Sect. 6.1 for details.

2 Well-separated sets and adaptive multipole algorithms

In a nutshell, the FMM algorithm is a tree-based algorithm which produces a continuous representation of the potential field (1.2) from all source points in a finite domain. Initially, all potentials are placed in a single enclosing box at the zeroth level in the tree. The boxes are then successively split into child-boxes such that the number of points per box decreases with each level.

The version of FMM that we consider is described in [7], and is organized around the requirement that boxes at the same level in the tree are either *decoupled* or *strongly/weakly coupled*. The type of coupling between the boxes follows from the *θ-criterion*, which states that for two boxes with radii r_1 and r_2, whose centers are separated with distance d, the boxes are *well-separated* whenever

$$R + \theta r \leq \theta d, \tag{2.1}$$

where $R = \max\{r_1, r_2\}$, $r = \min\{r_1, r_2\}$, and $\theta \in (0, 1)$ a parameter. In this paper, we use the constant value $\theta = 1/2$ which we have found to perform well in practice. At each level l and for each box b, the set $S(p)$ of strongly coupled boxes of its parent box p is examined; children of $S(p)$ that satisfy the θ-criterion with respect to b are allowed to become weakly coupled to b, otherwise they remain strongly coupled. Since a box is defined to be strongly connected to itself, this rule defines the connectivity for the whole multipole tree. In Fig. 1, an example of a multipole mesh and its associated connectivity pattern is displayed.

All boxes are equipped with an outgoing *multipole* expansion and an incoming *local* expansion. The multipole expansion is the expansion of all sources within the box around its center and is valid away from the box. To be concrete, in our two-dimensional implementation, we use the traditional format of a p-term expansion in

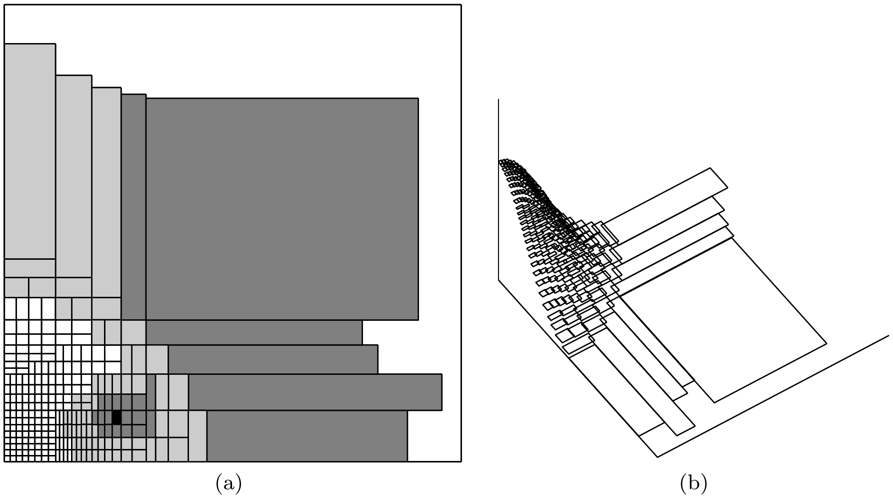


Fig. 1 The adaptive mesh is constructed by recursively splitting boxes along the coordinate axes in such a way that the number of source points is very nearly the same in the four resulting boxes. (**a**) Here, the boxes colored in *light gray* will interact via multipole-to-local shifts with the *black box*, that is, they satisfy the θ-criterion ($\theta = 1/2$). The *boxes in dark gray* are strongly connected to the black box and must be taken care of at the next level in the tree. (**b**) Same mesh as in (**a**), but visualized as a distribution by letting the height of each box be inversely proportional to its area. The source points in this example were sampled from a normal distribution

the complex plane,

$$M(z) = a_0 \log(z - z_0) + \sum_{j=1}^{p} \frac{a_j}{(z - z_0)^j}, \tag{2.2}$$

where z_0 is the center of the box. The local expansion is instead the expansion of sources far away from the box and can therefore be used to evaluate the contribution from these sources at all points within the box:

$$L(z) = \sum_{j=0}^{p} b_j (z - z_0)^j, \tag{2.3}$$

where again (2.3) is specific to a two-dimensional implementation.

The computational part of the FMM algorithm proceeds in an *upward* and a *downward* phase. During the first stage, the *multipole-to-multipole* (M2M) shift from children to parent boxes recursively propagates and accumulates multipole expansions. In the second stage, the important *multipole-to-local* (M2L) shift adds to the local expansions in all weakly coupled boxes which are then propagated downwards to children through the *local-to-local* shift (L2L). At the finest level, any remaining strong connections are evaluated through direct evaluation of (1.1) or (1.2). A simple optimization which was noted already in [4] is that, at the lowest level, strongly coupled boxes are checked with respect to the θ-criterion (2.1), but *with the roles of* *r and R interchanged*. If found to be true, then the potentials in the larger box can be directly shifted into a local expansion in the smaller box, and the outgoing multipole expansion from the smaller can directly be evaluated within the larger box.

The algorithm so far has been described without particular assumptions on the multipole mesh itself. As noted in [7], relying on *asymmetric adaptivity* when constructing the meshes makes a very convenient implementation possible. In particular, this construction avoids complicated cross-level communication and implies that the multipole tree is *balanced*, rendering the use of post-balancing algorithms [21] unnecessary. Also, the possibility to use a static layout of memory is particularly attractive when considering data-parallel implementations for which the benefits with adaptive meshes have been questioned [12].

In this scheme, the child boxes are created by successively splitting the parent boxes close to the median value of the particle positions, causing all child boxes to have about the same number of particles. At each level, all boxes are split twice in succession, thus producing four times as many boxes for each new level. The resulting FMM tree is a *pyramid* rather than a general tree for which the depth may vary. The cost is that with a balanced tree, the communication stencil is variable. Additionally, it also prevents the use of certain symmetries in the multipole translations as described in [14]. To improve on the communication locality, the direction of the split is guided by the eccentricity of the box since the algorithm gains in efficiency when the boxes have equal width and height (the θ-criterion is rotationally invariant).

The *algorithmic complexity* of the FMM has been discussed by many authors. Practical experiences [2], [10, Chap. 8.7], [11, Chap. 6.6.3], indicate that linear complexity in the number of source points is observed in most cases, but that simpler algorithms perform better in certain situations. Although it is possible to construct explicit distributions of points for which the FMM algorithm has a *quadratic complexity* [1], this behavior is usually not observed in practical applications.

With p terms used in both the multipole and the local expansions, we expect the serial computational complexity of our implementation to be proportional to $\theta^{-2} p^2 \cdot N$, with N the number of source points. This follows from assuming an asymptotically regular mesh such that $R \sim r$ in (2.1) and a total of on the order of N boxes at the finest level. Then each of those N boxes interact through M2L-interactions with about $\pi d^2 \times N$ other boxes. From (2.1), we get $d \sim (1+\theta)/\theta \times r \sim (\sqrt{N}\theta)^{-1}$, and since the M2L-interaction is a linear mapping between p coefficients this explains our stated estimate. This simple derivation assumes that the M2L-shift is the most expensive part of the algorithm. In practice, the cost of the direct evaluation of (1.1) may well match this part. From extensive experiments with the serial version of the algorithm, we have seen that it is usually possible to balance the algorithm in such a way that these two parts take roughly the same time.

With a given relative target tolerance TOL, the analysis in [7] implies $p \sim \log \mathrm{TOL} / \log \theta$, so that the total complexity can be expected to be on the order of $\theta^{-2} \log^{-2} \theta \cdot N \log^2 \mathrm{TOL}$. We now proceed to discuss an implementation which distributes this work very efficiently on a GPU architecture.

3 GPU implementation

The first major part of the adaptive fast multipole method is the *topological phase* which arranges the input into a hierarchical *FMM mesh* and determines the type of interactions to be performed. We discuss this part in Sect. 3.2. The second part is discussed in Sect. 3.3 and consists of the actual multipole evaluation algorithm with its upward and downward phases performing the required interactions in a systematic fashion.

To motivate some of our design choices, we choose to start with a brief discussion on the GPU hardware and the CUDA programming model *(Compute Unified Device Architecture)*. The interested reader is referred to [8, 17] for further details.

3.1 Overview of the GPU architecture

Using CUDA terminology, the GPU consists of several *multiprocessors* (14 for the Tesla C2075), where each multiprocessor contains many CUDA *cores* (32 for the C2075). The CUDA execution model groups 32 threads together into a *warp*. All threads in the same warp execute the same instruction, but operate on different data; this is simply the GPU-version of the *Single Instruction Multiple Data (SIMD)*-model for parallel execution. Whenever two threads within the same warp need to execute different instructions, the operations become serialized.

To perform a calculation on the GPU, the CPU launches a *kernel* containing the code for the actual calculation. The threads on the GPU are grouped together into *blocks*, where the number of blocks as well as the number of threads per block is to be specified at each kernel launch (for performance reasons the number of threads per block is usually a multiple of the warp size). All threads within a block will be executed on the same multiprocessor and thread synchronization can only be performed efficiently between the threads of a single block. This synchronization is required whenever two threads write to the same memory address. Although for this purpose there are certain built-in *atomic write* operations, none of these support double precision for the Tesla C2075.

The GPU typically has access to its own global memory and all data has to be copied to this memory before being used. Further, each multiprocessor has a special fast *shared memory*, which can be used for the communication between threads within the same block [8, Chap. 5]. For example, the C2075 has 48 kB of shared memory per multiprocessor for this purpose. Overuse of this memory limits the number of threads that can run simultaneously on a multiprocessor, which in turn has a negative effect on performance.

3.2 Topological phase

The topological phase consists of two parts, where the first part creates the boxes by partitioning the particles (we refer to this as *"sorting"*) and the second part determines the interactions between them (*"connecting"*).

The sorting algorithm successively partitions each box in two parts according to a chosen pivot point (Algorithm 3.1). The pivot element is obtained by first sorting

32 of the elements using a simple $\mathcal{O}(n^2)$ algorithm where each thread sorts a single point (32 elements was chosen to match the warp size). The pivot is then determined by interpolation of the current relative position in the active set of points so as to approximately land at the global median point (line 2, Algorithm 3.1).

Algorithm 3.1 (Partitioning with successive splits)

Input: Unordered array consisting of x- or y-coordinates.
Output: Array partitioned around its median coordinate.

```
1: while $size(array) > 32$ do
2:    $determine\_pivot\_32()$
3:    $split\_around\_pivot()$
4:    $keep\_part\_containing\_median()$
5: end while
6: $\{the\ array\ now\ consists\ of \leq 32 elements:\}$
7: $determine\_median\_32()$
```

The split in line 3 uses a two-pass scheme, where each thread handles a small set of points to split. The threads start by counting the number of points smaller than the pivot. Then a global cumulative sum has to be calculated. Within a block, the method described in [13] is used. For multiple blocks, *atomic addition* is used in between the blocks, thus allowing the split to be performed in a single kernel call (note that using atomic addition makes the code non-deterministic). Given the final cumulative sum, the threads can correctly insert their elements in the sorted array (second pass). After the split, only the part containing the median is kept for the next pass (line 4).

Algorithm 3.1 can be used to partition many boxes in a single kernel call using one block per box. If the number of boxes is low, it is desirable to use several blocks for each partitioning to better use the GPU cores. This requires communication between the blocks and the partitioning has to be performed with several kernel calls according to Algorithm 3.2. The splitting code is executed in a loop (lines 2 to 9) and a small amount of data transfer between the GPU and the CPU is required to determine the number of loops.

Algorithm 3.2 (Partitioning with successive splits, CPU part)

Input: Unordered array consisting of x- or y-coordinates.
Output: Array partitioned around its median coordinate.

```
1: $determine\_split\_direction()$
2: while $\max_i size(array_i) > single\_thread\_limit$ do
3:    $\{executed\ in\ parallel:\}$
4:    for all $splits$ do
5:       $determine\_pivot\_32()$
6:       $partition\_around\_pivot()$
7:       $keep\_part\_containing\_median()$
8:    end for
9: end while
```

```
10: split_on_single_block()
```

Running the code using multiple blocks forces the code to run synchronized, with equal amount of splits in each partitioning. If one bad pivot is encountered, then this split takes much longer time than the others resulting in bad parallel efficiency. By contrast, a *single* block running does not force this synchronization and, additionally, allows for a better caching of the elements since they remain in the same kernel call all the time. For these reasons, Algorithm 3.2 switches to single block mode (line 10) when all splits contain a small enough number of points ($single\_thread\_limit = 4096$ in the current implementation).

The second part of the topological phase determines the *connectivity* of the FMM mesh, that is, if the boxes should interact via near- or far-field interactions. This search is performed for each box independently and the parallelization one thread/box is used here. Each kernel call calculates the interactions of one full level of the FMM tree.

3.3 Computational part: the multipole algorithm

The computational part consists of all the multipole-related interactions, which include initialization (P2M), shift operators (M2M), (M2L), and (L2L), and local evaluation (L2P). Additionally, we also include the direct interaction in the near-field (P2P) in this floating point intensive phase of the FMM algorithm. During the computational part, no data transfer is necessary between the GPU and the host.

3.3.1 Multipole initialization

The initialization phase creates multipole expansions for each box via *particle-to-multipole* shifts (P2M). Since each source point gives a contribution to each coefficient a_j, using several threads per box requires intra-thread communication which in Algorithm 3.3 is accomplished by introducing a temporary matrix to store the coefficients.

Initially (line 5, Algorithm 3.3), one thread calculates the coefficients for one source particle (two threads if the number of particles is less than half the number of available threads). Then each thread calculates the sum for one coefficient (line 7). This procedure has to be repeated for a large number of coefficients, as it is desirable to have a small temporary matrix to limit the use of shared memory. The current implementation uses 64 threads per box (two warps) and takes 4 coefficients in each loop iteration (8 in the two threads/particle case).

The initialization also handles the special case where the particles are converted directly to local expansions via *particle-to-local* expansions (P2L). The principle for creating local expansions is the same as for the multipole expansions. All timings of this phase will include both P2M and P2L shifts.

Algorithm 3.3 (Multipole initialization)

Input: Positions and strengths for source particles in a box.

Output: Multipole coefficients a_i for the box.

```
{executed in parallel:}
for all sources in box do
    load_one_source_per_thread()
    for $k = 1$ to $p$ do
        $temp\_array := calc\_N\_cache\_coefficients()$
        synchronize_threads()
        $a_k := a_k + sum(temp\_array)$
    end for
end for
```

3.3.2 *Upward pass*

In the upward pass, the coefficients of each of four child boxes are repeatedly shifted to their parent's coefficients via the M2M-shift. This is achieved using Algorithm 3.4 (a), which is similar to the one proposed in [15]. Algorithm 3.4 (a) can be parallelized by allowing one thread to calculate one interaction, and at the end compute the sum over the four boxes (line 14, Algorithm 3.4). It should be noted that the multiplication in Algorithm 3.4 is a complex multiplication which is performed $\mathcal{O}(p^2)$ times. By introducing scaling, the algorithm can be modified to Algorithm 3.4 (b), which instead requires one complex division, $\mathcal{O}(p)$ complex multiplications and $\mathcal{O}(p^2)$ complex additions. The advantage of this modification in the GPU case is not the reduction of complex multiplications, but rather that the real and imaginary parts are independent. Hence, two threads per shift can be used thus reducing the amount of shared memory per thread.

Algorithm 3.4 (Multipole to multipole translation)

Input: Multipole coefficients a_j of child box at position z_c.
Output: Multipole coefficients a_j of parent box at position z_p.

```
(a) Without scaling                          (b) With scaling
 1: $r := z_c - z_p$                          $r := z_c - z_p$
 2:                                           for $j = 1$ to $p$ do
 3:                                               $a_j := a_j / r^j$
 4:                                           end for
 5: for $k = p$ downto 2 do                   for $k = p$ downto 2 do
 6:     for $j = k$ to $p$ do                     for $j = k$ to $p$ do
 7:         $a_j := a_j + r \cdot a_{j-1}$            $a_j := a_j + a_{j-1}$
 8:     end for                                   end for
 9: end for                                   end for
10: for $j = 1$ to $p$ do                     for $j = 1$ to $p$ do
11:     $a_j := a_j - r^j \cdot a_0 / j$          $a_j := (a_j - a_0/j) \cdot r^j$
12: end for                                   end for
13: {4 child boxes shift coefficients to
    the same parent:}
14: $a := sum\_translations()$                $a := sum\_translations()$
```

3.3.3 Downward pass

The downward pass consists of two parts, the translation of multipole expansions to local expansions (M2L), and the translation of local expansions to the children of a box (L2L). The translation of local expansions to the children is very similar to the M2M-shift discussed previously, and can be achieved with the scheme in Algorithm 3.5 (b). This shift is slightly simpler on the GPU since there is no need to sum the coefficients at the end, but instead requires more memory accesses as the calculated local coefficients must be added to already existing values.

Algorithm 3.5 (Local to local translation)

```
Input: Local coefficients $b_j$ of parent box at position $z_p$.
Output: Local coefficients $b_j$ of child box at position $z_c$.
$r := z_p - z_c$
for $j = 1$ to $p$ do
    $b_j := b_j \cdot r^j$
end for
for $k = 0$ to $p$ do
    for $j = p - k$ to $p - 1$ do
        $b_j := b_j - b_{j+1}$
    end for
end for
for $j = 1$ to $p$ do
    $b_j := b_j / r^j$
end for
```

The translation of multipole expansions to local expansions is the most time consuming part of the downward pass. The individual shifts can be performed with a combination of the reduction scheme in the M2M translation and the L2L translation; see Algorithm 3.6. Again, this implementation allows for two dedicated threads for each shift. We have not seen this algorithm described elsewhere.

Algorithm 3.6 (Multipole to local translation)

```
Input: Multipole coefficients $a_j$ of box at position $z_i$.
Output: Local coefficients $b_j$ of box at position $z_o$.
$r := z_o - z_i$
for $j = 1$ to $p$ do
    $b_{j-1} := a_j / r^j \cdot (-1)^j$
end for
$b_p := 0$
for $k = 2$ to $p$ do
    for $j = p - k$ to $p - 1$ do
        $b_j := b_j - b_{j+1}$
    end for
end for
for $k = p$ downto 1 do
    for $j = k$ to $p$ do
        $b_j := b_j + b_{j-1}$
    end for
end for
$b_0 := b_0 - a_0 \cdot \log(r)$
for $j = 1$ to $p$ do
    $b_j := (b_j - a_0/j)/r^j$
end for
$b := sum\_translations()$
```

With the chosen adaptive scheme, the number of shifts per box varies. Since our GPU does not support atomic addition in double precision, one block has to handle all shifts of one box in order to perform this operation in one kernel call. As the number of translations is not always a multiple of 16 (the number of translations per loop if 32 threads/block is used), this can result in idle threads. One can partially compensate for this by giving one block the ability to operate on two boxes in the same loop.

As the M2L translations are performed within a single level of the multipole tree, all translations can be performed in a single kernel call. This is in contrast to the M2M- and L2L translations, which both require one kernel call per level.

3.3.4 Local evaluation

The local evaluation (L2P) is scheduled in parallel by using one block to calculate the interactions of one box. Moreover, one thread calculates the interactions of one evaluation point from the local coefficients of the box. This operation requires no thread communication and can be performed in the same way as for the CPU. The local evaluation uses 64 threads/block.

This phase will also include the special case where the evaluation is performed directly through the multipole expansion (M2P). This operation is performed in a similar way as for the L2P evaluation. All timings of this phase therefore include both L2P- and M2P evaluations.

3.3.5 Near-field evaluation

In the near-field evaluation (P2P), the contribution F from all boxes within the near-field of a box should be calculated at all evaluation points of the box. Similar to the M2L translations, the number of boxes in the near-field varies due to the adaptivity.

Algorithm 3.7 (Direct evaluation between boxes)

Input: Positions and strengths of particles in the near field.
Output: The contribution F of the particles in the near field.

```
{executed in parallel:}
load_evaluation_point_positions() {one per thread}
```

```
3:  for all interaction boxes do
4:     $cache\_interaction\_positions()$
5:     if $cache\_is\_full$ or $all\_positions\_loaded$ then
6:        for all elements in cache do
7:           $F := F + add\_pairwise\_interaction()$
8:        end for
9:     end if
10: end for
```

The GPU implementation uses one block per box and one to four threads per evaluation point (depending on the number of evaluation points compared to the number of threads). The interaction is calculated according to Algorithm 3.7, where the source points are loaded into a cache in shared memory (line 4) and when the cache is full, the interactions are calculated (line 7). This part uses 64 threads per block and a suitable cache size is to use the same size as the number of threads.

4 Differences between the CPU and GPU implementations

In this section, we highlight the main differences between the two versions of the code. A point to note is that when we compare speed-up with respect to the CPU-code in Sect. 5, we have taken care in implementing several optimizations which are CPU-specific.

4.1 Topological phase

When sorting, the GPU implementation is based on sorting 32 element arrays for choosing a pivot element. This design was made to achieve a better use of the CUDA cores and, in the multiple block/partitioning case, to make all partitionings behave more similar to each other. The single-threaded CPU version uses the *median-of-three* procedure for choosing the pivot element, which is often used in the well-known algorithm *quicksort* [19, Chap. 9]. An advantage with this method compared to the GPU algorithm is that it is in-place and hence that there is no need for temporary storage.

4.2 Computational part

The direct evaluation can use symmetry on the CPU if the evaluation points are the same as the source points since the direct interaction is symmetric. Using this symmetry, the run time of the direct interaction can be reduced by almost a factor of two in the CPU version. This is not implemented on the GPU as it would require atomic memory operations to store the results (which is not available for double precision on our GPU).

The operations M2M, P2P, P2M, and M2P are all in principle the same in both versions of the code. For the M2L shift, the symmetry of the θ-criterion (2.1) can be used in the scaling phases on the CPU (lines 2–4 and 17–19 in Algorithm 3.6), while in the GPU version, the two shifts are handled by different blocks, making communication unpractical. Another difference is that the scaling vector is saved in memory from the prescaling part to the post-scaling part after the principal shift operator. This was intentionally omitted from the GPU version as the extra use of shared memory decreased the number of active blocks on each multiprocessor, and this in turn reduced performance.

4.3 Symmetry and connectivity

The CPU implementation uses symmetry throughout the multipole algorithm. With symmetry, it is only required to create one-directional interaction lists when determining the connectivity.

As the GPU implementation does not rely on symmetry when evaluating, it is beneficial to create directed interaction lists. This causes twice the work and twice the memory usage (for the connectivity lists). However, the time required to determine the connectivity is quite small ($\sim$1 %, see Table 1).

4.4 Further notes on the CPU implementation

The current CPU implementation is a single threaded version. Other research [6, 24] has shown that good parallel efficiency can be obtained for a multicore version (e.g., 85 % parallel efficiency on 64 processors). However, the work in [6] does not appear to use the symmetry in the compared direct evaluation.

In order to achieve a highly efficient CPU implementation, as suggested also by others [22, 27], the multipole evaluation part was written using SSE intrinsics. Using these constructs means that two double precision operations can be performed as one single operation. The direct- and multipole evaluation, as well as the multipole initialization all use this optimization, where two points are thus handled simultaneously. Additionally, all shift operators also rely on SSE intrinsics in their implementation.

4.5 Double versus single precision

The entire algorithm is written in double precision for both the CPU and the GPU. Using single precision would be significantly faster, both on the CPU (as SSE could operate on 4 floats instead of 2 doubles) and on the GPU (where the performance difference appears to vary depending on the mathematical expression [25]). It is likely that higher speedups could be obtained with a single precision code, but it would also seriously limit the applicability of the algorithm. With our approach, identical accuracy is obtained from the two codes.

5 Performance

This section compares the CPU and GPU codes performance-wise. The simulations were performed on a Linux Ubuntu 10.10-system with an Intel Xeon 6c W3680 3.33 GHz CPU and a Nvidia Tesla C2075 GPU. The compilers were GCC 4.4.5

and CUDA 4.0. For all comparisons of individual parts, the sorting was performed on the CPU to ensure identical multipole trees (the GPU sorting algorithm is nondeterministic). All timings have been handled using the timing functionality of the GPU. The total time includes the time to copy data between the host and the GPU, while the time of the individual parts does not include this. All simulations were performed a large number of times such that the error in the mean when regarded as an estimator to the expected value was negligible. Overall we found that the measured times displayed a surprisingly small spread with usually a standard deviation which was only some small fraction of the measured value itself.

All performance experiments were conducted using the *harmonic* potential,

$$G(z_i, z_j) \equiv \frac{\Gamma_j}{z_j - z_i}, \tag{5.1}$$

in (1.1), and hence $a_0 = 0$ in (2.2). Moreover, in Sects. 5.1 through 5.3, all simulations were performed using randomly chosen source points, homogeneously distributed in the unit square.

We remark again that the performance experiments reported here can be repeated using the scripts distributed with the code itself; see Sect. 6.1.

5.1 Calibration

From the perspective of performance, the most critical parameter is the number of levels in the multipole tree. Adding one extra level increases the number of boxes at the finest level in the tree by four. Assuming that each box connects to approximately the same number of boxes at each level, the total number of pairwise interactions therefore decrease with a factor of about 4. The initialization and multipole evaluation require the same amount of operations, but will operate on an increasing number of boxes, thus increasing the memory accesses. For all shift operations, one additional level implies about a threefold increase of the total number of interactions and the same applies for the determination of the connectivity information. For the sorting, each level requires about the same amount of work, but handling many small boxes easily causes additional overhead.

It is expected that the CPU code will follow this scaling quite well, while for the GPU, where several threads should run synchronously, this is certainly not always the case. As an example, L2P operates by letting one thread handle the interaction of one source point, P2M can use up to two threads, and P2P can use up to four threads/point (all these use 64 threads/block). On the tested Tesla C2075 system, this means that the local evaluation of a box containing 1 evaluation point takes the same amount of time as a box containing 64 evaluation points (on a Geforce GTX 480 system, this only applies to up to 32 evaluation points which is the warp size here). This shows the sensitivity of the GPU implementation with respect to the number of points in each box.

In Fig. 2, the GPU speedup as a function of the number of sources per box is studied. Within this range, the shift operators and the connectivity mainly depend on the number of levels and, therefore, obtain constant speedups (hence we omit them). All parts that depend on the number of particles in each box obtain higher speedups

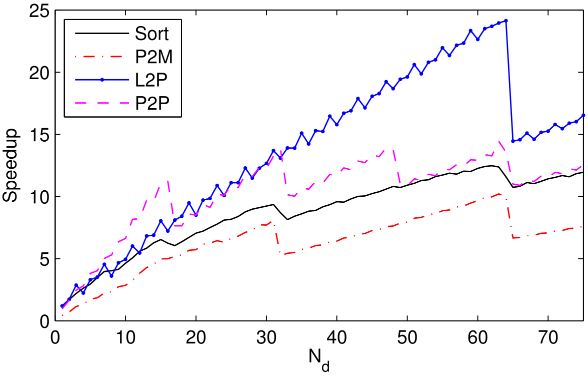


Fig. 2 Speedup of the individual parts of the GPU-implementation as a function of the number of sources per box N_d. Here the total number of sources varies between 2^{16} and 75×2^{16}

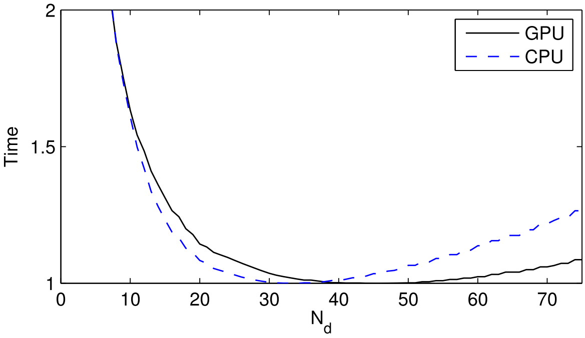


Fig. 3 Time as a function of the number of sources per box N_d for the CPU and the GPU implementation, both normalized so that the fastest time ≡ 1

for larger number of particles per box. This is expected, since it is easier to get a good GPU load for larger systems. There is also a performance decrease when the number of particles increases above 32, 64, and so on, that is, at multiples of the warp size. The direct evaluation additionally shows a performance decrease directly after each multiple of 16, which is due to the fact that the algorithm can use 4 threads per particle.

The small high frequency oscillations seen in the speedup of L2P and P2P originates from the CPU algorithm, and is due to the use of SSE instructions which makes the CPU code more efficient for an even number of sources per box.

It should be noted again that the direct evaluation and connectivity both make use of symmetry in the CPU version. This means that the speedup would be significantly higher (almost a factor 2) if the CPU version did not rely on this symmetry.

As adding one extra level reduces the computational time of the direct interaction, but increases the time requirement for most other parts, it is necessary to find the best range of particles per box. The number of levels N_l is calculated according to

$$N_l = \left\lceil 0.5 \log_2 \left(\frac{5}{8} \frac{N}{N_d} \right) \right\rceil, \tag{5.2}$$

where N is the number of particles and N_d is the desired number of particles per box. This parameter choice was studied for 150 simulations with different number of particles (from 1×10^4 to 2×10^6). The result (normalized against the lowest time on

Table 1 Time distribution of the GPU algorithm. The most expensive part in the case studied here is the direct evaluation (P2P), followed by sorting and M2L translations. The field "*other*" contains all data transfers between the host and the GPU

Part	Time	Part	Time
P2P	43 %	Connect	1 %
Sort	30 %	M2M	<1 %
M2L	11 %	L2L	<1 %
P2M	5 %		
L2P	2 %	Other	8 %

each platform) is found in Fig. 3, showing that a value around 45 is best for the GPU, while 35 is best for the CPU. Even though the GPU has poor speedup for low number of particles, it still scales better than the CPU in this case. The reason is that with low values of N_d, the multipole shifts dominate the computational time. This simulation was performed with 17 multipole coefficients, giving a tolerance of approximately 1×10^{-6}. The tolerance is here and below understood as

$$\mathrm{TOL} = \left\| \frac{\Phi_{FMM} - \Phi_{\mathrm{exact}}}{\Phi_{\mathrm{exact}}} \right\|_\infty \tag{5.3}$$

where Φ_{exact} is the exact potential and Φ_{FMM} is the FMM result.

For the optimal value 45 of N_d, the time distribution of the different parts of the algorithm is given in Table 1 for $N = 45 \times 2^{16}$, which gives 45 sources in each box at the finest level of the FMM tree. According to (5.2), using $N_d = 45$ gives 8 levels for $N \in (18 \times 2^{16}, 72 \times 2^{16}]$. Within this interval, the time of P2P relative to the total time varies between 25 % to 55 %. It is particularly interesting to note in Table 1 that the sorting dominates by a factor of about 3 over the usually very demanding M2L-operation.

5.2 Shift operators

The performance of the sorting and direct evaluation depends on the number of sources per box and the number of levels while the connectivity to a first approximation only depends on the number of levels. The rest of the operators also depend on the number of multipole coefficients (the number of multipole coefficients determine the error in the algorithm). Multipole initialization and evaluation depends linearly on the number of coefficients, while the shift operators have two linear parts (pre and postscaling) and one quadratic part (the actual shift). In the GPU case, all accesses to global memory are included in the linear parts while all data is kept in shared memory during the shift. A higher number of coefficients increases the use of shared memory and fewer shift operations can therefore be performed in parallel. The speedup as a function of number of coefficients is plotted in Fig. 4, where the simulation was performed on 10^6 particles with $N_d = 45$. The decrease in speedup due to lack of shared memory can be seen quite clearly, e.g. at 42 coefficients for the M2L-shift, where one block less (3 in total) can operate on the same multiprocessor.

The difference in speedup for L2P at low number of coefficients is likely due to overhead, since these values stabilize at high enough number of coefficients.

Fig. 4 Speedup as a function of the number of multipole coefficients p

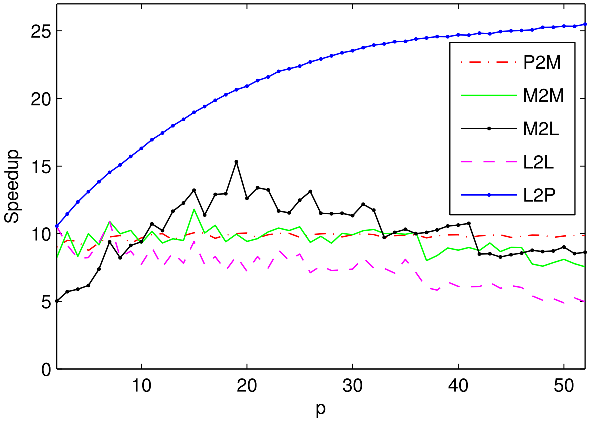
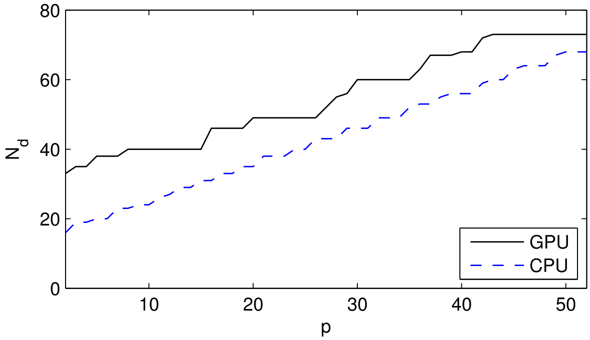


Fig. 5 Optimal value of N_d as a function of the number of multipole coefficients p



Considering that the time required for the shift operators increases with increasing number of coefficients, the optimal value for N_d changes as well. Figure 5 shows that the optimal number for N_d increases approximately linearly with increasing number of coefficients.

5.3 Break-even

If the number of sources is low enough, it may be faster to use a direct summation instead of the fast multipole method. In Fig. 6, the speed of the entire algorithm is compared with the speed of direct summation for both the CPU and the GPU implementation. The speedup of the GPU code increases with the number of particles since more source points provide a better load for the GPU. Looking at the direct summation times, the GPU scales linearly in the beginning where the number of working cores is still increasing, and later scales quadratically as the cores become fully occupied. Since a double sum is easily performed in parallel and is not memory bandwidth dominated the direct evaluation provides a good estimate of the maximum speedup that can be achieved with the GPU. Recall, however, that symmetry is used in the CPU implementation, which almost speeds up the calculation with a factor of 2. Figure 6 shows that it is more beneficial to use the FMM algorithm if the number of points exceeds about 3500 on the GPU. This result compares favorably with that

Fig. 6 Total time of the algorithm as a function of the number of sources N. For the FMM-algorithm, the simulation was performed with $p = 17$, implying a tolerance of about 10^{-6}

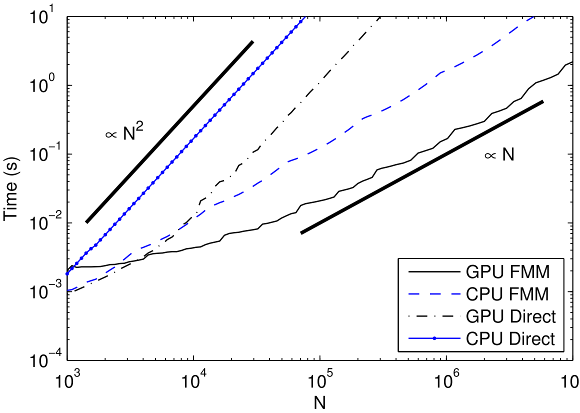
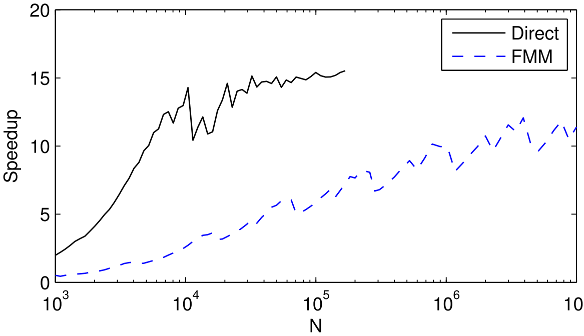


Fig. 7 Speedup as a function of the number of sources N



reported by others [26]. For large N, the speedup of the direct interaction is higher than that of the FMM (15 compared to about 11, see Fig. 7). Again, one should note that the CPU version uses symmetry here. For simulations where the source points and evaluation points are separate, the speedup is about 30 for the direct evaluation and 15 for the FMM. The lower increase in speedup for the FMM is due to the fact that only the P2P-evaluation of the algorithm uses this symmetry (compare Table 1).

Comparing the individual parts (Fig. 8), the M2L- and P2P-shifts quite rapidly obtain high speedups, while the sorting requires quite a large number of points. The poor values for M2M and L2L at low number of particles are due to the fact that few shifts are performed at the lower levels, causing many idle GPU cores. The situation is the same for the connectivity. As these algorithms have to be performed one level at a time, the low performance of the shifts high up in the multipole tree decreases the performance of the entire step. Consequently, the speedup increases with an increasing number of source points. The oscillating behavior of the multipole initialization and evaluation is related to the number of particles in each box (compare with Fig. 2).

The code has been tested both on the Tesla platform used for the above figures, and on a Geforce GTX 480 platform (which has 480 cores, compared to 448 for the Tesla card). The total run-time is approximately the same on both platforms. Notable differences are that the P2P interaction is faster on the Tesla if N_d is high, and in the simulation in Table 1, the GTX 480 card required 30 % longer time than the Tesla

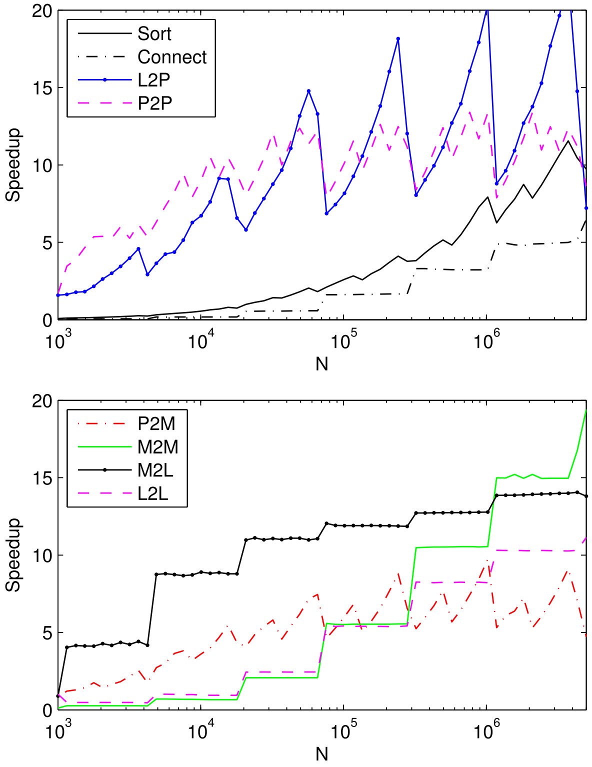


Fig. 8 Speedup of individual parts as a function of the number of sources

card. On the other hand, the Tesla card required 25 % longer time than the GTX 480 for the sorting (which is limited by memory access, rather than double precision math). The shift operators were approximately equally fast on both systems. The overall result is that the optimal value for N_d is lower for the GTX 480 card (35 instead of 45) for a total running time which was approximately the same. This shows that the much cheaper GTX 480 gaming card is a perfectly reasonable option for this implementation of the fast multipole method, despite the fact that it is written in double precision.

5.4 Benefits of adaptivity

As a final computational experiment, we investigated the performance of the adaptivity by using different point distributions. Under a relative tolerance of 10^{-6} ($p = 17$ in (2.2) and (2.3)), we measured the evaluation time for increasing number of points sampled from three very different distributions. As shown in Fig. 9, the code is robust under highly nonuniform inputs and scales well at least up to some 10 million source points.

When the distribution of particles is increasingly nonuniform, more boxes will be in each others near-field resulting in more direct interactions. This is tested in Fig. 10 for the two nonuniform distributions from Fig. 9. Both the CPU and GPU timings have been normalized to the time of a homogeneous distribution. The results indicate

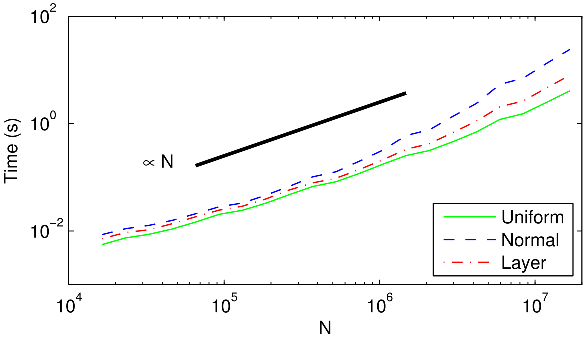


Fig. 9 Performance of the code when evaluating the harmonic potential for three different distributions of points. The source points were (i) uniformly distributed in [0, 1] × [0, 1], (ii) normally distributed with variance $1/100$, and (iii) distributed in a 'layer' where the x-coordinate is uniform, and the y-coordinate is again $N(0, 1/100)$-distributed. For the purpose of comparison, all distributions were rejected to fit exactly within the unit square. The FMM mesh for case (ii) is shown in Fig. 1

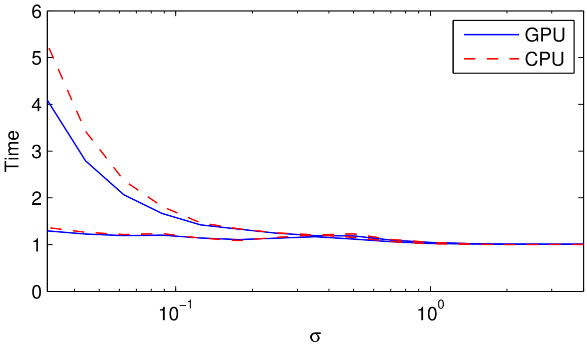


Fig. 10 Robustness of adaptivity. Time for two different nonuniform distributions normalized with respect to a uniform distribution of points. The *top two graphs* are for the normal distribution of sources, while the *lower two graphs* are for the "layer" distribution. See text for further details

that the decrease in performance for highly nonuniform distributions is less for the GPU version than for the CPU version. From the timings of the individual parts, it is seen that almost all the increase in computational time originates from the P2P-shift. The speedup for all time consuming individual parts is relatively constant with respect to the degree of nonuniformity (e.g. the parameter σ in Fig. 10) and the reason the GPU code handles a highly nonuniform distribution better is simply because the P2P evaluation has a higher speedup than the overall code.

6 Conclusions

We have demonstrated that all parts of the adaptive fast multipole algorithm can be efficiently implemented on a GPU platform in double precision. Overall, we obtain a speedup of about a factor of 11 compared to a highly optimized (including SSE intrinsics), albeit serial, CPU-implementation. This factor can be compared with the speedup of about 15, which we obtain obtained for the direct N-body evaluation, a task for which GPUs are generally understood to be well suited [18] (see Fig. 7).

In our implementation, all parts of the algorithm achieve speedups in about the same order of magnitude. Generally, we obtain a higher speedup whenever the boxes contain some 20–25 % more particles than the CPU version (see Figs. 5 and 8). Given the data-parallel signature of this particular operation, this result is quite expected. Another noteworthy result is that our version of the GPU FMM is faster than the direct N-body evaluation at around $N = 3500$ source points; see Fig. 6. Our tests also show that the asymmetric adaptivity works at least as well on the GPU as on the CPU, and that in some cases it even performs better.

When it comes to *coding complexity* it is not so straightforward to present actual figures, but some observations at least deserve to be mentioned. The topological phase was by far the most difficult part to implement on the GPU. In fact, the number of lines of codes approximately *quadrupled* when transferring this operation. We remark also that the topological phase performs rather well on the GPU, an observation which can be attributed to its comparably high internal bandwidth. Thus, there is a performance/software complexity issue here, and striking the right balance is not easy.

By contrast, the easiest part to transfer was the direct evaluation (P2P), where, due to SSE-intrinsics, the CPU-code is in fact about twice the size than the corresponding CUDA-implementation.

These observations as well as our experimental results, suggest that a *balanced* implementation, where parts of the algorithm are off-loaded to the GPU while the remaining parts are parallelized over the CPU-cores, would be a reasonable compromise. This has also been noted by others [3] and is ongoing research.

6.1 Reproducibility

Our implementation as described in this paper is available for download via the second author's web-page.[1] The code compiles both in a serial CPU-version and in a GPU-specific version and comes with a convenient Matlab mex-interface. Along with the code, automatic Matlab-scripts that repeat the numerical experiments presented here are also distributed.

Acknowledgement We like to acknowledge inputs from the UPMARC@TDB research group and Olov Ågren.

This work was financially supported by the Swedish Energy Agency, Statkraft AS, Vinnova and Uppsala University within the Swedish Centre for Renewable Electric Energy Conversion (A. Goude), and the Swedish Research Council within the UPMARC Linnaeus Center of Excellence (S. Engblom).

References

1. Aluru S (1996) Greengard's N-body algorithm is not order N. SIAM J Sci Comput 17(3):773–776. doi:10.1137/S1064827593272031

[1] http://user.it.uu.se/~stefane/freeware.

2. Blelloch G, Narlikar G (1997) A practical comparison of N-body algorithms. In: Parallel algorithms. Series in discrete mathematics and theoretical computer science, vol 30
3. Burtscher M, Pingali K (2011) An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In: Hwu W-M (ed) GPU computing Gems Emerald edition, vol 6. Morgan Kaufmann/Elsevier, San Mateo, pp 75–92
4. Carrier J, Greengard L, Rokhlin V (1988) A fast adaptive multipole algorithm for particle simulations. SIAM J Sci Stat Comput 9(4):669–686. doi:10.1137/0909044
5. Cipra BA (2000) The best of the 20th century: editors name top 10 algorithms. SIAM News 33(4)
6. Cruz FA, Knepley MG, Barba LA (2011) Petfmm–a dynamically load-balancing parallel fast multipole library. Int J Numer Methods Eng 85(4):403–428. doi:10.1002/nme.2972
7. Engblom S (2011) On well-separated sets and fast multipole methods. Appl Numer Math 61(10):1096–1102. doi:10.1016/j.apnum.2011.06.011
8. Farber R (2011) CUDA application design and development, 1st edn. Morgan Kaufmann, San Francisco
9. Greengard L, Rokhlin V (1987) A fast algorithm for particle simulations. J Comput Phys 73(2):325–348. doi:10.1016/0021-9991(87)90140-9
10. Griebel M, Knapek S, Zumbusch G (2007) Numerical simulation in molecular dynamics. Texts in computational science and engineering, vol 5. Springer, Berlin
11. Gumerov NA, Duraiswami R (2004) Fast multipole methods for the Helmholtz equation in three dimensions. Elsevier series in electromagnetism. Elsevier, Oxford
12. Gumerov NA, Duraiswami R (2008) Fast multipole methods on graphics processors. J Comput Phys 227(18):8290–8313. doi:10.1016/j.jcp.2008.05.023
13. Harris M, Sengupta S, Owens JD (2007) Parallel prefix sum (scan) with CUDA. In: Nguyen H (ed) GPU gems 3. Addison Wesley, Reading, pp 851–876. chapter 39
14. Hrycak T, Rokhlin V (1998) An improved fast multipole algorithm for potential fields. SIAM J Sci Comput 19(6):1804–1826. doi:10.1137/S106482759630989X
15. Liu Y (2009) Fast multipole boundary element method: theory and applications in engineering. Cambridge University Press, Cambridge
16. NVIDIA's next generation CUDA compute architecture: Fermi. NVIDIA (2009) Available at http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
17. CUDA C programming guide. NVIDIA (2012). Version 4.2. Available at http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDACProgrammingGuide.pdf
18. Rapaport DC (2011) Enhanced molecular dynamics performance with a programmable graphics processor. Comput Phys Commun 182(4):926–934. doi:10.1016/j.cpc.2010.12.029
19. Sedgewick R (1990) Algorithms in C. Addison-Wesley series in computer science. Addison-Wesley, Reading
20. Shanker B, Huang H (2007) Accelerated Cartesian expansions—a fast method for computing of potentials of the form $R^{-\nu}$ for all real ν. J Comput Phys 226(1):732–753. doi:10.1016/j.jcp.2007.04.033
21. Sundar H, Sampath RS, Biros G (2008) Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. SIAM J Sci Comput 30(5):2675–2708. doi:10.1137/070681727
22. Tanikawa A, Yoshikawa K, Okamoto T, Nitadori K (2012) N-body simulation for self-gravitating collisional systems with a new SIMD instruction set extension to the x86 architecture, advanced vector extensions. New Astron 17(2):82–92. doi:10.1016/j.newast.2011.07.001
23. Vikram M, Baczewzki A, Shanker B, Aluru S (2009) Parallel accelerated Cartesian expansions for particle dynamics simulations. In: Proceedings of the 2009 IEEE international parallel and distributed processing symposium, pp 1–11. doi:10.1109/IPDPS.2009.5161038
24. Vuduc R, Chandramowlishwaran A, Choi J, Guney M, Shringarpure A (2010) On the limits of GPU acceleration. In: Proceedings of the 2nd USENIX conference on hot topics in parallelism, HotPar'10. Berkeley, CA, USA, USENIX Association
25. Wong H, Papadopoulou M-M, Sadooghi-Alvandi M, Moshovos A (2010) Demystifying GPU microarchitecture through microbenchmarking. In: 2010 IEEE international symposium on performance analysis of systems software (ISPASS), pp 235–246. doi:10.1109/ISPASS.2010.5452013
26. Yokota R, Barba L (2011) Treecode and fast multipole method for N-body simulation with CUDA. In: Hwu W-M (ed) GPU computing Gems Emerald edition Morgan Kaufmann/Elsevier, San Mateo, pp 113–132
27. Yokota R, Barba L (2012) A tuned and scalable Fast Multipole Method as a preeminent algorithm for exascale systems. Int J High Perform Comput Appl. doi:10.1177/1094342011429952