Noname manuscript No.
(will be inserted by the editor)

Adaptive fast multipole methods on the GPU

Anders Goude · Stefan Engblom

May 21, 2012

Abstract We present a highly general implementation of fast multipole methods on graphical processor 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 with the GPU architecture.

Mathematics Subject Classification (2000) 65Y05 · 65Y20.

Keywords adaptive fast multipole method · CUDA · graphical processor 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 graphical processor units (GPUs) which have seen an increased popularity in Scientific Computing in later years. This is mainly thanks to their high peak floating point performance and memory bandwidth,

Corresponding author: S. Engblom, telephone +46-18-471 27 54, fax +46-18-51 19 25.

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

S. Engblom
Division of Scientific Computing, Department of Information Technology, Uppsala University, SE-751 05 Uppsala, Sweden.
E-mail: stefane@it.uu.se, *URL:* http://user.it.uu.se/~stefane.

implying a theoretical performance which is an order of magnitude faster than for CPUs (or even more). However, in practice for problems in Scientific Computing, the floating point peak performance can be difficult to realize since many such problems are bandwidth limited. 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, in the GPU computational model, each *warp*, consisting of many *threads* (typically 32), has to be run in a totally synchronous fashion. For these reasons, near optimal performance can generally only be expected for algorithms of predominantly data-parallel character.

Another difficulty with many algorithms from Scientific Computing is that the GPU off-chip bandwidth is comparably small such that the ability to mask this communication becomes very important. Since the traditional form of many algorithms typically involves intermediate steps for which the GPU architecture is sub-optimal, a fair degree of rethinking is usually necessary in order to obtain good speed-up.

Fast multipole methods appeared first in [3, 7] and have remained important computational tools for evaluating pairwise interactions of the type

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

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 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 $\mathcal{O}\left(N^2\right)$, the task is trivially parallelizable and can be performed using GPUs much more efficiently than on CPUs. For sufficiently large N, however, tree-based codes in general and the FMM algorithm in particular becomes 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 [16, 18].

Successful implementations of the FMM algorithm to GPUs have been reported previously [10, 20] under certain limitations. Specifically, with GPUs the performance of single precision algorithms is a factor of *at least* 2 times better than double precision [14, p. 11]. For computationally intensive applications, however, this factor can reach as high as 8 times [19], 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 [8, Chap. 8.7], and that the FMM is of interest mainly for higher accuracy demands. Also, our implementation is fully *adaptive*. Removing this feature simplifies implementation issues considerably at the cost of a much higher computational complexity for many important applications.

In Section 2 we give an overview of our version of the adaptive FMM. The details of the GPU implementation are found in Section 3 and in a separate Section 4 we highlight the algorithmic changes that were made to the original serial code described in [6]. In Section 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 20th century [4], it is our hope that insights obtained here is of general value. A final concluding discussion around these matters is found in Section 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 Section 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.

All boxes are equipped with an outgoing *multipole* expansion and an ingoing *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 an expansion in the complex plane,

$$M(z) = a_0 \log(z - z0) + \sum_{j=1}^{p} \frac{a_j}{(z - z_0)^j}, \tag{2.1}$$

with z_0 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:

$$L(z) = \sum_{j=0}^{p} b_j (z - z_0)^j, \tag{2.2}$$

where again (2.2) is specific to a two-dimensional implementation.

The version of FMM that we consider and as described in [6] 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

$$R + \theta r \leq \theta d, \tag{2.3}$$

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 Figure 2.1 an example of a multipole mesh and its associated connectivity pattern are displayed.

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 is 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 [3] is that, at the lowest level, strongly coupled boxes are checked with respect to the θ-criterion (2.3), 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 [6], 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 [17] 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 [10].

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. Hence the 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 [12].

Accordingly, at each level, all boxes are split twice in succession, thus producing four times as many boxes for each new level. The smallest bounding

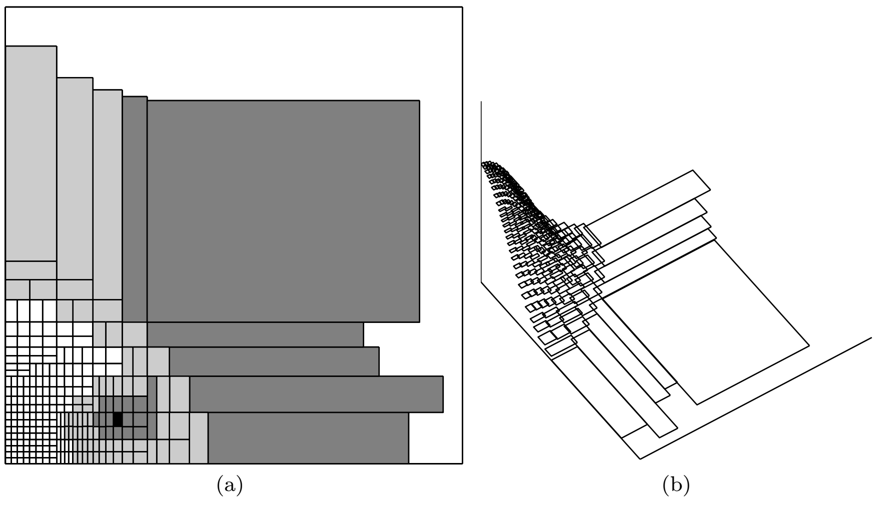

Fig. 2.1 The adaptive mesh is constructed by recursively splitting boxes along the coordinate axes in such a way that the number of source points is very nearly the same in the four resulting boxes. *(a)* Here the boxes colored in light gray will interact via multipole-to-local shifts with the black 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.

box of each box is determined (box shrinking), which later is used with the θ-criterion. Reducing the box size may allow M2L interactions to occur on a higher level in the tree, and hence may help reducing the total number of interactions. Also, 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], [8, Chap. 8.7], [9, 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 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 (2.3) and a total of on the order of N boxes at the finest level. Then we get that each of those N boxes will interact through M2L-interactions with about $\pi d^2 \times N$ other boxes. From (2.3) 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. For a sufficiently non-regular mesh, the cost of the direct evaluation of (1.1) may well match this part. In practice, from lots of experiments, 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 [6] 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

In general, when implementing the adaptive fast multipole method, the first major part 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 Section 3.1. The second part is to be discussed in Section 3.2 and consists of the actual multipole evaluation algorithm with its upward and downward phases performing the required interactions in a systematic fashion.

3.1 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 is based on successively partitioning each box in two parts according to a chosen pivot point (see Algorithm 3.1). The pivot element is obtained by first sorting 32 of the elements that should be split. 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. The specific choice of 32 elements was made to match the warp size of the GPU and the explicit sorting is done with a simple $\mathcal{O}\left(n^2\right)$ algorithm where each thread sorts a single point.

Algorithm 3.1 (Partitioning with successive splits)

```
while size(array) > 32 do
    determine_pivot_32()
    partition_around_pivot()
    keep_part_containing_median()
end while
{the array now consists of ≤ 32 elements:}
determine_median_32()
```

The split itself is performed using a two-pass scheme, where all threads are given a certain small amount of points to split. The threads start by calculating how many of the points that are smaller than the pivot. When all threads have this information, the cumulative sum has to be calculated. Within a block, this can be calculated with the method described in [11], while if multiple blocks are involved, *atomic addition* is used between the blocks, which makes it possible to perform the split in a single kernel call (note that using atomic addition makes the code non-deterministic). When the cumulative sum has been calculated, all threads can determine where to insert their elements on the two sides of the split, and this is performed in the second pass.

If many boxes are to be partitioned, each partitioning can be assigned to a single block and performed with a single kernel call using Algorithm 3.1. If few partitionings are to be performed, 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 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)

```
determine_split_direction()
while $\max_i size(array_i) > single\_thread\_limit$ do
  {executed in parallel:}
  for all splits do
    determine_pivot_32()
    partition_around_pivot()
    keep_part_containing_median()
  end for
end while
split_on_single_block()
```

It should be noted that running the code using multiple blocks forces the code to run synchronized, with equal amount of splits in each partitioning. If the pivot is chosen poorly for one partitioning, this split takes much longer time than the others resulting in bad parallel efficiency. By contrast, running on a single block 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, the partitioning code switches to single block mode when all splits contain a small enough number of points (4096 in the current implementation).

The second part of the topological phase is to determine 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 of the other, making parallelization easy. In the current implementation, the CPU is responsible for allocating an array of the proper size for each level in the tree, and to determine the cumulative sum indicating the starting position for each box in the interaction list. The rest of the procedure is handled by the GPU. Considering that a single search for a box is a quite fast operation, no attempt has been made to write a version that uses multiple threads for a single box, and the trivial parallelization one thread/box is used.

3.2 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.

3.2.1 Multipole initialization

In the initialization phase multipole expansions should be created for each box via *particle-to-multipole* shifts (P2M). Considering that each source particle gives a contribution to each coefficient a_k in the multipole expansion, using several threads for one box will require communication between the threads. The current parallel solution, described in Algorithm 3.3, solves this by introducing a temporary matrix to store coefficients. First, one thread calculates the coefficients for one source particle (two threads in the case the number of particles is less than half the number of available threads). Then each thread calculates the sum for one coefficient, which also can be done in parallel. It should be noted that this procedure has to be iterated for a large number of coefficients, as it is desirable to have as small temporary matrix as possible 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)

```
{executed in parallel:}
for all sources in box do
    load_one_source_per_thread()
    for $k = 1$ to $p$ do
        temp_array := calc_Ncache_coefficients()
        synchronize_threads()
        $a_k$ := $a_k$ + sum_over_TMParray()
    end for
end for
```

3.2.2 Upward pass

The upward pass consists of shifting the coefficients of four child boxes to its parent via the M2M-shift, and is performed 'upwards' at each level in the tree. This is achieved using the recursive Algorithm 3.4, which is similar to the code proposed in [13]. Algorithm 3.4 can be parallelized by allowing one thread to calculate one interaction, and at the end compute the sum over the four boxes. It should be noted that the multiplication in Algorithm 3.4 is a complex multiplication which is performed $\mathcal{O}\left(p^2\right)$ times. By introducing scaling, the algorithm can be modified to Algorithm 3.5, which instead requires one complex division, $\mathcal{O}\left(p\right)$ complex multiplications and $\mathcal{O}\left(p^2\right)$ 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, allowing for two threads per each shift (compared to one in Algorithm 3.4), thus reducing the amount of shared memory used per thread.

Algorithm 3.4 (Multipole to multipole translation)

```
$r$ := *distance_between_centers*
**for** $k = p$ **downto** *2* **do**
  **for** $j = k$ **to** $p$ **do**
    $a_j := a_j + r \cdot a_{j-1}$
  **end for**
**end for**
**for** $j = 1$ **to** $p$ **do**
  $a_j := a_j - r^j \cdot a_0/j$
**end for**
$a$ := *sum_translations()*
```

Algorithm 3.5 (Multipole to multipole translation (with scaling))

```
$r$ := *distance_between_centers*
**for** $j = 1$ **to** $p$ **do**
  $a_j := a_j/r^j$
**end for**
**for** $k = p$ **downto** *2* **do**
  **for** $j = k$ **to** $p$ **do**
    $a_j := a_j + a_{j-1}$
  **end for**
**end for**
**for** $j = 1$ **to** $p$ **do**
  $a_j := (a_j - a_0/j) \cdot r^j$
**end for**
$a$ := *sum_translations()*
```

3.2.3 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 3.6. This shift is slightly simpler on the GPU since there is no need to sum the coefficients at the end, but on the other hand it requires more memory accesses as the calculated local coefficients must be added to already existing values. Similar to the M2M-shift, scaling is introduced here as well.

Algorithm 3.6 (Local to local translation)

```
$r := distance\_between\_centers$
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 $p$ do
    $b_j := b_j - r \cdot b_{j+1}$
  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, according to Algorithm 3.7. Again, this implementation allows for two dedicated threads for each shift. We have not seen this algorithm described elsewhere.

Algorithm 3.7 (Multipole to local translation)

```
$r := distance\_between\_centers$
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 $p$ 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 for each box varies between the boxes. Since our GPU does not support atomic addition in double precision, to perform this operation in one kernel call, one block has to handle all shifts of one box. 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 give idle threads. One can partly 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.

3.2.4 Local evaluation

The local evaluation (L2P) is parallelized by letting one block handle the interactions of one box. Moreover, one thread handles the interactions of one evaluation point, which interacts through the local coefficients of the particular box. As one thread can handle one point, only one thread needs to access one memory location, and the calculation for one point can be performed 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 through the multipole expansion instead. This operation is performed in a similar way as for the evaluation from local coefficients. All timings of this phase includes both local evaluation and evaluation through the multipole coefficients.

3.2.5 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 asymmetric adaptivity.

Algorithm 3.8 (Direct evaluation between boxes)

```
  {executed in parallel:}
  load_evaluation_point_positions()
  for all interaction boxes do
    cache_interaction_positions()
    if cache_is_full or all_positions_loaded then
      for all elements in cache do
        $F := F +$ add_pairwise_interaction()
      end for
    end if
  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 3.8, where the source points are loaded into a cache in shared memory and when the cache is full, the interactions are calculated. 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 Section 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 in order to obtain better usage of the available 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* [15, Chap. 9]. An advantage with this method is that it is in-place and hence that there is no need for temporary storage. The GPU implementation is rather a two-pass method where the first pass counts the elements and the second pass sorts them properly into a new array.

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 mutual. Using this symmetry, the run time of the direct interaction can almost be reduced by a factor two on the CPU version. This symmetry 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 upward and downward shifts are in principle the same in both versions of the code. The same applies for the initialization and the multipole evaluation. The difference comes with the M2L translation. This operator is symmetric as a result of the symmetricity of the θ-criterion (2.3) and this symmetry can be used in the scaling phases of the CPU algorithm, 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 pre-scaling 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 did decrease 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) in the GPU implementation. However, the time required to determine the connectivity is quite small ($\sim 1\%$, see Figure 5.3).

4.4 Further notes on the CPU implementation

The current CPU implementation is a single threaded version. Other research [5] 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 [5] does not appear to use the symmetry in the compared direct evaluation.

To create a highly efficient CPU implementation, the multipole evaluation part have been written using SSE intrinsics, where 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 at once. Additionally, all shift operators also rely on SSE intrinsics in their implementation. No SSE optimizations has been tried for the topological phase of the code, as SSE is clearly more suited for the floating point intense parts of the algorithm.

4.5 Double versus single precision

The entire algorithm is written in double precision for both the CPU and the GPU. Using a single precision code would probably be a lot 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 [19]). It is likely that higher speedups could be obtained with a single precision code, but it would also seriously limit the usability of the algorithm. With our approach, identical or very nearly 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 3.33GHz CPU and a Nvidia Tesla C2075 GPU. The compilers were GCC 4.4.5 and CUDA 4.0. For all comparisons of individual parts (except sorting), all sorting was performed on the CPU to make sure that both algorithms operated on the same multipole tree (the GPU sorting algorithm is non-deterministic). All timings are done using the timing function 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, unless the copy is performed within that part (which occurs for sorting and connectivity).

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 (2.1). Moreover, in Sections 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 Section 6.1.

5.1 Calibration

From the perspective of performance, the most critical parameter is the number of levels in the multipole tree. Having more levels will give less sources in each box which will result in less direct evaluations, but also in more multipole shifts. Since each box is split into four children, assuming that each box connects to approximately the same number of boxes at each level, this implies that by increasing the number of levels with one the total number of pairwise interactions should decrease with a factor of about 1/4. The initialization and multipole evaluation should require the same amount of operations, but will operate in an increasing number of boxes, thus increasing the memory accesses. For all shift operations, one additional level implies an increase of about 4 times for the total number of interactions. The same applies for determining the connectivity information. For the sorting itself, each level should require approximately the same amount of work, but handling many small boxes can easily cause 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. An an example, L2P operate 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 making the local evaluation of a box containing 1 evaluation point takes the same amount of time as a

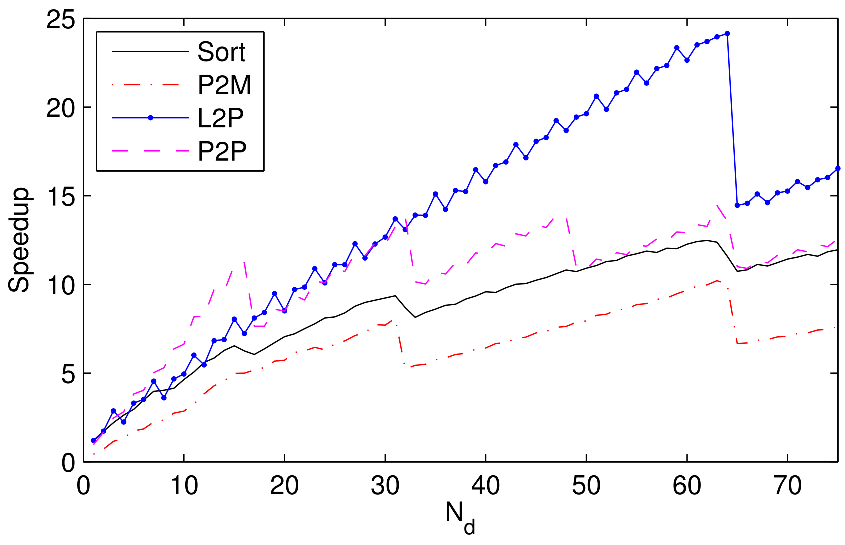

Fig. 5.1 Speedup for the GPU-implementation of individual parts of the algorithm as a function of number of sources per box N_d. Here the total number of points increased from 2^{16} to 75×2^{16}.

box containing 64 evaluation points (on a Geforce GTX 480 system, this only applied 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 Figure 5.1, the dependence of the number of sources per box has been studied by letting the number of levels be constant and increasing the number of particles. Within this test, the shift operators and the connectivity mainly depend on the number of levels and they therefore obtain constant speedups (hence we omit them in the figure). All parts that depend on the number of particles in each box obtains higher speedups for large number of particles in each box. This is expected, as it is easier to get a good GPU load for larger systems. It can also clearly be seen that there is 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 which makes the CPU code more efficient for an even number of sources per box.

When it comes to the actual speed improvements, it should be noted that the direct evaluation and connectivity both uses symmetry in the CPU version. This means that the speedup would be significantly higher (almost a factor 2) if the CPU version would not use this symmetry.

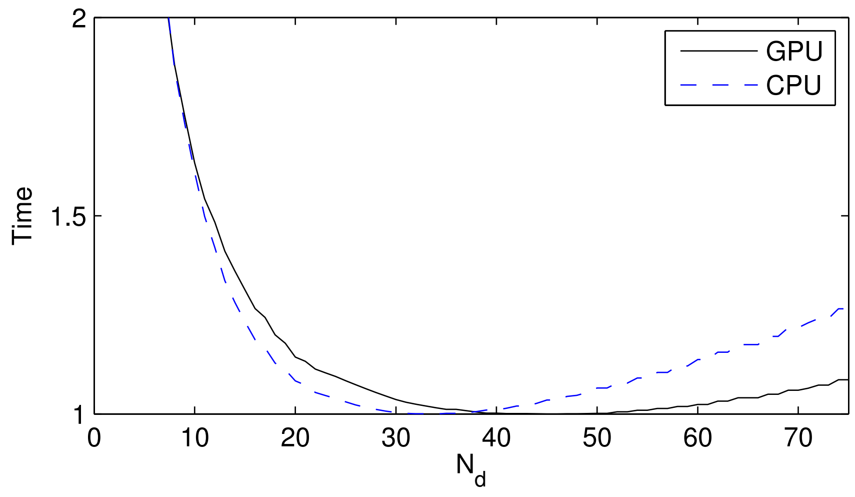

Fig. 5.2 Time as a function of the number of sources per box N_d for the CPU and the GPU implementation, both normalized so that the fastest time $\equiv 1$.

The number of sources per box clearly changes with the total number of sources, and an increase in the number of levels decreases the average number of sources per box with a factor of 4. It is necessary to find the best range of particles per box to operate with. 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 results (normalized against the lowest time on each platform) can be found in Figure 5.2, 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. This is because at low values of N_d, the multipole shifts are fully dominating 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

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

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

For the optimal value 45 of N_d, the time distribution between the different parts of the algorithm is plotted in Figure 5.3 for $N = 45 \times 2^{16}$, which gives 45 sources in each box. The distribution will change depending on the number of source points in the box, and with $N_d = 45$, the time of P2P varies between

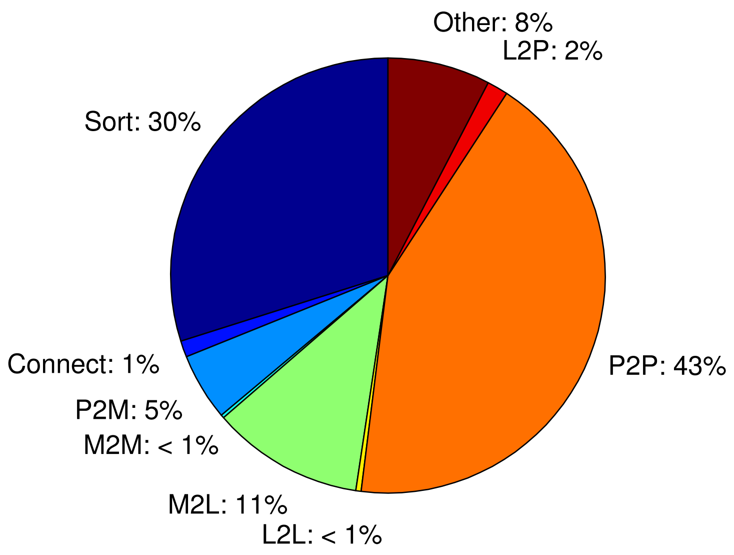

Fig. 5.3 Time distribution between different parts of the algorithm for the GPU. The most time consuming part in the case studied here is the direct evaluation (P2P), followed by sorting and M2L translations. The field *"other"* contains all the data transfers between host and GPU.

25% to 55% for 8 levels (actual number of sources/box varies from 18 to 72). It is particularly interesting to note 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 post-scaling) 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. Having a high number of coefficients will quickly fill up the shared memory and less shift operations can therefore be performed in parallel. The speedup as a function of number of coefficients is plotted in Figure 5.4, where the simulation was performed on 1×10^6 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.

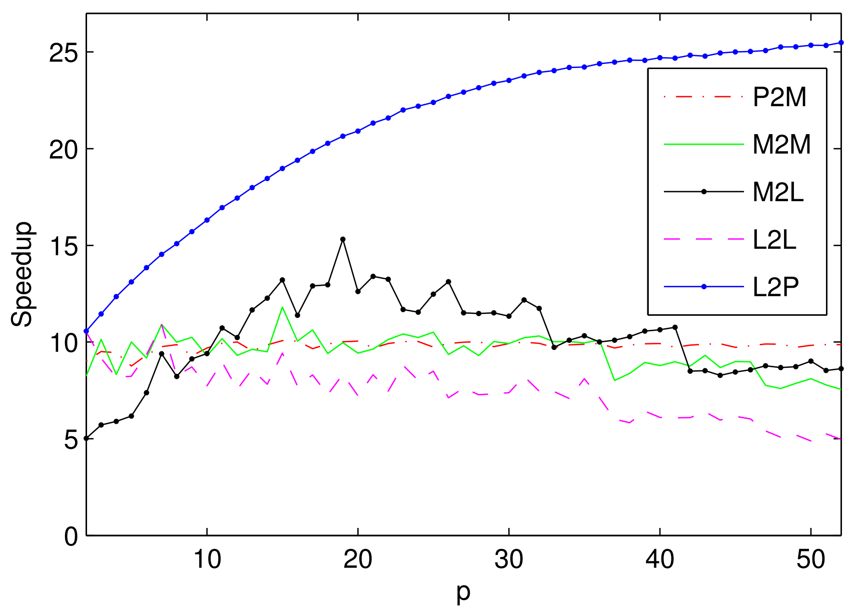

Fig. 5.4 Speedup as a function of the number of multipole coefficients.

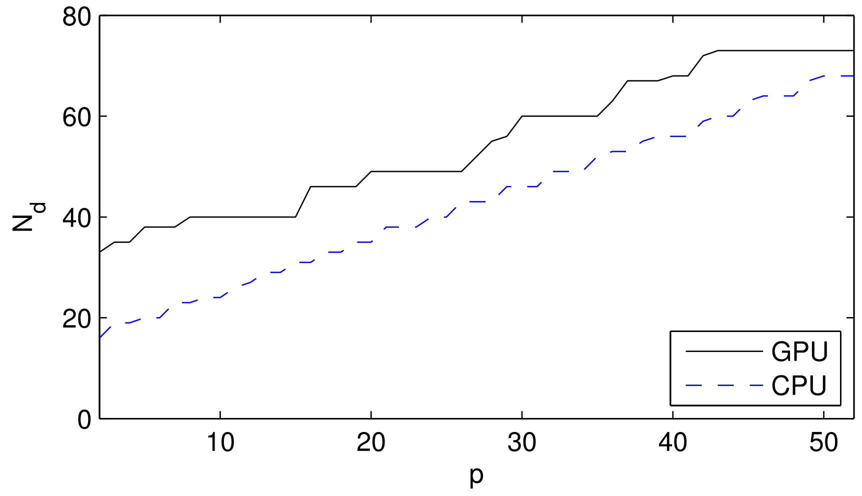

Fig. 5.5 Optimal value of N_d as a function of the number of multipole coefficients.

The difference in speedup for L2P at low number of coefficients is likely due to other overhead in the calculations, as these values converges to stable values at high number of coefficients.

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

From Figure 5.5, it can be seen 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 Figure 5.6, the speed of the entire algorithm is compared with the speed of a direct summation for both the CPU and the GPU implementation. It can be seen that the speedup of the GPU code is increasing with the number of particles, which is expected, as too few particles does not load the GPU enough. Looking at the direct summation times, the GPU scales linearly in the beginning where the number of working cores still increases, and scales quadratically later when all cores are being used fully. The direct evaluation is also a good comparison to what speedups that can be achieved with the GPU code, as a double sum is very easy to perform in parallel, and the computational time is dominated by evaluating the interaction kernel (i.e. not memory bandwidth dominated). Recall, however, that symmetry is being used in the CPU implementation, which almost speeds up the calculation there with a factor of 2. From Figure 5.6, it can be seen that it is more beneficial to use the FMM algorithm if the number of points exceeds about 3500 on the GPU. When comparing the speedup (Figure 5.7) of the direct interaction compared to the FMM, direct evaluation is better (15 compared to about 11) for large N. 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 Figure 5.3).

Looking at the individual parts (Figure 5.8), 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 because very few shifts are performed at the lower levels, causing many idle GPU cores. The situation is the same for the connectivity. As these algorithms has to be performed each level at a time, the low performance of the shifts high up in the multipole tree decreases the performance of the entire step, which is a reason why they show such high increase in speedup towards higher 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 Figure 5.1).

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 Figure 5.3, the GTX 480 card took 30% longer than the Tesla card, while the Tesla card took 25% longer

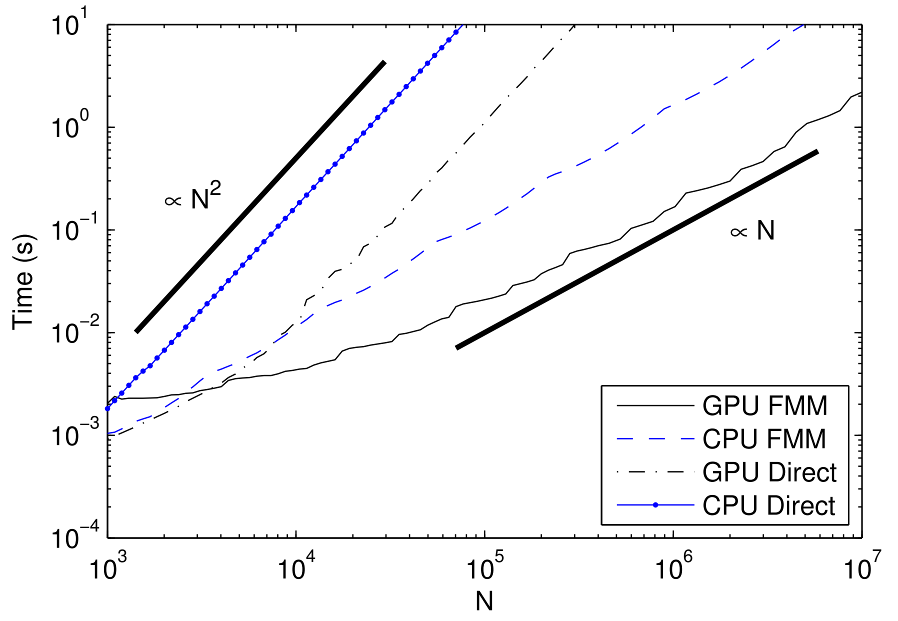

Fig. 5.6 Total time of the algorithm as a function of the number of sources. For the FMM-algorithm, the simulation was performed with $p = 17$, implying a tolerance of about 10^{-6}.

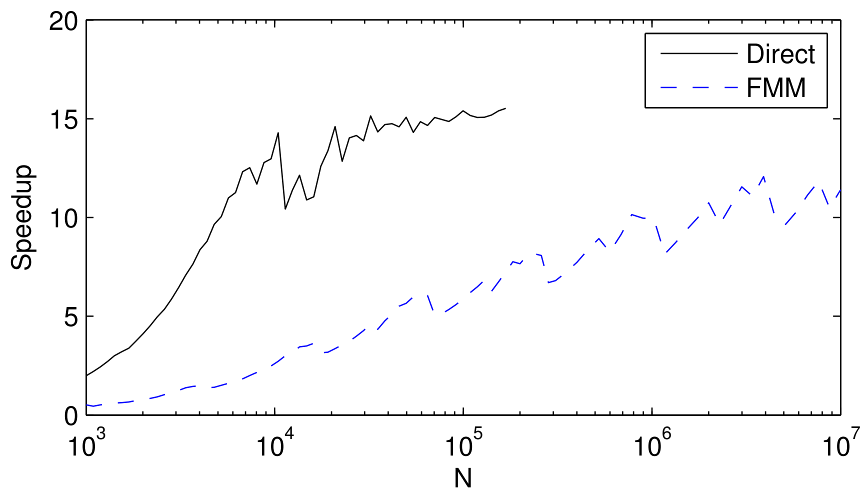

Fig. 5.7 Speedup as a function of the number of sources N.

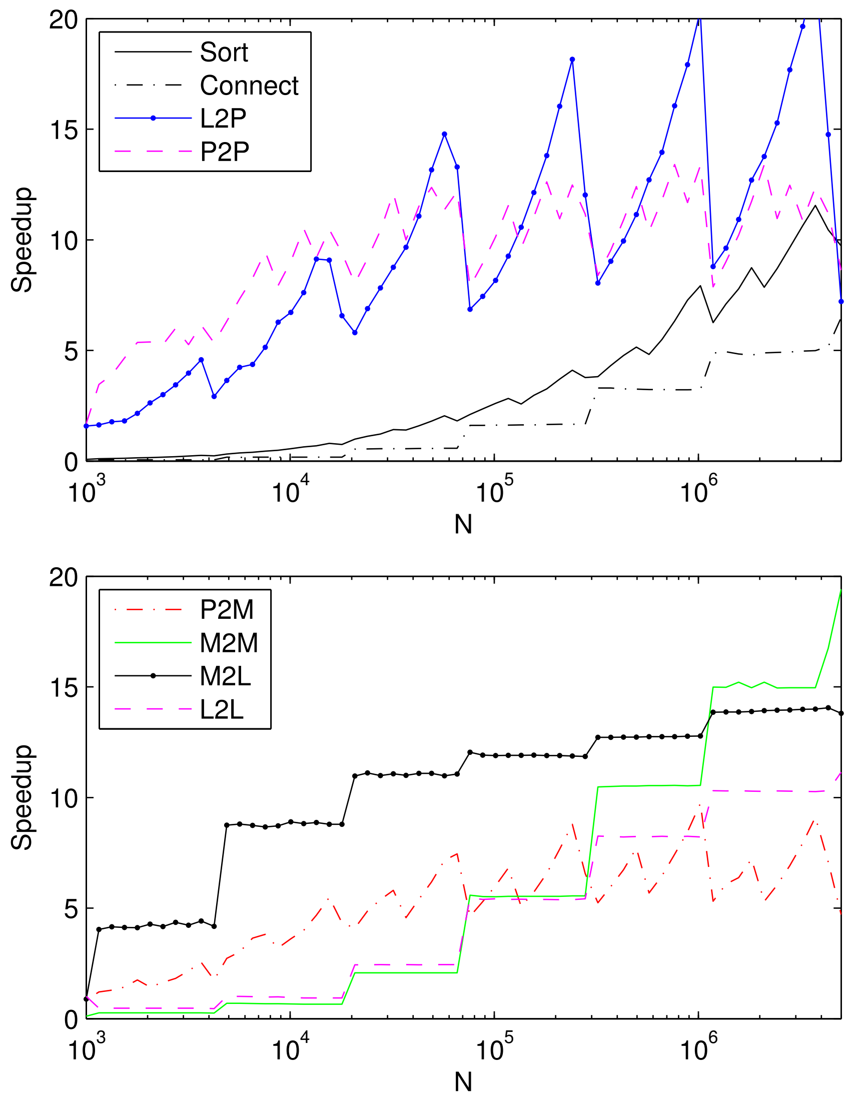

Fig. 5.8 Speedup of individual parts as a function of the number of sources.

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). This shows that the much cheaper GTX 480 gaming card is a perfectly reasonable option for this implementation of the fast multipole method, even though it is in double precision.

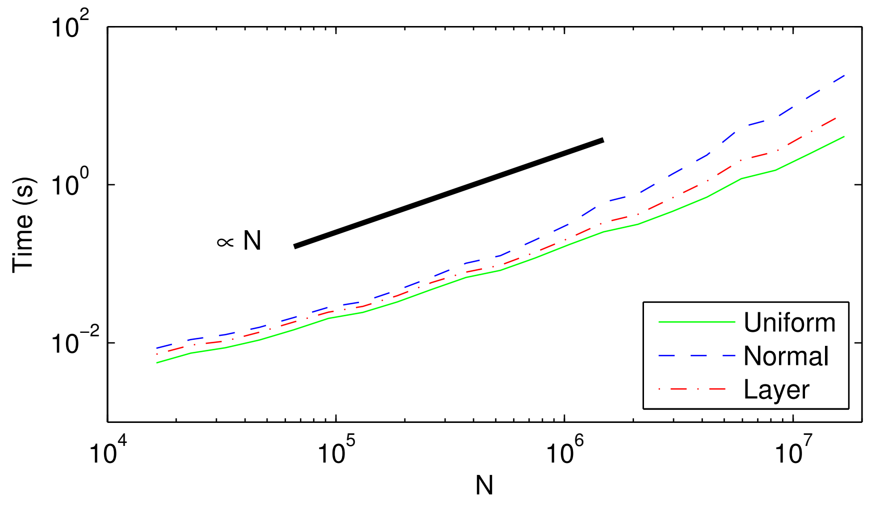

Fig. 5.9 Performance of the code when evaluating the harmonic potential for three different distributions of points. The source points were *(i)* uniformly distributed in $[0,1] \times [0,1]$, *(ii)* normally distributed with variance $1/100$, and *(iii)* distributed in a 'layer' where the x-coordinate 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 Figure 2.1.

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 (2.1) and (2.2)) we measured the evaluation time for increasing number of points sampled from three very different distributions. As shown in Figure 5.9, the code is robust under highly non-uniform inputs and scales well at least up to some 10 million source points.

When the distribution of particles is increasingly non-uniform, more boxes will be in each others near-field resulting in more direct interactions. This is tested in Figure 5.10 for the two non-uniform distributions from Figure 5.9. Both the CPU- and GPU timings have been normalized to the time of a homogeneous distribution. The results indicate that the decrease in performance for highly non-uniform distributions is less for the GPU version than the CPU version. From the timings of the individual parts, it is seen that almost all the increase in computational time comes from the P2P-shift. The speedup for all time consuming individual parts is relatively constant with respect to the degree of non-uniformity (e.g. the parameter σ in Figure 5.10) and the reason the GPU code handles a highly non-uniform distribution better is because the P2P evaluation has a higher speedup than the overall code.

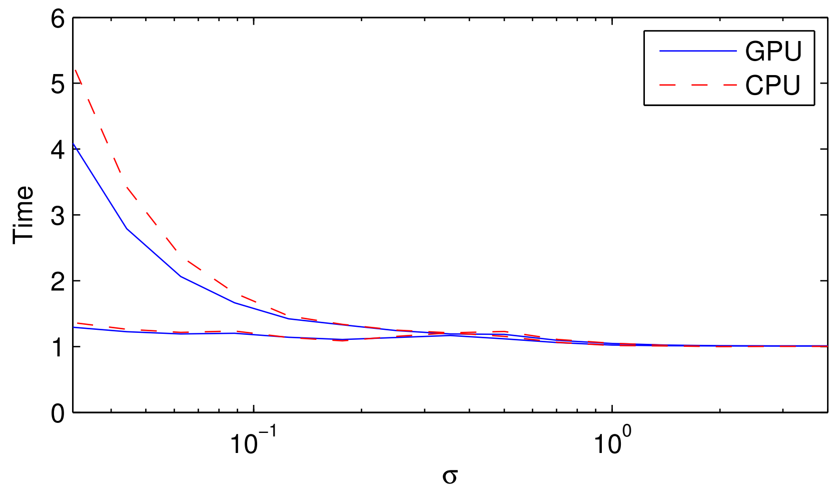

Fig. 5.10 Robustness of adaptivity. Time for two different non-uniform distributions normalized with respect to a uniform distribution of points. The top two graphs are for the normal distribution of sources, while the lower two graphs are the 'layer' distribution. See text for further details.

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 for the direct N-body evaluation, a task for which GPUs are generally thought to be well suited (see Figure 5.7).

In our implementation, all parts of the algorithm obtain 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 Figures 5.5 and 5.8). Given the data-parallel signature of this particular operation, this result is quite the expected one. 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 Figure 5.6. Our tests also shows 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 deserves 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, all in all naturally 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, will be able to perform quite close to optimally. This 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.

Acknowledgment

We like to acknowledge inputs from Pavol Bauer and Marcus Holm.

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. S. Aluru. Greengard's N-body algorithm is not order N. *SIAM J. Sci. Comput.*, 17(3):773–776, 1996. doi:10.1137/S1064827593272031.
2. G. Blelloch and G. Narlikar. A practical comparison of N-body algorithms. In *Parallel Algorithms*, volume 30 of *Series in Discrete Mathematics and Theoretical Computer Science*, 1997.
3. J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM J. Sci. Stat. Comput.*, 9(4):669–686, 1988. doi:10.1137/0909044.
4. B. A. Cipra. The best of the 20th century: Editors name top 10 algorithms. *SIAM News*, 33(4), 2000.
5. F. A. Cruz, M. G. Knepley, and L. A. Barba. Petfmm-a dynamically load-balancing parallel fast multipole library. *Internat. J. Numer. Methods Engrg.*, 85(4):403–428, 2011. doi:10.1002/nme.2972.

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

6. S. Engblom. On well-separated sets and fast multipole methods. *Appl. Numer. Math.*, 61(10):1096–1102, 2011. doi:10.1016/j.apnum.2011.06.011.
7. L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987. doi:10.1016/0021-9991(87)90140-9.
8. M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics*, volume 5 of *Texts in Computational Science and Engineering*. Springer Verlag, Berlin, 2007.
9. N. A. Gumerov and R. Duraiswami. *Fast multipole methods for the Helmholtz equation in three dimensions*. Elsevier Series in Electromagnetism. Elsevier, Oxford, 2004.
10. N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *J. Comput. Phys.*, 227(18):8290–8313, 2008. doi:10.1016/j.jcp.2008.05.023.
11. M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, 2007.
12. T. Hrycak and V. Rokhlin. An improved fast multipole algorithm for potential fields. *SIAM J. Sci. Comput.*, 19(6):1804–1826, 1998. doi:10.1137/S106482759630989X.
13. Y. Liu. *Fast Multipole Boundary Element Method: Theory and Applications in Engineering*. Cambridge University Press, Cambridge, 2009.
14. *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`.
15. R. Sedgewick. *Algorithms in C*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1990.
16. B. Shanker and H. Huang. 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, 2007. doi:10.1016/j.jcp.2007.04.033.
17. H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J. Sci. Comput.*, 30 (5):2675–2708, 2008. doi:10.1137/070681727.
18. M. Vikram, A. Baczewzki, B. Shanker, and S. Aluru. Parallel accelerated Cartesian expansions for particle dynamics simulations. In *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, pages 1–11, 2009. doi:10.1109/IPDPS.2009.5161038.
19. H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235–246, 2010. doi:10.1109/ISPASS.2010.5452013.
20. R. Yokota and L. A. Barba. Treecode and fast multipole method for N-body simulation with CUDA. In W.-M. Hwu, editor, *GPU Computing Gems Emerald Edition*, chapter 9, pages 113–132. Morgan Kauf-

mann/Elsevier, 2011.