Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures

Qi Hu
huqi@cs.umd.edu

Nail A. Gumerov*
gumerov@umiacs.umd.edu

Ramani Duraiswami†
ramani@umiacs.umd.edu

Institute for Advanced Computer Studies (UMIACS) and Department of Computer Science
University of Maryland, College Park

ABSTRACT

We fundamentally reconsider implementation of the Fast Multipole Method (FMM) on a computing node with a heterogeneous CPU-GPU architecture with multicore CPU(s) and one or more GPU accelerators, as well as on an interconnected cluster of such nodes. The FMM is a divide-and-conquer algorithm that performs a fast N-body sum using a spatial decomposition and is often used in a time-stepping or iterative loop. Using the observation that the local summation and the analysis-based translation parts of the FMM are independent, we map these respectively to the GPUs and CPUs. Careful analysis of the FMM is performed to distribute work optimally between the multicore CPUs and the GPU accelerators. We first develop a single node version where the CPU part is parallelized using OpenMP and the GPU version via CUDA. New parallel algorithms for creating FMM data structures are presented together with load balancing strategies for the single node and distributed multiple-node versions. Our implementation can perform the N-body sum for 128M particles on 16 nodes in 4.23 seconds, a performance not achieved by others in the literature on such clusters.

ACM computing classification: C.1.2 [**Multiple Data Stream Architectures**]:Parallel processors; C.1.m [**Miscellaneous**]: Hybrid systems; F.2.1: [**Numerical Algorithms and Problems**]
General terms: Algorithms, Design, Performance, Theory

1. INTRODUCTION

The $N-$body problem, in which the sum of the "influence" or "potential" Φ of N particles ("sources") at locations $\mathbf{x}_i, i = 1, \ldots, N$ with "strength" q_i, $i = 1, \cdots, N$ is computed at M locations ("receivers") $\mathbf{y}_j$, $j = 1, \ldots, M$, arises in a number of contexts including stellar dynamics, molecular dynamics, boundary element methods, wave propagation, and statistics. This problem may also be viewed as the computation a matrix-vector product (MVP) of a dense rectangular $M \times N$ matrix $\mathbf{\Phi}$, which is derived from the kernel function Φ, with an arbitrary vector $\mathbf{q}$

$$\phi = \mathbf{\Phi q} \quad \Longleftrightarrow \quad \phi(\mathbf{y}_j) = \sum_{i=1}^{N} \Phi(\mathbf{y}_j, \mathbf{x}_i) q_i, \qquad (1)$$

$j = 1, ..., M$. Here $\Phi(\mathbf{y}, \mathbf{x})$ is the kernel function. The cost of this operation is quadratic in N (for $M \sim N$), which can be prohibitive for large N. In many applications such sums are computed as a part of a time stepping procedure, where the particles move according to forces computed with the potential, and this computation must be repeated. In other applications, such MVPs arise in the iteration used for solution of linear systems involving matrices similar to $\mathbf{\Phi}$, and its expense becomes a major constraint to scaling the algorithm to larger sizes and greater fidelity.

The fast multipole method (FMM) is an algorithm that, given a specified accuracy ϵ, computes (1) to this guaranteed accuracy with linear time and memory complexity. It was first developed for the Coulomb kernel [1], which in 3D is

$$\Phi(\mathbf{y}, \mathbf{x}) = \begin{cases} |\mathbf{y} - \mathbf{x}|^{-1}, & \mathbf{x} \neq \mathbf{y}, \\ 0, & \mathbf{x} = \mathbf{y}. \end{cases} \qquad (2)$$

In all the text below, we use this and its gradient, although our algorithm is general.

The FMM is a divide-and-conquer approximation algorithm which uses a construct from computational geometry called "well separated pair decomposition" (WSPD) with the aid of recursive data-structures based on octrees (which have a cost of $O(N \log N)$). The FMM decomposes the matrix $\mathbf{\Phi}$ in (1) into a dense and a sparse part, $\mathbf{\Phi} = \mathbf{\Phi}^{(dense)} + \mathbf{\Phi}^{(sparse)}$, where $\mathbf{\Phi}^{(dense)}(\mathbf{y}_j, \mathbf{x}_i) = 0$ for $\mathbf{x}_i \notin \Omega(\mathbf{y}_j)$, i.e. for the sources located outside some neighborhood of point $\mathbf{y}_j$, while $\mathbf{\Phi}^{(sparse)}(\mathbf{y}_j, \mathbf{x}_i) = 0$ for $\mathbf{x}_i \in \Omega(\mathbf{y}_j)$. The sparse MVP can be performed directly, whereas a hierarchical algorithm is used to compute $\mathbf{\Phi}^{(dense)}\mathbf{q}$ to a specified error ϵ via use of appropriate data structures, truncated series/integral representations of the kernel function (multipole/local expansions), and translations of these.

Subsequently the FMM was also developed for other kernels. In particular, its behavior for oscillatory kernels (those arising from the Helmholtz or Maxwell equations) is different, and several modifications to the original FMM ansatz were necessary to achieve $O(N)$ performance [2,

*Also Fantalgo LLC, Elkridge, MD; and Center for Micro and Nanoscale Dynamics of Dispersed Systems, Bashkir State University, Ufa, Russia.
†Also Fantalgo LLC, Elkridge, MD.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

3]. The FMM can be extended for computation of vector kernels (e.g. dipoles), and gradients of the potential (or forces) can be also computed. Kernel independent multipole methods that do not require development of a separate mathematical apparatus for new non-oscillatory kernels have been developed [4].

1.1 Fast multipole method and scalability

The FMM, because of its linear complexity, allows the algorithm size to scale well, if it were implementable efficiently on new architectures. On modern multicore and GPU architectures, this requires parallelization of the algorithm. The FMM has been sought to be parallelized almost since its invention – see e.g., [5, 6, 7, 8]. However, these works are mostly related to more coarse-grained parallel algorithms and often focus on tree codes rather than the FMM. With the advent of multicore processors and GPUs there was a fresh opportunity. GPU parallelization of the FMM was first achieved in [9].

This work has since continued in several papers in SC 2009 and 2010, including the 2009 Gordon Bell prize winner [10]. The FMM was considered on a cluster of GPUs in [11, 10], and the benefits of architecture tuning on networks of multicore processors or GPUs was considered in [12, 13, 14]. In these papers adaptations of previous FMM algorithms were used, and impressive performance was achieved.

1.2 Present contribution

We fundamentally reconsider the FMM algorithm on heterogeneous architectures to achieve a significant improvement over recent implementations and to make the algorithm ready for use as a workhorse simulation tool for both time-dependent vortex flow problems and for boundary element methods. Our goal is to perform large rotorcraft simulations [15] using coupled free vortex methods with sediments and ground effect [16] as well as develop fast methods for simulation of molecular dynamics [17], micro and nanoscale physics [11], and astrophysics [18].

We consider essentially similar hardware of the same power consumption characteristics as the recent papers, and our results show that the performance can be significantly improved. Since GPUs are typically hosted in PCI-express slots of motherboards of regular computers (often with multicore processors), an implementation that just uses GPUs or multicore CPUs separately wastes substantial available computational power. Moreover, a practical workstation configuration currently is a single node with one or more GPU-accelerator cards and a few CPU sockets with multicore processors. With a view to providing simulation speed-ups for both a single node workstation and a cluster of such interconnected nodes, we develop a heterogeneous version of the algorithm.

To achieve an optimal split of work between the GPU and CPU, we performed a cost study of the different parts of the algorithms and communication (see Fig. 1). A distribution of work that achieves best performance by considering the characteristics of each architecture is developed. The FMM in practical applications is required to handle both the uniform distributions often reported on in performance testing and the more clustered non-uniform distributions encountered in real computations. We are developing strategies to balance the load in each of these cases.

Since our algorithm is to be used in time-stepping where the data distribution changes during the simulation, we wanted to improve scalability in the algorithms for creating the FMM data-structures and interaction lists. Extremely fast algorithms, which generate these lists in small fractions of the time of the FMM sum itself, were developed.

Our single node version with two Intel Nehalem 5560 2.8 GHz processors per node (total eight cores) and two NVIDIA Tesla S1070 GPUs provides 8 million particle simulation for 1.6 s (potential) and 2.8 s (potential+force) (truncation number $p = 8$). The algorithm was extensively tested on clusters with two through thirty two nodes with the same basic architecture on which 1 billion particle simulations were achieved. Note that the largest size for the FMM achieved on a run on 256 GPUs reported on SC2009 was 16 million particles with approximately 1 s for a vortex element method velocity+stretching computation ($p = 10$) [10]. Similar timings for a N body (potential+force) problem were achieved by our version on 4 nodes (8 GPUs). While not explicitly indicated in [10], according to [11] the two problems have a theoretical flop count difference of about a factor of 5.5, providing an indication of our speedup relative to [10]. (Note, in a real implementation using efficient representations of differentiation operators [19, 15] better speedups may be possible.)

2. FMM ALGORITHM

The FMM splits the matrix vector product (1) into a sparse and dense part. The products $\mathbf{\Phi}^{(dense)}\mathbf{q}$ and $\mathbf{\Phi}^{(sparse)}\mathbf{q}$ can be computed independently and in parallel. For optimal performance the decomposition must be performed in a way to balance the time for computation of the respective parts. This in turn determines the depth of the hierarchical space partitioning and the performance achieved in the two parts.

The "analysis part " of the FMM computes $\mathbf{\Phi}^{(dense)}\mathbf{q}$ as:

- **Generation of data structure**. The domain is scaled to a unit cube and recursively divided via an octree structure to level $l_{\max}$. Level l contains 8^l boxes. The source and receiver data structures exclude empty boxes, allow fast neighbor finding, and build interaction lists (e.g., via bit-interleaving [3]).
- **Precomputations** of parameters common to several boxes, such as elements of translation matrices.
- **Upward pass**. This generates the multipole basis (M) expansion coefficients $\{C_n^m\}$ at each source box from level $l = l_{\max}$ to $l = 2$. At level $l_{\max}$ this is done via summation of multipole expansion coefficients $\left\{C_n^{(i)m}\right\}$ for all sources i at their box centers. For levels $l = l_{\max} - 1, ..., 2$ this is done via a multipole-to-multipole (M2M) translation of coefficients from children to parent boxes followed by a consolidation.
- **Downward pass.** Local basis (L) expansion coefficients $\{D_n^m\}$ are generated for each receiver box from levels $l = 2$ to $l = l_{\max}$ via a two step procedure at each level: first, a multipole-to-local (M2L) translation is done followed by consolidation from a special neighborhood stencil implied by the WSPD. Then, a local-to-local (L2L) translation from the parent to child receiver box is performed followed by consolidation with the result of the first step.
- **Evaluation.** Evaluate local expansions (L) at level $l_{\max}$ at all receiver locations in the box.

Several different bases and translation methods have been proposed for the Laplace kernel. We used the expansions and methods described in [19, 9] and do not repeat details. Real valued basis functions that allow computations to be performed recursively with minimal use of special functions, complex arithmetic, or large renormalization coefficients are used. L- and M-expansions are truncated to contain p^2 terms with p selected to provide the required accuracy. Translations are performed using the RCR decomposition [20, 19] (rotation – coaxial translation – back rotation) of the matrix translation operators, which provides $O(p^3)$ translation with a low asymptotic constant. It was shown in [19] that these methods are more efficient than the asymptotically superior methods in [21] for problems usually encountered in practice.

The above algorithm can also be modified to work well with nonuniform distributions by skipping empty boxes, by adjusting the value of p for the P2M and L2P steps to satisfy the error bound for each particle, and by stopping the downward pass for sparse boxes before the finest level. Even more adaptive algorithms are possible (see e.g., [22, 3]), but were not implemented here.

The "sparse part" of the FMM computes $\mathbf{\Phi}^{(sparse)}\mathbf{q}$ by taking all receiver particles in a given box at the finest level and computing the influence of sources in that box and neighboring boxes directly.

3. FMM DATA STRUCTURES ON THE GPU

The need for a fast code for data structures is a manifestation of Amdahl's law. In a serial FMM code, generation of basic data structures usually takes a small portion of the total algorithm execution time. In some applications of the FMM, such as for iterative solution of large linear systems, this step is even less important as it is amortized over several iterations. The typical way of doing the data structures is via an $O(N \log N)$ algorithm that uses sorting, which is usually done on the CPU [9]. However, for large dynamic problems, when the particle positions change every time step, the cost of this step would dominate, especially when the FMM itself is made very fast, as we propose to do.

Reimplementing the CPU algorithm for the GPU would not have achieved the kind of acceleration we sought. The reason is that the conventional FMM data structures algorithm employs sorting of large data sets and operations such as set intersection on smaller subsets, that require random access to the global GPU memory, which is not very efficient.

Instead, we were able to device a new parallelizable algorithm, which generates the FMM data structure in $O(N)$ time, bringing the overall complexity of the FMM to $O(N)$ for a given accuracy. Our efficient algorithm is based on use of occupancy histograms (i.e., the counts of particles in all boxes), bin sorting, and parallel scans [23]. A potential disadvantage of our approach is the fact that the histogram requires allocation of an array of size $8^{l_{\max}}$ where zeros indicate empty boxes. Nonetheless this algorithm for GPUs with 4 GB global memory enables of data structures up to a maximum level $l_{\max} = 8$, which is sufficient for many problems. In this case accelerations up to two orders of magnitude compared to CPU were achieved. For problems that required greater octree depth, we developed a distributed multi-GPU version of the algorithm, where the domain is divided via octrees spatially and distributed to the GPUs, each GPU performs independent structuring of data residing in its domain, and global indexing is provided by applying prefixes associated with each GPU. The problem decomposition needed for load balancing can also be done with this data structure.

We describe the algorithm for a single GPU. For a multi-GPU setup, some more or less obvious modifications are needed. The same algorithm is repeated for the source and receiver hierarchies. Each step has $O(N)$ complexity, and the steps are easily parallelizable on the GPU.

- *Determine of the Morton index [24] for each particle*, after scaling of all data to a unit cube using bit-interleaving (e.g. [25, 3]). On the GPU this does not require communication between threads.
- *Construction of occupancy histogram and bin sorting.* The histogram shows how many particles reside in each spatial box at the finest level. In this step, the box index is its Morton index. Bin sorting occurs simultaneously with the histogram construction. Note that there is no need to sort the particles inside the box — bin sorting just results in an arbitrary local rank for each particle in a given box. In this step we use the `atomicAdd()` GPU function. While this may cause the threads to access the same memory address sequentially to update the value, the performance is not compromised because most threads are usually working on different boxes and the method rarely serializes.
- *Parallel scan and global particle ranking.* Parallel scan [23, 26] is an efficient algorithm that provides a pointer to the particles in a given box in the final array. Particle global ranking is simply a sum of its global bookmark and local arbitrary rank.
- *Final filtering.* This process simply removes entries for empty boxes and compresses the array, again using a scan, so the empty boxes are emitted in the final array.
- *Final bin sorting.* Particle data is placed into the output array according to their global ranking.

The second part of the algorithm determines the interacting source boxes in the neighborhood of the receiver boxes. The histogram for the receivers can be deallocated while retaining the one for sources. We also keep the array A of source boxes obtained after the parallel scan (before compression). This enables fast neighbor determination **without sort, search, or set intersection operations.**

For a given receiver box i, its Morton index n is available as the ith entry of the array `ReceiverBoxList`. This index allows one to determine the Morton indices of its spatial neighbors. As a new neighbor index is generated, the occupancy map is checked. If the box is not empty, the corresponding entry in array A provides its global rank, which is stored as the index of the neighbor box.

Computation of the parent neighborhoods and subdivision of the domains for translation stencils, which require a more complex data access pattern, is performed on the CPU, which creates the arrays `ReceiverBoxList, SourceBoxList, NeighborBoxList` and bookmarks (values indicating the starting and ending values of the particle number in a box).

Table 1 shows the time for data structures generation using a NVIDIA GTX480 and CPU Intel Nehalem quad-core 2.8 GHz (a single core was used) for $N = 2^{20}$ source

$l_{\max}$	CPU (ms)	Improved CPU (ms)	GPU (ms)
3	1293	223	7.7
4	1387	272	13.9
5	2137	431	13.0
6	8973	1808	34.6
7	30652	6789	70.8
8	58773	7783	124.9

Table 1: FMM data structure computation for 2^{20} uniform randomly distributed source and receiver particles using our original CPU $O(N \log N)$ algorithm, the improved $O(N)$ algorithm on a single CPU core, and its GPU accelerated version.

and $M = 2^{20}$ receiver points uniformly randomly distributed inside a cube. The octree depth was varied in the range $l_{\max} = 3, ..., 8$. Column 2 shows the wall clock time for a standard algorithm, which uses sorting and hierarchical neighbor search using set intersection (the neighbors were found in the parent neighborhood domain subdivided to the children level). Column 3 shows the wall clock time for the present algorithm on the CPU. It is seen that our algorithm is several times faster. Comparison of the GPU and CPU times for the same algorithm show further acceleration in the range 20-100. As a consequence, the data-structure step is reduced to a small part of the computation time again.

4. PARALLELIZATION OF THE FMM

We present first the algorithm for a single heterogeneous node (i.e., typical contemporary workstation with several shared-memory CPU cores and one or more GPUs). Next, an approach for clusters consisting of several heterogeneous nodes, is presented. This algorithm is sufficient for the number nodes we have access to. We also briefly consider the case of even larger numbers of nodes.

4.1 Single heterogeneous node

Different stages of the FMM have very different efficiency when parallelized on the GPU (Fig. 1). The lowest efficiency (due to limited GPU local memory) is for translations. On the other hand, computation of $\mathbf{\Phi}^{(sparse)}\mathbf{q}$ on the GPU is very efficient (making use of special instructions for the reciprocal square root and multiply-and-add operations), as well as the generation of M-expansions and evaluation of L-expansions. In fact, anything having to do with particles is very efficient on the GPU, and translations are relatively efficient on the CPU.

Fig. 2 illustrates the work division between the CPU cores and GPU(s) on a single node. The large source and receiver data sets are kept in GPU global memory, and operations related to particles are performed only on the GPU. This makes dynamic simulations (where particles change location) efficient, since particle update can be done efficiently on the GPU minimizing CPU-GPU data transfer. The GPU does the jobs that it can do most efficiently, specifically, generating the data structure, generating M-expansions for source boxes at the finest level, performing the sparse MVP, evaluating L-expansions for the receiver boxes at the finest level, and producing final results. Because the result is obtained on the GPU, it can be used immediately for the next time step.

The CPU performs all work related to operations with boxes. It receives as input the box data structure from the

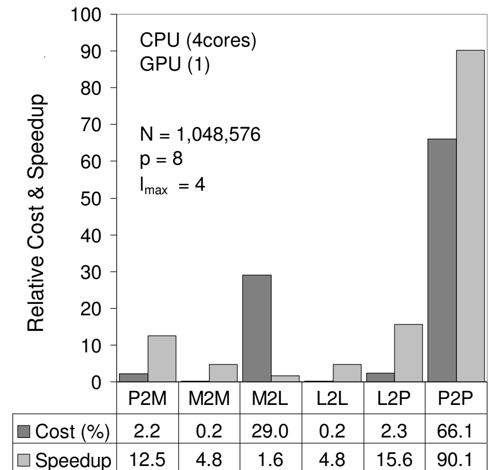

Figure 1: The relative cost and speedup of different steps of the FMM on uniform data on a GPU (NVIDIA GeForce 8800GTX) vs a 4 core CPU (Intel Core 2 extreme QX, 2.67GHz). The relative cost of steps is given for the GPU realization ($l_{\max} = 4$, $p = 8$, $N = M = 2^{20}$). The CPU wall clock time is measured for the same settings as for GPU (not necessarily optimal for the CPU). From [9].

GPU, which is used to generate a translation data structure (note that we use the reduced translation stencils described in [9], instead of the standard 189 per box), and M-expansion coefficients for the non-empty source boxes at the finest level. Then the CPU performs the upward and downward passes of the FMM and returns L-expansions for the non-empty receiver boxes at the finest level.

This strategy has several advantages

1. The CPU and GPU are tasked with the most efficient jobs they can do.
2. The CPU is not idle during the GPU-based computations and our tests show the loads on CPU and on GPU are reasonably balanced.
3. Data transfer between the CPU and GPU includes only p^2 expansion coefficients for each non-empty box, which usually is smaller than the particle data.
4. The CPU code can be better optimized as it may use more complex data structures, e.g. for complex translation stencils. More efficient automatic compilers are available for the CPU.
5. We use double precision without much penalty on the CPU. This is helpful since translation operations are more sensitive to round off.
6. If the required precision is below 10^{-7} single precision can be used for GPU computations. If the error tolerances are more strict, then double precision can be used on GPUs that support them.
7. The algorithm is efficient for dynamic problems.
8. Visualization of particles for computational steering is easy, as all the data always reside in GPU RAM.

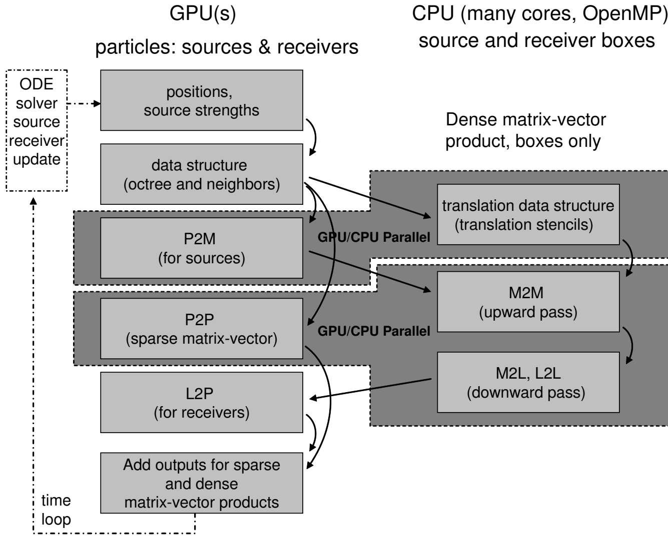

Figure 2: Flow chart of the FMM on a single heterogeneous node (a few GPUs and several CPU cores). Steps shared in dark gray are executed in parallel on the CPU and the GPU.

4.2 Several heterogeneous nodes

For distributed heterogeneous nodes, the above algorithm can be efficiently parallelized by using the spatial decomposition property inherent to the FMM (see Fig. 3). The separation of jobs between the CPU cores and GPUs within a node remains the same. The M2L translations usually are the most time consuming and take 90% or more of the CPU time in the single node implementation. However, if we have P nodes and each node serves only N/P sources located compactly in a spatial domain $\Omega_j^{(s)}$, $j = 1, ..., P$ covered by $N_j^{(s)} \approx N^{(s)}/P$ source boxes at level $l_{\max}$ such that $\Omega_j^{(s)}$ do not intersect, then the number of the M2M and M2L translations for all receiver boxes due to sources in $\Omega_j^{(s)}$ is approximately $(C_{M2M} + C_{M2L})/P$, where C_{M2L} and C_{M2M} are the numbers of all M2L and M2M translations for the entire domain, respectively, and $N^{(s)}$ the number of source boxes. This is achieved because our implementation of the FMM skips empty source boxes.

To achieve scalability for the L2L translations, we introduce an intermediate communication step between the nodes at levels $l = 2, ..., l_{\max}$. This is between the CPUs alone, and does not affect the parallel computation of the sparse MVP on the GPU. For each node we subdivide the receiver boxes handled by it into 4 sets. These are built based on two independent criteria: belonging to the node, and belonging to the neighborhood containing all source boxes assigned to the node. "Belonging" means that all parents, grandparents, etc. of the box at the finest level for which the sparse MVP is computed by a given node also belong to that node. Receiver boxes that do not satisfy both criteria are considered "childless" and the receiver tree for each node is truncated to have that boxes as leaves of the tree.

During the downward pass a synchronization instruction is issued after computations of the L-expansion coefficients for receiver boxes in the tree at each level. The nodes then exchange only information about the expansions for the leaf boxes, and each node sums up only information for the boxes which belong to it. These steps propagate until level $l_{\max}$ at which all boxes are the leave boxes and information collected by each node is passed to its GPU(s), who evaluate the expansions at the node source points and sum the result with the portion of the sparse MVP computed by this node.

4.2.1 *Simplified algorithm*

If the number of nodes is not very large, the above algorithm can be simplified to reduce amount of synchronization instructions and simplify the overall data structure. In a simplified algorithm, each node performs an independent job in the downward pass to produce the L-expansion coefficients for *all* receiver boxes at level $l_{\max}$. These coefficients are not final, since they take into account only contribution of the sources allocated to a particular node. To obtain the final coefficients, all expansions for a given receiver box must be summed up and sent to the node that computes the sparse MVP for that receiver box. This process for all nodes can be efficiently performed in parallel using hierarchical (golden) summation according to the node indices.

In contrast to the general algorithm, in the simplified algorithm some L2L translations are duplicated (see Fig. 3), which deteriorates the algorithm scalability and increases data transfer between the nodes. However, even in the worst case when all L2L translations are repeated on all nodes, the effect of L2L-translation duplication may have a substantial effect on the overall complexity only if the number of all L2L translations violates the restriction $C_{L2L} \ll (C_{M2M} + C_{M2L})/P$. This inequality holds for moderate clusters. Indeed, if we use a scheme with a maximum of 119 or 189 M2L translations per box and one L2L translation then for $P \lesssim 100$ the scheme is acceptable, though sufficient memory per node is needed to keep L-expansion coefficients for all boxes in the tree. Our tests show that satisfactory performance for $N \lesssim 10^9$, which is comparable with the number of particles used in any FMM realizations we are aware of.

For illustration, the flow chart in Fig. 4 shows the algorithm for two heterogeneous nodes. Each node initially has in memory the sources and receivers assigned (randomly) to the node. Based on this, each node builds an octree. After that, all nodes are synchronized and receiver hierarchy data is scattered/gathered, so each node has complete information about all receiver boxes.

Since initial source/receiver data is redistributed between the nodes, each node takes care of a spatially compact portion of particle data. This distribution is done by prescribing weights to each box at a coarse level of the tree and splitting the tree along the Morton-curve to achieve approximately equal weights to each part. Then the single node heterogeneous algorithm described in the previous section is executed with some small modifications. The sparse MVP is computed only for the receiver boxes handled by a particular node, and the dense MVP is computed only for source boxes allocated on that node but for all receiver boxes (i.e. influence of a portion of the source boxes on all receivers is computed). The data on the L-expansions at the finest level is then consolidated for the receiver boxes handled by each node. The final summation consists of evaluation of the L-expansions and summation with partial

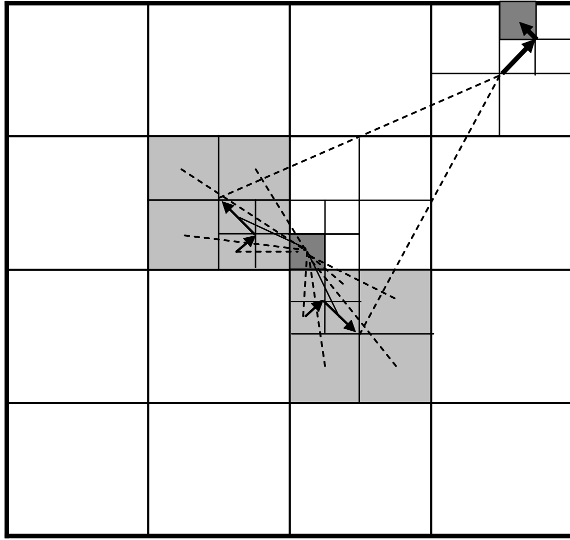

Figure 3: Illustration of separation of the M2M and M2L translation jobs between two nodes. Two nodes handle sources allocated in two light gray source boxes and compute L-expansions for the darker gray receiver boxes. Solid lines with arrows show M2M translations, the dashed lines show M2L translations, and bold solid lines with arrows show L2L translations. M2M and M2L steps do not overlap, and the same (adaptive, empty box skipping) algorithm with different inputs can be executed on each node. L2L translations for the same boxes are duplicated by each node for the simplified algorithm in §3.2.1 and are not duplicated in the general algorithm proposed in §3.2.

sparse MVPs.

5. IMPLEMENTATION ISSUES

We use OpenMP for intra-node CPU computation (as implemented in Intel Fortran/C v.11) and MPI for inter-node communication. As shown in Fig. 4, on each computing node we launch a single process, within which multiple OpenMP threads are used to control multiple GPUs and compute FMM translations on the CPU. MPI calls for communicating with other nodes are made from the master thread of this process.

CUDA 3.2, which we used, allows only one active GPU device in a single CPU thread. Hence, to parallelize the multi-threaded CPU translation and multi-GPU direct sum, OpenMP threads have to be divided into two different groups. To avoid performance degradation due to the nested OpenMP parallel regions performing quite different computational tasks, the threads that control GPUs are spawned first. After a thread launches its GPU kernel function call, it immediately rejoins the master thread. At this point the threads for CPU translations are spawned, while the GPUs perform the local direct summation in the mean time.

After initial partition, each node has its own source and receiver data. The receiver data are mutually exclusive amongst all the nodes; however, the same source data might

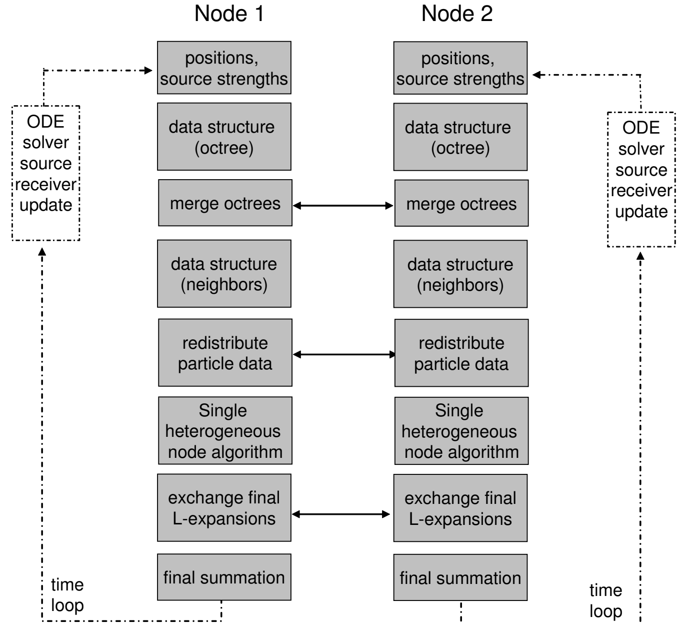

Figure 4: A flow chart of the FMM on two heterogeneous nodes. The single heterogeneous node algorithm (see Fig. 2) is executed in parallel on CPU cores and GPUs available to the node.

repeatedly appear on many nodes since they belong to the interaction neighborhood of many receiver boxes. As shown in Fig. 2, the main thread spawns multiple threads to copy data onto different GPUs and performs initial data structure building. Then each thread sends information on the receiver boxes to the master thread. The master thread computes the global receiver box information and broadcasts it to all other nodes. Based on the previous data structures and the global receiver box information, each GPU then builds its own data structure for the CPU translations, performs initial multipole expansions, and copies them to CPU. Next, the CPU translation and GPU direct sum are performed simultaneously using the scheme described above.

For the simplified algorithm described above, the master thread collects the local expansion coefficients as a binary tree hierarchy within l rounds given 2^l processes. When it finishes, the master process has the local expansion data for all receiver boxes and sends the corresponding data to all other processes.

6. PERFORMANCE TESTS

6.1 Hardware

We used a small cluster ("Chimera") at UMIACS at the University of Maryland (32 nodes) and a large one ("Lincoln") at NCSA at the University of Illinois (192 nodes). In both clusters the same basic node architecture was interconnected via Infiniband. At UMIACS, each node was composed of a dual socket quad-core Intel Nehalem 5560 2.8 GHz processors, 24 GB of RAM per node, and two Tesla S1070 accelerators each with 4 GB of RAM. At NCSA, each node had the same GPU architecture with the same Tesla,

Time (s) \ N	1,048,576		2,097,152		4,194,304		8,388,608		16,777,216	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall clock	0.13	0.13	1.06	1.08	1.07	1.11	1.02	1.10	8.53	8.98
C/G parallel region	0.58	0.30	1.06	1.08	1.58	1.11	4.38	2.21	8.55	8.98
Force+Potential total run	0.71	0.39	1.23	1.22	1.96	1.34	5.11	2.63	10.3	10.1
Potential total run	0.40	0.24	1.16	0.89	1.27	1.25	2.94	1.52	9.76	6.30
Partitioning	–	0.14	–	0.32	–	0.58	–	1.14	–	3.09

Table 2: Performance on a single heterogeneous node: force with potential (best settings). For potential computations, only the total run time is provided.

but the CPUs were Intel Harpertown 5300 processors at 2.33 GHz with 16 GB of RAM. The double precision single node tests were run on a workstation with a NVIDIA Tesla C2050 GPU accelerator with 3 GB, and an Intel Xeon E5504 processor at 2.00 GHz and 6 GB RAM.

6.2 Remarks on the reported results

Partitioning times: Our algorithm takes some time for global partition of the data. However, in a dynamic problem, this step is only needed at the initial step, and this time is amortized over several time steps, after which the global repartitioning may again be necessary. Accordingly, we report the partitioning time separately from the total run time. The run time, however, includes the cost of generating the entire FMM data structure on each node since this will have to be done at each time step. In the tests we measured the time for potential + force (gradient) computations and also for a faster version where only the potential was computed.

Most cases are computed with $p = 8$, which is sufficient to provide single precision accuracy (relative errors in the L^2-norm are below 10^{-5}). When comparisons are made with [10], we set $p = 10$ to match their choice. The benchmark cases include random uniform distribution of particles inside the cube and on the surface of a sphere (spatially non-uniform). In our wider tests we varied the number of sources and receivers N and M. In all the reported cases, $N = M$, while the source and receiver data are different.

6.3 Single heterogeneous node

The algorithm for a single node was extensively tested. The first test was performed on a single node of the Chimera cluster using one and two GPUs with spatially uniform random particle distributions. Table 2 shows the measured performance results in optimal settings (in terms of the tree depth $l_{\max}$) for potential + force computations (the total run time for potential only computations is also included). Fig. 5 plots data only for potential computations in optimal settings. Even though these timing results appear to outperform the results of other authors on similar hardware which we are aware of, one may question whether the algorithm is scalable with respect to the number of particles and number of GPUs since the run time changes quite non-uniformly. An explanation of the observed performance is that the GPU sparse MVP has optimum performance for certain data cluster size $s_{opt}^{(sparse)}$ [9], and when clusters with $s < s_{opt}^{(sparse)}$ are used, this increases both the GPU and CPU times (due to increase of $l_{\max}$). Cluster sizes with $s > s_{opt}^{(sparse)}$ can be optimal, and this can be found from the balance of the CPU and GPU times. Since $l_{\max}$ changes discretely and the CPU time depends only on the number of boxes, or $l_{\max}$ (for uniform distributions), the CPU time jumps only when the level changes. Increase of $l_{\max}$ by one increases the CPU time eight times, and such scaling is consistent with the observed results. On the other hand, the GPU time for fixed $l_{\max}$ and $s > s_{opt}^{(sparse)}$ is proportional to N^2. So there is no way to balance the CPU and GPU times for a fixed $l_{\max}$, except perhaps increasing the precision of CPU computation.

If the GPU time dominates, then use of the second GPU reduces the time, as seen for the cases $N = 2^{20}$ and $N = 2^{23}$. Note also that the parallelization efficiency for 2 GPUs is close to 100%. On the other hand, if the CPU time dominates, then the second GPU does not improve performance if $l_{\max}$ remains the same (see case $N = 2^{22}$ for the potential). Cases $N = 2^{21}$ and $N = 2^{24}$ show a reduction of the time due to the use of the second GPU because of a different reason. For these cases optimal $l_{\max}$ is different when using one or two GPUs. This causes reduction of the CPU time and increase of the single GPU time for two GPUs compared to the case with one active GPU.

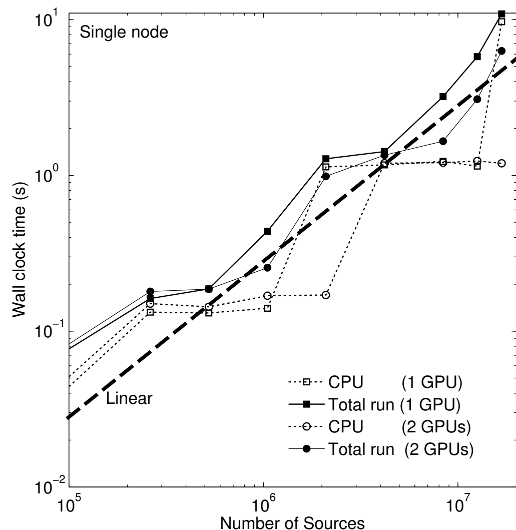

Figure 5: The wall clock time (potential-only computation) for the heterogeneous FMM running on a single node (8 CPU cores) with one and two GPUs. The CPU part is plotted by the thin dashed lines. The thick dashed line shows linear scaling.

We also performed tests with spatially non-uniform distributions (points or the sphere surface). This requires deeper

octrees to achieve optimal levels (usually increase of $l_{\max}$ by 2) and provides more non-uniform loads on the GPU threads, which process the data box by box (the number of points in the non-empty boxes varies substantially). This results in the increase of both CPU and GPU times. In some cases this increase is not substantial (e.g. for $N = 2^{20}$ the CPU/GPU parallel region time is 0.44 s and the total run time is 0.56 s), while we never observed increase more than 2.5 times (e.g. for $N = 2^{22}$ the CPU/GPU parallel region time is 2.82 s and the total run time is 3.26 s for potential only computations) (compare with Table 2).

6.3.1 Double precision GPU performance

Accepted wisdom in scientific computing often requires double precision computations for many problems. We accordingly demonstrate our algorithm with double precision on the CPU, and both single and double precision on the GPU. Table 3 shows results of our tests for potential computation for $N = 2^{20}$ using different truncation numbers p and algorithms with different GPU precision. We used a workstation with a Tesla C2050 and a 4 core Intel e5504 2.00 GHz CPU (which explains the difference in run times in comparison with Tables 2 and 3). The error was measured in the relative L^2-norm using 100 random comparison points. Direct double precision computations were used to provide the baseline result. It is seen that for $p \leqslant 8$ there is no need to use double precision (which slows down the GPU by at least a factor of two). The CPU time grows proportionally to p^3, and the GPU time depends on p as $A + Bp^2$ with a relatively small constant B. This changes the CPU/GPU balance of the heterogeneous algorithm and reduces the optimal tree depth $l_{\max}$ as p increases.

	Prec	$p = 4$	8	12	16
Time (s)	S	0.37	0.62	1.48	2.92
	D	1.36	1.40	1.49	2.95
Error	S	2.8·(−4)	1.4·(−6)	2.5·(−7)	1.2·(−7)
	D	1.6·(−4)	6.9·(−7)	4.3·(−8)	4.3·(−9)

Table 3: Performance and error for single and double precision GPU ($(-m)$ means 10^{-m})

6.4 Multiple heterogeneous nodes

On heterogeneous clusters we varied the numbers of particles, nodes, GPUs per node, and the depth of the space partitioning to derive optimal settings, taking into account data transfer overheads and other factors.

The weak scalability test is performed by fixing the number of particles per node to $N/P = 2^{23}$ and varying the number of nodes (see Table 4 and Fig. 6) for a simplified (small cluster) parallel algorithm. For perfect parallelization/scalability, the run time in this case should be constant. In practice, we observed an oscillating pattern with slight growth of the average time. Two factors affect the perfect scaling: reduction of the parallelization efficiency of the CPU part of the algorithm and the data transfer overheads. The results in Table 2 for 2 GPUs were computed with $l_{\max} = 5$ for cases $P = 1$ and 2 and $l_{\max} = 6$ for cases $P = 4, 8, 16$. In the case of the ideal CPU algorithm the time should reduce by a factor of two when the number of nodes is doubled at constant $l_{\max}$ and increase eight-fold when $l_{\max}$ increases by one. In our case, P is constant, so when $l_{\max}$ increases and P doubles the CPU time should increase by factor of 4. Qualitatively this is consistent with the observed pattern, but the simplified algorithm that we used is not perfectly scalable due to overheads (e.g. due to L2L-translations and related unnecessary duplication of the data structure) becoming significant at large sizes. The deficiency of the simplified algorithm also shows up in the data transfer overheads. The amount of such transfers depends on the number of boxes and the table clearly shows that the overhead time increases with $l_{\max}$. Fig. 6 shows that for relatively small number of nodes this imperfectness is acceptable for practical problems.

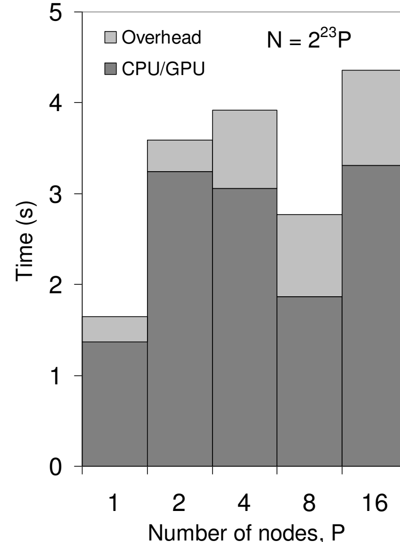

Figure 6: Contribution of the CPU/GPU parallel region time and the overhead (data transfer between the nodes and CPU/GPU sequential region) to the total run time for two GPUs per node (UMIACS Chimera cluster). The data size increases proportionally to the number of nodes. The time is measured for computations of potential only.

We also performed the strong scalability test, in which N is fixed and P is changing (Fig. 7). The tests were performed for $N = 2^{23}$ and $P = 1, 2, 4, 8, 16$ with one and two GPUs per node. The deviations from the perfect scaling can be explained as follows. In the case of one GPU/node, the scaling of the CPU/GPU parallel region is quite good. We found that in this case the GPU work was a limiting factor for the parallel region. This is consistent with the fact that the sparse MVP alone is well scalable. In the case of two GPUs, the CPU work was a limiting factor for the parallel region. Scalability of the algorithm on the CPU is not as good as for the sparse MVP part because of the reasons explained above when the number of nodes increase. However, we can see approximate correspondence of the times obtained for two GPUs/node to the ones with one GPU/node, i.e. doubling of the number of nodes with one GPU or increasing the number of GPUs results in approximately the same timing. This shows a reasonably good balance between the CPU and GPU work in the case of 2 GPUs per node (so this is more or less the optimal configuration for a given problem size).

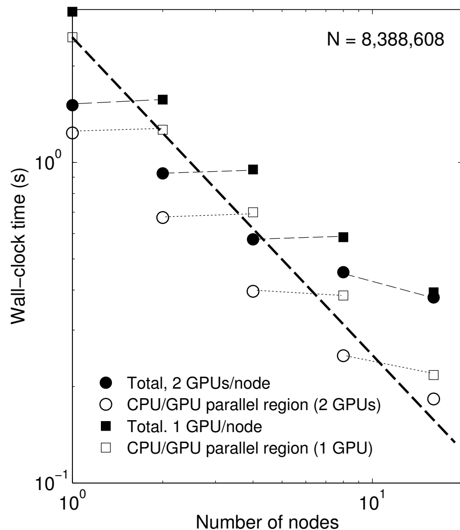

Figure 7: The results of the strong scalability test for one and two GPUs per node of the UMIACS Chimera cluster. The thin lines connect pairs of data points for which the same total number of GPUs was used. The thick dashed line shows perfect scalability $t = O(1/P)$. The time is measured for potential only computations.

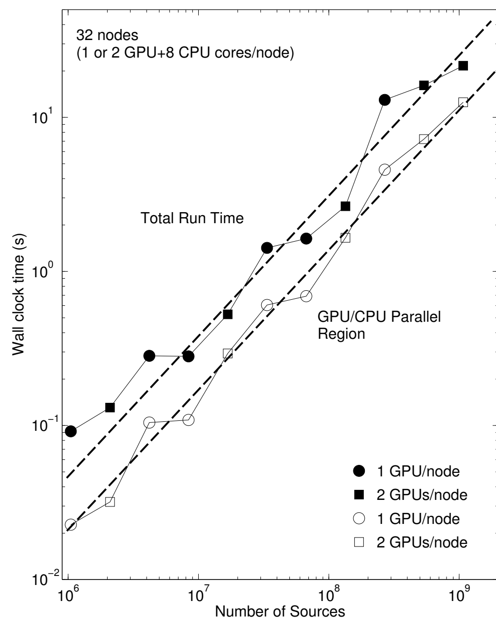

Figure 8: The wall clock time for the heterogeneous FMM running on 32 nodes and using one or two GPUs per node (potential only computations). The number of GPUs per node is optimized according to the load balance between CPU and GPU(s) in the parallel region. The thick dashed lines show the linear scaling.

More significant imperfections are observed for the total run time at increasing P, which is related to the data transfer overheads between the nodes (we also saw that in the weak scalability tests).

We did similar tests on the Lincoln cluster, which produced almost the same results. These best results are obtained for optimal settings, when the GPU(s) reach their peak performance (in terms of the particle cluster size). For such sizes the UMIACS cluster is limited by $N = 2^{28}$. For larger problems a suboptimal performance is obtained; however, such cases are still of practical importance, as solving billion-size problems in terms of seconds per time step is quite useful. Fig. 8 presents the results of the run employing 32 nodes with one or two GPUs per node, which shows a good scaling with N (taking into account that the FMM has jumps when the level changes). In the figure, we plot the best of one-GPU and two-GPU run times (as it was shown above, the use of the second GPU in the heterogeneous algorithm is not always beneficial and may create additional overhead). The largest case computed in the present study is $N = 2^{30}$ for 32 two-GPU nodes. For this case, the CPU/GPU parallel region time was 12.5 s and the total run time 21.6 s.

We believe that the achieved total run times are among the best ever reported for the FMM for the sizes of the problems considered (e.g, comparing with [10, 11, 13, 14, 27]). Fig. 9 compares the wall clock time required by the potential+force computations for the present algorithm with the velocity+stretching timing reported in [10]. The present algorithm was executed with two GPUs per node for 2 and 8 nodes and one GPU for a single node (the total number of GPUs employed is shown in the figure).

6.5 Performance assessment

There are several ways to assess the performance of our algorithm. (See `http://folding.stanford.edu/English/FAQ-flops` for a discussion). One way to assess this performance is to look at the actual number of operations performed in the FMM. This yields much more modest numbers of flops, but may be more useful in comparing the performance of different implementations of the FMM. The most arithmetically intensive part of the FMM is performed by the GPU during the sparse MVP. For uniform distributions and cluster size s, the number of operations at max. tree depth $l_{\max}$ is approximately $N_{op}^{(sp)} = 2^{3l_{\max}} 27 s^2 x$, where x is the number of operations per direct evaluation and summation of the potential or potential+force contribution. For a cluster of size $s = 2^8$ and $l_{\max} = 5$ we have $N_{op}^{(sp)} = 2^{31} 27x$. This corresponds to $N = 2^{3l_{\max}} s = 2^{23}$. Our tests show a very consistent GPU/CPU parallel region time per GPU for this problem size, which is strongly dominated by GPU, $t^{(sp)} = 2.45$ s (two GPUs do the same job for 1.23 s). This means that a single GPU performs at peak rate not less than $R = N_{op}^{(sp)} / \left(2^{30} t^{(sp)}\right) = 22x$ GFlops. The estimation of the x is rather tricky, and somewhat controversial, since a larger value tends to indicate better performance for one's algorithm. For potential only computations it can be estimated either as $x = 9$ in terms of GPU instructions, or as $x = 27$ in terms of CPU instructions, which is the commonly accepted way of counting. The latter value,

Time (s) \ N (P)	8,388,608 (1)		16,777,216 (2)		33,554,432 (4)		67,108,864 (8)		134,217,728 (16)	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall	1.02	1.10	4.49	0.61	2.71	2.80	1.41	1.75	0.85	1.22
CPU/GPU	2.45	1.23	4.49	2.91	2.71	2.80	2.67	1.75	6.25	3.17
Overhead	0.50	0.30	1.03	0.36	0.95	0.85	1.12	0.96	1.31	1.07
Total run	2.95	1.53	5.46	3.27	3.66	3.65	3.79	2.71	7.56	4.24

Table 4: Performance for P heterogeneous nodes with $N/P = 2^{23}$ (potential only).

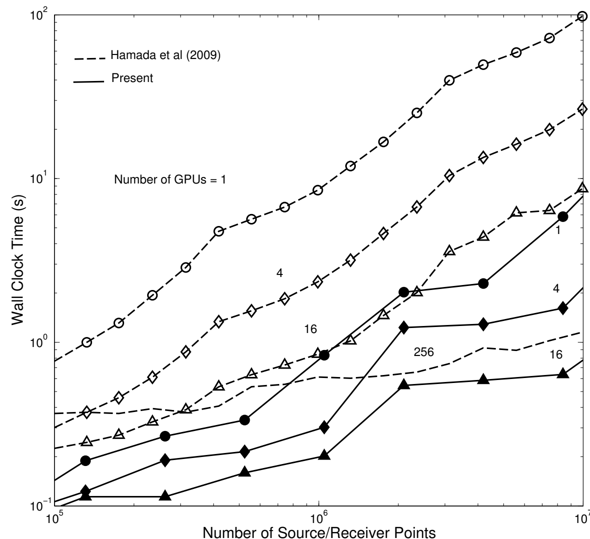

Figure 9: A comparison of the present implementation for N-body force+potential calculations with the 2009 Gordon Bell prize winner [10] for velocity+stretching calculations, which theoretically require about 5.5 times more computation for the same N (see Fig. 12 there). In all cases the truncation number $p = 10$.

which is consistent with the tests of the algorithms using standard CPU counters provides $R_{\max} = 594$ GFlops while $x = 9$ results in $R_{\min} = 198$ GFlops. Both these numbers are within the limit of the GPU used (933 GFlops peak performance reported by NVIDIA). Similar numbers can be computed for the potential+force computations by using $x = 38$ or $x = 15$, and the timing data from Table 2.

The contribution of the CPU part of the algorithm in the parallel region improves this marginally. We count all M2M, M2L, and L2L translations, evaluated the number of operations per translation, and used the measured times for the GPU/CPU parallel region at $l_{\max} = 5$. That provided as a estimate of 27 GFlops per node for 8 CPU cores. Thus we can bound the single heterogeneous node performance for two GPUs between 423 and 1215 GFlops. We used at most 32 nodes with two GPUs each, and our cluster's performance bounds are [13.2, 38] TFlops.

Following [28], several authors, used a fixed flop count for the original problem and computed the performance that would have been needed to achieve the same results via a "brute-force" computation [10, 11, 29, 30, 31]. For $N = 2^{30}$ sources/receivers and a computation time of 21.6 s (total run), which we observed for 32 nodes with two GPUs per node, the brute-force algorithm would achieve a performance of **1.25 Exaflops**.

7. CONCLUSION

Algorithmic improvements presented in this paper substantially extend the scope of application of the FMM for large problems, especially for dynamic problems, where performance on data structure computations became a bottleneck. The heterogeneous algorithm substantially simplifies the GPU job and makes it much more efficient. It also provides a full load on the CPUs, enables efficient double precision computations, and brings other benefits of parallel use of the CPU cores and GPUs. With the present algorithm, dynamic problems of the order of ten million particles per GPU can be solved in a few seconds per time step, extending capabilities of single workstations equipped with one or several GPUs and relatively small (several node) low-cost heterogeneous clusters.

Weak and strong scalability of the algorithm for small and midsize clusters was demonstrated. A FMM run for $2^{30} \gtrsim 1$ billion particles was performed with this algorithm on a midsize cluster (32 nodes with 64 GPUs). A more advanced fully scalable algorithm for large clusters is presented.

Perhaps our biggest contribution is the use of the neglected resource (CPU) in heterogeneous CPU/GPU environments, allowing a significant improvement in hardware utilization. A user with a gaming PC worth less than $3000 (2 graphics cards and two-quad-core CPUs plus 16 GB RAM) can achieve FMM performance comparable with the 2009 Bell prize winner, at one percent of the cost and power, and a performance of 405 MFlops/dollar. Our algorithm can tackle two orders of magnitude larger problems of interest in fluid, molecular and stellar dynamics due to vastly improved handling of data structures and algorithmic improvements compared to current state-of-the-art algorithms. It also scales well, allowing the user to add more CPU cores and GPU cards to further improve the price/performace ratio. The Chimera machines cost $220K and achieve a performance of 177 MFlops/dollar—the best performance reported so far on the FMM [10].

Acknowledgements

Work partially supported by Fantalgo, LLC; by AFOSR under MURI Grant W911NF0410176 (PI Dr. J. G. Leishman, monitor Dr. D. Smith); in addition NG was partially supported by Grant G34.31.0040 (PI Dr. I. Akhatov) of the Russian Ministry of Education & Science. We acknowledge NSF award 0403313 and NVIDIA for the Chimera cluster at the CUDA Center of Excellence at UMIACS. We thank UMIACS staff for their help, Dr. M. Tiglio for access to Lincoln via Teragrid allocation TG-PHY090080, and Dr. D. Zotkin for help with editing.

8. REFERENCES

[1] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Phys.*, vol. 73, pp. 325–348, December 1987.

[2] V. Rokhlin, "Diagonal forms of translation operators for the helmholtz equation in three dimensions," *Appl. and Comp. Harmonic Analysis*, vol. 1, pp. 82–93, 1993.

[3] N. A. Gumerov and R. Duraiswami, *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions.* Oxford: ELSEVIER, 2005.

[4] L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *J. Comput. Phys.*, vol. 196, pp. 591–626, May 2004.

[5] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," *Computers Mathematics with Applications*, vol. 20, no. 7, pp. 63–71, 1990.

[6] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta, "A parallel adaptive fast multipole method," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, (New York, NY, USA), pp. 54–65, ACM, 1993.

[7] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *J. Parallel Distrib. Comput.*, vol. 27, pp. 118–141, 1995.

[8] S.-H. Teng, "Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation," *SIAM J. Sci. Comput.*, vol. 19, pp. 635–656, March 1998.

[9] N. A. Gumerov and R. Duraiswami, "Fast multipole methods on graphics processors," *J. Comput. Phys.*, vol. 227, pp. 8290–8313, September 2008.

[10] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "42 Tflops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 1–12, ACM, 2009.

[11] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka, "Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence," *Computer Physics Communications*, vol. 180, no. 11, pp. 2066–2078, 2009.

[12] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 58:1–58:12, ACM, 2009.

[13] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pp. 1–12, 2010.

[14] A. Chandramowlishwarany, K. Madduri, and R. Vuduc, "Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[15] Q. Hu, M. Syal, N. A. Gumerov, R. Duraiswami, and J. G. Leishman, "Toward improved aeromechanics simulations using recent advancements in scientific computing," in *Proceedings 67th Annual Forum of the American Helicopter Society*, May 3–5 2011.

[16] D. A. Griffiths, S. Ananthan, and J. G. Leishman, "Predictions of rotor performance in ground effect using a free-vortex wake model," *Journal of the American Helicopter Society*, vol. 49, October 2005.

[17] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618–2640, 2007.

[18] J. Makino, T. Fukushige, M. Koga, and K. Namura, "GRAPE-6: Massively-parallel special-purpose computer for astrophysical particle simulations," *Publication of Astronomical Society of Japan*, vol. 55, pp. 1163–1187, December 2003.

[19] N. A. Gumerov and R. Duraiswami, "Comparison of the efficiency of translation operators used in the fast multipole method for the 3D Laplace equation," Tech. Rep. CS-TR-4701, UMIACS-TR-2005-09, University of Maryland Department of Computer Science and Institute for Advanced Computer Studies, April 2005.

[20] C. A. White and M. Head-Gordon, "Rotating around the quartic angular momentum barrier in fast multipole method calculations," *The Journal of Chemical Physics*, vol. 105, pp. 5061–5067, September 1996.

[21] L. Greengard and V. Rokhlin, "A new version of the fast multipole method for the Laplace equation in three dimensions," *Acta Numerica*, vol. 6, pp. 229–269, 1997.

[22] H. Cheng, L. Greengard, and V. Rokhlin, "A fast adaptive multipole algorithm in three dimensions," *J. Comput. Phys.*, vol. 155, pp. 468–498, November 1999.

[23] G. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, pp. 1526–1538, 1987.

[24] H. Samet, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[25] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," in *IBM Germany Scientific Symposium Series*, 1966.

[26] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3* (H. Nguyen, ed.), ch. 39, pp. 851–876, Addison Wesley, August 2007.

[27] E. Darve, C. Cecka, and T. Takahashi, "The fast multipole method in parallel clusters, multicore processors and graphic processing units," *Comptes Rendus Mecanique*, vol. 339, pp. 185–193.

[28] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, and T. Sterling, “Pentium pro inside: I. a treecode at 430 gigaflops on ASCI Red, ii. price/performance of $50/Mflop on Loki and Hyglac,” in *Proc. Supercomputing 97*, in CD-ROM. IEEE, 1997.

[29] M. S. Warren, T. C. Germann, P. Lomdahl, D. Beazley, and J. K. Salmon, “Avalon: An Alpha/Linux cluster achieves 10 Gflops for $150k,” in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 1–10, 1998.

[30] A. Kawai, T. Fukushige, and J. Makino, “$7.0/mflops astrophysical N-body simulation with treecode on GRAPE-5,” in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing ’99, (New York, NY, USA), pp. 1–6, ACM, 1999.

[31] A. Kawai and T. Fukushige, “$158/Gflops astrophysical N-body simulation with reconfigurable add-in card and hierarchical tree algorithm,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC ’06, (New York, NY, USA), pp. 1–8, ACM, 2006.