A massively parallel adaptive fast-multipole method on heterogeneous architectures

Ilya Lashuk*, Aparna Chandramowlishwaran*, Harper Langston*,
Tuan-Anh Nguyen*, Rahul Sampath*, Aashay Shringarpure*,
Richard Vuduc*, Lexing Ying†, Denis Zorin‡, and George Biros*
* Georgia Institute of Technology, Atlanta, GA 30332
† University of Texas at Austin, Austin, TX 78712
‡ New York University, New York, NY 10002
ilashuk@cc.gatech.edu, aparna@cc.gatech.edu, harper@cc.gatech.edu,
tuananh.nguyen@gatech.edu rahul.sampath@gmail.com, aashay.shringarpure@gmail.com,
richie@cc.gatech.edu, lexing@math.utexas.edu, dzorin@cs.nyu.edu, gbiros@acm.org

***Abstract*—We present new scalable algorithms and an implementation of the kernel-independent fast multiple method (KIFMM), employing hybrid distributed memory message passing (via MPI) and shared memory/streaming using graphics processing unit (GPU) acceleration to rapidly evaluate two-body non-oscillatory potentials. On traditional CPU-only systems, our implementation scales well up to 30 billion unknowns on 65k cores (AMD/CRAY-based Kraken system at NSF/NICS) on tree data structures with 25 levels between leaves. On GPU-enabled systems, we achieve $30\times$ speedup for problems of up to 256 million points on 256 GPUs (Lincoln at NSF/NCSA) over a comparable CPU-only implementation. Both of these demonstrations represent the largest and fastest of their kind of which we are aware.**

We achieve scalability at extreme core counts by extending the initial work of Ying et al. (ACM/IEEE SC '03) with a new approach to scalable MPI-based tree construction and partitioning. For the sub-components of KIFMM, which direct- and approximate-interactions, target evaluation, and source-to-multipole translations, we use CUDA-based GPU-acceleration to achieve excellent performance. To do so requires carefully constructed data structure transformations, which we describe, and whose cost we show is minor. Taken together, these components show promise for ultrascalable FMM in the petascale era and beyond.

SC09 November 14-20, 2009, Portland, Oregon, USA (c) 2009 ACM 978-1-60558-744-9/09/11 $10.00

I. Introduction

We consider the problem of rapidly evaluating sums of the form,

$$f_i = f(x_i) = \sum_{j=1}^{N} K(x_i, y_j)s(y_j) = \sum_{j=1}^{N} K_{ij}s_j,\ i = 1, \ldots, N, \tag{1}$$

which expresses a need to compute $\mathcal{O}(N^2)$ interactions in a system of N particles. We refer to f as the *"potential"*, s the *"density"*, x the target points, y the source points, and K the interaction kernel. For example, for electrostatic or gravitational interactions $\mathrm{K}(x,y) = \frac{1}{4\pi}\frac{1}{\|x-y\|_2}$. Such problems, in which we wish to compute pairwise interactions in a particle system, arise throughout mathematical physics and are often related to the fundamental solutions (kernels) of partial differential equations (PDEs).

By "rapid evaluation," we imply an asymptotic time complexity of $\mathcal{O}(N)$, in contrast to a direct evaluation requiring $\mathcal{O}(N^2)$ operations. This speedup is possible for a large class of kernels and can be performed using ideas based on the original fast multipole method algorithm of Greengard and Rokhlin [6]. The trade-off is that FMM algorithms introduce an approximation error. The constant in the complexity estimate is related to the desired accuracy of the approximation and depends on the details of the algorithm.

Outline of the method. This paper extends prior work by us and others by developing and evaluating parallel implementation of the FMM on hybrid distributed memory CPU/GPU architectures. It uses three well-known algorithmic frameworks:

- *Sequential FMM method.* Our implementation is based on the *kernel independent FMM* [20], [21], and is particularly effective for non-oscillatory kernels.
- *Distributed memory parallelism using MPI:* Our algorithms use space filling curves and the notion of *Local Essential Trees* (LET) [18]. For each process, the LET is the subtree of the global FMM tree that this process needs in order to evaluate the interaction on particles it owns.
- *Streaming parallelism:* We use GPU acceleration of the per-octant calculation of the FMM operators. Our implementations employ the NVIDIA CUDA programming model [1].

These three frameworks form the foundation of our method, which can be summarized as follows: given the points (equi-distributed in an arbitrary way across MPI processes) and their associated source densities, we seek to compute the potential at each point (for simplicity in this paper we assume that source and target points coincide). First, we sort the points in Morton ordering and redistribute them so that each process owns a contiguous chunk of the sorted array. Second, we create the LET for each process in parallel. Third, we evaluate the sums using the LETs across MPI processes and using GPU acceleration independently within each process. In our scheme, each MPI process is assumed to have private access to an accelerator. We have not employed CPU multithreading within an MPI process.

Contributions: Starting from the frameworks of FMM, MPI/GPU, and LET, we introduce the following algorithmic improvements:

- We give a new parallel tree construction algorithm that is based on our work for linear octrees for finite elements [16]. In our previous FMM implementation, the tree construction was over 15 times slower than the evaluation part, on 3000 cores [21]. By contrast, experimental results on up to 65,536 cores show that the tree construction in our new implementation is only 10% of the evaluation phase.
- We present a novel all-reduce-on-hypercube scheme for the communication during the evaluation phase. Our previous implementations, based on `alltoallv()` exchanges or unoptimized point-to-point non-blocking communication, greatly taxed MPI system resources for problems with more than 32K cores. We give theoretical and experimental evidence that shows that our new implementation circumvents this bottleneck.
- We integrate GPU acceleration of the direct- and approximate-interaction evaluations in the FMM phase. Unlike MPI- or GPU-only implementations, here we need to translate between different data structures. We use linear arrays for the tree construction, pointers for the LETs, and a streaming-friendly data structure for the GPU; the translation has a somewhat high-memory footprint, but we show that it can be accomplished efficiently. Finally, the GPU acceleration, which is specific to KIFMM, is novel.

We have conducted weak and strong scalability studies on the NSF Kraken system on up to 30 billion unknowns on 65K cores, for a highly nonuniform (over 20 levels difference between coarse and fine leaves) tree; and we have tested the algorithm on up to 256 GPUs (NVIDIA Tesla S1070) for 256 million unknowns.

Limitations: Our new implementation has certain limitations. First, the GPU acceleration is implemented in single precision (the rest of the code can work in both single and double precision). Second, we do not consider GPU acceleration of the entire computation, and in particular omit the tree-construction and the certain parts for the evaluation phase (we discuss this later). Third, to load balance we use Morton curves and work estimates (at the leave-level), whereas more sophisticated schemes could be used. Finally, we do not thoroughly overlap computation and communication; in the GPU-accelerated version, we do not presently invoke the GPU-accelerated phases fully asynchronously, and therefore do not exploit the possibility of overlapping GPU evaluation with work on the CPU.

Related work: To our knowledge, no FMM that supports highly nonuniform distribution of multi-billion particles has been parallelized and scaled successfully beyond 3000 processes. Most implementations, including ours, are based on the work of Warren and Salmon [18]. In Ying *et al.* [21] (Table 4.2), the time consumed by the tree generation for 2048 processes is already unsatisfactory. Other recent publications about parallel FMM, *e.g.*, Kurzak [10] and Ogata [11], do not handle nonuniform distributions of particles and do not report results for more than 1024 processes. See Ying et al. for a review of the literature prior to 2003 [21]. Additional surveys and work on tree-partitioning, efficient data structures, and discussions on the theory of partitioning and complexity can be found elsewhere [8], [15], [17].

Several groups have been working on GPU acceleration. For a review and details on a particular GPU implementation (without MPI), see Gumerov et al. [7], in which 30–60$\times$ speedups were achieved on the evaluation phase. Other authors consider GPU-accelerated tree construction [2]. NVIDIA has an N-body example for direct summation that can be used as template for most GPU implementations [1]. To our knowledge, the only heterogeneous attempt for N-body algorithms can be found in Phillips, et al. [12], but the problem sizes considered in that work were very small (98k particles).

II. High-level description of the fast multipole algorithm

Here, we describe the *sequential* FMM without any details on how the different parameters need to be chosen for a given numerical accuracy. We refer to [20] and the references therein for further details. As mentioned before, we assume that the set of source points y_j and the set of the target points x_i coincide. In Table I, we introduce the notation used throughout the rest of the paper.

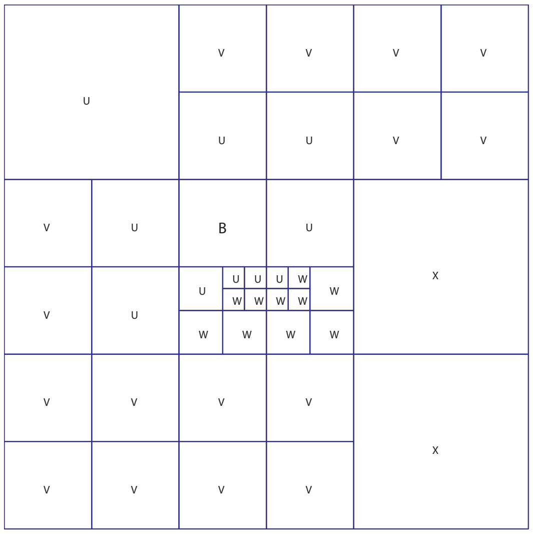


Fig. 1. FMM lists. *Here we show a canonical example of the U,V,W, and X lists in two dimensions for a tree node B. One can verify that $\mathcal{I}(B)$ is inside the domain enclosed by $\mathcal{C}(P(B))$.*

α, β	octants
ℓ	tree level
q	maximum number of points/octant
Vectors	
x	target points
y	source points
s	densities at the source points
f	potential at the target points
u	upward densities
d	downward densities
Linear operators	
K	direct source density-to-target potential
S	source-to-up density
U	up-to-up density
D	down-to-down density
E	down density-to-target potential
Q	source-to-down density
R	up density-to-target potential
T	up-to-down density
Octant lists	
T	the global FMM tree
L	all leaf octants
$P(\beta)$	parent octant of β ($\emptyset$ for root of T)
$\mathcal{A}(\beta)$	ancestor octants of β
$\mathcal{K}(\beta)$	children octants of β ($\emptyset$ for $\beta \in T$)
$\mathcal{D}(\beta)$	descendant octants of β in T
$\mathcal{C}(\beta)$ (colleagues)	adjacent octants to β, same level
$J(\beta)$	adjacent octants to β (arbitrary level)
U-list:$\alpha \in \mathcal{U}(\beta)$	$\alpha \in L$ adjacent to β (note:$\beta \in \mathcal{U}(\beta)$)
V-list:$\alpha \in \mathcal{V}(\beta)$	$\alpha \in \mathcal{K}(\mathcal{C}(P(\beta))) \setminus \mathcal{C}(\beta)$
W-list:$\alpha \in \mathcal{W}(\beta)$	$\alpha \in \mathcal{D}(\mathcal{C}(\beta)) \setminus J(\beta)$, $P(\alpha) \in J(\beta)$
X-list:$\alpha \in \mathcal{X}(\beta)$	iff $\beta \in \mathcal{W}(\alpha)$
$\mathcal{I}(\beta)$ (interaction)	$\mathcal{V}(\beta) \cup \mathcal{U}(\beta) \cup \mathcal{W}(\beta) \cup \mathcal{X}(\beta)$

TABLE I

Notation. *The lists $\mathcal{U}(\beta)$ and $\mathcal{W}(\beta)$ are defined only for $\beta \in L$, whereas $\mathcal{V}(\beta)$, $\mathcal{X}(\beta)$ are defined for all leaf and non-leaf octants β. Note that $\alpha \in \mathcal{W}(\beta)$ need not be a leaf octant; conversely, since $\mathcal{W}(\beta)$ exists only when $\beta \in L$, $\alpha \in \mathcal{X}(\beta)$ implies that $\alpha \in L$. We say that α is adjacent to β if β and α share a vertex, an edge, or a face. Also,* U, D *depend only the box,* T *depends on pair of boxes and* E, S *depend on targets and sources. boxes, and* E, S *depend on source and targets. See Figure 1, for an example of the W,X,V, and U lists in two dimensions.*

The FMM tree T can be thought of as constructed in a top-down fashion (actual parallel algorithms are discussed below). Consider a user-specified parameter q, which sets the maximum allowable number of points per box. We start at the unit cube (suppose it contains all the points), and we subdivide an octant if it has more than q points.

For each octant we create four lists of octants, the so called U-,V-,W- and X-list. Octants in these lists "interact" (i.e., are coupled) with β in the course of FMM computation. For a leaf octant β, the U-list of β consists of all leaf octants adjacent to β, including β itself. (We say that α is adjacent to β if β and α share a vertex, an edge, or a face.) The V-list of an octant β (leaf or non-leaf) consists of those children of the colleagues of β's parent octant, $P(\beta)$, which are not adjacent to β. The W-list is only created for a leaf octant β and contains an octant α if and only if α is a descendant of a colleague of β, α is not adjacent to β, and the parent of α is adjacent to β. The X-list of an octant β consists of those octants α which have β on their W-list. These definitions are summarized in the bottom of Table I.

For each octant $\beta \in T$, we store its level ℓ, and two vectors u and d. u is to be understood as a compressed approximate representation of the potential generated by the densities in β. This representation is sufficiently accurate only if the evaluation point is *outside the volume covered by* β *and the colleagues of* β. The d vector is to be understood as some compressed approximate representation of the potential generated by the densities *outside the volume covered by* β *and colleagues of* β. This approximate representation is sufficiently accurate only if the evaluation point is *enclosed by* β.

For leaf octants ($\beta \in L$), we also store x, s, and f, referring to Table I for definitions. Table I also introduces several linear operators. These are typically small matrices (dimension between 100-1000) whose precise interpretation depends on the particular FMM implementation. Their exact definition is beyond the scope of this paper and not necessary in our context. We simply note that the main difficulty in FMM is in deriving the efficient mathematical representation for these operators in order to obtain algorithmic efficiency without significantly compromising accuracy [20]. Let us also remark that the algorithm supports several other constructions of translation operators, including analytic methods described in [4].

Given the above definitions, approximately evaluating the sum in Equation (1) involves an upward computation pass of T, followed by a downward computation pass of T, seen more specifically in Algorithm 1.

Let us clarify that all the computations in Algorithm 1 can be viewed as simple matrix-vector multiplications (e.g., $u_\beta = \mathrm{S}_\beta\, s_\beta$ means convert the source densities s of the leaf octant β to its up-densities u by multiplying S with s; the subscript notations in Algorithm 1 denote octant dependence).

A. Parallelizing the evaluation phase

Here, we will make some generic observations about the concurrency of the evaluation phase. These observations are valid for many types of FMM. (This not the case for the tree construction.)

Algorithm 1 is presented as a sequential algorithm. The discussion of its parallelization can be separated to distributed and shared memory viewpoints. These viewpoints can be then connected to MPI, OpenMP, and GPU application programming interfaces.

Algorithm 1: FMM

```
Input: $\{x_i, y_i, s_i\}_{i=1}^N$, octree
Output: $\{f_i\}_{i=1}^N$

// APPROXIMATE INTERACTIONS
// (1) S2U: source-to-up step
  $\forall \beta \in L:\ u_\beta = \mathrm{S}_\beta\, s_\beta$
// (2) U2U: up-to-up step (upward)
  Postorder traversal of $T$
    $\forall \beta \in T:\ u_{P(\beta)} \mathrel{+}= \mathrm{U}_\beta\, u_\beta$;
// (3a) VLI: V-list step
  $\forall \beta \in T:\ \forall \alpha \in \mathcal{V}(\beta):\ d_\beta \mathrel{+}= \mathrm{T}_{\beta\alpha}\, u_\alpha$;
// (3b) XLI: X-list step
  $\forall \beta \in T:\ \forall \alpha \in \mathcal{X}(\beta):\ d_\beta \mathrel{+}= \mathrm{Q}_{\beta\alpha}\, s_\alpha$;
// (4) D2D: down-to-down step (downward)
  Preorder traversal of $T$
    $\forall \beta \in T:\ d_\beta \mathrel{+}= \mathrm{D}_{\beta P(\beta)}\, d_{P(\beta)}$;
// (5a) WLI: W-list step
  $\forall \beta \in L:\ \forall \alpha \in \mathcal{W}(\beta):\ f_\beta \mathrel{+}= \mathrm{R}_{\beta\alpha}\, u_\alpha$;
// (5b) D2T: down-to-targets step
  $\forall \beta \in L:\ f_\beta \mathrel{+}= \mathrm{E}_\beta\, d_\beta$;

//DIRECT INTERACTIONS
// ULI: U-list step (direct sum)
  $\forall \beta \in L:\ \forall \alpha \in \mathcal{U}_\beta:\ f_\beta \mathrel{+}= \mathrm{K}_{\beta\alpha}\, s_\alpha$;
```

There are multiple levels of concurrency in FMM: across steps (e.g., the S2U and ULI steps have no dependencies), within steps (e.g., a reduction on octants α in V-list of an octant β during the VLI step), and in the per-octant calculations (e.g., vectorizing the $Q_{\beta\alpha}s_\alpha$ calculation in the XLI step).[1]

The generic dependencies of the calculations outlined in Algorithm 1 are as follows: The APPROXIMATE INTERACTIONS and DIRECT INTERACTIONS parts can be executed concurrently. For the approximate interaction calculations, the order of the steps denotes dependencies, e.g., step (2) must start after step (1) has been completed. Steps (3a) and (3b) can be executed in any order. However, concurrent execution of steps (3a) and (3b) requires concurrent write with accumulation. This is also true for the steps (5a) and (5b), that is they are independent up to

[1]In fact, one can expose more concurrency by using pipelining (e.g., one can pipeline the S2U, U2U, and VLI steps to expose more parallelism). However, we do not take advantage of this concurrency in this work as such an approach reduces the modularity and generality of the implementation.

concurrent writes.

Our overall strategy is to use MPI-based distributed memory parallelism (that also addresses the tree construction) to partition the trees into LETs to remove dependencies between steps (3a)/(5a) and (3b)/(5b) and by handling the concurrent writes explicitly) and then use shared-memory based parallelism on GPUs within each MPI-process to accelerate the direct interactions and steps (1), (3), (5) of the indirect interactions in Algorithm 1. In the following sections, we give the details of our approach.

III. Distributed memory parallelism

In this section, we present distributed memory algorithms for our FMM method. The main components of our scheme are (1) *the tree construction*, in which the global FMM tree is built and each process receives its local essential tree and a set of leaves for which it assumes ownership; and (2) *the evaluation*, in which each process evaluates the sum at the target points of the leaf octants it owns. This sum has two components, a direct-interaction (and exact) evaluation and an approximate-interaction evaluation.

The input consists of the source points and their densities. In our analysis and all of our experiments, these points are assumed to be equally-distributed randomly across all processes. The output of the algorithm is the potential at the target points. The final distribution of the points is determined by the algorithm.

Before we describe the algorithm we need the following definition:

Locally essential tree (LET): Given a partition of L across processes so that L_k is the set of leaf-octants assigned to the process k (i.e., the potential in these octants is computed by process k), the LET for process k is defined as the union of the interaction lists of all owned leaves and their ancestors:

$$\mathrm{LET}(k) := \cup_{\forall \beta \in [L_k \cup \mathcal{A}(L_k)]} \mathcal{I}(\beta).$$

The basic idea [18] for a distributed memory implementation of the FMM algorithm is to partition the leaves of the FMM tree across processes, construct the LET of each process and then compute the N-body sum in parallel. There are two communication intensive phases in this approach: the first phase is the LET construction and the second phase is an all-reduce, which is required to ensure correctness of the approximate interaction computations. Next, we discuss the main components in these two phases.

A. Tree construction

In our original implementation of the parallel FMM, we used a lightweight copy of the entire global tree on each process. This approach became problematic above 2048 MPI-processes and highly inefficient in the 3000-processes case. Alternative approaches for tree construction are well-known, for example [15]. To our knowledge however, no scalability results for large core counts have been reported in the literature; hence, it is difficult to assess the real performance of alternative approaches. Here, we use a bottom-up variant of those algorithms that uses infrastructure that we have developed in previous work in our group for linear finite element octrees [14].

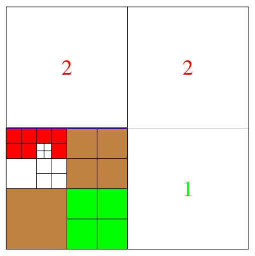
The input in the tree construction are the target and source points (which we treated as a single set of points). The output is the local essential tree on each process, which is subsequently used in the computation, along with geometrical domain decomposition of the unit cube across processes. The latter is used throughout the algorithm. The tree construction involves (1) the construction of a distributed linear complete octree that contains only the leaf octants; and (2) the construction of the per-process LETs.

We start by creating the distributed and globally Morton-sorted array containing all the leaves from the global tree. To build this array, we use the `PointsToOctree` method, which is part of our `DENDRO` package, whose description can be found in [16] (Algorithm 5). This routine generates a distributed linear octree, which is Morton-order sorted globally.

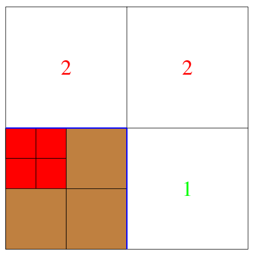
We should mention that the current version of DENDRO package can, in theory, produce octrees which are finer than necessary (some octants containing less than q points may be subdivided in specific circumstances). This is due to the fact that DENDRO was developed for the case $q = 1$, and we use it for general q. In practice, we do not observe any performance problems related to possible extra subdivisions.

The distribution of the leaves between processes induces a geometric partitioning of the unit cube: each process controls the volume covered by the leaves it owns. Each process has complete information about the overall geometric partitioning (processes communicate via `MPI_AllGather` to exchange this information). By Ω_k, we will denote the region "controlled" by process k.

Next, each process adds all ancestor octants to its local leaves, thus, creating a local tree. The task of exchanging information about "ghost" octants must still be completed.



(a) finer level of octree



(b) coarser level of octree

Fig. 2. Communication of ghost octants. *Process 0 sends green octants to process 1, red octants to process 2, and brown octants to both 1 and 2. White octants in lower-left corner are "internal" to process 0 and not sent to anyone. The procedure is applied to both leaves and non-leaf octants.*

ALGORITHM 2: LET CONSTRUCTION

Input: distributed set of points x;
Output: LET on each process

```
1. $L_k$ = Points2Octree($x$)                         //MPI
2. $B_k = L_k \cup \mathcal{A}(L_k)$
3. $I_{kk'} := \mathcal{C}(B_k) \cap \Omega'_k$
4. $\forall k' : \; k' < p \;\&\&\; k'! = k$;
     Send $I_{kk'}$ to process $k'$                   //MPI
     Recv $I_{k'k}$ from process $k$                  //MPI
     Insert $I_{k'k}$ in $B_k$
5. Return $B_k$
```

Let us introduce the following definitions:

- *"Contributor"* processes to an octant $\beta \in T$: $\mathcal{P}_c(\beta) := k \in 1 \ldots p : \beta$ overlaps with Ω_k
- *"User"* processes to an octant β: $\mathcal{P}_u(\beta) := k \in 1 \ldots p : \mathcal{C}(P(\beta))$ overlaps with Ω_k.

Let $I_{kk'}$ be the set of octants to which process k contributes and which process k' uses. Process k sends all octants in $I_{kk'}$ to process k'. Figure 2 provides an illustration of this process. Then, all processes insert received octants into their local trees. This concludes the construction of the per-process LETs. We summarize the construction of the local essential trees in Algorithm 2.

Remark: *Only the leaves of T are assigned "ownership" across processes.*

To prove the correctness of the LET construction, we need to show that for every $\beta \in B_k$ all of the octants $\alpha \in \mathcal{I}(\beta)$ are in the LET of the process k. Indeed, if $\alpha \in \mathcal{I}(\beta)$ then $\beta \in \mathcal{I}(\alpha)$ because the U- and V-lists are symmetric[2] and the union of W and X lists is also symmetric.

Thus, $\mathcal{I}(\alpha) \cap \Omega_k \neq \{\emptyset\}$. Assuming that $\alpha \in \Omega_{k'}$ and $k' \neq k$, then $\alpha \in I_{k'k}$. Hence, α has been sent to process k from process k' and has been inserted to LET$_k$. (If $k = k'$, then $\alpha \in B_k$ by construction.)

In the last step of Algorithm 2, every process independently builds U,V,W, and X-lists for those octants in its LET which enclose the local points (where the potential is to be evaluated). All necessary octants are already present in the LET, and no further communication is required in this step.

B. Load balancing

Work-per-leaf-based partitioning: Assigning each process an equal chunk of leaves may lead to a substantial load imbalance during the interaction evaluation for nonuniform octrees. In order to overcome this difficulty, we use the following load-balancing algorithm.

After the LET setup, each leaf is assigned a weight, based on the computational work associated with its U,V,W, and X-lists (which are already built at this point). Then, we repartition the leaves to ensure that total weight of the leaves owned by each process is approximately equal. We use Algorithm 1 from [16] to perform this repartitioning in parallel. Note that similarly to the previous section, each process gets a contiguous chunk of global (distributed) Morton-sorted array of leaves. In the final step, we re-build the LET and the U,V,W, and X-lists on each process.

Note that we repartition the leaves based solely on work balance ignoring the communication costs. Such an approach is suboptimal, but is not expensive to compute and works reasonably well in practice. In addition to leaf-based partitioning, the partitioning at a coarser level can also be considered; however, we have not tried it at this point.

C. FMM evaluation

Three communication steps are required for the potential evaluation. The first is to communicate the exact densities for direct calculation. This communication is "local" in a sense that each process typically communicates only with its "geometrical" neighbors. Thus a straightforward implementation (say, `MPI_Isend`) is acceptable.

The second communication step is to sum up the upward densities of all the contributors of each octant.

[2]If an octant α belongs to the U/V-list of octant β, then octant β belongs to the U/V-list of octant α.

The third step is to "broadcast" the complete densities to the users of each octant. These two steps must take place after the U2U step (bottom-up traversal) in which the *local* up-densities have been computed and before the VLI and XLI steps can take place.

In our previous work, we used a simple approach for these two communication steps. Each octant (both leaf and non-leaf) was assigned an owner process. The owner received the partial densities from all the contributors, summed them up and sent the result to each user. Such an approach worked well on up to 32K processes, but failed in the 64K case. Note that octants close to the root of the tree have numerous contributors and even more users (up to all the processes). This suggests using some tree-based reduction and broadcast scheme.

Algorithm 3 describes a communication procedure that we use now. The procedure combines second and third steps mentioned above. Remarks:
We assume that the size of the MPI communicator is a power of two. We term an octant to be "shared", if the union of its contributors $\mathcal{P}_c$ and users $\mathcal{P}_u$ contains more than one process.
Step 2: u_s and u_e define a range of processes (inclusive), to which octants sent to s may be eventually broadcasted. There is no need sending to s octants that are not destined for $u_s, \ldots, u_e$.
Step 5: q_s and q_e define range of processes (inclusive), to which octants from this process may be eventually broadcasted during remaining communication rounds. There is no need storing octants that are not destined for processes within the $q_s, \ldots, q_e$ range.

Time complexity of this algorithm is not worse than $\mathcal{O}(\sqrt{p})$. To be more specific, assuming that no process uses more than m shared octants and no process contributes to more than m shared octants, for a hypercube interconnect, the communication complexity of Algorithm 3 is $\mathcal{O}(t_s \log p + t_w m(3\sqrt{p} - 2))$, where t_s and t_w are the latency and the bandwidth constants, respectively. This bound can be derived as follows.

We have $d = \log p$ communication rounds, so the latency-related cost is $t_s \log p$. Now consider any communication round; i.e., fix some $i \in \{d-1, \ldots, 0\}$. During this round, the process r with binary representation $r = (b_{d-1} \cdots b_{i+1}\, 0\, b_{i-1} \cdots b_0)_2$ sends data to process $s = (b_{d-1} \cdots b_{i+1}\, 1\, b_{i-1} \cdots b_0)_2$. Note, that if some octant β is sent from r to s during the round, then two conditions are met:

- Some process with rank $(c_{d-1} \cdots c_{i+1} 0 b_{i-1} \cdots b_0)_2$ contributes to β. That is, the $i+1$ least significant binary digits of some contributor of β must be $0, b_{i-1}, \cdots, b_0$. (Note, there are at most 2^{d-i-1} such processes. Thus, at most $2^{d-i-1}m$ octants have this property.)
- Some process with rank $(b_{d-1} \cdots b_{i+1} 1 c_{i-1} \cdots c_0)_2$ uses β. That is, $d-i$ most significant binary digits of some user of β must be $b_{d-1}, \cdots, b_{i+1}, 1$. (At most $2^i m$ octants have this property, similar to previous condition).

The number of octants satisfying both conditions is not greater than $m \cdot \min(2^{d-i-1}, 2^i)$. Note that $\sum_{i=0}^{d-1} \min(2^{d-i-1}, 2^i) \le 3\sqrt{p} - 2$. Then the bound easily follows.

After the three communication steps, all the remaining steps of Algorithm 1 can be carried out without further communication.

ALGORITHM 3: REDUCE AND SCATTER

Input: partial upward densities of "shared" octants at "contributor" processes
Input: r–rank of current process; we assume communicator size is 2^d
Output: complete upward densities of "shared" octants at "user" processes

```
//Define shared octants (for each process):
$\mathcal{S} = \{\beta \in \text{LET} : \#(\mathcal{P}_u(\beta) \cap \mathcal{P}_c(beta)) > 1\}$
//Loop over communication rounds (hypercube dimensions)
For $i := d-1$ to $0$
  //Process $s$ is our partner during this communication round
  1. $s := r$ XOR $2^i$
  2. $u_s = s$ AND $(2^d - 2^i)$
  3. $u_e = s$ OR $(2^i - 1)$
  4. Send to $s$ : $\{\beta \in \mathcal{S} : \mathcal{I}(\beta) \cap (\cup_{u_s}^{u_e} \Omega_s) \neq \emptyset\}$
  5. $q_s = r$ AND $(2^d - 2^i)$
  6. $q_e = r$ OR $(2^i - 1)$
  7. Delete $\{\beta \in \mathcal{S} : \mathcal{I}(\beta) \cap (\cup_{q_s}^{q_e} \Omega_s) = \emptyset\}$
  // Reduction
  8. Recv from $s$ and append $\mathcal{S}$
  9. Remove duplicates for $\mathcal{S}$
  10. Sum up densities for duplicate octants.
```

D. Complexity

We have all the necessary ingredients to derive the overall complexity of the distributed memory algorithm. Let n be number of points and let p be number of processes. We will assume uniform distribution of points in the unit cube. This, in particular, implies that the number of *octants* is proportional to number of points. The main communication cost is associated with the parallel sort of the input points. Its time complexity is $\mathcal{O}(\frac{n}{p}\log\frac{n}{p} + p\log p)$ (combination of sample sort and bitonic sort) [5]. Exchanging the "ghost" octants has the same complexity as the reduce-broadcast algorithm described in Section III-C, i.e., $\mathcal{O}(\sqrt{p}m)$, where m is the maximal number of "shared" octants. For a uniform grid, m can be estimated as $\mathcal{O}\left(\frac{n}{p}\right)^{2/3}$. The communication also includes the exchange of exact densities. Their cost is of order $\mathcal{O}(m)$, since mostly geometrical neighbors communicate. This cost is negligible compared to $\mathcal{O}(\sqrt{p}m)$ term. The overall complexity of the setup phase is thus $\mathcal{O}\left(\frac{n}{p}\log\frac{n}{p} + p\log p + \sqrt{p}\left(\frac{n}{p}\right)^{2/3}\right)$. For the evaluation we have to add up the linear complexity of local FMM pass and the communication complexity of reduce-broadcast. We end up with $\mathcal{O}\left(\frac{n}{p} + \sqrt{p}\left(\frac{n}{p}\right)^{2/3}\right)$.

In the next section, we consider shared memory acceleration for the S2U,D2T,ULI, and VLI steps using a streaming architecture.

IV. The GPU acceleration

Naturally, we should consider shared memory acceleration and exploiting single-socket and streaming speed ups. The S2U,D2T, ULI, WLI, VLI, XLI steps can be implemented in parallel. Also, the U2U and D2D steps can be also executed in parallel using Euler tours [9], but as mentioned in the limitations discussion in the introduction, our current implementation does not support such parallelism.

In this section, we give details that are valid only for the specific FMM we're using—our kernel independent method [20]. We accelerate the S2U, VLI, ULI, and D2T steps. (The U2U and D2D traversals and XLI, WLI remain sequential.) The basic idea is the following. In all of the steps that we have accelerated with GPUs, visiting the octants can be done in an embarrassingly parallel way. The second observation is that all box visits include box-to-box or box-to-point interactions that are expressed in terms of matrix vector multiplications. All the multiplications are dense with the exception of the VLI calculations, which correspond to a diagonal translation. This allows a two-level parallelism: across boxes, and across the rows of the corresponding matrix.

In the CUDA programming model, there is a two-level thread hierarchy consisting of (i) individual threads and (ii) thread blocks [1]. A thread block consists of a group of b individual threads. Threads within thread block may explicitly synchronize and communicate via a local-store memory ("shared memory" in CUDA parlance), but two threads in different thread blocks cannot synchronize or communicate.

Algorithm 4: GPU_U-List

```
Input: $\mathcal{U}$, $s$;
Ouput: $f$;

1. Assign target box $\beta$ to some thread block B.
2. Assign each target point $t_i \in \beta$ to some thread $p_i \in$ B.
3. Transfer $\mathcal{U}$ and $s$ from CPU to GPU.
4. Each thread $t_i$ executes the following (in parallel):    //GPU
5.   $\forall \alpha \in \mathcal{U}(\beta)$;
       // Each thread $p_i$ loads a different source point $s_j$:
6.     Load source point $s_j \in \alpha$ into B's shared memory;
7.     SynchronizeThreads(B);
8.     $\forall s_j \in$ B's shared memory: $f_i$ += $K_{i,j} \cdot s_j$
9.     SynchronizeThreads(B);
10. Transfer $f$ from to GPU to CPU
```

The CUDA model also includes a memory hierarchy, consisting of a global address space on the GPU that all threads share, as well as a local-store address space shared only among threads within a given thread block. When reading from the global address space, current generation GPUs strongly favor *coalesced* memory accesses in which all threads within a thread block reference contiguous consecutive memory addresses. For data residing in the local-store, threads may access data in a random-access fashion with no penalty.

We sketch the GPU U-List implementation in Algorithm IV. First, we copy the U-List data structure, $\mathcal{U}$, into a GPU-friendly format in which target boxes are padded to the next largest multiple of the thread block size, b. We then assign groups of b target points to thread blocks and assign individual target points to individual threads. All threads cooperatively load blocks of b source points into shared memory, accumulate the potential due to those source points, and repeat until no source points remain. The U-List is sparse, meaning that loading source point data may not always be coalesced, depending on the number of source points per source box. Since we perform $O(b^2)$ flops for every $O(b)$ loads, we do not expect this effect to be large for sufficiently large b.

Although our direct computation is similar to the CUDA reference implementation [1], there are two key differences. First, we use a sparse adjacency structure (the U-List, $\mathcal{U}$). Second, we avoid self-interactions not by using a softening parameter, but by exploiting the IEEE-compliant implementation of the $\max(a, b)$ function available on the GPU. In particular, in IEEE arithmetic, $\max(\texttt{NaN}, x) = \max(x, \texttt{NaN}) = x$. If the distance between two particles is 0 so that the potential $x = \infty$, then $x = x+(x-x) = \texttt{NaN}$ and $\max(x, 0.0) = 0.0$. The $\max$ function is implemented efficiently on the GPU, making this trick an effective way to avoid self-interactions without a conditional test.

The accelerations for the S2M and L2T lists follow a similar pattern and in fact are even easier to implement. The main difference with the U2U list is that, in some sense, the locations of either the targets (S2M) or the sources (L2T) are in known regular positions per octant, and thus they do not need to be read in the memory. Instead, given the octant level and coordinates (typically, just one of its vertices) we can produce the coordinates of the target/source points using information that is permanently resident in the shared memory of the blocks. This minimizes memory fetches and allows for over 50X speed-up for those phases (over 30 GFlops/s).

The calculation for the V-list is different: in our FMM scheme, it is diagonal, that is it corresponds to a pointwise vector-vector multiplication. It is based on a Fast Fourier Transform-based diagonalization of the T operator. In our current implementation, the per-octant FFTs are done in the CPU and the diagonal translation (in the frequency space) is done in the GPU. This calculation is the least efficient in the GPU as it the ratio between computation and memory fetches is small. Our ongoing work includes transferring the W,X-lists on the GPU.

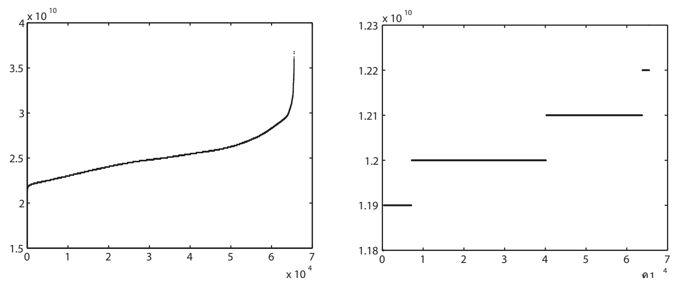


Fig. 5. Variance of Flops across processes. Left: nonuniform; Right: uniform *Here we see the variability of flops across processes. On the x-axis we have the process id. Notice the different scales on the y-axis between the uniform and nonuniform case.*

Event	Max. Time	Avg. Time	Max. Flops	Avg. Flops
Total eval	1.37e+02	1.20e+02	5.48e+10	3.72e+10
Upward	3.83e+01	1.85e+01	1.69e+10	7.68e+09
Comm.	8.83e+00	8.83e+00	0.00e+00	0.00e+00
U-list	5.84e+01	2.67e+01	1.61e+10	9.57e+09
V-list	4.73e+01	2.63e+01	2.06e+10	1.15e+10
W-list	1.63e+01	5.47e+00	4.43e+09	2.26e+09
X-list	1.28e+01	5.13e+00	4.25e+09	2.22e+09
Downward	1.89e+01	9.06e+00	8.74e+09	3.97e+09
Comp	1.18e+02	9.11e+01	5.48e+10	3.72e+10

TABLE II
65,536 PROCESSES ON KRAKEN. RESULTS FROM THE TIMING THE EVALUATION PHASE. *In this example, the problem size was 150K points per process. Since we are using the Stokes kernel with three unknowns per point, a total of 30 billion potentials on TeraGrid/NICS Kraken were computed. The tree used in this calculation spanned seven orders of spatial scales (the coarsest leaf is at level 2 and the finest tree is at level 27). The setup took 27 seconds, 15 of which were spent in the particle sort. Here, "Max. Time" is maximum wall-clock time across processes; "Avg. Time" is the wall-clock time averaged across processes. "Max" and "Avg" are defined similarly for the "Flops" numbers.*

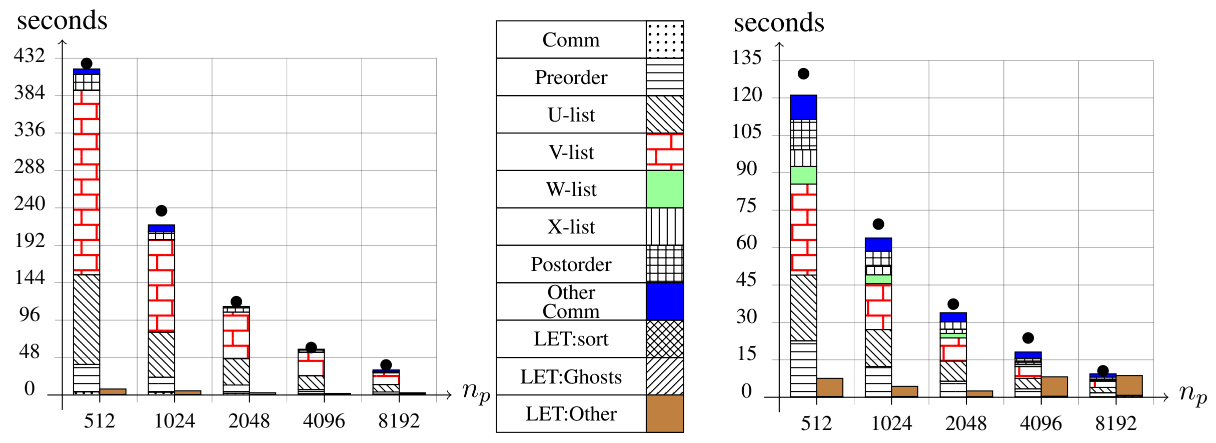
V. Numerical Experiments

In this section, we describe the results from numerical experiments that demonstrate the scalability of our implementation across different architectures and for different problems. Below we summarize the different parameters in our tests.

Particle distributions. We test two particle distributions, a *uniform* and a *nonuniform* one. The uniform corresponds to random sampling with uniform probability density distribution on the unit cube. The nonuniform distributions corresponds to a distribution of points on the surface of an ellipsoid of ration 1:1:4 with uniform distribution of angle spacing in spherical coordinates.

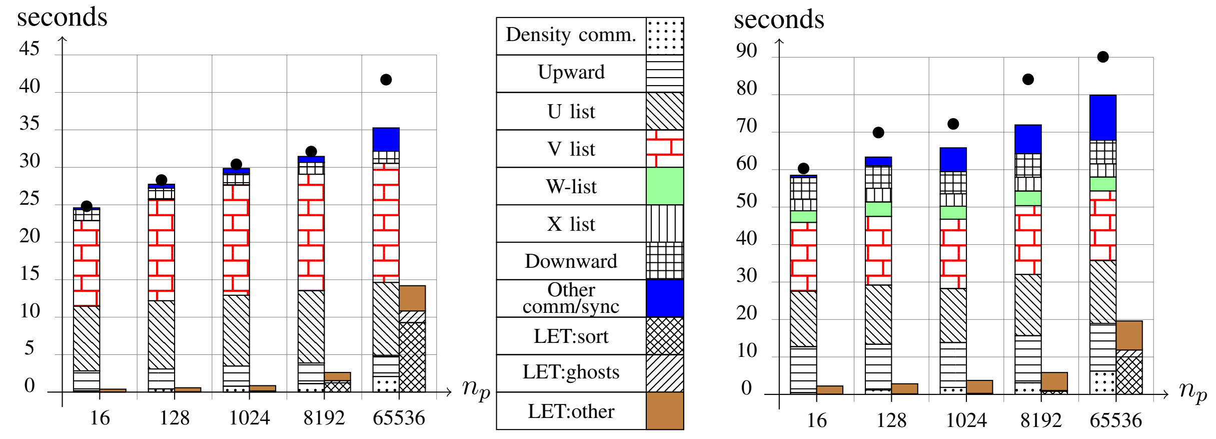
Machines. The main scalability results have been obtained on TeraGrid's *Kraken* at the National Institute of Computational Sciences (UT/ORNL/NSF), a Cray XT5 system with 66048 cores (2.3 GHz quad-core AMD opteron, 1/2GB/core) and a 3D-torus topology. Kraken is the biggest NSF-sponsored machine in production as of April 2008. The GPU scalability results have been obtained on TeraGrid's *Lincoln* at the National Center for Supercomputing Applications (UIUC/NSF), a Dell cluster with NVIDIA Tesla S1070 accelerators, 1536 cores(Intel Harpertown/2.33 Ghz dual-socket quad-core 2GB/core), 384 GPUs (4GB/GPU), and InfiniBand (SDR) interconnect.

Implementations and libraries. The code is written in C++ and the accelerator modules in CUDA. We use the



(a) uniform (b) ellipse

Fig. 3. Strong scaling on Kraken. *The left figure shows the results for the uniform particle distribution. The right figure shows the results for the nonuniform distribution. In the first case, the problem size is kept fixed to 200M points, and in the second to 100M points. We report average (across processes) wall-clock times for the different FMM setup and evaluation phases using the bars; the black dots denote maximum time across all processes and they correspond to the overall wall-clock time.*



(a) uniform (b) ellipse

Fig. 4. Weak scaling on Kraken. *For the uniform case we use 25K points per process and for the nonuniform case in the right, we use 100k per process. Unlike our SC'03 algorithms and implementation, now the tree construction takes only a small part of the overall computation.*

PETSc [3] for profiling and certain parts of communication, and the DENDRO [13] package for sorting and linear octree construction. The current implementation is based on our past implementation [19]. The new version will be released under GPL in the near future.

Kernels: We used the Stokes single-layer kernel for the examples on Kraken. This is a vector potential, which has three unknowns per grid point. For the GPU results, we used the Laplacian kernel, which is a scalar potential. The former is related to our target applications (fluid mechanics). The latter is less computation intensive and it is a good test for the GPU accelerator. For more details on the kernels, see [20].

MPI strong scalability tests on Kraken. The strong scalability results are reported in Figure 3. The bars indicate the average time spent in the different phases and the black dot indicates the overall wall-clock time (equal to the maximum across processes) for the setup and evaluation phases. The problem size is 200M unknowns for the uniform case and 100M unknowns for the nonuni-

q	30	244	1953
Total evaluation	5.13	1.17	2.15
Upward Pass	0.58	0.13	0.07
U list	0.29	0.45	1.9
V list	3.76	0.44	0.06
Downward Pass	0.35	0.1	0.07

TABLE III
SINGLE GPU. *Here we experimentally study the effect of varying the points-per-box q on the GPU performance. By varying this number, we can study the effects of the relative size of V and U-lists on the overall computation time.*

form case for a range of 512–8K processes. In both cases we observe excellent speed-up with efficiencies between 80%-90% and with very good load balance (note the small difference between the black dot, which is the maximum time across processes and the average timings, which are denoted in the bars).

MPI weak scalability tests on Kraken. The weak scalability results are reported in Figure 4. The problem size (number of input points) per core is kept fixed to 100K points for the nonuniform case and 25K points for the uniform case. The observed increase in timings (1.5x increase as we go from 16 to 64k cores) is due to three reasons: theoretical complexity, load imbalance, and the heterogeneity of processes (on Kraken, half of the cores have 1GB and the other half has 2GB). In figure 5, we report the total flops per core for the 64K-core run for the uniform and nonuniform distributions. Overall, however, the timings are excellent—ranging from 25–45 secs for 512–64K cores for the uniform case, to 60–90 secs for the nonuniform case for the same range of cores. In Table II, we report the exact timings for each phase for a nonuniform distribution problem with 150K particles per core for a total of 30 billion unknowns.

GPU single-core speed-up on Lincoln. We report these results in Table III. We study the effect of the points per box on the GPU performance. We consider 1M points for the uniform case. We can see the optimal value of points per box that can be used in a production run. This tests resembles the tuning phase and can be part of an auto-tuning algorithm.

GPU weak scalability results on Lincoln. We report these results in Figure 6. We only report results for the uniform distribution using one million points per GPU. We use one socket per MPI process and one GPU per

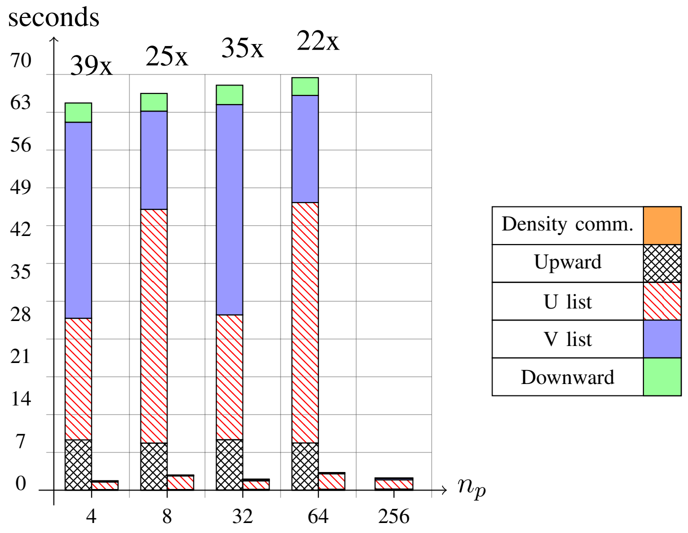


Fig. 6. GPU weak scaling.*Here we compare CPU-only with GPU/CPU configuration on up to 256 processes. For the largest run the total evaluation on 256 million points takes 2.2 secs (we did not run a CPU only example for the largest case). Throughout the computation, we maintain a 25X speed-up over a CPU-only implementation. For the GPU runs, we use a shallower tree by allowing a higher number of points per box. In this way, we favor ULI over VLI computations. The former has a favorable computation/memory communication ratio and performs favorably on a GPU. In this examples, we used roughly 400 points per box for the GPU runs, and 100 points per box for the CPU runs. Both numbers were optimized for their respective architectures. We were able to maintain a 1.8-3 secs / evaluation for the GPU-based implementation.*

socket. Each socket has 4 cores, but at this point we are not using them. The results on GPUs are excellent on up to 256 processes/GPUs. We get over a 25X speedup (compared to CPU-only run) consistently and we were able to evaluate a 256-million particle sum in 2.3 seconds, which corresponds to approximately 8 TFlops/s.

VI. DISCUSSION AND CONCLUSIONS

We have presented several algorithms that taken together expose and exploit concurrency at all stages of the fast multipole algorithms and employ several parallel programming paradigms. We showed that we efficiently scale the tree-setup with the major cost being the parallel sort, which in turn exhibits textbook scalability. We describe a new reduction scheme for the FMM algorithm and we demonstrate overall scalability. We explored per-core concurrency using the streaming paradigm on GPU accelerators with excellent speed-ups. FMM is a highly non-trivial algorithm with several different phases, a combination of multiresolution data structures, fast transforms, and highly irregular data access. Despite these obstacles, we were able to achieve significant speed-ups.

The single core CPU performance for the evaluation part is roughly 500 MFlops/s. On our largest calculation on 64K cores on Kraken we roughly get (Table II) 260 MFlops/s. So overall, we loose 50% of science flops as we scale. Even with this loss, our present code could achieve one PetaFlop/s on a hypothetical 64K-GPU/CPU machine without any further modifications.

Further acceleration is possible, by introducing multicore multithreading for the CPU-to-GPU data transformations, the acceleration of the setup phase using GPU-accelerated sorting and tree construction, and using GPUs/OpenMP for the upward and downward computations.

Acknowledgments.

This work was supported in part by the U.S. National Science Foundation (NSF) grants CNS-0929947, CCF-0833136, OCI-0749285, OCI-0749334, and a grant from the U.S. Defense Advanced Research Projects Agency (DARPA). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or DARPA. Computing resources on the TeraGrid systems were provided under the TeraGrid allocation grants ASC070050N, CCR090024, and MCA04N026. We would like to thank the TeraGrid support staff, and also the staff and consultants at NCSA, PSC, TACC, and NICS from whom we have received significant assistance.

References

[1] *NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.1*, December 2008.

[2] P. Ajmera, R. Goradia, S. Chandran, and S. Aluru, *Fast, parallel, gpu-based construction of space filling curves and octrees*, in I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games, ACM, 2008, pp. 1–1.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc home page*, 2001. http://www.mcs.anl.gov/petsc.

[4] H. Cheng, L. Greengard, and V. Rokhlin, *A fast adaptive multipole algorithm in three dimensions*, Journal of Computational Physics, 155 (1999), pp. 468–498.

[5] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, Addison Wesley, second ed., 2003.

[6] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325–348.

[7] N. A. Gumerov and R. Duraiswami, *Fast multipole methods on graphics processors*, Journal of Computational Physics, 227 (2008), pp. 8290 – 8313.

[8] B. Hariharan and S. Aluru, *Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods*, Parallel Computing, 31 (2005), pp. 311 – 331.

[9] J. JÁJ'a, *An introduction to parallel algorithms*, Addison Wesley, 1992.

[10] J. Kurzak and B. M. Pettitt, *Massively parallel implementation of a fast multipole method for distributed memory machines*, Journal of Parallel and Distributed Computing, 65 (2005), pp. 870 – 881.

[11] S. Ogata, T. J. Campbell, R. K. Kalia, A. Nakano, P. Vashishta, and S. Vemparala, *Scalable and portable implementation of the fast multipole method on parallel computers*, Computer Physics Communications, 153 (2003), pp. 445 – 461.

[12] J. C. Phillips, J. E. Stone, and K. Schulten, *Adapting a message-driven parallel application to GPU-accelerated clusters*, in SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008, pp. 1–9.

[13] R. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, DENDRO *home page*, 2008.

[14] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, *Dendro: parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees*, in SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA, 2008, IEEE Press, pp. 1–12.

[15] F. E. Sevilgen and S. Aluru, *A unifying data structure for hierarchical methods*, in Proceedings of Supercomputing, The SCxy Conference series, Portland, Oregon, November 1999, ACM/IEEE.

[16] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708.

[17] S.-H. Teng, *Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation*, SIAM Journal on Scientific Computing, 19 (1998).

[18] M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree N-body algorithm*, in Proceedings of Supercomputing, The SCxy Conference series, Portland, Oregon, November 1993, ACM/IEEE.

[19] L. Ying, G. Biros, H. Langston, and D. Zorin, KIFMM3D*: The kernel-independent fast multipole (FMM) 3D code*. GPL license.

[20] L. Ying, G. Biros, and D. Zorin, *A kernel-independent adaptive fast multipole method in two and three dimensions*, Journal of Computational Physics, 196 (2004), pp. 591–626.

[21] L. Ying, G. Biros, D. Zorin, and H. Langston, *A new parallel kernel-independent fast multipole algorithm*, in Proceedings of SC03, The SCxy Conference series, Phoenix, Arizona, November 2003, ACM/IEEE.