A CPU–GPU Hybrid Implementation and Model-Driven Scheduling of the Fast Multipole Method

Jee Choi[1], Aparna Chandramowlishwaran[3], Kamesh Madduri[4], Richard Vuduc[2]
[1] *Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA*
[2] *Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA*
[3] *CSAIL, Massachusetts Institute of Technology, Cambridge, MA*
[4] *Computer Science and Engineering, Pennsylvania State University, University Park, PA*

ABSTRACT

This paper presents an optimized CPU–GPU hybrid implementation and a GPU performance model for the kernel-independent fast multipole method (FMM). We implement an optimized kernel-independent FMM for GPUs, and combine it with our previous CPU implementation to create a hybrid CPU+GPU FMM kernel. When compared to another highly optimized GPU implementation, our implementation achieves as much as a 1.9× speedup. We then extend our previous lower bound analyses of FMM for CPUs to include GPUs. This yields a model for predicting the execution times of the different phases of FMM. Using this information, we estimate the execution times of a set of static hybrid schedules on a given system, which allows us to automatically choose the schedule that yields the best performance. In the best case, we achieve a speedup of 1.5× compared to our GPU-only implementation, despite the large difference in computational powers of CPUs and GPUs. We comment on one consequence of having such performance models, which is to enable speculative predictions about FMM scalability on future systems.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems* ; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Parallel and vector implementations*

Keywords

fast multipole method, performance model, GPU, multicore, hybrid, exascale

1. INTRODUCTION

This paper presents a new hybrid implementation and a performance model of the fast multipole method, which is an asymptotically optimal algorithm for approximately computing pairwise interactions with a guaranteed level of accuracy [13]. The FMM is among the most important algorithms for a variety of scientific simulations used to study electromagnetic, fluid, and gravitational phenomena, among others [5]. Furthermore, it has been hypothesized to be of increasing importance on exascale systems [27]. Such systems are likely to include GPU co-processors; therefore, it is natural to ask how one might exploit GPUs for the FMM.

Contributions and findings. The FMM itself has multiple phases, each having different compute and memory characteristics (section 2). We previously analyzed the two most expensive phases of FMM: the near field interaction (U list step), and the far field interaction (V list step) [6]. In this paper, we extend this analysis for GPUs. More specifically, we claim two technical contributions in this paper.

(I) We implement a highly optimized kernel-independent FMM for GPUs. Our implementation improves upon earlier work [19].[1] Specifically, it has GPU implementations of the upward and downward phases; uses shared memory more aggressively; and uses dynamic thread-to-work mappings that amortize the cost of kernel overheads on all phases of the FMM. It performs close to the *realistic* peak performance of the given hardware. It also performs at least 1.9× faster than another highly optimized GPU implementation.

Futhermore, we have combined our GPU implementation with our earlier CPU version [8] into a hybrid implementation, where different phases of the FMM can be assigned to either the CPU or the GPU for asynchronous execution.

(II) We adapt our previous lower bounds on cache complexity to model the performance of the kernel-independent FMM on GPUs (section 4). As with our previous CPU model [6], we can accurately estimate the execution times of various phases of the FMM. The combined CPU+GPU model allows us to estimate the execution times of various static schedules, where we map different phases to the CPU and GPU for simultaneous execution, and therefore predict which schedule will perform the best (section 5). The study in this paper is admittedly limited to a small number of hand-picked schedules. However, since our model accurately predicts the time it takes to do various phases of the FMM on both CPUs and GPUs, it still allows a comprehensive exploration of the scheduling space.

One consequence of such a performance model is that it becomes possible to estimate performance of FMM on hypothetical future systems. Though we do not fully explore this idea, we offer a suggestive illustration in section 6.

GPGPU-7, March 01 2014, Salt Lake City, UT, USA
http://dx.doi.org/10.1145/2576779.2576787
Copyright 2014 ACM 978-1-4503-2766-4/14/03 ...$15.00.

[1]Our source code is publicly available at: `https://github.com/jeewhanchoi/kifmm--hybrid--double-only`

2. THE FAST MULTIPOLE METHOD

Given a system of N *source* particles, with positions given by $\{y_1, \ldots, y_N\}$, and N *targets* with positions $\{x_1, \ldots, x_N\}$, we wish to compute the N sums,

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_i) \cdot s(y_j), \quad i = 1, \ldots, N \qquad (1)$$

where $f(x)$ is the desired *potential* at target point x; $s(y)$ is the *density* at source point y; and $K(x, y)$ is an *interaction kernel* that specifies "the physics" of the problem. For instance, the single-layer Laplace kernel, $K(x, y) = \frac{1}{4\pi}\frac{1}{||x-y||}$, might model electrostatic or gravitational interactions.

Evaluating these sums appears to require $O(N^2)$ operations. The FMM instead computes *approximations* of all of these sums in optimal $O(N)$ time with a guaranteed user-specified accuracy ϵ, where the desired accuracy changes the complexity constant [13]. The FMM is based on two key ideas:

- organizing the points in a *spatial tree*; and
- using *fast approximate evaluations*, in which we compute summaries at each node using a constant number of tree traversals with constant work per node.

We model and implement the *kernel-independent* variant of the FMM, or KIFMM [26]. KIFMM has the same structure as the classical FMM [13]. Its main advantage is that it avoids the mathematically challenging analytic expansion of the kernel, instead requiring only the ability to evaluate the kernel. This feature of the KIFMM allows one to leverage our optimizations and techniques and apply them to new kernels and problems.

Tree construction: Given the input points and a user-defined parameter q, we construct an octree T (or quad-tree in 2D) by starting with a single box representing all the points and recursively subdividing each box if it contains more than q points. Each box (octant in 3D or quadrant in 2D) becomes a tree node whose children are its immediate sub-boxes. During construction, we associate with each node one or more neighbor *lists*. Each list has bounded constant length and contains (logical) pointers to a subset of other tree nodes. These are canonically known as the U, V, W, and X lists. For example, every leaf box B has a U *list*, $U(B)$, which is the list of all leaves adjacent to B. Figure 1 shows a quad-tree example, where neighborhood list nodes for B are labeled accordingly.

Tree construction has $O(N \log N)$ complexity, and the $O(N)$ optimality of FMM refers to the evaluation phase (below). However, tree construction is typically a small fraction of the total time; moreover, many applications build the tree periodically, thereby enabling amortization of this cost over several evaluations.

Evaluation: Given the tree T, evaluating the sums consists of six distinct computational phases: there is one phase for each of the U, V, W, and X lists, as well as *upward* (up) and *downward* (down) phases. These phases involve traversals of T or subsets of T. Rather than describe each phase in detail, we refer the readers to the following publications [13, 25, 26].

There are multiple levels of concurrency during evaluation: across phases (e.g., the upward and U-list phases can be executed independently), within a phase (e.g., each leaf box can be evaluated independently during the U-list phase), and within the per-octant computation (e.g., vectorizing each direct evaluation).

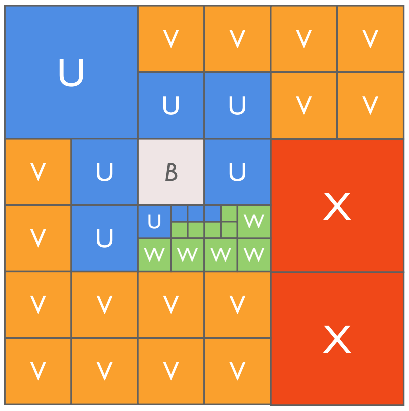


Figure 1: U, V, W, and X lists of a tree node B for an adaptive quadtree.

3. PRIOR STUDIES AND RELATED WORK

There is extensive literature on the FMM and its parallelization, from fast shared memory implementations to highly scalable distributed memory codes, to GPU-based accelerated implementations [2, 4, 10, 12, 14–21, 24, 25]. Within-node, the current state-of-the-art FMM implementation is our own prior multicore code [7, 8, 19, 22]. Therefore, we take this implementation as the baseline CPU code for our study.

For hybrid-FMM on CPU-GPU systems, Hu et al. [17] discuss a work partitioning scheme where all computation on the leaf nodes is done on the GPU and everything else is handled by the CPU. Moreover, this partitioning scheme is fixed for all input sizes and desired accuracy. To our knowledge, our paper is the first to discuss a systematic model-based work division, which may result in optimal scheduling decisions that run *counter* to the Hu et al. scheme.

One promising alternative to our scheduling approach is to use generic task runtimes, such as StarPU [1, 3]. In particular, adding an accurate performance model for each architecture on a heterogeneous system greatly improves load balancing on StarPU [3]. However, the model we present allows the spatial tree data structure itself to be tuned via q, which may be outside the scope of a generic task scheduler. Nevertheless, combining these approaches is a promising avenue for future work.

4. GPU MODELING

This section presents a performance model of the FMM when running on a GPU, building on the analysis method described in our previous work for CPUs [6].

4.1 Adapting for GPUs

Unfortunately, our earlier model does not directly translate to the GPU case without modification, due to significant architectural differences. The most important is that the effective value of the fast memory size parameter, Z, will be smaller on GPUs due to their relatively smaller caches and local stores. As a result, data blocking is inherently more difficult on GPUs.

To adapt the earlier model, we need to consider the U list, V list, and Up and $Down$ phases in turn.

First, consider the U list phase. In our previous work, we modeled the time it takes to do computation ($T_{comp,u}$) and the times it takes to move data ($T_{mem,u}$) for the U list computation on CPUs [6]. For our GPU implementation, optimistically assuming asynchronous execution of computation and data movement, we use the equation for $T_{comp,u}$ to model the total execution time, since the computation is compute-bound and both the CPU and the GPU do the same number of floating point operations. That is,

$$T_{u,gpu} = \frac{C_{u,gpu}(3b^{1/3}-2)^3 q^2}{C_{peak,gpu}}, \qquad (2)$$

where b is the number of leaf boxes, q is the number of points in each box, $C_{u,gpu}$ is a kernel- and implementation-dependent constant, and $C_{peak,gpu}$ is the effective computational peak of the GPU (section 4.2.2). Basically, we are taking the total number of floating point operations, as defined by b and q, and dividing it by the system's expected performance to get the theoretical execution time, and then multiplying it by $C_{u,gpu}$ to take into account any kernel- and implementation-dependent factors.

Next, consider the V list phase. In contrast to the U list, the V list computation is memory-bound and therefore must be modeled using memory costs. However, we cannot directly apply the equation used for CPUs to predict time for GPUs because the CPU model assumes that all of the translation operators are re-used in the fast memory. GPUs, unfortunately, do not have enough fast memory to do the same. Thus, we instead use a *data streaming* model for the GPU implementation of V list where *all* data is assumed to be coming from the main memory, and that there is little or no caching. That is,

$$T_{v,gpu} = \frac{C_{v,gpu}(3bp^{3/2})189}{\beta_{mem,gpu}}, \qquad (3)$$

where p is a parameter related to the user's desired level of accuracy, 189 is the maximum number of neighbors in the V list, $C_{v,gpu}$ is a kernel- and implementation-dependent constant, and $\beta_{mem,gpu}$ is the achievable memory bandwidth of the GPU (section 4.2.1).

Lastly, consider the Up and $Down$ phases. Like the V list, these phases are memory-bound and we may therefore use a data streaming model. Since we did not have a CPU model to base the GPU model upon, we simply calculate an upper-bound on data movement based on the data structures used in our implementation.

These bounds are, in particular,

$$T_{up,gpu} = \frac{C_{up,gpu}\,(4N + 2bf_1(p)\,(f_2(p)+1))}{\beta_{mem}}, \qquad (4)$$

$$T_{down,gpu} = \frac{C_{down,gpu}\,\left(N + 2bf_1^2(p) + 2bf_1(p)\right)}{\beta_{mem}}, \qquad (5)$$

where N is the total number of points and f_1 and f_2 are some functions of the accuracy, p. $C_{up,gpu}$ and $C_{down,gpu}$ are again the kernel- and implementation-dependent factors.

4.2 Estimating effective peak for GPUs

Our goal is to accurately estimate time, and our model needs a value for peak throughput. Rather than relying on a vendor's theoretical peak, we use computation-specific microbenchmarks to measure a more realistic *effective* peak throughput, separately for compute and memory.

4.2.1 Effective peak memory throughput

Our microbenchmark for estimating effective β_{mem} is similar to the `bandwidthTest` included in NVIDIA's software development kit. The main difference is that we tune it more carefully than the SDK version, using standard techniques and some autotuning [9].

4.2.2 Effective peak compute throughput

Coming up with a realistic peak computation throughput requires specializing the benchmark to the target computation of interest.

Consider that for GPUs, theoretical peak assumes that the maximum number of fused multiply-add (FMA) is being issued every cycle, which accounts for two FLOPs for every instruction issued. For the U list, the inner-most loop executes 3 subtracts, 4 multiply-adds, and a reciprocal square root. Since only half of our instructions are FMAs, the performance that our FMM implementation can achieve will be lower than the theoretical peak. Moreover, although GPUs provide hardware support for computing single-precision reciprocal square root, there is no indication of such hardware for double-precision, which we need in our GPU implementation.

Using another highly-tuned microbenchmark, we were able to deduce that a reciprocal square root in double-precision has a latency of approximately 14 cycles of non-pipelined execution, or equivalently, 14 pipelined instructions. Therefore, we may estimate that it would take approximately $7 + 14 = 21$ instructions to execute one iteration of the inner loop in the U list. Thus, an estimate of effective peak compute throughput is,

$$C_{peak,gpu} = \frac{11\text{ FLOPs}}{21\text{ instrs}} \times \text{freq} \times \text{proc}, \qquad (6)$$

where $freq$ and $proc$ are processor frequency and count, respectively. Note that equation (6) has units of cycles per second; and that it also optimistically assumes that there is enough thread-level parallelism to hide all instruction latencies and ignores loop overhead.

4.3 Deriving the constants

Now that we have our model and a way to calculate the peak throughputs for the GPU, we can derive the constants to predict the execution time and to see how close to the peak our implementation gets.

For a given phase, we measure the execution times for varying values for N, q, and p, and then use linear regression to derive the constant terms. Table 1 shows our peak throughput estimates and constants for the different phases of FMM on our two systems. We discuss the implications of the constant values in Section 5.

5. PERFORMANCE ANALYSIS

In this section, we first analyze the performance of the different phases of the FMM based on the performance model and the constants we derived in the previous section. We then discuss how closely our model predicts the execution times of the various phases as well as those of the different

	Tesla M2090	GTX Titan
P_{comp} (GFLOP/s)	174.3	392.9
P_{mem} (GB/s)	129.4	237.2
$C_{up,gpu}$	2.99	4.16
$C_{u,gpu}$	1.56	2.09
$C_{v,gpu}$	0.95	1.40
$C_{down,gpu}$	7.61	6.83

Table 1: Peak throughputs and constants for different FMM phases on our GPU test platforms.

hand-picked hybrid schedules. We also compare the performance of our FMM implementation across different hardware architectures shown in Table 2. We briefly compare the performance of our implementation against that of another highly optimized GPU implementation [17]. Finally, we project our analysis to predict the performance of FMM on future systems.

Name	Type
CPU–1	Intel Xeon X5650 “Westmere-EP”
CPU–2	Intel Xeon E5-2603 “Sandy Bridge-EP”
GPU–1	NVIDIA Tesla M2090 “Fermi”
GPU–2	NVIDIA GTX Titan “Kepler”
Hybrid–1	CPU–1 + GPU–1
Hybrid–2	CPU–2 + GPU–2

Table 2: Experimental testbeds.

We consider two different particle distributions, namely a uniform random distribution and an elliptical (non-uniform) distribution. The uniform case distributes particles uniformly within a unit cube (see Figure 2). The elliptical case distributes particles on the surface on an ellipsoid with an aspect ratio of 1:1:4. The former leads to a relatively uniform (regular) tree, whereas the latter leads to a highly adaptive (and therefore more irregular) tree.

We use the analytical performance model to estimate the optimal scheduling of the different phases of FMM for *only* the uniform case, where an analytic model is known. An analytical model for general non-uniform distributions is more complex and is part of our future work. As such, results for the elliptical case serve to evaluate our *implementation*, rather than the *model*.

5.1 GPU Constants

The GPU constant values measured and reported in Table 1 warrant some comment.

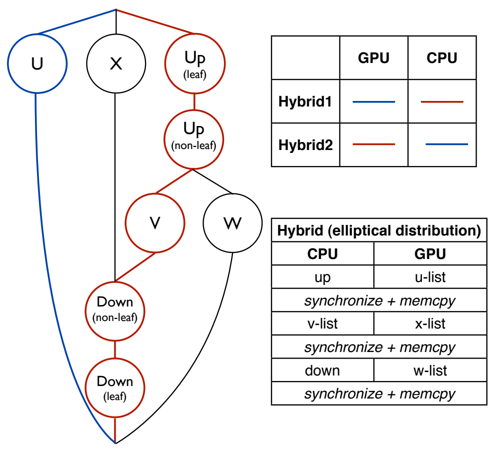
The values of $C_{u,gpu}$ and $C_{v,gpu}$ are relatively close to the desired value of 1 on both platforms, indicating that our implementation is efficient. The values of $C_{u,gpu}$ are slightly larger than $C_{v,gpu}$, likely due to significant loop overheads which cannot be hidden for compute-bound kernels. The value of $C_{v,gpu}$ on the Tesla M2090 is slightly below 1. This observation may reflect a data caching effect in L1 that is present in Fermi GPUs and not in Kepler GPUs.

The constants for both the U list and V list are smaller on the Tesla M2090 than on the other platform. This is most likely due to the fact that *Kepler* GPUs require a higher degree of thread-, instruction-, and memory-level parallelism to fully utilize the system. In both cases, however, variations are still small enough that predicted times are not severely affected.

The *Up* and *Down* phases have higher constant values and show a strong dependence on precision on both platforms. Indeed, our GPU implementation for *Up* and *Down* is suboptimal in the following sense. It traverses and processes the tree synchronously, level-by-level; going from one level to the other changes the amount of work logarithmically, meaning some levels may have less available work than needed to saturate the GPU. Moreover, at lower precision, some of the translation matrices involved can be re-used via the L2 cache. As we will see in the next section, our model predictions for *Up* and *Down* phases are less accurate than for the U and V lists. However, this effect does not significantly impact the accuracy of the overall model because *Up* and *Down* phases only take up a small percentage of the total execution time.

5.2 Predicting performance and scheduling

The directed acyclic graph (DAG) of Figure 3 reveals the dependencies between the various phases of FMM.



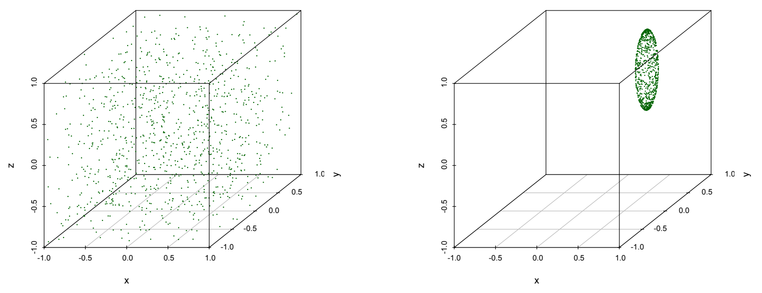
	GPU	CPU
Hybrid1	🟦	🟥
Hybrid2	🟥	🟦

Hybrid (elliptical distribution)	
CPU	GPU
up	u-list
synchronize + memcpy	
v-list	x-list
synchronize + memcpy	
down	w-list
synchronize + memcpy	

Figure 3: Directed Acyclic Graph for FMM.

The DAG for the uniform distribution is identical, with the X and W phases removed. Figure 3 also shows the distribution of work between CPU and GPU for various hybrid scheduling strategies, for both uniform and non-uniform distributions.

For the uniform distribution there are, heuristically, two scheduling strategies, which we label *hybrid1* and *hybrid2*. The DAG in the uniform case has two independent paths: the U list path and the $Up \rightarrow V$ list $\rightarrow Down$ path, with no communication between the two paths until the very end. The *hybrid1* scheme runs the compute-intensive U list step on the GPU and the other data-intensive path on the CPU. The *hybrid2* scheme reverses these assignments. There is also an extra cost at the end when the result from the GPU is sent back to the host over PCIe, which we include in our overall execution time. Although these are not the only possible schedules for the given DAG, they are reasonable heuristics in that U list and V list account for approximately



(a) (b)

Figure 2: Distribution of particles inside a unit cube. Left: uniform random distribution. Right: ellipsoidal distribution with an aspect ratio of 1:1:4.

90% of the overall execution time. Using our CPU and GPU performance models, we can analytically predict which of the two schedules will yield the best performance.

Our model accurately predicts execution time, as Figure 4(a) shows in the uniform case. All errors listed below were taken over a range of values for the total number of points N, the number of points per leaf box q, and the accuracy, p. For the GPU-only cases, the model predicts time with median errors of 7.5% and 6.9% for Tesla M2090 and GTX Titan respectively. For the CPU-only cases, the median errors are 2.2% and 2.0% for X5650 and E5-2603 respectively. For Hybrid–1 and Hybrid–2 systems, the median errors were 8.6% and 7.1% respectively for *hybrid1* scheduling, and 2.8% and 2.0% respectively for *hybrid2* scheduling. In terms of choosing the best schedule amongst *hybrid1*, *hybrid2*, CPU–only, and GPU–only for the two hybrid systems, the model *always* chose the best schedule on both systems.

We omit performance results for Hybrid–2. Our model can predict this case, but the results are not interesting due to a large difference in performance of the two processors: we use a low-end CPU with a high-end GPU for Hybrid–2. In particular, the GPU outperforms the CPU by a larger factor and hybrid implementation is barely, if at all, better than the GPU-only implementation.

For a non-uniform distribution, there are a larger number of possible paths from which to choose. First, there are more ways to schedule the DAG itself, which makes the search space for the *best* schedule larger. Secondly, we can tune the number of points per box, q, which allows us to vary the execution time for the different phases of FMM to achieve maximum overlap. Doing so might improve resource utilization. However, we would no longer solely tune for U list and V list phases, since they may no longer be the dominant part of the computation. Furthermore, modeling non-uniform distributions in general is a hard problem. We present one chosen scheduling strategy which works relatively well for the given architecture and implementation. There is definitely a strong need for a model-driven hybrid scheduling framework for non-uniform distributions, which will be part of our future work.

5.3 Performance of CPU vs GPU vs Hybrid

To compare the performance of the FMM on CPU–1 and GPU–1 systems, for both uniform and elliptical distributions, see Figure 5. Time is broken down into different phases for $N = 4$ million particles with γ set to yield 6 digits of accuracy. For the uniform case, the majority of the time is spent in the U list and V list steps. For the elliptical case, time is more spread out over the different phases. Nevertheless, in both cases the GPU achieves an overall performance improvement of 1.7× over the CPU.

To see the impact of hybrid scheduling, we vary γ and Figure 4 shows the performance of the three variants. Hybrid scheduling performs the best and the improvement over GPU increases as we move to larger accuracy requirements.

5.4 Comparing to other GPU implementations

The closest "alternative" implementation is Hu et al. [17]. Unfortunately, it is not possible to carry out a fair comparison due to several critical differences.

For instance, we implement the kernel-independent method, whereas Hu et al. uses the classical one; our implementation is only in double-precision and for 3, 4, and 6 digits of accuracy, whereas their implementation uses both single- and double-precision for 4, 8, 12 and 16 digits of accuracy; and the evaluation platforms differ: both efforts use Fermi-based GPUs, but Hu et al. uses a Tesla C2050 with a peak of 515.2 GFLOP/s whereas we use the Tesla M2090 having a peak of 666.1 GFLOP/s, both of which are "Fermi" GPUs.

Nevertheless, to get a sense of the relative performance, consider the following comparison. The Hu et al. GPU implementation takes 1.36 seconds for $N = 1048576$ and $p = 4$, whereas our GPU implementation only takes 0.71 seconds for the same value of N and $\gamma = 4$, achieving a near 2× speedup. Although the platforms used differ, even accounting for the 30% differences in hardware computing power, our implementation is measurably faster.

The Hu et al. study does parallelize tree construction for the GPU, whereas we perform tree construction on the CPU only and have excluded it from the results. However, as Hu et al. note, this cost is very small and can be amortized over multiple kernel executions [17]. Their approach could be simply "dropped in" to our implementation since tree construction is a distinct preprocessing step.

6. CONCLUSIONS AND DISCUSSION

The main aims of this paper are (i) to develop a well-tuned CPU+GPU implementation of the KIFMM algorithm, and

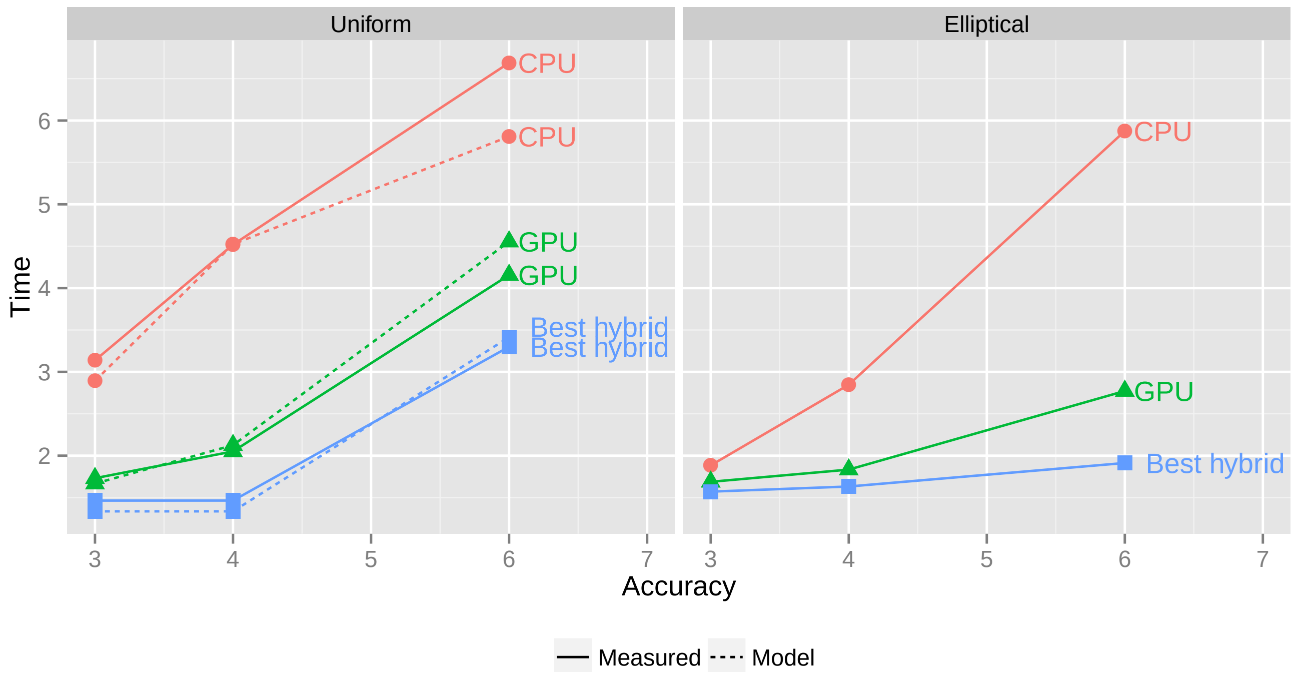


Figure 4: Comparison of run time and modeled time on Hybrid–1 for $N = 4M$ for uniform and elliptical distributions for varying γ.

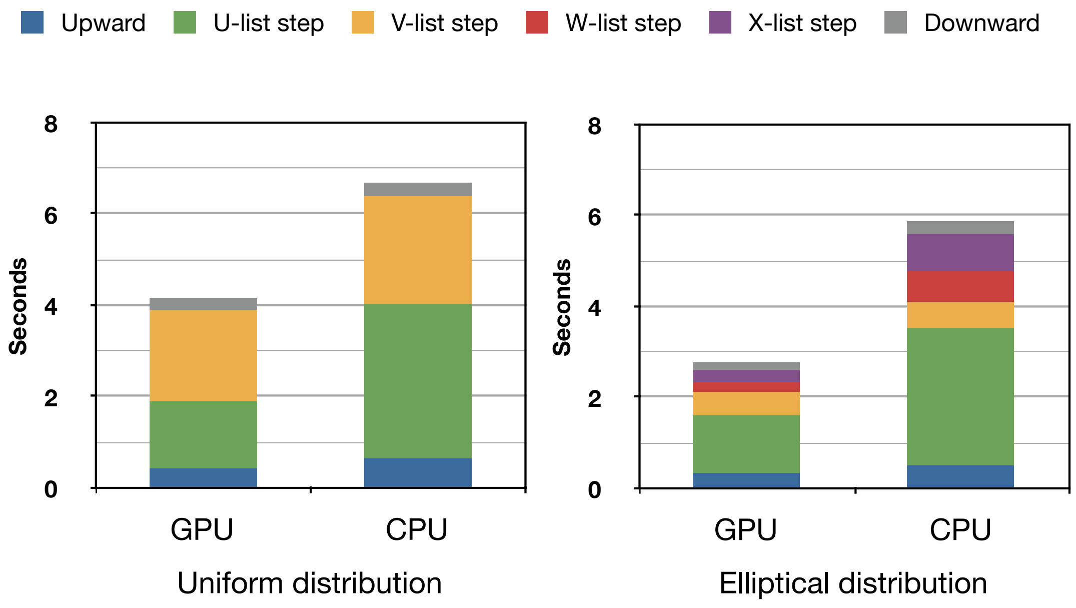
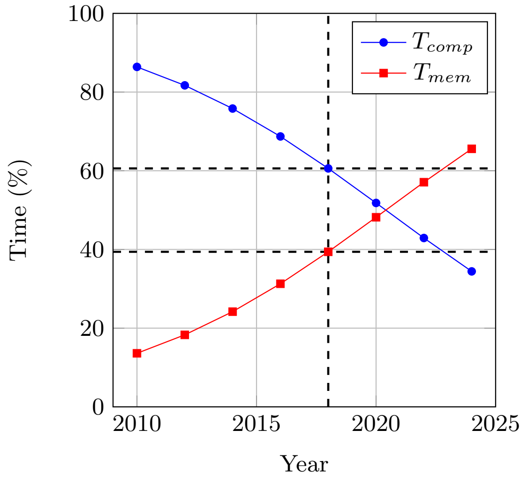


Figure 5: Breakdown of running time for a Laplace kernel potential calculation and uniform and elliptical particle distributions ($N = 4M$, $\gamma = 6$, double precision) on CPU–1 and GPU–1.

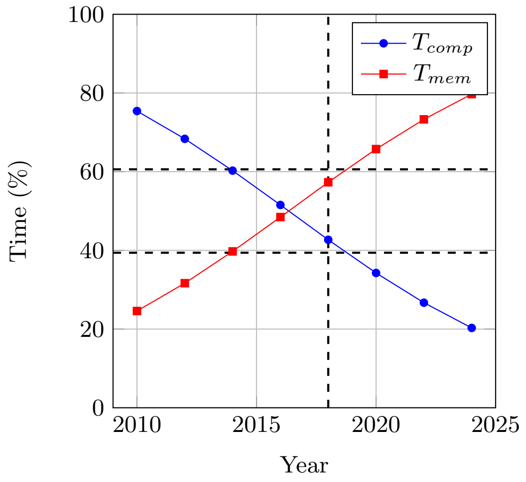
then (ii) solve the CPU/GPU work-division problem for this implementation using a model-driven approach. The model is primarily analytical, with machine parameters exposed. However, such a model also has utility beyond scheduling.

In particular, we can consider how future architectures with different machine parameters might perform and scale. As a suggestive example, consider the following back-of-the-envelope projection that our model makes possible.

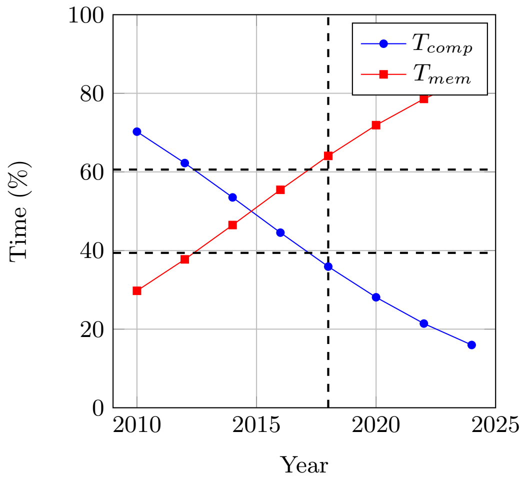
Using the analytic expression for execution time and the optimal choice of q [6], we can estimate the execution time on possible future CPU–, GPU–, and hybrid–based exascale systems, for a problem instance that is large relative to the last-level cache. For the values of the machine parameters of these hypothetical systems, suppose we use previously



(a) CPU



(b) GPU



(c) Hybrid

Figure 6: Projected optimistic run time on extrapolated architectures. The problem size N starts at 4 million points in 2010, and is scaled at the same rate as the cache size Z.

published values that were based on extrapolating historical technology trends [23]. For the purpose of the projections, we use the theoretical throughputs for both compute and memory, as well as for the fast memory size Z. We also optimistically assume implementation-dependent constant values of 1, hoping that future implementations will be free of inefficiencies or overheads. For each technology scaling step, we compute the expected execution times for U and V on the CPU, GPU and hybrid systems for a problem size that also scales with the size of the fast memory Z.

Figure 6 shows the execution time split into computational (blue) and memory access (red) time for three different systems namely, (a) CPU-based system, (b) GPU-based system, and (c) a hybrid CPU-GPU system. We observe that the crossover point when the memory access time T_{mem} matches the compute time T_{comp} occurs at different time frames for each of these system configurations. This implies that the more *imbalanced* the system, the sooner we will observe the crossover. That is, in a power-constrained future [11] where we will only have a limited amount of power (and therefore performance) to allocate to both compute and memory, we will need to carefully balance these two parameters in order to allow FMM to scale further into the future.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under CAREER award number 0953100 and ACI award number 1253881. Additional support provided by the U.S. Dept. of Energy (DOE), Office of Science, Advanced Scientific Computing Research through the X-Stack 1.0 program under award DE-FC02-10ER26006/DE-SC0004915. 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 DOE.

7. REFERENCES

[1] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-Based FMM for Multicore Architectures. Technical Report hal-00911856, Inria, 2013.

[2] P. Ajmera, R. Goradia, S. Chandran, and S. Aluru. Fast, parallel, GPU-based construction of space filling curves and octrees. In *Proc. Symp. Interactive 3D Graphics (I3D)*, Redwood City, CA, USA, 2008. (*poster*).

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] L. A. Barba and R. Yokota. How will the fast multipole method fare in the exascale era? *SIAM News*, 46(6), 2013.

[5] J. Board and K. Schulten. The fast multipole algorithm. *Computing in Science and Engineering*, 2(1):76–79, January/February 2000.

[6] A. Chandramowlishwaran, J. W. Choi, K. Madduri, and R. Vuduc. Towards a communication optimal fast multipole method and its implications for exascale. In *Proc. ACM Symp. Parallel Algorithms and Architectures (SPAA)*, Pittsburgh, PA, USA, June 2012. Brief announcement.

[7] A. Chandramowlishwaran, K. Madduri, and R. Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, November 2010.

[8] 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. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Atlanta, GA, USA, April 2010.

[9] J. Choi, R. Vuduc, R. Fowler, and D. Bendard. A roofline model of energy. In *In Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 13)*, 2013.

[10] F. A. Cruz, M. G. Knepley, and L. A. Barba. PetFMM-A dynamically load-balancing parallel fast multipole library. *International Journal for Numerical Methods in Engineering*, 85(4):403–428, Jan. 2011.

[11] K. Czechowski and R. Vuduc. A theoretical framework for algorithm-architecture co-design. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 791–802, 2013.

[12] W. Fong and E. Darve. The black-box fast multipole method. *J. Comp. Phys.*, 228(23):8712–8725, December 2009.

[13] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, 73:325–348, 1987.

[14] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *J. Comp. Phys.*, 227:8290–8313, 2008.

[15] T. Hamada, T. Narumi, R. Yokota, K. Y. K. Nitadori, and M. Taiji. 42 TFlops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Portland, OR, USA, November 2009.

[16] B. Hariharan and S. Aluru. Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods. *Parallel Computing (ParCo)*, 31(3–4):311–331, March–April 2005.

[17] Q. Hu, N. A. Gumerov, and R. Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Seattle, WA, USA, November 2011.

[18] J. Kurzak and B. M. Pettitt. Massively parallel implementation of a fast multipole method for distributed memory machines. *J. Parallel Distrib. Comput.*, 65:870–881, July 2005.

[19] 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 *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Portland, OR, USA, November 2009. Finalist, Best Paper.

[20] 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 Phys. Comm.*, 153(3):445–461, July 2003.

[21] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Austin, TX, USA, November 2008.

[22] A. Rahimian, I. Lashuk, D. Malhotra, A. Chandramowlishwaran, L. Moon, R. Sampath, A. Shringarpure, S. Veerapaneni, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, New Orleans, LA, USA, November 2010.

[23] R. Vuduc and K. Czechowski. What GPU computing means for high-end systems. *IEEE Micro*, July/August 2011.

[24] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, pages 12–21, Portland, OR, USA, November 1993.

[25] L. Ying, G. Biros, D. Zorin, and H. Langston. A new parallel kernel-independent fast multipole method. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Phoenix, AZ, USA, November 2003.

[26] L. Ying, D. Zorin, and G. Biros. A kernel-independent adaptive fast multipole method in two and three dimensions. *J. Comp. Phys.*, 196:591–626, May 2004.

[27] R. Yokota and L. A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *Int. J. High-perf. Comput.*, 2011.