Diagnosis, Tuning, and Redesign for Multicore Performance: A Case Study of the Fast Multipole Method

Aparna Chandramowlishwaran†, Kamesh Madduri⋆, Richard Vuduc†

†*School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA*
⋆*Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA*

***Abstract*—Given a program and a multisocket, multicore system, what is the process by which one understands and improves its performance and scalability? We describe an approach in the context of improving within-node scalability of the fast multipole method (FMM). Our process consists of a systematic sequence of modeling, analysis, and tuning steps, beginning with simple models, and gradually increasing their complexity in the quest for deeper performance understanding and better scalability. For the FMM, we significantly improve within-node scalability; for example, on a quad-socket Intel Nehalem-EX system, we show speedups of 1.7× over the previous best multithreaded implementation, 19.3× over a sequential but highly tuned (e.g., SIMD-vectorized) code, and match or outperform a state-of-the-art GPGPU implementation. Our study sheds new light on the form of a more general performance analysis and tuning process that other multicore/manycore tuning practitioners (end-user programmers) and automated performance analysis and tuning tools could themselves apply.**

I. Introduction

This study was prompted by the following question: What is the *process* by which one might start with a program and then *systematically* analyze and tune its single-node multisocket multicore performance?[1] Though there are many principles, techniques, and promising tools [2]–[10], [10]–[16], defining a general and practical process is exceedingly difficult, owing both to the diversity of programs and to the complexity of modern multicore systems. The practitioner (end-user programmer), whose job it is to identify, to understand, and to fix within-node performance bottlenecks, often has little or no guidance on how to proceed.

Summary and contributions. This paper attempts to document just such a process, within the specific context of improving the within-node scalability of a distributed memory implementation of the *fast multiple method (FMM)* [1], [17]–[19]. The FMM is regarded as one of the most important algorithms in scientific and engineering computing [20], [21]. A deep understanding of how to improve its scalability has wide-ranging implications not just for physical science and engineering applications, but also for those in the emerging area of massive-scale statistical data analysis [22], [23].

[1]John Mellor-Crummey posed this question during a discussion of our earlier work on tuning the fast multipole method (FMM) [1].

For our study, we claim three contributions.

(I) We document a systematic process for transforming a conventionally parallelized FMM into a highly-tuned one. We explain this process in a way that will be useful to other tuning practitioners and performance analysis tool builders.

(II) We decompose this process into three stages. In the first stage, we treat the program and hardware as a black box, limiting analysis and tuning to simple modeling and measurement techniques. In the second stage, we assume more knowledge of the computation and machine, enabling deeper inferences and at least a limited set of code transformation techniques. In the final stage, we assume deep knowledge of the code and machine, and therefore not only arrive at the deepest insights, but also apply the most aggressive transformations. At each stage, we show by example what models, insights, hypotheses, and performance-enhancing optimizations—including aggressive asynchronous scheduling and reordering optimizations for explicit locality and bandwidth management—might be discovered and applied.

(III) As a by-product, we significantly improve the within-node scalability of the FMM, enabling it to scale in principle to future designs based on manycore processors. Our evaluations include a state-of-the-art 1-node x 4-socket x 8-core (32 cores total) Intel Nehalem-EX system, for which we achieve speedups of 1.7× over the previous best double precision implementation [1]. That earlier implementation (a) was itself highly tuned, incorporating SIMD vectorization, numerical optimizations, and memory hierarchy transformations; and (b) matched or outperformed a state-of-the-art GPGPU implementation [19].

Implications and limitations. At first glance, the main limitation of this study may be its seemingly narrow focus on the FMM, which raises the natural question of how well this process could generalize. We mitigate this issue in two ways. First, we use the FMM because of its inherent implementation complexity. The FMM consists of many phases of varying computational intensity and memory behavior, thereby making it exhibit many of the general computational characteristics and features of full-scale applications while remaining compact enough to study in detail. Entire program tuning will be

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

more complicated, but we believe algorithm or component-level tuning in the style we describe will be a useful starting point. Secondly, we choose to characterize the overall process by "level of practitioner," where the analysis and optimization techniques that require the least expertise are likely to be the simplest to generalize and to apply to other programs; and, more importantly, the easiest to automate and to incorporate into existing performance analysis tools [13], [15], [24]–[31].

II. Architectural Summary

Table I provides a summary of our evaluation architectures. These systems have a range of characteristics of particular interest to multicore performance analysis and optimization.

First, we consider both dual-socket quad-core (8 cores total) and quad-socket oct-core (32 cores) systems. Relative to prior FMM multicore studies, our use of a 32-core multisocket multicore configuration is among (if not the) largest single-node configuration tested to date in terms of core counts [1], [32]. Our study sheds light on what algorithmic or architectural features may be necessary to continue scaling the FMM on future systems.

Secondly, the systems span a range of cache capacity and cache sharing configurations. The last-level cache capacities range from as little as 512 KB per core (Barcelona) to as high as 3 MB per core (Harpertown and Nehalem-EX). Thus, we expect to be able to explore the effect of cache capacity. Moreover, the Harpertown system has on each socket two L2 caches, each of which is shared by a pair of cores. We will see how this configuration permits insight into the code.

Thirdly, we consider platforms both with and without non-uniform memory architectures (NUMA). Thus, we can evaluate the effects of data placement for the FMM, as well as strategies for diagnosing and improving NUMA-related performance issues.

Finally, we consider both simultaneous multithreading (SMT) and non-SMT processors. Thus, we will be able to see the extent to which existing x86 SMT designs provide the right kind of support for an FMM style computation.

III. The Fast Multipole Method

This section gives a cursory overview of the FMM. The algorithm is complex and a detailed discussion is beyond the scope of this paper; we instead refer the interested reader elsewhere for details [17], [18].

Overview. 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.

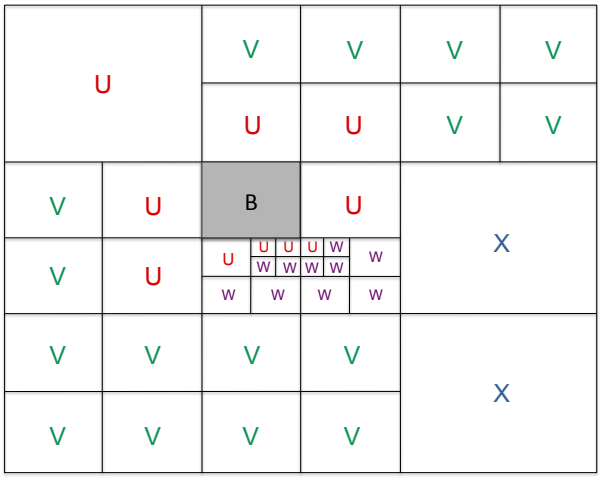

Fig. 1: U, V, W, and X lists of a tree node B for an adaptive quadtree in 2-D.

A straightforward evaluation of these sums requires $\mathcal{O}(N^2)$ operations. The FMM instead computes *approximations* of all of these sums in $\mathcal{O}(N)$ time with a guaranteed user-specified accuracy, where the desired accuracy changes the complexity constant [17].

The FMM is based on two key ideas: (i) a *tree representation* for organizing the points spatially; and (ii) *fast approximate evaluation*, in which we compute summaries at each node using a constant number of tree traversals with constant work per node.

There are several variations of the FMM, and we implement the *kernel-independent* (KIFMM) variant [18]. KIFMM has the same structure as the classical FMM [17]. Its main advantage is that it avoids the mathematically challenging analytic expansion of the kernel, instead requiring only the ability to evaluate the kernel.

Tree construction. Given the input points and a user-defined parameter q, we construct an oct-tree T (or quad-tree in 2-D) 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 3-D or quadrant in 2-D) 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 $B \in \text{leaves}(T)$ 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 so the $O(N)$ optimality refers to the evaluation phase (below).

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. The execution of these phases is the dominant cost of the FMM. These phases involve traversals

TABLE I: Architectural details of parallel systems used in our study. †shared among cores on a socket. ‡shared among 2 cores on a socket.

System **Core Architecture**	Intel E5405 **Harpertown**	AMD Opteron 2356 **Barcelona**	Intel X5550 **Nehalem-EP**	Intel X7560 **Nehalem-EX**
Sockets×cores×threads	$2 \times 4 \times 1$	$2 \times 4 \times 1$	$2 \times 4 \times 2$	$4 \times 8 \times 2$
# threads	8	8	16	64
Clock (GHz)	2.00	2.30	2.66	2.27
DP (SP) GFlop/s	64 (128)	73.6 (146.2)	85.33 (170.6)	290 (58)
L1/L2/L3 cache (KB)	32/6144‡/-	64/512/2048†	32/256/8192†	32/256/24576†
DRAM Capacity (GB)	4	16	12	64
Bandwidth (GB/s)	21.33	21.33	51.2	170.6
Compiler	Intel C v11.0	GNU C v4.4.1	Intel C v11.0	Intel C v11.0

TABLE II: Characteristics of the computational phases in KIFMM. N is the number of source particles, the number of boxes M is roughly N/q, and p denotes the number of expansion coefficients. The user chooses p to trade-off time and accuracy, and may tune q to minimize time. †Size is determined by the chosen accuracy, generally smaller than q.

Phase	Computational Complexity	Algorithmic Characteristics
Upward	$O(Np + Mp^2)$	postorder tree traversal, small† dense matrix-vector computations (matvecs)
U-list	$O(27Nq)$	direct computation as in Equation 1 (matvecs on the order of q)
V-list	$O(Mp^{3/2}logp + 189Mp^{3/2})$	consists of small FFTs, pointwise vector multiplication (convolution)
X-list	0 uniform distribution $O(Nq)$ non-uniform distribution	matvecs
W-list	0 uniform distribution $O(Nq)$ non-uniform distribution	matvecs
Downward	$O(Np + Mp^2)$	preorder tree traversal, small† matvecs

of T or subsets of T. Rather than describe each phase in detail, which are well-described elsewhere [17], [18], [33], we summarize the main algorithmic characteristics of each phase in Table II. For readers familiar with traditional descriptions of the FMM, in this paper we henceforth refer to the so-called "M2L" and "near interaction" computational phases by their associated list names, V list and U list.

IV. Prior Studies and Our Baseline FMM

There is an extensive literature on the FMM and its parallelization, from fast shared memory implementations to highly scalable distributed memory codes, to GPU-based accelerated implementations [19], [33]–[41]. Within-node, the current state-of-the-art FMM implementation is our own prior multicore code [1], which matches or exceeds an equivalent GPU-based code in both performance and energy efficiency [1], [19], [34], [37]. Therefore, we take this implementation as the baseline for our study.

Baseline code. This baseline has extensive single-core performance enhancements already, including manual SIMD vectorization and "smarter" low-level numerical tricks. It is parallelized using OpenMP, applying the basic `omp parallel for` with static scheduling at the outermost loop of each of the six phases of the FMM (Section III). That is, imagine the FMM being written as the sequence of phases, "upward" followed by "U list" followed by "V list," and so on, where each phase is (roughly) a set of (imperfectly nested) loops; then, our parallel baseline uses `#pragma omp parallel for schedule(static)` at the outermost loop of each phase, respecting dependences. This approach is the minimum level of parallelization we might reasonably expect of any multicore parallelization.

A scalability bottleneck. Although this multicore implementation performs well, it also exhibits a scalability bottleneck as core count increases. We can observe this bottleneck as follows. First, recall that the FMM consists of several phases (Section III). For one common input, a uniform point distribution, the dominant phases are the U list and V list computations. Figure 2 shows, for each of these two phases, its parallel cost, $p \cdot T_p$, where p is the number of threads and T_p is the phase's parallel execution time in seconds, on the Intel Harpertown system. If a phase scales linearly, then its $p \cdot T_p$ is a constant. This behavior holds for the U-list (squares) but not for the V-list (triangles), which becomes the performance bottleneck at four or more cores. Indeed, there is actually a slowdown at eight threads compared to four. Thus, it is in this context that, for the remainder of the paper, we wish to systematically investigate whether and how to improve scalability.

V. Stage I: Diagnosis and Initial Tuning

The process of improving the V-list scaling problem in the baseline code (Section IV) begins with an exploratory analysis. Initially, we assume

- high-level knowledge of the application and algorithm and full access to the source code, but with perhaps limited knowledge of the code;
- a basic understanding of the architecture, such as the properties listed in Table I, including whether the system has a uniform or non-uniform memory architecture;

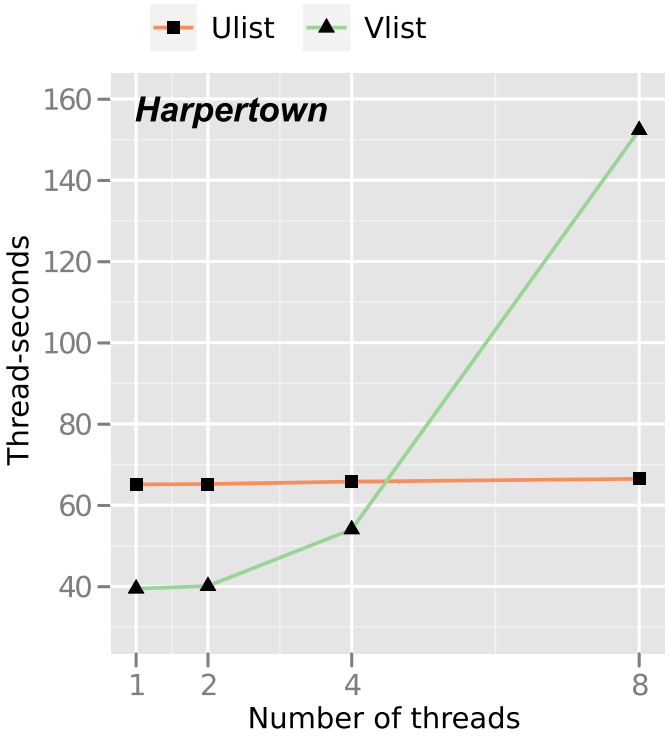

Fig. 2: U list and V list parallel scaling (with OpenMP static scheduling) for an 8 million point problem instance (uniform particle distribution in a cube). Unless specified otherwise, this is the default problem instance used in the paper. The y-axis shows parallel cost, $p \cdot T_p$, so flat lines indicate perfect scaling.

- knowledge of several "rules-of-thumb" analytical techniques, such as Amdahl's Law, Little's Law, notions of arithmetic or computational intensity, and the roofline model [9], [42];
- full access to all of the available performance measurement, analysis, and compilation tools, such as VTune, HPCToolkit, TAU, and Open|SpeedShop, among others [13], [24]–[27].

A. First Measurements and Analysis

Using standard tools, such as VTune, we profile the code to make the initial observation of poor V-list scaling, as outlined in Section IV and Figure 2. Again through performance counter measurement, we estimate the computational or arithmetic intensity (ratio of flops to *main memory* bytes) of these two phases, finding that the U-list is compute-bound while the V-list is memory-bound. Thus, one would expect U-list to scale and suspect memory system contention as the culprit for poor V-list scaling. We set out to discover the cause.

First, we ask whether there is any load imbalance, given our initial choice of static scheduling. Using performance tools, we check the per-thread time and determine that each thread takes nearly exactly the same amount of time. Thus, we rule out load imbalance as a problem.

Given that the V-list is memory bound, we then ask what part of the memory system the V-list may be stressing as threads increase. We recognize that our platform, a dual-socket

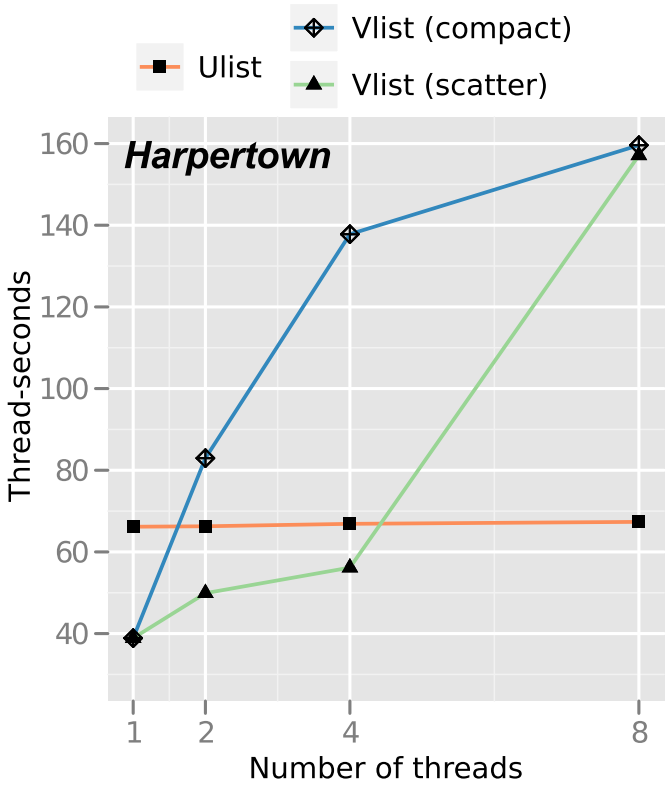

Fig. 3: U list and V list performance scaling on Harpertown with compact and scatter thread pinning schemes.

quad-core Intel Harpertown, has pairs of cores sharing the L2 cache. To gain some additional insight and intuition into what effect this cache architecture has, we conduct two experiments. Importantly, these experiments are simple as neither requires modifying the code.

In the first experiment, we use the OpenMP thread affinity option to "scatter" consecutively numbered threads first across sockets, then across cores not sharing L2, and then within pairs sharing L2. We observe precisely Figure 3. In particular, the code only achieves $3\times$ scaling on four cores, when there is no cache sharing, suggesting that bandwidth contention is at least one specific issue. We also note that the precipitous drop occurs at 8 threads when pairs of threads share the L2 cache. This observation suggests a second experiment, in which we change the affinity policy to "compact assignment," where consecutively numbered threads are first assigned to cores sharing L2, then within the core, then across sockets. If L2 cache sharing is an issue, then we expect to see the scalability issue with even just 2 threads, since they will under compact assignment be mapped to cores that share L2. Indeed, we make exactly this observation, as shown in Figure 3. Thus, we conclude that in addition to bandwidth contention when there is no cache sharing, that there is also the potential for cache contention, most likely due either to capacity or conflict issues. These observations do not yet suggest how to improve V-list scaling, but they do lead to an interesting inference, described next.

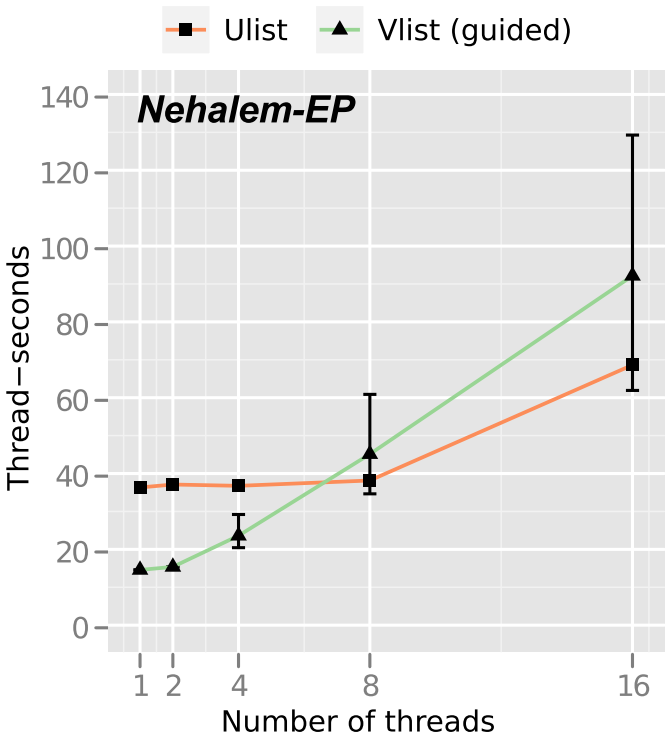

Fig. 4: U list and V list parallel scaling on Nehalem-EP. The black error lines indicate variation in execution time between the fastest and slowest thread with static scheduling.

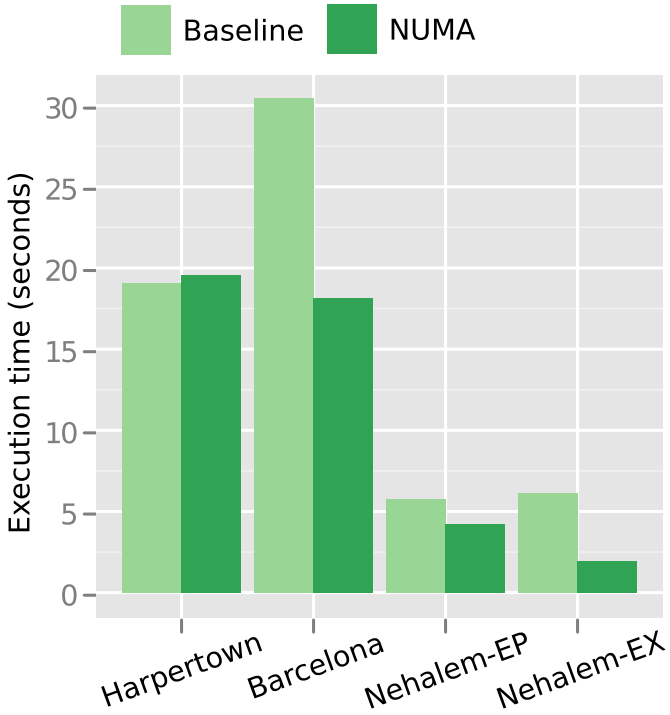

Fig. 5: The impact of NUMA-aware allocation: V list performance at full concurrency and explicit static thread scheduling.

B. Tuning Exploration and Results

Since cache sharing leads to poor scalability, there may be some natural affinity of threads to *independent* pieces of data. That is, if two threads operate on independent data, we would expect no improvement from assigning them to cores that share a cache. We then ask the following question: although Harpertown platform uses a uniform memory architecture, what would happen if we moved to a *NUMA* one? Our observations about affinity imply that we will see even more performance problems there, since data placement will be critical on a NUMA platform. Thus, even though we are not yet ready to fix the V-list scalability problem, we can "future-proof" our code for a NUMA architecture using either of two potential optimizations, one which does not require modifying the code substantially, and one which involves some modest modifications.

Guided scheduling. We run our code on a Nehalem-EP system, which has a NUMA architecture. Whereas we previously observed no load imbalance issues, here we observe one! Figure 4 shows the issue, where, for each thread configuration, we also show both the maximum *and* the minimum execution time of any V-list thread. The ratio between these two is as high as 2×. Checking the flop counts, we see they are roughly the same across threads. As such, the imbalance must be coming from an increase in the cost of some other operation, and for a memory-bound computation, that culprit is most likely the memory operations themselves.

One simple way to cope with any load imbalance is to switch the work sharing scheduling policy, here from static to *guided* scheduling.[2] This "fix" does not address the root cause of improper data placement but does mitigate the problem somewhat, as seen in Figure 4.

NUMA optimizations. Though guided scheduling helped on Nehalem-EP, it still does not address the fundamental problem of data placement. To do so requires modifying the code. In particular, to exploit NUMA, we use the first-touch page allocation policy of the operating system to also parallelize the data initialization loops, a still relatively non-invasive change. The results are shown in Figure 5. Indeed, performance improves by 1.4–3.2×. However, on both Harpertown and Nehalem-EP, the fundamental V-list scalability issue remains.

VI. Stage II: Intermediate-level Tuning

In Stage II, we assume more knowledge of the computation and machine, thereby enabling deeper inferences and more complex performance-enhancing transformations. In particular, we assume in addition to Stage I,

- a more detailed understanding of the processor microarchitecture;
- a deeper understanding of the application's data and control dependency structure;
- consequently, a willingness to try some "riskier" optimization strategies.

Based on the Stage I analysis, we know that the U-list computation is compute-bound and nearly perfectly scalable, whereas the V-list is memory-bound. Although the NUMA

[2] Although we chose guided, one can try others.

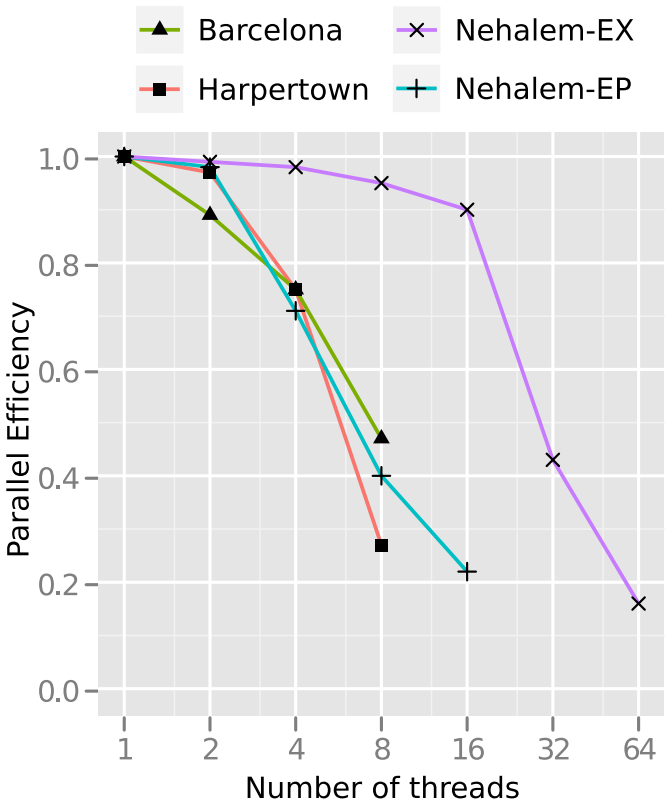

Fig. 6: The parallel efficiency of V list computation after NUMA-aware memory allocation on various platforms.

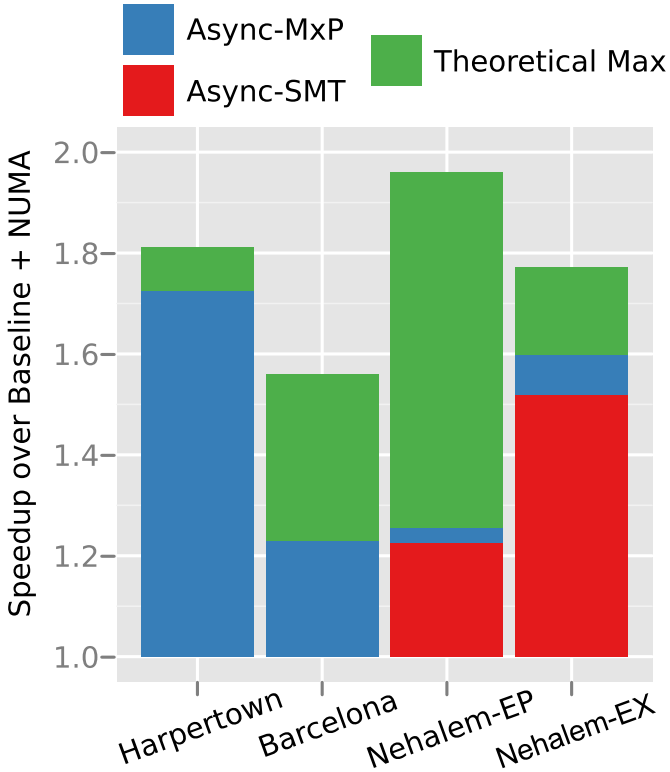

Fig. 7: The performance improvement achieved using asynchronous implementations (red indicates explicit SMT assignment and blue depicts additional improvement with mixed phase execution). The green bar is the an indicator of the theoretical maximum possible speedup (based on U list and V list execution times at full concurrency).

optimizations enhanced performance significantly, there could still be room for improvement.

A. Analysis and Implied Optimizations

The simplest way to avoid the scalability "cliff" is to use at most the number of threads for which the computation still scales, provided the remaining threads can scalably do other useful work. Indeed, in Stage II, being more informed about the code, we discover that the U- and V-list computations are actually independent of one another and may therefore run concurrently. The trade-off is that the code, which was parallelized in the straightforward bulk-synchronous style from its sequential counterpart (Section IV), will require more extensive modification than that of the NUMA-aware optimizations.

In the best case, we make a quick back-of-the-envelope estimate of the best possible improvement from this technique. In the current code, let T_U be the U-list time and T_V the V-list time. If we can successfully hide the entire cost of the V-list, we might achieve a $\frac{T_U+T_V}{\max\{T_U,T_V\}} \leq 2$ speedup. For the four platforms featured in this study, we collect this data and estimate that a 1.56–$1.96\times$ speedup might be possible.

Under this assumption, we generate two ideas for exploiting this concurrency through *asynchronous-parallel* execution.

Idea 1: Mixed phase execution. We consider specific schedules of the U- and V-list components in which some threads operate on V-list work, up to the V-list's scalability limit as seen from Figure 6; the remaining threads work on the U-list, which scales well on any number of cores. The degree of work partitioning can be a runtime parameter subject to (automated) tuning.

Idea 2: Explicit SMT assignment. Since the arithmetic intensities of U- and V-list components differ markedly, we hypothesize that we might profitably exploit the simultaneous multithreading (SMT) features of the Nehalem platforms. The intuition is simply that the U-list computations will make heavy use of the computational units, whereas the V-list is mostly idle waiting for memory. Thus, concurrently executing U- and V-list components on the same core could in principle benefit from SMT, assuming sufficient within-core functional units.

B. Results

We apply these ideas to the NUMA-aware code of Section V. In Figure 7, we summarize the improvement over the NUMA-aware code of the preceding section. Recall that the best possible performance according to our back-of-the-envelope estimate, shown here the highest (green) bar, and ranges from 1.56–$1.96\times$.

For mixed phase execution (blue bar), we see significant improvements of 73% and 60% on Harpertown and Nehalem-EX, respectively. The benefit is somewhat smaller ($\approx 1.2\times$) though still appreciable on Barcelona and Nehalem-EP. This

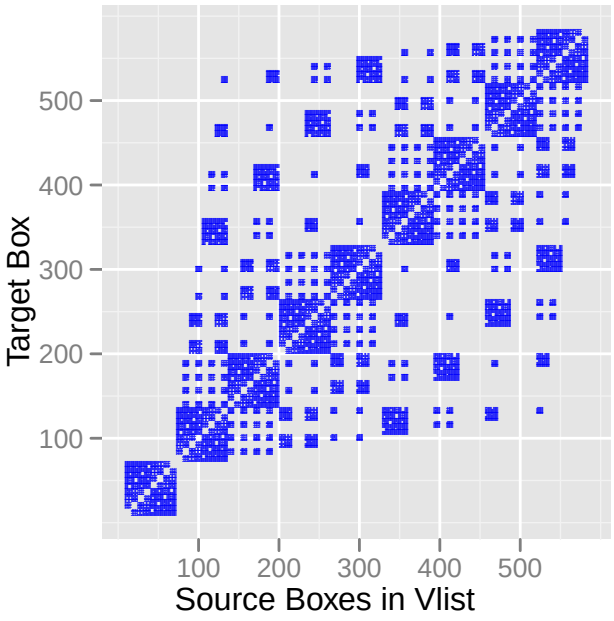

Fig. 8: A plot of the sparsity pattern observed in the V list target box–source box implicit mapping matrix (200K points, uniform random distribution).

observation lends support to the notion of investing in programming, compile-time, and run-time mechanisms that can readily facilitate this style of execution.

On the two SMT-enabled Nehalem platforms, our intuition about explicit SMT assignment does in fact lead to a pay-off that nearly matches that of mixed phase execution, as shown by the red bars in Figure 7. Still, the achieved performance falls short of our estimated ideal in both cases. In fact, although the V-list is memory bound, it is evidently not strictly idle and is likely still competing with the co-scheduled U-list thread for processor-core functional units. We expect that further investigation of the U- and V-list resource requirements could help inform the balance of functional units in future SMT designs.

VII. Stage III: Advanced Tuning, Algorithm Redesign

We now detail some optimizations that one cannot recommend or implement without an in-depth analysis of the computation. We envision that practitioners at this stage of the performance-tuning hierarchy have a detailed understanding of the application, and are also experts in programming multicore architectures.

In the context of our KIFMM code tuning, we attempt to model the computation, analytically validate and reason about the VTune performance counter data (collected in Stage I, as discussed in Section V), and finally evaluate if we can achieve a further performance by investing effort in a complete redesign or restructuring of the algorithm.

A. Modeling U list and V list computations

In Section V, we classified U list and V list as *compute-intensive* and *memory-intensive* respectively, based on performance counter data gathered for a few problem instances. We now describe our methodology of source code inspection and analytic estimation to determine the floating point computation performed and the memory traffic sustained, and validate performance counter data. While the analyses here may not be as generally applicable as the ones discussed in the earlier sections, we hope that the following outline of efforts would give the reader a sense of the complexity involved at this stage.

U list computation. From Table II and from prior analysis by Ying et al. [18], we have that U list's complexity is $\mathcal{O}(27Nq)$. We provide a short explanation for this term. Asymptotically, for a uniform random distribution in a three-dimensional space and rectilinear discretization, we can assume that each target box has 27 neighbors in its U list (we can easily visualize this by counting the number of unit cubes in a $3 \times 3 \times 3$ cube). Thus, we perform a total of $27q^2$ pairwise kernel computations per target box. We have roughly $M = N/q$ such boxes, and so we get to the cost term of $27Nq$. This term ignores the fact that in our experiments, the cube is of a finite size. Accounting for the boundary cases (boxes on the surface), we refine our counting argument to come up with a better approximation: $(3M^{1/3} - 2)^3 q$. Next, we inspect our source code and generated assembly instructions to approximately count the number of floating point operations performed, which would be a multiplicative factor to this pairwise box computations. In our case, we count 19 instructions, which we expect to take roughly 17 cycles after accounting for possible instruction level parallelism (ILP), instruction latencies, and throughputs [43]. Our count agrees very closely with the statistically-sampled VTune performance numbers for Harpertown. We could also verify that the execution time is indeed dominated by floating point computations. Next, the expected number of memory words transferred can be similarly determined based on the average U list neighbor count, and it closely correlates with the observed bus transactions (memory) value. This corresponds to a very small fraction of the total execution time, and hence any further memory optimizations will only be of limited benefit. Our inspection of the inner loop and manual counting of floating point instructions gives us an estimate of any further performance improvements that could be realized for U list computation, either with improved pseudo-code or better architectural support for the higher latency instructions.

V list computation. Based on scalability analysis in prior sections, we suspect that V list performance is likely hindered by cache capacity misses and saturation of available memory bandwidth. Another indicator of this problem is the relative increase in performance counter values as we increase thread-level concurrency, due primarily to bus transactions (memory) and last-level cache misses. Thus, we now attempt to estimate the bus transaction counts and cache working set requirement per thread in terms of FMM problem parameters. Recall that V list's asymptotic computational complexity is given by

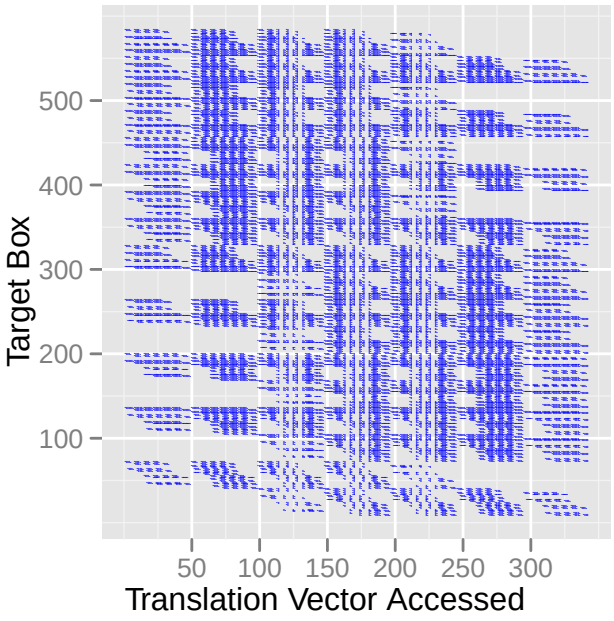

Fig. 9: A plot indicating translation vector accesses in V list computation (200K points, uniform random distribution).

$O(Mp^{3/2}\log p + 189Mp^{3/2})$, where M is the number of target boxes and p denotes the number of expansion coefficients. The first term corresponds to computation of FFT's to transform a convolution into multiple element-wise vector products in Fourier space. The 189 multiplicative constant denotes the asymptotic average V list size for a target box, assuming a uniform random distribution ($6^3 - 3^3$; visualize by counting the number of unit cubes within a $6 \times 6 \times 6$ cube, minus the V list box count). In practice, we observe that $p^{3/2}\log p$ is an over-estimate for each FFT computation. We link to tuned FFTW [44] routines, and this step only constitutes 4% of the total V list execution time.

The vector size after FFTs is determined by the number of expansion coefficients, which is chosen to set the accuracy desired for the KIFMM summations. For four digits of accuracy, the FFT size is 640, and for six-digit accuracy, this is typically 2016. Our code is currently configured for six digits of accuracy. The element-wise vector products are performed between source boxes (i.e., the vectors corresponding to source boxes after FFTs) and vectors chosen from a set of what are referred to as *translation vectors*. The appropriate translation vector chosen for a particular source-target combination depends, among other aspects, on the depth of the source and target boxes under consideration. To summarize, given six digits of precision, for each target box vector, we perform approximately 189 element-wise vector products (source box – translation vector combinations, each of size 2016 elements) in the Fourier space and update then the target box vector. This gives us a better estimate of the computation than the $p^{3/2}$ term in the asymptotic worst-case analysis. The performance counter data accurately matches our estimate.

The count for memory references is slightly more involved. Figure 8 denotes the sparsity pattern of the source boxes that appear in the V list of each target box, for a small problem instance of 200K random particles in a cube. We access target boxes in the order of their identifier (i.e., 1 to N), and the source box FFTs for each target box are contiguously stored in memory. Thus, in the implicit matrix shown in the figure, the memory accesses are row-wise. Each dot in the figure also corresponds to a vector of size 2016. We observe substantial temporal reuse (note the blocked structures along the diagonal of the matrix). Based on geometry arguments, we can provide a weak upper bound estimate of 86% reuse between every consecutive pair of target boxes. (The proof is based on considering unit stride translations of a $5 \times 5 \times 5$ cube in all directions and estimating overlap.) Also, on closer inspection, we see that this matrix pattern is reminiscent of a stencil-like traversal, with approximately 189 points per row of the matrix. We further refine our reuse estimates by manually computing the pairwise overlap. Next, we empirically estimate the reuse distance of every source box as a fraction of the total number of boxes. This number further refines our visual estimate that roughly half the boxes are lumped together along the diagonal, and other analytic arguments based on the geometric structure. We thus conclude that the source box accesses are bandwidth-bound asymptotically, but with a reuse pattern along the diagonal. If we are able to fit these blocked structures (roughly half the average number of boxes per target box, per row, 2016 doubles each) in the last-level cache, we can substantially reduce memory traffic.

The other source of memory references is accesses to the translation vectors. Note that the translation vector count is independent of the number of particles or boxes. In this case, we have roughly 343 translation vectors, and for each target box, about 189 unique ones are accessed. As seen in Figure 9, this pattern is much more complicated to analyze. However, note that the degree of reuse is considerably higher in this case, given that we choose 189 out of 343 for every iteration. This suggests that one may require a minimal cache size for ensuring all of the translation vectors stay close to the processor, and to avoid fetches from memory for every target box evaluation.

For the 8 million point problem on Harpertown, we are able to match the performance counter data for memory bus transactions by up to 99%, essentially by combining both of the preceding arguments and refining our counting estimates to consider the boundary conditions (the average source boxes turns out to be 161 rather than 189 for the 8 million point case, and an even smaller 90 for the 200K-point instance). Further, we can determine precisely the cause for a number of empirical observations in prior sections. We observed an increase in execution time on the Harpertown system when two concurrently-running threads were sharing the L2 cache, due to insufficient per-thread cache size for exploiting the available blocked structure pattern for the source vectors. The bandwidth-bound nature of streaming source boxes is also apparent in all the performance results. For the 8 million-

point problem size, we estimate a per-thread cache working set of 5.2MB for the source boxes and translation vectors. This explains the relatively-good scaling observed on the Nehalem-EX system.

B. Optimization and Results

Based on the preceding analysis, we now suggest algorithmic improvements. One approach to reduce capacity misses is to increase temporal locality, if any. We note that the relatively high access rate of the translation vectors (Figure 9) is a promising avenue to pursue. Next, we observe that threads are executing concurrently rather than cooperatively when streaming the source boxes, and this leads to a p-way pressure on cache utilization.

Based on these observations, we experiment with two ideas for further improving performance.

Idea 1: Blocking of the translation vector. A common optimization in sparse matrix computations is to block a matrix (i.e., stride through it in regular chunks), such that the blocks fit in cache. We could block the translation matrix (essentially the virtual matrix in Figure 9) since it is reused by all the threads. The blocking size can be a runtime parameter or a parameter that can be determined based on the last-level cache size available.

Idea 2: Hybrid scheduling. Since the current static scheduling strategy of assigning fixed chunks of target box computation to each thread results in a loss of temporal locality between consecutive target boxes, we devise an improvement to exploit this locality. Instead of static scheduling throughout, we employ hybrid scheduling. That is, we perform static scheduling across different sockets, but within a socket, if the threads are sharing the cache, we switch to cyclic scheduling. This way, we avoid fetching potentially entirely different source box vectors into the cache, thus alleviating capacity misses.

We apply these techniques individually to the NUMA-aware code of Section V. Figure 11 summarizes the performance when we block the translation vectors across all four platforms considered. We achieve significant speedups from 2.1× to 4× when running at full concurrency. To fit all the translation vectors and all the source boxes in one target box's V-list in cache, we would require 5.5 MB and 2.6 MB respectively. This amount would consume the entire L3 cache of a Nehalem-EP socket (8 MB). Adding an additional thread doubles contention whereas blocking significantly mitigates it.

Similarly, we can first determine whether hybrid scheduling would provide a performance boost before going on to implement it. On the Nehalem-EP, when running at full concurrency, the working set size for the source boxes alone would be 2.6×8 MB per socket. This again exceeds the L3 cache capacity and would result in capacity and conflict misses. Hence we expect this scheduling scheme to help. Figure 11 summarizes the benefit across all the platforms. Performance improves by 1.7× to 3.4×, except on Barcelona. This can be explained by the significantly smaller cache size, where even the working set of a single thread does not fit entirely in cache.

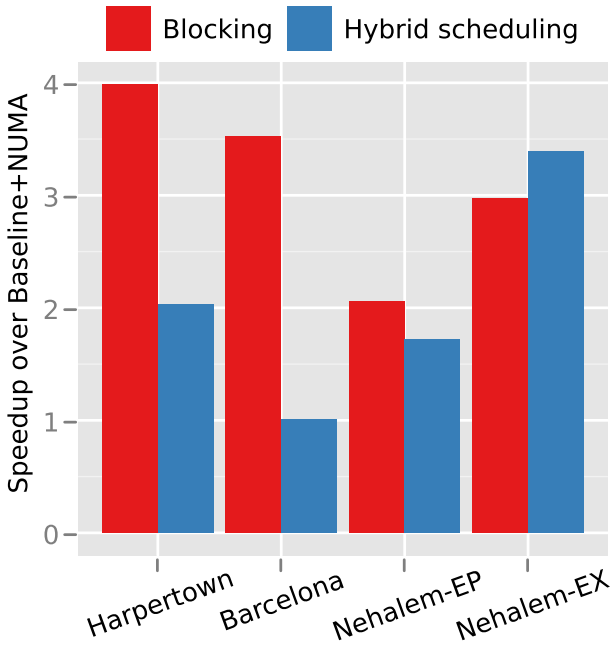

Fig. 11: V list parallel performance on all platforms with blocking and hybrid scheduling optimizations at full concurrency.

Across the board, we observe better scalability after applying the above optimizations as seen from Figure 10 compared to Figure 6.

VIII. Conclusions and Future Directions

Our study builds on the considerable collective wisdom on the principles, techniques, and tools needed for enhancing multicore scalability [2], [6]–[10], [10]–[16]. Indeed, it is inspired by the detailed examples of memory hierarchy transformations for regular computations (matrix multiply) on single-core systems [3]–[5]. In our case, each transformation is motivated by a model, measurement, and/or analysis, resulting in the overall sequence of performance improvements over a state-of-the-art baseline as shown in Figure 12. In terms of within-node FMM performance, our code now represents the new state-of-the-art.

Although our decomposition into stages may seem simplistic, we nevertheless believe that from the perspective of building tools, it might be useful to try to decompose tuning into processes that vary by the amount of information and aggressiveness of program transformation permitted.

Our "Stage I" and "Stage II" analyses could in principle be incorporated into "performance wizards" and "what-if scenarios" for existing or new analysis, tuning, and prediction tools. Except for the code-specific NUMA transformation, the Stage I analyses required only measurement and exploration of tuning parameters that the programming model exposed directly (e.g., choice of OpenMP schedule). As for the NUMA transformation, even if it could not be fully automated, it may be possible that a "smart" performance analysis tool could at least suggest it based on program observation.

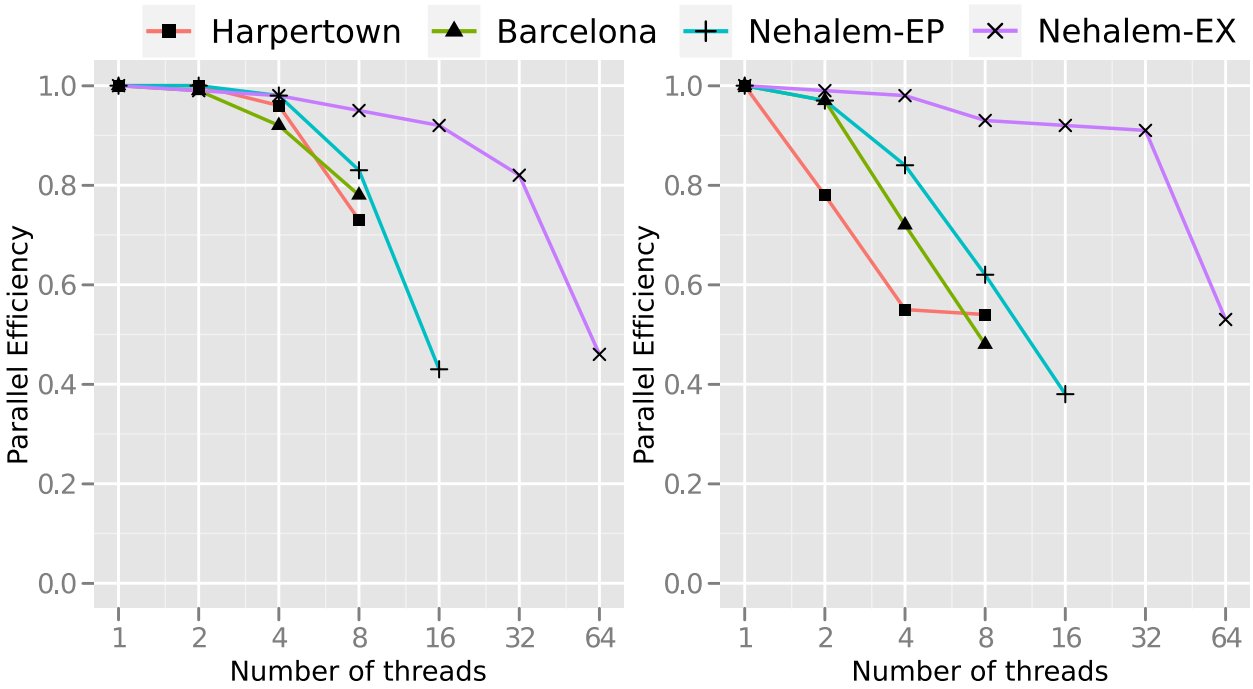

Fig. 10: V list efficiency plots for the optimized implementations with blocking (left) and hybrid scheduling (right) optimizations.

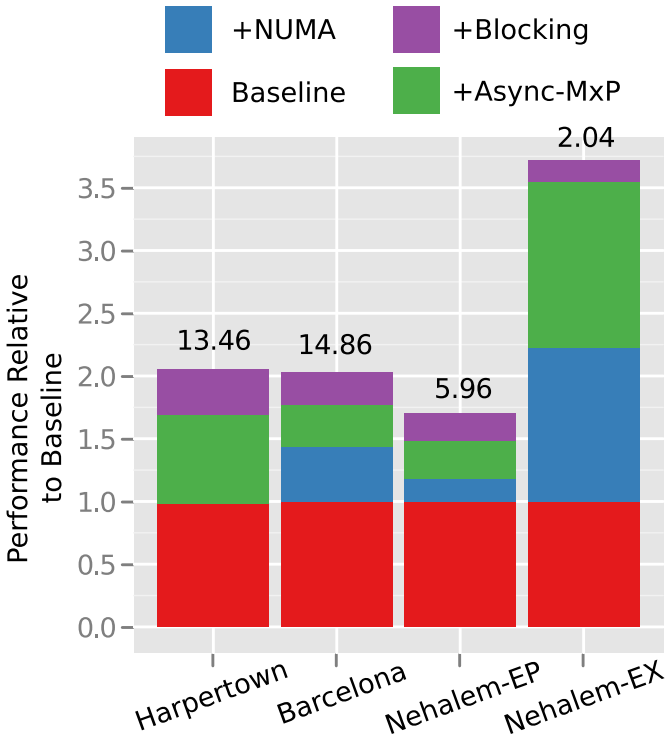

Fig. 12: A summary chart depicting the impact of performance optimizations on each architecture, with speedup given with respect to the baseline implementation (U-list + V-list) at full concurrency. Labels show the final execution time (secs) after all optimizations.

The Stage II transformations, which apply concurrent or "asynchronous-parallel" scheduling and execution, is another example of a trend in multicore performance exploitation [45]–[47]. As a technique for effectively managing bandwidth and locality, as well as exploiting SMT, this approach will become only more important over time, and warrant additional research and development on programming models and systems support for this style of execution.

Stage III constitutes the deepest and most FMM-specific analyses and optimizations. Nevertheless, it also shows the feasibility and utility of analytical modeling, dependency analysis, and transformation for indirect and semi-irregular programs like the FMM. Such programs remain extremely challenging cases.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under award number 0833136, NSF TeraGrid allocation CCR-090024, NSF CAREER award 0953100, joint NSF 0903447 / Semiconductor Research Corporation (SRC) Award 1981, and the U.S. Department of Energy (DOE) under Contract No. DE-AC02-05CH11231, and grants from the Defense Advanced Research Projects Agency (DARPA) and Intel Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, SRC, DARPA, DOE, or Intel.

REFERENCES

[1] 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, (to appear).

[2] R. J. Fowler, T. Gamblin, A. K. Porterfield, P. Dreher, S. Huang, and B. Joó, "Performance engineering challenges: The view from RENCI," *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 125, no. 1, pp. 1–6, July 2008.

[3] K. Yotov, X. Li, G. Ren, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance BLAS?" *Proc. IEEE*, vol. 93, no. 2, pp. 358–386, February 2005.

[4] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, San Diego, CA, USA, June 2007, pp. 93–104.

[5] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Mathematical Software (TOMS)*, vol. 34, no. 12, p. 25pp, May 2008.

[6] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman, "Memory hierarchy performance measurement of commercial dual-core desktop processors," *J. Sys. Arch.*, vol. 54, no. 8, pp. 816–828, August 2008.

[7] G. Hager, T. Zeiser, and G. Wellein, "Data access optimization for highly threaded multi-core CPUs with multiple memory controllers," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Miami, FL, USA, April 2008, pp. 1–7.

[8] A. Mandal, R. Fowler, and A. Porterfield, "Modeling memory concurrency for multi-socket multi-core systems," in *Proc. IEEE Int'l. Symp. Performance Analysis of Systems and Software (ISPASS)*, March 2010.

[9] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Comm. ACM (CACM)*, vol. 52, no. 4, pp. 65–76, April 2009.

[10] F. Song, S. Moore, and J. Dongarra, "Analytical modeling and optimization for affinity based thread scheduling on multicore systems," in *Proc. IEEE Int'l. Conf. Cluster Computing (CLUSTER)*, New Orleans, LA, USA, October 2009.

[11] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proc. ACM Int'l. Conf. Supercomputing (ICS)*, Island of Kos, Greece, June 2008, pp. 368–377.

[12] S. R. Alam, N. Bhatia, and J. S. Vetter, "An exploration of performance attributes for symbolic modeling of emerging processing devices," in *Proc. High Performance Computation Conference (HPCC)*, Houston, TX, USA, September 2007. [Online]. Available: http://www2.cs.uh.edu/~openuh/hpcc07/papers/64-Alam.pdf

[13] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel, "HPCToolkit: Performance tools for scientific computing," *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 125, 2008.

[14] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely, "A genetic algorithms approach to modeling the performance of memory-bound computations," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, no. 47, Reno, NV, USA, November 2007.

[15] H. Mathis, A. Mericas, J. D. McCalpin, R. Eickemeyer, and S. Kunkel, "Characterization of simultaneous multithreading (SMT) efficiency in POWER5," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 555–564, July/September 2005.

[16] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: Computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, 1984.

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

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

[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.

[20] J. Dongarra and F. Sullivan, Eds., *Computing in Science and Engineering: Special issue on The Top 10 Algorithms*. IEEE, 2000, vol. 2, no. 1, pp. 22–79, http://doi.ieeecomputersociety.org/10.1109/MCISE.2000.814652.

[21] J. Board and K. Schulten, "The fast multipole algorithm," *Computing in Science and Engineering*, vol. 2, no. 1, pp. 76–79, January/February 2000.

[22] A. G. Gray and A. W. Moore, "'n-body' problems in statistical learning," in *Proc. Advances in Neural Information Processing Systems (NIPS)*, Vancouver, British Columbia, Canada, December 2000.

[23] P. Ram, D. Lee, W. March, and A. Gray, "Linear-time algorithms for pairwise statistical problems," in *Proc. Advances in Neural Information Processing Systems (NIPS)*, Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, Eds. MIT Press, 2009, pp. 1527–1535. [Online]. Available: http://books.nips.cc/papers/files/nips22/NIPS2009_0436.pdf

[24] "Intel® VTune™ Performance Analyzer," http://software.intel.com/en-us/intel-vtune/, April 2010. [Online]. Available: http://software.intel.com/en-us/intel-vtune/

[25] J. Levon, "OProfile manual," http://oprofile.sourceforge.net/doc/index.html, 2004. [Online]. Available: http://oprofile.sourceforge.net/doc/index.html

[26] S. Shende and A. D. Maloney, "The TAU parallel performance system," *Int'l. J. High Performance Computing Applications (IJHPCA)*, vol. 20, no. 2, pp. 287–311, 2006.

[27] "Open|SpeedShop™, version 1.9.3," http://www.openspeedshop.org/wp/, October 2009. [Online]. Available: http://www.openspeedshop.org/wp/

[28] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Baltimore, MD, USA, November 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=762785

[29] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proc. USENIX Ann. Technical Conf.*, San Diego, CA, USA, January 1996. [Online]. Available: http://lmbench.sourceforge.net/

[30] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995. [Online]. Available: http://www.cs.virginia.edu/~mccalpin/papers/balance/index.html

[31] D. Pase, "pChase benchmark," http://pchase.org, March 2008.

[32] O. Coulaud, P. Fortin, and J. Roman, "High performance BLAS formulation of the multipole-to-local operator in the fast multipole method," *J. Comp. Phys.*, vol. 227, no. 3, pp. 1836–1862, January 2008.

[33] 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. [Online]. Available: http://portal.acm.org/citation.cfm?id=1050165

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

[35] 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.

[36] 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*).

[37] 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.

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

[39] 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 comptuers," *Computer Phys. Comm.*, vol. 153, no. 3, pp. 445–461, July 2003.

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

[41] B. Hariharan and S. Aluru, "Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods," *Parallel Computing (ParCo)*, vol. 31, no. 3–4, pp. 311–331, March–April 2005.

[42] G. M. Amdahl, "Validity of the single processor approach to achieving

large-scale computing capabilities," in *Proc. AFIPS Joint Computer Conf.*, vol. 30, Atlantic City, NJ, USA, April 1967, pp. 483–485.

[43] "Intel® 64 and IA-32 architectures software developer's manuals," http://www.intel.com/products/processor/manuals/.

[44] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE: Special issue on "Program Generation, Optimization, and Platform Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

[45] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures." Innovative Computing Laboratory, University of Tennessee Knoxville, Tech. Rep. UT-CS-07-600 (LAPACK Working Note 191), September 2007, http://www.netlib.org/lapack/lawnspdf/lawn191.pdf.

[46] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn, "SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, San Diego, CA, USA, June 2007, pp. 116–125.

[47] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of Concurrent Collections on high-performance multicore computing systems," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Atlanta, GA, USA, April 2010, (to appear).