Performance Analysis of HPC Applications with Irregular Tree Data Structures

Ahmed Khawaja, Jiajun Wang, Andreas Gerstlauer, Lizy K. John
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas
Email: akhawaja@utexas.edu

Dhairya Malhotra and George Biros
Institute for Computational Engineering and Science
University of Texas at Austin
Austin, Texas
Email: dmalhotra@ices.utexas.edu

***Abstract*—Adaptive mesh refinement (AMR) numerical methods utilizing octree data structures are an important class of HPC applications, in particular the solution of partial differential equations. Much effort goes into the implementation of efficient versions of these types of programs, where the emphasis is often on increasing multi-node performance when utilizing GPUs and coprocessors. By contrast, our analysis aims to characterize these workloads on traditional CPUs, as we believe that single-threaded intra-node performance of critical kernels is still a key factor for achieving performance at scale. Especially irregular workloads such as AMR methods, however, exhibit severe underutilization on general purpose processors.**

In this paper, we analyze the single core performance of two state-of-the-art, highly scalable adaptive mesh refinement codes, one based on the Fast Multipole Method (FMM) and one based on the Finite Element Method (FEM), when running on a x86 CPU. We examined both scalar and vectorized implementations to identify performance bottlenecks. We demonstrate that vectorization can provide a significant benefit in achieving high performance. The greatest bottleneck to peak performance is the high fraction of non-floating point instructions in the kernels.

Keywords—HPC, PVFMM, MANGLL, irregular tree, PAPI, SIMD, AVX , adaptive mesh refinement, Fast Multipole Method, Finite Element Method

I. INTRODUCTION

Traditionally, performance analysis of scientific computing codes for partial differential equations has been focused on regular discretizations, such as the stencil based seven-point Laplacian operator. Such codes have a regular nested for-loop structure that is amenable to extensive optimizations. But such regular stencil codes are no-longer representative of state-of-the-art techniques and cannot be used to predict the performance of currently deployed codes. Current codes, typically employ high-order accurate discretization schemes and nonuniform spatial discretizations. Such discretizations are also known as adaptive mesh refinement methods (AMR). In this paper, we focus on AMR methods that use octrees. Other variants exist, but octree-based AMR methods have been shown to scale to thousands of cores and retain some of the nice properties of uniform spatial discretizations (e.g., memory access and precomputation of element operators). The two AMR methods we consider are the Fast Multipole Method (FMM) and the Finite Element Method (FEM) [1,2,3]. A key issue in obtaining good overall performance in these methods is their single-core performance. As such, in this paper we focus in detail on the performance of these two codes on a x86 architecture.

The two codes that we examine are PVFMM [1] for the FMM scheme and MANGLL [2] for the FEM scheme, both of which are optimized to achieve high performance on high-performance computing clusters. PVFMM is a high-order solver used to solve N-body problems and elliptic partial differential equations (e.g., the Poisson problem) using the fast multipole method. MANGLL is a scalable adaptive high-order nodal discontinuous and continuous Galerkin discretization library for hexahedral elements. In this paper, we only test the MANGLL's discontinuous Galerkin method that is used for hyperbolic partial differential equations. Also, we only test PVFMM's V-list. Both algorithms are explained in greater detail in Section II. PVFMM and MANGLL are chosen as representative candidates for our analysis for three main reasons:

- Hyperbolic and elliptic partial differential equations appear in a myriad of problems in science and engineering. By examining these two state-of-the-art codes for such problems we cover a significant portion of HPC workloads.
- PVFMM and MANGLL implement different mathematical methods and target different classes of problems, but both use octrees and support highly non-uniform discretizations.
- They implement state-of-the-art algorithms and have scaled to 100s of thousands of cores. These two codes are being used to simulate problems in geosciences, energy, and life sciences.

In performing a detailed analysis for these two workloads, this paper makes the following contributions:

- A comprehensive performance analysis of the two octree codes for different parameter settings.
- An analysis of the benefits of vectorization and programmer inserted prefetching.
- A careful evaluation of performance bottlenecks for this class of applications, including reasons for stalls and deviation from peak performance.

The rest of the paper is organized as follows: Section II briefly describes the core algorithms of PVFMM and MANGLL, with an emphasis on showing how irregular tree

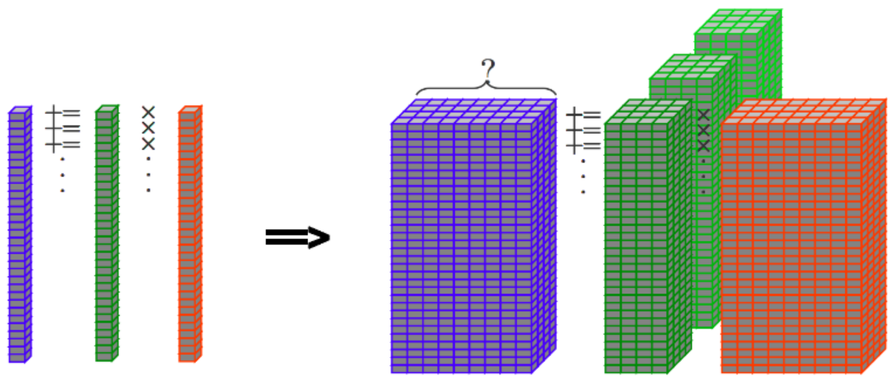

Figure 1: (Left) Naïve Hadamard product computing one interaction at a time. (Right) Optimized Hadamard product evaluated as stack of matrix-matrix products. Matrix width (block size) is a tuning parameter dependent on the cache size.

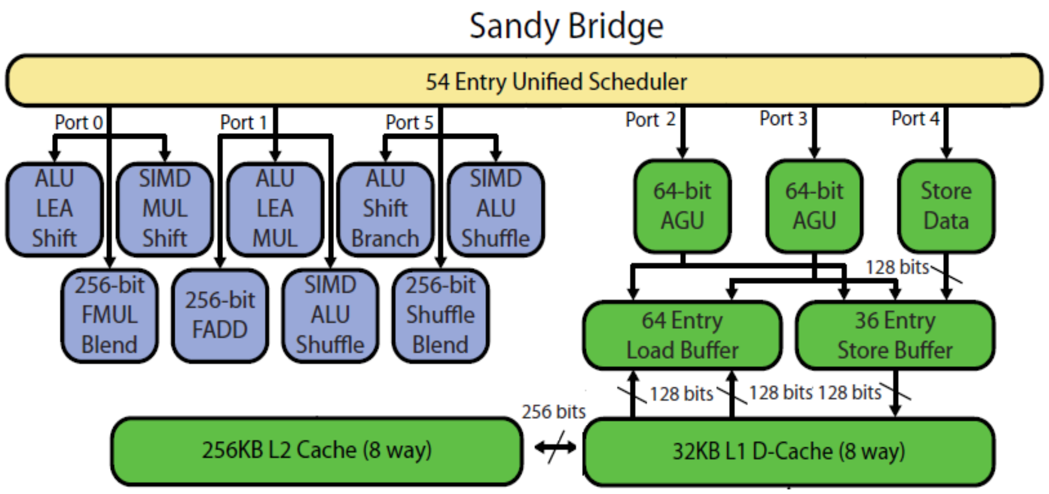

Figure 2: Overview of the Sandy Bridge microarchitecture. The unified scheduler is connected to the 6 uop issue ports. Adapted from [12].

topologies are used in two different kernels. In our analysis, we utilize CPUs with an Intel Sandy Bridge microarchitecture, and we give a brief introduction to this microarchitecture in Section III. Our experimental setup is explained in Section IV, and Section V contains the analysis of factors limiting performance. We conclude in Section VI.

II. Core Algorithms

A. PVFMM

The Fast Multipole Method is used to solve N-body problems. It has optimal work complexity of O(N), for problems with N interacting particles. However, the constants in the complexity estimate can be large. There are several works [4,5,6,7], which give different algorithmic variations of the method and make trade-offs between applicability of the algorithm to different problem types and efficiency. Here, we analyze the variant described in Ying et. al [4], which is applicable to many different problems and is also very efficient. However, even with a work efficient algorithm, it is still necessary to have an optimal implementation. Below, we briefly describe the main features of the PVFMM implementation. For a more detailed discussion, we refer the interested readers to Malhotra and Biros [1].

The PVFMM implementation constructs an octree based partitioning of the computational domain. Next, interactions between octree nodes are computed. There are broadly two interaction types, near and far interactions. Near interactions are computed directly through kernel evaluation for particle problems and dense matrix-vector products for volume potential problems. These interactions have high arithmetic intensity, however, kernel evaluations are generally costly and can involve computing fractional powers and transcendental functions. Hardware support for evaluating such functions will be necessary to improve performance. Until such hardware support becomes available, there is very little which can be done to improve performance of these interactions.

The far-field interactions in PVFMM involve computing a complex-valued Hadamard product. The Hadamard product computation, if done naively, can have very low arithmetic intensity (Figure 1: left). The PVFMM implementation uses a novel data rearrangement, interleaving the vectors (source and target) for sibling octants to form stacks of vectors of length 8. All 8x8 interactions between two sibling groups can then be evaluated as a stack of matrix-vector products where the matrices are 8x8. Several such interactions (in the same direction) can be combined and represented as a stack of matrix-matrix products, evaluated in a level-by-level fashion (Figure 1: right). The small matrix-matrix products are carefully tuned and implemented efficiently using AVX and SSE vector intrinsics (which correspond to 4-wide and 2-wide vectorized x86 instructions, respectively).

Blocks of octants sorted in Morton order are processed one at a time. Interactions in all possible directions are computed while keeping one layer of the stack in cache, thus reusing the source and target vectors already loaded from the main memory. The block size (width of matrices) is a tuning parameter, which depends on the cache size of the processors. The implementation also uses OpenMP parallelism along the height of the stack, allowing for the utilization of all cores on multi-core systems.

B. MANGLL

MANGLL is a library designed for solving scalable adaptive high-order discretizations for hexahedral elements on top of the octree data structure. It can be used in simulations of convection in the Earth's mantle, global seismic wave propagation, and melting of the Antarctic ice sheet. One of its main contributions is supporting dynamic parallel adaptive mesh refinement and coarsening (through an interface to the p4est library [8]), which is essential for numerical solutions of the partial differential equations (PDEs) arising in many multiscale physical problems [9].

The adaptive spatial data-structure used in the MANGLL code is a general, unstructured hexahedral mesh. The mesh is refined using one octree data-structure per hexahedron (thus, the term "forest of octrees") [10]. Octrees are axis-aligned. Given this arrangement, when mapping an octree to a problem domain, the recursive subdivision of octrees corresponds to subdivision of the mapped domain. Hence, such a subdivision generates a mesh which can be used for the discretization and numerical solution of a PDE. Recursive octree divisions support efficient octree-based adaptivity of the mesh.

The essential concept that makes octree-based AMR methods parallel at a large-scale is the space-filling curve, identified by the traversal of an octree across its leaves. By connecting the space-filling curve between octrees, a total ordering of all octants in the domain is generated. The total ordering can be used for fast binary search, finding any of Np local octants in O(log Np) steps.

III. SANDY BRIDGE MICROARCHITECTURE

Our studies are conducted on the Stampede system at TACC (Texas Advanced Computing Center) [11]. It comprises over 6,400+ compute nodes, each containing two 8-core Xeon E5-2680 processors and a Xeon Phi MIC coprocessor. The Xeon E5-2680 processors feature the Sandy Bridge microarchitecture, which was Intel's first to implement the AVX (Advanced Vector Extensions) ISA extensions. AVX, as implemented on Sandy Bridge, doubles the width of packed vector operations and allows 8 double precision FLOPs/cycle, which at a core frequency of 2.7 GHz provides a theoretical capability of 21.6 GFLOPS/core.

The core can decode and retire up to 4 instructions per cycle, giving a maximum IPC (instructions per cycle) of 4. Sandy Bridge features a large 168 entry reorder buffer (ROB) and a unified scheduler featuring 54 reservation stations to enable a large out of order execution window. Register files with 144 floating-point and 160 integer entries enable the entire instruction window to be all floating point, all integer, or a combination of the two[12]. As shown in Figure 2, Sandy Bridge uses a unified scheduler to issue up to 6 micro-operations(uops) over 6 issue ports, with each port providing access to multiple functional units. Floating-point multiply and add operations are issued on two separate ports that are also connected to the integer vector units. Each of these two ports can issue a 256-bit operation (producing 4 double precision floating-point numbers in a single vector) per cycle, providing a theoretical peak of 8 FLOPs/cycle. Two additional ports are for memory loads and have separate address generation units and 128-bit data paths. The address generation units are shared with the store port. A single AVX vector operation that requires a memory operand thereby requires both load units to provide the memory operand. Finally there is a dedicated store port that can leverage both of the address generation units from the load ports. Overall, this means that during peak performance, the core is able to issue both a floating-point multiply and add, two loads, a store, and an integer operation all in a single cycle. This only occurs during peak bursts and is

Variant	Description
PVFMM Basic	Features no blocking, unvectored
PVFMM Blocked	Blocked, unvectored
PVFMM SSE	Blocked, two wide vectored
PVFMM AVX	Blocked, four wide vectored
MANGLL -l 5 -N 4	Compiler optimized, octree forest level of 5, polynomial degree of 4
MANGLL AVX -l 5 -N 4	Four wide manually vectored, octree forest level of 5, polynomial degree of 4
MANGLL -l 4 -N 7	Compiler optimized, octree forest level of 4, polynomial degree of 7
MANGLL AVX -l 4 -N 7	Four wide manually vectored, octree forest level of 4, polynomial degree of 7
MANGLL -l 3 -N 11	Compiler optimized, octree forest level of 3, polynomial degree of 11
MANGLL AVX -l 3 -N 11	Four wide manually vectored, octree forest level of 3, polynomial degree of 11

Table 1: PVFMM and MANGLL variants and a brief description.

PAPI Event Name	Event Description
PAPI_TOT_CYC	Total execution cycles
INSTRUCTION_RETIRED	Instructions executed and retired by the processor
BR_INST_RETIRED	Branch instructions retired
L1-DCACHE-LOADS	Total data memory reads
L1-DCACHE-LOAD-MISSES	Read misses in the L1 data cache
L1-DCACHE-STORES	Total data memory writes
L1-DCACHE-STORE-MISSES	Write misses in the L1 data cache
PAPI_L2_DCA	Data memory reads and writes to the L2 cache
PAPI_L2_DCH	Data access hits in the L2 cache
PAPI_DP_OPS	Double precision floating point instructions
SIMD_FP_256:PACKED_DOUBLE	SIMD floating point instructions operating on 4 double precision floats
FP_COMP_OPS_EXE: SSE_FP_PACKED_DOUBLE	SIMD floating point instructions operating on 2 double precision floats
PAPI_FP_INS	Scalar floating point instructions
UOPS_DISPATCHED_PORT: PORT_X	Micro-operations dispatched over port X, where X is 0,1,2,3,4,5
RESOURCE_STALLS:X	Cycles stalled due to X, where X is ROB, load buffer, store buffer, reservation station

Table 2: List of PAPI events used in the analysis and brief descriptions of what they measure. Events that start with the PAPI are preset events and the others are provided through PAPI's native interface.

possible because x86 instructions are cached in micro-operation form after the initial decode. The retire width is 4 and hence the peak IPC is limited to 4. The capability to issue more than 4 uops per cycle does not violate the maximum of 4 retired IPC, but is necessary for x86 instructions that translate into multiple uops. This capability also enables compensation for cycles when less than 4 micro-operations are issued. Each core has its own L1 and L2 caches, with a shared L3 between all cores. The L1 cache is divided into two 32KB 8-way associative instruction and data caches. The L2 cache is a 256KB unified cache, while the shared L3 cache is 20 MB. Overall, these cores are capable of providing high performance for highly vectorized workloads.

IV. EXPERIMENTAL SETUP

Both PVFMM and MANGLL are capable of scaling to multiple cores/nodes, but since we are trying to measure and improve single core performance, we limited execution to a single core. Both programs were built using the Intel C Compiler (ICC) which provides compiler intrinsics for vector instructions that make it easier to code kernels leveraging these capabilities. ICC is capable of auto-vectorizing code if certain criteria are met to guarantee the correctness of the produced code. In total, we evaluated 4 variants of PVFMM and 2 variants of MANGLL. Each variant of MANGLL was run with three different pairs of input parameters. The pairs consist of an initial octree forest level (denoted as the -l argument) and the polynomial degree (denoted as the -N parameter). PVFMM did not exhibit a sensitivity to its input parameters (both for a given variant and between variants), so their variations are not reported in this paper. The studied variants for PVFMM and MANGLL are listed in Table 1.

Several methods of collecting performance metrics were considered, including gprof [13], VTune [14], and OProfile [15]. We ultimately decided to use PAPI (Performance API) [16] for the following reasons: PAPI collects low level performance metrics in a relatively non-intrusive manner. PAPI builds on hardware counters, which increases accuracy and reproducibility while minimizing the overhead of collecting data. PAPI is susceptible to measuring events that occur as a result of other programs, which we aimed to minimize by using reserved nodes and repeated measurements. The first step in our work was to identify a subset of measurable events (summarized in Table 2) that would allow us to analyze relevant performance characteristics.

Note that, due to the fact that vectorized and unvectorized operations are counted differently, we were required to use different events to measure floating-point operations. One of the sanity checks we used was that we instrumented each program at the source code level to keep track of how many operations should have been executed. These numbers were compared with the PAPI results to validate the execution.

V. RESULTS

A. Floating Point Performance

The primary metrics for assessing overall performance are total execution time and the fraction of theoretical peak floating-point performance achieved during a run. Table 3 summarizes the floating point performance for PVFMM and MANGLL. Percent of peak is calculated based on the theoretical maximum of 8 double precision floating point operations per cycle (FLOPs/cycle) in Sandy Bridge based CPUs. Normalized running time is calculated as the ratio of the number of cycles executed by each variant over the fastest variant. The variants were sorted in order of increasing performance (percentage of peak). There is a direct correlation between sustained floating point performance and total execution time.

Variant	FLOPs/ cycle	Percent of Peak	Normalized Running Time
PVFMM Basic	1.05	13.1 %	4.93
PVFMM Blocked	1.44	18.0 %	3.46
PVFMM SSE	2.95	36.9 %	1.65
PVFMM AVX	4.86	60.8 %	1.00

Variant	FLOPs/ cycle	Percent of Peak	Normalized Running Time
MANGLL AVX -l 5 -N 4	1.10	13.8 %	2.75
MANGLL -l 5 -N 4	1.14	14.3 %	2.67
MANGLL -l 4 -N 7	1.33	16.6 %	1.87
MANGLL AVX -l 4 -N 7	1.42	17.8 %	1.75
MANGLL -l 3 -N 11	1.44	18.0 %	1.06
MANGLL AVX -l 3 -N 11	1.52	19.0 %	1.00

Table 3: PVFMM (top) and MANGLL (bottom) floating-point performance and normalized execution times.

Variant	IPC	Normalized Instruction Count
PVFMM Basic	2.14	4.88
PVFMM Blocked	2.56	4.09
PVFMM SSE	3.04	2.32
PVFMM AVX	2.16	1.00

Variant	IPC	Normalized Instruction Count
MANGLL -l 5 -N 4	1.94	3.05
MANGLL AVX -l 5 -N 4	1.80	2.90
MANGLL -l 4 -N 7	1.90	2.09
MANGLL AVX -l 4 -N 7	1.75	1.79
MANGLL -l 3 -N 11	1.97	1.23
MANGLL AVX -l 3 -N 11	1.70	1.00

Table 4: PVFMM (top) and MANGLL (bottom) instructions/cycle.

PVFMM in general outperforms all MANGLL variants. The fastest executing PVFMM variant (PVFMM AVX) is 3x faster than the fastest MANGLL variant (MANGLL AVX -l 3 -N 11). PVFMM greatly benefits from blocking, prefetching, and vectorization. The basic PVFMM variant represents a naive implementation and runs 4.93 times slower than the hand-optimized AVX variant. The super-linear increase in performance when going from a blocked, non-vectorized PVFMM to a SSE PVFMM is due to prefetching [21]. The SSE/AVX PVFMM variants have programmer inserted prefetches for selected data structures.

The hand-optimized AVX variants of MANGLL either achieve a minor performance increase or a slight degradation in performance. The MANGLL program appears to gain little to no benefit from vectorization. The instruction count decreases by approximately 20% for each AVX variant of MANGLL, whereas in the PVFMM code it is reduced by 4x (see Table 4). Floating-point performance of vectorized MANGLL variants achieves at best approximately a 10% increase. In comparing between different MANGLL parameter sets, we can see that in all cases, floating-point performance is increasing with an increasing polynomial degree.

B. *Instructions per Cycle*

Secondary to execution time and raw floating point performance, we measured the total number of cycles and instructions executed by each kernel. As mentioned previously, Sandy Bridge is a 4-wide superscalar machine, which means that the maximum achievable IPC (instructions per cycle) is 4. Table 4 lists the IPC and normalized instruction counts for all variants.

The SSE variant of PVFMM did manage to achieve the highest IPC of all three variants. This can be attributed to two factors: each SSE intrinsic does more work per instruction than the unvectored variant, and each SSE instruction (as we will see in the follow subsection) is less likely to have a cache miss compared to the AVX variant. Note, however, that the SSE variant executes more than twice as many instructions as the AVX variant leading to an overall lower performance despite the higher IPC.

MANGLL's IPC of about 2 instructions/cycle (irrespective of variant) is overall lower than PVFMM's. Sandy Bridge can simultaneously dispatch up to six uops/cycle across its six ports. The number of uops dispatched is determined by the number of independent uops that are ready (source data is available), the number of uops requiring the same issue port, and the status of the execution units. To investigate the low IPC of MANGLL, we further analyzed how many uops were being dispatched every cycle. As Figure 3 demonstrates, 50 % of the time only 2 or less uops are sent to execution units, which hinders overall performance. We also see that execution is stalled more than 10% of the time. Figure 3 shows that PVFMM performed much better and issued 3 or more uops nearly 50% of the time or better.

However, observing that more than half of the cycles use less than 50 % of the port resources is a sign of inefficiency. This is due to cycles being stalled during execution because of pending cache loads, data dependencies, unavailable execution units, and issue port conflicts (if two uops need to use the same port). Port conflicts can be avoided, at the expense of increased hardware costs, by either providing extra ports or increasing the width of the ports. Such changes would be beneficial to two AVX instructions trying to reference memory in a single cycle. The inability to compensate for some loss in performance might be greatly outweighed by the hardware costs and, as we propose in the next section, can be regained via different means.

C. *Cache and Memory Performance*

Caches are an important target when identifying performance bottlenecks, which is why it was one of the main areas of optimizations when PVFMM was implemented.

Variant	L1 Miss Rate	L2 Miss Rate	L3 Miss Rate
PVFMM Basic	9.10 %	11.55 %	4.44 %
PVFMM Blocked	5.47 %	81.5 %	4.76 %
PVFMM SSE	8.96 %	86.8 %	5.12 %
PVFMM AVX	16.39 %	76 %	7.40 %
MANGLL -l 5 -N 4	2.75 %	31.1 %	22.67 %
MANGLL AVX -l 5 -N 4	2.93 %	29.96 %	27.47 %
MANGLL -l 4 -N 7	4.18 %	17.49 %	25.89 %
MANGLL AVX -l 4 -N 7	2.23 %	27.39 %	31.25 %
MANGLL -l 3 -N 11	6.94 %	7.38 %	28.22 %
MANGLL AVX -l 3 -N11	5.56 %	8.60 %	31.94 %

Table 5: PVFMM and MANGLL L1 (data), L2, and L3 cache miss rates. MANGLL has low L1 and L2 miss rates, but has fairly high L3 miss rates. PVFMM has low L1 and L3 misses rates, but suffers from very high L2 cache misses. Miss rates should always be considered with total instruction count, as we have done in Table 6.

Variant	L1 MPKI	L2 MPKI	L3 MPKI	Ideal CPI	Ideal + MCPI
PVFMM Basic	61.5	5.38	0.23	1.00	1.88
PVFMM Blocked	13.9	9.49	0.45	1.00	1.61
PVFMM SSE	20.6	14.4	0.74	0.50	0.97
PVFMM AVX	40.7	24.0	1.77	0.25	0.68
MANGLL -l 5 -N 4	11.9	3.71	0.84	1.00	1.44
MANGLL AVX -l 5 -N 4	12.8	3.84	1.05	0.25	1.49
MANGLL -l 4 -N 7	18.1	3.17	0.82	1.00	1.47
MANGLL AVX -l 4 -N 7	13.4	3.68	1.15	0.25	1.51
MANGLL -l 3 -N 11	30.7	2.26	0.64	1.00	1.53
MANGLL AVX -l 3 -N 11	33.2	2.86	0.91	0.25	1.63

Table 6: PVFMM and MANGLL cache misses per 1,000 instructions (MPKI) and memory adjusted CPI estimate assuming L1 (data), L2, L3 miss latencies of 10, 40, 200 cycles.

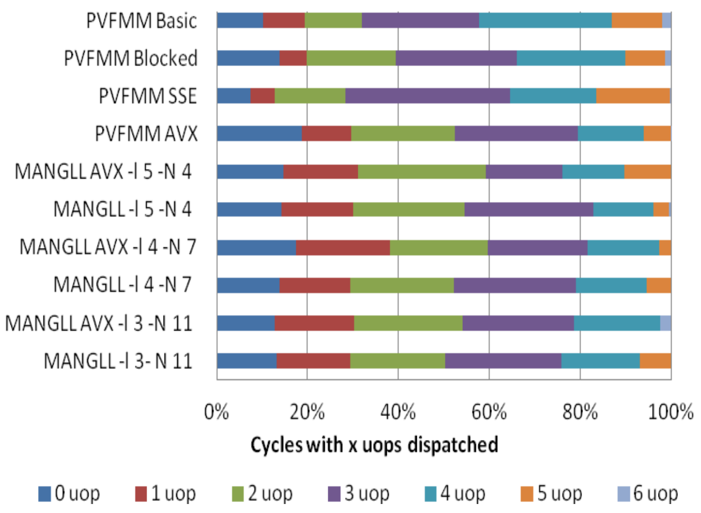

Figure 3: PVFMM and MANGLL breakdown of uops dispatched per cycle. PVFMM AVX spends roughly 20% of its cycles without issuing any uops which is indicative of full-window stalls caused by long latency cache misses.

PVFMM relies on blocking and profile guided prefetching to optimize cache performance. MANGLL currently does not have manually inserted prefetching. The two primary metrics to consider when analyzing cache performance are miss rate and misses per instruction (MPI). Miss rate is the ratio of misses to total accesses for a cache level. A miss at a given cache level results in an access to the next level cache (e.g. an L3 cache miss results in a main memory access). Table 5 gives the miss rates for all three cache levels for all PVFMM and MANGLL variants. The PVFMM AVX variant had the worst L1 cache miss rate, despite having the best performance overall. With the exception of the basic PVFMM, most PVFMM variants had high L2 miss rates. Across all variants, PVFMM had low L3 miss rates. All MANGLL variants had low L1 miss rates, which is significant because it translates into less lower level (further from the CPU) cache accesses. This means that despite MANGLL's L3 miss rate being relatively high, the impact is minimal.

Comparing miss rates can be deceiving as they do not factor in total number of accesses. MPI is a better metric for determining the cache performance of an application, as it is able to determine the relative significance of a miss. Table 6 presents the misses per 1,000 instruction (MPKI) for all PVFMM and MANGLL variants. MPKI in MANGLL is strongly dependent on the parameter set. The basic PVFMM variant had poor L1 cache performance, suggesting that PVFMM greatly benefitted from blocking and prefetching. In order to approximate the effect of cache misses we calculated the memory adjusted cycles per instruction (MCPI) as shown in Table 6. This calculation was done assuming L1, L2, L3 miss latencies of 10, 40, 200 cycles [23]. Sandy Bridge does feature memory bank interleaving that does expose memory level parallelism that we have not considered. The MCPI shows that the AVX variant of PVFMM is impacted the most from cache misses. We also see this trend for all AVX variants of MANGLL. However, unlike in vectorized PVFMM variants, the AVX MANGLL variants were not consistently better or worse than their regular counterparts. The MCPI of vectorized variants suggests to us that cache performance needs to be improved to better accommodate wider vectorization. It still needs to be explored if MANGLL can benefit from manually inserted prefetching.

D. Instruction Mix

We categorized uops as either branches, loads, stores, floating-point, or other operations. We determined that using uops provided a clearer picture of the types of operations that were being executed internally, especially in regards to the other category. Uops that would fall under the other category are primarily ALU type operations that would be used to calculate addresses. We had to use different PAPI events to measure floating-point uops based on the variant under analysis. This was done due to the fact that PAPI treats vectored and unvectored floating-point uops differently (it also differentiates between 128-bit and 256-bit SIMD floating-point operations).

Variant	Fraction FP	Fraction Supporting Instructions	Supporting Instructions/ Floating-point
PVFMM Basic	0.45	0.55	1.22
PVFMM Blocked	0.56	0.44	0.79
PVFMM SSE	0.49	0.51	1.04
PVFMM AVX	0.56	0.44	0.79
MANGLL -l 5 -N 4	0.24	0.76	3.17
MANGLL AVX -l 5 -N 4	0.22	0.78	3.55
MANGLL -l 4 -N 7	0.27	0.73	2.70
MANGLL AVX -l 4 -N 7	0.25	0.75	3.00
MANGLL -l 3 -N 11	0.35	0.65	1.86
MANGLL AVX -l 3 -N 11	0.20	0.80	4.00

Table 7: PVFMM and MANGLL floating-point to support instruction comparison. PVFMM manages to achieve high performance due to having a high fraction of floating-point operations. MANGLL suffers due to having a significantly higher fraction of supporting instructions compared to floating-point operations.

Figure 4 presents the uop mixes for PVFMM and MANGLL. The X-axis corresponds to the type of uop and the Y-axis is the fraction of the total uop count accounted for by that instruction type. For both programs, branches account for about 2% to 6% of the uop mix, which results in very high instruction locality and large basic blocks, exposing a large amount of instruction level parallelism. This low fraction of branch uops is typical of most scientific workloads. Loads accounted for about 23% of the uops in all PVFMM variants except the basic variant, in which over 48% of the uops being loads. This demonstrates that PVFMM greatly benefits from blocking, which is responsible for a drastic reduction in load uops. For MANGLL, loads accounted for a narrower range

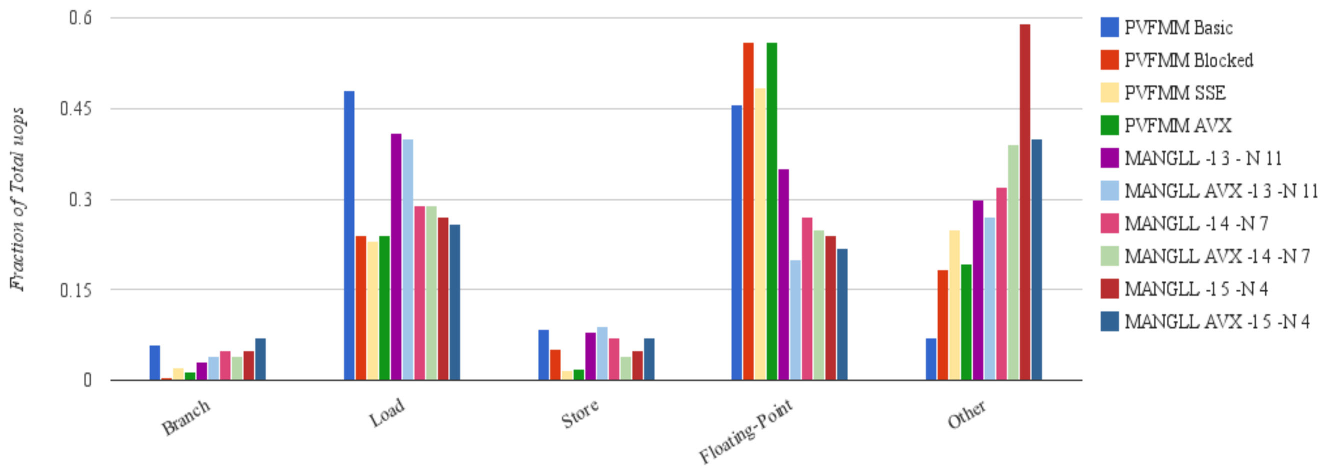

Figure 4: PVFMM and MANGLL uop mix.

between 35% and 44% of the total uop mix. The MANGLL variant with the highest load percentage is the hand-optimized MANGLL AVX -l 3 -N 11 variant, which means that some extra work (more support operations) is introduced to allow vectorization to take place. All other MANGLL variants had very similar load percentages. Since manually vectorizing MANGLL resulted in only marginal performance gains, this suggests that vectorization gains can be negated by the additional overhead that is introduced in the process. For both applications, stores accounted for substantially less of the instruction count than loads. We again see that blocking benefited PVFMM by cutting down the store percentage from 8.5% to 5% or less. In all MANGLL variants, stores accounted for between 4% to 9% of the total uop mix, which, similarly to PVFMM, is much less than the load percentage. Other uops in PVFMM accounted for 7% to 25% of the uop mix in all variants. In all MANGLL variants, other uops accounted for approximately 25 to 50 % of the uop mix. Overall MANGLL had a much higher proportion of non-floating-point uops when compared to PVFMM.

E. *Stalls*

The final class of metrics we considered in our analysis was cycles stalled and the correlation to CPI (cycles per instruction). As Figure 5 shows, reservation station (RS) stalls were the major source of stall cycles for both PVFMM and MANGLL. Reservation station stalls are caused when a uop is either waiting for a source operand or functional unit to become available. The floating-point units are fully pipelined, meaning a uop can be issued per port per cycle, which means most of the RS stalls are due to waiting for source operands. For all PVFMM variants these reservation stalls account for the majority of all stalls. This leads us to believe that the two primary factors holding PVFMM performance back are inherent data dependencies and waiting for values from the cache. The PVFMM AVX variant stalls more often than the SSE variant, and due to the fact that it executes less cycles, this means it spends more cycles stalled both in absolute and relative terms. All MANGLL AVX variants exhibit similar stalling behavior when compared to their non-AVX counterparts.

In Figure 5, we attempt to explain the observed CPI using the various stalls. The Sandy Bridge architecture has an ideal CPI of 0.25 since the ideal IPC is 4. Hence if all stalls of an instruction are added to the ideal CPI, one must get the observed CPI if the stalls are not overlapped. We add the various stalls per instruction and the resulting stalls per instruction is plotted in Figure 5 along with the actual observed CPI. In 9 out of 10 cases experimented, if the ideal CPI of 0.25 is added to the stalls per instruction, one can obtain the observed CPI. Apparently there are some unaccounted stalls in the base PVFMM code which we are further investigating. It may also be noted that the stalls per instruction are about half as much as the ideal CPI in PVFMM, whereas the stalls per instruction in MANGLL are more than the ideal CPI (i.e. more than 0.25). There is a lot of room for improvement in MANGLL.

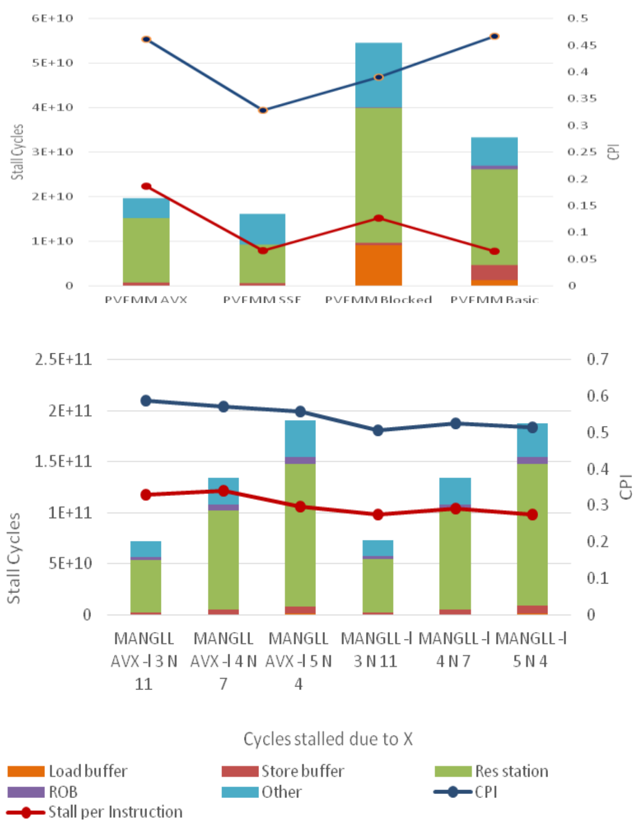

Figure 5: PVFMM and MANGLL stall cycles, stalls per instruction, and CPI.

F. *Access Execute Balance*

The primary focus of our research was on analyzing performance bottlenecks of PVFMM and MANGLL, such that future studies can focus on architectural changes aimed at increasing performance in these and similar workloads. Our primary finding was that the dynamic instruction count includes a large fraction of non-floating-point overhead instructions, especially in MANGLL.

One may view the instruction stream as composed of the core compute instructions and other instructions required to support the compute process. For instance, the access instructions are simply an outcome of the fact that the data is in memory and needs to be brought into the processor to perform the computation. The Decoupled Access Execute (DAE) paradigm [17,18,19,20,21] treats all access instructions as supporting instructions to the compute thread. Past research has investigated the access-execute balance of programs [22] separating the instruction stream into compute and supporting threads. We present the access-execute balance of PVFMM and MANGLL in Table 7, by taking a ratio of supporting and compute instructions. It shows that 8 out of the 10 experimented codes have a high proportion of supporting instructions compared to the core computing instructions.

In order for compute instructions to happen at peak speed, the supporting instructions have to finish their tasks in perfect overlap with the compute instructions. The supporting instructions must also execute early enough to bring the data into the registers in such a fashion that the compute instruction does not have to wait. If the access instruction encounters a cache miss, the register will not get the data in time for the compute instruction to proceed without stalls.

The machine level parallelism available to support the supporting instructions is also of issue. In the Sandy Bridge architecture, 2 of the 6 uops are compute and hence 4 uops can be of the supporting type. Hence the Sandy Bridge architecture can take a maximum access execute imbalance of 2. However, Table 7 shows that the access-execute imbalance of many of the codes studied are 2.5 or higher. In such codes the supporting thread will consume more cycles than the compute thread and hence peak flops cannot be obtained. The SSE and AVX variants of PVFMM are well-balanced, whereas MANGLL is very unbalanced explaining the observed performance.

VI. CONCLUSION

This paper presents an analysis of the single-node performance of two HPC applications with tree data structures, PVFMM and MANGLL. Analysis of PVFMM demonstrates that vectorization, cache tuning, and programmer inserted prefetching can greatly increase performance. With these optimizations, PVFMM achieves 61% of the peak flop rate of 8 double precision flops possible in the Intel Sandy Bridge architecture. Vectorization in PVFMM places increased pressure on the cache and results in a loss of some performance. PVFMM represents a mature code base, while MANGLL is relatively new. MANGLL is not fully vectorized and ongoing effort is going into changing this. PVFMM scaled well because the supporting non-compute instructions scaled down with the reduction in floating-point instructions, which unfortunately did not happen in MANGLL. Supporting instructions comprised a large portion of the MANGLL kernel and we believe significant code changes or architectural changes are necessary to increase performance in light of this finding.

ACKNOWLEDGEMENTS

This material is based upon work supported by AFOSR grants FA9550-12-10484 and FA9550-11-10339; and NSF grants CCF-1337393, CCF-1117895, CCF-1218474, OCI-1029022, and OCI-1047980; and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0010518, DE-SC0009286, and DE-FG02-08ER2585. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFOSR or the NSF.

REFERENCES

[1] D. Malhotra and G. Biros. "A Distributed Memory Fast Multipole Method for Volume Potentials." Institute for Computational Engineering and Science, University of Texas at Austin. padas.ices.utexas.edu/static/papers/pvfmm.pdf

[2] C. Burstedde, et al. "Scalable adaptive mantle convection simulation on petascale supercomputers." Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. 2008.

[3] C. Burstedde, et al. "Extreme-scale AMR." Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. 2010.

[4] L. Ying, G. Biros, and D. Zorin. "A kernel-independent adaptive fast multipole algorithm in two and three dimensions." Journal of Computational Physics 196.2 (2004): 591-626.

[5] L. Greengard and V. Rokhlin. "A new version of the fast multipole method for the Laplace equation in three dimensions." Acta numerica 6 (1997): 229-269.

[6] Z. Gimbutas and V. Rokhlin. "A generalized fast multipole method for nonoscillatory kernels." SIAM Journal on Scientific Computing 24.3 (2003): 796-817.

[7] W. Fong and E. Darve. "The black-box fast multipole method." Journal of Computational Physics 228.23 (2009): 8712-8725.

[8] C. Burstedde, L. Wilcox, and O. Ghattas. "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees." SIAM Journal on Scientific Computing 33.3 (2011): 1103-1133.

[9] J. Diamond, et al. "Multicore optimization for ranger." TeraGrid Conference. 2009.

[10] H. Sundar, et al. "Parallel geometric-algebraic multigrid on unstructured forests of octrees." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2012.

[11] “Stampede User Guide”. Texas Advanced Computing Center. https://portal.tacc.utexas.edu/user-guides/stampede

[12] D. Kanter. "Intel's Sandy Bridge Microarchitecture." www.realworldtechnolgoies.com, 2010.

[13] S. Graham, P. Kessler, and M. McKusick. "Gprof: A call graph execution profiler." ACM Sigplan Notices. Vol. 17. No. 6. ACM, 1982.

[14] R. Malladi. "Using Intel® VTune™ Performance Analyzer Events/Ratios & Optimizing Applications." (2009).

[15] J. Levon. "OProfile manual." Victoria University of Manchester (2004).

[16] P. Mucci, S. Browne, C. Deane, and G. Ho. "PAPI: A portable interface to hardware performance counters." Proceedings of the Department of Defense HPCMP Users Group Conference. 1999.

[17] J. Smith. "Decoupled access/execute computer architectures." ACM SIGARCH Computer Architecture News. Vol. 10. No. 3. 1982.

[18] D. Talla and L. John. "Mediabreeze: a decoupled architecture for accelerating multimedia applications." ACM SIGARCH Computer Architecture News 29.5 (2001): 62-67.

[19] M. Farrens, P. Ng, and P. Nico. "A comparison of superscalar and decoupled access/execute architectures." Proceedings of the 26th annual International Symposium on Microarchitecture. 1993.

[20] J. Parcerisa and A. González. "The latency hiding effectiveness of decoupled access/execute processors." Euromicro Conference. Proceedings. 24th. Vol. 1. IEEE, 1998.

[21] A. Tondon. M.S. Thesis. "Performance Impact of Programmer-Inserted Data Prefetches for Irregular Access Patterns with a Case Study of FMM Vlist Algorithm", The University of Texas at Austin, December 2013.

[22] L. John, V. Reddy, P. Hulina, and L. Coraor. "Program balance and its impact on high performance RISC architectures." Proceedings of the International Conference on High-Performance Computer Architecture, 1995.

[23] D. Levinthal. "Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors." Intel Performance Analysis Guide (2009).