# On the Programmability and Performance of Heterogeneous Platforms

Konstantinos Krommydas, Thomas R.W. Scogland, Wu-chun Feng
Department of Computer Science, Virginia Tech
{kokrommy, tom.scogland, wfeng}@vt.edu

***Abstract*—General-purpose computing on an ever-broadening array of parallel devices has led to an increasingly complex and multi-dimensional landscape with respect to programmability and performance optimization. The growing diversity of parallel architectures presents many challenges to the domain scientist, including device selection, programming model, and level of investment in optimization. All of these choices influence the balance between programmability and performance.**

**In this paper, we characterize the performance achievable across a range of optimizations, along with their programmability, for multi- and many-core platforms – specifically, an Intel Sandy Bridge CPU, Intel Xeon Phi co-processor, and NVIDIA Kepler K20 GPU – in the context of an n-body, molecular-modeling application called GEM. Our systematic approach to optimization delivers implementations with speed-ups of $194.98\times$, $885.18\times$, and $1020.88\times$ on the CPU, Xeon Phi, and GPU, respectively, over the naïve serial version. Beyond the speed-ups, we characterize the incremental optimization of the code from naïve serial to fully hand-tuned on each platform through four distinct phases of increasing complexity to expose the strengths and weaknesses of the programming models offered by each platform.**

***Keywords*-performance, programmability, optimization, AVX, GPU, Intel MIC, NVIDIA Kepler K20, Xeon Phi, CUDA, OpenACC**

## I. Introduction

Many application areas, including finance, life sciences, physics, and manufacturing, have begun to use computational co-processors such as graphics processing units (GPUs), field programmable gate arrays (FPGAs), digital signal processors (DSPs), and even customized application-specific integrated circuits (ASICs) to achieve substantial gains in performance per watt and performance per dollar over traditional CPU implementations. Each of these solutions require programmers to adopt a different programming mindset than the typical, and well-studied, multi-core programming paradigm. This shift in mindset decreases the perceived programmability of these devices, and in turn, increases the cost to optimize and maintain code. While many scientists and industrial programmers possess a working knowledge of basic programming concepts, they typically lack expertise in parallel programming. Programmability of a parallel platform is consequently a deciding factor in its adoption by such audiences.

Each platform is attempting to bridge the gap between performance and programmability in its own way. The Intel Xeon Phi co-processor attempts to ease programmability by offering a standard Linux environment on the device, which can be programmed with standard multi-core programming techniques. GPU and compiler vendors seek to increase the programmability of GPUs via extensions to familiar CPU interfaces, such as the development of the OpenACC directives to provide OpenMP-like functionality for fundamentally non-CPU architectures. In each case, there are highly programmable but imprecise and, comparatively, low performance interfaces as well as extremely difficult but high performance interfaces. The push and pull between programmability and performance comes down to a balance between cost and benefit, that is how much performance you can get and for how much effort.

In this paper, we characterize the programmability and performance of multi- and many-core processors across a range of optimization levels, starting from naïve serial CPU code and extending to fully optimized CPU, Xeon Phi, and GPU code. Rather than skipping directly to the fully hand-tuned optimized versions for each target platform, we realize multiple versions of our molecular modeling code (i.e., GEM [1]) at different levels of optimization, ranging from the most programmable to the least and correspondingly from the worst performing to the best. Our contributions are as follows:

- An analysis of the trade-off between performance and programmability across various levels of optimization on CPU, Xeon Phi, and GPU.
- A characterization of architecture-aware optimizations and their portability across architectures.
- An evaluation of the effectiveness of directive-based parallelism along with compiler-assisted vectorization versus hand-tuned alternatives.

In the next section, we provide background information on our case-study application called GEM and on the multi- and many-core platforms used in our study — Intel Sandy Bridge CPU, Intel Xeon Phi and NVIDIA Kepler GPU. Section III discusses the compiler-assisted parallelization approaches that we evaluate and addresses programmability issues. In Section IV, we describe the mapping and optimization process onto the above multi- and many-core platforms. Section V presents our experimental set-up and results, along with a rigorous performance evaluation and comparison between our CPU, Xeon Phi, and GPU implementations. Finally, we discuss related work in Section VI and conclude in Section VII.

## II. Background

In this section, we provide an overview of the molecular modeling application that we use as a case study in this paper, i.e., GEM [1]. Then, we provide background information on

the Many Integrated Core (MIC) architecture that underlies Intel Xeon Phi and on the NVIDIA GPU Kepler architecture.

### A. GEM: Modeling the Electrostatic Surface Potential (ESP) of Macromolecules

Molecular modeling refers to the mathematical models that seek to describe the behavior and properties of biological molecules and the corresponding computational techniques. An important part of molecular modeling simulation in areas like materials science, computational chemistry, and rational drug design is the calculation of electrostatic surface potential (ESP) in support of locating bonding sites and other features.

The computational pattern of GEM [1] is an all-to-all n-body interaction between points near the molecular surface and the atoms within the biomolecule. The overall result of ESP calculation, i.e. the electrostatic map of a biomolecule, provides useful information about its function. The long-range nature of electrostatic interactions results in a computationally intense workload of order $O(nm)$ where $n$ is the number of atoms and $m$ is the number of surface points. Many approximation methods have been proposed over the numerical solutions of the Poisson-Boltzmann equation [2] that constitute the core of traditional ESP calculation algorithms. One such method is employed by GEM, the ESP application we study in this paper.

In GEM, the biomolecule is divided in three distinct regions, and separate functional forms of the electrostatic potential $\phi_i$ apply for each.[1] The electrostatic potential at each point near the surface (*vertex*) is the sum of electrostatic potentials contributed by each single point charge to that point. Similarly, the sum of potentials at all surface points define the total electrostatic potential of the system. The above computation and communication pattern classifies GEM as an n-body dwarf [4] with the subtle difference that it performs all-pair computations between two sets (versus one). Thus, we expect many of our conclusions regarding performance and programmability of GEM to apply to most n-body applications.

### B. Platforms

In the context of optimizing GEM, we evaluate three distinct parallel platforms and their attendant programming models. The baseline multi-core CPU is represented by Intel's Sandy Bridge x86-64 CPUs, specifically two Xeon model E5-2680s, and uses the C language and the Intel compiler suite with Intel OpenMP directives for parallelism. Our CPU platform is indicative of a standard server node with cache-coherent, moderate-latency NUMA memory; large well-tuned caches; and all the niceties of traditional "fat" CPU cores.

Moving to the Intel Xeon Phi, much of the CPU architecture is preserved. The instruction set is highly similar to that of the x86-64 CPUs and can be natively programmed by the same interfaces. In fact, our evaluations in this paper use the same libraries and compilers at all phases for both the CPU and Xeon Phi. Even so, the Xeon Phi differs substantially at the architectural level. The Xeon Phi uses multiple banks of high-throughput but high-latency graphics memory and offers 512-bit SIMD units and four thread contexts on each core, double the width offered by AVX and double the thread contexts on the Sandy Bridge CPUs. Thus, the Xeon Phi architecture shifts the compute/memory ratio to favor throughput rather than latency-centric computing. In the same vein, the cores are comparatively simple in-order cores with only minimal prefetching support.

Finally, the GPU, represented by an NVIDIA K20c, eliminates the cache-coherent memory offered by the other platforms. Otherwise, the GPU is architecturally more similar to the Xeon Phi. Both use graphics memory and wide SIMD units to offer high throughput and many thread contexts to mitigate the effects of latency. The difference is in the programming model. Since GPUs are SIMD engines, GPU programming models such as CUDA have no concept of running a single thread with scalar mathematics. Instead, the programming model assumes that many threads will execute every instruction in SIMD fashion. While in the other platforms, SIMD support is either compiled in or added with intrinsics; in CUDA/OpenCL, SIMD is the standard state of affairs, and single-threading must be produced manually.

## III. Programmability

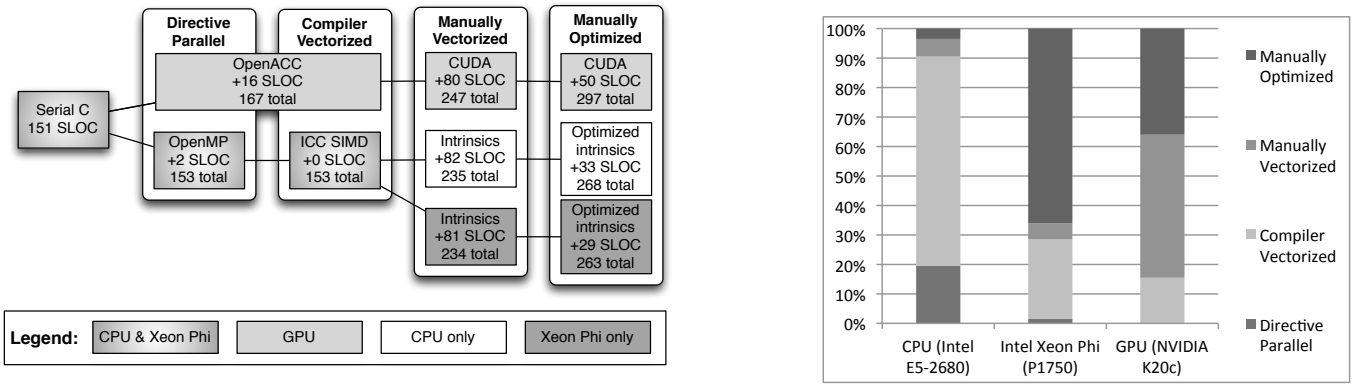
Here we provide insight on the programmability of the platforms that we evaluate, both from a qualitative and quantitative standpoint. In order to provide a quantitative metric for programmability, or more generally, code complexity, we use the classical *source lines of code* (SLOC) metric. We also include code examples and discussion to provide a more qualitative "feel" for the programming models in terms of portability, readability, and maintainability. As the level of optimization increases, programmability decreases. To show the benefit gained at each level, we discuss the percentage of best performance achieved at each step.

### A. Optimization Levels

Figure 1a provides a high-level overview of the optimization levels that we evaluate, starting from the original serial implementation and concluding with the manually hand-tuned implementation for each of the three platforms. We describe each method and evaluate its programmability aspects.

**Directive-based parallelization:** The first set of implementations uses OpenMP and exploits the compiler support for the CPU and Xeon Phi to provide hinted multithreading as well as OpenACC, a variant of OpenMP for the GPU, which we will discuss in further detail below. Using OpenMP, the programmer can exploit all cores in a compatible device with the sole inclusion of the OpenMP library and the appropriate OpenMP directive on each section that should execute in parallel. This straightforward approach also facilitates code portability. With the computational kernel of the application unchanged, the same source can be compiled to serial code or run on systems with any number of cores. While OpenACC at first appears to provide an equivalent approach for GPUs,

[1] Details for each of the regions and the corresponding functional forms are given in [3].




(a) (b)

Fig. 1: (a) The progression and instantiation of each level of optimization on each architecture with the number of Source Lines of Code (SLOC) used in each implementation. (b) The percentage of best achieved performance achieved with each level of optimization.

it is inherently *both* multi-threaded and vectorized, hence its spanning of both *directive parallel* and *compiler vectorized*.

**Compiler-assisted vectorization:** Modern compilers can transform scalar arithmetic to vector arithmetic for regular algorithms and loops. This set of implementations makes use of this compiler feature available in the Intel compiler for multi-core CPUs and Xeon Phi co-processor and in the PGI compiler suite for GPUs via the native vectorization that comes with compiling OpenACC for GPUs. The approach of the latter bears many similarities with OpenMP and the Intel compiler's offload model, combined with the newly released (and unimplemented as of our testing) OpenMP SIMD directives.

As with the Intel compiler, various parameters/hints can be used to tune OpenACC regions for better performance. For example, OpenACC defines clauses to tune the division of loop nests across parallel blocks and threads, (*gang* and *vector* parameters) and control the independence, or lack thereof, of iterations in a given loop. However, OpenACC initialization routines and data movement directives require an additional 16 lines of code. In contrast, for the CPU or Xeon Phi, *no extra lines of code are needed*; only the appropriate setting of compiler flags is needed. In all cases the serial compute code is retained entirely in its original form and can be compiled to that serial version without alteration.

**Manual vectorization:** Explicit use of SIMD intrinsics offers far greater control over the vectorization of any given algorithm. Therefore, it can be worth abandoning automatic cross-architecture compatibility and manually vectorizing the code. This is the phase where the CPU and Xeon Phi codes diverge. While they each employ similar vector intrinsics, they have different vector widths, and thus must use different registers and different sets of intrinsics. For this phase, the CPU and Xeon Phi require an additional 82 SLOC. For the GPU, the corresponding approach uses the CUDA programming model directly, which implicitly specifies all computations as vector operations and requires significant setup and data-management code to be added. These operations require 80 additional SLOC, nearly the same number needed for transitioning between the corresponding optimization levels for the CPU and Xeon Phi.

```
float sum2=(1.f/d_int−1.f/d_ext)/(one_plus_a_b∗A);
```

(a)

```
__m512 sum2_vect=_mm512_div_ps (
    _mm512_sub_ps (
        _mm512_div_ps (ONE,D_INT),
            _mm512_div_ps (ONE,D_EXT)
        ),
    _mm512_mul_ps (ONE_PLUS_A_B,A)
);
```

(b)

```
__m256 sum2_vect=_mm256_div_ps (
    _mm256_sub_ps (
        _mm256_div_ps (ONE,D_INT),
            _mm256_div_ps (ONE,D_EXT)
        ),
    _mm256_mul_ps (ONE_PLUS_A_B,A)
);
```

(c)

Fig. 2: (a) Scalar/CUDA code. (b) Vector intrinsics code for Xeon Phi. (c) Vector intrinsics code for Sandy Bridge CPU.

This level of optimization imposes extra intellectual burden on the programmer, specifically "thinking in parallel is required." For the CPU and Xeon Phi, the process is quite similar, as we see in Figures 2b and 2c. Each employ compiler intrinsic functions to explicitly specify the vector operations to use. To CPU optimization veterans, this may look familiar, but otherwise it obscures the intent of the code significantly. Alternatively, Figure 2a shows the line of code as it is in serial C, OpenMP, OpenACC, or CUDA, the computation remains visually the same. The CUDA version does the same thing as the explicit vector instructions in Figures 2b and 2c, but it preserves the readability of the original. The burden on the GPU is mostly in the setup required to call GPU kernels, but it leaves the computational kernel largely unchanged. The CPU and Xeon Phi are the reverse at this level, requiring virtually no setup but a great deal of changes in the computational kernel.

**Manually optimized code:** For the final set of implementations, we provide an extra level of optimizations for each of the evaluation platforms by explicitly using blocking/tiling, shuffling, and explicitly altering the specific hardware instructions used to target faster execution, such as fused multiply-add (FMA) and approximate reciprocal division/square root. More

details about the best set of optimizations for each platform and a detailed description are given in Section IV.

In CUDA, the applicable optimization techniques are quite different than those used in typical x86 code. The optimization search space itself is bigger as well, with minor changes in the code severely affecting performance (e.g., data structures, memory coalescing, and efficient use of the memory hierarchy). In optimizing for the Intel MIC architecture, details about the underlying architecture (e.g., memory hierarchy and interconnect details) are essential, but Intel MIC's resemblance to traditional multi-core CPU architectures implies similarity in the optimizations, most of which parallel programmers are already familiar with.

### B. Performance Impact

So far, we have discussed the programming effort required for each optimization level and provided the number of SLOC as a rough quantitative measure. In Figure 1b, we show how close to the best achieved performance for each platform we get with each optimization level. We observe that different levels of optimization help reach best performance at a different rate, depending on the platform. In the case of the CPU, directive parallel and compiler-assisted vectorization help attain over 90% of the best achieved CPU performance. Auto SIMD, in particular, accounts for 71.05% of the achieved performance. Considering the above, the programmer can rely on the extensive and highly mature CPU compiler infrastructure along with OpenMP and still attain high performance.

On the other hand, manual optimizations are quite important for both the GPU and Xeon Phi. OpenACC is a relatively new standard and its correspondingly young compilers can only take performance so far. Explicit use of CUDA or very careful tuning is essential for achieving acceptable performance. Manual optimizations are required to fill the last 35.9% of the gap between the performance attained through use of naïve CUDA code and the best achieved performance.

Finally, in the Xeon Phi case, using CPU code directly on the device in the first two levels delivers extraordinary programmability, but also a very low percentage of the best possible performance. Auto-vectorization makes a significant difference, but even with that and manual vectorization, our implementation only reached 35% of the best performance that we achieved overall. Manual optimizations are the most important (65.7% of overall performance) for Intel MIC, due to its high sensitivity to caching behavior. We discuss these optimizations and their effect on performance in detail in Section IV and Section V.

We note that the above conclusions refer to the case of n-body class problems or, more generally, data-parallel workloads with an emphasis on floating-point arithmetic. Different problem classes might benefit less from directive-based multithreading or compiler-assisted vectorization due to irregular data access patterns or complex dependencies. In such cases, manual SIMD and more complex threading using appropriate synchronization constructs (e.g., semaphores and barriers) would be of paramount importance.

| Instruction | Vector register | | | |
|---|---|---|---|---|
| LOAD | v0 | v0 | v0 | v0 |
| LOAD | a0 | a1 | a2 | a3 |
| ... | ... | ... | ... | ... |
| LOAD | v3 | v3 | v3 | v3 |
| LOAD | a0 | a1 | a2 | a3 |

⇒

| Instruction | Vector register | | | |
|---|---|---|---|---|
| LOAD | v0 | v1 | v2 | v3 |
| LOAD | a0 | a1 | a2 | a3 |
| SHUFFLE | a1 | a2 | a3 | a0 |
| SHUFFLE | a2 | a3 | a0 | a1 |
| SHUFFLE | a3 | a0 | a1 | a2 |

Fig. 3: Transformation for shuffling optimization (example).

## IV. Mapping and Optimization

In this section we discuss the optimizations applied across each platform. Most of the optimizations presented are beneficial for all platforms under consideration. Removal of conditional statements and flattening of data structures have been applied to the serial CPU version we use as a baseline. We explicitly mention when an optimization only applies on a subset of the target platforms.

**Vectorization and multithreading:** Since GEM is a data-parallel n-body code, each output potential can be calculated independently of all others. This state is commonly referred to as "embarrassingly parallel," and makes the first and most important optimization the use of parallelism to divide the workload across as many compute resources as possible. On the CPU and Xeon Phi, we use all thread contexts across all cores, as well as all SIMD lanes wherever possible. We use hand-tuned AVX/MIC vector code to pack and operate on 8/16 atoms at a time for a given vertex. GPUs, featuring an abundance of thread contexts, allow mapping the potential calculation for each given vertex to a separate GPU thread.

**Removal of conditional statements:** Conditional branches incur execution time overhead on all three platforms, despite efficient branch prediction on the CPU. Since the conditionals in the parallel portion of GEM are all pre-determined, rather than diverging on a per-vertex basis, all of them can be hoisted out to a single conditional nest used to choose a final computational function with no conditionals in it. This saves us both dynamic instructions as well as potential branch mis-predictions on all devices at the cost of having several replicated versions of the function expressing each necessary code path.

**Flattening of data structures:** Laying out data as an array of structures (AoS) can seriously impact vector code performance. The AoS layout is a major cause of misaligned (in CPUs) and non-coalesced (in GPUs) memory accesses. More importantly, AoS complicates the mapping of data to vector units. For example, given a structure of two ints A and B, the AoS layout intersperses As with Bs, forcing at least two vectorized gather loads to load a vector register. If, on the other hand, the As are in one array and Bs in another, only a single load is required. In GEM, we transform the AoS used to store the coordinates and charge of surface points (*vertices*) and atoms, into multiple arrays, each containing a single component (e.g., charge) of the structure.

**Approximate reciprocal instructions:** Floating-point division and square root are high-latency operations that stall the

pipelines of the CPU and Xeon Phi devices as well as working in a lower-width mode on the CPU. In order to avoid as many of them as possible we replace them with their low-latency approximate reciprocal counterparts. These instructions have much lower latency and make use of look-up tables to calculate the result. Their drawback is their reduced accuracy. On the Sandy Bridge CPU they are accurate to the 12 most significant bits of the mantissa. We tackle the reduced accuracy problem by using an iteration of the Newton-Raphson (NR) method, which increases accuracy to a minimum of 23 of 24 bits for single precision numbers. More details about this method can be found in [5]. On Xeon Phi, the corresponding reciprocal instructions natively provide accuracy of 23 of 24 bits of the mantissa. On K20, the corresponding *__frsqrt_rn()* intrinsic we use is fully IEEE-compliant. For the two latter cases, we do not need to apply Newton-Raphson, keeping the number of instructions lower than in the CPU (Figure 4c). In any case, the root mean squared error (RMSE) of calculated potential values for all implementations against the original serial version's output does not exceed 0.000084.

**Outer loop unrolling:** Each of the $m$ iterations of the outer loop of an n-body problem entails computation against a set of $n$ bodies of the inner loop. In ESP calculation this corresponds to surface points (vertices) and atoms, respectively. As a result, each iteration of the outer loop requires $n$ memory loads (all atoms that contribute to the potential of a given vertex). By "unrolling" the outer loop by a factor $k$ (i.e, calculating potential at $k$ vertices at a time), we reduce the innermost loop's atom loads by the same factor.

**Cache blocking and software prefetching:** Converting arrays of structures to multiple arrays enhances spatial locality and cache use efficiency in all three platforms. Moreover, the algorithm's regular memory access patterns facilitate hardware data prefetching in our multi-core platforms. However, the large number of atoms in the innermost loop leads to eviction of relevant atom data from the lower level caches, before they are fully reused. To alleviate this problem we apply cache blocking, where each thread loops over all its assigned surface points and calculates potential contribution for blocks of atoms at a time. Block size is theoretically calculated, based on the data size accessed with each iteration and cache details, and experimentally tuned and verified. Finally, the programmer can assist the hardware prefetcher by emitting the prefetch intrinsic with appropriate prefetch distance as a parameter.

**Shuffling method:** This optimization is not applicable to all kinds of algorithms, but is especially useful in specific n-body problems, as in our case. In this method (Figure 3), we change the default computation pattern, where each (same) vertex point is loaded at all positions of a vector register and loops over all atoms. Instead, we load $N$ *distinct* vertices at a time in a vector register, load $N$ atoms in another vector register (where $N$ is 8 for AVX) and then shuffle (i.e., rotate in a wrap-around fashion) the data elements of the latter, using the corresponding shuffle intrinsics. This way we can obtain all the possible combinations of vertices and atoms, as defined by the algorithm's all-to-all computation pattern, with a reduced number of vector loads. In addition to CPUs and Xeon Phi, Kepler architecture has introduced a shuffle instruction that achieves similar functionality in the context of a thread warp.

**8-byte shared memory access (in Kepler):** Kepler GPU architecture features 32 shared memory banks, 8-bytes wide each, with a corresponding bandwidth of 8 bytes/bank/clock per streaming multiprocessor (SMX). Default mode defines 4-byte access to support backward compatibility and similar bank-conflict behavior with Fermi, a behavior that leads to sub-optimal bandwidth for certain access patterns. To exploit the Kepler-supported 8-byte access mode, the programmer needs to use the appropriate CUDA function and then, in the case of GEM, transform floating point (FP) type variables to *float2* type variables in a suitable manner (e.g., six FP ones to three float2).

## V. Results and Discussion

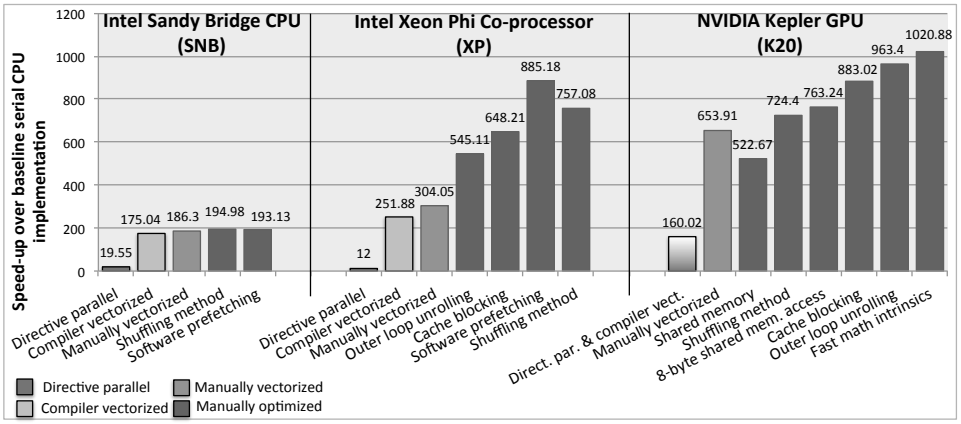
In this section we describe the experimental setup and discuss the effects of the application of optimizations on each platform in terms of performance. At the end of this section, we discuss the best optimization methods set for each platform and perform a cross-platform comparison of the corresponding implementations. Finally, we discuss their efficiency with respect to the theoretical peak throughput, as experimentally derived by means of assembly code inspection and taking into account each platform's specifications.

### A. Experimental Setup

We evaluate our GEM [1] implementations across various optimization levels on three multi- and many-core platforms: a Sandy Bridge CPU (SNB), Xeon Phi co-processor (XP), and Kepler K20 GPU (K20), as noted in Figure 4b. Results for all structures we experimented with, which comprised a different number of vertices and atoms, show similar trends, which is characteristic of n-body methods once the workloads are big enough to saturate available computation units on each platform. For brevity, we only present results for the *tobacco ring virus capsid (1A6C)* biomolecular structure, which requires ESP calculation between 593,615 surface points and 476,040 atoms. In all experimental runs, we use the full parallelization available on the platform 32 threads on SNB, 240 threads on XP (leaving one core for the system software) and 1024 threads per block on the GPU with enough blocks to cover the workload (theoretically achieving 100% occupancy according to the NVIDIA occupancy calculator). Our results are all reported based on the runtime of the computational kernel, setup and data transfer costs are not included. While these costs are important, our focus in this paper is the effectiveness of each level of optimization on each platform, which is independent of the data transfer costs and unaffected by them.

### B. Performance Progression by Architecture

In Figure 4a we present speed-up over the reference single core implementation. Optimizations are cumulative as we move to the right for each platform, unless otherwise noted and the *Manually vectorized* results include the approximate




(a)

| Model (Architecture) | Intel E5-2680 (Sandy Bridge) | Intel Xeon Phi P1750 (MIC) | NVIDIA K20c (Kepler) |
|---|---|---|---|
| Frequency | 2.7 GHz | 1.09 GHz | 706 MHz |
| Cores | 16 (8/socket) | 61 | 13 SMXs |
| Threads/core | 2 | 4 | 16 (blocks/SMX) |
| SIMD (SP) | 8-way | 16-way | 192-way |
| GFLOPS (SP) | 691.2 | 2092.8 | 3524.35 |
| Mem. BW | 102.4 GB/s | 320 GB/s | 208 GB/s |
| L1/L2/L3 cache (KB) | 32/256/20480 | 32/512/- | 48(max)/1280/- |
| Power | 260W | 300W | 225W |
| Compiler | ICC 13.0 | ICC 13.0 | NVCC 5.0 |

(b)

| | SNB | XP | K20 |
|---|---|---|---|
| SP FP vector ops | 52 | 35 | 32 |
| Cycles/iteration | 36 | 25 | 23 |
| Ideal vector IPC | 1.444 | 1.4 | 1.391 |
| Theoretical peak throughput (GFLOPS) | 499.2 | 1465 | 2451.7 |
| Achieved throughput (GFLOPS) | 413.8 | 1265.1 | 1332.3 |
| Achieved efficiency | 82.89% | 86.35% | 54.34% |

(c)

Fig. 4: (a) Step by step optimizations. (b) Architectural parameters. (c) Achieved performance over theoretical peak.

reciprocal instructions. Below we conduct a quantitative analysis of the effect of each optimization series on each platform.

*1) Sandy Bridge CPU (SNB):* Looking at the performance of directive-based parallelization and compiler-assisted vectorization, we observe that the compiler proves very efficient, when compared to the manually vectorized implementation that makes explicit use of vector intrinsics. This is the case both when we use accurate and approximate versions of the intrinsics and the corresponding compiler flags. On SNB we observed, however, that when using the approximate reciprocal intrinsics (included in *Manually vectorized*), we get even better performance compared to using compiler flags for approximate division and square root (*Compiler vectorized*). The reason is apparent in assembly code level. Intel Compiler is not able to perform the algebraic changes we manually make to accommodate the fast reciprocal square root and division intrinsics. As such, the optimized code only uses approximate fast division and fast square root. Even in this case, we obtain performance improvement against using the regular division and square root. The main reason is that the execution unit used for division and square root is still 128-bit and 256-bit packed division/square root is broken down into two 128-bit operations. Using the shuffle method, as described in Section IV, we achieve an extra 4.7% improvement. This optimization reduces the number of vertex loads by a factor equal to the SIMD vector width divided by the size of the data type we are using (e.g., 8 for AVX and float data type). It also helps reducing the number of times atoms' coordinates are loaded, by the same factor. See Figure 3 for a visual explanation. While the number of loads are reduced by a factor of 16 the achieved speed-up is much less impressive, as the loads occur at worst case in our SNB's large (20MB) L3 cache, which is fast by itself and even faster when combined with efficient hardware prefetching to this and lower -and faster- cache levels. Finally, using software prefetching does not offer any significant performance benefits in the case of multi-core CPUs. The reason behind this behavior is the advanced hardware prefetching capabilities of modern multi-core CPUs. For algorithms with regular memory accesses the hardware prefetcher can efficiently move data between the main memory and L2 or between L2 and L1 caches ahead of time based on previous access patterns.

*2) Xeon Phi (XP):* In contrast to SNB, we observe that using vector intrinsics on XP is slightly better than compiler vectorized code (1.20x). As in SNB, use of fast reciprocal math prevents bottlenecks in the corresponding units observed in the exact division/square root cases. The performance gap after multi-threading and vectorization have been applied has to be filled by manual code optimizations. In the case of architectures, such as Intel MIC, where there is a lack of large L3 cache, techniques that make efficient use of the available cache hierarchy are of great importance. One such technique, whose effectiveness on that aspect is algorithm-specific, is outer loop unrolling. In our case, where the outer loop's vertices loop over the same set of atoms, unrolling the outer loop by a factor of two instantly reduces memory accesses by a factor of two and increases performance by 1.79x.

Cache blocking techniques enhance cache usage and result in a 1.19x speed-up versus not using them. Software prefetching instructions, when added on top of the earlier optimizations, yield an additional 1.37x improvement. As opposed to SNB, software prefetching is important in XP. One reason is that in XP, the hardware prefetcher proactively loads data between memory and L2 cache, but not from L2 to L1. This gap can be filled by blocking for L1 cache or software prefetching.

Last, we should mention that the shuffling method we used for SNB is a technique worth trying on XP, as well. As a matter of fact, shuffling by itself reduces atom coordinates' loads by a factor of 16 (i.e., SIMD-width). However, using shuffling with the optimizations we mentioned earlier results in a slight slowdown, as it mainly contributes unnecessary overhead, since outer loop unrolling, cache blocking and software prefetching address the expensive main memory transfers

in an efficient way.

*3) Kepler K20 GPU (K20):* For K20 the naïve CUDA version performs 4.08x faster than the one produced by using OpenACC directives. This is as far as we can get by directive-based programming or naïve CUDA programming and compiler directives. To get anywhere beyond this performance we need to resort to lower-level optimizations. Ensuring coalesced global memory accesses is one of the first optimizations one has to consider on the GPU but since ours already are, optimization efforts should be geared towards utilizing shared memory for data that are accessed by all threads in a block, such as the atom coordinates and charge. We should note that even when not using shared memory, the regular memory access patterns facilitate caching. As a matter of fact, our shared memory implementation boosts the number of registers used and along with shared memory limitations leads to reduced occupancy and performance, with respect to the preceding implementation. Adding the shuffling method to a shared memory implementation boosts performance to 724.4x over the single core CPU implementation, 10% faster than the preceding implementation. For K20 using one float2, instead of 2 float variables, allows successive 8-byte words allocation in successive banks and increased 8-byte wide shared memory access. This results in a 1.054x speed-up. Blocking, which proved to increase performance on SNB and XP, is beneficial for K20 as well. In particular it accounts for an extra 1.16x. Finally, we perform similar algebraic changes as the ones we described for SNB and XP and make use of the fast reciprocal square root and division CUDA intrinsics, together with fused multiply and add instructions. An extra 6% performance gain is achieved by this optimization, leading us to the fastest of our K20 implementations at 1020.88x over the baseline.

### C. Intrinsics and Approximation

Last, but not least, we leave the discussion of further optimizing the code using special intrinsics for fast approximate versions of instructions such as square root, division, fused multiply-and-add. Compilers, such as Intel Compiler (icc) and NVIDIA CUDA Compiler (nvcc) provide flags for automatic detection of the regular instructions (or combinations thereof- such as multiplications followed by an addition). Indeed, inspecting the assembly/PTX code respectively, we verify that both icc and nvcc make use of the corresponding fast instructions (given the algebraic changes mentioned in Section IV). We should note that without performing these algebraic changes none of the compilers were able to automatically perform all the aforementioned optimizations, which is a field of further research. It is also worth mentioning that nvcc provided more efficient code with minimal use of manual intrinsics and the `-use_fast_math` parameter than our fully intrinsic based version. Given the nature of the problem, there must be a manual set which would behave as well, but the compiler does better than most. On the other hand, manually adding the intrinsics under consideration on the XP implementation drastically improved performance.

On the surface it might sound as though the NVIDIA CUDA compiler is performing more advanced conversion of instructions than those in the Intel compiler. The truth is somewhat more complicated. Since the CUDA programming model is implicitly vectorized, it does not require intrinsics to specify the intended width of instructions. In practice it just assumes all instructions are of width 32 and masks off the extra. On the other hand, the standard programming model used on the XP is serial and must be explicitly vectorized. Once intrinsics are used to ensure the correct vector width, it appears that they are not converted by the compiler even though it would have the right given the supplied options. Since intrinsics are meant to be a way to directly insert a particular instruction, it makes sense that the compiler does not change it, but it restricts the compiler from performing a potentially important set of transformations on those instructions. Adding explicit vectorization to the programming model without intrinsics, either through directives such as the `simd` directive in OpenMP 4.0 or through a language extension like CUDA, should solve this issue.

### D. Performance Efficiency

While we discussed the effectiveness of optimizations in terms of *percentage of best achieved performance* in Section III-B, that only covers the percentage of the best performance we managed to achieve, not the peak performance possible from each device. By examining the assembly/PTX code of the best performing implementations for each device, we count the number of floating point (FP) instructions in the algorithm's critical region (innermost loop)– the larger number of single precision floating point operations in the SNB version is due to the accuracy correction approach applied with the addition of reciprocal divisions and square roots. Taking into account the potential overlapping of instructions on different units along with their cycle time, we calculate the expected number of cycles per iteration of that code region. From these numbers we can infer the expected vector instructions per cycle (IPC) and, given each platform's clock frequency, the maximum theoretical throughput in GFLOPS ignoring memory load costs. Subsequently, we calculate the achieved throughput and efficiency as the ratio of achieved to ideal performance for the particular algorithm and instruction mix, as shown in Figure 4c.

These results show a different side to the application performance than is portrayed either by performance, as in Figure 4a, or by the percentage of achieved, as in Figure 1b. Specifically, while K20 is the best performing overall, and despite the optimization effort expended on it, it remains at only 54.64% of theoretical peak performance. In principle that means that we should be able to get nearly *double* the performance we on that architecture. In practice our application is running with full occupancy and the most optimized instruction mix, shared memory behavior and instruction mix we have found. On the other hand, both the SNB and XP parts achieve greater than 80% of peak performance. This trend in performance efficiency has been noted before between CPUs and GPUs, but we find it telling that the XP achieves not just good efficiency,

but higher than SNB in this case.

## VI. Related Work

The need for faster ESP calculation execution, as part of molecular dynamics applications, has led to the development of multi- and many-core CPU and GPU implementations [6], [7], [8], as well as implementations on other heterogeneous computing platforms (IBM Cell [9]). Specialized hardware implementations using application-specific integrated circuits (ASIC), such as MD-GRAPE3 [10] have been deployed to provide molecular dynamics acceleration, as well. In contrast to these works, which optimize molecular dynamics on previous generations of parallel architectures, our paper is one of the first to address optimization of a molecular modeling application on the most recent NVIDIA GPU architecture (i.e., Kepler) and Intel Xeon Phi co-processor.

Various efforts in determining optimization techniques for multi-core CPUs and GPUs have been reported. A detailed study on optimization techniques for the CPU and GPU, along with an architectural comparison with respect to performance differences is presented in [11]. Satish et al. [12] research the *Ninja Gap*, i.e., the programming effort required to close the performance gap between mostly automated parallel implementations and hand-tuned code versions. The work in [11] does not consider Xeon Phi. Moreover, our work complements [12] in that it explores a wider range of optimizations (including the new *shuffle* feature of Kepler) and provides a more detailed analysis of the optimizations and their impact for the n-body class of problems. Finally, as part of directive-based parallel implementations, we contribute an analysis of compiler-hinted parallelization for GPUs using OpenACC and provide direct comparisons between the corresponding optimization levels across *all* three platforms, rather than focusing on the speed-ups and performance gap *within* a single platform.

## VII. Conclusion

In this paper, we presented a characterization of the performance and programmability of GEM across three multi- and many-core architectures: Intel Sandy Bridge multi-core CPU, Intel Xeon Phi (MIC) co-processor, and the NVIDIA Kepler K20 GPU. With respect to performance, we presented a systematic optimization approach via manual hand-tuning that achieved overall speed-ups of 194.98, 885.18 and 1020.88 over the unoptimized serial CPU version for the Sandy Bridge CPU, Xeon Phi MIC, and Kepler K20 GPU, respectively.

From a programmability perspective, the CPU programming model is arguably the most programmable, mainly due to its widespread adoption. Xeon Phi, which follows the same programming paradigm, benefits from allowing programmers to leverage their existing knowledge and expertise. On the other hand, GPU programming for high performance demands the use of native code, such as CUDA. This entails familiarizing oneself with a different mindset of parallel programming, a pertinent array of platform-specific optimization techniques, as well as taking care of mundane details, such as data transfers and kernel configuration, but while preserving more of the original computational code in its original state. Which is more programmable remains a matter of opinion, but the GPU and Xeon Phi devices require higher levels of optimization to reach their performance potential. New compilers and tools are needed to bridge this gap, and bring high-performance accelerated computing into the reach of the automatic optimizations.

## Acknowledgments

This work was supported in part by the Institute for Critical Technology and Applied Science (ICTAS) and by the National Science Foundation under Grant No. 0916719. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] "Software: Dr. Alexey Onufriev." [Online]. Available: http://people.cs.vt.edu/~onufriev/software.php

[2] N. A. Baker, "Poisson-Boltzmann Methods for Biomolecular Electrostatics," *Methods in Enzymology*, vol. 383, pp. 94–118, 2004. [Online]. Available: http://dasher.wustl.edu/chem478/reading/methenzymol-383-94-04.pdf

[3] J. C. Gordon, A. T. Fenley, and A. Onufriev, "An Analytical Approach to Computing Biomolecular Electrostatic Potential. II. Validation and Applications." *The Journal of Chemical Physics*, vol. 129, no. 7, Aug. 2008. [Online]. Available: http://dx.doi.org/10.1063/1.2956499

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A View of the Parallel Computing Landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1562764.1562783

[5] Intel, "AP-803: Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method," 1999.

[6] L. Kalé, R. Skeel, M. Bh, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater Scalability for Parallel Molecular Dynamics," *Journal of Computational Physics*, vol. 151, pp. 283–312, 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.94.5777

[7] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten, "GPU-Accelerated Molecular Modeling Coming of Age." *Journal of Molecular Graphics & Modelling*, vol. 29, no. 2, pp. 116–125, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jmgm.2010.06.010

[8] M. Daga and W.-c. Feng, "Multi-Dimensional Characterization of Electrostatic Surface Potential Computation on Graphics Processors," *BMC Bioinformatics*, vol. 13, pp. 1–12, 2012. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-13-S5-S4

[9] G. Shi and V. Kindratenko, "Implementation of NAMD Molecular Dynamics Non-Bonded Force-Field on the Cell Broadband Engine Processor," in *IEEE International Parallel and Distributed Processing Symposium*, April 2008, pp. 1–8.

[10] T. Narumi, K. Yasuoka, M. Taiji, F. Zerbetto, and S. Höfinger, "Fast Calculation of Electrostatic Potentials on the GPU or the ASIC MD-GRAPE-3," *The Computer Journal*, vol. 54, no. 7. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxq079

[11] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[12] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 440–451.