Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?

Nadathur Satish†, Changkyu Kim†, Jatin Chhugani†, Hideki Saito⋆, Rakesh Krishnaiyer⋆, Mikhail Smelyanskiy†, Milind Girkar⋆, and Pradeep Dubey†

†Parallel Computing Lab, Intel Corporation ⋆Intel Compiler Lab, Intel Corporation

ABSTRACT

Current processor trends of integrating more cores with wider SIMD units, along with a deeper and complex memory hierarchy, have made it increasingly more challenging to extract performance from applications. It is believed by some that traditional approaches to programming do not apply to these modern processors and hence radical new languages must be discovered. In this paper, we question this thinking and offer evidence in support of traditional programming methods and the performance-vs-programming effort effectiveness of common multi-core processors and upcoming many-core architectures in delivering significant speedup, and close-to-optimal performance for commonly used parallel computing workloads.

We first quantify the extent of the *"Ninja gap"*, which is the performance gap between naively written C/C++ code that is parallelism unaware (often serial) and best-optimized code on modern multi-/many-core processors. Using a set of representative throughput computing benchmarks, we show that **there is an average** *Ninja gap* **of 24X (up to 53X)** for a recent 6-core Intel® Core™ i7 X980 Westmere CPU, and that this gap if left unaddressed will inevitably increase. We show how a set of well-known algorithmic changes coupled with advancements in modern compiler technology can bring down the Ninja gap to an average of **just 1.3X**. These changes typically require low programming effort, as compared to the very high effort in producing Ninja code. We also discuss hardware support for programmability that can reduce the impact of these changes and even further increase programmer productivity. We show equally encouraging results for the upcoming Intel® Many Integrated Core architecture (Intel® MIC) which has more cores and wider SIMD. We thus demonstrate that we can contain the otherwise uncontrolled growth of the Ninja gap and offer **a more stable and predictable performance growth** over future architectures, offering strong evidence that radical language changes are not required.

1. INTRODUCTION

Performance scaling across processor generations has previously relied on increasing clock frequency. Programmers could ride this trend and did not have to make significant code changes for improved code performance. However, clock frequency scaling has hit the power wall [32], and the free lunch for programmers is over.

Recent techniques for increasing processor performance have a focus on integrating more cores with wider SIMD units, while simultaneously making the memory hierarchy deeper and more complex. While the peak compute and memory bandwidth on recent processors has been increasing, it has become more challenging to extract performance out of these platforms. This has led to the situation where only a small number of expert programmers ("Ninja programmers") are capable of harnessing the full power of modern multi-/many-core processors, while the average programmer only obtains a small fraction of this performance. We define the term "Ninja gap" as the performance gap between naively written parallelism unaware (often serial) code and best-optimized code on modern multi-/many-core processors.

There have been many recent publications [45, 43, 14, 2, 28, 33] that show 10-100X performance improvements for real-world applications through adopting highly optimized platform-specific parallel implementations, proving that a large Ninja gap exists. This typically requires high programming effort and may have to be re-optimized for each processor generation. However, these papers do not comment on the effort involved in these optimizations. In this paper, we aim at quantifying the extent of the Ninja gap, analyzing the causes of the gap and investigating how much of the gap can be bridged with low effort using traditional C/C++ programming languages.[1]

We first quantify the extent of the Ninja gap. We use a set of real-world applications that require high throughput (and inherently have a large amount of parallelism to exploit). We choose throughput applications because they form an increasingly important class of applications [13] and because they offer the most opportunity for exploiting architectural resources - leading to large Ninja gaps if naive code does not take advantage of these resources. We measure performance of our benchmarks on a variety of platforms across different generations: 2-core Conroe, 4-core Nehalem, 6-core Westmere, Intel® Many Integrated Core (Intel® MIC), and the NVIDIA C2050 GPU. Figure 1 shows the Ninja gap for our benchmarks on three CPU platforms: a 2.4 GHz 2-core E6600 Conroe, a 3.33 GHz 4-core Core i7 975 Nehalem and a 3.33 GHz 6-core Core i7 X980 Westmere. The figure shows that there is up to a 53X gap between naive C/C++ code and best-optimized code for a recent 6-core Westmere CPU. The figure also shows that this gap has been increasing across processor generations - the gap is 5-20X on a 2-core Conroe system (average of 7X) to 20-53X on Westmere (average of 25X). This gap has been increasing in spite of micro-architectural improvements that have reduced the need and impact of performing various optimizations.

We next analyze the sources of the large performance gap. There are a number of reasons why naive code performs badly. First, the code may not be parallelized, and compilers do not automatically identify parallel regions. This means that the increasing core count is not utilized in the naive code, while the optimized code takes full advantage of it. Second, the code may not be vectorized, leading to under-utilization of the increasing SIMD widths. While auto-

[1]Since measures of ease of programming such as programming time or lines of code are largely subjective, we show code snippets with the code changes required to achieve performance.

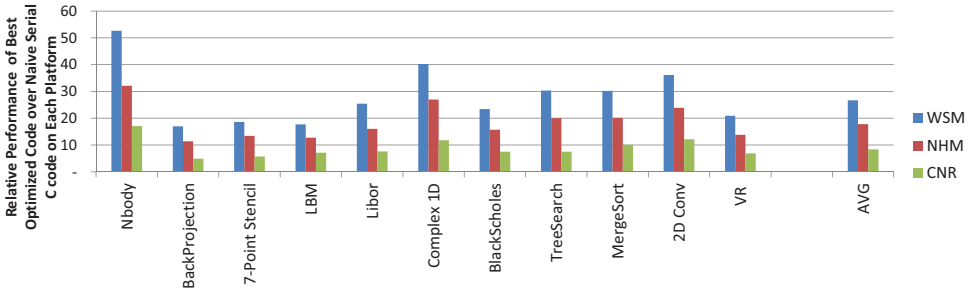


Figure 1: **Growing performance gap between Naive serial C/C++ code and best-optimized code on a 2-core Conroe (CNR), 4-core Nehalem (NHM) and 6-core Westmere (WSM) systems.**

vectorization has been studied for a long time, there are many difficult issues such as dependency analysis, memory alias analysis and control flow analysis which prevent the compiler from vectorizing outer loops, loops with gathers (irregular memory accesses) and even innermost loops where dependency and alias analysis fails. A third reason for large performance gaps may be that the code is bound by memory bandwidth - this may occur, for instance, if the code is not blocked for cache hierarchies - resulting in cache misses.

Recent compiler technologies have made significant progress in enabling parallelization and vectorization with relatively low programmer effort. Parallelization can be achieved using OpenMP pragmas that only involve annotation of the loop that is to be parallelized. For vectorization, recent compilers such as the Intel® Composer XE 2011 version have introduced the use of a pragma for the programmer to force loop vectorization by circumventing the need to do dependency and alias analysis. This version of the compiler also has the ability to vectorize outer level loops, and the Intel® Cilk™ Plus feature [22] helps the programmer to use this new functionality when it is not triggered automatically.[2]. Using these features, we show that the Ninja gap **reduces to an average of 2.95X for Westmere**. The remaining gap is either a result of bandwidth bottlenecks in the code or the fact that the code gets only partially vectorized due to irregular memory accesses. While the improvement in the gap is significant, the gap will however inevitably increase on future architectures with growing SIMD widths and decreasing bandwidth-to-compute ratios. To overcome this gap, programmer intervention in the form of algorithmic changes is then required.

We identify and suggest three critical algorithmic changes: **blocking** for caches, bandwidth/SIMD friendly **data layouts** and in some cases, choosing an **alternative SIMD-friendly algorithm**. An important class of algorithmic changes involves blocking the data structures to fit in the cache, thus reducing the memory bandwidth pressure. Another class of changes involves eliminating the use of memory gather/scatter operations. Such irregular memory operations can both increase latency and bandwidth usage, as well as limit the scope of compiler vectorization. A common data layout change is to convert data structures written in an Array of Structures (AOS) representation to a Structure of Arrays (SOA) representation. This helps prevent gathers when accessing one field of the structure across the array elements, and helps the compiler vectorize loops that iterate over the array. Finally, in some cases, the code cannot be vectorized due to back-to-back dependencies between loop iterations, and in those cases a different SIMD-friendly algorithm may need to be chosen. We also discuss **hardware support for programmability**, that can further improve productivity by reducing the impact of these algorithmic changes.

We show that **after performing algorithmic changes**, we have an **average performance gap of only 1.3X** between best-optimized and compiler-generated code. Although this requires some programmer effort, this effort is amortized across different processor generations and also across different computing platforms such as GPUs. Since the underlying hardware trends towards increasing cores, SIMD width and slowly increasing bandwidth have been optimized for, a **small and predictable** performance gap will remain across future architectures. We demonstrate this by repeating our experiments for the new **Intel® MIC architecture** [41], the first x86 based manycore platform. We show that the **Ninja gap is almost the same (1.2X)**. In fact, the addition of hardware gather support makes programmability easier for at least one benchmark. We believe this is the first paper to show programmability results for MIC. Thus the combination of algorithmic changes coupled with modern compiler technology is an important step towards enabling programmers to ride the trend of parallel processing using traditional programming.

2. BENCHMARK DESCRIPTION

For our study, we analyze compute and memory characteristics of recently proposed benchmark suites [3, 11, 4], and choose a representative set of benchmarks from the suite of **throughput computing applications**. Throughput workloads deal with processing large amounts of data in a given amount of time, and require a fast response time for all the data processed as opposed to the response time for a single data element. These include workloads from the areas of High Performance Computing, Financial Services, EDA, Image Processing, Computational Medicine, Databases, etc [11]. Throughput computing applications have plenty of data- and thread-level parallelism, and have been identified as one of the most important classes of future applications [3, 4, 11], with *compute* and *memory characteristics* influencing the design of current and upcoming multi-/many-core processors [16]. Furthermore, they offer the most opportunity for exploiting architectural resources – leading to large Ninja gaps if naive code does not take advantage of the increasing computational resources. We formulated a representative set of benchmarks described below that **cover this wide range of application domains** of throughput computing.

1. NBody: NBody computations are used in many scientific applications, including the fields of astrophysics [1] and statistical learning algorithms [20]. For given $\mathcal{N}$ bodies, the basic computation is an $O(\mathcal{N}^2)$ algorithm that has two loops over the bodies, and computes pair-wise interactions between them. The resulting

[2]For more complete information about compiler optimizations, see the optimization notice at [24]

forces for each body are added up and stored into an output array.

2. BackProjection: Backprojection is a commonly used kernel in performing cone-beam image reconstruction of CT projection values [26]. The input consists of a set of 2D images that are "back-projected" onto a 3D volume in order to construct the 3D grid of density values. As far as the computation is concerned, for each input image (and the corresponding projection direction), each 3D grid point is projected onto the 2D image, and the density from the neighboring 2X2 pixels is bilinearly interpolated and accumulated to the voxel's density.

3. 7-Point Stencil: Stencil computation is used for a wide range of scientific disciplines [14]. The computation involves multiple sweeps over a spatial input 3D grid of points, where each sweep computes the weighted sum of each grid point and its +/-X, +/-Y and +/-Z neighbors (total of 7 grid points), and stores the computed value to the corresponding grid point in the output grid.

4. Lattice Boltzmann Method (LBM): LBM is a class of computational fluid dynamics capable of modeling complex flow problems [44]. It simulates the evolution of particle distribution functions over a 3D lattice over many time-steps. For each time-step, at each grid point, the computation performed involves directional density values for the grid point and its face (6) and edge (12) neighbors (also referred to as D3Q19).

5. LIBOR Monte Carlo: The LIBOR market model is used to price a portfolio of swaptions [8]. It models a set of forward rates as a log-normal distribution. A typical Monte Carlo approach would generate many random samples for this distribution and compute the derivative price using a large number of paths, where computation of paths are independent from each other.

6. Complex 1D Convolution: This is widely used in application areas like image processing, radar tracking, etc. This application performs a 1D convolution on complex 1D images with a large complex filter.

7. BlackScholes: The Black-Scholes model provides a partial differential equation (PDE) for the evolution of an option price. For European options, where the option can only be exercised on maturity, there is a closed form expression for the solution of the PDE [5]. This involves a number of math operations such as the computation of a Cumulative Normal Distribution Function (CNDF) exponentiation, logarithm, square-root and division operations.

8. TreeSearch: In-memory tree structured index search is a commonly used operation in commercial databases, like Oracle TimesTen [37]. This application involves multiple parallel searches over a tree with different queries, with each query tracing a path through the tree depending on the results of comparison of the query to the node value at each tree level.

9. MergeSort: MergeSort is commonly used in the area of databases [12], HPC, etc. MergeSort sorts an array of $\mathcal{N}$ elements using $\log\mathcal{N}$ *merge passes* over the complete array, where each pass *merges* sorted lists of size twice as large as the previous pass (starting with sorted lists of size one for the first pass). MergeSort [40] is shown to be the sorting algorithm of choice for future architectures.

10. 2D 5X5 Convolution: Convolution is a common image filtering operation used for effects such as blur, emboss and sharpen, and is also used with high-resolution images in EDA applications. For a given 2D image and a 5X5 spatial filter, each pixel computes and stores the weighted sum of a 5X5 neighborhood of pixels, where the weights are the corresponding values in the filter.

11. Volume Rendering: Volume Rendering is a commonly used benchmark in the field of medical imaging [15], graphics visualization, etc. Given a 3D grid, and a 2D image location, the benchmark spawns rays (perpendicular to the image plane (orthographic projection)) through the 3D grid, which accumulates the density, color and opacity to compute the final color of each pixel of the image. For our benchmark, we spawn rays perpendicular to the X direction of the grid.

Benchmark	Dataset	Best Optimized Performance
NBody [2]	10^6 bodies	7.5 X 10^9 Pairs/sec
BackProjection [26]	500 images on $1K^3$	1.9 X 10^9 Proj./sec
7 Point 3D Stencil [33]	512^3 grid	4.9 X 10^9 Up./sec
LBM [33]	256^3 grid	2.3 X 10^8 Up./sec
LIBOR [19]	10M paths on 15 options	8.2 X 10^5 Paths/sec
Complex 1D Conv. [25]	8K on 1.28M pixels	1.9 X 10^6 Pixels/sec
BlackScholes [38]	1M call+put options	8.1 X 10^8 Options/sec
TreeSearch [28]	100M queries on 64M tree	7.1 X 10^7 Queries/sec
MergeSort [12]	256 M elements	2.1 X 10^8 Data/sec
2D 5X5 Convolution [30]	2K X 2K Image	2.2 X 10^9 Pixels/sec
Volume Rendering [43]	512^3 volume	2.0 X 10^8 Rays/sec

Table 1: **Various benchmarks and the respective datasets used, along with the best optimized (*Ninja*) performance on Core i7 X980.**

2.1 Ninja Performance

Table 1 provides details of the representative dataset sizes for each of the benchmarks used in this paper. For each benchmark, there exists a corresponding best performing code for which the performance numbers have been previously cited[3] on different platforms than those used in our study. Hence, in order to perform a fair comparison, we **implemented and aggressively optimized (including the use of intrinsics/assembly code) each of the benchmarks by hand**, and obtained *comparable performance* to the best reported numbers on the corresponding platform. This code was then executed on our platforms to obtain the corresponding Best Optimized Performance for platforms we use in this paper. Table 1 (column 3) show the Best Optimized (*Ninja*) performance for all the benchmarks on Intel® Core™ i7 X980. For the rest of the paper, **Ninja Performance refers to the performance numbers** obtained by executing this code on our platforms.

3. BRIDGING THE NINJA GAP

In this section, we take each of the benchmarks described in Section 2, and attempt to bridge the Ninja gap starting with naively written code with low programming effort.

Platform: We measured the performance on a 3.3GHz 6-core Intel® Core™ i7 X980 (architecture code-named Westmere). The peak compute power is 158 GFlops and the peak bandwidth is 30 GBps. The Core i7 processor cores feature an out-of-order super-scalar micro-architecture, with 2-way Simultaneous Multi-Threading (SMT). In addition to scalar units, it also has 4-wide SIMD units that support a wide range of SIMD instructions. Each core has an individual 32KB L1 cache and a 256KB L2 cache. All six cores share a 12MB last-level cache (LLC). Our system has 12 GB RAM and runs SuSE Enterprise Linux version 11. We use the Intel® Composer XE 2011 compiler for Linux.

Methodology: For each benchmark, we attempt to first get good single thread performance through exploiting instruction and data

[3]The best reported numbers are cited from the most recent top-tier publications in the area of Databases, HPC, Image processing, etc. and include conference/journals like Super Computing, VLDB, SIGMOD, MICCAI, IEEE Vis. To the best of our knowledge, there *does not exist* any faster performing code for any of the benchmarks.

level parallelism. In an attempt to fully exploit the available data level parallelism, we measure the SIMD scaling we obtain for each benchmark by running the code with auto-vectorization enabled and disabled (using the -no-vec flag) in the compiler. If SIMD scaling is not close to peak (we expect close 4X scaling with single precision data on SSE), we analyze the generated code to identify architectural bottlenecks. We then obtain thread level parallelism by adding OpenMP pragmas to parallelize the benchmark and evaluate thread scaling - again evaluating bottlenecks to scaling. After evaluating bottlenecks to core and SIMD scaling, we make any necessary algorithmic changes to overcome these bottlenecks.

Compiler pragmas used: We use OpenMP for thread-level parallelism, and use the auto-vectorizer or recent technologies such as array notations introduced as part of the Intel® Cilk™Plus (hereafter referred to array notations) for data parallelism. The compiler directives we add to the code and command line are the following:

- ILP optimizations: We use the **#pragma unroll** directive just before an innermost loop that needs to be unrolled, and an **#pragma unroll_and_jam** primitive outside an outer loop that needs to be unroll-jammed. Both accept an optional parameter which is the number of times the loop is to be unrolled.

- Vectorizing at innermost loop level: If auto-vectorization fails due to assumed memory alias or dependence analysis, the programmer can force vectorization using **#pragma simd**. This is a recent feature introduced in the Intel® Cilk™Plus. The use of the pragma is an indication that the programmer asserts that the loop is safe to vectorize. The pragma has clauses to describe the data environment: private, firstprivate, lastprivate, and reduction clauses adopted from OpenMP, and a new linear clause. For details, please refer to [22].

- Vectorizing at outer loop levels: This can be done in two different ways: 1) directly vectorize at outer loop levels using auto-vectorization or using the simd pragma, and 2) Strip-mine outer loop iterations and change each statement in the loop body to operate on the strip. Intel® CilkPlus array notation extension helps the programmers to express the second approach in a natural fashion as illustrated in Figure 7(b). In this study, we used the second approach with array notations.

- Parallelization: We use the OpenMP **#pragma omp** to parallelize loops. We typically use this over an outer for loop using a **#pragma omp parallel for** construct.

- Fast math: We use the **-fimf-precision** flag selectively to our benchmarks depending on precision needs.

3.1 NBODY

We implement a NBody algorithm [1], performing computation over 1 million bodies. The computation consists of about 20 floating point operations (flops) per body-body interaction, spent in computing the distance between each pair of bodies, and in computing the corresponding local potentials using a reverse square-root of the distance. The dataset itself requires 16 bytes for each body, for a total of 16 MB - this is larger than the available cache size.

Figure 2 shows the breakdown of the various optimizations. Note that the figure uses a log scale on the y-axis since the impact of various optimizations is multiplicative. The code consists of two loops that iterate over all bodies and computes potentials for each pair. We first performed unrolling optimizations to improve ILP, both over the inner and outer loops using the relevant pragmas.

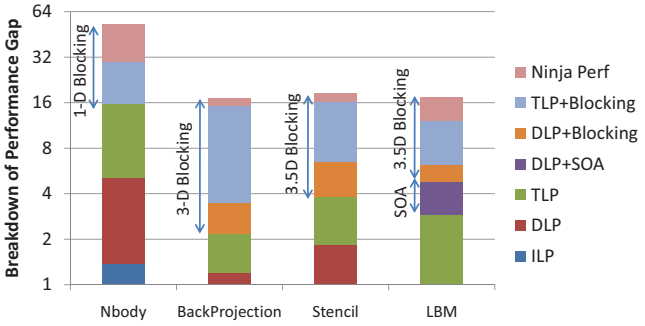


Figure 2: **Breakdown of Ninja Performance Gap in terms of Instruction (ILP), Task (TLP) and Data Level Parallelism (DLP) before and after algorithm changes for NBody, BackProjection, 7-point Stencil and LBM. The algorithm change involves blocking.**

Unrolling gives us a benefit of about 1.4X. The compiler **auto-vectorizes** the code well with no programmer intervention and provides a good scaling of 3.7X with vector width of 4. We only obtained a parallel scaling of 3.1X, far lower than the peak of 6X on our processor. The reason for this is that the benchmark is **bandwidth-bound**. The code loads 16 bytes of data to perform 20 flops of computation, and thus requires 0.8 bytes/flop of memory bandwidth – while our system only delivers 0.2 bytes/flop. This means that the benchmark cannot utilize all computation resources.

This motivates the need for our algorithmic optimization of blocking the data structures to fit in the last level (L3) cache (referred to as **1-D blocking** in Figure 2). The inner body loop is split into two - one iterating over blocks of bodies (fitting in cache), and the other on bodies within each block. This allows the inner loop to reuse data in cache. A code snippet for NBody blocking code is shown in Section 4.1.

Once blocking is done, the computation is now **bound by compute resources**. We obtain an additional 1.9X thread scaling for a total of 5.9X - close to our peak. We find only a 1.1X performance gap between compiled and best-optimized code.

3.2 BackProjection

We back-project 500 images of dimension 2048x2048 pixels onto a 1024x1024x1024 uniform 3D grid. Backprojection requires about 80 ops to project each 3D grid point to an image point and perform bilinear interpolation around it. This requires about 128 bytes of data to load and store the image and volume. Both the image (of size 16 MB) and volume (4 GB) are too large to reside in cache.

Figure 2 shows that we get poor SIMD scaling of 1.2X from auto-vectorization. Moreover, parallel scaling is also only around 1.8X. This is because the code is **bandwidth-bound**, requiring 1.6 bytes/flop of bandwidth. Most of this bandwidth comes because of gathers from external memory in the code - the code projects multiple contiguous 3D points in SIMD lanes, but the projected points in the 2D image are not contiguous. Reading the 2x2 surrounding pixels thus requires gather operations.

We perform blocking over the 3D volume to reduce bandwidth (called **3D blocking** in Figure 2). Due to spatial locality, the image working set also reduces accordingly. This results in the code becoming **compute bound**. However, due to the gathers which cannot be vectorized on the CPU, SIMD scaling only improved by an additional 1.6X (total 1.8X). We obtained additional 4.4X thread scaling (total 7.9X), showing the benefits of SMT. The resulting performance is only 1.1X off the best-optimized code.

3.3 7-Point 3D Stencil

7-Point Stencil iterates over a 3D grid of points, and for each point (4 bytes), performs around 8 flops of computation. For grid sizes larger than the size of the cache, the resultant b/w requirement is around 0.5 bytes/flop, which is much larger than that available on the current architectures. The following performance analysis is done for a 3D dataset of dimension 512x512x512 grid points.

Figure 2 shows that we get a poor SIMD scaling of around 1.8X from auto-vectorization. This is due to the fact that the implementation is **bandwidth bound**, and is not able to exploit the available vector processing flops. The bandwidth bound nature of the application is further exemplified by the low thread-level scaling of around 2.1X on 6-cores. In order to improve the scaling and exploit the increasing computational resources, we perform both **spatial** and **temporal blocking** to improve the performance.

In order to perform *spatial blocking*, we block in the XY dimension, and iterate over the complete range of Z values (referred to as 2.5D blocking [33]). We compute the blocking dimensions in X and Y directions such that *three* of the blocked XY planes are expected to fit in the LLC. Since the original 3D stencil performs the stencil computation for multiple time-steps, we can further perform temporal blocking to perform *multiple time-steps* (3.5D blocking [33]), and further increase the computational efficiency.

The resultant code performs *four* time-steps simultaneously, and improves the DLP by a further 1.7X to achieve a net SIMD scaling of around 3.1X. It is important to note that although the code vectorizes well, the SIMD scaling is lower than 4X due to the overhead of repeated computation at the boundary elements of each blocked XY sub-plane, which increases the net computation as compared to an unblocked stencil computation. This results in a slightly reduced SIMD scaling. Note that this reduction is expected to *be stable* with increasing SIMD widths, and is thus a one-time reduction in performance. The thread-level scaling is further boosted by around 2.5X, to achieve a net core-scaling of around 5.3X. Our net performance is within 10.3% of the best-optimized code.

3.4 Lattice Boltzmann Method (LBM)

The computational pattern of LBM is similar to the stencil kernel described in the previous section. In case of LBM [45], the computation for each grid cell is performed using a combination of its face and edge neighbors (19 in total including the cell), and each grid cell stores around 80 bytes of data (for the 19 cells and some auxiliary data). The data is usually stored in an AOS format, with each cell storing the 80 bytes contiguously. For grid sizes larger than the size of the cache, the resultant bandwidth requirement is around 0.7 bytes/flop, which is much larger than that available on the current architectures. The following performance analysis is done for a 3D dataset of dimension 256x256x256 grid points.

Figure 2 shows that our initial code (taken from SPEC CPU2006) does not achieve any SIMD scaling, and around 2.9X core-scaling. The reason for no SIMD scaling is that the AOS data layout results in gather operations during the grid computations for 4 simultaneous cells. In order to improve the performance, we perform the following **two algorithmic changes**. Firstly, we perform an AOS to SOA conversion of the data. The resultant auto-vectorized code improves SIMD scaling to 1.65X. Secondly, we perform a 3.5D blocking (similar to Section 3.3). The resultant auto-vectorized code further boosts SIMD scaling by 1.3X, achieving a net scaling of around 2.2X. The resultant thread-level scaling was further increased by 1.95X (total 5.7X). The overall SIMD scaling is lower than the scaling for 7 point stencil, and we found that the compiler generated extra spill and fill instructions that were reduced by the best-performing code to leave a final performance gap of 1.4X.

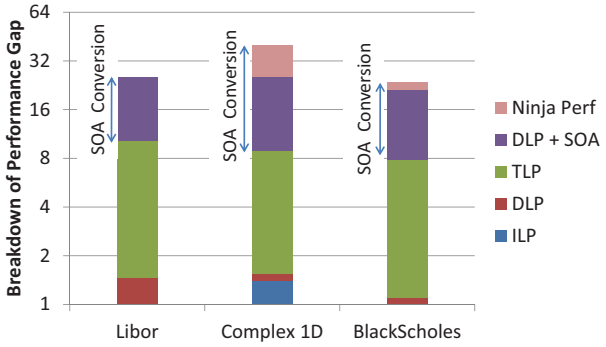


Figure 3: **Breakdown of Ninja Performance Gap Libor, Complex 1D convolution and BlackScholes. All three benchmarks require AOS to SOA conversion to obtain good SIMD scaling.**

3.5 LIBOR

LIBOR code [8] has an outer loop over all the paths of the Monte Carlo simulation, and an inner loop over the forward rates on a single path. A typical simulation runs over several tens to hundreds of thousands of independent paths. LIBOR has very low bandwidth-to-compute requirements (0.02 bytes/flop) and is compute bound.

Figure 3 shows only a 1.5X performance benefit from compiler auto-vectorization for LIBOR. The cause of this low SIMD scaling is that the current **compiler by default only attempts to vectorize the inner loop** - this loop has back-to-back dependencies and can only be partially vectorized. In contrast, the outer loop has completely independent iterations (contingent on parallel random generation) and is a good candidate for vectorization. However, outer loop vectorization is inhibited by data structures stored in a AOS format (in particular, the results of path computations). This requires gathers and scatters resulting in poor SIMD scaling. The code achieves a good parallel scaling of 7.1X; this number being greater than 6 indicates that the use of SMT threads provided additional benefits over just core scaling.

To solve the vectorization issue, we performed an algorithmic change to **convert the memory layout from AOS to SOA**. We use the array notations technology available in the compiler to express outer loop vectorization. The LIBOR array notations example is straightforward to code and is shown in Figure 7(b). Performing the algorithmic change and using array notations allowed the outer loop to vectorize and provides additional 2.5X SIMD scaling, a total of about 3.8X scaling. We found that this performance is similar to the best-optimized code.

3.6 Complex 1D Convolution

We perform a 1D complex convolution on an image with 12.8 million points, and a kernel size of 8K complex floating point numbers. The code consists of two loops: one outer loop iterating over the pixels, and one inner loop iterating over the kernel values. The data is stored in an AOS format, with each pixel storing the real and imaginary values together.

Figure 3 shows the performance achieved (the first bar) by the unrolling enabled by the compiler, which results in around 1.4X scaling. The auto-vectorizer only achieves a scaling of around 1.1X since the the compiler vectorizes by computing the convolution for four consecutive pixels, and this involves gather operations owing to the AOS storage of the input data. The TLP achieved is around 5.8X. In order to improve the performance, we perform a **rearrangement of data from AOS to SOA format**, and store the real values for all pixels together, followed by the imaginary values for all the pixels. A similar scheme is adopted for the kernel. As a

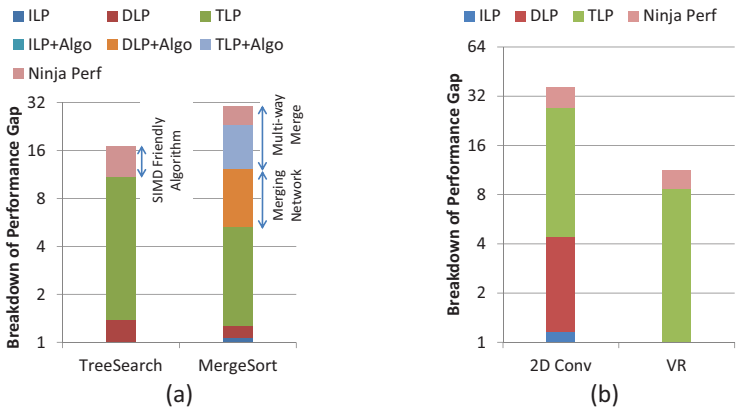


Figure 4: **Breakdown of Ninja Performance Gap for (a) Treesearch and Mergesort, and (b) for 2D convolution and VR . The benchmarks in (a) require rethinking algorithms to be more SIMD-friendly. The benchmarks in (b) do not require any algorithmic changes.**

result, the compiler produces efficient SSE code, and the resultant code scales up by a further 2.9X. Our overall performance is about 1.6X slower than the best-optimized numbers. This is because the best-optimized code is able to block some of the kernel weights in SSE registers and avoids reloading them, while the compiler does not perform this optimization.

3.7 BlackScholes

BlackScholes computes the call and put options together. Each option is priced using a sequence of operations involving computing the inverse CNDF, followed by math operations involving exp, log, sqrt and division operations. The total computation performed is around 200 ops (including the math ops), while the bandwidth is around 36 bytes. The data for each option is stored contiguously.

Figure 3 shows a SIMD speedup of around 1.1X using auto-vectorization. The low scaling is primarily due to the AOS layout, which results in gather operations (performed using scalar ops on CPUs). The TLP scaling is around 7.2X, which includes around 1.2X SMT scaling, and near linear core-scaling. In order to exploit the vector compute flops, we performed an **algorithmic change**, and *changed the data layout* from AOS to SOA. After this change, the auto-vectorizer generated SVML (short vector math library) code, resulting in an increase of SIMD scaling of 2.7X (total 3.0X). The resultant code is within 1.1X of the best performing code.

3.8 TreeSearch

The input binary tree is usually laid out in a breadth-first fashion, and the input queries are compared against the a tree node at each level, and traverse down the left or right child depending on the result of the comparison.

Figure 4(a) shows that the auto-vectorizer achieves a SIMD speedup of around 1.4X. This is because the vectorizer operates on 4 queries simultaneously, and since they all traverse down different paths, gather instructions are required, devolving to scalar load operations. The thread-level scaling of around 7.8X (including SMT) is achieved.

In order to improve the SIMD performance, we perform an **algorithmic change**, and traversed 2 levels at a time (similar to SIMD width blocking proposed in [28]). However, the compiler did not generate the code sequence described in [28], resulting in a 1.55X gap from Ninja code. We note that this algorithmic change is only required because gather operations in SSE devolve to scalar loads. For future architectures such as the Intel MIC architecture that have hardware support for SIMD gathers, this change can become unnecessary. This is an example where hardware support makes programming easier. We show more details in Section 4.2.4.

3.9 MergeSort

MergeSort sorts an input array of $\mathcal{N}$ elements using $\log\mathcal{N}$ *merge phases* over the complete array, where each phase merges sorted lists of size twice as large as the previous pass (starting with sorted lists of size one for the first pass). *Merging* two lists involves *comparing* the heads of the two lists, and appending the smaller element to the output list. The performance for small lists is largely dictated by the branch misprediction handling of the underlying architecture. Furthermore, since each *merge phase* completely reads in the two lists to produce the output list, it becomes bandwidth bound once the list sizes grow larger than the cache size, with the bandwidth requirement per element being around 12 bytes. Our analysis is done for sorting an input array with 256M elements.

Figure 4(a) shows that we only get a 1.2X scaling from auto-vectorization. This is largely due to gather operations for merging *four* pairs of lists. Parallel scaling is also only around 4.1X because the last few merge phases being bandwidth bound, and not scaling linearly with number of cores. In order to improve performance, we perform the following *two* **algorithmic changes**.

Firstly, in order to improve the DLP scaling, we implement merging of lists using a merging network [12], that merges two sorted sub-lists of size $\mathcal{S}$ (SIMD width) into a sorted sub-list of size $2\mathcal{S}$ using a series of min/max and interleave operations (code snippet is shown in Section 4.1). Each merging phase is decomposed into a series of such sub-list merge operations. This code sequence is vectorized by the compiler to produce an efficient SSE code. Furthermore, the number of comparisons is also reduced by around 4X, and the resultant vector code speeds up by around 2.3X. Secondly, in order to reduce the bandwidth requirements, we perform multiple merge phases together. Essentially, instead of merging two lists, we combine *three* merge phases, and merge eight lists into a single sorted list. This reduces the bandwidth requirement, and makes the *merge phases* compute bound. The parallel scaling of the resultant code further speeds up by 1.9X. The resultant performance is within 1.3X of the best-optimized code.

3.10 2D Convolution

We perform convolution of a 2K X 2K image with a 5 X 5 kernel. Both the image and kernel consists of 4-byte floating point values. The convolution code consists of four loops. The two outer loops iterate over the input pixels (X and Y directions), while the two

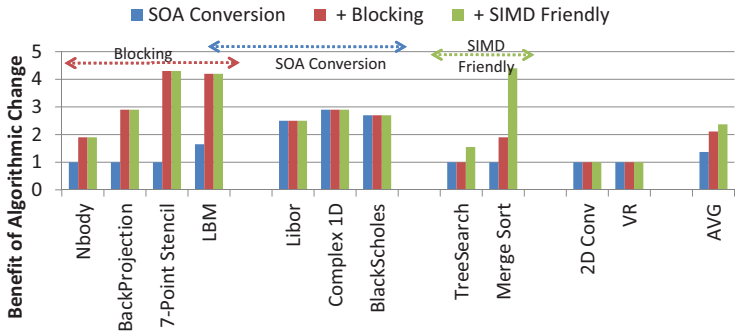


Figure 5: **Benefit of three different algorithmic changes to our benchmarks normalized to code before any algorithmic change. The effect of algorithmic changes is cumulative.**

inner loops iterate over the kernel (X and Y directions).

Figure 4(b) shows that we obtained a benefit of 1.2X through loop unrolling. The most efficient way to exploit SIMD is to perform stencil computations on 4 consecutive pixels, with each performing a load operation and a multiply-add with the appropriate kernel value. This implies performing a vectorization for the outer X loop, something that the current compiler does not perform. We instead implemented the two inner loops using the array notations technology available in the compiler. That enabled vectorization of the outer X loop, and produced SIMD code that scaled 3.8X with SIMD width. The thread-level parallelism was around 6.2X. Our net performance was within 1.3X of the best-optimized code.

3.11 Volume Rendering

The VR rendering code iterates over various rays, and traverses a volume for each ray. During this traversal, the density and color are accumulated for each ray till a pre-defined threshold value of the opacity is reached, or the ray intersects all the voxels in its path. These early exit conditions make the code control intensive.

As shown in Figure 4(b), we achieve a TLP scaling of around 8.7X, which includes a SMT scaling of 1.5X, and a near-linear core-scaling of 5.8X. As for SIMD scaling, earlier compiler versions did not vectorize the code due to various control-intensive statements. However, recent compilers do, in fact, vectorize the code using mask values for each branch instruction, and using proper masks to execute both execution paths for each branch. Since CPUs do not have masks, this is emulated using 128-bit SSE registers. The Ninja code also performs similar optimizations. There is only a small difference of 1.3X between Ninja code and compiled code.

3.12 Summary

In this section, we looked at each benchmark, and were able to narrow the Ninja gap to within 1.1 - 1.6X by applying necessary algorithmic changes coupled with the latest compiler technology.

4. ANALYSIS AND SUMMARY

In this section, we generalize our findings in the previous section and identify the steps to be taken to bridge the Ninja performance gap with low programmer effort. The key steps to be taken are to first perform a set of well-known and simple algorithmic optimizations to overcome scaling bottlenecks either in the architecture or in the compiler, and secondly to use the latest compiler technology with regards to vectorization and parallelization. We will now summarize our findings with respect to the gains we achieve in each of these steps. We also show using representative code snippets that the changes required in exploiting latest compiler features are small and that they can be done with low programming effort.

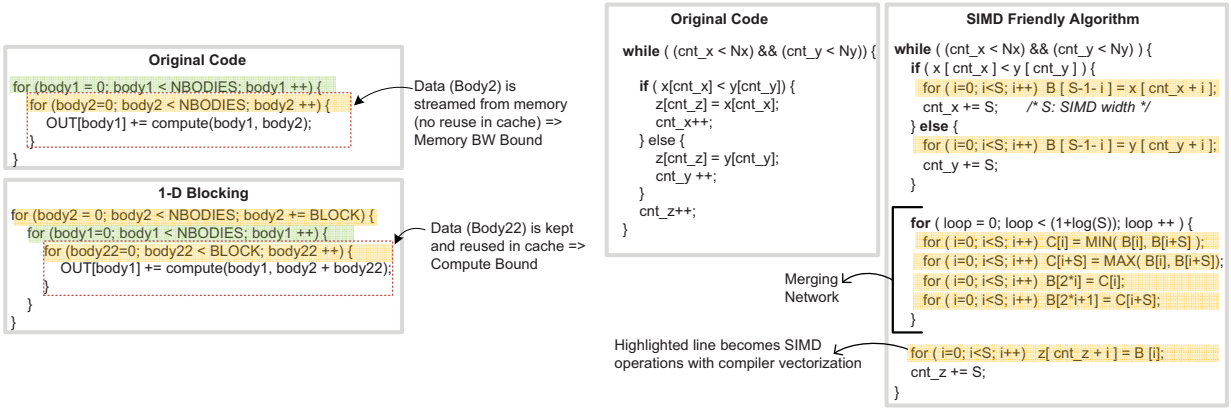
4.1 Algorithmic Changes

We first describe a set of well-known algorithmic techniques that are necessary to avoid vectorization issues and memory bandwidth bottlenecks in compiler generated code. Incorrect algorithmic choices and data layouts in naive code can lead to Ninja gaps that will only grow larger with recent hardware trends of increasing SIMD width and decreasing bandwidth-to-compute ratios. It is thus critical to perform optimizations like blocking data structures to fit in cache hierarchies, layout data structures to avoid gathers and scatters, or rethink the algorithm to allow data parallel computation. While such changes do require some programmer effort, they can be used across multiple platforms (including GPUs) and multiple generations of each platform, and are a critical component of keeping the Ninja gap small and stable over future architectures. Figure 5 shows the performance improvements due to various algorithmic optimizations. We describe these optimizations below.

AOS to SOA conversion: A common optimization that helps prevent gathers and scatters in vectorized code is to convert data structures from Array-Of-Structures (AOS) to Structure-Of-Array (SOA) representation. Keeping separate arrays for each structure keeps memory accesses contiguous when vectorization is performed over structure instances. AOS structures require gathers and scatters, which can impact both SIMD efficiency as well as introduce extra bandwidth and latency for memory accesses. The presence of a hardware gather/scatter mechanism does not eliminate the need for this transformation - gather/scatter accesses commonly need significantly higher bandwidth and latency than contiguous loads. Such transformations are also advocated for a variety of architectures including GPUs [36]. Figure 5 shows that for our benchmarks, AOS to SOA conversion helped by an average of 1.4X.

Blocking: Blocking is a well-known optimization that can help avoid memory bandwidth bottlenecks in a number of applications. The key idea behind blocking is to exploit the inherent data reuse available in the application by ensuring that data remains in caches across multiple uses. Blocking can be performed on 1-D, 2-D or 3-D spatial data structures, and some iterative applications can further benefit from blocking over multiple iterations (commonly called temporal blocking) to further mitigate bandwidth bottlenecks.

In terms of code change, blocking typically involves a combination of loop splitting and interchange. For instance, the code snippet in Figure 6(a) shows an example of blocking NBody code. There are two loops (body1 and body2) iterating over all bodies. The original code on the top streams through the entire set of bodies in the inner loop, and must load the body2 value from memory in each iteration. The blocked code at the bottom is obtained by



(a) Example of the use of Blocking

(b) Example of the use of SIMD Friendly Algorithm

Figure 6: **Code snippets showing algorithmic changes for (a) blocking in NBody and (b) SIMD-friendly MergeSort algorithm.**

splitting the body2 loop into an outer loop iterating over bodies in multiple of $BLOCK$, and an inner body22 loop iterating over elements within the block, and interleaving the body1 and body2 loops. This code reuses a set of $BLOCK$ body2 values across multiple iterations of the body1 loop. If $BLOCK$ is chosen such that this set of values fits in cache, memory traffic is brought down by a factor of $BLOCK$. Such changes extend to further dimensional data structures (3D in BackProjection, plus temporal blocking in 7-Point stencil, LBM and MergeSort) as well - more loops may need to be split and reordered. In tree search, there is hierarchical rearrangement of the tree in order to maximize data reuse at the granularity of both memory pages as well as cache lines.

In terms of performance improvement, Figure 5 shows that blocking results in an average of 1.6X (up to 4.3X for LBM and 7-point stencil) performance improvement. This benefit will grow as bandwidth-to-compute ratios continually decrease.

SIMD-friendly algorithms: In some cases, the naive algorithm cannot easily be vectorized either due to back-to-back dependencies between loop iterations or due to the heavy use of gathers and scatters in the code. A different algorithm that is more SIMD friendly may then need to be chosen. In some cases, dependencies between loop iterations can be resolved using cross-lane SIMD operations such as shuffle, maskmov or related operations. For instance, MergeSort involves a sequence of min, max and shuffle instructions. Using this code sequence results in a 2.5X speedup over scalar code. The inner loop of the code is shown in Figure 6(b). The code on the left shows the traditional MergeSort algorithm, where only two elements are merged at a time and the minimum written out. There are back-to-back dependencies due to the array increment operations, and hence the code cannot vectorize. Moreover, the code also heavily suffers from branch misprediction. The figure on the right shows code for a SIMD-friendly merging network [12], which merges two sequences of SIMD-width S sized elements using a sequence of min, max and interleave operations. This code auto-vectorizes with each highlighted line corresponding to one SIMD instruction. Moreover, branch mispredictions now occur every S number of elements. However, this code does have to do more computation (by a constant factor of $\log(S)$), but still yields a gain of 2.3X for 4-wide SIMD.

Since these algorithmic changes involve tradeoff between total computation and SIMD-friendliness, the decision to use them must be consciously taken by the programmer. Such changes do require effort on the part of the programmer - however, they will pay off over multiple platforms and generations of them.

Summary: Using well-known algorithmic techniques, we get an average of **2.4X performance gain** on 6-core Westmere. Moreover, as the number of cores and SIMD widths increase, and with reducing bandwidth-to-compute ratios, gains due to algorithmic changes will further increase.

4.2 Compiler Technology

Once algorithmic changes have been taken care of, we show the impact of utilizing the parallelization and vectorization technology present in recent compilers in bridging the Ninja gap.

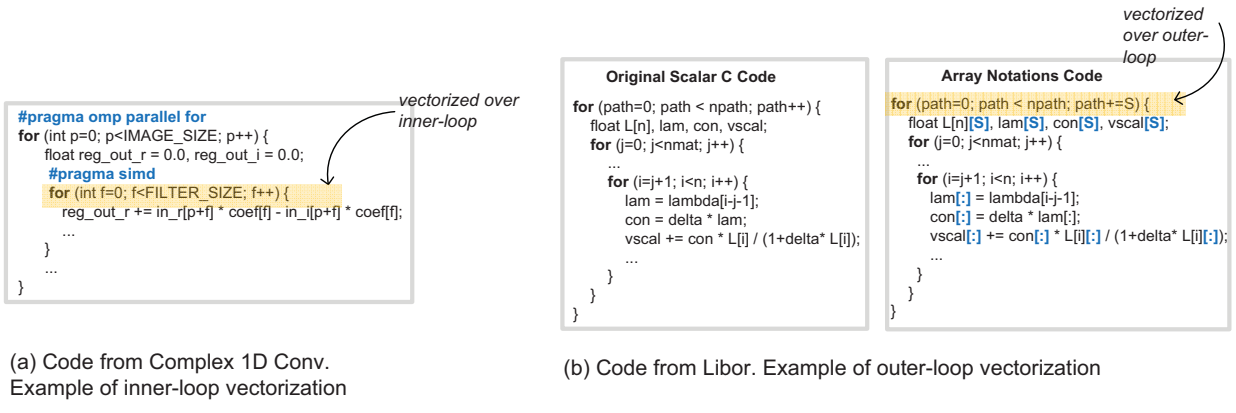
4.2.1 Parallelization

We parallelize our benchmarks using OpenMP pragmas typically over the outermost loop. OpenMP offers a portable solution that allows for specifying the number of threads to be launched, thread affinities to cores, specification of thread private and shared variables, as well as scheduling policies. Since throughput benchmarks offer significant thread-level parallelism that are typically present as an outer for loop, we generally use a **omp parallel for** pragma. One example is shown in Figure 7(a) for complex 1D convolution. In most cases, we obtain linear scaling with the number of cores after performing algorithmic optimizations to eliminate memory bandwidth bottlenecks. The use of SMT threads can help hide latency in the code - hence we sometimes obtain more than 6X scaling on our 6-core system.

4.2.2 Vectorization

SSE versus AVX: Figure 8(a) shows the benefit from inner and outer loop auto-vectorization for our benchmarks on our Westmere system, once proper algorithmic changes are made. We also compare it to the SIMD scaling for the manual best-optimized code. In terms of future scalability, we also show SIMD scaling on AVX (8-wide SIMD) in Figure 8(b) using a 4-core 3.4 GHz Intel® Core i7-2600K Sandybridge system. We use the same compiler for this study, and only change compilation flags to **-xAVX** from **-xSSE4.2**.

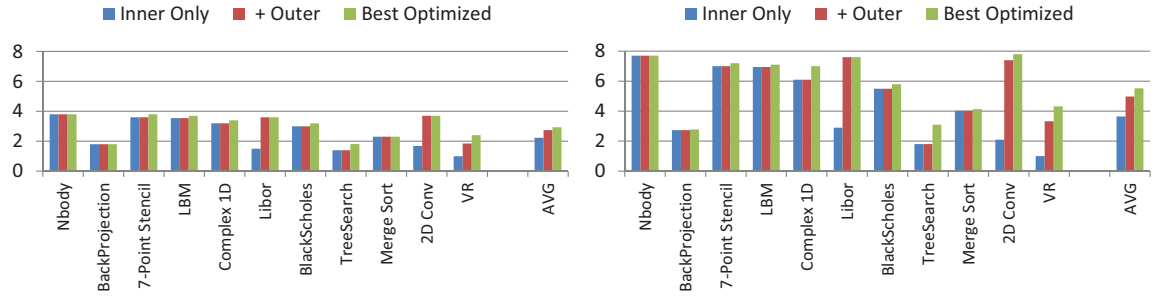
In terms of overall performance, we obtain on average about 2.75X SIMD scaling using compiled code, which is within 10% of the 2.9X scaling using best-optimized code on 4-wide SSE. With 8-wide AVX, we obtain 4.9X and 5.5X scaling (again very close to each other) using compiled and best-optimized code. TreeSearch accounts for most of the difference between compiled and best-optimized code. In TreeSearch, after we perform our algorithmic changes, we can perform SIMD comparisons over tree nodes at multiple tree levels, transfer the obtained bitvector register into a



(a) Code from Complex 1D Conv. Example of inner-loop vectorization

(b) Code from Libor. Example of outer-loop vectorization

Figure 7: **Code snippets showing compiler techniques for (a) Parallelization and inner loop vectorization in complex 1D convolution and (b) Outer loop vectorization in LIBOR. Note that the code changes required are small and can be achieved with low programmer effort.**



(a) SSE

(b) AVX

Figure 8: **Breakdown of benefits from inner and outer loop vectorization on (a) SSE and (b) AVX. We also compare to the best-optimized performance.**

general purpose register, and use it to compute the next child index. This sequence of operations is currently not generated by the compiler. The compiler currently only vectorizes over the query loop, involving gather/scatter operations. For current CPUs with no gather/scatter support, these are inefficient.

Our overall SIMD scaling for best-optimized code is good for most of our benchmarks, with the exceptions being MergeSort, TreeSearch and BackProjection. As explained in this section and in Section 4.1, we performed algorithmic changes in MergeSort and TreeSearch to enable SIMD scaling at the expense of performing more overall operations per element. Our SIMD scaling numbers take this into account, resulting in lower than linear speedups. Backprojection does not scale linearly due to the presence of unavoidable gathers and scatters in the code. Such operations cannot be vectorized and must be emulated using a series of sequential loads/stores. This limits SIMD scaling to 1.8X on SSE and 2.7X on AVX for backprojection.

Inner loop vectorization: Most of our benchmarks vectorize over the inner loop of the code, either by using compiler auto-vectorization or requiring the use of **#pragma simd** when dependence or memory alias analysis fails. The addition of this pragma, where required, is straightforward - this is a directive to the compiler that the loop must be (and is safe to be) vectorized. Figure 7(a) shows an example where this pragma is used in complex 1D convolution. Our average speedup for inner loop vectorization is 2.2X for SSE and 3.6X on AVX.

Outer loop vectorization: Vectorizing an outer-level loop has a unique set of challenges over vectorizing at the innermost loop level: Induction variables need to be analyzed for multiple loop levels; and loop control flows such as zero-trip test, number of iterations, and exit condition checks have to be converted into conditional (or predicated) execution on multiple vector elements. Register allocator is also impacted since vector variable live ranges are now larger and cross multiple basic blocks and multiple loop levels. The array notations technology helps the programmer avoid some of those complications without elaborate changes to the program structure. There are three benchmarks where we gain benefits from outer-loop vectorization. The first is LIBOR, where the inner loop contains a back-to-back dependence and is only partially vectorizable. In order to vectorize the outer (completely independent) loop, we currently use array notations. A part of the LIBOR code written in array notations is shown in Figure 7(b). Here vectorization occurs over the outer path loop. The scalar code is modified to change the loop index of the path loop to reflect the vectorization, as well as to compute results for multiple (equal to the simd width) loop iterations in parallel using arrays. Note that the programmer declares arrays of size S, the simd width for intermediate values, and the X[a:b] index notation stands for accessing b elements of array X starting with index a (X[:] is a shortcut that accesses the entire array X). It is usually straightforward to change scalar code to array notations code. This change results in high SIMD speedups of 3.6X on 4-wide SSE and 7.5X on AVX. Outer loop vectorization also benefits 2D convolution, where the inner loop does not have sufficient iterations (two nested loops of 5 iterations each) to have SIMD benefits. Vectorizing over the outer loop results in near-linear overall SIMD scaling. The third example is Volume Rendering, where the outer ray loop is vectorized using masks to handle control flow. These masks are emulated using registers for CPUs.

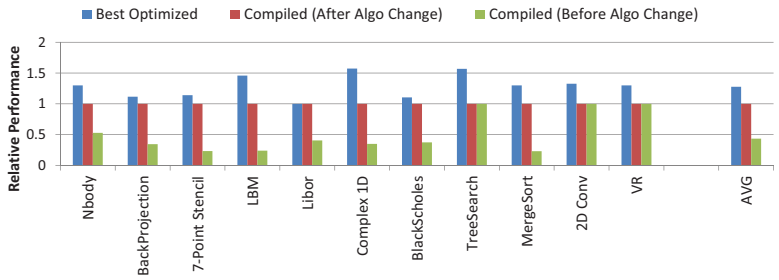


Figure 9: **Relative performance between the best-optimized code, the compiler-generated code after algorithmic change, and the compiler-generated code before algorithmic change. Performance is normalized to the compiled code after algorithmic change.**

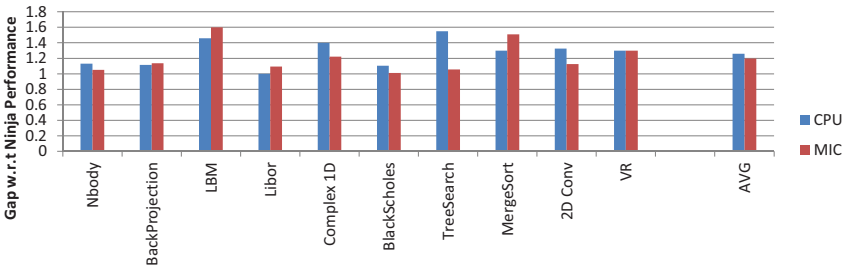


Figure 10: **Gap between best-optimized and compiler-generated code after algorithmic changes for MIC and CPU architectures.**

4.2.3 Fast Math

Applications such as NBody, and financial benchmarks (e.g, LIBOR and BlackScholes) are math-intensive, and use a variety of operations such as sqrt, rsqrt, divide, cos, sin, exp, etc. In many cases, the programmer has information about the final precision of the results of such computations, and may be willing to trade-off performance for accuracy. The compiler does not have knowledge of the precision requirements, and hence this needs to be achieved using compiler flags. The compiler however allows for user-defined flags such as **-fimf-precision** to control precision. The use of such flags enables the generation of lower precision code in NBody.

4.2.4 Hardware Gather Support

Our TreeSearch application requires gather operations for vectorization of the independent loop over queries. The auto-vectorizer emulates these gathers using scalar loads on SSE and AVX due to the absence of gather hardware support. While it is possible to perform algorithmic changes using SIMD blocking to vectorize the traversal of a single query, the SIMD benefit is inherently limited to a logarithmic factor of SIMD width (see [28] for details).

However, future architectures such as the Intel® MIC architecture [42] as well as the future Intel® Haswell architecture [23] have announced support in hardware for gathers. As such, the SIMD blocking algorithmic change is not required for MIC, and there is negligible difference between SIMD blocking code and code with gathers. In fact, the gap between compiled code and best-optimized code for TreeSearch on MIC is small. The addition of such hardware, along with compiler support to use the new instructions, thus has the benefit of reducing the Ninja gap.

5. SUMMARY

Figure 9 shows the relative performance of best-optimized code versus compiler generated code before and after algorithm changes (numbers are relative to compiled code after algorithmic change). We assume that the programmer has put in the effort to introduce the pragmas and compiler directives we described in previous sections. The figure shows that there is a **3.5X average gap** between compiled code and best-optimized code **before we perform algorithmic changes**. This gap is primarily because of the compiled code being bound by memory bandwidth or due to low SIMD efficiency due to sub-optimal memory layout. After we perform algorithmic changes described in Section 4.1, this gap shrinks to an average of 1.4X. The only benchmark with a significant (more than 1.5X gap) are Tree Search where the compiler vectorizes the outer loop with gathers. The rest of the benchmarks show 1.1-1.4X Ninja gaps, primarily due to extra instructions being generated due to additional spill/fill instructions, extra loads and stores instead to reusing certain values in registers - these are hard problems where the compiler relies on heuristics.

Impact of Future Architectures:
In order to see whether the Ninja gap will be low even on future platforms, we performed the same experiments on the Intel® MIC architecture, an Aubrey Isle [41] based silicon platform. We use the Intel® Knights Ferry software development platform with 32-cores running at 1.2 GHz. Each core features an in-order micro-architecture with 4-way SMT and a 512-bit SIMD unit.

Figure 10 shows the Ninja performance gap for MIC as well as for Westmere (SSE). Using an internal compiler for MIC, the average Ninja gap for MIC is only 1.2X, which is almost the same (slightly smaller) than the Ninja gap for CPUs. The main difference between the two performance gaps comes from Tree Search. As described in Section 4, TreeSearch benefits from the hardware gather support on MIC, and the compiled code that uses gathers is close in performance (1.1X) to the best-optimized code – as opposed to about 1.6X for CPU code. The rest of the benchmarks show similar performance gaps on MIC and CPU.

The remaining Ninja gap between best-optimized and compiled code after algorithmic changes remains small and stable across MIC and CPUs, inspite of the much larger number of cores and SIMD width on MIC. This is because our algorithmic optimizations focused on resolving vectorization and memory bandwidth bottlenecks in the code. Once these issues have been taken care of, future architectures will be able to exploit higher cores and SIMD width without being bottlenecked by the decreasing bandwidth-to-

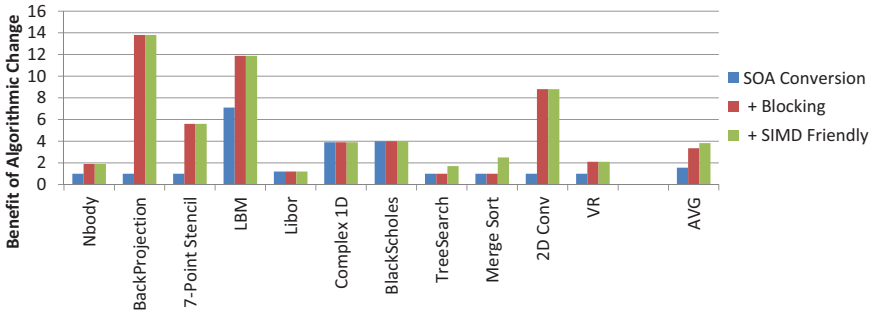


Figure 11: **Benefit of the algorithmic changes described in Figure 6 on the NVIDIA Tesla C2050 GPU.**

compute ratios. This will result in stable and predictable performance growth over future architectures.

Hardware Support For Programmability:
While we believe that the algorithmic changes we proposed - blocking, AOS to SOA conversion and SIMD-friendly algorithms are well-known, future hardware changes can reduce their impact, and further improve programmer productivity. We have already discussed how gather support on MIC makes SIMD-friendly algorithms for TreeSearch unnecessary; we now discuss potential hardware support for the other algorithmic changes. For blocking, the development of memory technologies such as 3-D stacked memory such as Hybrid Memory Cube [9, 46] can help reduce the impact of cache blocking. For AOS to SOA conversion, small Level-0 like AOS caches could be used. The cache lines involved in the data structure conversion could be loaded into the cache and the conversion performed there. This approach would have high latency for the first conversion, but subsequent conversions only access the cache and can have high throughput and low latency.

Another area where hardware support can help programmability is fast math operations. Currently, for applications where accuracy can be reduced, the compiler and/or programmer emulate these operations. Fast math hardware can either help programmers or simplify the compiler's job in trading off precision for performance.

6. DISCUSSION

The algorithmic optimizations that we described in Section 4.1 are applicable to a variety of architectures including GPUs. A number of previous publications [18, 39, 36] have discussed the optimizations needed to obtain best performance on GPUs. In this section, we show the impact of the same algorithmic optimizations described in Section 4.1 on GPU performance as applied to our benchmarks. We use the recent NVIDIA C2050 Tesla GPU for this study.

Although GPUs have hardware to handle gathers/scatters, GPU best coding practices (e.g. the CUDA C programming guide [36]) state the need to avoid uncoalesced global memory accesses – including converting data structures from AOS to SOA, for reducing memory latency and bandwidth usage. GPUs also require blocking optimizations, which here refers to the transfer and management of data into the shared memory (or caches) of the GPU. Finally, the use of SIMD-friendly algorithms greatly benefits the GPU that has a wider SIMD width than current CPUs.

Figure 11 shows the overall gains from performing algorithmic changes on the GPU. The average **performance gain from algorithmic optimizations is 3.8X** - higher than the 2.5X we gain on CPUs. This is because GPUs have more SMs and larger SIMD width, and hence sub-optimal algorithmic choices have a large impact on performance.

7. RELATED WORK

There have a number of papers published in a variety of fields that show 10-100X performance gains over previous work using carefully tuned code [45, 43, 14, 2, 28, 33]. Lee et al. [30] summarized relevant hardware architecture features and a platform-specific software optimization guide for a class of throughput applications on CPU and GPUs. While these works make it evident that a large performance gap exists between best-optimized and naively written code, they do not describe the programming effort involved or how to bridge the Ninja performance gap.

In this work, we analyze the sources of the Ninja gap and use traditional programming models to bridge the gap using low programmer effort. A previous version of this paper is available as a technical report [29]. Our work shows that by combining modern compiler technology with a set of simple and well-known algorithmic techniques to overcome architectural bottlenecks, we can bridge the gap to just 1.4X. Production compilers have recently started to support parallelization and vectorization technology that have been published in compiler research. Examples of such technology include OpenMP [10] for parallelization available in recent GCC and ICC compilers, as well as auto-vectorization technology [34], dealing with alignment constraints [17] and outer loop vectorization [35]. These technologies have been made available using straightforward pragmas and technology like array notations, a part of Intel® Cilk™ Plus [22].

However, naively written code may not scale with number of cores or SIMD width even with compiler support since they are bottlenecked by architectural features such as memory bandwidth, presence of gathers/scatters or because the fundamental algorithm cannot be vectorized due to tight dependencies. In such cases, algorithmic changes such as blocking, SOA conversion and SIMD-friendly algorithms are required. There have been various techniques proposed to address these algorithmic changes, either using compiler assisted optimization [27], using cache-oblivious algorithms [6] or specialized languages like Sequoia [21]. Such changes usually require programmer intervention and programmer effort, but they can be used across a number of architectures and generations. For instance, a number of papers have shown the impact of similar algorithmic optimizations on GPUs in CUDA [18, 39]. Further, a number of papers have made similar changes to CPUs and GPUs and shown benefit to both [33, 40].

While our work focuses on traditional programming models, there have been radical programming model changes proposed to bridge the gap. Recent suggestions include Bamboo [48] for an object oriented many-core programming approach, GPGPU approaches for parallelism management [47], the Berkeley View project [3] and OpenCL for programming heterogeneous systems. Our work makes a case that for a set of important real-world throughput applications, it is not necessary to adopt such models. There have also been library oriented approaches proposed such as Intel® Thread-

ing Building Blocks, Intel® Math Kernel Library, Microsoft Parallel Patterns Library (PPL), Intel® Integrated Performance Primitives etc. We believe these are orthogonal and used in conjunction with traditional models.

There is also a body of literature in adopting auto-tuning as an approach to bridging the Ninja gap for selected applications [44, 14]. Autotuning results can be significantly worse than the best-optimized code. For example, 7-point stencil computation is a specific application from our benchmark list for which auto-tuning results have been shown [14]. Our best-optimized code is about 1.5X better in performance than the auto-tuned code [33]. Since our Ninja gap for stencil is only 1.1X, our compiled code performs around 1.3X better than auto-tuned code. We expect our compiled results to be in general competitive with auto-tuned results, while offering the advantages of using standard tool-chains that can ease portability across processor generations.

Finally, there has been recent work that attempts to analyze the Ninja gap both for GPUs [7] and CPUs [31]. These works either focus narrowly on a single benchmark or do not use highly optimized code as a target to bridge the Ninja gap. For example, the benchmarks analyzed by Luk et al. [31] are 2-10X slower than our optimized performance numbers.

8. CONCLUSIONS

In this work, we showed that there is a large Ninja performance gap of 24X for a set of real-world throughput computing benchmarks for a recent multi-processor. This gap, if left unaddressed will inevitably increase. We showed how a set of simple and well-known algorithmic techniques coupled with advancements in modern compiler technology can bring down the Ninja gap to an average of **just 1.3X**. These changes only require low programming effort as compared to the very high effort in Ninja code.

9. REFERENCES

[1] S. J. Aarseth. *Gravitational N-body Simulations Tools and Algorithms*. 2003.

[2] N. Arora, A. Shringarpure, and R. W. Vuduc. Direct N-body Kernels for Multicore Platforms. In *ICPP*, pages 379–387, 2009.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. *Technical Report UCB/EECS-183*, 2006.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[5] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[6] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *SPAA*, pages 189–199, 2010.

[7] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. *IBM Research Report*, RC25033, August 2010.

[8] A. Brace, D. Gatarek, and M. Musiela. The Market Model of Interest Rate Dynamics. *Mathematical Finance*, 7(2):127–155, 1997.

[9] B. Casper. Reinventing DRAM with the Hybrid Memory Cube. `blogs.intel.com/research/2011/09/hmc.php`, 2010. Research@Intel.

[10] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. Parallel Programming in OpenMP, 2010.

[11] Y. K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, et al. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5):790–807, 2008.

[12] J. Chhugani, A. D. Nguyen, et al. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2):1313–1324, 2008.

[13] W. J. Dally. The End of Denial Architecture and the Rise of Throughput Computing. Keynote speech at Desgin Automation Conference, 2010.

[14] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[15] R. A. Drebin, L. C. Carpenter, and P. Hanrahan. Volume rendering. In *SIGGRAPH*, pages 65–74, 1988.

[16] P. Dubey. A Platform 2015 Workload Model: Recognition, Miniming and Synthesis Moves Computers to the Era of Tera. *Intel*, 2005.

[17] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93, 2004.

[18] F. Feinbube, P. Troger, and A. Polze. Joint Forces: From Multithreaded Programming to GPU Computing. *IEEE Softw.*, 28:51–57, January 2011.

[19] M. B. Giles. Monte carlo evaluation of sensitivities in computational finance. Technical report, Oxford University Computing Laboratory, 2007.

[20] A. G. Gray and A. W. Moore. 'N-Body' Problems in Statistical Learning. In *NIPS*, pages 521–527, 2000.

[21] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *PPoPP*, pages 143–152, 2008.

[22] Intel. A quick, easy and reliable way to improve threaded performance. `http://software.intel.com/en-us/articles/intel-cilk-plus/`, 2010.

[23] Intel. Intel Advanced Vector Extensions Programming Reference. *White paper*, June 2011.

[24] Intel. Optimization Notice. `http://software.intel.com/en-us/articles/optimization-notice/`, 2012.

[25] L. Ismail and D. Guerchi. Performance Evaluation of Convolution on the Cell Broadband Engine Processor. *IEEE PDS*, 22(2):337–351, 2011.

[26] M. Kachelrieb, M. Knaup, and O. Bockenbach. Hyperfast perspective cone-beam backprojection. *IEEE Nuclear Science*, pages 1679–1683, 2006.

[27] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. Irwin, et al. Cache topology aware computation mapping for multicores. In *PLDI*, 2010.

[28] C. Kim, J. Chhugani, N. Satish, et al. FAST: Fast Architecture Sensitive Tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[29] C. Kim, N. Satish, J. Chhugani, et al. Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology. Technical report, Intel Labs, 2011.

[30] 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. *ISCA*, pages 451–460, 2010.

[31] C.-K. Luk, R. Newton, et al. A synergetic approach to throughput computing on x86-based multicore desktops. *IEEE Software*, 28:39–50, 2011.

[32] T. N. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, 2001.

[33] A. Nguyen, N. Satish, et al. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *SC10*, pages 1–13, 2010.

[34] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO*, pages 281–294, 2006.

[35] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT*, pages 2–11, 2008.

[36] Nvidia. CUDA C Best Practices Guide 3.2, 2010.

[37] Oracle. Oracle TimesTen In-Memory Database Technical FAQ, 2007.

[38] V. Podlozhnyuk. Black-Scholes option pricing. *Nvidia*, 2007.

[39] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.

[40] N. Satish, C. Kim, J. Chhugani, et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[41] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *SIGGRAPH*, 27(3), 2008.

[42] K. B. Skaugen. HPC Technology-Scale-Up and Scale-Out. `lecture2go.uni-hamburg.de/konferenzen/-/k/10940`. ISC10 Keynote.

[43] M. Smelyanskiy, D. Holmes, et al. Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1563–1570, 2009.

[44] M. C. Sukop and D. T. Thorne, Jr. *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers*. 2006.

[45] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. A. Yelick. Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms. *J. Parallel Distrib. Comput.*, 69(9):762–777, 2009.

[46] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *HPCA*, pages 1–12, 2010.

[47] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.

[48] J. Zhou and B. Demsky. Bamboo: a data-centric, object-oriented approach to many-core software. In *PLDI*, pages 388–399, 2010.