

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

By Nadathur Satish, Changkyu Kim,\* Jatin Chhugani,\* Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey

## Abstract

Current processor trends of integrating more cores with wider Single-instruction multiple-data (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 designed. In this paper, we question this thinking and offer evidence in support of traditional programming methods and the performance-versus-programming effort effectiveness of 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 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 show equally encouraging results for the upcoming Intel® Xeon Phi™ architecture 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,<sup>16</sup> and the free lunch for programmers is over.

Recent techniques for increasing processor

performance have a focus on integrating more cores with wider Single-instruction multiple-data (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<sup>8, 14, 17, 24</sup> that show 10–100X performance improvements for real-world applications through 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.<sup>a</sup>

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<sup>7</sup> and because they offer the most opportunity for exploiting architectural resources—leading to large Ninja gaps

<sup>a</sup> 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.

The original version of this paper was published in the *Proceedings of the 39<sup>th</sup> Annual International Symposium on Computer Architecture* (June 2012). IEEE Computer Society, Washington, D.C., 440–451.

\* This work was done when these authors were at Intel.

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® Xeon Phi™<sup>b</sup>, 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 is 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 many 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 naive code, while the optimized code takes full advantage of it. Second, the code may not be vectorized, thus under-utilizing the increasing SIMD widths. While auto-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 compilers 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.

Our analysis of code written by Ninja programmers show that such programmers put in significant effort to

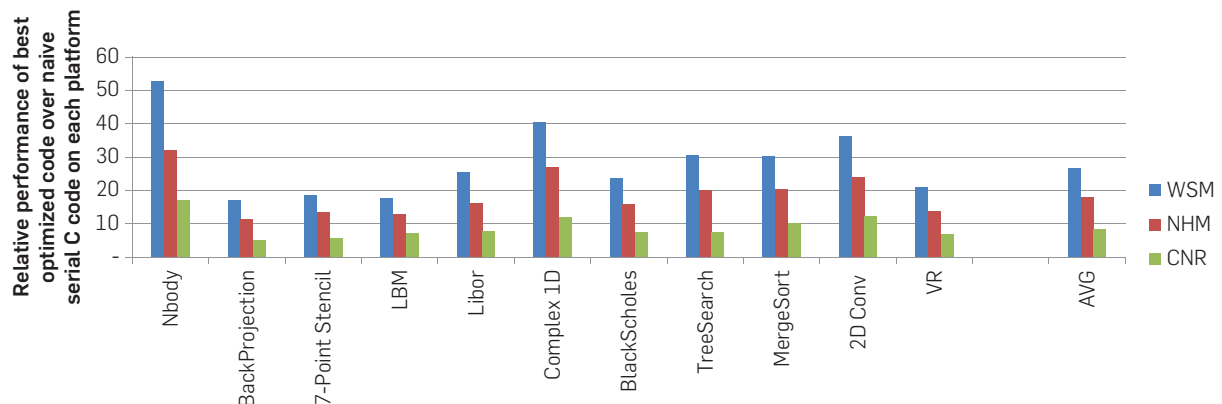
use threading technologies such as pthreads along with low level intrinsics for vectorization to obtain performance. This can result in very complex code especially when vectorizing irregular loops. In this work, we show that we can leverage recent compiler technologies that enable parallelization and vectorization of code with relatively low programmer effort. Parallelization can be achieved using OpenMP pragmas over the loop to be parallelized, and the programmer avoids writing complex threading code. For simple loops without dependencies, automatic loop parallelization is also possible—we assume the use of pragmas in this work. 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<sup>11</sup> helps the programmer to use this new functionality when it is not triggered automatically.<sup>c</sup> These features allow programmers to move away from using lower level intrinsics and/or assembly code and immensely boost performance. 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. This remaining gap, while relatively small, will however inevitably increase on future architectures with growing SIMD widths and decreasing bandwidth-to-compute ratios.

In order to bridge the remaining gap, programmer intervention is required. Current compilers do not automate changes at an algorithmic level that involve memory layout changes, and these must be manually performed by the programmer. We identify and suggest three critical algorithmic changes: **blocking** for caches, bandwidth/SIMD friendly **data layouts** and in some cases, choosing an

<sup>b</sup> Intel, Xeon and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

<sup>c</sup> For more complete information about compiler optimizations, see the optimization notice at <http://software.intel.com/en-us/articles/optimization-notice/>.

**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.**



**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.

Performing algorithmic changes does require programmer effort and insights, and we expect that education and training in parallel programming will play a big role in enabling programmers to develop better parallel algorithms. The payoffs are large—we show that **after performing algorithmic changes**, we have an **average performance gap of only 1.3X** between best-optimized and compiler-generated code. Moreover, this effort can be amortized across different processor generations and also across different computing platforms such as GPUs. Since the underlying hardware trends toward 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 **Intel® Xeon Phi™ Knights Corner co-processor architecture**, a recent 86X 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. The combination of algorithmic changes coupled with modern compiler technology is an important step toward 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,<sup>2, 3, 5</sup> 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. These include workloads from areas of High Performance Computing, Financial Services, Image Processing, Databases, etc.<sup>5</sup> 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 with *compute* and *memory characteristics* influencing the design of current and upcoming multi-/many-core processors. They also offer the most opportunity for exploiting architectural resources—leading to large Ninja gaps if naive code does not take advantage of increasing computational resources. We formulated a representative set of benchmarks described

below that **cover this wide range of application domains** of throughput computing. We capture the key computational kernels where most time is spent in throughput computing applications. As such, reducing Ninja gap in our benchmarks will also translate to the applications themselves.

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

**2. BackProjection:** Backprojection is a commonly used kernel in performing cone-beam reconstruction of CT data.<sup>13</sup> A set of 2D images are “back-projected” onto a 3D volume in order to construct the grid of density values. For each input image, each 3D grid point is projected onto the image, and the density from the neighboring  $2 \times 2$  pixels is bilinearly interpolated and accumulated.

**3. 7-Point Stencil:** Stencil computation is used for a wide range of scientific disciplines.<sup>8</sup> 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 neighbors, and stores the computed value to the output grid.

**4. Lattice Boltzmann Method (LBM):** LBM is a class of computational fluid dynamics capable of modeling complex flow problems.<sup>25</sup> 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<sup>4</sup> is used to price a portfolio of swaptions. It models a set of forward rates as a log-normal distribution. A typical Monte Carlo approach would generate random samples for this distribution and compute the derivative price using a large number of independent paths.

**6. Complex 1D Convolution:** This is widely used in application areas like image processing, radar tracking, etc. It performs 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, there is a closed form expression for the solution of the PDE.<sup>20</sup> This involves a number of math operations: computation of exponent, log, square-root, and division operations.

**8. TreeSearch:** In-memory tree structured index search is commonly used in commercial databases.<sup>14</sup> 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 to the node value at each tree level.

**9. MergeSort:** MergeSort is commonly used in the area of databases, etc.,<sup>6</sup> and also shown to be the sorting algorithm of choice for future architectures.<sup>22</sup> It sorts an array of  $\mathcal{N}$  elements using  $\log \mathcal{N}$  merge passes over the complete array.

**10. 2D  $5 \times 5$  Convolution:** Convolution is a common

image filtering operation used for effects such as blur, emboss, etc.<sup>15</sup> For a given 2D image and a  $5 \times 5$  spatial filter, each pixel computes and stores the weighted sum of a  $5 \times 5$  neighborhood of pixels, where the weights are the corresponding values in the filter.

**11. Volume Rendering:** Volume Rendering is commonly used in the fields of medical imaging,<sup>24</sup> etc. Given a 3D grid, and a 2D image location, the benchmark spawns rays perpendicular to the image plane through the 3D grid, which accumulates the color and opacity to compute the final color of each pixel of the image.

**Ninja Performance:** Table 1 provides details of the representative dataset sizes for each of the benchmarks. There exists a corresponding best performing code for each, for which the performance numbers have been previously cited<sup>d</sup> on different platforms than those used in our study. In order to perform a fair comparison, we **implemented and aggressively optimized (including the use of intrinsics/assembly code) the benchmarks**, 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 numbers 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. For a detailed performance analysis, we refer the reader to our ISCA paper.<sup>23</sup>

**Platform:** We measured the performance on a 3.3 GHz 6-core Intel® Core™ i7 X980 (code-named Westmere, peak compute: 158 GFlops, peak bandwidth: 30 GBps). The cores feature an out-of-order super-scalar micro-architecture,

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

Benchmark	Dataset	Best Optimized Performance
NBody <sup>1</sup>	10 <sup>6</sup> bodies	$7.5 \times 10^9$ Pairs/sec
BackProjection <sup>13</sup>	500 images on 1K <sup>3</sup>	$1.9 \times 10^9$ Proj./sec
7 Point 3D Stencil <sup>17</sup>	512 <sup>3</sup> grid	$4.9 \times 10^9$ Up./sec
LBM <sup>17</sup>	256 <sup>3</sup> grid	$2.3 \times 10^8$ Up./sec
LIBOR <sup>10</sup>	10M paths on 15 options	$8.2 \times 10^5$ Paths/sec
Complex 1D Conv. <sup>12</sup>	8K on 1.28M pixels	$1.9 \times 10^8$ Pixels/sec
BlackScholes <sup>20</sup>	1M call + put options	$8.1 \times 10^8$ Options/sec
TreeSearch <sup>14</sup>	100M queries on 64M tree	$7.1 \times 10^7$ Queries/sec
MergeSort <sup>6</sup>	256M elements	$2.1 \times 10^8$ Data/sec
2D 5X5 Convolution <sup>15</sup>	2K $\times$ 2K Image	$2.2 \times 10^9$ Pixels/sec
Volume Rendering <sup>24</sup>	512 <sup>3</sup> volume	$2.0 \times 10^8$ Rays/sec

<sup>d</sup> The best reported numbers are cited from recent top-tier publications in the area of Databases, HPC, Image processing, etc. To the best of our knowledge, there *does not exist* any faster performing code for any of the benchmarks.

with 2-way Simultaneous Multi-Threading (SMT). It also has 4-wide SIMD units that support a wide range of instructions. Each core has an individual 32 KB L1 and 256 KB L2 cache. The cores share a 12 MB last-level cache (LLC). Our system has 12 GB RAM and runs SuSE Enterprise Linux (ver. 11). We use the Intel® Composer XE 2011 compiler.

**Methodology:** For each benchmark, we attempt to first get good single thread performance by exploiting instruction and data level parallelism. To exploit data level parallelism, we measure the SIMD scaling for each benchmark by running the code with auto-vectorization enabled/disabled (-no-vec compiler flag). If SIMD scaling is not close to peak, 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. Finally, we make necessary algorithmic changes to overcome the 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. Details about the specific compiler techniques are available in Tian et al.<sup>26</sup> The compiler directives we add to the code and command line are the following:

- ILP optimizations: We use `#pragma unroll` directive before an innermost loop, and `#pragma unroll_and_jam` primitive outside an outer loop. Both optionally accept the number of times a loop is to be unrolled.
- Vectorizing at innermost loop level: If auto-vectorization fails, the programmer can force vectorization using `#pragma simd`. This is a recent feature introduced in Cilk Plus.<sup>11</sup>
- Vectorizing at outer loop levels: This can be done in two different ways: (1) directly vectorize at outer loop levels, and (2) Stripmine outer loop iterations and change each statement in the loop body to operate on the strip. 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.

**1. Nbody:** We implemented Nbody on a dataset of 1 million bodies (16 MB memory). Figure 2 shows the breakdown of the various optimizations. The code consists of two loops that iterate over all the pairs. We first performed unrolling optimizations to improve ILP, which gives a benefit of 1.4X. The compiler **auto-vectorizes** the code well with no programmer intervention and provides a 3.7X SIMD scaling. We obtained a parallel scaling of 3.1X, which motivates the need for our algorithmic optimization of blocking the data structures to fit in L3 cache (**1-D blocking**, code in Section 4.1). Once blocking is done, we obtain an additional 1.9X thread scaling, and a 1.1X performance gap between compiled and best-optimized code.

**2: BackProjection:** We back-project 500 images of dimension  $2048 \times 2048$  pixels onto a  $1024 \times 1024 \times 1024$  3D grid. Backprojection requires 80 ops per grid point. Both the image (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 only 1.8X. This is because the code is **bandwidth-bound** (1.6 bytes/flop). We perform blocking over the 3D volume to reduce bandwidth (**3D blocking** in Figure 2). Due to spatial locality, the image working set reduces accordingly. This results in the code becoming **compute-bound**. However, due to gathers which cannot be vectorized on CPU, SIMD scaling only improved by additional 1.6X (total 1.8X). We obtained additional 4.4X thread scaling (total 7.9X), showing benefits of SMT. The net performance is 1.1X off the best-optimized code.

**3: 7-Point 3D Stencil:** Application iterates over a 3D grid of points, and performs 8 flops of computation per point. A 3D dataset with  $512 \times 512 \times 512$  grid points is used. Figure 2 shows that we get a poor SIMD scaling of 1.8X from auto-vectorization (**bandwidth bound**). In order to improve the scaling, we perform both **spatial** and **temporal blocking** to improve the performance.<sup>17</sup> The resultant code performs *four* time-steps simultaneously, and improves the DLP by a further 1.7X (net SIMD scaling of 3.1X—lower than 4X due to the overhead of repeated computation on the boundary). The thread scaling is further boosted by 2.5X (overall 5.3X). The net performance is within 10.3% of the best-optimized code.

**4. Lattice Boltzmann Method (LBM):** The computational pattern is similar to the stencil kernel. We used a  $256 \times 256 \times 256$  dataset. Figure 2 shows that our initial code (SPEC CPU2006) does not achieve any SIMD scaling, and 2.9X core-scaling. The reason for no SIMD scaling is the AOS data layout that results in gather operations. In order to improve 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 3.5D blocking. The resultant code further boosts SIMD scaling by 1.3X, achieving a net scaling of 2.2X. The resultant thread scaling was further increased by 1.95X (total 5.7X). However, the

compiler generated extra spill/fill instructions, that resulted in performance gap of 1.4X.

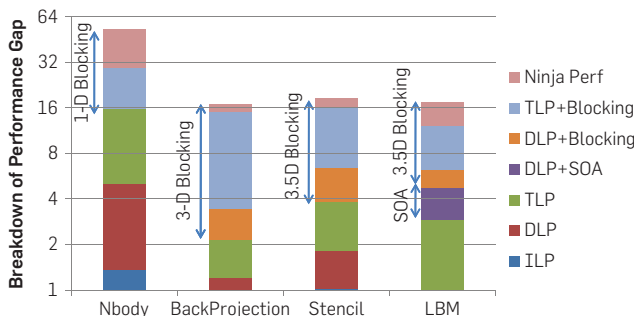
**5: LIBOR:** LIBOR<sup>4</sup> has an outer loop over all paths of the Monte Carlo simulation, and an inner loop over the forward rates on a single path. Figure 3 shows performance benefit of only 1.5X from auto-vectorization, since the current **compiler only attempts to vectorize the inner loop**, which has back-to-back dependencies and can only be partially vectorized. To solve this issue, we performed an algorithmic change to convert layout from AOS to SOA. We use the array notations technology to express outer loop vectorization (code in Figure 7b). Performing these changes allowed the outer loop to vectorize and provides additional 2.5X SIMD scaling (net 3.8X). The performance is similar to the best-optimized code.

**6. Complex 1D Convolution:** We use an image with 12.8 million points, and a kernel size of 8K. The first bar in Figure 3 shows the performance achieved by the unrolling enabled by the compiler, which results in 1.4X scaling. The auto-vectorizer only achieves a scaling of 1.1X. The TLP achieved is 5.8X. In order to improve the SIMD performance, we perform a **rearrangement of data** from AOS to SOA format. As a result, the compiler produces efficient SSE code, and the performance scales up by a further 2.9X. Our overall performance is about 1.6X slower than the best-optimized numbers (inability of the compiler to block the kernel weights).

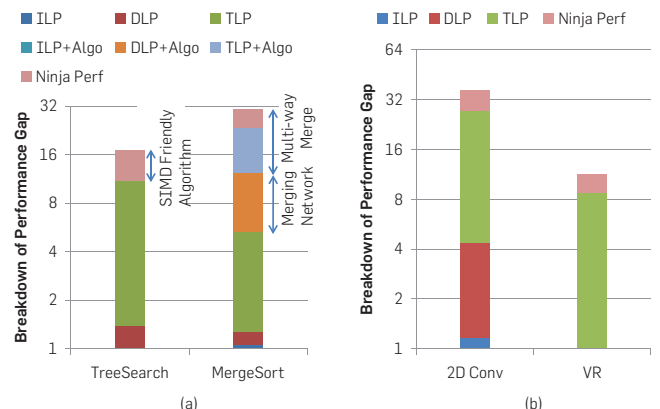
**7. BlackScholes:** BlackScholes computes the call and put options together. The total computation is 200 ops, while the bandwidth is 36 bytes. Figure 3 shows a SIMD speedup of 1.1X using auto-vectorization. The low scaling is primarily due to the AOS layout, which results in gather operations. To improve performance, we *changed the data layout* from AOS to SOA. As a result, the auto-vectorizer generated SVML (short vector math library) code, resulting in an increase in scaling by 2.7X (total 3.0X). The net performance is within 1.1X of the best performing code.

**8. TreeSearch:** The binary tree is laid out in a breadth-first fashion. The auto-vectorizer achieves a SIMD speedup

**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, Stencil, and LBM. The algorithm change involves blocking.**



**Figure 3. Breakdown of Ninja Gap for (a) Treesearch and Mergesort and (b) 2D convolution and VR. The benchmarks in (a) require rethinking algorithms to be more SIMD-friendly, while in (b) do not require any algorithmic changes.**



of 1.4X (Figure 4a). This is because it operates simultaneously on 4 queries, and each query may traverse down a different path — resulting in a gather operation. In order to improve performance, we perform an **algorithmic change**, and traversed 2 levels at a time (similar to SIMD width blocking<sup>14</sup>). However, the compiler did not generate the described code sequence, resulting in a 1.55X Ninja gap.

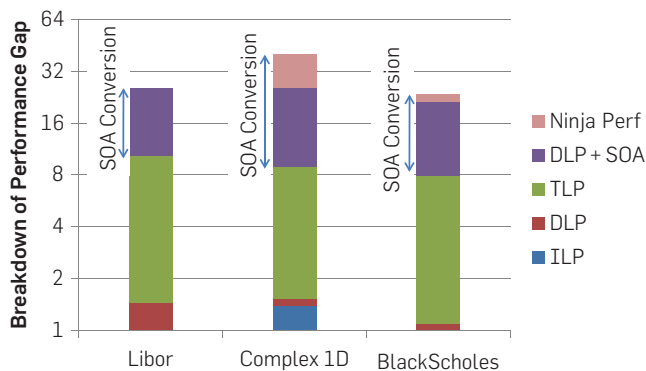
**9. MergeSort:** Our analysis is done for sorting an input array with 256 M elements. Figure 4a shows that we get a 1.2X scaling from auto-vectorization. This is due to gather operations for merging *four* pairs of lists. Parallel scaling is only 4.1X because the last few merge phases being bandwidth bound. In order to improve performance, we perform the following *two* **algorithmic changes**. Firstly, we implement merging of lists using a merging network<sup>6</sup> (code in Section 4.1). Secondly, in order to reduce the bandwidth requirement, we perform multiple merge phases together. 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.

**10. 2D Convolution:** We perform convolution of a  $2K \times 2K$  image with a  $5 \times 5$  kernel. The code consists of four loops. Figure 4b shows that we obtained a benefit of 1.2X by loop unrolling. We implemented the two inner loops using the array notations technology. That enabled vectorization of the outer loop, and scaled 3.8X with SIMD width. The thread-level parallelism was 6.2X. Our net performance was within 1.3X of the best-optimized code.

**11. Volume Rendering:** As shown in Figure 4b, we achieve a TLP scaling of 8.7X (SMT of 1.5X). As far as DLP is concerned, earlier compiler versions did not vectorize the code due to various control-intensive statements. However, recent compilers vectorize the code using mask values for each branch instruction, and using proper masks to execute both execution paths for each branch. There is only a small 1.3X Ninja performance gap.

**Summary:** In this section, we analyzed each benchmark, and reduced the Ninja gap to within 1.1–1.6X by applying necessary algorithmic changes coupled with the latest compiler technology.

**Figure 4. Breakdown of Ninja Performance Gap for Libor, Complex 1D convolution, and BlackScholes. All benchmarks require AOS to SOA conversion to obtain good SIMD scaling.**



#### 4. ANALYSIS AND SUMMARY

In this section, we identify the steps taken to bridge the Ninja gap with low programmer effort. The key steps taken are to first perform a set of well-known algorithmic optimizations to overcome scaling bottlenecks, and secondly to use the latest compiler technology for vectorization and parallelization. We now summarize our findings with respect to the gains we achieve in each step.

##### 4.1 Algorithmic Changes

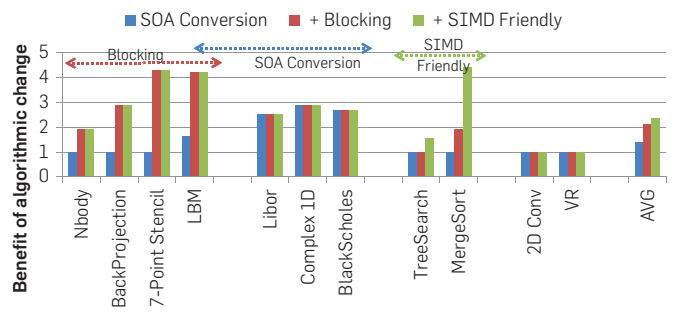
Algorithmic changes do require programmer effort and some insights, but are essential to avoid vectorization issues and bandwidth bottlenecks. Figure 5 shows the performance improvements due to set of well-known algorithmic optimizations that we describe below.

**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). Separate arrays for each field allows contiguous memory accesses when vectorization is performed. AOS structures require gathers and scatters, which can impact both SIMD efficiency and 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 advocated for a variety of architectures including GPUs.<sup>19</sup> Figure 5 shows that for our benchmarks, AOS to SOA conversion helped by an average of 1.4X.

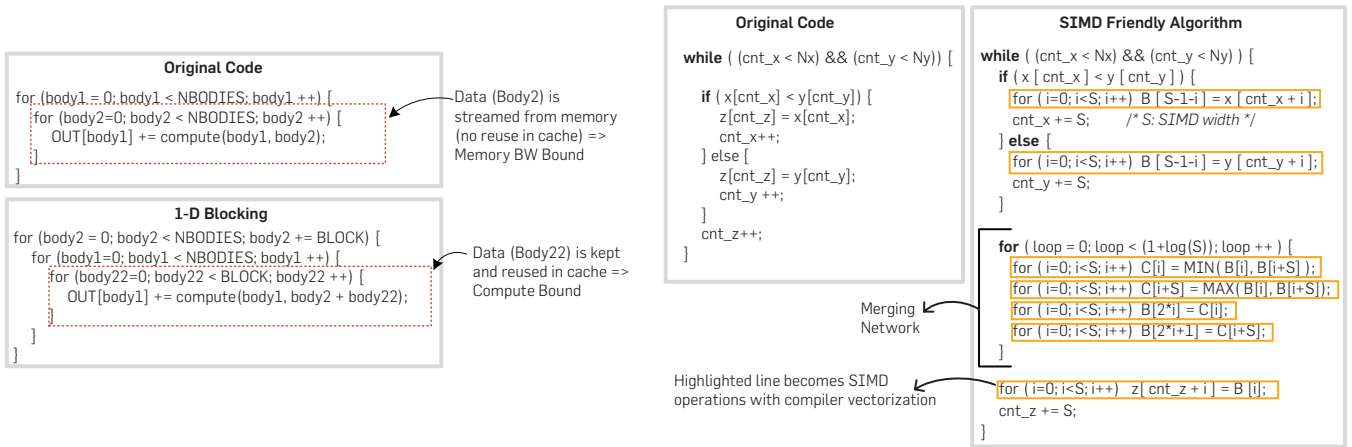
**Blocking:** Blocking is a well-known optimization that can help avoid bandwidth bottlenecks in a number of applications. The key idea is to exploit the inherent data reuse available in the application by ensuring that data remains in caches across multiple uses, both in the spatial domain (1-D, 2-D, or 3-D), and temporal domain.

In terms of code change, blocking involves a combination of loop splitting and interchange. The code snippet in Figure 6a 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* is

**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.**



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



obtained by 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. 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*. In terms of performance (Figure 5), we achieve an average of 1.6X improvement (up to 4.3X for LBM and 7-point stencil).

**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 be required. Figure 6b shows an example of this in MergeSort. The code on the left shows the traditional 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. The figure on the right shows code for a SIMD-friendly merging network,<sup>6</sup> which merges two sequences of SIMD-width *S* sized elements using a sequence of min, max and interleave ops. This code auto-vectorizes with each highlighted line corresponding to one SIMD instruction. 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*.

**Summary:** Using well-known algorithmic techniques, we get an average of **2.4X performance gain** on 6-core Westmere. With increasing cores, SIMD width, and compute-to-bandwidth 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.

**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/shared variables. Since throughput benchmarks offer significant TLP (typically outer for loop), we generally use a **omp parallel for** pragma. One example is shown in Figure 7a for complex 1D convolution. 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.

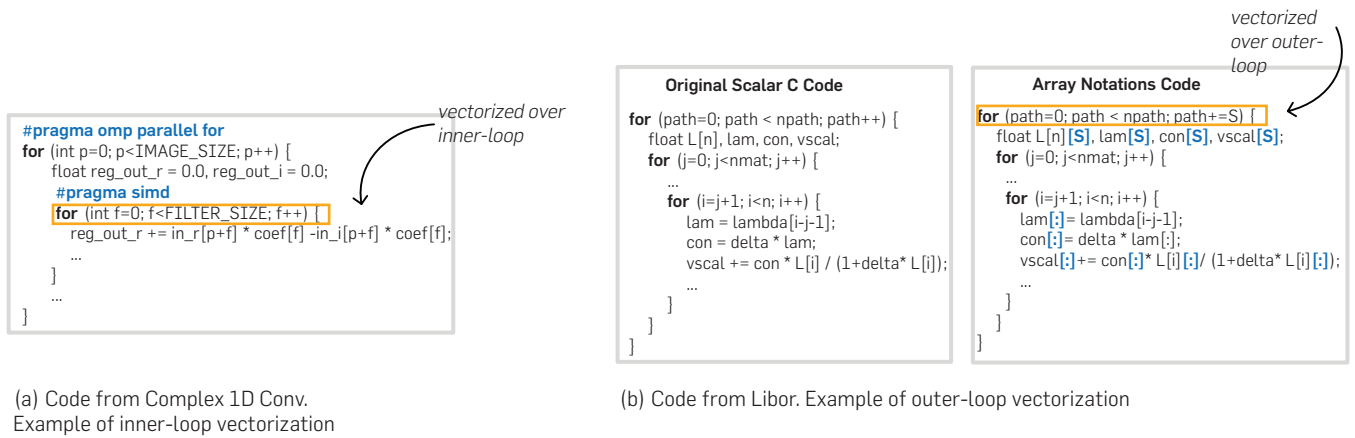
### Vectorization.

**SSE versus AVX:** Figure 8a shows the benefit from inner and outer loop auto-vectorization on Westmere, once proper algorithmic changes are made. We also compare it to the SIMD scaling for the manual best-optimized code, and show scaling on AVX (8-wide SIMD) in Figure 8b using a 4-core 3.4 GHz Intel® Core i7-2600 K Sandybridge system. We use the same compiler, and only change compilation flags to **-xAVX** from **-xSSE4.2**. In terms of performance, we obtain on average 2.75X SIMD scaling using compiled code, which is within 10% of the 2.9X scaling using best-optimized code. With 8-wide AVX, we obtain 4.9X and 5.5X scaling (again very close to each other) using compiled and best-optimized code.

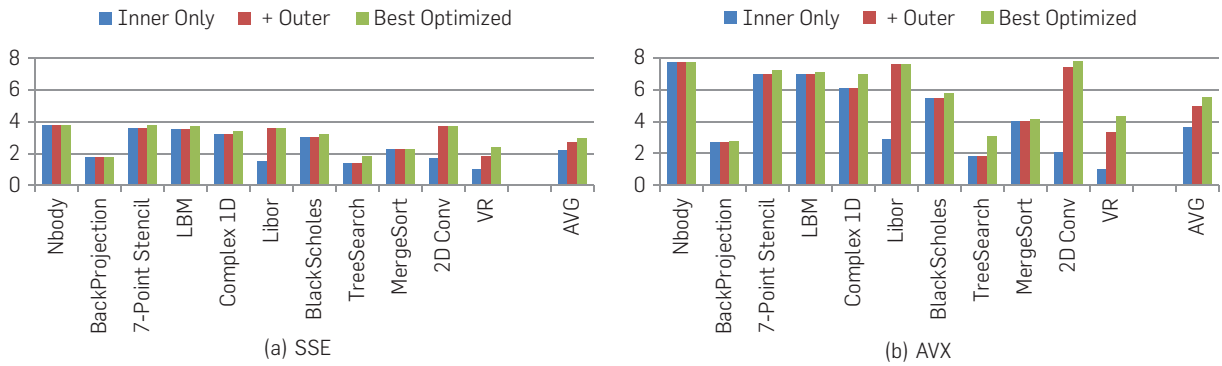
Our overall SIMD scaling for best-optimized code is good for most of our benchmarks, with the exceptions being MergeSort, TreeSearch and BackProjection. As also explained in Section 4.1, we performed algorithmic changes in MergeSort and TreeSearch at the expense of performing more operations, resulting in lower than linear speedups. Backprojection does not scale linearly due to the presence of unavoidable gathers/scatters in the code. This limits SIMD scaling to 1.8X on SSE (2.7X on AVX) for backprojection.

**Inner loop vectorization:** Most of our benchmarks

**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.**



**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.**



vectorize over the inner loop of the code, either by using compiler auto-vectorization or `#pragma simd` when dependence or memory alias analysis fails. The addition of this pragma is a directive to the compiler that the loop must be (and is safe to be) vectorized. Figure 7a shows an example where this pragma is used. Our average speedup for inner loop vectorization is 2.2X for SSE (3.6X on AVX).

**Outer loop vectorization:** Vectorizing an outer-level loop is challenging: Induction variables need to be analyzed for multiple loop levels; and loop control flows such as zero-trip test and exit conditions have to be converted into conditional/predicated execution on multiple vector elements. The array notations technology helps the programmer avoid those complications without elaborate program changes. We gain benefits from outer-loop vectorization in LIBOR, where the inner loop is only partially vectorizable. We currently use array notations to vectorize the outer (completely independent) loop (Figure 7b). The scalar code is modified to change the outer loop index to reflect the vectorization, and compute results for multiple iterations in parallel. Note that the programmer declares arrays of size  $S$  (simd width), and  $X[a:b]$  notation stands for accessing  $b$  elements of  $X$ , starting with index  $a$ . It

is usually straightforward to change scalar code to array notations code. This change results in high speedups of 3.6X on SSE (7.5X on AVX).

### 5. SUMMARY

Figure 9 shows the relative performance of best-optimized code versus compiler generated code before and after algorithm changes. We assume that the programmer has put in the effort to introduce the pragmas and compiler directives described in previous sections. 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 low SIMD efficiency. After we perform algorithmic changes described in Section 4.1, this gap shrinks to avg. 1.4X. The only benchmark with a significant gap is TreeSearch, where the compiler vectorizes the outer loop with gathers. The rest show 1.1–1.4X Ninja gaps, primarily due to extra instructions being generated due to additional spill/fill instructions, loads/stores—these are hard problems where the compiler relies on heuristics.

**Impact of Many-core Architectures:** In order to test the Ninja gap on many-core platforms, we performed the same



experiments on the Intel® Xeon Phi™ “Knights Corner” co-processor (KNC), which has 60/61 cores on a single die, and each core features an in-order micro-architecture with 4-way SMT and 512-bit SIMD unit.

Figure 10 shows the Ninja performance gap for KNC as well as for Westmere. The average Ninja gap for KNC is only 1.2X, which is almost the same (slightly smaller) than the CPUs. The main difference between the two performance gaps comes from TreeSearch, which benefits from the hardware gather support on Xeon Phi, and is close in performance (1.1X) to the best-optimized code.

The remaining Ninja gap after algorithmic changes remains small and stable across KNC and CPUs, inspite of the much larger number of cores and SIMD width on KNC. This is because our algorithmic optimizations focused on resolving vectorization and bandwidth bottlenecks in the code. Once these issues have been taken care of, future architectures will be able to exploit increasing hardware resources yielding stable and predictable performance growth.

## 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<sup>19,21</sup> have discussed the optimizations needed to obtain best performance on GPUs. In this section, we show the impact of the same

algorithmic optimizations on GPU performance. We use the NVIDIA C2050 Tesla GPU for this study.

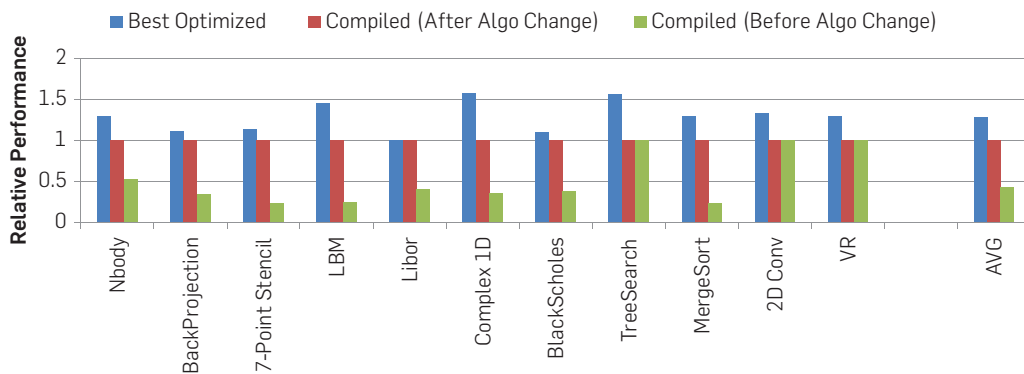
Although GPUs have hardware gather/scatters, best coding practices (e.g., the CUDA C programming guide<sup>19</sup>) state the need to avoid uncoalesced global memory accesses—including converting data structures from AOS to SOA for reducing latency and bandwidth usage. GPUs also require blocking optimizations, which refers to the transfer/management of data into the shared memory (or caches) of GPUs. Finally, the use of SIMD-friendly algorithms greatly benefits GPUs that have a wider SIMD width than current CPUs. The average **performance gain from algorithmic optimizations is 3.8X** (Figure 11)—higher than the 2.5X gain on CPUs, since GPUs have more SMs and larger SIMD width, and hence sub-optimal algorithmic choices have a large impact on performance.

## 7. RELATED WORK

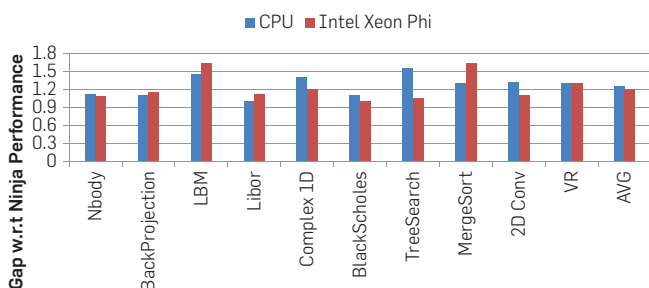
There are a number of published papers that show 10–100X performance gains over previous work using carefully tuned code.<sup>1,8,14,17,24</sup> Lee et al.<sup>15</sup> summarized relevant hardware architecture features and a platform-specific software optimization guide on CPU and GPUs. While these works show the existence of a large Ninja performance gap, they do not describe the programming effort or how to bridge the Ninja gap.

In this work, we analyze the sources of the Ninja gap and use traditional programming models to bridge it

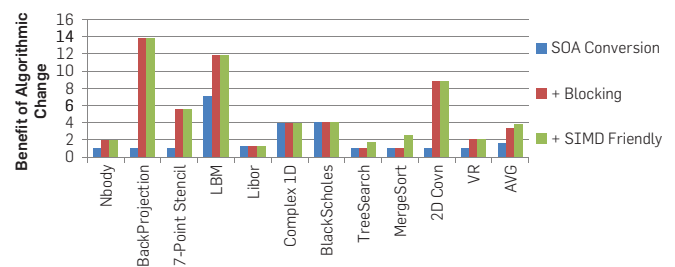
**Figure 9. Relative performance between the best-optimized code, the compiler-generated code after algorithmic change, and the compiler-generated code before algorithmic change. Performance is normalized to the compiled code after algorithmic change.**



**Figure 10. Gap between best-optimized and compiler-generated code after algorithmic changes for Intel® Xeon Phi™ and CPUs.**



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




using low programmer effort. A previous version of this paper was published in Satish et al.<sup>23</sup> Production compilers have recently started to support parallelization and vectorization technology that have been published in compiler research. Examples of such technology include OpenMP for parallelization available in recent GCC and ICC compilers, as well as auto-vectorization technology,<sup>18</sup> dealing with alignment constraints and outer loop vectorization. These technologies have been made available using straightforward pragmas and technology like array notations, a part of Intel® Cilk™ Plus.<sup>11</sup>

However, naively written code may not scale even with compiler support since they are bottlenecked by architectural features such as memory bandwidth, gathers/scatters or because the algorithm cannot be vectorized. 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, using cache-oblivious algorithms or specialized languages. Such changes usually require programmer intervention and programmer effort, but 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.<sup>21</sup>

While our work focuses on traditional programming models, there have been radical programming model changes proposed to bridge the gap. Recent suggestions include Domain Specific Languages (a survey is available at Fowler<sup>9</sup>), the Berkeley View project<sup>2</sup> and OpenCL for programming heterogeneous systems. There have also been library oriented approaches proposed such as Intel® Threading Building Blocks, Intel® Math Kernel Library, Microsoft Parallel Patterns Library (PPL), 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 gap.<sup>8, 25</sup> Autotuning results can be significantly worse than the best-optimized code. For, for example, for autotuned stencil computation,<sup>8</sup> our best-optimized code is 1.5X better in performance. Since our Ninja gap for stencil is only 1.1X, our compiled code performs 1.3X better than auto-tuned code. We expect our compiled results to be in general competitive with autotuned results, while offering the advantages of using standard tool-chains that can ease portability across processor generations.

## 8. CONCLUSION

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. 

## References

- Arora, N., Shringarpure, A., Vuduc, R.W. Direct N-body Kernels for multicore platforms. In *ICPP* (2009), 379–387.
- Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/ECS-183, 2006.
- Bienia, C., Kumar, S., Singh, J.P., Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT* (2008), 72–81.
- Brace, A., Gatarek, D., Musiela, M. The market model of interest rate dynamics. *Mathematical Finance* 7, 2 (1997), 127–155.
- Chen, Y.K., Chhugani, J., et al. Convergence of recognition, mining and synthesis workloads and its implications. *IEEE* 96, 5 (2008), 790–807.
- Chhugani, J., Nguyen, A.D., et al. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB* 1, 2 (2008), 1313–1324.
- Daly, W.J. The end of denial architecture and the rise of throughput computing. In *Keynote Speech at Design Automation Conference* (2010).
- Datta, K. Auto-tuning Stencil Codes for Cache-based Multicore Platforms. PhD thesis, EECS Department, University of California, Berkeley (Dec 2009).
- Fowler, M. *Domain Specific Languages*. 1st edn. Addison-Wesley Professional, Boston, MA 2010.
- Giles, M.B. *Monte Carlo Evaluation of Sensitivities in Computational Finance*. Technical report. Oxford University Computing Laboratory, 2007.
- Intel. A quick, easy and reliable way to improve threaded performance, 2010. [software.intel.com/articles/intel-cilk-plus](http://software.intel.com/articles/intel-cilk-plus).
- Ismail, L., Guerchi, D. Performance evaluation of convolution on the cell broadband engine processor. *IEEE PDS* 22, 2 (2011), 337–351.
- Kachelriebe, M., Knaup, M., Bockenbach, O. Hyperfast perspective cone-beam backprojection. *IEEE Nuclear Science* 3, (2006), 1679–1683.
- Kim, C., Chhugani, J., Satish, N., et al. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD* (2010), 339–350.
- Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., et al. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *ISCA* (2010), 451–460.
- T. N. Mudge. Power: A first-class architectural design constraint. *IEEE Computer* 34, 4 (2001), 52–58.
- Nguyen, A., Satish, N., et al. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *SC10* (2010), 1–13.
- Nuzman, D., Henderson, R. Multi-platform auto-vectorization. In *CGO* (2006), 281–294.
- Nvidia. *CUDA C Best Practices Guide* 3, 2 (2010).
- Podlozhnyuk, V. Black-Scholes option pricing. *Nvidia*, 2007. [http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/BlackScholes/doc/BlackScholes.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/BlackScholes/doc/BlackScholes.pdf).
- Ryoo, S., Rodrigues, C.I., Bagsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.M.W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP* (2008), 73–82.
- Satish, N., Kim, C., Chhugani, J., et al. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *SIGMOD* (2010), 351–362.
- Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., et al. Can traditional programming bridge the Ninja performance gap for parallel computing applications? In *ISCA* (2012), 440–451.
- Smelyanskiy, M., Holmes, D., et al. Mapping high-fidelity volume rendering to CPU, GPU and many-core. *IEEE TVCG*, 15, 6(2009), 1563–1570.
- Sukop, M.C., Thorne, D.T., Jr. *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers*, 2006.
- Tian, X., Saito, H., Girkar, M., Preis, S., Kozhukhov, S., Cherkasov, A.G., Nelson, C., Panchenko, N., Geva, R. Compiling C/C++ SIMD extensions for function and loop vectorization on multi-core-SIMD processors. In *IPDPS Workshops* (Springer, NY, 2012), 2349–2358.

Nadathur Satish, Mikhail Smelyanskiy, and Pradeep Dubey, Parallel Computing Lab, Intel Corp.

Changkyu Kim, Google Inc.

Jatin Chhugani, Ebay Inc.

Hideki Saito, Rakesh Krishnaiyer, and Milind Girkar, Intel Compiler Lab, Intel Corp.

Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.