Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study

Kaixi Hou, Hao Wang, and Wu-chun Feng
Department of Computer Science
Virginia Tech, Blacksburg, Virginia, U.S.A.
Email: {kaixihou, hwang121, wfeng}@vt.edu

***Abstract*—Moore's Law effectively doubles the compute power of a microprocessor every 24 months. Over the past decade, however, this doubling in performance has been due to the doubling of the number of cores in a microprocessor rather than clock speed increases. Perhaps nowhere is this more evident than with the Intel Xeon Phi coprocessor. This manycore architecture exhibits not only massive inter-core parallelism but also intra-core parallelism via a wider SIMD width. However, for data-intensive applications, the bandwidth constraint of MIC hinders the full utilization of computational resources, especially when massive parallelism is required to process big data sets. Furthermore, the process of optimizing the performance on such platforms is complex and requires architectural expertise.**

To evaluate the efficacy of the Intel MIC ecosystem for "big data" applications, we use the Floyd-Warshall algorithm as a representative case study for graph applications. Our study offers evidence that traditional compiler optimizations can deliver parallel programmability to the masses on the Intel Xeon Phi platform. That is, developers can straightforwardly create manycore codes in the Intel Xeon Phi ecosystem that deliver significant speedup. The optimizations include reordering data-access patterns, adjusting loop structures, vectorizing branches, and using OpenMP directives. We start from the default serial algorithm and apply the above optimizations one by one. Overall, we achieve a 281.7-fold speedup over the default serial version. When compared with the default OpenMP Floyd-Warshall parallel implementation, we still achieve a 6.4-fold speedup. We also observe that the identically optimized code on MIC can outperform its CPU counterpart by up to 3.2-fold.

***Index Terms*—Intel Xeon Phi, MIC, graph, Floyd-Warshall, manycore, programmability**

I. Introduction

To address the power and area constraints of microprocessors, the rise of manycore hardware moves towards integrating a substantial number of symmetric cores on one die. These manycore accelerators, including graphics processing units (GPUs) and Intel Xeon Phi coprocessor, are being adopted in the field of high-performance computing (HPC) due to their superior performance and energy efficiency compared with traditional CPUs. For example, Tianhe-2, the No. 1 supercomputer on the June 2014 Top500 list [1] is equipped with Intel Xeon Phi coprocessors. Compared with traditional multicore CPUs, Intel Xeon Phi has an increased number of cores (61), an increased width in the single-instruction, multiple-data (SIMD) vector units, and the elimination of aggressive, on-die hardware optimizations, including out-of-order execution and branch prediction. Although optimizing applications on CPUs and GPUs has been studied extensively in recent years, accelerating applications on Intel Xeon Phi has not been as well understood due to the evolution of the architecture. Thus, there remains a major challenge in developing high-performance applications on Intel Xeon Phi architecture.

The Intel Xeon Phi coprocessor uses graphics, double data rate (GDDR) SDRAM as its primary physical memory on the device, which is distinct from the (slower) DDR SDRAM in the main memory of the host. In addition to the physical bandwidth of GDDR on Xeon Phi being much higher than DDR for the CPU, the compute capability of the coprocessor is even higher. For example, in our experimental environment, we have Intel Sandy Bridge-EP processors and Intel Xeon Phi coprocessor within the same node. We use Stream benchmarks [2] to get the sustainable memory bandwidth for both. The Intel Sandy Bridge-EP processor has 665.6 single-precision GFLOPS (2 $\times$ 8 cores $\times$ 8 SIMD width $\times$ 2.6 GHz $\times$ 2 for fused multiply-add instruction, FMA for short) with 78GB/s sustainable memory bandwidth, leading to 8.54 ops/byte. The Intel Xeon Phi coprocessor has 2148 single precision GFLOPS (61 cores $\times$ 16 SIMD width $\times$ 1.1 GHz $\times$ 2 for FMA) with 150GB/s sustainable memory bandwidth, leading to 14.32 ops/byte. The operations per byte ratio means that to fully utilize all parallel capability of hardware, the application should at least contain that amount of operations for each byte access on the memory. From the point of view of applications, having higher operations per byte often leads to greater difficulty in achieving the peak performance of the hardware. In other words, the bandwidth constraint is more likely to be encountered on the hardware with higher number of operations per byte, like with the Intel Xeon Phi coprocessor.

Modern compiler techniques can automatically vectorize and speed-up a myriad of applications without much modification of algorithms on the modern multicore processors [3]. For applications containing complex loop structures or data dependency, smart algorithmic adjustments and appropriate directives are essential to guide the compiler to generate effective and efficient codes. As a contrast, explicitly handle

This work was supported in part by NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC).

such optimizations so as to improve application performance on multicore and manycore processors by utilizing SIMD intrinsics and Pthreads are also productive as shown in the previous studies [4], [5], [6]. Common to these studies is the application specific solution, which requires expertise knowledge of specific applications and hardware architectures, leading to the complexity for the manual optimizing and tuning methods. Furthermore, the obstacles of the portability will be encountered. For example, with the increasing SIMD unit width and the evolving SIMD intrinsic sets, the method explicitly using SIMD intrinsics to get the fine-grained data level parallelism in previous multicore systems cannot directly boost the performance of Intel Xeon Phi, which would require tons of labor work to rewrite whole programs using the upgraded ISA instructions. As a result, the optimizations using compiler directives are still promising on this emerging architecture. Reasonably taking advantage of such resources is of the significance for the data-intensive applications. However, these problems are not investigated thoroughly.

In this paper, we take the Floyd-Warshall algorithm as one case study of data-intensive applications to show how to use simple algorithmic adjustments with appropriate compiler directives to increase the application's performance on Intel Xeon Phi. The algorithmic adjustment includes the data blocking to improve the data locality of the memory access and the loop reconstruction for the efficient data-level parallelism. The compiler directives contain a set of pragmas to take advantage of both intra- and inter-core parallelism. Besides, we explore to search for the best combination of compiler and runtime parameters on Intel Xeon Phi by using a statistical tree-based partitioning approach, i.e., Starchart introduced in [7]. After applying these optimizations, we tune our implementation to achieve significant speedup and demonstrate the simplified programmability on Intel Xeon Phi. In our evaluations, we have observed up to 281.7-fold and 6.4-fold speedup on Intel Xeon Phi over the default serial implementation and the parallel OpenMP implementation, respectively. We also observe that the exactly same optimized implementation on Intel Xeon Phi can get up to 3.2-fold speedup over that on Intel Xeon multicore processor.

The remaining of this paper is organized as follows. Section II will provide the necessary background for our research. Section III will describe the methodology we used to optimize the Floyd-Warshall algorithm on Intel Xeon Phi. Section IV will present the evaluation results. Section V will discuss the related work in this field. We will summarize our research and present the future work in Section VI.

II. Background

A. Intel Xeon Phi Coprocessor

The Intel Xeon Phi coprocessor in our experiments consists of 61 in-order cores, each of which contains 32 512-bit SIMD registers and supports 4 hardware threads in an effort to hide memory access latency. To achieve the power efficiency, the cores operate at a relatively lower frequency and fewer pipeline stages. The memory hierarchy of Xeon Phi contains two levels of cache, i.e. 32 KB L1 data cache and 512 KB L2 cache. The Intel Xeon Phi coprocessor is physically connected to the host via PCIe. There are two programming models supported by the coprocessor. One is the *offload* mode, and the other is the *native* mode. The *offload* mode provides an explicit way to transfer data between host and coprocessor, just like using GPU. In contrast, the *native* mode reflects another salient feature of Intel Xeon Phi, i.e., the operating system installed on the coprocessor supports x86-compatible programs to execute directly, just like on traditional multicore CPUs. With the *native* mode, we can run OpenMP or Pthread applications without any modification on the coprocessor. In this paper, we will focus on the *native* mode.

Not only providing inter-core parallelism with numerous physical cores, the Intel Xeon Phi is also offering a rich ISA (Instruction Set Architecture) to support a quantity and a variety of vector, shuffle, and scalar operations. These new operations enhance the arithmetic computation performance and the data processing functionality. More importantly, this provides application developers the flexibility to take advantage of the intra-core parallelism by efficiently utilizing wider 512-bit SIMD registers. For example, the fused vector multiply-add operations provide more accurate results, the swizzle operations work as the lightweight version of their shuffle counterparts, and the reduction operations improve the programmability of using vectors. On the other hand, in order to use SIMD instructions, we usually prepare the data to each slot of the vector in advance, which will inevitably bring certain overheads. For example, considering the fact that the 512-bit register is comprised of 4 128-bit lanes, programmers often need to carry out the intra-lane and cross-lane shuffle operations to accommodate data for the subsequent SIMD operations, leading to performance penalty and increased complexity. This requires us to keep the balance between the benefits of using SIMD and the overheads caused by the data rearranging.

B. Floyd-Warshall Algorithm

The shortest path problem belongs to one of the basic and classic problems in the graph theory. There are two major variants of the shortest path problem: first is the single source shortest paths (SSSP) problem to compute the least-cost distance between the given pairs of vertices, and second is the all-pairs shortest paths (APSP) problem to compute the least-cost distances between all pairs of vertices in the graph. The Floyd-Warshall algorithm [8], [9] is a dynamic programming solution for APSP problem by using an increasing subset of the entire vertices as intermediate steps along the way. It has the complexity of $O(n^3)$ to keep track of the shortest path for all pairs of vertices. Algorithm 1 exhibits the naive Floyd-Warshall algorithm, which compares all the possible paths between each pair of vertices from *u* to *v* through the intermediate vertex *k*. The $|V|$ represents the number of vertices. The *dist* matrix keeps the original distances at the beginning and then changes to the shortest distances between each pair of vertices. Thus, $dist[u][v]$ represents the current

optimal distance from the vertex *u* to *v*. The *path* matrix is used to store the highest intermediate vertex on the path of each pair. Thus, $path[u][v]$ denotes the index of highest intermediate vertex that vertex *u* goes through to vertex *v*. The path flow reconstruction can be conducted recursively based on the *path* matrix.

Algorithm 1 Naive Floyd-Warshall Algorithm

```
1: INPUT: Distance matrix dist
2: OUTPUT: Updated Distance matrix dist storing the shortest distances between each pair of vertices; Path matrix path for path flow reconstruction
3: for k = 0 to |V| do
4:     for u = 0 to |V| do
5:         for v = 0 to |V| do
6:             if dist[u][k] + dist[k][v] ≤ dist[u][v] then
7:                 dist[u][v] ← dist[u][k] + dist[k][v]
8:                 path[u][v] ← k
9:             end if
10:        end for
11:    end for
12: end for
```

III. Methodology

In this section, we use Floyd-Warshall algorithm as a case study to explore the parallel programmability on Intel Xeon Phi. We attempt to start from the optimizations of the single-threaded version through exploiting better memory access pattern and data level parallelism. In an attempt to understand the SIMD instructions adopted by the compiler, we also show our preliminary implementation that uses the SIMD intrinsics explicitly to exploit the data level parallelism. Then, we achieve thread level parallelism by inserting appropriate OpenMP directives before loops. Finally, we use a statistical method to search the best combination of the compiler and runtime parameters for the optimized Floyd-Warshall algorithm on Intel Xeon Phi.

A. Improve Data Reuse

The high ratio of data access to computation is a challenge generally encountered by the parallel graph processing [10]. The high ratio may cause applications to be memory bandwidth bound, thereby leading to the under-utilization of computing resources. This motivates the need to put the reusable data into the cache. As a widely accepted optimization to enhance the data reuse, the data blocking [11] can help mitigate the negative effects of memory bandwidth bottlenecks in numerous graph algorithms. A typical loop transformation for data blocking involves a well-designed combination of loop splitting, interchanging, and fusing. However, due to the data dependency in the three-level nested loop of Floyd-Warshall algorithm shown in Algorithm 1, directly applying the aforementioned blocking techniques on such loops is impractical. Thus, we first change the naive Floyd-Warshall algorithm to the blocked Floyd-Warshall algorithm [12] to improve the data locality.

As shown in Figure 1 and Algorithm 2, the whole matrix *dist* is divided into blocks, whose dimension size is donated as *block_size*. The computations are divided into three steps based on the data dependency among the blocks: (1) update the block (k, k) along the diagonal. In each iteration, the block (k, k) is self-dependent, meaning that only the data inside this block are used for the computation; (2) update the blocks (k, j) on the same row and the blocks (i, k) on the same column with the block (k, k); (3) update remaining blocks (i, j), whose computation depends on the blocks (i, k) on its row and the blocks (k, j) on its column. The performance of the blocked Floyd-Warshall algorithm is affected by many factors, including the loop structure, the block size, and the underlying hardware features. Accordingly, we will carry out the loop reconstruction in Section III-B and explore the best combination of parameters in Section III-E, respectively. With these optimizations, the performance of the blocked algorithm will significantly outperform the naive implementation.

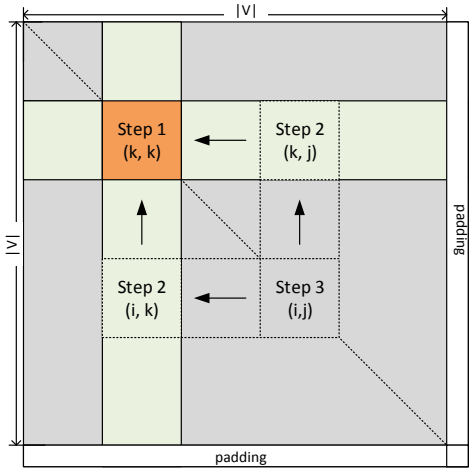

Fig. 1. The procedure of the blocked Floyd-Warshall algorithm and the data dependency in each step (the working area has been padded to the multiple of block size.)

B. Data Level Parallelism

The core computation of Floyd-Warshall algorithm, shown in the innermost loop of the Algorithm 1 and Algorithm 2, is updating the distance matrix based on the results of the comparison between two reals, which represent the cost of different routes. Although the presence of control flow may generally invalidate the auto-vectorization attempts of the compiler, it is still possible to auto-vectorize such codes since the comparisons in the "if" statements can be implemented as the masked operations [13]. However, in practice, for the blocked Floyd-Warshall, the compiler is still stalled on solving such loops and reports that the existence of potential vector dependence has prevented the auto-vectorization attempts. Therefore, we need to modify the implementation and tell the compiler it is safe to make the vectorization. Before we

Algorithm 2 Blocked Floyd-Warshall Algorithm

```
1: function UPDATE(int k_0, int u_0, int v_0)
2:     for k = k_0 to MIN(k_0 + block_size,|V|) do
3:         for u = u_0 to MIN(u_0 + block_size,|V|) do
4:             for v = v_0 to MIN(v_0 + block_size,|V|) do
5:                 if dist[u][k] + dist[k][v] ≤ dist[u][v] then
6:                     dist[u][v] ← dist[u][k] + dist[k][v]
7:                     path[u][v] ← k
8:                 end if
9:             end for
10:        end for
11:    end for
12: end function
13: //main function
14: for k = 1 to |V| : block_size do
15:    //step 1: update diagonal block (k, k)
16:    call UPDATE(k, k, k)
17:    //step 2: update blocks (k, j) on the row
18:    for j = 1 to |V| : block_size do
19:        call UPDATE(k, k, j)
20:    end for
21:    //step 2: update blocks (i, k) on the column
22:    for i = 1 to |V| : block_size do
23:        call UPDATE(k, i, k)
24:    end for
25:    //step 3: update other blocks (i, j)
26:    for i = 1 to |V| : block_size do
27:        for j = 1 to |V| : block_size do
28:            call UPDATE(k, i, j)
29:        end for
30:    end for
31: end for
```

describe our adjustment of the algorithm, we first introduce several directives that guide the Intel compiler to vectorize these loops. More information can be found in the guide of the Intel compiler [13].

- *pragma vector always*: vectorize the loop regardless of the efficiency, but the prerequisite is that the compiler has to believe it is safe to do so.
- *pragma ivdep*: tell the compiler the potential dependencies don't exist and it is safe to ignore them.
- *pragma simd*: launch user-mandated vectorization, which is the most aggressive directive.

Therefore, in this algorithm, the use of the *pragma ivdep* is sufficient to tell the compiler that the potential dependencies can be safely ignored and then make the innermost loop vectorized. This directive works fine with the step 1 (line 16) and the first loop in step 2 (line 19) of Algorithm 2. However, while dealing with the remaining two loops, the compiler reports "Top test could not be found" indicating the loop doesn't fulfill the requirements of auto-vectorization. This is caused by the compiler's analysis of the vector dependencies and the existence of the MIN operations. In Algorithm 2, we use the data padding technique to generate more efficient codes for SIMDization by aligning the data of each row. As a double-edged sword, the additional operations are necessary to check whether the data can fill the whole block in the last block of the column. As a result, three MIN operations are used in the UPDATE function to avoid the computation on the padded area, shown in Figure 1, when the value of $|V|$ is not a multiple of $block\_size$. The top portion (version 1) of Figure 2 illustrates the effective computation area and its corresponding codes. Because of the presence of such boundary checking in the loops, especially the innermost one, the compiler fails to generate efficient SIMDized codes. Even if we replace such MIN operations with variables prior to the loops (version 2), the same problem is still encountered. In order to resolve this problem, we modify the code by conducting the redundant computation on the padded area (version 3). Since we don't take the results of the redundant computation back as input, i.e., set k always within 1 to $|V|$, we can guarantee the correctness of the result. In the codes of version 3 in the Figure 2, we remove both MIN operations in the two innermost loops and at the same time make sure that the elements outsides the boundary (dark gray area) don't be used as inputs by keeping the MIN operation in the outermost loop to load data. After such loop reconstructions, all the innermost loops in Algorithm 2 can be efficiently auto-vectorized by Intel compiler.

C. Manual DLP Optimization

To better understand the mechanisms of the auto-vectorization adopted by the compiler, we also explore the SIMD operations by explicitly using the intrinsics from Intel Xeon Phi's ISA. The pseudo-code for the core computation of Floyd-Warshall algorithm is listed in Algorithm 3. The intermediate vertex between any pair of vertices along the possible shortest paths is designated by *k*. The result of the vector comparison operation between the old and new distance values, stored in SIMD variable *upd_v* and *sum_v* respectively, is represented by one 16-bit mask *cmp_m*, where each bit is set to one if the comparison of corresponding pair of elements is true. Once the mask is available, it is then served as the write mask for the masked variant of store operation to update the distance and path values in the destination memory *dist* and *path*.

In our experiments, we have observed better performance of the auto-vectorization described in the previous subsection than our manual implementation in this subsection. Since this is only our preliminary work to explore the vectorization on Xeon Phi, there still exists optimization space for better performance, such as prefetching. On the other hand, the outperforming result of the auto-vectorized code illustrates even with the traditional techniques, i.e., the data blocking, the loop reconstruction, and the proper pragmas, the compiler can generate the highly efficient code.

D. Thread Level Parallelism

As we move to a higher level of optimization, we use OpenMP pragmas to exploit thread-level parallelism of In-

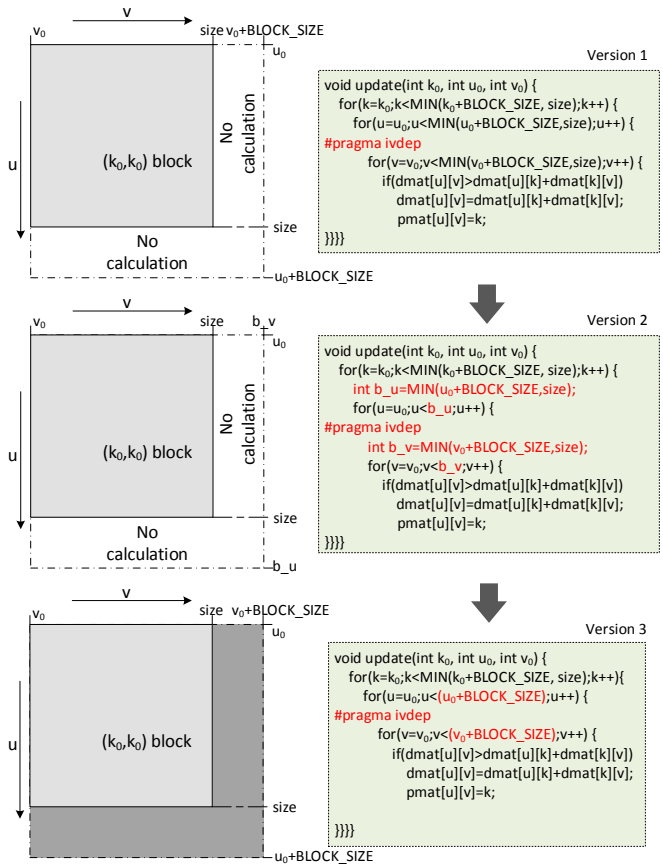

Fig. 2. Three versions of the loop structure modification. Only the loops in version 3, modified by inserting redundant calculations, are SIMD-friendly

Algorithm 3 Pseudo-code for implementing the 16-wide comparison of Floyd-Warshall algorithm

```
1:  for k = k_0 to k_0 + simd_width do
2:      path_v = avx512_set1(k)
3:      row_v = avx512_load(dist[k][v_0])
4:      for u = u_0 to u_0 + simd_width do
5:          col_v = avx512_set1(dist[u][k])
6:          sum_v = avx512_add(col_v, row_v)
7:          upd_v = avx512_load(dist[u][v_0])
8:          cmp_m = avx512_compare_mask(sum_v, upd_v, >)
9:          avx512_mask_store(dist, sum_v, cmp_m)
10:         avx512_mask_store(path, path_v, cmp_m)
11:     end for
12: end for
```

tel Xeon Phi's manycore architecture. OpenMP provides a portable way to parallelize serial programs on shared memory parallel systems with many convenient features including the run-time specification of thread number, thread affinity to cores, as well as clauses defining lists of private or shared variables [14]. In our algorithm, because of the data dependencies from the different passes through the outermost loop, i.e., line 14 of Algorithm 2, this loop is not a good candidate for OpenMP parallelization. Even for the inner loops, each computing step relies on the previous step's result, as shown in Figure 1. The loops in step 2 and 3, i.e., line 18, 22, and 26, exhibit most parallelism opportunities and dominate the overall performance. When taking all these into account, we decide to apply OpenMP pragmas on them to improve the performance.

E. Select Appropriate Configuration Settings

Our Floyd-Warshall algorithm adopts multiple optimizations with many configurable parameters, such as the block size, thread number, and runtime scheduling policy. The choice of appropriate combination of parameters is a real challenge, because the amount of possible combinations are numerous and sometimes the parameters are affected by each other. For example, the number of threads per core might affect the block size, because the threads are sharing the same physical resources. The poor configuration will cause severe performance downgrade. One possible solution to find the appropriate combination of parameters is the exhaustive study, but this is time-consuming and impractical. Especially, when we need to adjust the size of the input datasets, we have to re-calculate the best combination of parameters exhaustively.

To obtain some insights of the parameters, some statistical machine learning methods are proposed in previous research [7], [15], [16] to prune the optimization space. In this paper, we adopt a tree-based partition approach, known as Starchart [7]. The general idea can be described as below. First, the construction of this tree is based on the performance values from randomly selected samples, which have the format of $(par_1, par_2, ..., par_n, perf)$. The par_n represents the possible value of parameter n, while the $perf$ can be defined according to the optimized objective, such as the execution time or the power measurement. Then, the differences of the squared sum between the original whole set and the subsets partitioned by the possible values of parameters will be calculated. The parameter which creates the maximum gap in current level of partitions will be selected and then the execution will be moved on to the partitions of the two subsets of next level. This method is based on the application design parameters rather than performance counter measurements to search for the best configuration settings. The generated view of the partition tree can be used to provide insights of the significance of each parameter and even their relationships. Since the tool Starchart provides the detailed examples to accomplish the parameter selection, we use this tool and choose five parameters to implement the parameter selection, which are listed in the Table I below. The pool of input values consists of 480 samples generated from our optimized version with various combinations of the five parameters. We follow the guide of the paper [7] and randomly select 200 samples to build the partitioning tree step by step shown in Figure 3.

In the Figure 3 below, we can see that the algorithm exhibits different behaviors given the two scales of input number of vertices. In both cases, the choice of appropriate block size and thread number is most significant, which means these two

TABLE I
PARAMETER OVERVIEW

Parameter Name	Values	Description
Data Size	2000,4000	number of vertices (small, large)
Block Size	16,32,48,64	block dimension (multiple of SIMD width)
Task Allocation	blk,cyc1,cyc2, cyc3,cyc4	block or cyclic (various chunk sizes) scheduling
Thread Number	61,122,183,244	OpenMP thread number
Thread Affinity	balanced,scatter, compact	thread binding to each core

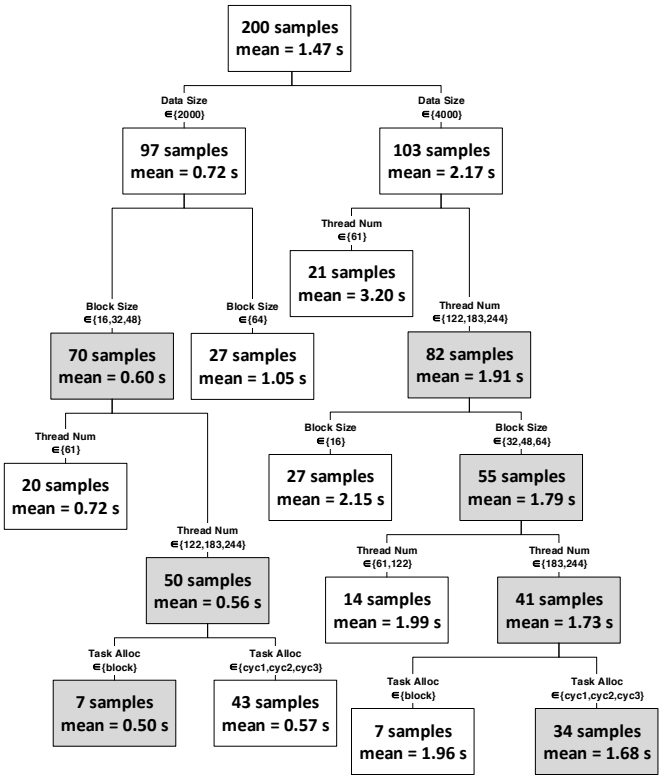

Fig. 3. The tree-based partitioning view of the compiler and runtime parameters of the Floyd-Warshall algorithm on Intel Xeon Phi

factors affect the performance most and are prior parameters to be considered. As a contrast, the other parameters, such as thread affinity, can be "ignored" if we don't have a suitable block size. The poor choice of the significant parameters may hide the effects caused by the less significant parameters. Aggregating the results from the view of Figure 3, we select the block size of 32, thread number of 244, OpenMP allocation method *block* for number of vertices le 2000 and *cyclic* for number of vertices $>$ 2000, and OpenMP thread affinity *balanced*. In the experimental section of strong scaling, we will also show the results of the effects by using different types of thread affinity. Note that although 240 threads are usually adopted because one core takes charge of the micro-OS on Xeon Phi, we still observe better performance achieved by using 244 threads.

IV. PERFORMANCE ANALYSIS

We carry out our experimental evaluations on a compute node with Intel Xeon Phi coprocessor. This compute node consists of two Intel Xeon processor E5-2670 running at 2.60 GHz along with 64 GB DDR3 SDRAM, and one Intel Xeon Phi Knight Corner coprocessor. The detailed hardware configuration is presented in Table II. The operating system on the host is the 64-bit CentOS distribution with the 2.6.32-279 Linux kernel. The Linux micro OS in the coprocessor is Intel MIC Platform Software Stack release 2.1 on the 2.6.38.8 Linux kernel. The compiler we used is the *icc* from the Intel Composer XE 2013 [17]. We use the flag *-mmic* and default optimization *-O2* to compile the codes for the *native* mode. We use OpenMP 3.1 [14] to parallelize the outer loops and "*pragma ivdep*" in the inner loops as described in the previous section. In our evaluations, we use the graph generator GTgraph [18] to create input datasets of vertices. This tool allows users to specify the number of vertices and edges.

TABLE II
TESTING PLATFORMS

	Intel CPU	Intel Xeon Phi
Code Name	Sandy Bridge	Knight Corner
Cores	8 ×2	61
Clock Frequency	2.60 GHz	1.238 GHz
Hardware Threads	2	4
SIMD Width	256-bit	512-bit
L1/L2/L3 Cache (KB)	32/256/20480	32/512/-
Memory Type	DDR3	GDDR5
Memory Size (GB)	8×8	16
Stream Bandwidth	78 GB/s	150 GB/s

A. Performance Improvements

In this section, we first choose one dataset consisting of 2000 vertices to illustrate the performance improvements brought by each optimization, including the data blocking, the directive-based SIMDization, the loop reconstruction, and the OpenMP-based parallelism. Then, we compare the default Floyd-Warshall algorithm parallelized by OpenMP with our optimized algorithm over various scales of input data sets. Moreover, we show the performance of the same optimized codes on CPU to demonstrate the programming portability between CPU and MIC.

1) Step-by-step Performance Improvement: Figure 4 shows the performance benefits after applying different levels of optimizations presented in Section III. Counter-intuitively, the blocked modification of the algorithm downgrades the performance by 14%. There are two major reasons. First, the blocked version brings redundant computations in the step 2 and the step 3. For example, the blocks (i, k) and (k, j) are recomputed in the step 3, even though they have been updated in the step 2. Second, the loop structure hinders the compiler to generate efficient code. In this algorithm, the performance degradation most comes from the second reason. Therefore, after applying the loop reconstruction by removing the MIN operations of two innermost loops, we

can successfully achieve 1.76-fold speedup over the default version.

By using the SIMD directives, we can reach another 4.1-fold speedup over the blocked version from 102.1s to 24.9s. Actually, we also tried to apply SIMD directives (*pragma ivdep*) on the blocked version without loop reconstruction. Unfortunately, it turns out that the loops inside the last two UPDATE function, i.e., line 23 and 28 in Algorithm 2, cannot be correctly auto-vectorized. After removing the MIN operations, all the innermost loops are successfully vectorized. Note that even we substitute the MIN operations with corresponding variables whose values are given by the MIN operations before the loops, as shown in the version 2 of Figure 2, we cannot vectorize the loops neither. After checking the assembly codes, we believe that the MIN operations in the nested loops (k, i, k) and (k, i, j) prevent the compiler from analyzing the correct data dependencies and generating efficient codes. In the future, we need to design more experiments to investigate how the compiler optimizations are hindered by such operations.

Considering the width of vector instructions on Intel Xeon Phi, i.e., 512 bits (16), the speedup we achieved is about one forth of the theoretical speedup. The gap comes from two major factors. The first is that not all the portion of the code can be vectorized, leading to the suboptimal speedup. The second is due to the memory-bound nature of the Floyd-Warshall algorithm. Our implementation of the algorithm performs 2 float operations on three floats including $dist[u][k]$, $dist[k][v]$, $dist[u][v]$ that means 3×4=12 bytes of data, and thus generates 0.17 (ops/byte) of memory bandwidth requirements. In contrast, the Intel Xeon Phi coprocessor can provide 14.32 (ops/byte) (in Section I), indicating the hardware computing resources are under-utilized in this application since we cannot be fast enough to feed the computing resources with sufficient data.

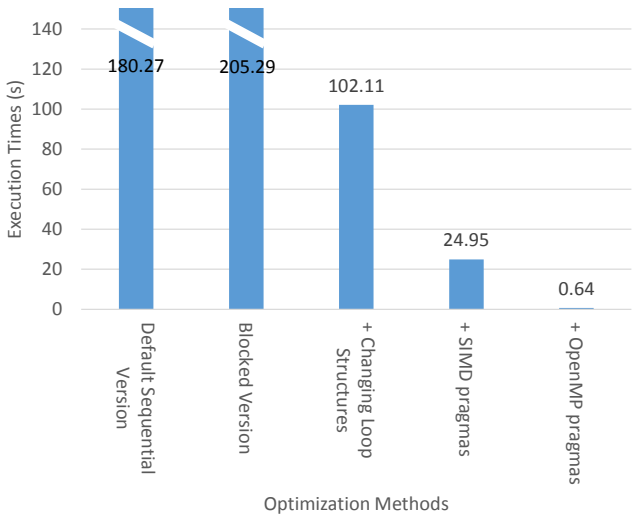

Fig. 4. The benefits of different optimization methods on the Floyd-Warshall algorithm (using 2,000 vertices)

We examine the benefits of OpenMP in accelerating computation by exploiting the abundant thread-level parallelism. We apply the OpenMP pragmas over the step 2 and step 3 (line 18, 22, and 26 in Algorithm 2). The runtime parameters include 244 threads and the *balanced* thread affinity. The results show that our optimized implementation of the Floyd-Warshall algorithm benefits most from the thread-level parallelism, leading to another 40-fold speedup. The speedup is relevant with the data block and the thread binding. In our implementation, the working sets of the distance and path matrix are rearranged block by block so as to match the requirement of SIMD operations and data reuse in the cache. Each block having 32×32 floating-point elements will take up 4 kB memory. As the computing steps move on in each iteration, more blocks will be loaded into the cache, i.e., 4 kB, 8 kB, 12 kB for the three phases respectively. Because the possibility of sharing data between neighboring threads is relatively high, it is more possible to reuse the data in the L1 cache loaded by the adjacent threads running in the same core with the *balanced* thread binding. For example, in the third computing step, four threads working on the same row can share the block (i, k), thereby leading to 36 kB occupied space (4 blocks (k, j), 4 blocks (i, j), and 1 shared block (i, k)) rather than 48 kB. Considering the 32 kB size of the L1 cache for each core, running the Floyd-Warshall algorithm using 4 threads per core is preferable, which also explains the results shown in Section III-E.

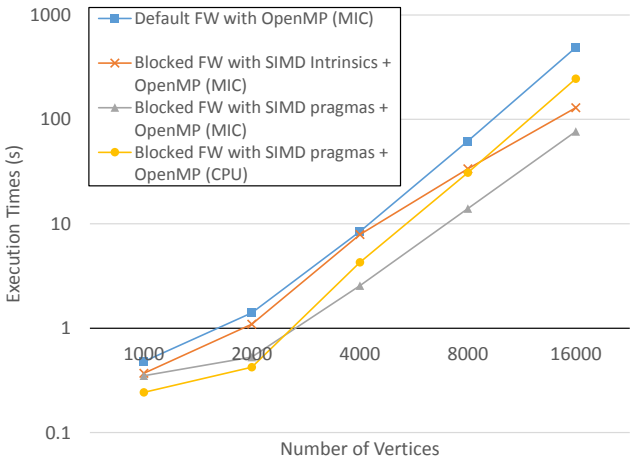

Fig. 5. OpenMP of three different version of Floyd-Warshall algorithms

In Figure 5, we show the performance of three versions of Floyd-Warshall algorithm with OpenMP enabled over the growing data sets from 1,000 to 16,000 vertices. In addition, considering the compatibility of optimizations on Intel Xeon Phi and x86 CPUs, we also compare the performance of the totally same code on CPU and on Xeon Phi. The first version "Default FW with OpenMP (MIC)"is the default algorithm (shown in Algorithm 1) using OpenMP pragmas on line 4. Because OpenMP is a general method to accelerate programs in multicore/manycore systems, we treat this version

as the baseline. The figure presents that our optimized version "Blocked FW with SIMD pragmas + OpenMP (MIC)" can achieve 1.37-fold to 6.39-fold speedup over the baseline. We also compare the performance of the manual optimization using SIMD intrinsics denoted as "Blocked FW with SIMD Intrinsics + OpenMP (MIC)". This version has 1.2-fold to 3.7-fold speedup over the baseline but slower than the compiler directive version "Blocked FW with SIMD pragmas + OpenMP (MIC) ". After comparing the assembly codes generated by these two versions, we found that the compiler can generate more efficient prefetching instructions and conduct better loop unrolling than the manual optimization we implemented. This provides a positive evidence of using "simple" techniques to fill the Ninja gap [3], which is defined as the performance gap between the code generated by the expert programmers (prefer to re-design and re-write the programs, sometimes even use assembly codes) and that generated by the general programmers (prefer to use "simple" techniques, such as pragmas of OpenMP). Compared with the performance on CPU, our optimized implementation can achieve up to 3.2-fold speedup without any code modifications. This illustrates the portability benefit of using Intel Xeon Phi as the accelerator for existing and optimized applications on multicore CPU.

2) Scalability: In this section, we evaluate the strong scaling of our optimized Floyd-Warshall algorithm over an increasing number of threads with different thread affinity types, i.e. *balanced*, *scatter*, and *compact*, on the 61-core Intel Xeon Phi. All of the experiments are conducted on the graph with 16,000 vertices.

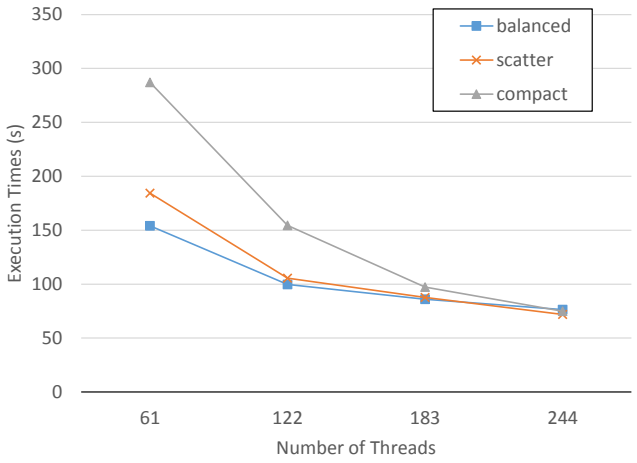

Fig. 6. Strong scaling of our optimized Floyd-Warshall algorithm with different thread affinity types, i.e., *balanced*, *scatter*, and *compact* (using 16,000 vertices)

Figure 6 illustrates the strong scaling of our optimized Floyd-Warshall algorithm. We change the thread number from 61 threads to 4×61 threads where 61 is the number of physical cores on the coprocessor. The results indicate that at the beginning using 61 threads with the *balanced* thread binding is preferable and with the growth of thread number, the application can obtain up to 2-fold, 2.6-fold, and 3.8-fold speedup using *balanced*, *scatter*, and *compact*, respectively. Therefore, our optimized Floyd-Warshall algorithm exhibits good strong scaling. As we mentioned in Section IV-A1, because of the low ratio of computation to memory access, increasing the thread number in each core by the hyper threading technique, which is designed to hide the data access latency from the memory, can benefit the performance obviously. Our experiment provides the insight that is for the memory bound applications, here the Floyd-Warshall algorithm, set all threads is an effective method to achieve better performance.

V. Related Work

Data-intensive applications face some significant challenges on modern parallel architectures, such as data-driven computations, irregular data access, and high data load to computation ratio [10]. There are many studies proposed to resolve these problems for the graph algorithms on modern CPU and GPU. Li et al. [19] devise an adaptive runtime system to switch between different GPU implementations based on the topology of the input data and runtime parameters to handle the irregularity of the graphs. Merrill et al. [20] introduce a scalable BFS implementation on GPUs to resolve the imbalance problem and achieve significant performance for various kinds of graphs. Chhugani et al. [21] present another efficient and scalable BFS on multi-socket, multi-core CPU inside the single-node. For Floyd-Warshall algorithm, several researchers have tried to take advantage of modern parallel architecture to improve the performance. Katz et al. [22] use the shared memory in GPU to explore the efficiency of Floyd-Warshall on large-scaled graphs. Matsumoto et al. [23] present a blocked Floyd-Warshall implementation for the hybrid CPU-GPU system and focus on reducing the communication overhead between the host and device. Bluluc et al. [24] use the Floyd-Warshall as a case study for this genre of algorithms, including the LU decomposition and transitive closure, to show the potential performance benefits that can be obtained by using GPU.

One of the salient trend of the parallelism of processor design is the growing SIMD width of vector units. In some applications, the untapped hardware potential prevents the realizable performance of the parallel architecture from being effectively exploited. There is a large body of related work on using manual or automatic vectorization to obtain performance gains and competitive advantages. Chhugani et al. [25] propose an efficient implementation of MergeSort using 128-bit SSE on CPU, which requires programmers to explicitly write SIMD intrinsics. Park et al. [5] extend this type of research on the communication avoid FFT algorithm [26] to Intel Xeon Phi coprocessor. They use the cross-lane instructions `load_unpack` and `store_pack` to reduce the memory accesses required in the transposition. Satish et al. [3] summarize several compiler auto-vectorization techniques and SIMD-friendly algorithms. In that paper they also provide a modified version of MergeSort, where the loops can be auto-vectorized. Generally, the algorithmic changes to take advantages of the SIMD instruction set will cause some indispensable overhead

by increasing the total computational operations. McFarlin et al. [27] conduct a quantitative analysis of vectorization efficiency and present one peephole-based automatic SIMD vectorization system.

With the advent of Intel Xeon Phi co-processors, many applications from different domains are optimized on this architecture. Heinecke et al. [4] optimize the Linpack benchmark on Intel Xeon Phi using both of the *native* and hybrid implementation based on a dynamic scheduling scheme. Wende et al. [6] extend the Swendsen-Wang multi-cluster algorithms on Xeon Phi and GPU for the simulation of the two- and three-dimensional Ising model. In their work, they use the Xeon Phi's SIMD intrinsics to optimize the cluster self-labeling method. Liu et al. [28] use a specialized data structure to implement an efficient sparse matrix-vector multiplication on Xeon Phi. For our work in this paper, we focus on using the "simple" method, including the algorithmic adjustments and appropriate compiler directives, to tune a graph processing application and illustrate the parallel programmability on Intel Xeon Phi.

VI. Conclusion

Intel Xeon Phi architecture provides a promising solution for parallel applications on accelerators, since the program for multicore CPUs can be running directly on it without any modification. However, if the intra- and inter-core parallelism are not utilized appropriately, one can only see a little benefits of such parallel architecture. Besides, the performance of data-intensive applications on Xeon Phi coprocessor is constrained by the memory access bandwidth. Although using AVX-512 registers and SIMD intrinsics of Xeon Phi architecture is a solution for the bandwidth constraint, the programming complexity hinders it to be accepted by most of the application developers. In this paper, we use the Floyd-Warshall algorithm as the case study of data-intensive applications to show the step-by-step performance when porting the default program on Intel Xeon Phi coprocessor. We use the algorithmic adjustments, including the data blocking and the loop reconstruction, and appropriate compiler directives, to achieve significant performance gains. With our optimizations, we observed the 281.7-fold speedup over the default serial implementation. Compared with the default OpenMP implementation, we can achieve up to 6.4-fold speedup on Intel Xeon Phi. In addition, we also observe that the exactly same optimized code on Intel Xeon Phi can get up to 3.2-fold speedup over that on Intel Xeon multicore processor.

In our experiments, we have observed that the manual optimization using the SIMD intrinsics doesn't obtain comparable performance gains as the code optimized by the auto-vectorization of the compiler. There still exist spaces to optimize performance, such as better prefetching and loop unrolling. We also plan to extend our work on other classes of graph processing applications. For example, BFS with the data-driven computation pattern and the poor data locality, may have many challenges while being applied on Intel Xeon Phi. As a result, the road map of our future work has two branches: first is to use the expertise knowledge to optimize the graph applications, and try to generalize the common methods or primitives for the same genre of applications; second is to use the "simple" method using the algorithmic adjustments with the traditional optimizations to achieve good performance. Furthermore, we plan to extend our research on the large scale systems with Intel Xeon Phi coprocessor to match the requirement of data processing in big data era.

References

[1] Top500.Org, "Top 500 Supercomputer Sites," 2014. [Online]. Available: http://www.top500.org

[2] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," University of Virginia, Tech. Rep., 1991-2007, a continually updated technical report. [Online]. Available: http://www.cs.virginia.edu/stream/

[3] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?" in *Proceedings of the 39th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[4] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. Shet, G. Chrysos, and P. Dubey, "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor," in *Proceedings of the 27th IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2013.

[5] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim, "Tera-scale 1D FFT with Low-communication Algorithm and Intel Xeon Phi Coprocessors," in *Proceedings of the 25th IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[6] F. Wende and T. Steinke, "Swendsen-Wang Multi-cluster Algorithm for the 2D/3D Ising Model on Xeon Phi and GPU," in *Proceedings of the 25th IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[7] W. Jia, K. A. Shaw, and M. Martonosi, "Starchart: Hardware and Software Optimization Using Recursive Partitioning Regression Trees," in *Proceedings of the 22nd ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[8] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM (CACM)*, vol. 5, no. 6, ACM, 1962.

[9] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM (JACM)*, vol. 9, no. 1, ACM, 1962.

[10] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in Parallel Graph Processing," *Journal of Parallel Processing Letters*, vol. 17, no. 1, World Scientific, 2007.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.

[12] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A Blocked All-Pairs Shortest-Paths Algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 8, ACM, 2003.

[13] Intel Cooperation, "A Guide to Vectorization with Intel C++ Compilers," 2012. [Online]. Available: http://download-software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf

[14] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1," 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[15] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A Case for Machine Learning to Optimize Multicore Performance," in *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2009.

[16] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and Use of Linear Regression Models for Processor Performance Analysis," in *Proceedings of the 12th IEEE International Symposium on High-Performance Computer Architecture (HPCA).*, 2006.

[17] Intel Cooperation, "Intel Composer," 2013. [Online]. Available: http://software.intel.com/en-us/intel-composer-xej

[18] D. A. Bader and K. Madduri, "GTgraph: A Synthetic Graph Generator Suite," 2006. [Online]. Available: http://www.cse.psu.edu/~madduri/software/GTgraph

[19] D. Li and M. Becchi, "Deploying Graph Algorithms on GPUs: An Adaptive Solution," in *Proceedings of the 27th IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2013.

[20] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[21] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency," in *Proceedings of the 26th IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2012.

[22] G. J. Katz and J. T. Kider, Jr, "All-pairs Shortest-paths for Large Graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH)*, 2008.

[23] K. Matsumoto, N. Nakasato, and S. Sedukhin, "Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System," in *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2011.

[24] A. Buluç, J. R. Gilbert, and C. Budak, "Solving Path Problems on the GPU," *Journal of Parallel Computing*, vol. 36, no. 5-6, Elsevier Science Publishers B. V., 2010.

[25] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 2, 2008.

[26] P. T. P. Tang, J. Park, D. Kim, and V. Petrov, "A Framework for Low-communication 1-D FFT," in *Proceedings of the 24th IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[27] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets," in *Proceedings of the 25th ACM International Conference on Supercomputing (ICS)*, 2011.

[28] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors," in *Proceedings of the 27th ACM International Conference on Supercomputing (ICS)*, 2013.