A Survey on Evaluating and Optimizing Performance of Intel Xeon Phi

Sparsh Mittal

Abstract

Intel's Xeon Phi combines the parallel processing power of a many-core accelerator with the programming ease of CPUs. In this paper, we present a survey of works that study the architecture of Phi and use it as an accelerator for a broad range of applications. We review performance optimization strategies as well as the factors that bottleneck the performance of Phi. We also review works that perform comparison or collaborative execution of Phi with CPUs and GPUs. This paper will be useful for researchers and developers in the area of computer-architecture and high-performance computing.

Index Terms

Review, many-core processor, many-integrated core (MIC), vectorization, prefetching, compiler optimization.

1 Introduction

As power budget and clock frequency of modern processors reach a plateau, parallelization has become *the* way to continue to scale performance. This has motivated the researchers to design multi-core and even, many-core processing units. Specifically, Intel's Xeon Phi [1] brings together the parallel processing power of a many-core computing unit and the programming ease of traditional CPUs [2]. In fact, in June 2018 list of Top500 supercomputers, 19 supercomputers used Phi as the main processing unit [3] and seven supercomputers used Phi as the co-processor. Also, researchers from a wide range of background have deployed Phi for accelerating compute-intensive tasks and have also compared it with other processing units, such as multi-core CPUs and GPUs.

Recently, KNC and KNL Phis have been discontinued. As of October 2019, KNM Phi is still shipping, but Intel has no roadmap for Phi. Further, all the salient features/optimizations of Xeon Phi have been incorporated into recent CPUs. As such, it is the right time to *look back* on the performance 'score-card' of Phi and also reflect on the salient features and limitations of Phi. These insights and lessons will be useful for designers of next-generation computing systems.

Contributions: In this paper, we present a survey of works that evaluate and optimize efficiency of Phi for a broad-range of applications. Figure 1 shows the outline of the paper. We first present a background on Phi design and management policies (Section 2) and then discuss the strengths and limitations of Phi (Section 3). We then review the works that study Phi architecture and optimization techniques (Section 4). Next, we review the techniques for achieving data, thread and node-level parallelization (Section 5). Further, we discuss the works in terms of their application domain (Section 6). We then discuss works that perform comparative evaluation and collaborative execution of Phi with CPUs and/or GPUs (Section 7). Finally, we conclude the paper with a mention of future outlook (Section 8). The insights provided by this survey will be valuable for further optimizing performance of Phi and even other processing units.

- *Dr. Sparsh is with IIT Hyderabad, Kandi, Sangareddy 502285, Telangana, India. E-mail:sparsh@iith.ac.in. Support for this work was provided by Science and Engineering Research Board (SERB), India, award number ECR/2017/000622 and by Semiconductor Research Corporation. This paper has been accepted in Concurrency and Computation: Practice and Experience, 2020.*

This paper is expected to be useful for researchers, computer-architects, chip-designers, programmers and users of HPC [1] facilities.

Paper organization

Fig. 1. Organization of the paper

2 Background

In this section, we provide background on Phi and refer the reader to previous works for more details [1, 4–13]. We first discuss different product generations of Phi and their specifications (Section 2.1) and then review the microarchitecture of KNC, KNL and KNM Phis (Section 2.2). After this, we review the operational-modes of Phi and its thread-affinity strategies (Section 2.3-2.4). Finally, we discuss the optimization strategies that have been used on Phi (Section 2.5).

2.1 Phi product generations

Intel's many-core architecture is termed as "many integrated core" (MIC) whereas the commercial product based on MIC architecture is termed as Xeon Phi. Table 1 shows the codenames of different generations of Phi, their generic series numbering along with some examples. Table 1 also highlights the works that use different models of Phi. Table 2 shows the architectural parameters of selected models of CPU, GPU and Phi.

TABLE 1
Codenames and products of different generations of Phi [1]

Code name	Series	Examples
Knights Ferry		(prototype only)
KNC	x100	3110X, 3115A [14], 3120 [15], 3120A [16], 3120P [17–19], 31SP [20, 21], 31S1P [22], 5100 [15], 5110P [2, 4, 6, 7, 14, 23–33], 5120D [34], 5120P, 5510P [35], SE10P [13, 36–38], SE10X [39], 7110P [40], 7110X [25] , 7120A [41], 7120D, 7120P [14, 31, 38, 42–49], 7120X.
KNL	x200	7210P [50], 7210F [51], 7210 [12, 52–57], 7230 [11], 7230F [56, 58], 7250 [17, 34, 38, 59–66], 7250F[47], 7290 [52], 7290F, 7290B [67]
Knights Hill		canceled
Knights Mill	x205	7235, 7285, 7295 [51, 65, 68, 69]

1. The following acronyms are frequently used in this paper: advanced vector extensions (AVX), array of structures (AoS), array of structures of arrays (AoSoA), basic linear algebra subroutine (BLAS), caching/home agent (CHA), direct media interface (DMI), (Quad) fused-multiply-and-accumulate (QFMA/FMA), floating-point (FP), (Graphics) double data rate (GDDR/DDR), high-performance computing (HPC), initial many core instruction set (IMCI), instruction set architecture (ISA), Knights Corner/Ferry/Landing/Mill (KNC/KNF/KNL/KNM), math kernel library (MKL), memory controller (MC), multiple/single instruction multiple data (MIMD/SIMD), Non-uniform memory access (NUMA) peripheral component interconnect express (PCIe), single/double/half precision (SP/DP/HP), structure of arrays (SoA), sparse matrix-vector multiplication (SpMV), streaming SIMD extension (SSE), thermal design power (TDP), thread building block (TBB), translation lookaside buffer (TLB), vector processing unit (VPU)

TABLE 2
Parameters of selected models of CPU, GPU and Phi [70, 71] (Microarch. = microarchitecture, Titan V has 5120 CUDA and 640 tensor cores. *=per SM; §=per core. † = discontinued)

	CPU	GPU	Phi			
Model	i7-7820X	Titan V	3120A	5110P	7250	7295
Microarch.	Skylake	Volta	KNC	KNC	KNL	KNM
Node	14nm	12nm	22nm	22nm	14nm	14nm
Year	2017	2017	2013	2012	2016	2017
# of cores	8	5120+640	57	61	68	72
Threads	16	-	228	244	272	288
Clock(GHz)	3.6	1.2	1.1	1.05	1.40	1.50
L1 Cache	32KB§	96KB*	32KB§	32KB§	32KB§	32KB§
L2 cache	1MB	4.5MB	512KB	512KB	512KB	512KB
L3 cache	11MB	N/A	N/A	N/A	N/A	N/A
Memory	128GB	12GB	6GB	8GB	16GB	16 GB
BW(GB/s)	79.5	653	240	320	490	115
Max TDP	140W	250W	300W	225W	215W	320W
Peak SP(TFLOP)	1.84	14.9	2	2	6.1	12.2
Peak DP(TFLOP)	0.92	7.45	1	1	3.05	1.52
Price	$599	$2,999	†	†	$2,436†	$2,172

2.2 Architecture of KNC, KNL and KNM Phis

KNC Phi: Figure 2 shows the architecture of KNC Phi. KNC connects to the host system via a PCIe bus. KNC has 57 to 61 in-order cores which are connected via a bidirectional ring bus. Through the ring bus, a core can access other cores and the GDDR5 memory. The private L2 cache of every core is kept coherent by a tag directory. Each core runs up to 4 threads. Each core has a VPU, which supports Intel's 512b IMCI ISA [13]. IMCI provides in-built scatter/gather operations that perform irregular memory accesses, hardware-support for "`mask` data type", and "`write-mask` operations" using which an operation can be performed on specific elements of a single SIMD register. Compared to achieving the same functionality using SSE, use of IMCI allows implementing irregular parallelism in an easier manner.

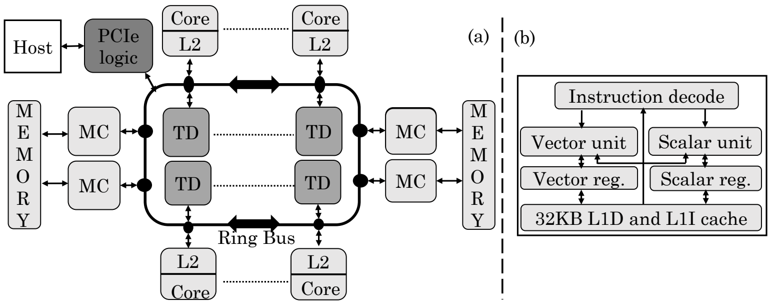


Fig. 2. (a) Architecture of KNC Phi (TD = tag directory) and (b) each core [1, 45]

KNL Phi: Figure 3 shows the architecture of KNL Phi. KNL comes in three product types: "self-boot processor", "self-boot with integrated fabric", and a co-processor connected over PCIe. KNL has up to 72 out-of-order cores with up to four threads per core. Cores are organized in tiles. Every tile has two cores which share a 1 MB L2 cache. CHA works as distributed tag directory for keeping L2 caches in different tiles coherent. CHA is the connecting point of every tile.

KNL has a "near" and a "far" memory [72]. The "near" memory is composed of eight 2GB MCDRAM modules based on "hybrid memory cube" which provides high bandwidth (400-500 GB/s). The "far" memory has six memory channels with up to 64GB DDR4 memory per channel (total of 384GB) and provides bandwidth up to 90 GB/s. The near memory can be configured in one of the following modes,

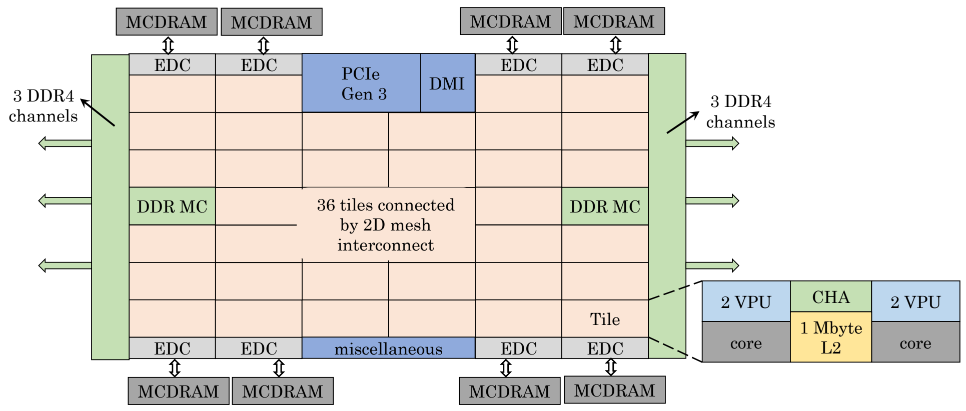


Fig. 3. Architecture of KNL Phi (EDC=MCDRAM controller, DDR MC = DDR memory controller) [10]

which are shown in Figure 4. (1) Flat: both memories are in a single address space but appear as different NUMA nodes. (2) Cache: MCDRAM is used as a "direct-mapped cache" for DRAM. (3) Hybrid: A part of MCDRAM is used in cache mode (4 or 8GB) and the remaining part in flat mode (12 or 8GB).

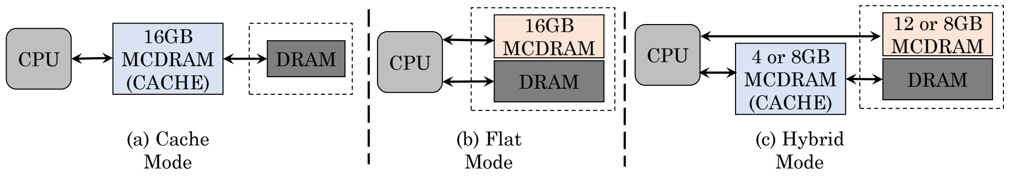


Fig. 4. Memory modes of KNL [10]

KNL uses "mesh on-die interconnect architecture" which is based on the ring architecture. It supports three clustering modes which provide different levels of address affinity for lowering the distance that requests/data travel on the chip. (1) In "all-to-all mode", there is no affinity between the tile, directory, and memory. However, due to this, the performance of this mode is lower than that of other modes. (2) "Quadrant mode" logically divides the chip into four quadrants, each of which provides affinity between the memory and the directory. A tile has no affinity with the directory or the memory and thus, a request from a tile can end in any directory, which will access the memory in its own quadrant only. Hence, the latency in this mode is lower than that in "all-to-all" mode. To use this mode, the total capacity on both DDR memory controllers should be same. (3) "Sub-NUMA clustering" (SNC) mode is an extension of the quadrant mode where the tile has affinity with both the directory and the memory. The chip is logically partitioned into two or four NUMA clusters and these variants are called SNC-2 and SNC-4, respectively. A request from a tile accesses a directory in its cluster, which accesses the memory controllers within that cluster. By virtue of containing majority of traffic within a cluster, this mode leads to least latency among all the modes. To use this memory, software should allocate memory in the same NUMA cluster where it is executed.

KNM Phi: KNM chip is identical to KNL in number of cores, frequency, PCIe 3.0 lanes, use of 16GB MCDRAM and 6 DRAM channels. However, KNM allows utilizing AVX-512 units more effectively by supporting new AVX-512 instructions [73]. In KNL, every VPU offers identical 512b interfaces to the AVX-512 unit, which supports SP and DP computations. Half-precision computations are performed by the SP block. This is shown in left-part of Figure 5. By comparison, a VPU in KNM has asymmetrical ports: instead of SP/DP block, it has separate DP and SP/VNNI blocks (VNNI = vector neural network instruction) and one DP block has been removed, as shown in right part of Figure 5. Hence, compared to KNL, KNM has 0.5× DP performance but 2× SP performance and 4× VNNI performance. KNM has been designed especially for machine learning applications since they can work with low-precision data-values [74, 75].

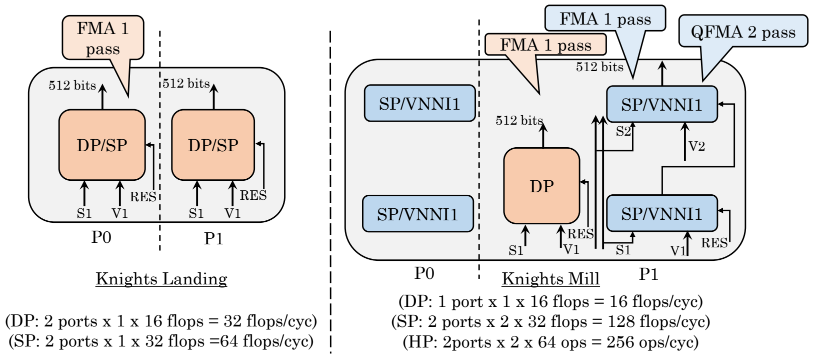

FMA 1 pass
512 bits
DP/SP
S1
V1
RES
P0
P1
Knights Landing
(DP: 2 ports x 1 x 16 flops = 32 flops/cyc)
(SP: 2 ports x 1 x 32 flops =64 flops/cyc)
FMA 1 pass
QFMA 2 pass
SP/VNNI1
DP
S2
V2
Knights Mill
(DP: 1 port x 1 x 16 flops = 16 flops/cyc)
(SP: 2 ports x 2 x 32 flops = 128 flops/cyc)
(HP: 2ports x 2 x 64 ops = 256 ops/cyc)

Fig. 5. Comparison of port connections in KNL and KNM [73]

2.3 Modes of operation

A system with Phi may be operated in one of the three modes [76], which are illustrated in Figure 6(a). The native mode requires minimum porting effort and data-transfer, but it requires the application to be cross-compiled for the OS of Phi. In the offload mode, data-transfer over PCIe bus may become a bottleneck. In the symmetric mode, Phi is treated as one of the nodes of a heterogeneous cluster. Load-balancing schemes are required to efficiently use both Phi and CPU.

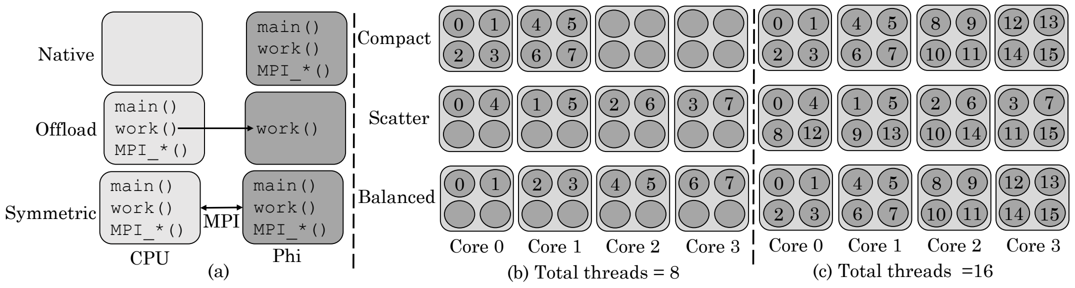


Fig. 6. (a) Modes of operation [77]. Thread-to-core assignment with different thread-affinity strategies when mapping (b) 8 threads and (c) 16 threads to 4 cores [78]

2.4 Thread-affinity strategies

On Phi, there are three predefined strategies for assigning threads to a core for obtaining improved or predictable performance [13, 23, 37]. (1) The “compact” strategy uses all four threads of a core before it begins using the threads of subsequent cores. (2) The “scatter” strategy allocates the threads as uniformly as possible over the whole processor such that consecutive threads are executed in different cores. (3) The “balanced” strategy maps threads on different cores until all the cores have at least one thread, as done in the “scatter” strategy. However, when multiple threads need to use the same core, the “balanced” strategy ensures that consecutive thread-IDs are close to each other, which is not done by the “scatter” policy. Figure 6(b)-(c) illustrates these strategies with two examples: (1) when 8 threads are assigned to 4 cores and (2) when 16 threads are assigned to 4 cores. It is noteworthy that on using maximum number of threads (=16 for 4 cores), compact strategy is equivalent to balanced strategy.

2.5 Optimization strategies used on Phi

Loop collapsing and collapsing: Loop collapsing combines multiple levels of a loop nest which increases the number of iterations in the `for` loop. This reduces loop overhead and allows parallelization over higher number of threads. Loop splitting creates multiple loops from a single loop, each of which can be separately vectorized. Figures 7(a)-(b) show an example of loop-splitting.

Gather/scatter operations: The gather operation loads values from non-contiguous memory positions into contiguous positions, whereas the scatter operation stores values of a contiguous array into non-contiguous memory positions [13]. These operations are useful for programs that use indirect addressing or access memory with a non-unit stride.

```
for i= 0 to MAX-1 {
 for j =0 to HIGH-1 {
  Val[i][j] = Get(i,j)
 }
 NewState(Val[i],i)
}
```

(a) Original code

```
for i= 0 to MAX-1 {
 for j =0 to HIGH-1 {
  Val[i][j] = Get(i,j)
}}
for i= 0 to MAX-1 {
 NewState(Val[i],i)
}
```

(b) Code after loop-splitting

```
int *p1, *p2;
...

for i = 0 to N-1
 p1[i] = p2[i]*C;
```

(c) Original code

```
int * restrict p1,
* restrict p2;
...
#pragma ivdep
for i = 0 to N-1
 p1[i] = p2[i]*C;
```

(d) Code after dependency disambiguation

Fig. 7. Examples of (a)-(b) loop-splitting and (c)-(d) dependency disambiguation

Data-alignment and padding: Data-alignment ensures that memory allocated to a structure begins at an address that is also the beginning of a cache line (64B on Phi). This avoids fetching of multiple cache lines for execution of a single SIMD instruction. However, in many applications, satisfying data-alignment is difficult. For example, in stencil computations, if address of Array[i] is aligned to 64B, addresses of its neighbors viz., Array[i-1] and Array[i+1] may not be aligned. Further, since nearby SIMD lanes can only access contiguous memory addresses, memory accesses arising from AoS or indirection arrays cannot be directly vectorized [79]. Padding alleviates "false sharing" between threads working on the same cacheline.

Programmer-assisted dependency disambiguation: To be conservative, a compiler does not vectorize a code if it assumes presence of data-dependencies. Figure 7(c) shows a code where two pointers are dereferenced in the same loop. Since the compiler may not be able to ascertain whether the memory regions pointed to by them are entirely distinct, it cannot vectorize the code. However, based on her knowledge of the code, a user can instruct the compiler to ignore any incorrectly assumed dependencies and proceed with vectorization. This can be achieved using the directive `#pragma ivdep`, as shown in Figure 7(d). Use of `restrict` keyword is an assurance to compiler that over the scope of the pointer, the data pointed by the pointer will be accessed only through that pointer and the pointers copied from it.

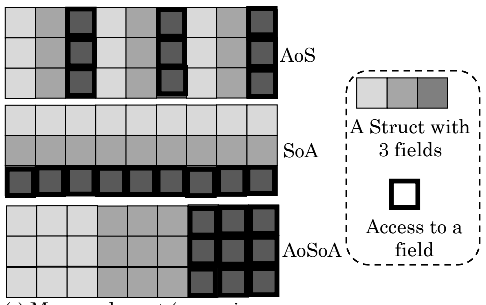
"Array-of-structure" (AoS) and "structure-of-array" (SoA) layout: Figures 8(a) and (b) show source-codes which use AoS and SoA styles, respectively and Figure 8(c) shows the corresponding layouts. Clearly, on using AoS layout, the fields of a record (struct) are stored contiguously and multiple records are stored contiguously. By comparison, in SoA layout, the same fields of all the records are stored contiguously. On accessing the same field of different records, the AoS layout shows poor performance since it gathers non-contiguous data which may lie in different cache-lines. This degrades cache and bandwidth efficiency and reduces the efficacy of vectorization. By comparison, with the SoA layout, the memory accesses can be satisfied from the contiguous regions of memory which facilitates vectorization and improves performance. The AoSoA layout, shown in Figure 8(c), combines the benefits of AoS and SoA layouts by storing same fields of VW records contiguously, where VW is a multiple of vector width.

```
struct Record{
  string name, age, ID;};
Record Data[N];
for(i=0; i<N; i++)
  Process(Data[i].ID);
```

(a) Code using AoS

```
struct DataType{
string NameData[N],
AgeData[N], IDData[N];};
DataType myData;
for(i=0; i<N; i++)
Process(myData.IDData[i]);
```

(b) Code using SoA



(c) Memory layout (assuming row-major order) and access pattern

Fig. 8. Illustration of code using (a) AoS and (b) SoA styles. (c) Memory layout and access pattern of AoS, SoA and AoSoA styles.

Prefetching: On Phi, SW prefetching prefetches all regular access patterns that the compiler is able to analyze [41]. As for the hardware prefetcher, the L2 cache has an unsophisticated streaming prefetcher that only detects strides at most two cachelines apart [7, 80]. "Intrinsic prefetches" are software prefetch instructions inserted by the programmer herself and not by the compiler. Also, the programmer can use pragmas to give hints about the loops. The software prefetching is turned on by default at and above optimization levels `-O2`. However, the challenge in this is to determine the variables to be prefetched and inserting prefetch instructions at the right place to ensure that the prefetched data arrives in timely manner [80].

3 Features and challenges of Phi

We now review the features and limitations of Phi and the challenges in optimizing its performance.

3.1 Features of Phi

Ease of programming: The biggest advantage of Phi is that it is x86-compatible and can run standard programming languages and APIs (application program interfaces), such as C/C++/Fortran, MPI, Pthreads and OpenMP. Hence, compared to other platforms such as GPU, FPGA and ASIC, Phi incurs much lower overhead of programming and development. This is especially beneficial for large-scale legacy codes [33].

Reducing data-transfer overhead: Similar to other accelerators, all Phi models can be connected to the host through the PCIe bus. However, KNL and KNM Phis are different from KNC in that they can also work as standalone many-core processors. Thus, KNL and KNM can work both as an accelerator and a co-processor. Native execution of code on Phi reduces the data-transfer overhead, which is a major performance bottleneck in GPU for many applications [81].

High performance: With its hundreds of threads and wide vector units, Phi can exploit both thread- and data-level parallelism and hence, provide large performance for applications with abundant parallelism and high computation to communication ratio [29, 82]. Phi also has higher memory bandwidth than Skylake CPU (refer Table 2).

Architectural improvements over time: To reduce power consumption despite high number of cores, the initial generations of Phi (KNF and KNC) used in-order cores and low clock frequency which led to poor single-core performance. However, KNL and KNM use out-of-order execution, high-bandwidth memory and interconnect [10, 53, 73]. Hence, they achieve high single core performance and overall throughput. Also, KNL/KNM support AVX-512 and all legacy ISAs such as x87/MMX/SSE/AVX/AVX2 [83]. Hence, KNL/KNM can run nearly all applications that run on Xeon processors such as Broadwell. Similarly, unaligned instructions processing aligned data lead to performance overheads on KNC, but not on KNL/KNM.

3.2 Challenges of Phi

Challenges in parallelization: As per Amdahl's law, a parallel processor such as Phi is useful for boosting the performance of only those applications that have abundant parallelism and work on large-size problems. Small size problems do not benefit from parallel processors since the initialization-related overheads such as OpenMP thread-creation and data-transfer themselves account for a large fraction of execution time [28, 45].

Need of thread-level parallelization: Since a thread of KNC can issue a "vector instruction" only in every alternate cycle, to fully utilize the vector unit, each core of KNC need to host at least two threads, compared to only one thread in CPU [14, 84]. Since KNL/KNM perform out-of-order execution, they do not depend on running multiple threads per core for hiding memory access latency [12]. Still, in general, since a Phi core is weaker than a CPU core, to achieve comparable performance as CPU, on Phi, the application may need to be parallelized over much larger number of threads [14]. On KNC, due to factors such as thread-affinity strategy, load-balancing and data-sharing patterns, the increase in performance with rising thread-count may not be uniform [13, 37], which may not be intuitive for a designer. Overall, compared to CPU, Phi is more sensitive to optimization schemes and tuning parameters [23, 54] since due to its stronger cores, CPU can more easily tolerate imperfect vectorization and thread-scaling.

Need of vectorization: Compared to CPU, Phi has poor serial performance but wide (512-bit) vector units. Hence, on Phi, vectorization is extremely important to achieve high performance [35, 46, 84]. Although the compiler can automatically vectorize the code, significant efforts may still be required from the programmer [2]. Similarly, loops with multiple exits [43], small trip-count [54], control flow [79], indirect memory accesses [85], complex pointer-computations [37], or data-dependence [13, 23, 36, 44, 84] may be difficult to vectorize. Some works propose moving all the conditional statements to CPU to enable vectorization on Phi [22]. In fact, while node- and thread-level parallelization is easier to achieve using MPI and Pthreads/OpenMP (respectively), extracting vectorization performance is much more challenging [79].

Need of code-rewriting: On KNC, attaining high performance may require significant code-rewriting such as programming in intrinsics [7, 18, 20, 40, 52, 53, 59, 82, 84, 86, 87], although the need of intrinsics is not as stringent on KNL due to the improvements in microarchitecture, compiler and ISA compared to KNC [83]. Use of intrinsics is error-prone and increases code-development time [88]. Also, it requires in-depth understanding of both the algorithm and SIMD intrinsics. Further, since arbitrary data-movement is not feasible, multiple data-reordering functions may need to be used to arrange the data in desired order for computations [88]. Also, since the ISA and vector width supported by different processors is different, use of intrinsics may lead to non-portable code [88].

Some works propose writing individual versions of all functions for both Phi and CPU [36, 84]. For example, Rosales et al. [36] note that, by default, offloaded arrays are allocated on Phi at the beginning of offloading and deallocated at the end of offloading. Thus, unlike GPUs, the arrays do not persist. To avoid the allocation/deallocation overheads, the deallocation of arrays can be avoided after first use which allows using these arrays in the next data-transfer. This increases the speed of data-transfer, although it increases code-development time by virtue of requiring separate versions of most functions for CPU and Phi. These code-changes also harm portability since different processors from even same vendor support different SIMD instruction sets [18, 83].

Tradeoff between multithreading and caching: To hide the high memory latency of Phi, running large number of threads and improving cache efficiency through techniques such as prefetching and cache blocking are important. However, these goals are often at odds with each other. For example, while performing multithreading improves core-utilization efficiency, it also increases cache contention [89]. Similarly, Xue et al. [22] note that converting the data-layout from AoS to SoA improves vectorization but also reduces the cache hit ratio. Further, since parallel processors allocate larger fraction of their resources to compute logic than the caches, unlike recent high-end CPUs, Phi does not have a large shared last level cache. Hence, if the working set of a program does not fit in L2 cache, it has to reside in off-chip memory and not in the on-chip L3 cache as in the case of CPUs [14].

Architectural challenges: KNC Phi has coherent L2 cache with "ring interconnection". On an L2 cache miss in a core, a request is forwarded to the ring. If the missing address is present in the L2 cache of another core, the data is sent over the ring. Although this design reduces the number of L2 misses that need to be sent to off-chip memory, it increases the latency of L2 cache hit, since in the worst case, searching the missing address in caches of other cores may take hundreds of cycles. Clearly, high data-locality is required for offsetting these overheads [79].

Similarly, although MCDRAM present in KNL/KNM Phi has higher bandwidth than the DRAM, its capacity is much smaller and its latency is also high [12]. Hence, when it is used as a cache, a miss in it adds to the total latency. Even the bandwidth advantage of MCDRAM is realized only when multiple threads access MCDRAM simultaneously [12, 55]. For example, Ramos et al. [12] show that for bitonic sort (a memory-bound code), use of MCDRAM provides no benefit over use of DRAM. This happens because in bitonic sort, all the cores access memory only in the initial stage, where the size of merged arrays is small. In subsequent stages, the thread-count is reduced to half until only one thread is performing the sorting. Since the bandwidth for a single-thread is same (nearly 8 GB/s) in both the memories, MCDRAM provides no advantage over DRAM.

Lack of certain features: KNC used IMCI ISA which was incompatible with SSE/AVX/AVX2/AVX-512 extensions [2, 9]. Hence, legacy code vectorized and optimized using SSE instructions does not run on KNC [2, 9]. Similarly, the KNC binaries need to be recompiled for porting to KNL self-boot platform [83]. KNC Phi performs division operation in software since it does not have a hardware divide instruction [13]. To avoid the overhead of division operation, a programmer may need to implement it in an imprecise manner, e.g., using multiplication operations [54]. Similarly, scalar versions of mathematical functions such as `log`, `exp` and `pwr` take more than $5\times$ higher time on KNL than on Haswell [54].

4 PHI ARCHITECTURE AND OPTIMIZATION SCHEMES

We now review the works that seek to gain insights into Phi architecture (Section 4.1), study its operational-modes, thread-affinity strategies and memory/cluster modes (Sections 4.2-4.4) and implement prefetching techniques (Section 4.5). Table 3 shows the optimization techniques used by different works. In terms of optimization metric, apart from performance, researchers have also looked at energy optimization [4, 6, 18, 29, 41, 49, 90].

TABLE 3
Optimization strategies

Loop-optimizations	unrolling [9, 17, 23, 24, 29, 50, 84, 90], collapsing [4, 6, 7, 13, 20, 21, 44, 54], splitting [22, 28]
Blocking (tiling) in	cache [14, 15, 18, 20–22, 27, 39, 44, 52, 54, 69], registers [68, 69]
Compile-time optimizations	using pre-computed values [35, 52], specifying array and loop bounds at compile time [6, 54]
Compute-related optimizations	Reusing intermediate variables [22, 35], using conflict-detection instruction of AVX-512 [52, 85], performing redundant computation to avoid data-communication or atomic operations [52, 82]
Array transpose	[6, 79]
Using huge pages	[6, 23, 53]
Data layouts	SoA [20, 22, 23, 28, 30, 36, 50, 79, 82, 90], AoS [9], AoSoA [57, 90]
Data alignment	[6, 9, 14, 18, 20, 24, 44, 45, 52, 53, 66, 79, 84, 90]
Padding	[4, 7, 9, 20, 24, 44, 52, 53, 79, 82, 91]
Dependency disambiguation	[15, 28, 36, 82, 91]
Prefetching	Software [4, 7, 9, 14, 17, 22, 23, 40, 41, 50, 52, 53, 57, 66, 69, 84], hardware [7, 14, 41, 84]
Avoiding overhead of scatter/gather operations	by reordering the edges of computational graph [52, 54, 79], by using unit-stride [13, 52], by implementing convolution as direct convolution [69]
Reducing overhead of division operations	implementing division using multiplication [7, 9, 35, 40, 52, 54], rewriting algebraic expressions to reduce number of division operations [54]
Approximate computing	[32, 44, 50, 54, 68, 69]
Cache bypassing for data with no temporal locality	[12, 42, 57, 90]
Others	CNN layer fusion [68, 69], narrow data-types [18], grouping multiply and add to use FMA instruction [52], reducing OpenMP fork/join operations by performing multiple FFT computations in each library call [54], handling writes using software-managed buffer [23], only one (and not all) core or thread handles the communication [28, 66]

4.1 Gaining insights into Phi architecture

Fang et al. [84] microbenchmark KNC Phi to reveal its architecture. They find that the latency of different vector instructions are between 2 to 6 cycles and the latency of L1 cache, L2 cache and main memory are 3, 24 and 306 cycles, respectively. The latency of accessing remote cache is 250 cycles, irrespective of the coherency state of the cacheline. Although the peak memory bandwidth is 370 GB/s, the peak read bandwidth is 164 GB/s on using at least one thread per core, and the peak write bandwidth is 76 GB/s only when 240 threads are used. Phi uses error-correcting code which increases the memory bandwidth by around 25%. Different thread-affinity schemes achieve similar bandwidths which indicates that the cores are placed symmetrically over the "bi-directional ring". Finally, they optimize leukocyte tracking on Phi and observe that in decreasing order of performance, processors are: GPU, Phi and CPU. The high performance on GPU is due to efficient reduction in shared memory.

Latu et al. [14] compare Phi with Sandy Bridge CPU for a gyrokinetic code. They note that on accessing an array-size greater than the L1 cache size, the latency of Phi is more than 4× than that on the CPU. In fact, on accessing an array-size greater than the L2 cache size (512KB), the data resides in memory for Phi and L3 cache for CPU, which incur the latency of 330ns and 16ns, respectively. This effect is slightly mitigated by the usage of L2 cache of other cores. They further evaluate Lagrange 1D and 2D kernels which are bandwidth-bound. On Phi, these kernels achieve 85% of the maximum bandwidth but only up to 25% of the peak performance. They also test Lagrange 3D and 4D kernels which are compute-bound. For Lagrange 3D, Phi is nearly 1.5× faster than CPU, but for Lagrange 4D, Phi is only slightly faster due to its complex memory access pattern.

Schmidl et al. [33] compare the performance of a Phi (whose configuration is similar to 5110P) with that of Sandy Bridge CPU. They note that the bandwidth of Phi is highest for 60 threads and degrades on increasing the number of threads. Yet, the bandwidth of Phi is higher than that of CPU. As for memory latency, for small and large strides, the latency of CPU is 55ns and 7ns respectively, whereas that of Phi is 130 ns and 400 ns respectively. Further, the overhead of nested parallel regions is higher on Phi than on CPU. Even though Phi has much higher number of threads than CPU, the OpenMP thread and task creation overhead on Phi is not much higher than that on CPU.

Further, the peak performance of SpMV on CPU and Phi are 11 GFlops and 18 GFlops (respectively) which are obtained using one and two (respectively) threads per core. For all the NAS benchmarks, Phi is slower than CPU even on using all 240 threads. On using single thread, the performance of Phi is between 7.5× to 15× lower than that of CPU, even though the theoretical peak performance of one physical core is almost the same for both. Overall, Phi does not provide comparable performance to CPU as a stand-alone shared memory processor.

Pennycook et al. [82] accelerate molecular dynamics code with SIMD vectorization. Branches are handled using masking. Their code performs “gather and scatter operations” which implicitly transpose the layout between AoS and SoA. Every gather/scatter *instruction* accesses the target elements of a single cache line only. A single “gather/scatter *operation*” references W elements and they may be present in up to W cache lines. Hence, for performing a full “gather/scatter operation”, the instruction needs to be placed in a loop for handling all the elements and thus, each operation requires multiple iterations of this loop. To reduce this instruction count, they utilize the fact that since the position array is stored in AoS format, all the coordinates of atom can be loaded/stored using a single 128-bit instruction. Also, scalar insertion/extraction is replaced with an “in-register transpose”. They perform parallelization at node, socket and thread levels. They do not use Newton’s third law between difference sockets which leads to a few redundant computations but halves the communication over PCIe. Also, communication over PCIe is overlapped with computation of forces for those atoms that do not require cross-domain communication. They show that a single Phi matches the performance of two Xeon processors and the CPU+Phi execution provides even higher performance.

4.2 Mode of operation

Table 4(a) shows the mode of operation used by different works.

TABLE 4
Operation mode of Phi used by different works and parameter settings providing best performance

(a) Operation mode used	
Native	[2, 4, 13, 14, 16, 29–31, 36, 38, 43, 44, 44–46, 48, 76, 84, 91–94]
Offload	[2, 4, 6, 15, 16, 22, 29, 31, 32, 34, 36–39, 42, 47, 76, 84, 93, 95]
Symmetric	[16, 36, 43, 76, 92]
(b) Parameter setting providing best performance	
Thread-affinity strategy	Balanced [4, 14, 20, 21, 23, 26, 36], scatter [37], compact [92], no single winner [13, 84, 94]
Memory mode	Cache [60, 62], flat [55, 90, 96], hybrid (none), no single winner [10–12, 52, 54, 57, 97]
Interconnect clustering mode	All-to-all (none), quadrant [11, 62], sub-NUMA [10, 55, 57, 96, 97], no single winner [52, 96]

Bernaschi et al. [4] compared the performance of 5110P Phi with Tesla K20 GPU for over-relaxation algorithm and a reactive fluid-dynamics problem. Each MPI process runs 220 OpenMP threads. With native mode, use of 16 Phi processors provides only 2× speedup compared to use of one Phi (strong scaling results), even with non-blocking primitives. For hiding the communication latency, offload mode should be used where CPU manages the MPI primitives. In offload mode, increasing the number of Phi processors from 1 to 16 provides a speedup of 6.8×, however, the performance with 1 Phi with offload mode is itself 1.5× lower than that with native mode. In decreasing order of performance, processors are 16 GPUs, 16 Phis and 16 E5-2687W CPUs. As for weak-scaling results, with increasing number of processors, the performance of Phi (native) does not increase well, however, the performance of CPU, GPU and Phi (offload) continues to increase.

For the second problem, the best configuration for CPU is 4 MPI processes and 4 OpenMP threads and for Phi, it is 4 MPI processes with 55 OpenMP threads each. Further, Phi (native), a 16-core CPU and the GPU provided comparable performance, whereas Phi (offload) provided half the performance.

4.3 Thread-affinity strategies

Different thread-affinity strategies have their own merits and hence, they work well in different scenarios. The compact strategy minimizes computation-distances and allows exploiting data locality. Hence, it works well when neighboring threads can benefit from sharing the cache of the same core [27]. However, compact strategy also leads to contention on the core and if the thread-count is below 4 times the core-count, it leaves some cores unused. Hence, it may lead to poor performance than other strategies for some applications [45]. The scatter strategy reduces contention and hence, works well with few threads. The balanced strategy achieves balanced allocation of threads to cores and achieves high cache utilization if the nearby threads operate on contiguous data. On Phi, balanced strategy is the reasonable starting point for any user [78]. Table 4(b) shows the strategy which provides best performance in different research studies.

Ramachandran et al. [13] note that the compact strategy provides best performance only on using all the 244 threads, which is due to its unique cache sharing pattern. For lower thread-counts, scatter or balanced strategies provide the best performance, but on using all the threads, different strategies provide best result for different applications.

Rosales et al. [36] evaluate lattice Boltzmann code on Phi in native mode. On using one-thread per core, balanced and scatter strategies provide the same performance, however, on increasing the number of threads per core, the balanced strategy performs much better. Since the code has high memory-access intensity, compact scheme provides poor performance at low-thread count. However, at high thread-count, both balanced and compact scheme have the same thread distribution and hence, they provide same performance with 240 threads.

4.4 Memory and interconnect-clustering modes

Table 4(b) shows the memory and interconnect clustering modes that provide highest performance in KNL/KNM. Since MCDRAM capacity is limited and data-movement is costly, data-allocation in MCDRAM should be done carefully. Khaldi et al. [55] present an LLVM-analysis based technique which allocates selected variables in MCDRAM. Since data reuse may happen within the same instruction inside a single loop, across the loops in the same function or between loops in different functions, their technique finds number of memory references to a variable in multiple functions. Let R show the reference pattern to a variable. For each variable, they define a priority function as

$P(R) = bandwidth(R) \sum_{r \in R} cost(r) \times workshare(r)$.

If all accesses of R are regular, $bandwidth(R)$ is 1, otherwise, it is 0. This is because irregular accesses make the program latency-bound and allocating them in MCDRAM harms performance. Since stores are costlier than loads, $cost(r)$ equals 2 for stores and 1 for loads. A memory access happening in sequential code or in the sequential region of the parallel code (such as single or master) are said to be "individual" references. By comparison, parallel access is the access made by multiple threads simultaneously. They note that physically, MCDRAM is placed on-package with many wires connected to it that allow accessing multiple memory locations concurrently. Individual accesses cannot benefit from high memory bandwidth and hence, for individual accesses, MCDRAM provides no improvement compared to DDR. As such, they set $workshare(r)$ to 0 for individual accesses and 1 for parallel accesses. They consider only indirect accesses as irregular accesses.

In this way, priority function $P(R)$ is generated for each variable. Then, a transformation pass is added to LLVM, which, at run-time, allocates variables with non-zero value of $P(R)$ in MCDRAM if it exists and has sufficient capacity. Otherwise, these variables are allocated in DDR memory. To achieve this, `malloc` operations are converted to `memkind_alloc`. They illustrate their technique for conjugate gradient benchmark from the NAS Parallel suite. Compared to the malloc-version of code which uses only DDR memory, using the priority value generated by their algorithm in Intel's compiler and LLVM compiler provides large speedup.

HPL (high-performance linpack) benchmark solves a dense system of linear equations, whereas HPCG (high-performance conjugate gradient) performs sparse-matrix operations. Hence, the arithmetic intensity of HPL is high whereas that of HPCG is low. Kang et al. [60] evaluate HPL and HPCG benchmarks on a cluster with 16 KNL nodes. For HPL benchmark, 60% of the theoretical peak performance is obtained for both cache and flat memory modes. By comparison, the efficiency of HPCG is 1.8% and 0.5% when only MCDRAM and only DRAM are used, respectively. Since HPCG benchmarks perform sparse-matrix computations, they are memory-bound and hence, the higher bandwidth of MCDRAM leads to higher efficiency. They further evaluate 6 other applications from N-body problem, structured grid and sparse linear algebra domains. They note that the cache mode generally provides good performance, except when the memory footprint exceeds the size of MCDRAM. On using MCDRAM in flat mode with "`prefer`" option, performance improvement over DRAM increases with rising number of nodes, since the MCDRAM capacity is not sufficient for handling the memory needs when the number of nodes is small. Hence, with small number of nodes, it is better to use MCDRAM in cache mode, whereas for large number of nodes, MCDRAM should be used in flat mode so that the entire workset can fit inside MCDRAM.

4.5 Prefetching

Guttman et al. [41] divide the workloads into three categories based on the effectiveness of data prefetching: (1) insensitive to prefetching, (2) showing larger improvement with hardware prefetching than software prefetching and (3) vice versa. In case the software prefetching is successful, the hardware prefetcher throttles itself and hence, use of both software and hardware prefetching brings the largest improvement for most workloads. Prefetching saves energy in majority of their workloads and has negligible impact on energy or power of workloads that are insensitive to prefetching. Due to the use of extra instructions, software prefetching increases energy consumption in some workloads. Overall, the power overhead of prefetching is justified when it can provide significant improvement in performance. Use of intrinsics improves performance of workloads with random or complex memory access, where both hardware and software prefetching are ineffective. However, some workloads see equal benefit from intrinsics and automatic prefetching by compiler and for them, programmer's effort of adding intrinsics is not justified.

Fang et al. [84] note that on disabling both hardware and software prefetching, memory bandwidth remains low. On using only software prefetching, the bandwidth obtained is same as that obtained on using both of them and thus, hardware prefetching remains disabled when software prefetching performs well. On using only hardware prefetching, the bandwidth obtained is half of that obtained with software prefetching.

Ding et al. [50] apply four optimizations to a particle-in-cell code on Phi. (1) Their code performs several square-root operations, which incur higher overhead on a 512-bit register than on a 256-bit register. Before performing a square-root operation, the compiler internally converts a single-precision variable to double-precision to avoid loss of accuracy. They use static variable to prevent extension of precision. Also, virtual functions are replaced with normal functions. (2) Processing of threads is parallelized using both MPI and OpenMP.

(3) In the code, every cell keeps a linked list of particles residing in this cell. Hence, during list traversal, prefetching cannot be used. They use a look-ahead table for every cell which provides the address of next node (particle). Based on this, the next particle is prefetched using intrinsics. (4) Since the nodes of linked list are stored far-apart in memory, any operation on the fields of a node cannot be vectorized. They fetch and pack fields of four nodes and store them in aligned arrays. This allows vectorization to concurrently process 16 particles instead of only one particle. However, this optimization reduces performance since

the benefit of prefetching is slightly offset and overlapping between computation and memory access is avoided.

Kanamori et al. [66] accelerate "lattice quantum chromodynamics" (QCD) code on KNL. For the complex vector data, the real and imaginary parts are placed consecutively in the memory. The fermion field is shown as a complex vector on lattice sites, which has 4 "spinor" components and 3 "color" components. These components are stored in different registers. For `float` datatype, data on eight sites are packed into one register for concurrent processing. For data-allocation, `std::vector` is used with an aligned allocator that ensures allocation to aligned data in contiguous portion of memory. Arithmetic functions on variables are vectorized using a library which provides wrapper to AVX-512 intrinsics. At each site, 8 stencil values are accumulated, from +x, -x, . . . , +t, -t directions in order. They insert instructions for prefetching data to L1 and L2 caches one and three steps (respectively) before the computation. Thus, before summing the values from (+x) direction, values required for (-x) direction are prefetched in the L1 cache and those required for (-y) direction are prefetched to L2 cache.

Thread affinity is set to compact when multiple threads are mapped to a core and unset if only 1 thread runs on a core. For single-node case, manual prefetching improves performance by more than 20% over auto-prefetching by compiler. However, with increasing number of nodes, the advantage of manual prefetching vanishes. Further, (64 cores/process, 1 thread/core) configuration achieves nearly half the performance of (2,2) and (1,2) configurations. This is because when all 64 cores of a node share the memory, there is contention in access to memory which leads to imbalanced memory access latency to different cores.

5 Parallelization approaches on Phi

In this section, we discuss the works that focus on performing SIMD-level (Section 5.1-5.2) and thread/node-level parallelization (Section 5.3).

5.1 Addressing challenges in vectorization

Farhan et al. [52] accelerate PETSc-FUN3D, which is an unstructured aerodynamics code, on KNL. The main kernel of the code makes calls to the "pseudo-transient Newton Krylov Schwarz" solvers of PETSc library. After loading of the input mesh file from the disk, processing of every mesh components happens in the memory itself. For stream buffer parsing, chunks of mesh components are processed using OpenMP `task` pragmas. Then, data-reordering is performed to ensure that mesh components are stored contiguously in memory which improves access locality and hence, cache hit rate. Data-reordering is parallelized over OpenMP threads using static scheduling.

The PETSc kernels take most of the time in PETSc-FUN3D code. Out of these, majority of time is spent in edge-based loop kernels. These kernels are dominant in four routines of which flux evaluation takes the largest time. In this routine, indirect memory addressing leads to poor cache efficiency. Hence, they rearrange the vertices of the edges in a sequential order using a merge-sort algorithm which is parallelized using OpenMP tasks. This algorithm recursively swaps pointers for building an index array which is used through a "vertex-based loop" for sorting the left-vertex of the edges in ascending order. After this, the right vertex and the edges' normals are sorted. This allows traversing the "index table" of the edges by incrementing a single iterator pointer in a sequential manner. This transforms edge-based loops into vertex-based loops where traversal happens in the order of left-vertices. This increases reuse of each data-item brought in the cache. They perform several time-step updates to a vertex before a cacheline is evicted from the cache. This increases arithmetic intensity (ratio of the computations performed and the data transferred to/from main memory) and reduces the chances of reuse of evicted data-items. Similarly, the "degrees-of-freedom" of each vertex are gathered at contiguous locations in memory which improves chances of its fitting in cacheline boundaries. This allows using strided gather/scatter instructions which also enables vectorization of the kernel.

To reduce overheads of indirect memory references from "unstructured meshing", geometric data are stored in several SoA structures. As shown in Figure 9(a), use of nested C `structs` creates a tree structure where the leaves point to multiple contiguous regions of heap memory. Due to this, geometric data is stored in aligned manner in multiple cache lines. Although this facilitates manual-vectorization, the tree of pointers may leads to pointer-chasing. To avoid this, pointers of only leaf nodes of the tree are passed to the functions, which allows directly accessing the memory location without dereferencing the pointers.

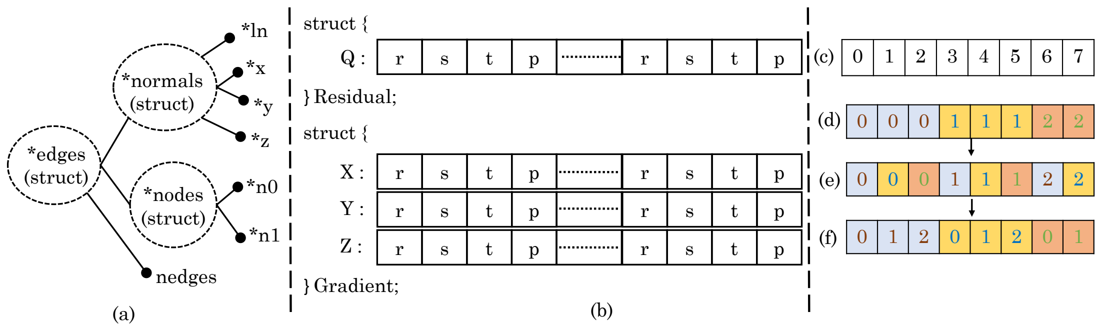


Fig. 9. (a) Tree resulting from the data-structure “`edges`” [52] (b) Gradient and residual vectors data structures. (c) A conflict-free array (d) An array-with conflict [52] (e) identifying lanes that can and cannot be updated concurrently (f) three groups of lanes generated after permuting data-elements. Lanes in a group can be updated concurrently.

They store gradient and residual vectors in “array-of-structs-of-strided-arrays” layout where the arrays of struct are tiled, as shown in Figure 9(b). The tile-size equals the number of state variables, which is 4 in Figure 9(b). This layout allows loop unrolling and also facilitates straightforward vectorization using intrinsics via strided vector gather/scatter instructions. They use a graph partitioning algorithm to uniformly divide the edges and their workloads across the running threads. Their allocation approach avoids atomic operations but still ensures thread-safe operation by doing redundant computations. Specifically, if an edge is shared by two threads, both the threads work on the edge but only the thread that owns the vertex writes to the shared buffer. Thus, global synchronization barrier is avoided and only local synchronization is required due to branch instructions for performing SIMD operations and shared write-operations. The graph partitioning algorithm is parallelized using OpenMP.

For utilizing the MCDRAM of the KNL, they design a heap allocator function for allocating the address space. Based on the needs of the data-structure, memory alignment is performed at the boundary of one/two/four (64B/128B/256B) cache lines. For simplifying gradient kernel, some of its instructions are pre-computed and memoized. Also, to reduce indirect addressing, copies of pre-computed results are stored in memory for each edge. Although it increases the memory footprint and the number of load instructions, it reduces the number of gather instructions.

Above modifications allow Intel compiler to auto-vectorize majority of the kernels. To vectorize remaining kernels having indirect memory accesses, they restructure them and add intrinsics. For loading neighboring edges and indirectly-addressed locations, vector load and gather instructions (respectively) are used. All the writes are done using scatter instruction. Multiply and add/subtract operations are grouped together which allows using FMA instructions. Branch/compare instructions are replaced by masking instructions and division/square-root operations are changed to reciprocal operations. At the start of each loop iteration, the next SIMD lane is prefetched using software gather prefetching instructions.

Writeback (update) operations are performed using “conflict-detection” instructions of AVX-512. For example, the index-array shown in Figure 9(c) can be updated in a SIMD lane without any conflict. This, however, is not true for the array shown in Figure 9(d) since indirect-addressing updates using this array lead to race conditions in a SIMD lane. To mitigate this issue, they perform update multiple times on different indices. For this, data elements of source operands are permuted using swizzle instructions. Thus, input data elements are cloned for creating multiplexed patterns. For example, for the array in Figure 9(f), only three write-back instructions are required on three groups shown by different colors. Thus, three distinct concurrent updates can be performed, which is better than the worst-case of sequentially updating 8 elements. In fact, their reordering optimizations bring the average number of distinct updates to only 2-3.

For increasing the size of independent data subset inside a SIMD lane, the edge data is further reordered using bucket sorting. This ensures that indirect increments of vertex indices form subsets of “maximum conflict-free edges”. Conflicts, if any, are handled by the “conflict-detection” instructions. On 64-core KNL, their techniques and code-modifications improve speed of dominant functions by 2.9× and also achieves linear scaling till 64 threads. Also, they achieve 1.7× speedup over a 36-core Haswell CPU and comparable performance as a 56-core Skylake scalable processor, with much lower power consumption. However, the overall performanc is lower on KNL than on Haswell/Skylake due to poor single-thread performance of KNL.

Lu et al. [91] accelerate MapReduce framework on KNC Phi. From Phoenix++ design, they use two features viz., efficient "combiners" and different "container structures". Further, they propose four Phi-specific optimizations. (1) In the map phase, instead of immediately performing the combiner for every intermediate pair as shown in Figure 10(a), they buffer multiple pairs. Every map operation independently writes to the buffer and once the buffer gets full, the combiner works on the pairs in serial manner. Thus, the dependency between map operations is removed which allows the compiler to perform auto-vectorization of map operations, as shown in Figure 10(b). However, the overhead of using buffer is offset only when the vectorization is effective. (2) Hash computations are manually vectorized. (3) Since map and reduce functions are compute-intensive and memory-intensive respectively, pipelining of map and reduce phases are done using MIMD thread execution. (4) For improving efficiency of combiners, Phoenix++ utilizes a local container for every worker during the map phase. During reduce phase, these containers are merged. However, for large containers, this approach incurs high overhead. They avoid local arrays and since atomic data types incur low overhead on Phi, they perform atomic operations directly on the global-containers. This optimization is applied only when it is expected to improve cache efficiency.

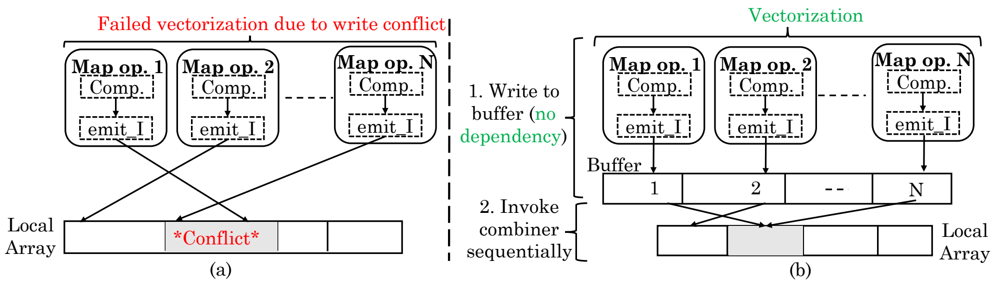

Failed vectorization due to write conflict
Map op. 1
Map op. 2
Map op. N
Comp.
emit_I
Local Array
Conflict
(a)
Vectorization
1. Write to buffer (no dependency)
Buffer
1
2
N
2. Invoke combiner sequentially
Local Array
(b)

Fig. 10. (a) Naive and (b) vectorization-friendly map phase operation [91] (op. = operation)

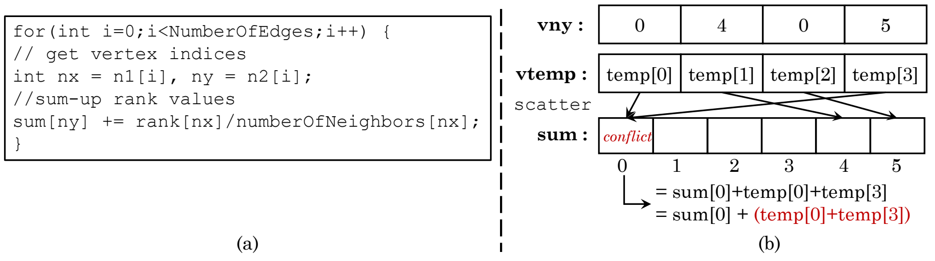
They note that the initialization overhead of each thread on Phi 5110P and Xeon CPU are 0.75 ms and 0.067 ms, respectively. Hence, they use a thread pool to ensure that the threads are initialized only once. The first and fourth optimizations improve the performance significantly. Since hash computation is a not a major contributor to the overall latency, the second optimization provides negligible benefit. Similarly, the storage and memory-cleanup overheads offset the benefits of third optimization. Compared to naively implementing Phoenix++ on Phi, their technique provides large speedup. Compared to running Phoenix++ on CPU, running their technique on Phi improves performance for applications that can leverage vectorization, but reduces performance for applications with large number of random memory accesses due to the small caches of Phi.

Jha et al. [23] use Phi for accelerating main memory hash joins. They run the code written for CPU on Phi and observe that it has poor performance due to random memory accesses. Also, the compiler cannot auto-vectorize performance-critical loops due to data dependency. They perform several optimizations on the code: (1) They use 512-bit SIMD intrinsics to manually vectorize the code. Specifically, hash computations for sixteen 32-bit keys are performed concurrently using SIMD. Also, the SIMD *gather* intrinsic is used for picking only the keys from the relation, i.e., sixteen 32-bit tuples are gathered in a single call of load intrinsic. Further, SIMD vector units are used during build and probe phases for storing and searching tuples in groups of eight for 64-bit keys and sixteen for 32-bit keys. (2) They insert prefetch intrinsics in a manner to bring the data in timely manner.

(3) In the partition phase, large number of memory accesses create a bottleneck. To mitigate this issue, they use software-managed buffers of cache line size for handling writes. A buffer can store eight tuples of 8 byte. When the buffer gets full, these tuples are written in one cache line. (4) Instead of 4KB pages, they use 2MB pages which reduces page faults and improves the hit rate of L2 TLB [98]. (5) Load-balancing is achieved by redistributing the workload of a heavily-loaded thread to another thread. (6) As for thread scheduling strategies, balanced scheme performs the best by virtue of efficiently using all the cores and

data locality in cache. Scatter scheme performs slightly worse due to inefficient use of cache whereas compact scheme performs the worst for less than 240 threads due to inability to use all the cores. Overall, their optimizations improve the performance of hash joins significantly.

Jiang et al. [85] present a technique for accelerating associative irregular reductions. An example of reduction is shown in Figure 11(a) which shows inner loop of PageRank function.



```
for(int i=0;i<NumberOfEdges;i++) {
// get vertex indices
int nx = n1[i], ny = n2[i];
//sum-up rank values
sum[ny] += rank[nx]/numberOfNeighbors[nx];
}
```

Fig. 11. (a) Associative irregular reduction [85] (b) SIMD processing of code from PageRank algorithm

Here, the indices of end-points of all the edges in a graph are stored in arrays n1 and n2. In this code, division operation can be safely vectorized but addition operation is an irregular reduction. To avoid conflicts, updates to `sum` array have to be conducted sequentially, which prohibits vectorization. Previous works propose two strategies for vectorizing irregular reductions. In "inspector/executor" strategy, the data-layout is reorganized for avoiding conflicts in SIMD processing and then, computations are performed in SIMD manner. However, data-reorganization incurs large overhead and may not be possible in some cases. In "conflict-masking" strategy, conflict-free lanes are identified in SIMD vectors at runtime and only these lanes are written in each iteration of execution. In this strategy, hardware utilization depends on the input distribution. If majority of the lanes write to the same memory location, these writes have to be serialized and hence, this strategy offers no performance-benefit.

The technique of Jiang et al. works on the idea that since associativity allows changing the order of reduction without affecting correctness, a partial-reduction can be performed first within a SIMD vector and then, conflict-free lanes holding the partial reduction results can be written to the main memory. Consider the code in Figure 11(a). On vectorizing it, multiple edges will be processed in the SIMD vector and then, sum array will be concurrently updated. Figure 11(b) shows memory accesses in one step of vectorization assuming vector width of 4. Here, vector 'vny' stores the indices of the one end-point of the edges. The computation-results are stored in vector 'vtemp', which is updated in the array 'sum' according to the indices of vny. Here, scattering vt leads to conflict at sum[0] as both temp[0] and temp[3] have to be added to sum[0]. Based on associativity, their technique first adds temp[0] and temp[3], and then, adds the result to sum[0]. Thus, their technique first reduces the values inside the SIMD vector and then, scatters the reduced vector into the main memory. Since the indices of reduced vector are distinct and conflict-free, the conflicts between SIMD lanes are avoided. Their technique alleviates the need of data-reorganization and achieves high SIMD usage efficiency irrespective of input pattern. Their technique only requires in-vector reduction, which is implemented using the "conflict-detection instruction" (`vpconflict`) in the Intel AVX-512 ISA. It detects conflicting updates across the lanes of a SIMD vector. This instruction starts with LSB and checks equality of every element in the index vector with all preceding elements. The output vector shows conflicts of a lane with other lanes, e.g., if k^{th} bit in i^{th} lane is set, then, in the index vector, the value of i^{th} lane is same as that of k^{th} lane.

Figure 12 illustrates the working of their technique on a data vector with an index vector. First, conflict-free lanes are identified which are shown as shaded cells. In iteration 1, beginning with the first conflicting lane (lane 3), all lanes with the same index are identified and are merged with the first lane among them (lane 2). Then, the merged lanes are deactivated. In the next iteration, this procedure is repeated, beginning with first active but conflicting lane. For the example in Figure 12, all conflicting lanes are processed at the end of four iterations. They evaluate their technique on KNL and show that compared to inspector/executor strategy, their technique lowers data-reorganization overhead. Also, unlike conflict-masking strategy, it achieves high SIMD usage efficiency even under unfavorable input pattern.

Ramachandran et al. [13] note that codes with strided access pattern do not vectorize properly since (1)

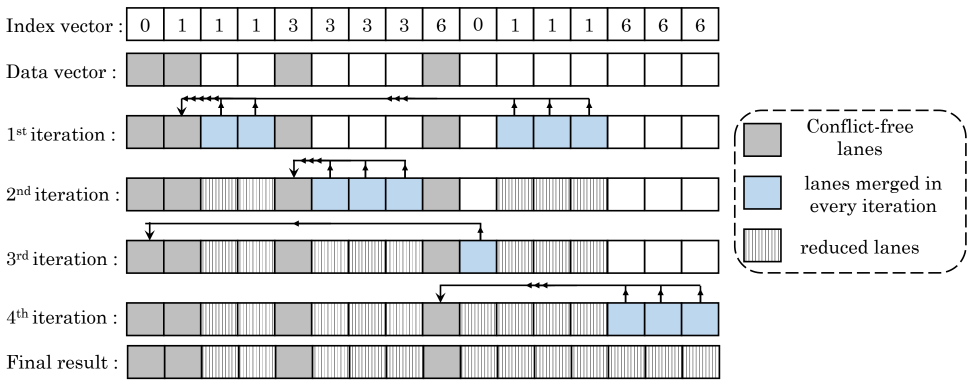


Fig. 12. Illustration of in-vector reduction technique [85]

loading/storing the vector registers requires gather/scatter operations which reduces performance and (2) if the stride is significant compared to the system page size, then the efficacy of hardware prefetching is reduced. However, on using unit-stride, gather/scatter operations are avoided and hence, vectorization is effective.

5.2 APIs for performing vectorization

Huo et al. [79] present an API and runtime for leveraging both thread-level (MIMD) parallelization and vectorization (SIMD) parallelization on Phi. Their technique leverages the communication pattern for partitioning and scheduling the computation for "MIMD parallelism", and reorganizing the data for improving "SIMD parallelism". They illustrate their technique for three types of communication patterns: "irregular reductions", "generalized reductions" and "stencil computation". Their MIMD API provides a `Task` class and a `run()` function. The run function executes a task, while applying runtime optimizations, task partitioning, and scheduling. The SIMD API defines three data types: scalar (e.g., `int`, `float`), vector (e.g., `vint`, `vfloat`) and mask. A mask variable is a bitset and it shows which computations can be performed on which elements. Their key idea is performing overloading of most operators (e.g., +, -, *, %, etc.) on vector types due to which a code vectorized by their API has negligible difference from the serial code (e.g., `vfloat` is used instead of `float`). An example of a code using overloaded datatypes and operators is shown in Figure 13(a) and the expansion of the code is shown in Figure 13(b). Clearly, manual vectorization requires adding large number of lines to serial code.

```
int func(vfloat *p, vfloat
*q, vfloat *r){
for(int i = 0; i < n; ++i)
r[i] = p[i] + q[i];
}
```

(a) A function vectorized by using overloaded functions

```
int func(float *p, float *q, float *r){
for(int i = 0; i < n; i+=16){
__mm512 *s_p = (__mm512*)p[i];
__mm512 *s_q = (__mm512*)q[i];
__mm512 *s_r = (__mm512*)r[i];
*s_r = _mm512_add_ps(*s_p, *s_q);}}
```

(b) Expansion of overloaded functions in (a)

Fig. 13. (a) Vectorization using overloaded functions [79] (b) expansion of overloaded functions

In their API, branches are substituted by mask operations. The MIMD runtime performs task partitioning and runtime scheduling. As for task partitioning in generalized reductions and stencil computations, different loops are assigned to different threads and for irregular reductions, different reduction spaces are assigned to different threads. For runtime scheduling, they provide static, dynamic and user-defined scheduling schemes. The static scheme works well for stencil computations due to low scheduling-overhead, whereas dynamic scheme works well for generalized/irregular reductions by virtue of achieving better load-balancing.

The SIMD parallelization implements overloaded functions using SIMD instructions. Data-contiguity is achieved using AoS-to-SoA transformation and access-alignment is achieved by padding the member array. They note that in irregular reduction, memory accesses can be highly random. Vectorizing them requires scatter/gather operations which take large latency when the target data is present in multiple cache lines. To increase data locality, they present data-reorganization scheme which reorders edge-data based on their first nodes, as shown in step 1 of Figure 14. Due to this, the data of at least one of vertex of the edge is expected to be in the same cache line.

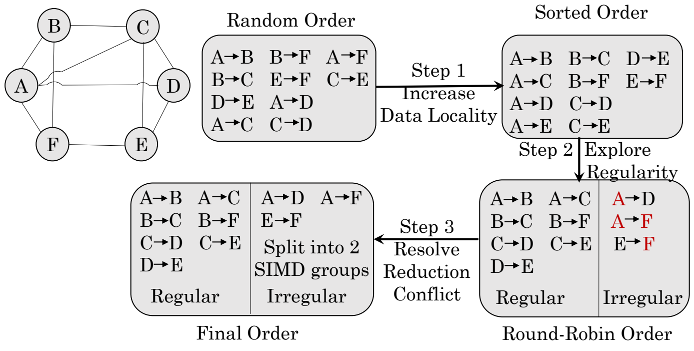


Fig. 14. Reordering edges in an irregular reduction operation (assuming SIMD width = 4) [79]

For maximally replacing scatter/gather operations with normal SIMD store/load operations, the edges are divided into regular and irregular partitions. Then, the edge-data is reordered to have the edges arranged in round-robin fashion based on their first vertices, as shown in Step 2 of Figure 14. This gives a consecutive set of first vertex for the set of edges that can be operated in single SIMD instruction. This forms a "regular partition". The edges that can be operated in single SIMD instruction but whose first vertices do not form a consecutive set make an "irregular partition". Then, AoS-to-SoA transformation is applied for duplicating all the first vertices of edges in the "regular partition". Then, SIMD store/load can be applied to first vertices of edges in "regular partition" and scatter/gather operations need to be applied on remaining vertices only.

To avoid write conflicts in a SIMD register for the second vertices of edges in a "regular partition" and for all the vertices in an "irregular partition", they propose (1) serializing reductions or (2) further reordering the edges, as shown in step three of Figure 14. On using both MIMD and SIMD parallelization, their technique provides higher performance than (1) Pthreads with ICC based vectorization and (2) MIMD and SIMD parallelization using OpenMP.

Hou et al. [88] present a technique for automatically generating SIMD code for parallel sorting on (x86-based) CPU and Phi. Their technique works in bottom-to-top manner and proceeds in four steps: sorting, transpose, merging and code-generation. Figure 15(a) illustrates sort and transpose operations in their technique. (1) Given a sorting network, sort function generates a sequence of comparators. Figure 15(b) shows two 4-key bitonic "sorting networks". Figure 15(d) shows an example of a 4x4 matrix passing through a 4-key sorting network. Here, a dot shows a vector and a vertical line shows comparison between the vectors. Due to data-dependency, not all comparisons can be performed in parallel. For example, in Figure 15(b), comparison(0,1) and comparison(2,3) can be performed simultaneously, but comparison(0,3) can be performed after above two comparisons. Their technique groups comparators based on data-dependency, with the goal of reducing the number of groups.

(2) The vectorized sort function scatters "partially-sorted data" in column-major order, hence, transpose operation is performed for gathering the data into the same vectors (i.e., rows). For this, they use "in-register matrix transpose", which avoids latency-overhead of accessing non-contiguous memory locations but requires complex data-reordering operations. This approach works for even those architectures which do not support scatter/gather intrinsics. For decoupling the binding between the transpose operations and the actual intrinsics, they express data-reordering operations using a sequence of permutation operators. Based on this, the "code generator" phase generates SIMD code for transpose function. (3) Merge operation is performed for iteratively combining pairs of sorted data into a bigger sequence. For this, they use, bitonic merging network since it is easily extensible to any 2^n sized data and it can be easily vectorized as it performs symmetric operations on the elements. Figure 15(c) shows a 8-key bitonic merging network.

(4) Their technique formalizes sort function as sequence of "comparators", and transpose and merge functions as sequence of "vector-matrix multiplications". In code-generation phase, these functions are mapped to operations from a selected "pattern pool", in accordance with the features of parallel sorting. Then, vectorized code is generated with actual ISA intrinsics. They illustrate their technique using

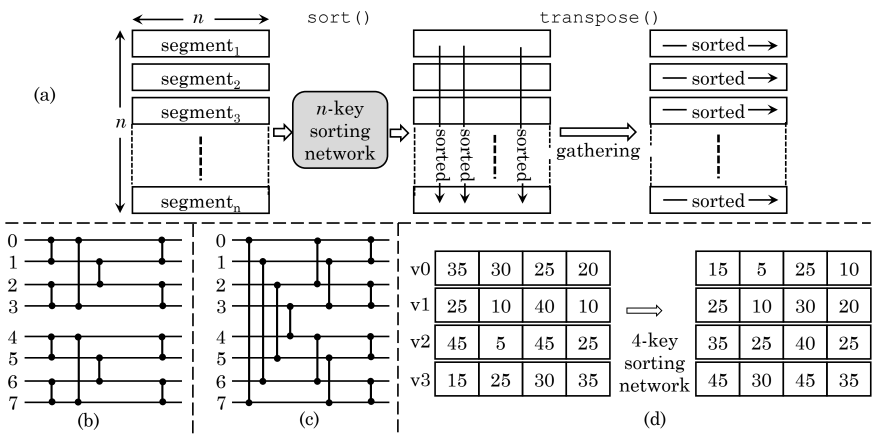


Fig. 15. (a) Operations generated by SIMD `sort()` and `transpose()` functions [88] (b) two 4-key bitonic "sorting networks" (c) one 8-key bitonic "merging network" (d) Use of a 4-key sorting network on four 4-element vectors. Data is sorted after six comparisons and is stored in each column of output matrix.

combinations of 5 sorting networks, 2 merging networks and integer/float/double-datatypes. The sorting code generated by their technique achieves higher performance than Boost, STL and Intel TBB.

5.3 Thread and node-level parallelization

Table 5 shows the parallel-programming languages used by different works. In OpenMP, both static [52, 54, 63, 79] and dynamic [54, 63, 79] scheduling schemes have been used. Also, some works use OpenMP `task` pragmas for parallelization [33, 52, 63].

TABLE 5
Parallelization language/library used on Phi

MPI	[4, 9, 12, 16, 22, 24, 25, 28, 35, 36, 43, 44, 50, 54, 59, 60, 63, 66, 66, 76, 82, 86, 92]
OpenMP	[4–7, 9, 12–14, 16–18, 20, 21, 24, 27–30, 33–37, 39, 42–48, 50, 52, 55, 57, 59, 63, 66, 84, 86, 90, 92, 93, 95, 99]
Intel MKL	[2, 17, 19, 31, 32, 40, 93, 99]
Others	Pthreads [11, 23, 76, 84, 91, 95], Intel TBB [8, 48], Cilk Plus [48], OpenCL [49, 100]

Chatzikonstantis et al. [28] study inferior-olivary nucleus (InfOli) simulation which is used in brain modeling. They accelerate the simulation using (1) MPI, (2) OpenMP, and (3) hybrid MPI+OpenMP. In MPI implementation, each rank processes multiple neurons. To reduce the communication overhead, instead of performing MPI communication for each neuron, they pack the data-values to be sent in a single buffer. Then, based on the neuron connectivity map, the data in the received buffer is distributed to the target neurons.

In the OpenMP implementation, different threads handle different regions of the network. Due to sharing of data between cores, there are overheads due to cache coherence protocol. This reduces performance especially when the network size is small and the number of threads is very large. Hence, deciding the optimal number of threads is necessary to avoid these overheads. In the hybrid implementation, the cores of a processor are organized into groups. Each group forms a single MPI rank and the cores in a group communicate via shared memory. Here, the ratio of "MPI ranks" to "OpenMP threads in each rank" need to be carefully chosen.

They perform experiments on a system with dual-socket Xeon E5-2697v2 CPU and Phi 5110P. For MPI implementation, they use up to 20 and 50 ranks on CPU and Phi. Only one thread is used in each rank (core) of Phi, hence, CPU provides higher performance than Phi. For small workload size, the computation resources of Phi are not fully utilized and hence, the performance gain remains small. For larger workload size, the performance gain increases almost linearly with increasing number of MPI ranks even though increasing the number of MPI ranks increases data-transfer and packing/unpacking overheads. This is because, due to the large memory bandwidth of Phi, the message-exchange costs of as many as 50 ranks can be handled as long as the workload per rank is sufficiently high. As the number of neurons and synapses per neuron increase, the relative performance advantage of CPU over Phi is reduced.

On Phi, the OpenMP implementation uses up to 240 threads and hence, it provides higher performance than the MPI implementation. On CPU, the OpenMP and MPI implementations provide comparable performance since CPU has much smaller number of threads than Phi. For hybrid implementation, they fix the total number of OpenMP threads in all ranks as 20 and 200 for CPU and Phi, respectively, and vary the number of MPI ranks and threads-per-rank. They observe that best performance is obtained on CPU and Phi when running 5 ranks with 4 threads each and 20 ranks with 10 threads each, respectively. Thus, using a balanced number of ranks and threads provides best performance since it balances the overhead of message passing across appropriate number of core-groups and also assigns sufficient workload to every group to leverage the compute capabilities of OpenMP threads.

They further evaluate the best configurations of above three implementations for larger problems. On Phi, for running very large-problems on a multi-node system, only MPI-based implementations are feasible. OpenMP provides better performance than MPI only for smaller problem-sizes. The hybrid implementation provides only marginal improvement over the MPI implementation. On CPU, OpenMP and MPI implementations provide comparable performance which is better than that of hybrid due to lack of support for multithreading. Finally, they vectorize the code and show that vectorization provides larger benefit on Phi than on CPU.

Langguth et al. [35] perform "tissue-scale 3D cardiac simulations" on a heterogeneous CPU-Phi system. They perform parallelization at four levels: across compute nodes, across CPU and Phi in the same node, across cores of the same device and across SIMD lanes of each core. Each node is assigned an equal number of cardiac cells to process. These nodes exchange data using MPI. In a node, cells are assigned to CPU or Phi in proportion to their relative computing power. In the simulation, each cell has a large number (e.g., 10,000) of "dyads". They find that processing a single cell completely on a thread provides much higher performance than processing it in different threads by allocating the work at dyad-granularity.

Since the compiler is unable to vectorize the loops with complex computations, they split such loops in multiple loops. Of these, the loops with simple control flow but heavy computations are auto-vectorized and those with complex control flow but light computations are manually vectorized. The limitation of this approach is that it requires creating multiple intermediate variables that are stored and retrieved from memory. Further, they use pre-computed values of items such as binomial coefficients. In execution of functions, partial values that are repeatedly computed are stored in registers and reused. They note that "random number generation" does not scale on Phi and becomes a bottleneck. To reduce the requirement of generating random numbers, they replace only those random numbers that are actually consumed in a time-step. They perform experiments on a system where each node has dual Xeon E5-2670 processors and two 5510P Phis. Overall, they find that the performance of one Phi matches that of two CPUs.

6 Application Domains

Due to its features, Phi has been used to accelerate applications from different domains and has been evaluated using workloads from various benchmark-suites as shown in Table 6. In this section, we look at the research works in terms of their application areas such as high-performance computing (Section 6.1), AI/machine-learning (Section 6.2) and cryptography (Section 6.3).

6.1 High-performance computing

Barnes et al. [54] accelerate multiple HPC algorithms on KNL. (1) The first application performs "planewave density functional theory simulations" of systems in materials science. Its code has nested `for` loop, such that inside inner-`for` loop, there are vector-multiplication-like operations interleaved by FFTs. Every vector-multiplication is calculated within one OpenMP region, and multiple OpenMP "fork and join operations" are needed for every iteration of the inner loop. Hence, this algorithm shows poor scalability with increasing number of threads.

To lower the OpenMP overheads, they change the loop construction in a manner that every OpenMP region includes a loop over bands. Further, FFT computations are bundled such that every library call performs multiple FFTs. This decreases the number of "fork/join operations" and allows cache-reuse

TABLE 6
Application-domain where Phi is used and benchmark-suite/kernels used for evaluation

Medical/biological/biochemical	[7, 18, 28, 30, 31, 35, 37, 60, 95]
Computational fluid dynamics	[14, 20, 21, 33, 36, 42, 50, 52, 57, 59, 87]
Physics/astrophysics	[34, 43, 46, 47, 59, 66]
Database/data-mining	[2, 23, 33, 39, 55, 64, 91, 100]
Image/Signal processing	[2, 7, 19, 37, 38, 63]
Artificial intelligence (AI)	[45, 62, 94, 99]
Others	Climatic modeling [8, 22, 44], cryptography [26, 27], hologram generation [32]
Benchmark-suite/kernel	SHOC [5, 29, 41], NAS [13, 33, 55], Rodinia [29], AMD accelerated parallel processing software development kit [49], stencil code [15, 22, 52, 79], SpMV [17, 33], matrix multiplication [19, 93]

among multiple FFTs. By virtue of these code-changes, this algorithm achieves much better improvement in performance with increasing thread-count. As for the memory mode on KNL, the cache mode leads to twice the performance of DDR in flat mode. However, they use `FASTMEM` directive to place the important arrays in MCDRAM which allows flat mode to achieve higher performance than the cache mode. Overall, their optimizations enable single-node execution on KNL to outperform execution on Haswell CPU.

(2) Nyx algorithm is a cosmology code from the BoxLib framework and has five key kernels. Of these, three require information from neighboring cells (called "horizontal" kernel) whereas other two do not require data from neighboring cells (called "vertical" kernel). Horizontal kernels such as stencils perform huge amount of data movement and have low arithmetic intensity. Hence, they can be optimized through cache reuse. The original Nyx code used `collapse(2)` directive for the triply-nested {x, y, z} loops in these kernels. However, it leads to large number of misses in last-level cache. Hence, strong scaling performance reaches a plateau at only 5 threads per MPI process. For improving cache reuse, they perform loop tiling whereby large boxes are split into smaller tiles and distributed across the threads. In detail, an MPI process prepares a list of all the tiles of all the boxes it owns. In each parallel region, the OpenMP threads process the list of tiles using "static scheduling scheme". This list is used in place of triply-nested loops and hence, collapse clause is not required. Loop tiling leads to improved data locality and load-balance between threads. Hence, most horizontal operations show effective strong-scaling up to ~64 threads on KNL. They use "pencil"-shape tiles which have large stride-1 dimension for maximizing the efficacy of prefetching and are small in remaining two dimensions to easily fit the tiles in the small 1MB last-level cache of KNL which is shared between two cores. The typical size of tile and box are (64, 4, 4) and (64, 64, 64), respectively. Vertical kernels are more difficult to optimize, e.g., the kernels having data-dependence across iterations cannot be vectorized. They rewrite the algebraic expression of their functions to reduce the number of division operations.

Above optimizations improve the performance of Nyx on KNL by ~5× when running with one MPI process and 64 threads. Also, even for a problem size having memory footprint of tens of gigabyte, the L2 cache miss rate is below 1%. Still, Nyx code is 40% faster on Haswell than on KNL since in absence of an L3 cache, the data traffic from DRAM or MCDRAM is 5× higher on KNL.

(3) From "community earth system model" application, two kernels are accelerated, viz., MG2 and HOMME. For MG2, the expression is simplified for reducing the call to mathematical operations. Division is replaced with inversion of multiplication and compiler flags are used that trade-off accuracy for speed. Directives are added for guiding optimizations and instead of Intel GAMMA function, an internally coded GAMMA function is used which is faster. To allow vectorization, elemental attributes are removed from subroutines and loops are explicitly defined in the routines. These optimizations improve the speed on both Haswell and KNL, but the performance on Haswell is still higher than that on KNL. For HOMME, OpenMP threading scheme was redesigned. Some other optimizations include restructuring loops and data for vectorization, rearranging computations for optimizing data reuse, specifying array and loop bounds at compile time, and implementing specialized communication operations.

(4) XCG1 is a "particle-in-cell code". Its dominant kernel has loops with small trip counts such as 3 and 4, and hence, they are not automatically vectorized by the compiler. They refactor the loops such that the innermost loop is over a block of particles whose size can be changed to that of the vector register.

Compiler directives are used to assist the compiler in vectorizing the loops over particle blocks which leads to reasonably efficient vectorized code. In the interpolation loops, reading data from the specific grid points leads to random memory accesses which requires scatter/gather instructions. To partially avoid this, they perform sorting before the sub-cycle which enhances data locality and improves performance on KNL. Due to absence of L3 cache on KNL, amount of DRAM loads is 3× higher on KNL than on Haswell. Hence, despite higher number of threads on KNL, Haswell achieves higher performance than KNL.

(5) WARP-PICSAR is a particle-in-cell code and is memory-bound due to its low arithmetic intensity. Since successive particles in memory may be far apart in terms of physical positions, various parts of the field arrays are frequently accessed which leads to poor cache efficiency. To enhance locality, they partition MPI domains into tiles for achieving cache-tiling of the field arrays. Parallelization over tiles is achieved using OpenMP and load-balancing is achieved using dynamic scheduling. To further improve locality, particles are sorted after a fixed number of iterations. Since tiling requires particle-exchanges across tiles, they propose suitable mechanism for this. Data-exchange for tiling is merged with that for MPI to further boost performance. To efficiently vectorize the current/charge deposition code, dependencies are removed from the particle loop and the data structure is modified such that grid vertices in memory are aligned. These optimizations improve the performance on both KNL and Haswell and finally, KNL achieves higher performance than Haswell.

Wagner et al. [63] discuss strategies for optimizing FFTXlib on KNL. FFTXlib is the representative FFT kernel of Quantum ESPRESSO which is used by researchers in material science. They use OmpSs, which is a "task-based programming" paradigm based on OpenMP and StarSs. It provides directives for supporting "asynchronous parallelism" through heterogeneity and data dependencies. The baseline code scales poorly with rising number of MPI ranks. The code has execution phases with high and low computation-intensity. In the baseline execution, all processes run compute-intensive phases at almost the same time due to use of static parallelization and synchronization with MPI collective calls. They propose executing every loop iteration, i.e., every FFT, as a single task. As there are no intra-iteration dependencies, different tasks can be dynamically scheduled based on other dependencies and availability of resources. Thus, at any moment, only few processes execute computation-intensive phases whereas others execute computation-unintensive phases. This mitigates resource contention on scaling the number of processes. This optimization is especially effective on Phi since its clock frequency is lower than that of CPU. Their technique reduces execution time by up to 10%.

Chen et al. [64] interface HARP (a Hadoop based communication library) with DAAL (a "data analytics acceleration library"). Their work replaces Java kernels of original HARP project with efficient native kernels from Intel's DAAL at the node level. They evaluate three algorithms: (1) K-means clustering which is compute-bound, (2) "matrix factorization by stochastic gradient descent" (MF-SGD) which is bound by both compute and communication and (3) "alternating least squares" (ALS) which is a communication-bound algorithm. For single-node experiments, they use 64 cores of KNL for computations and 4 cores for running the OS. Each core runs 1 thread.

On metric of execution time of each training iteration, they find that HARP-DAAL outperforms SPARK and NOMAD frameworks. HARP-DAAL reduces instruction execution latency by packing multiple FP operations into a single SIMD instruction by virtue of using AVX-512. A single AVX-512 instruction may issue multiple concurrent L1 cache accesses and thus, it utilizes L1 cache bandwidth more efficiently. By comparison, SPARK, due to it Java-only codebase, executes more than 10× higher number of instructions for K-means and ALS. SPARK works on top of Java virtual machine and hence, it cannot leverage AVX-512 instructions. Hence, its memory accesses do not show temporal locality and cannot saturate the bandwidth provided by MCDRAM. With increasing number of threads, cache contention and inter-core communication increase. Hence, with HARP-DAAL, best performance for ALS, MF-SGD and K-means are obtained at 64, 128 and 64 threads, respectively. By comparison, with SPARK, best performance is obtained at 256, 128 and 256 threads, respectively.

Compared to use of MKL and no AVX-512, use of AVX-512 does not improve performance for ALS and K-means since the MKL functions are already optimized with AVX-512. However, for MF-SGD, significant improvement in performance is seen on using AVX-512 since VPUs of KNL improve throughput for matrix multiplications. Similarly, ALS, non-vectorized MF-SGD and K-means do not benefit from MCDRAM since their L2 cache hit rate is already high. However, for MF-SGD, using both AVX-512 and MCDRAM boosts performance significantly, as AVX-512 instructions produce many memory accesses in each cycle which benefit from high bandwidth of MCDRAM.

6.2 Machine Learning

Domke et al. [51] note that conventional chips allocated a large portion of silicon area to DP computing units, however, recent processors, including KNM, allocate a large portion of chip area to single/half-precision/integer units. They study the impact of this change on the performance of HPC applications, when they run on KNL, KNM and a Broadwell CPU. As for benchmarks, they use 22 applications including HPL, HPCG and apps from "exascale computing project" proxy-apps and "RIKEN fiber" miniapp suite. They note that out of 22 apps, only 4 rely on FP32 instructions and only one uses both FP32 and FP64 instructions. Further, 16 apps execute at least 50% integer instructions. Except HPL, all apps achieve at most 10.5% FP efficiency (i.e., percentage of peak performance) on KNL, 15.1% FP efficiency on KNM and 21.5% FP efficiency on Broadwell CPU. This shows that the utility of FP units is limited. Further, most apps achieve comparable performance on KNL and KNM. Some apps achieve higher performance on KNM due to its higher core-count and frequency, whereas some memory-bound apps show lower performance on KNM due to higher bandwidth contention from larger number of cores. The memory throughput of KNL and KNM are similar for most apps and is higher than that of Broadwell CPU due to presence of MCDRAM on KNL/KNM.

Further, they note that enabling turbo-boost benefits only one app on Broadwell, but nearly all the apps on KNL/KNM. Thus, the apps are more computation-bound and less memory-bound on Phi which is due to use of MCDRAM and its balanced ratio of bandwidth and flop/s. They also note that since the performance of HPCG and AMG (algebraic multi-grid) does not scale with frequency on Phi, these apps are memory-latency bound. Overall, reduction in double-precision compute-capability has small or no impact on performance of most apps. This supports the decision to use low/hybrid-precision compute units in KNM.

Das et al. [68] present an approach for "mixed-precision training" of CNNs using integer operations on KNM Phi. Their scheme keeps precision-sensitive operations such as SGD in single-precision and computation-intensive operations in half-precision. Half-precision training may use either FP16 or INT16. In comparison to FP16, INT16 provides lower dynamic range but higher precision. They present a "dynamic fixed point" (DFP) format which represents tensors using a combination of an integer tensor I and an exponent Es, shared between all integer elements. DFP tensors are shown as DFP-Q where Q is the bit-width of integer elements in I, e.g., DFP-16 has 16b integers. Figure 16 shows FP32, FP16 and DFP-16 format. DFP-16 achieves higher computation density than FP32 and higher effective precision than FP16 by virtue of using 15b mantissa compared to 11b mantissa in FP16.

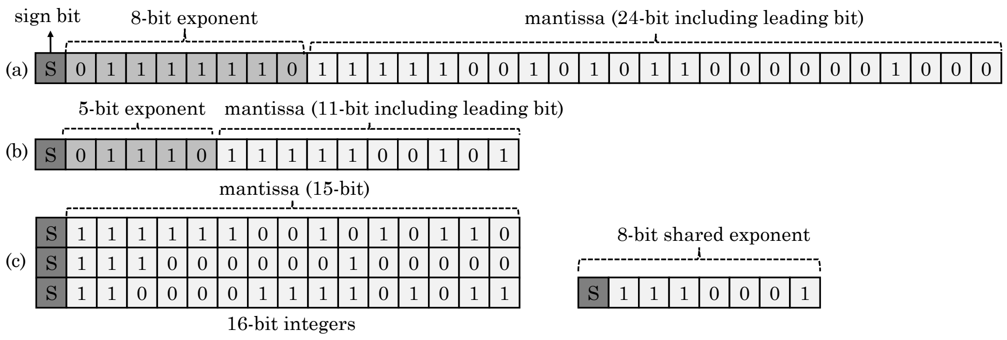

sign bit
8-bit exponent
mantissa (24-bit including leading bit)
(a) S 0 1 1 1 1 1 1 0 1 1 1 1 1 0 0 1 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0
5-bit exponent
mantissa (11-bit including leading bit)
(b) S 0 1 1 1 0 1 1 1 1 1 0 0 1 0 1
mantissa (15-bit)
(c)
S 1 1 1 1 1 1 0 0 1 0 1 0 1 1 0
S 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0
S 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1
16-bit integers
8-bit shared exponent
S 1 1 1 0 0 0 1

Fig. 16. (a) IEEE-754 FP32 (b) IEEE-754 FP16 and (c) Dynamic Fixed Point (DFP-16) data formats [68]

In their work, a single exponent is shared between the tensors. The integers are stored in 2's complement format and the shared exponent is an 8-bit signed integer. For DFP, handling of exponent and precision are done in the software. They further define multiplication/addition operations on DFP tensors and conversion-operations between DFP and float. Since forward/back-propagation and weight-gradient computations can be modeled as GEMM-like convolution operations, they implement these kernels using

INT16 operations, for example, using `AVX512_4VNNI` instruction. These kernels are interfaced with other CNN training operations through DFP to floating-point conversions.

Multiplying two INT16 numbers produces a 30-bit result and accumulating 3 such products can lead to overflow in the INT32 accumulator. During NN training, the length of accumulation chains can be more than a million and hence, they discuss technique for overflow management and show that its instruction overhead is generally less than 1%. For visual-understanding CNNs, the Top-1 accuracies achieved by their technique equals or exceeds that achieved by FP32 training. This is achieved in the same number of iterations as FP32 training and with the same hyper-parameters. Further, their technique also improves the end-to-end training throughput.

Georganas et al. [69] note that implementing convolution as GEMM leads to large memory footprint and memory bandwidth dependency. Hence, they implement convolution using direct convolution which avoids expensive memory accesses due to shuffle/scatter/gather operations. Notice that Das et al. [68] implement convolution as matrix-multiplication whereas Georganas et al. [69] implement it as direct convolution. Figure 17(a) shows the forward-propagation code using direct convolution. They note that for direct convolution, although use of static compilation provides near-peak performance for a given architecture, it incurs overhead of static compilation and necessitates fine-tuning for every CNN topology individually. Especially for small GEMMs, this approach does not achieve high performance on x86 systems due to large overheads. Since the matrices used in CNNs are generally small, they use dynamic compilation approach for generating microkernels for fast direct convolution. This just-in-time (JIT) compilation approach also avoids recompilation and tuning overheads.

(a) Original code

```
for n = 0 to N-1
 for k = 0 to K-1
  for c = 0 to C-1
   for oj = 0 to P-1
    for oi = 0 to Q-1
    ij = stride * oj; ii = stride * oi
     for r = 0 to R-1
      for s = 0 to S-1
      Output[n][k][oj][oi] +=
      In[n][c][ij+r][ii+s]*Weight[k][c][r][s]
```

(b) Code with vectorization and register-blocking

```
Cb = C/VecLen
Kb = K/VecLen
Pb = P/RBlockP
Qb = Q/RBlockQ
for n   = 0 to N-1
 for kb  = 0 to Kb-1
  for cb  = 0 to Cb-1
   for ojb = 0 to Pb-1
    for oib = 0 to Qb-1
    ij = stride * ojb * RBlockP
    ii = stride * oib * RBlockQ
    oj = ojb * RBlockP;
    oi = oib * RBlockQ
    for r = 0 to R-1
     for s = 0 to S-1
      for k = 0 to VecLen
       for c = 0 to VecLen
        for p = 0 to RBlockP
         for q = 0 to RBlockQ
         ij0 = ij + stride * p
         ii0 = ii + stride * q
         Out[n][kb][oj+p][oi+q][k] +=
         Weight[kb][cb][r][s][c][k]*
         In[n][cb][ij0+r][ii0+s][c]
```

(c) Code with register-blocking, microkernel calls and layer-fusion

```
for n   = 0 to N-1
 for kb  = 0 to Kb-1
  for cb  = 0 to Cb-1
   for ojb = 0 to Pb-1
    for oib = 0 to Qb-1
    ij = stride * ojb * RBlockP
    ii = stride * oib * RBlockQ
    oj = ojb * RBlockP; oi = oib * RBlockQ
    CONV (&In[n][cb][ij][ii][0],
    &Weight[kb][cb][0][0][0][0],
    &Out[n][kb][oj][oi][0]) //microkernel
    if (fuse(L()) && cb == Cb-1)//layer fusion
     APPLY (L(), &Output[n][kb][oj][oi][0])
```

Fig. 17. (a) Original forward-propagation code (b) Code with vectorization and register-blocking (c) code with register-blocking, microkernel calls and layer-fusion. Dimensions of input tensor are N, C, H and W and the corresponding output tensor dimensions are N, K ("output feature maps"), P and Q ("output spatial dimensions"). Dimensions of weight tensor are C, K, R and S.

They vectorize computation of output feature maps since they do not have dependency. For this, feature maps are blocked by a factor of `VecLen` and vectorization block is pulled as the innermost dimension of the tensors. Value of `VecLen` is chosen based on ISA and data-type, e.g., for AVX-512 and FP32 data, `VecLen` is 16. Further, register blocking is performed in the spatial domain of the output tensor since points in the spatial iteration space can be computed in parallel. This creates independent accumulation chains in registers which helps in hiding FMA latency. The resultant code is shown in Figure 17(b), where register-blocking factors are `RBlockP` and `RBlockQ`. Register blocking helps in improving data-reuse from registers and reducing traffic to L1 cache. To further improve data-reuse from caches, cache blocking is performed in the feature map and spatial dimensions.

They further use two-level software prefetching scheme: (1) L1 cache prefetches are issued to bring the data to be used later by the *same* microkernel invocation. (2) L2 cache prefetches are issued to bring sub-tensors of *future* microkernel invocations. Although prefetching alleviates cache miss latency overheads almost entirely, it requires a complex logic for ascertaining pointers of sub-tensors to be used by future microkernel invocations. They parallelize microkernel execution to the available threads.

They further perform CNN layer fusion which saves bandwidth by using the data-items that are currently hot in cache. The limitation of layer-fusion is that it introduces conditional statements for deciding when the operator can be applied. Figure 17(c) (except last-two lines) show the forward-propagation code with convolution microkernel and the last-two lines show the code for layer-fusion. The microkernel takes pointers to output, input and weight sub-tensors.

For ResNet-50 CNN, they present performance results of (1) forward/backward/weight-update propagation kernels and (2) full network execution. (1) For execution of individual kernels, KNM Phi provides higher absolute performance than Skylake CPU, although the percentage of peak performance provided by Phi is lower than that of CPU. This is especially pronounced for weight-update propagation kernel since unlike CPU, Phi lacks a shared last-level cache that absorbs majority of data-movement operations due to weight-reduction. On Phi, for forward-propagation, INT16 kernels provide 1.63$\times$ speedup over FP32 kernels. The speedup remains less than 2$\times$ since even though tensor size is half, the output is still 32b and hence, its bandwidth needs are not reduced. Also, for avoiding overflows in output registers, length of FMA accumulation chain need to be restricted which reduces register data reuse and speedup achieved.

(2) For full-network, they compare end-to-end performance of training ResNet-50 using a lightweight framework on CPU/Phi with Tensorflow framework using cuDNN on a P100 GPU. The performance on CPU, Phi and GPU are 136, 192 and 219 image/second, respectively. On CPU and Phi, they further strong-scale to 16 nodes which uses 896 CPU cores and 1152 Phi cores, respectively. With increasing number of nodes, they achieve near-linear scaling and for 16 nodes, the performance of CPU and Phi are 1696 and 2430 image/second, respectively.

Dixon et al. [99] compare 7120 Phi with Xeon E5-2690 v2 CPU for accelerating a deep neural network based financial market predictor. They note that since MKL BLAS routines are optimized for every platform, they should be preferred over OpenMP. Also, to efficiently utilize hardware resources, matrix-dimension should equal or exceed the number of threads. Further, to avoid race conditions while also optimizing data locality, each core should access a single, contiguous yet distinct portion of memory. They note that with increasing batch size, higher speedup is obtained, although use of large batch size reduces the convergence of the learning algorithm. Hence, they suggest setting the batch-size to 1000. The back-propagation step and feed-forward network construction step account for a majority of time in serial execution. On using 240 threads, the speedup of every mini-batch iteration of size 1000 over a serial implementation on CPU is 11.4$\times$. Thus, after parallelization of back-propagation, only feed-forward step remains the bottleneck.

Viebke et al. [45] perform supervised learning of convolution neural networks on 7120P Phi. They create multiple workers (OpenMP threads) that share weights. Remaining variables are private to each thread, which allows processing multiple images in parallel. The images are allocated to the workers dynamically which allows fast workers to process more images than the slow workers. The shared weights are updated only at the end of the computation of each layer which avoids unnecessary invalidation of cache lines. Intermediate updates are performed on local weight parameters. Memory allocations and accesses are aligned to 64 byte. Computations of weight gradients and partial derivatives are vectorized to exploit SIMD parallelism. Compared to single-threaded execution on Phi and sequential execution on a Xeon E5-2695v2 CPU, use of 244 threads provides more than 100$\times$ and 10$\times$ speedup respectively. However, due to non-deterministic updates of weights, the classification accuracy on Phi is slightly lower than that on CPU.

6.3 Cryptography

Yao et al. [27] accelerate RSA (Rivest, Shamir and Adleman) cryptography operations in SSL (secure sockets layer) protocol. The main kernel of cryptographic operations is "big integer multiplication". To leverage the 512 bit wide SIMD instructions, they use an asymmetric parsing multiplication approach which is shown in Figure 18. It parses the multiplier at the granularity of a vector (512 bit) and the multiplier at the granularity of a word (32 bit). The multiplication algorithm has two nested loops: the outside loop goes over the multiplier "vector-by-vector" and the inside loop goes over the multiplicand "word-by-word". In every iteration of the inside-loop, a word of the multiplicand is broadcast into a vector operand, which multiplies a vector obtained from the multiplier using the "SIMD-multiply intrinsics". Its output is accumulated into the "result vector register" which is initialized to zero. Intrinsics provide high efficiency due to operating in registers.

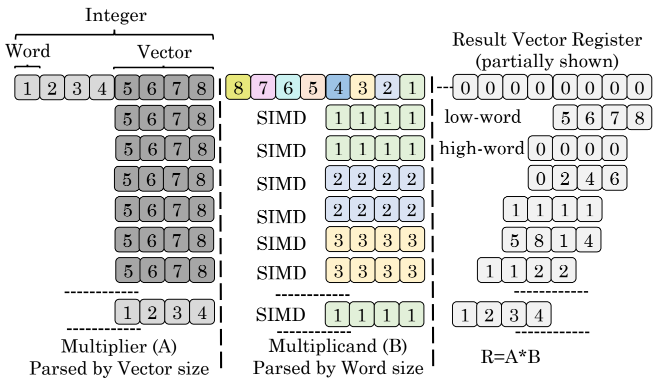

Integer
Word
Vector
Result Vector Register (partially shown)
SIMD
low-word
high-word
Multiplier (A) Parsed by Vector size
Multiplicand (B) Parsed by Word size
R=A*B

Fig. 18. Asymmetric parsing mechanism [27]

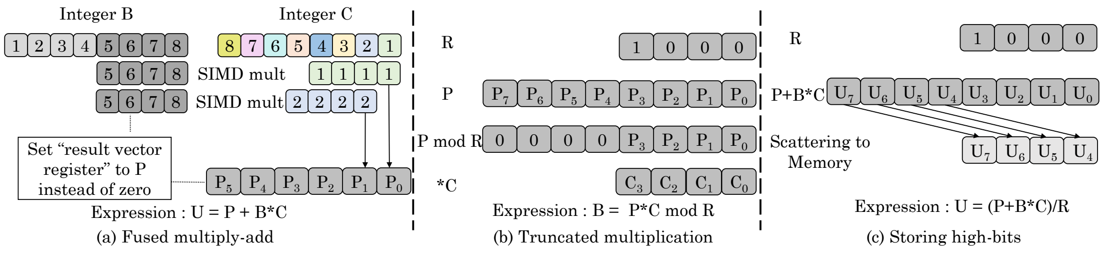


Fig. 19. Phi-specific optimizations for Montgomery algorithms [27]

To compute a big integer's modular exponentiation, Montgomery algorithms are used. They propose three Phi-specific optimizations to these algorithms which are shown in Figure 19: (1) For fused-multiply-add operation ($P + B \cdot C$), the result vector is initialized with P which avoids the need of a separate addition operation and caching the intermediate results. (2) The expression $P \cdot C \, mod \, R$ is converted into an equivalent form: $(P \, mod \, R) \cdot C \, mod \, R$. Here C and R are q-bit integers and P is $2q$-bit integers. Since the $mod \, R$ is achieved by bit-truncating the upper q bits, the multiplication between C and the higher q bits of P is redundant and hence, avoided. (3) For $(P + B \cdot C)/R$, a naive implementation stores the result of $(P + B \cdot C)$ into memory, then loads it and performs right-shift operation. They use SIMD intrinsics to directly load/save $(P + B \cdot C)$ results from/to the right memory locations. They also propose algorithmic optimizations for RSA library. They also note that for their application, the compilation process of Phi is very complex which may make Phi unattractive for many users. On Phi, their implementation has higher throughput than the *libcrypto* library by Intel and the default OpenSSL implementation.

Chang et al. [26] accelerate RSA cryptographic algorithm on Phi. They present several optimizations to RSA algorithm without compromising its security. Among the thread affinity schemes, balanced scheme was found to provide the best performance. They show that on the latency metric, their proposed RSA implementation on Phi is better than OpenSSL on Phi and is comparable to OpenSSL on CPU. Similarly, the peak throughput of their implementation on Phi is much higher than that of OpenSSL on Phi or CPU.

7 Comparative evaluation and collaborative execution

Several works have compared CPU or GPU with Phi. The results of these performance comparison studies are summarized in Table 7. In general, GPU provides higher performance than Phi which, in turn, provides higher performance than CPU. Note that some works do not perform vectorization on Phi [16, 32, 33, 41, 76, 93] whereas others perform only sequential execution on CPU [45, 99]. Table 8 shows the factors causing performance bottleneck on Phi. In this section, we first review the works that focus on comparing Phi with CPU (Section 7.1) and GPU (Section 7.2). We then discuss the works that utilize heterogeneous computing techniques (Section 7.3).

TABLE 7
The processor providing better performance for all benchmarks:

CPU vs. Phi.	
CPU	[13, 18, 19]
Phi	[4, 6, 8, 14, 15, 31, 32, 35, 38–40, 44–47, 52, 54, 57, 60, 82, 84, 99]
No single winner	[2, 20, 23, 28, 29, 33, 37, 43, 50, 59, 94]
GPU vs. Phi	
Phi	[38, 40]
GPU	[4, 7, 16, 18, 19, 21, 25, 31, 32, 46, 57, 84, 93]
No single winner	[8, 29, 31, 34, 37, 62]

TABLE 8
Performance bottleneck reasons

Memory and communication related factors	
Irregular/indirect memory accesses	[9, 13, 14, 17, 23, 29, 37, 52, 84, 87, 91]
Cache-related	excessive cache misses [17, 54], lack of shared last level cache [14, 29, 69]
Slow communication on	Memory [29, 35, 50], PCIe bus [14, 19, 24, 36, 50, 82, 95], between two Phis in the same node [43]
Core and performance-scalability related factors	
Poor single-core performance	[20, 33, 52, 59]
Vectorization-related	Difficulties in or imperfect vectorization [2, 22, 25], incompatibility of SSE-style vectorization [2]
Poor scalability with increasing threads/nodes	[2, 16, 20, 29, 37, 57, 62, 66, 84]
Slow operations	reduction operation [86], gather-scatter operations [82], divide operations [13, 26]
Data dependency	[99]
Load-imbalance between	threads of Phi [13, 17, 17, 23, 50], CPU and Phi [18, 43, 50, 76]
Contention on shared components	ring stop [29, 37, 84], disk-access [37]
Others	lack of support for vector atomic instructions [37], thread-creation overhead [30], TLB misses [42, 90]

7.1 Comparison with CPU

Surmin et al. [59] use Phi to accelerate a "particle-in-cell laser-plasma simulation" code. They use MPI to achieve distributed computing by using spatial domain decomposition. Also, OpenMP threads are used for processing particles in different cells. Also, vectorization is achieved using SIMD instructions and intrinsic-based coding of some stages. They perform experiments using 7250 Phi (KNL), 7120 Phi (KNC) and Xeon E5-2697 v3 CPU and use double precision. On both KNC and KNL, use of 4 threads per core provides the best performance. Without applying any optimization and using one MPI process per device, KNL provides 2.4× and 1.5× improvement over KNC and CPU, respectively.

On optimizing further, they note that using 8 MPI processes and 34 OpenMP threads in each process provides best speedup on KNL. Furthermore, grouping multiple nearby cells into "supercells" improves performance. They choose an optimal size of supercell for CPU, KNC and KNL based on the following constraints: (1) the data processed can fit in the L1 cache and (2) with increasing size of supercells, the number of independent problems that can be concurrently solved is reduced. Finally, on auto-vectorizing a loop, the execution time is increased on KNC, but reduced on KNL which is due to the AVX-512 instruction set. Overall, KNL provides significant improvements over KNC which makes it a promising platform.

Crimi et al. [87] compare KNC Phi with Tesla C2050 GPU and two-socket Sandy Bridge CPU for a lattice Boltzmann code. In their implementation on Phi, every thread processes a portion of the lattice. In every thread, Q sites are concurrently processed for leveraging the data-parallelism provided by 512-bit vector instructions. Here, $Q = 8$ for double-precision instructions and $Q = 16$ for single-precision instructions. They use intrinsic functions for introducing vector variables and processes. This approach provides better performance compared to the compiler performing auto-vectorization.

They evaluate two kernels of the lattice Boltzmann code, viz., propagate and collide. The "propagate kernel" performs copy operations within memory with a sparse addressing sequence. Hence, its performance depends on calculating addresses and accessing memory for short data sequences. They observe that for the propagate kernel, the bandwidth obtained increases nearly linearly until 120 threads. Beyond this, there is only marginal improvement in performance until 240 threads and beyond this, the bandwidth degrades. Due to the frequent and complex memory access-pattern of propagate kernel, only 16% of peak-bandwidth (GB/thread) is achieved. The absolute value of bandwidth is lower than that achieved on CPU and GPU.

The "collide kernel" is the most computation-intensive portion of the code. It scales nearly linearly till 60 threads. Beyond this, it shows sub-linear scaling until 240 threads, where nearly 25% of peak performance (GFlop/thread) is obtained. One reason for this is that the collide kernel performs a variety of math operations and expressing them in the form of "multiply-add pattern" is not straightforward. Still, the absolute value of performance is higher than that on CPU and GPU.

Li et al. [29] compare 5110P Phi with Sandy Bridge Xeon E5-2620 CPU using Rodinia benchmarks and with Tesla c2050 GPU using SHOC benchmarks. They report the performance/energy of computational kernel of the benchmark. As for SHOC benchmark suite, they note that `MD`, `GEMM` and `FFT` benchmarks do not scale for more than 120 threads whereas the performance of `reduction` stops scaling at nearly 30 threads. In fact, `GEMM` shows performance loss after 120 threads. The reasons for this are: (1) small L1/L2 caches and absence of L3 cache, which leads to high number of memory accesses (2) memory contention due to high "memory to computation ratio" and large thread-count (3) congestion in the ring interconnect. The trends of energy consumption are similar to that of execution time. Further, during data transfer, both host and Phi consume large amount of power.

As for comparison with GPU, on the memory-bound benchmark `reduction`, Phi provides higher performance than GPU for larger benchmark-sizes (s3 and s4) but dissipates higher power for all benchmark-sizes. In terms of energy, use of extra energy by Phi is justified only for the s4 benchmark-sizes. For `GEMM` benchmark, which is compute-bound, Phi outperforms GPU for s3 and s4 benchmark-sizes. For s1 and s2 benchmark-sizes, Phi consumes higher power than GPU, however, for s3 and s4, their power consumption are comparable. In summary, Phi provides higher performance than GPU on computationally-dense kernels and consumes equal or higher power than GPU in all cases. This is because both idle and active power consumption of Phi is higher.

On Rodinia benchmarks, Phi provides lower performance than CPU for memory bound kernels with frequent random memory accesses. This is because CPU has superscalar execution, has high frequency and 15MB L3 cache, whereas KNC Phi has simple cores with lower frequency. For compute-bound kernels, the performance of Phi matches that of CPU for `streamcluster` kernel and exceeds for `kmeans` kernel.

Summary: KNC has poor performance due to low frequency and in-order execution. KNL has several improvements over KNC. CPU has high single-core performance due to its high frequency and optimized memory hierarchy. Also, CPU is suitable for a wide range of applications, especially those which are not amenable to parallelization or vectorization.

7.2 Comparison with GPU

Gawande et al. [62] compare NVIDIA DGX-1 against a cluster with Intel KNL and Omni-Path interconnect. DGX-1 has 8 Pascal P100 GPUs with NVLink interconnect. They evaluate vendor-provided models of four CNNs, viz., CifarNet, AlexNet, CaffeNet and GoogleNet. They use Caffe frameworks provided by respective vendors optimized for their systems. For single-node execution on KNL, they evaluate three configurations: quadrant clustering with MCDRAM in (c1) cache or (c2) flat mode and (c3) "all-to-all clustering" with MCDRAM in flat mode. In decreasing order of performance, these configurations are (c1), (c2) and (c3). Hence, they use configuration (c1) for scaling experiments with KNL.

For scaling studies, they evaluate use of 1, 2, 4 and 8 GPUs on DGX-1 and 1, 2, 4 and 8 KNL nodes. For CifarNet, performance of KNL shows sub-linear increase with rising number of nodes due to communication overheads. For DGX-1, the performance improves for 2 GPUs due to efficient use of memory of 2 GPUs as buffer during computation and due to NVLink, which provides high-bandwidth. However, the performance drops with 4 and 8 GPUs, since the computation requirements of CifarNet are not large enough to utilize all the GPUs. As for power efficiency, for CifarNet, DGX-1 provides higher power efficiency and performance than KNL cluster. For AlexNet, DGX-1 with one GPU gives similar results with one KNL node. However, the performance of KNL does not scale with nodes whereas that of DGX-1 scales well, although not linearly. Overall, DGX-1 lies on the power-performance Pareto front. Since AlexNet and CaffeNet are similar, all their results are also similar. GoogleNet could not run on one or two GPUs due to its high memory requirement which required 4 or 8 GPUs. DGX-1 with 8 GPUs shows higher performance than even KNL with 16 nodes. The performance of KNL improves slowly with rising number of nodes and the highest power efficiency is shown by the single-node KNL. In summary, GPU provides higher raw performance and NVLink is crucial for GPU scaling. Also, KNL can provide performance/watt comparable to that of GPU.

Hofmann et al. [7] study "Feldkamp-Davis-Kress" (FDK) algorithm which is used for reconstructing 3D images. They use FMA instructions to reduce the number of arithmetic instructions. They replace the divide operations with multiplication with the reciprocal, which is fully pipelined and offers higher accuracy compared to existing CPU implementations. To load a large amount of data from various offsets, they use the vector gather operation. This is more efficient than sequential loads since intermediate writing of data in stacks to store them in scalar registers is not required.

Since their algorithm is instruction throughput-bound, use of zero-padding for handling the `if` statements provides higher performance than using predicated instructions. Further, in L2 cache, use of software prefetching provides better performance than hardware prefetching. For OpenMP parallelization, they perform loop-collapsing. The "compact" thread-affinity scheme allows exploiting spatial locality and avoids cache pre-emptions.

They find that the kernel runtime is dominated by the gather operation. Further, their hand-written assembly kernel outperforms the code auto-vectorized by Intel compiler. The code vectorized by `#pragma simd` of OpenMP 4 provides even lower performance since it does not optimize the code aggressively due to its guarantee of providing vectorization results identical with that of scalar code. The implementation of FDK algorithm on GeForce GTX680 GPU provides nearly 7$\times$ higher performance than their assembly-level implementation on Phi. This is because on GPU, texture units are used and the massive multithreading approach of GPU can hide the latencies of instruction-execution and memory-access more effectively than the in-order core of KNC Phi which has only 4 threads in each core.

Teodoro et al. [37] analyzed the performance of KNC Phi, Tesla K20 GPU and Xeon E5-2680 CPU for operations in object segmentation and feature detection algorithms. For Phi, on increasing the number of threads, the peak in performance is observed for thread-counts of 60, 120, 180 and 240 due to uniform distribution of threads (workload) across cores in these cases. The performance of an operation with regular access-pattern scales only till 120 threads. Beyond this, congestion in memory degrades memory throughput. The performance of another operation with irregular memory access pattern continues to scale till the number of threads reach 240 since the memory bandwidth achieved continues to increase.

Further, for operations with regular access pattern, GPU provides higher speedup than Phi for memory-bound operations whereas Phi provides higher speedup for compute-bound operations. For operations with irregular access, GPU provides higher speedup than Phi due to its higher read/write-bandwidth for random data access. In fact, write-bandwidth of Phi is even lower than that of CPU. This is because Phi has to ensure cache consistency among its large number of cores. This leads to high data traffic and congestion in its ring-bus that connects the caches. For operations that use atomic instructions, the processors in decreasing order of performance are: GPU, CPU and Phi. Phi does not support vector atomic instructions. They further evaluate CPU-Phi collaborative execution using a scheduling strategy which allocates tasks to CPU or Phi, based on an estimated speedup of the task on Phi and on the current loads on the CPU and Phi. This strategy improves performance compared to the first-come first-serve task-allocation strategy.

Fang et al. [30] evaluate Phi for "non-bonded electrostatic computation" which is used for drug discovery applications, and compare it with K20x GPU. On Phi, converting from AoS to SoA format doubles the performance, whereas on GPU, AoS style data structure is more efficient. The GPU outperforms Phi for all problem sizes. The performance of Phi is especially low for smaller problem size, since the thread-creation overhead dominates the execution time.

Plazolles et al. [46] explore acceleration of Monte-Carlo simulation of the drift descent in the study on "stratospheric balloon envelope". They compare 7120P Phi with a K40 GPU and a two sockets E5-2680 v2 CPU. Monte Carlo simulation is an example of an embarrassingly parallel kernel. For Phi, use of the following optimization strategies provided significant speedup on top of an implementation using task-parallelism: storing the vectors in the local memory of every OpenMP process using private variables and fixing the vector-size to 32 at compilation time. Overall, for the largest workload size, Phi and GPU provided 2× and 4× speedup compared to the CPU implementation.

Jeong et al. [25] compare the performance of 5110P Phi with GPUs for conjugate gradient solver, which is used for quantum chromo dynamic simulation of lattice. They note that their code is incompatible with the format of SIMD registers required for vectorization. Hence, GTX Titan and GTX 480 provide 10× and 5× speedup compared to Phi.

Salim et al. [93] evaluated the performance of 7120P Phi and Tesla K20c GPU for multiplication of large matrices. They used Intel MKL and MAGMA ("matrix algebra on GPU and multicore architectures") libraries on Phi and CU-BLAS library on the GPU. They observe that on Phi, the performance of MAGMA is better than that of MKL, and is comparable with the performance of CU-BLAS on GPU. However, due to its limited memory capacity, GPU cannot perform multiplication of square matrices of size larger than 20,000, whereas Phi can multiply matrices of size as large as 60,000.

Chen et al. [34] accelerate gravitational microlensing simulations on Xeon E5–2670 CPU, 5120 Phi (KNC) and 7250 Phi (KNL) and Tesla M2050 and Tesla K20 GPUs. Both KNC Phi and Tesla M2050 GPU provide nearly 2X speedup compared to CPU, whereas Tesla K20 GPU and KNL Phi provide 4.1× and 11.8× speedup, respectively.

Summary: KNL and KNM Phis have higher memory capacity than GPU, which allows them to run even those codes which cannot run on GPU [62, 93]. The strength of GPU lies in use of massive multithreading and high memory bandwidth. Also, texture units bring large speedup for graphics applications.

7.3 Heterogeneous Computing

Table 9 shows the works that utilize multiple processing-units to bring their best together.

TABLE 9
Heterogeneous computing using Phi

CPU+Phi	[4, 6, 15, 18, 22, 35–38, 42, 43, 47, 49, 76, 82, 92, 95]
CPU+GPU+Phi	[15, 19]

Vialle et al. [15] accelerate shortest path computation on a machine which uses both Phi and GPU as accelerators. They partition the 2D grid in three horizontal strips that may have different heights. The entire current and previous grids are hosted in CPU memory and top/bottom parts are transferred to GPU/Phi, respectively. Since direct communication between GPU and Phi is impossible, placing CPU in the middle is effective. Every processor (CPU/GPU/Phi) stores its portion of the grids and the boundaries of other processors. The interaction of CPU with both GPU and Phi are mostly similar, except one difference: for optimizing data transfers and avoiding the need of intermediate arrays, the frontiers are transferred directly in their right place in the local arrays on CPU and GPU. However, for Phi, an intermediate buffer is required due to the slow memory allocation of offloaded data and the fact that transferring data to a locally allocated array on Phi is not possible. A frontier is transferred only if it has seen at least one modification. Computations and communication to/from Phi/GPU happen asynchronously, which allows overlapping computation with communication. At the end of every iteration, synchronization is performed to ensure that the computation of and data-transfer from previous iteration are completed before beginning the next iteration.

They note that when each processor is used alone, the decreasing order of performance is GPU, Phi and CPU. On using their heterogeneous computing approach, the performance is higher than that obtained with GPU alone and this confirms the effectiveness of using heterogeneous computing. On Phi, performing vectorization using `#pragma simd` and `#pragma ivdep` directives and AVX compiler options improved performance. They perform experiments on two machines with different models of CPU/GPU/Phi. On one machine, use of three processors provides no improvement over the best solo processor, whereas on the other machine, it provides significant improvement.

Lan et al. [95] accelerate biological sequence search using Smith-Waterman algorithm on a system with two Xeon E5-2620 CPUs and two 7110P Phi cards. They implement their algorithm using Pthreads and OpenMP. To perform data-distribution at thread level, they create and initialize threads on both CPU and Phi. Since independent portions of the database can be concurrently searched, they divide every sequence subset into multiple batches. CPU threads operate on one batch at a time, whereas Phi threads process multiple batches concurrently. On CPU, SSE intrinsics and multi-threading are used to achieve SIMD parallelism. On Phi, vector instructions are used to achieve parallelism.

They note that for query sequences larger than 1000 residues, their technique has poor performance since the L2 cache cannot store the entire working set. To resolve this issue, they partition long query sequences into multiple short sequences which are searched successively. This reduces the working set to a level that can fit in the L2 cache, although it requires storing some extra variables for ensuring correctness. Their technique achieves performance which is comparable to that achieved using state-of-the-art Smith-Waterman implementation on dual Tesla K40 GPUs and better than that of Liu et al. [101]. Also, only their technique can work with very large databases.

Apra et al. [6] implement a many-body quantum chemical code on a heterogeneous system with both CPU and Phi. They execute compute-intensive kernels on Phi using offload mode. They leverage thread-level parallelism using OpenMP and SIMD parallelism using auto-vectorization capability of the compiler. Multiple outer loops are collapsed into a single loop using the `collapse` clause. A hint about the smallest and largest trip counts of the loops is provided to the compiler to prevent it from applying transformations that harm performance. Further, they transpose one of the arrays which makes the accesses to this array as unit-stride and thus, avoids the overhead of gather/scatter operations. Finally, they ensure that the data is 64-byte aligned, which is required for efficient implementation of SIMD load/store operations on Phi. This improves the bandwidth of DMA (direct memory access) transfers and reduces the latency of aligned loads/stores. They perform evaluation on a system with dual-socket E5-2670 CPUs, having two 5110P Phis on each node. While offloading data to Phi, the data are stored in 2MB pages which reduces TLB misses and page faults. While Phi-only implementation outperforms CPU-only implementation, use of both CPU and Phi provides the best performance and scales to a total of 62560 Phi and CPU threads.

8 Conclusion and Future Challenges

In this paper, we presented a survey of works that use Xeon Phi as an accelerator for a broad-range of compute-intensive applications and compare its performance with other processors. We discussed works that study Phi architecture, performance bottlenecks and optimization strategies. We close this paper with lessons learnt from design and optimization of Phi.

With many Phi-powered systems getting into the Top500 supercomputers list for multiple years, Phi has definitely made a mark on the landscape of HPC systems. It has brought "host" and "accelerator" into one system which reduced data-transfer overheads and allowed using same ISA, programming models and libraries for both host and accelerator. Due to this, Phi can be used to accelerate a wide range of applications. Phi has implemented many powerful features and innovations, such as very wide vector-units, variable-precision floating-point operations, use of both high-bandwidth and high-capacity memory, etc. They will surely inspire the design of future computing systems.

The main motivation and justification for development of Phi was ease of programming. Certainly, *programming* Phi takes much less effort than that required for accelerators such as FPGA (field programmable gate array) and may be even GPU. However, *achieving peak performance* on Phi is not easy, as it requires myriad set of optimizations and understanding of the microarchitecture [11, 94]. The pragma-based programming style makes the code complex, especially for applications that use both CPU and Phi. Large coding-efforts and difficulties in debugging reduce programmer productivity and create a lag between progress in algorithm/application-domain and their acceleration on Phi. This is especially significant in

HPC domain where the scale of applications is so very large that ease of porting becomes a decisive factor. Future processors need to provide high-level APIs and effective compilers to bring the best from the underlying architecture with minimal efforts from programmer's side.

The HPC market is extremely competitive where the winner is decided based on multiple metrics such as performance on a range of applications, energy-efficiency, cost, ease of use, etc. [102–104]. A limitation of Phi is that it does not outperform CPU for serial applications/phases and also does not generally outperform GPU for parallel applications. Hence, justifying adoption of Phi may be difficult for a decision-maker. Further, the recent surge of interest in AI has placed further pressure on design of computing systems to provide high throughput for AI workloads too [105]. However, Phi versions before KNM were not optimized for AI workloads. Future computing systems need to be highly competitive on all metrics of interest, for all applications and especially AI applications.

Modern computing systems come in all sizes and shapes, ranging from handheld mobile systems to systems sitting in our lap/desk to clusters and supercomputers. Phi served only a niche market, namely HPC servers/supercomputers. By comparison, GPU which started as an accelerator for multimedia/gaming quickly appeared in multiple product-lines to cater to the demands of other market-segments also, ranging from mobile/embedded systems to HPC systems [106]. Future computing systems need to keep a strong hold in at least few segments but also spread their wings to other segments to meet all-around computing needs of users. This is certainly "easier said than done", because these segments differ greatly, not only in their design constraints but also in optimization objectives.

Compared to GPU, Phi has higher independence from the host since Phi executes its own OS in its own memory [27]. Even after compromising the host, an attacker can merely turn-off Phi but cannot leak its confidential data. Due to this, a cryptography algorithm running on Phi is insulated to attacks from host-side [27]. However, Phi (3200, 5200, 7200 series) is vulnerable to recently discovered vulnerabilities, viz. Spectre and Meltdown [107]. To use an accelerator/co-processor in mission-critical systems deployed for defense, health, finance and space, ensuring their security is extremely vital. However, research into security of accelerators receives less attention, partly because security often comes at the cost of performance. Given the immense implications of even one security breach, it is vital that designers should not just retrofit a processor for security but design for security as the first principle. This will ensure that future processors are insulated towards (at least) the known security vulnerabilities.

Acknowledgement: The author is thankful to Osho Anand and Visnu P Kumarr for their help in early stage of this work.

REFERENCES

[1] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Morgan Kaufmann, 2013.

[2] K.-C. Leung, D. Eyers, X. Tang, S. Mills, and Z. Huang, "Investigating large-scale feature matching using the Intel® Xeon Phi coprocessor," in *IVCNZ*, 2013.

[3] T. Trader, "GPUs Power Five of World's Top Seven Supercomputers," https://bit.ly/2uDMIj1, 2018.

[4] M. Bernaschi and F. Salvadore, "Multi-Kepler GPU vs. Multi-Intel MIC: A two test case performance study," in *HPCS*, 2014, pp. 1–8.

[5] C. J. Newburn *et al.*, "Offload compiler runtime for the Intel Xeon Phi coprocessor," in *IPDPSW*, 2013, pp. 1213–1225.

[6] E. Apra, M. Klemm, and K. Kowalski, "Efficient implementation of Many-body Quantum Chemical methods on the Intel® Xeon Phi coprocessor," in *SC*. IEEE, 2014, pp. 674–684.

[7] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Performance engineering for a Medical imaging Application on the Intel Xeon Phi accelerator," *ARCS*, 2014.

[8] J. Zhang and S. You, "Large-scale Geospatial Processing on Multi-core and Many-core processors: Evaluations on CPUs, GPUs and MICs," *CoRR, vol. abs/1403.0802*, 2014.

[9] I. Hadade, F. Wang, M. Carnevale, and L. di Mare, "Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures," *Computer Physics Communications*, 2018.

[10] A. Sodani *et al.*, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[11] A. Uta, A. L. Varbanescu, A. Musaafir, C. Lemaire, and A. Iosup, "Exploring HPC and big data convergence: A graph processing study on Intel Knights Landing," in *IEEE CLUSTER*, 2018, pp. 66–77.

[12] S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with Xeon Phi KNL," in *IPDPS*, 2017, pp. 297–306.

[13] A. Ramachandran *et al.*, "Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi," in *ICPP*, 2013.

[14] G. Latu, M. Haefele, J. Bigot, V. Grandgirard, T. Cartier-Michaud, and F. Rozar, "Evaluating Kernels on Xeon Phi to accelerate GYSELA application," *ESAIM Proc.*, vol. 53, pp. 211–231, 2016.

[15] S. Vialle, S. Contassot-Vivier, and P. Mercier, "Generic algorithmic scheme for 2D stencil applications on Hybrid machines," *ARCS*, pp. 115–129, 2016.

[16] V. G. Pinto, V. A. Herbstrith, and L. M. Schnorr, "Replicating the Performance Evaluation of an N-Body Application on a Manycore Accelerator," in *SBAC-PADW*. IEEE, 2015, pp. 19–24.

[17] A. Elafrou, G. Goumas, and N. Koziris, "Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi-and Many-Core Processors," in *ICPP*, 2017, pp. 292–301.

[18] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matías, "An energy-aware performance analysis of SWIMM: Smith–Waterman implementation on Intel's Multicore and Manycore architectures," *CPE*, 2015.

[19] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms," *TPDS*, vol. 29, no. 10, pp. 2176–2190, 2018.

[20] F. Wang, L. Deng, D. Zhao, and S. Li, "Acceleration of PDE-based FTLE calculations on Intel Multi-core and Many-core architectures," *ICCSNT*, 2015.

[21] L. Deng, H. Bai, D. Zhao, and F. Wang, "Kepler GPU vs. Xeon Phi: Performance case study with a high-order CFD application," in *ICCC*, 2015.

[22] W. Xue, C. Yang, H. Fu, X. Wang, Y. Xu, J. Liao, L. Gan, Y. Lu, R. Ranjan, and L. Wang, "Ultra-scalable CPU-MIC acceleration of Mesoscale Atmospheric Modeling on Tianhe-2," *IEEE TC*, vol. 64, no. 8, pp. 2382–2393, 2015.

[23] S. Jha *et al.*, "Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An experimental approach," *PVLDB*, 2015.

[24] M. Kreutzer *et al.*, "Ghost: building blocks for high performance sparse linear algebra on heterogeneous systems," *IJPP*, vol. 45, no. 5, pp. 1046–1072, 2017.

[25] H. Jeong *et al.*, "Performance of Kepler GTX Titan GPUs and Xeon Phi System," *arXiv preprint arXiv:1311.0590*, 2013.

[26] C. Chang, S. Yao, and D. Yu, "sRSA: High speed RSA on the Intel MIC architecture," in *ICPADS*. IEEE, 2015, pp. 609–616.

[27] S. Yao and D. Yu, "PhiOpenSSL: Using the Xeon Phi Coprocessor for Efficient Cryptographic Calculations," *IPDPS*, 2017.

[28] G. Chatzikonstantis, D. Rodopoulos, C. Strydis, C. I. De Zeeuw, and D. Soudris, "Optimizing extended Hodgkin-Huxley neuron model simulations for a Xeon/Xeon Phi node," *TPDS*, vol. 28, no. 9, pp. 2581–2594, 2017.

[29] B. Li *et al.*, "The power-performance tradeoffs of the Intel Xeon Phi on HPC applications," in *IPDPSW*, 2014, pp. 1448–1456.

[30] J. Fang *et al.*, "Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi." in *IWBBIO*, 2014, pp. 579–588.

[31] A. Tangherloni, M. S. Nobile, P. Cazzaniga, D. Besozzi, and G. Mauri, "Gillespie's Stochastic Simulation Algorithm on MIC coprocessors," *J SUPERCOMPUT*, 2017.

[32] K. Murano, T. Shimobaba, A. Sugiyama, N. Takada, T. Kakue, M. Oikawa, and T. Ito, "Fast computation of computer-generated hologram using Xeon Phi coprocessor," *Computer Physics Communications*, vol. 185, no. 10, pp. 2742–2757, 2014.

[33] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, "Assessing the performance of OpenMP programs on the Intel Xeon Phi," *Euro-Par*, pp. 547–558, 2013.

[34] B. Chen, R. Kantowski, X. Dai, E. Baron, and P. Van der Mark, "Accelerating Gravitational Microlensing simulations using the Xeon Phi coprocessor," *Astronomy and computing*, vol. 19, pp. 60–65, 2017.

[35] J. Langguth, Q. Lan, N. Gaur, and X. Cai, "Accelerating detailed tissue-scale 3D Cardiac simulations using Heterogeneous CPU-Xeon Phi computing," *IJPP*, vol. 45, no. 5, pp. 1236–1258, 2017.

[36] C. Rosales, "Porting to the Intel Xeon Phi: Opportunities and challenges," in *XSW*, 2013, pp. 1–7.

[37] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative performance analysis of Intel Xeon Phi, GPU, and CPU," *arXiv preprint 1311.0378*, 2013.

[38] J. Gomes, A. C. de Melo, J. Kong, T. Kurc, J. H. Saltz, and G. Teodoro, "Cooperative and out-of-core execution of the irregular wavefront propagation pattern on hybrid machines with intel xeon phi," *CPE*, vol. 30, no. 14, p. e4425, 2018.

[39] T. Rechkalov and M. Zymbler, "Accelerating Medoids-based Clustering with the Intel Many Integrated Core architecture," in *AICT*, 2015.

[40] X. Wang, W. Xue, J. Zhai, Y. Xu, W. Zheng, and H. Lin, "A fast tridiagonal solver for Intel MIC architecture," in *IPDPS*. IEEE, 2016, pp. 172–181.

[41] D. Guttman, M. T. Kandemir, M. Arunachalamy, and V. Calina, "Performance and energy evaluation of data prefetching on Intel Xeon Phi," in *ISPASS*, 2015, pp. 288–297.

[42] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Optimization of lattice Boltzmann simulations on heterogeneous computers," *IJHPCA*, vol. 33, no. 1, pp. 124–139, 2019.

[43] C. R. Ferreira and M. Bader, "Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors," in *PASC*. ACM, 2017, p. 12.

[44] J. Mielikainen, B. Huang, and H.-L. A. Huang, "Optimizing Purdue-Lin Microphysics Scheme for Intel Xeon Phi Coprocessor," *JSTARS*, 2016.

[45] A. Viebke and S. Pllana, "The potential of the Intel Xeon Phi for supervised deep learning," *HPCC*, pp. 758–765, 2015.

[46] B. Plazolles, D. El Baz, M. Spel, V. Rivola, and P. Gegout, "Parallel Monte-Carlo Simulations on GPU and Xeon Phi for Stratospheric Balloon Envelope Drift Descent Analysis," in *UIC*, 2016, pp. 611–619.

[47] K. Halbiniak, R. Wyrzykowski, L. Szustak, and T. Olas, "Assessment of Offload-Based Programming Environments for Hybrid CPU-MIC Platforms in Numerical Modeling of Solidification," *SMPT*, 2018.

[48] P. Stpiczyński, "Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1461–1472, 2018.

[49] R. Nozal, B. Perez, J. L. Bosque, and R. Beivide, "Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels," *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1123–1136, 2019.

[50] D. Ding, M. Wen, S. Zhou, M. Chen, and J. Lin, "Optimization and Evaluation of VLPL-S Particle-in-cell Code on Knights Landing," *HPC China*, 2016.

[51] J. Domke, K. Matsumura, M. Wahib, H. Zhang, K. Yashima, T. Tsuchikawa, Y. Tsuji, A. Podobas, and S. Matsuoka, "Double-precision FPUs in High-Performance Computing: an Embarrassment of Riches?" *arXiv preprint arXiv:1810.09330*, 2018.

[52] M. A. Al Farhan and D. Keyes, "Optimizations of unstructured aerodynamics computations for many-core architectures,"

IEEE Transactions on Parallel and Distributed Systems, 2018.
[53] J. M. Herruzo, S. G. Navarro, P. Ibánez, V. V. Yufera, J. Alastruey, and O. Plata, "Accelerating sequence alignments based on fm-index using the intel knl processor," *IEEE/ACM transactions on computational biology and bioinformatics*, 2018.
[54] T. Barnes *et al.*, "Evaluating and optimizing the nersc workload on knights landing," in *PMBS*, 2016, pp. 43–53.
[55] D. Khaldi and B. Chapman, "Towards automatic HBM allocation using LLVM: a case study with knights landing," in *LLVM-HPC*, 2016, pp. 12–20.
[56] Y. Gao and W.-M. Chen, "Family Relationship Inference Using Knights Landing Platform," in *CSCloud*, 2017, pp. 27–30.
[57] F. Robertsén, K. Mattila, and J. Westerholm, "High-performance simd implementation of the lattice-boltzmann method on the xeon phi processor," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 13, p. e5072, 2019.
[58] P. Madhavan, P. Young, and S. Chang, "Performance of scientific simulations on qct developer cloud: A case study of molecular dynamic and quantum chemistry simulations," in *CSCloud*, 2017, pp. 18–21.
[59] I. Surmin *et al.*, "Co-design of a Particle-in-cell Plasma Simulation code for Intel Xeon Phi: a first look at Knights Landing," in *ICA3PP*. Springer, 2016, pp. 319–329.
[60] J.-H. Kang *et al.*, "Performance Evaluation of Scientific Applications on Intel Xeon Phi Knights Landing Clusters," in *HPCS*, 2018, pp. 338–341.
[61] M. Jacquelin, W. De Jong, and E. Bylaska, "Towards highly scalable Ab initio molecular dynamics (AIMD) simulations on the Intel Knights Landing manycore processor," in *IPDPS*, 2017, pp. 234–243.
[62] N. A. Gawande, J. A. Daily, C. Siegel, N. R. Tallent, and A. Vishnu, "Scaling deep learning workloads: NVIDIA DGX-1/Pascal and intel knights landing," *Future Generation Computer Systems*, 2018.
[63] M. Wagner *et al.*, "Performance Analysis and Optimization of the FFTXlib on the Intel Knights Landing Architecture," in *ICPPW*, 2017, pp. 243–250.
[64] L. Chen *et al.*, "Benchmarking Harp-DAAL: High Performance Hadoop on KNL Clusters," in *Intl. Conf. on Cloud Computing (CLOUD)*, 2017, pp. 82–89.
[65] A. Breuer, A. Heinecke, and Y. Cui, "Tensor-optimized hardware accelerates fused discontinuous galerkin simulations," *SC*, 2018.
[66] I. Kanamori and H. Matsufuru, "Practical Implementation of Lattice QCD Simulation on Intel Xeon Phi Knights Landing," in *IEEE CANDAR*, 2017, pp. 375–381.
[67] K. Komatsu, T. Kishitani, M. Sato, A. Musa, and H. Kobayashi, "Search Space Reduction for Parameter Tuning of a Tsunami Simulation on the Intel Knights Landing Processor," in *IEEE MCSoC*, 2018, pp. 117–124.
[68] D. Das *et al.*, "Mixed precision training of convolutional neural networks using integer operations," *arXiv preprint arXiv:1802.00930*, 2018.
[69] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on SIMD architectures," in *SC*, 2018, p. 66.
[70] "Compare Intel Products," https://intel.ly/2AI9lrP, 2018.
[71] https://www.techpowerup.com/gpu-specs/titan-v.c3051.
[72] S. Mittal and J. Vetter, "A Survey Of Techniques for Architecting DRAM Caches," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 6, pp. 1852–1863, 2016.
[73] D. Bradford, S. Chinthamani, J. Corbal, A. Hassan, K. Janik, and N. Ali, "Knights Mill: New Intel Processor for Machine Learning," https://bit.ly/2AI1yrJ, 2017.
[74] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.
[75] S. Mittal and S. Vaishay, "A Survey of Techniques for Optimizing Deep Learning on GPUs," *Journal of Systems Architecture*, 2019.
[76] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Preliminary experiences with the Uintah Framework on Intel Xeon Phi and Stampede," in *XSEDE*, 2013, p. 48.
[77] F. Affinito, "Introduction to Intel Xeon Phi programming," https://bit.ly/2vBRTj8, 2015.
[78] "User and Reference Guide for the Intel C++ Compiler 15.0," https://software.intel.com/en-us/node/522518, 2015.
[79] X. Huo, B. Ren, and G. Agrawal, "A programming system for Xeon Phis with runtime SIMD parallelization," in *International Conference on Supercomputing*, 2014, pp. 283–292.
[80] S. Mittal, "A survey of recent prefetching techniques for processor caches," *CSUR*, 2016.
[81] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM CSUR*, 2015.
[82] S. J. Pennycook *et al.*, "Exploring SIMD for Molecular Dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors," in *IPDPS*, 2013, pp. 1085–1097.
[83] M. Greenfield, "Migrating Applications from Knights Corner to Knights Landing Self-Boot Platforms," https://intel.ly/2FL700p, 2016.
[84] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving Intel Xeon Phi," *ICPE*, 2014.
[85] P. Jiang and G. Agrawal, "Conflict-free Vectorization of Associative Irregular applications with recent SIMD architectural advances," in *CGO*. ACM, 2018, pp. 175–187.
[86] H. Shan, S. Williams, and C. W. Johnson, "Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression," *PMBS*, 2018.
[87] G. Crimi *et al.*, "Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi co-processor," *Procedia Computer Science*, vol. 18, pp. 551–560, 2013.
[88] K. Hou, H. Wang, and W.-c. Feng, "Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors," in *Intl. Conf. on Supercomputing*. ACM, 2015, pp. 383–392.
[89] S. Mittal, "A survey of cache bypassing techniques," *JLPEA*, 2016.
[90] E. Calore *et al.*, "Early experience on using Knights Landing processors for Lattice Boltzmann applications," in *International Conference on Parallel Processing and Applied Mathematics*, 2017, pp. 519–530.

[91] M. Lu *et al.*, "Optimizing the MapReduce Framework on Intel Xeon Phi coprocessor," in *Intl. Conf. on Big Data*. IEEE, 2013.
[92] M. Valmiki *et al.*, "Behavior of MDynaMix on Intel Xeon Phi Coprocessor," in *IEEE AIMS*, 2013, pp. 387–392.
[93] M. Salim, A. O. Akkirman, M. Hidayetoglu, and L. Gurel, "Comparative benchmarking: Matrix multiplication on a Multicore coprocessor and a GPU," in *CEM*, 2015.
[94] S. A. Mirsoleimani, A. Plaat, J. Vermaseren, and J. v. d. Herik, "Performance analysis of a 240 thread tournament level MCTS Go program on the Intel Xeon Phi," *arXiv preprint arXiv:1409.4297*, 2014.
[95] H. Lan, W. Liu, B. Schmidt, and B. Wang, "Accelerating large-scale Biological Database Search on Xeon Phi-based neo-heterogeneous Architectures," in *BIBM*. IEEE, 2015, pp. 503–510.
[96] P. Fernández *et al.*, "Parallelizing and Optimizing LHCb-Kalman for Intel Xeon Phi KNL Processors," in *PDP*, 2018, pp. 741–750.
[97] S. Rho, G. Park, J.-S. Kim, S. Kim, and D. Nam, "A Study on Optimal Scheduling Using High-Bandwidth Memory of Knights Landing Processor," in *(FAS*W)*, 2017, pp. 289–294.
[98] S. Mittal, "A Survey of Techniques for Architecting TLBs," *Concurrency and Computation: Practice and Experience*, 2017.
[99] M. Dixon, D. Klabjan, and J. H. Bang, "Implementing Deep Neural Networks for Financial Market Prediction on the Intel Xeon Phi," *WHPCF*, 2015.
[100] K. Krommydas *et al.*, "On the portability of the OpenCL Dwarfs on fixed and reconfigurable parallel platforms," in *ICPADS*, 2013, pp. 432–433.
[101] Y. Liu and B. Schmidt, "SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors," *ASAP*, pp. 184–185, 2014.
[102] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM CSUR*, 2015.
[103] A. Lastovetsky and R. R. Manumachu, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2016.
[104] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 160–177, 2017.
[105] S. Mittal, "A Survey of FPGA-based Accelerators for Convolutional Neural Networks," *Neural computing and applications*, 2018.
[106] S. Mittal, "A Survey on Optimized Implementation of Deep Learning Models on the NVIDIA Jetson Platform," *Journal of Systems Architecture*, 2019.
[107] "Facts about Security Research and Intel Products," https://intel.ly/2EU2sBS, 2018.

View publication stats