MrPhi: An Optimized MapReduce Framework on Intel Xeon Phi Coprocessors

Mian Lu, Yun Liang, Huynh Phung Huynh, Zhongliang Ong, Bingsheng He, and Rick Siow Mong Goh

Abstract—In this work, we develop *MrPhi*, an optimized MapReduce framework on a heterogeneous computing platform, particularly equipped with multiple Intel Xeon Phi coprocessors. To the best of our knowledge, this is the first work to optimize the MapReduce framework on the Xeon Phi. We first focus on employing advanced features of the Xeon Phi to achieve high performance on a single coprocessor. We propose a vectorization friendly technique and SIMD hash computation algorithms to utilize the SIMD vectors. Then we pipeline the map and reduce phases to improve the resource utilization. Furthermore, we eliminate multiple local arrays but use low cost atomic operations on the global array to improve the thread scalability. For a given application, our framework is able to automatically detect suitable techniques to apply. Moreover, we extend our framework to a heterogeneous platform to utilize all hardware resource effectively. We adopt non-blocking data transfer to hide the communication overhead. We also adopt aligned memory transfer in order to fully utilize the PCIe bandwidth between the host and coprocessor. We conduct comprehensive experiments to benchmark the Xeon Phi and compare our optimized MapReduce framework with a state-of-the-art multi-core based MapReduce framework (Phoenix++). By evaluating six real-world applications, the experimental results show that our optimized framework is 1.2 to 38× faster than Phoenix++ for various applications on a single Xeon Phi. Additionally, the performance of four applications is able to achieve linear scalability on a platform equipped with up to four Xeon Phi coprocessors.

Index Terms—Xeon Phi, Intel Many Integrated Core architecture (MIC), coprocessors, MapReduce, parallel programming, high performance computing, heterogeneous computing

1 Introduction

Nowadays, many applications require high performance computing. We have witnessed the success of coprocessors in speeding up applications, such as graphics processors (GPUs) [1], [2], [3]. In order to fully utilize the capability of those architectures, developers either use coprocessor specific programming languages (such as CUDA [4]) or adopt some simplified general-purpose parallel programming frameworks, such as MapReduce [5]. In this work, we follow the second direction to develop a MapReduce framework on state-of-the-art coprocessors.

Intel has released the x86 accelerator named Xeon Phi. It offers a much larger number of cores than conventional CPUs, while its architectural design is based on x86. Particularly, an Intel Xeon Phi coprocessor 5110P integrates 60 cores on a chip, with four hardware threads per core. The thread execution on the Xeon Phi does not suffer from the branch divergence problem. Different threads are able to execute different progarm paths without significant overhead (heterogeneous threads). Furthermore, it highlights the 512-bit width vector processing units (VPUs) for powerful SIMD processing. Besides, L2 caches are fully coherent through ring-based interconnection. It also provides low cost atomic operations. However, as designed as a coprocessor, the Xeon Phi has limited main memory (8 GB for our evaluated product Xeon Phi 5110P).

While Xeon Phi has been just released, it has already demonstrated its promising adoptions. A number of studies have demonstrated its performance advantage [6], [7], [8], [9], [10]. The supercomputer STAMPEDE [11] and Tianhe-2 [12] also have equipped the Xeon Phi coprocessors to unlock its hardware capability for scientific computing. Instead of optimizing individual applications like previous studies [6], [7], we investigate a productivity programming framework to facilitate users to implement data analytics tasks correctly, efficiently, and easily on Xeon Phi.

MapReduce [5] is a popular programming framework for parallel or distributed computing. It was originally proposed by Google for simplified parallel programming on a large number of machines. Users only need to define *map* and *reduce* functions according to their application logics. The MapReduce runtime automatically distributes and executes the task on multiple machines [5] or multiple processors in a single machine [13], or GPUs [14]. Thus, this framework reduces the complexity of parallel programming and the users only need to focus on the sequential implementations of map and reduce functions.

As Xeon Phi is based on the x86 architecture, one might suggest adopting existing state-of-the-art multi-core based MapReduce frameworks, such as Phoenix++ [15]. However, we find that Phoenix++ cannot fully utilize the hardware capability of Xeon Phi as Phoenix++ is not aware of the advanced hardware features of Xeon Phi. First, Phoenix++ pays little attention to utilize VPUs, which is critical for the performance on the Xeon Phi [6], [7]. Second, Phoenix++

- *M. Lu, H. P. Huynh, Z. Ong, and R.S.M. Goh are with the Institute of High Performance Computing, A*STAR, Singapore. E-mail: {lum, huynhph, ongzl, gohsm}@ihpc.a-star.edu.sg.*
- *Y. Liang is with the School of EECS, Peking University, Beijing, China. E-mail: ericlyun@pku.edu.cn.*
- *B. He is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: bshe@ntu.edu.sg.*

Manuscript received 24 Mar. 2014; revised 14 Sept. 2014; accepted 22 Oct. 2014. Date of publication 28 Oct. 2014; date of current version 7 Oct. 2015. Recommended for acceptance by Y. Liu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2365784

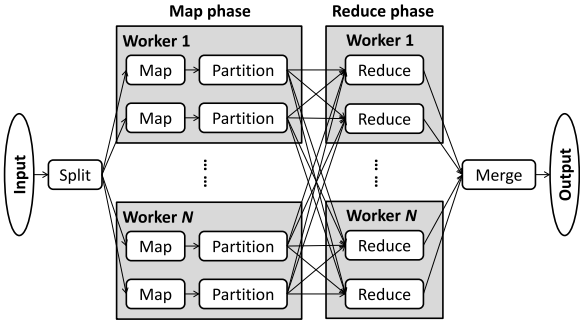


Fig. 1. The basic workflow of a MapReduce framework.

TABLE 1
Specification of an Intel Xeon Phi 5110P

Cores	60 x86 cores
Threads	4 hardware threads per core
Frequency	1.05 GHz per core
Memory size	8 GB in total
L1 cache	32 KB data + 32 KB instructions per core
L2 cache	512 KB per core
Vector processing unit	32 512-bit vector registers per core
Peak performance	1.01 TFLOPS of double precision computation

has high average memory access latency due to the relatively small L2 cache (512 KB) per core. Third, the relatively small memory capacity (8 GB) may limit the thread scalability. Unawareness of those hardware features results in significant performance loss, as we demonstrated in Section 5. On the other hand, nowadays, a heterogeneous computing platform equipped with multiple accelerators, e.g. Xeon Phi coprocessors, is common. Unfortunately, Phoenix++ is unable to be deployed on such a platform to utilize all resource.

To address the above-mentioned deficiencies as well as fully utilize Xeon Phi hardware capabilities, we develop *MrPhi* (pronounced as *Mr. Phi*), the first optimized **M**ap**R**educe framework on Xeon *Phi* with following contributions.

- Our framework is highly optimized to utilize the Xeon Phi's hardware features, including four major techniques:
  - Vectorization friendly map
  - SIMD hash algorithms
  - Pipelining for map and reduce phases
  - Eliminating local arrays.
- Our framework is able to run on a heterogeneous platform equipped with multiple Xeon Phi coprocessors. To efficiently utilize all hardware resources on such a platform, we adopt:
  - Dynamic scheduling
  - Non-blocking data transfer
  - Aligned memory transfer.

The rest of the paper is organized as follows. We introduce the background in Section 2. Section 3 gives detailed implementation on a single Xeon Phi coprocessor. The design on a heterogeneous platform is described in Section 4. The experimental results are presented in Section 5. We conclude this paper in Section 6.

2 Background

In this section, we first introduce the MapReduce framework and Xeon Phi coprocessor. Then we identify the challenges of developing the MapReduce framework on the Xeon Phi.

2.1 MapReduce Framework

MapReduce is a popular framework for simplified parallel programming. We briefly introduce its programming model and workflow in this section.

Programming model. The input of a MapReduce job is specified by users, usually in the form of an array. The output is a set of *key-value* pairs. A user specifies a MapReduce job mainly by two functions, which are *map* and *reduce*. With the user-defined functions, a MapReduce framework first applies the map function to every element in the input array and generates a set of intermediate key-value pairs (*map phase*). After the map phase, the reduce function is applied to all intermediate pairs with the same key and generates another set of result key-value pairs (*reduce phase*). Finally, the result key-value pairs are ordered (optional) and then output. The detailed programming model is presented in the original MapReduce paper [5].

MapReduce workflow. The MapReduce framework was originally designed for distributed computing [5]. Later, it was extended to other architectures such as multi-core CPUs [13], [15], [16], [17], GPUs [14], [18], [19], the coupled CPU-GPU architecture [20], FPGA [21] and Cell processors [22]. These different MapReduce frameworks share the common basic workflow, but differ in detailed implementation.

Fig. 1 illustrates the workflow of a MapReduce framework. At the beginning, a *split* function divides the input data across *workers*. On multi-core CPUs, a worker is handled by one thread. A worker usually needs to process multiple input elements. Thus the *map* function is applied to the input elements one by one. Such an operation of applying the map function for an input element is called a *map operation*. Each map operation produces intermediate key-value pairs. Then a *partition* function is applied to these key-value pairs. Then in the reduce phase, each *reduce operation* applies the reduce function to a set of intermediate pairs with the same key. Finally the results from multiple reduce workers are merged and output.

Compared with existing work, our MrPhi [23] is the first work for optimizing MapReduce on Xeon Phi. We propose specific optimization techniques that have not been thoroughly studied on other architectures before, such as the vectorization friendly map. On the other hand, this paper extends the design of our previous MrPhi to a platform equipped with multiple Xeon Phi coprocessors. Our framework is able to run on either a heterogeneous computing server with one host and multiple accelerators or a cluster, as long as MPI communication is supported.

2.2 Intel Xeon Phi Coprocessor

Intel Xeon Phi coprocessor was recently released in November 2012. The Xeon Phi is based on the Intel Many Integrated Core Architecture (MIC). In this work, we conduct our experiments on the Xeon Phi 5110P. Table 1 summarizes its hardware features. Note that a Xeon Phi coprocessor

should be attached to a host (conventional CPUs). Therefore the host and Xeon Phi have their own separate memory spaces with communication via the PCIe bus using specialized programming interfaces, e.g., MPI.

Xeon Phi coprocessor is relatively new to the HPC community, but has already attracted a number of researchers. As one of the most important HPC benchmark, Linpack has been ported and optimized on the Xeon Phi [7]. Similarly, another fundamental HPC algorithm SpMV has been carefully studied as well [8]. Moreover, Xeon Phi has been exploited to accelerate other scientific applications, such as molecular dynamics [6], FFT [9] and simulation of the critical Ising model [10]. Supercomputers, such as STAMPEDE [11] and Tianhe-2 [12] have also been equipped with Xeon Phi coprocessors to improve their performance.

Compared with conventional Xeon CPUs, Xeon Phi has a few unique features, such as 512-bit SIMD vectors and ring-based coherent L2 caches. It is important to understand these features in order to achieve high-performance on Xeon Phi. In the following, we briefly introduce the major features of the Xeon Phi.

512-bit vector processing units. Xeon Phi features wide 512-bit VPUs on each core. It doubles the vector width compared with the latest Intel Xeon CPU. Furthermore, it provides new SIMD primitives, such as scatter/gather. Therefore, utilizing VPUs effectively is the key to deliver high performance. The VPUs can be either exploited by manual implementations using SIMD instructions or *auto-vectorization* by the Intel compiler. The auto-vectorization tries to identify loops that can be vectorized to use SIMD VPUs at compilation time.

Multiple Instruction, Multiple Data (MIMD) Massive thread parallelism. Each core of the Xeon Phi supports up to four hardware hyper-threads. Thus, there are 240 threads in total. The MIMD thread execution allows different threads to execute different instructions at any time. Thus, we can assign different workloads to different threads to improve the hardware resource utilization.

Coherent L2 caches with ring interconnection. Each core on Xeon Phi has a 512 KB L2 unified cache. All the caches on the 60 cores are fully coherent. They are interconnected by a 512-bit wide bidirectional ring bus. If a L2 cache miss occurs on a core, the requests are forwarded to the caches on other cores via the ring network. If there is a cache on other cores containing the data, the data will be transferred to the current core via the ring. Compared with the main memory accesses caused by cache misses, data forwarding via the ring network is less expensive.

Low cost atomic operations in DRAM. Atomic data types are well supported on the Xeon Phi. When the conflict rate of data accesses is low and the data is stored in DRAM, the operations on atomic data types do not have significant overhead. Therefore, it is reasonable to exploit atomic operations when the conflict rate is low and memory accesses are random.

However, the memory size of the Xeon Phi is relatively small, which is only 8 GB for our Xeon Phi 5110P. This may become a bottleneck for some applications. We demonstrate such an issue for particular applications and use low cost atomic operations to address it in Section 3.5.

2.3 Challenges of a Shared Memory MapReduce Framework on Xeon Phi

State-of-the-art shared memory MapReduce frameworks on multi-core platforms (Phoenix++ [15]) are designed to have flexible intermediate key-value storage containers and effective combiner implementation. These techniques reduce memory storage requirement and traffic. However, we have identified three major performance issues of Phoenix++ when porting it onto Xeon Phi.

- *Poor VPU usage*. Phoenix++ takes little advantage of the VPUs on Xeon Phi. The compiler is unable to vectorize the code effectively. This suggests we should either rewrite the code in a suitable way to assist the auto-vectorization or manually re-implement algorithms using SIMD intrinsics.
- *High average latency of memory accesses*. Key-value pairs are hold by certain data structures, such as hash table. There are a large number of random memory accesses on such a data structure. As the local L2 cache per core on the Xeon Phi is small (512 KB per core), this results in high cache misses and high average latency of a memory access.
- *Limited memory size*. Due to the limited memory (8 GB) on the Xeon Phi, we find that Phoenix++ is unable to handle the array container efficiently when the array is large. Specifically, not all threads are able to be utilized in this case.

As a result, running Phoenix++ directly on the Xeon Phi does not give good performance. In our framework, we propose various techniques to address these performance issues.

3 Optimized MapReduce on Xeon Phi

In this section, we present our optimized MapReduce framework **MrPhi** for the Xeon Phi. We focus on the optimization on a single Xeon Phi in this section.

3.1 Overview

MrPhi adopts state-of-the-art techniques from the shared memory MapReduce framework as well as specific optimizations for the Xeon Phi coprocessor. Overall, we adopt the Phoenix++'s design as shown in the paper [15] to implement the basic MapReduce workflow as shown in Fig. 1. There are two major techniques taken from Phoenix++ adopted in our framework, which are efficient combiners and different container structures. We briefly introduce the two techniques and refer readers to the original Phoenix++ paper [15] for more details.

- *Efficient combiners*. Each map worker maintains a local container. When an intermediate key-value pair is generated by a map function, the reduce operator is immediately applied to that pair based on the local container. This step is performed using a *combiner*. Therefore, a map operation in fact consists of two components, which are the computation defined in the map function and combiner execution. After that, the local results stored in each local container are merged to a global container in the reduce phase. Our MrPhi adopts this similar design but

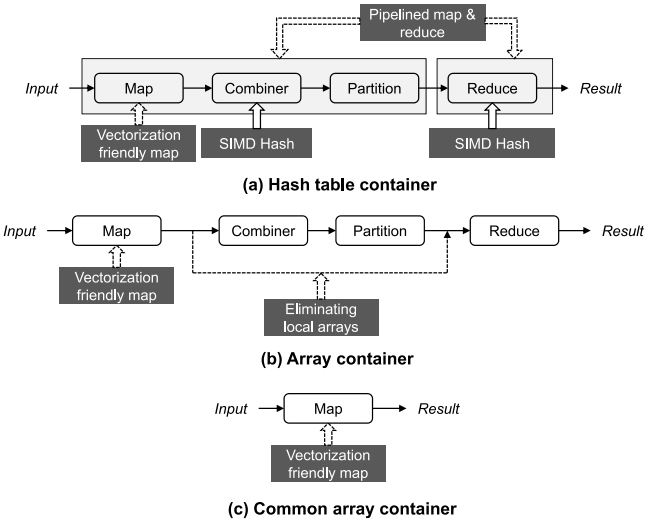


Fig. 2. Proposed techniques (in dark box) and their applicability in MrPhi for three different containers.

with particular improvements (Section 3.5). Note that both local and global containers are stored in the main memory of Xeon Phi.

- *Different container structures*. MrPhi supports all three data structures for containers implemented in Phoenix++, which are *hash table*, *array* and *common array*.The array container is efficient when the keys are integers and in a fixed range. The common array container is designed for the application that generates exactly one key-value pair per input element. The major difference between the array and common array is that there is no local container and combiner for the common array. Results in the common array are directly written out by each map worker to a global array without any conflicts. The hash table container is used when the array and common array containers are unavailable.

More importantly, we propose four optimization techniques specifically for Xeon Phi, which are *vectorization friendly map phase*, *SIMD hash*, *pipelined map and reduce*, and *eliminating local arrays*. These four techniques are not always applicable to all containers in all cases. Fig. 2 shows how these optimization techniques fit into the different components of the MapReduce framework with different containers. We introduce these techniques in details in this section.

- *Vectorization friendly map phase.* MrPhi implements the map phase in a vectorization friendly way, which clears the dependency between map operations. By doing this, the Intel compiler is able to automatically vectorize multiple map operations to take advantage of VPUs successfully.
- *SIMD parallelism for hash computation.* Hash computation is parallelized by SIMD intrinsics.
- *Pipelined execution for map and reduce phases.* In general, the user-defined map function contains heavy computation workload, while the reduce function has many memory accesses [15]. In order to better utilize the hardware resource with hyperthreading, we pipeline the map and reduce phases based on the MIMD thread execution.
- *Eliminating local arrays.* For the array container, if the array is large, it will introduce a number of performance issues due to the local arrays. We address these issues by eliminating local arrays but employing low overhead atomic operations on the global array.

Note that these techniques are not applicable for all cases and may introduce overhead. Our framework is able to either automatically detect whether a specific technique is applicable or provides helpful suggestions to users at compilation time.

3.2 Vectorization Friendly Map Phase

Utilizing VPUs is critical to high performance on the Xeon Phi. For the MapReduce framework itself, except the hash computation (presented in Section 3.3), there is little chance to employ SIMD instructions. However the user-defined map function has potential for auto-vectorization regardless of the usage of loops within the function. For the case of functions containing loops, we leave the compiler to identify opportunities for auto-vectorization within the map function. Our main focus is on the challenge of vectorizing map functions that do not contain loops.

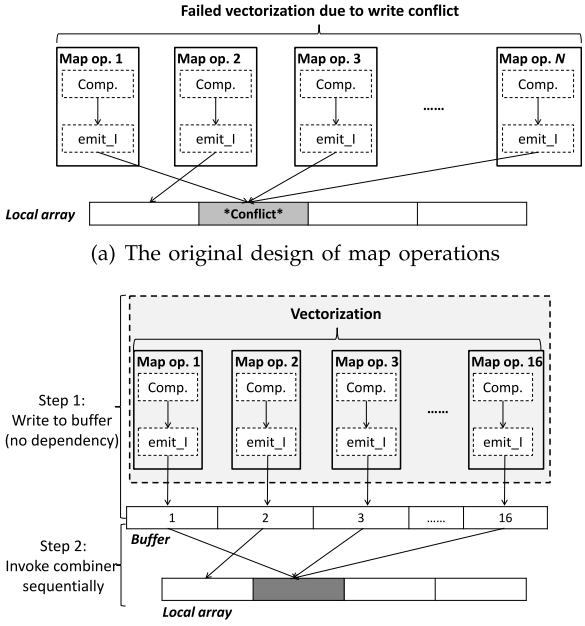
Recall that in the map phase, each thread processes multiple map operations. We use directives to guide the compiler to vectorize multiple map operations. Listing 1 shows the basic idea. *emit_intermediate* is a system-defined function to perform combiners. The *#pragma ivdep* (line 3) tells the compiler to vectorize this for-loop if there is no dependency.

Listing 1. Vectorization for multiple map operations.

```
1 //N: the number of map operations in the worker
2 //elems: the input array
3 #pragma ivdep
4 for (i = 0; i < N; i++) {
5     //the inlined map function
6     map(data_t elems[i]){
7         ... //some computation
8         emit_intermediate(key, value);
9     ...
10  }
11 }
```

This auto-vectorization can be effective if there is no dependency among map operations (from line 6 to 10 in Listing 1). However, in Phoenix++, if multiple *emit_intermediate* operations are performed concurrently, the execution will cause the conflict on a local container. This conflict exists for both array and hash table containers. Fig. 3a illustrates an example where map operations fail to be vectorized due to the dependencies between multiple *emit_intermediate* operations for an array container.

We propose a vectorization friendly technique to address this issue. Instead of performing the combiner for each intermediate pair generated by *emit_intermediate* immediately, we buffer a number of pairs. Writing to the buffer is independent for each map operation. When the buffer is full, we call the combiner for those pairs sequentially. Fig. 3b demonstrates this vectorization friendly map. Our vectorization friendly map clears the dependency among map operations and thus auto-vectorization by the compiler is possible.



(a) The original design of map operations

(b) Vectoriation friendly map

Fig. 3. Comparison of different designs for the map phase.

The vectorization friendly map is useful when the multiple map operations can be vectorized (no loop in the map function). On the other hand, this technique introduces overhead due to the temporary buffer. Therefore, if this technique is enabled but the map operations cannot be vectorized, it will hurt the performance. Since we rely on the compiler to enable vectorization, the framework itself does not know whether the map operations will be vectorized until the compilation for a specific application. If the map operations can be vectorized based on the printout from Intel compiler about vectorization eligibility, then it is worthwhile to adopt this technique. Due to the clear output by the compiler and our clean design of the interface, it is straightforwards for users to enable or disable this technique.

3.3 SIMD Parallelism for Hash Computation

Hash computation is a key component in the MapReduce framework as well as a fundamental building block for many other applications, such as database and encryption systems. We observed that the auto-vectorization often failed due to the complex logic for hash computation. Thus, we chose to manually implement the hash computation using SIMD instructions.

SIMD hash computation for native data types, such as integer and float, is straightforward. The same procedure is applied to different input elements, which fully employs the SIMD feature. However, it is challenging to process variable-sized data types, such as text strings. Overall, various hash functions for strings, such as bkdr [24], FNV [25] and djb2 [26], have the similar workflow, which processes characters one by one. Listing 2 illustrates the code of *bkdr* hash, which is used in our Word Count and Reverse Index applications. The challenge lies in efficiently handling different input words with variable lengths. Furthermore, SIMD instructions only can be applied to special VPU vector registers. Yet another challenge lies in efficiently packing the string data from memory into these vector registers.

Iteration	1	2	3	4	5	6	7	8	9
SIMD Unit 1	T	h	i	s	\0	P	h	i	\0
SIMD Unit 2	i	s	\0	X	e	o	n	\0	

Fig. 4. The workflow of SIMDH-Stream.

Listing 2. The code of bkdr hash functions for strings

```
1 int hash (char* str) {
2     int s = 131;
3     int v = 0;
4     while (*str) v = v*s + (*str++);
5     return v& 0x7FFFFFFF;
6 }
```

We propose two SIMD hash computation algorithms. The first one is easy to implement and fully takes advantage of SIMD scatter/gather, but may lead to low SIMD hardware utilization. The second one improves the SIMD hardware utilization but at expense of high control flow overhead. In the following, we name these two implementations as *SIMDH-Padding* and *SIMDH-Stream*.

3.3.1 SIMDH-Padding

It contains multiple rounds and each round processes characters from 16 consecutive strings in parallel. The intuition is that within each round, we treat 16 strings as equal-length strings with the length L_r. L_r is equal to the number of characters in the longest string among the 16 strings. If a string is shorter than L_r, we pad this string with empty characters. Note that this padding is implemented using masks for efficiency.

SIMDH-Padding has low control overhead due to its simplicity. It takes full advantage of SIMD instructions and is also able to utilize the SIMD gather for data packing. However, it underutilizes the computation resource due to the padding of empty characters.

3.3.2 SIMDH-Stream

This algorithm does not divide strings to groups for different rounds. Instead, we continuously feed the SIMD units with strings. We treat the input strings as a stream. In Fig. 4, the zero ($\backslash 0$) denotes the end of a string. Suppose we process two strings (16 strings in practice) in parallel using two SIMD units. The input array contains four strings: "This", "is", "Xeon", "Phi". First, we start to process words "This" and "is". In the third iteration, the word "is" in the second unit has been fully processed. Then we immediately pack the first character "X" from next word "Xeon" into the second unit and continue the process. Then in the fifth iteration, the word "This" has been fully processed in the first unit. We pack the first character "P" from the word "Phi" into the first unit.

For the data packing, we adopt a prefix-sum based method as well as utilizing SIMD primitives. First, we use one SIMD instruction to find which characters are zero in the vector and then store the result in a mask vector. Then we perform prefix-sum on the mask vector to obtain the

index of next strings to process. Finally, the SIMD gather is used to collect the characters. In order to allow fast prefix-sum computation we use a lookup table. Since each vector contains 16 32-bit elements, we store the prefix-sum result for eight elements instead of 16 elements for space efficiency. As a result, a lookup table of 2^8 entries with the size of around 2 KB is used, which can fit into the cache.

Compared with SIMDH-Padding, SIMDH-Stream introduces more complex control flow for data packing. SIMDH-Padding simply checks whether the 16 packed characters are all equal to zero in each iteration. If it is true, then we pack all first characters of the next 16 strings to the vector. These actions can be finished by two SIMD instructions (one comparison and one gather). However, in SIMDH-Stream, we check characters in the vector one by one for each iteration. If a character is zero, then we load the first character of the next unprocessed string to the vector. There is no direct SIMD instruction for this process. However, SIMDH-Stream does not waste computing resources on empty characters. We evaluate these two algorithms in Section 5.2.

3.4 Pipelined Map and Reduce Phases

Pipelined map and reduce has been adopted in the MapReduce framework for distributed computing [27]. Condie et al. show that since the intermediate data is delivered to downstream operators more promptly, it is able to improve resource utilization. We propose to pipeline map and reduce phases on the Xeon Phi based on the MIMD thread execution. The primary purpose is to overlap the computation-intensive and memory-intensive workloads to improve the resource utilization. MapReduce workloads are an ideal candidate for pipelining as the user-defined map functions are usually computation-intensive, while the reduce phase to construct the global container is memory-intensive. This technique is more effective for the hash table container than the array container. Because the time taken by the reduce phase using the array container is usually too short for any advantage to be obtained from pipelining.

We design a producer-consumer model to pipeline the map and reduce phases. There are three major data structures, which are local hash tables, a global hash table, and partition queues. Specifically, each map worker has a local hash table. The local table works on a pre-allocated fixed-size small buffer, e.g., smaller than the L2 cache, in order to improve the cache efficiency. When a local hash table is full, key-value pairs stored in this table are partitioned and pushed into corresponding queues. Meanwhile, one reduce worker works on one queue to merge the key-value pairs to the final global hash table.

If the final global hash table is very small, e.g., smaller than the L2 cache, the non-pipelined model will be more efficient. The major reason is the reduce phase will be too short to take advantage of pipelining because of the small hash table. On the other hand, the pipelined model introduces storage overhead. Our producer-consumer model is adaptable to this case. Recall that we allocate a fixed-size buffer (smaller than the L2 cache) for the local hash table. If the final hash table is smaller than this buffer, no data will be fed to the reduce worker (the consumer) until the map phase is finished. This way, our pipelined model essentially degrades to a non-pipelined model as we expected.

3.5 Eliminating Local Arrays

Recall that in order to support efficient combiners, Phoenix++ uses a local container for each worker in the map phase. Then the local containers are merged in the reduce phase for the final result. Such local containers are adopted for the hash table and array container (the common array directly writes their result to a global array without any conflicts). This design is efficient when the container size is small. However, it will introduce performance issues when the container becomes large. An alternative is to eliminate local containers and directly update the global container with low-cost atomic operations for combiners when the container size is large.

The technique of eliminating local arrays is applied to the array container, because the atomic data types only support basic arithmetics while the hash table usually requires more complex data types and operations, such as text strings and memory allocation.

Based on the low overhead of atomic data types on the Xeon Phi, using the global array directly is more efficient when the array becomes large. There are two major advantages.

Thread scalability. Due to the relatively small memory size on the Xeon Phi (8 GB), the thread scalability can be limited when using local arrays. Note that the local array is allocated in the memory of Xeon Phi. Suppose the local array size is L bytes, and the available memory is M bytes, then the maximum number of concurrent threads for the map phase is $\lfloor \frac{M}{L} \rfloor$. As an extreme example of using Bloom filter in bioinformatics [28] (evaluated in Section 5), if the whole human genome is used, the local array size is around 3.7 GB. In such a case, only two threads can be used on the Xeon Phi employing local arrays.

Cache efficiency. If the array is small enough to fit into the L2 cache, using local arrays has good cache efficiency. However, when the array becomes large, random memory accesses on local arrays have poor data locality. Eliminating the local arrays by using the global array for combiners improves cache efficiency by taking advantage of the ring interconnection between the L2 caches. Specifically, when using local arrays, every L2 local cache miss should cause a memory access. On the contrary, when using the global array directly, the global array is shared across multiple cores. When a L2 cache miss occurs on one core, the data may be copied from another core's L2 cache to avoid the memory access.

Our framework automatically determines whether the local arrays are eliminated. Specifically, we consider cache efficiency - our framework only keeps the local arrays if they fit in the L2 caches and eliminates them otherwise.

3.6 Summary of Optimization Techniques

In this section, we propose four Xeon Phi specific optimization techniques. We summarize their applicability for a given application in Table 2. Note that all suitable techniques except vectorization friendly map are performed automatically. Enabling the vectorization friendly map, if necessary (according to the output at compile-time), only requires modifying one line of code.

4 MapReduce on Heterogeneous Platforms

In this section, we present the extension of our optimized MapReduce framework on a single Xeon Phi coprocessor

TABLE 2
Applicability of Optimization Techniques for a Given Application

Technique	Available containers	Applicability	Major consideration for applicability
Vectorization friendly map	All	Vectorizable map function	SIMD vectorization
SIMD parallelism for hash	Hash table	Always applicable	N/A
Pipelined map and reduce	Hash table	Hash table size larger than L2	Cache efficiency
Eliminating local array	Array	Local array size larger than L2 cache	Cache efficiency

[23] to a heterogeneous platform equipped with hardware accelerators. Note that though our evaluation platform in this study is a server equipped with multiple Xeon Phi coprocessors, we generalize our design to run on a heterogeneous platform with various types of coprocessors.

4.1 System Design

We define a heterogeneous computing platform as a host equipped with a few hardware accelerators. The hardware accelerators communicate with the host via PCIe or networking. Normally, the host is a conventional CPU, and the accelerators can be one or multiple Xeon Phi, GPU, FPGA, and any other coprocessors. On such a platform, both the input and output data of a MapReduce application reside in the host's memory. Our target is that the MapReduce framework is will be able to utilize all hardware resource (both the host and accelerators) on such a platform.

According to the previous study of MapReduce on multiple GPUs, the major performance issues are the workload imbalance and data transfer overhead [19]. We propose three techniques to solve these problems.

1) *Dynamic data scheduling* is used to balance workloads on all processors. The input data is partitioned into relatively small chunks. When a computation process is available, it will fetch a chunk of data and process.
2) We fully *pipeline* the communication and computation. *Non-blocking data transfer* is employed for input data prefetching and sending partial result data back to the host. *Double buffering* is used to enable the non-blocking data transfer.
3) We use *aligned memory transfer* to transfer data between the host and accelerators. This technique significantly increases the communication bandwidth specifically between the host and Xeon Phi coprocessors.

In our design, there are three kinds of processes running on the host or accelerators, which are input data split and sending process (I), MapReduce task computation process (C), and result merge process (M). Table 3 summarizes the placement of these processes and symbols used throughout the paper. Fig. 5 shows how these three processes are coordinated to achieve dynamic data scheduling and pipelining.

TABLE 3
Processes and Their Placement

Process	Placement	Symbol
Input data split and sending	Host	I_h
MapReduce task computation	Host, accelerators	C_h, C_a
Result merge	Host	M_h

We introduce the three processes in details with an example of word count.

MapReduce computation process (C_h and C_a). The entire system is essentially driven by the MapReduce computation process. Fig. 5 illustrates three major steps in this process. Overall, it starts with requesting an input chunk from the *input buffer manager* (step 1). This buffer manager adopts prefetching (step 2 and 3) to hide the data transfer overhead. Once the data is ready, it performs the local MapReduce job based on this data chunk (step 4). When the MapReduce is done, the result key-value pairs are merged into a partial result buffer (step 5). The partial result manager employs asynchronous data sending to send result to the process M when the partial result buffer is full (step 6).

Using the word count application as an example, the computation processs first requests a certain amount of text from the input buffer manager (steps 1, 2, and 3). When the input data is ready, it performs MapReduce, and then produces pairs of words and counts (step 4). These pairs produced based on the current input chunk are denoted as R_i. Furthermore, the process maintains a buffer storing partial results (denoted as R_p). Then R_i is merged into the buffer with partial result R_p (step 5). Step 1-5 are repeated to process multiple input chunks from the input buffer manager. However, when the partial result buffer is full, the partial result R_p will be sent to the result merge process (M_h) on the host (step 6).

Input data split and sending process (I_h). When a request is received from any MapReduce computation process (step 2), it performs a split function and sends a data chunk back (step 3). Note that the split function here (denoted as *split'* in Fig. 5) can be the same function as the *split* within a MapReduce job (as shown in Fig. 1), but with a coarser

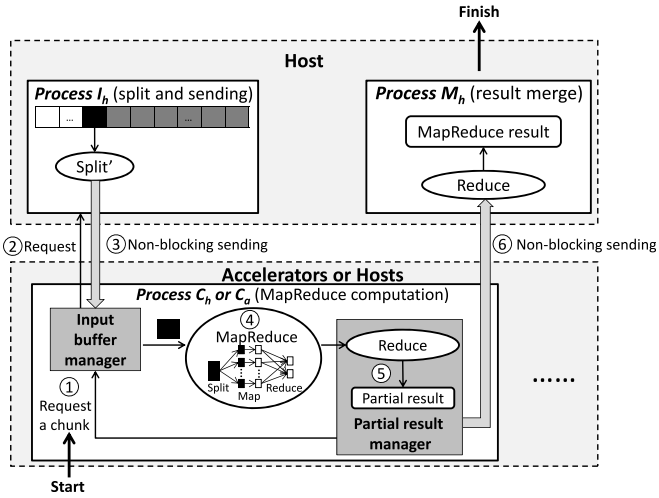


Fig. 5. The system design of MapReduce on a heterogeneous platform.

granularity to generate larger chunks. For example, for the word count, the *split'* here generates 1 million words each time for a MapReduce task. While the *split* within the MapReduce task (step 4) may generate 1,000 words each time.

Result merge process (M_h). When it receives any partial results from process C_h or C_a (step 6), it will apply the reduce function to merge the partial results with the existing results stored in its process. When all C_h and C_a processes are done, the final results are held by this process. The result merge here is similar to the partial result merge (Step 5). However, the input partial results are from MapReduce computation processes.

As shown in Fig. 5, the workload balance is achieved by partitioning the input data into chunks and each chunk is requested by a computation process. On the other hand, the key to pipelining lies in employing non-blocking data transfer in the input buffer manager for prefetching and partial result manager to send the result back to M_h asynchronously.

4.2 Non-Blocking Data Transfer

Non-blocking data transfer is employed by both the input buffer manager and partial result manager (Step 3 and 6 in Fig. 5). They use the same approach, *double buffering*, to pipeline the computation and data transfer. The double buffering requires two buffers, with one used to receive the next data chunk while the other to process the current data chunk. With the double buffering, computation and data transfer can be overlapped.

4.3 Aligned Memory Transfer

We find that the PCIe bandwidth between the host and Xeon Phi is sensitive to memory alignment with a 64 byte boundary. The bandwidth with aligned memory is 13× higher than that with misaligned memory (see Fig. 8b in Section 5). As many applications have user-defined *split'* functions, such as word count, aligned memory transfers are unable to be guaranteed by our system directly. We propose an approach to decompose a memory transfer into two transfers: the first transfer is for the first few bytes (< 64 bytes) with misaligned memory; the second transfer is for the major part with aligned memory.

4.4 Instantiation of Different Containers

Recall that our MrPhi implements three different containers, which are hash table, array, and common array. Overall, our design for heterogeneous platforms is adaptive to all containers. The application programming interface is the same for three containers. However, the difference of containers behind the framework does affect the performance.

Our optimization techniques, including non-blocking data transfer and aligned memory transfer, are proposed to hide the communication overhead. However, the effectiveness of these techniques depends on the workload. If the data transfer dominates the whole workload, its overhead cannot be perfectly hidden. As a result, it leads to poor performance scalability.

Particularly, the common array container usually introduces much more data transfer workload than the hash table and array containers. For an application with N input elements, the common array will generate exactly N result key-value pairs. Recall that, in Fig. 5, if the partial result buffer is full in the MapReduce computation process, it will send the partial result back to the host's result merge process. This will cause a problem for the common array container. Specifically, when the input data size increases, the partial result will be sent back more frequently than the hash table and array containers. For hash table, it usually generates much fewer key-value pairs than the input elements. For example, in our experiments (Table 5 in Section 5), the ratio between the result key-value pairs and input elements for Word Count is around 0.0014. For the array container, since the array size is fixed, the partial result will never be sent back until all input elements have been processed. Therefore, the percentage of data transfer workload for the common array based applications is normally higher than those based on the hash table or array containers. Consequentially, the performance scalability will also be worse for the common array based applications. Our experimental results in Section 5.4 verify this conclusion.

TABLE 4
Benchmark Applications

Application	Container	Applied optimization
Monte Carlo	Common array	Vectorization friendly map
Black Scholes	Common array	Vectorization friendly map
Word Count	Hash table	SIMD hash, pipelining
Reverse Index	Hash table	SIMD hash, pipelinine
Histogram	Array	Eliminating local arrays
Bloom Filter	Array	Eliminating local arrays

5 Experimental Evaluation

Hardware setup. We conduct our experiments on a platform equipped with four Intel Xeon Phi 5110P coprocessors. The hardware specification of each Xeon Phi has been summarized in Section 2.2. Additionally, our host contains two 8-core Intel Xeon E5-2687W CPUs, and each of which has 16 threads. The L2 and L3 cache sizes on each CPU processor are 2 and 20 MB, respectively.

Benchmark applications. We choose six MapReduce applications as shown in Table 4. Specifically, *Histogram*, *Word Count*, and *Reverse Index* are the sample applications from Phoenix++. We implement *Monte Carlo* and *Black Scholes*, which follow the GPU-based parallel implementations [4]. We also implement the building phase of *Bloom Filter*, which simulates its use in bioinformatics [28].

Implementation detail. MrPhi is developed using C++. Pthreads is used for thread parallelization on a single Xeon Phi. MPI is used to coordinate the communication between the host and Xeon Phi coprocessors to implement the heterogeneous MapReduce model. This is because MPI is the popular programming interface for the communication between the host and Xeon Phi. Additionally, we organize the threads in a *scatter* way such as thread i belongs to core $(i \bmod 60)$, where 60 is the number of cores.

For the experiments, we first characterize the performance of the Xeon Phi coprocessor. Then we study the performance impact of our various optimization techniques on a single Xeon Phi and perform end-to-end performance comparison with state-of-the-art multi-core based MapReduce framework

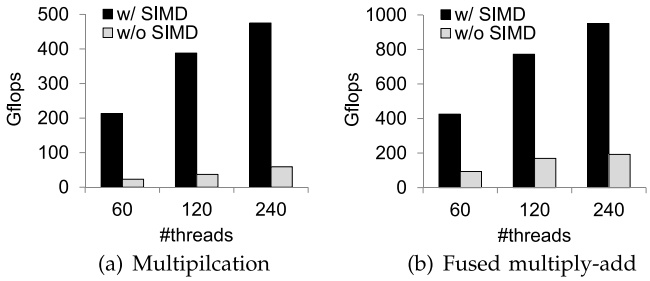


Fig. 6. Arithmetic throughput with and without SIMD.

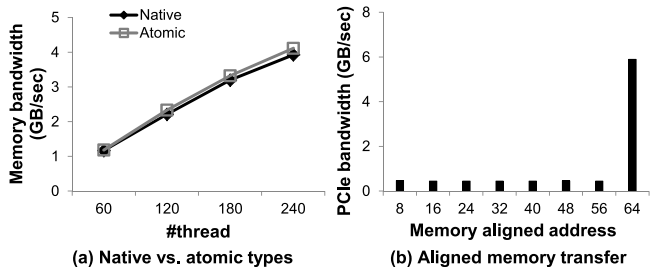


Fig. 8. (a) The random memory access bandwidth with native and atomic data types. (b) PCIe bandwidth between the host and Xeon Phi with different memory aligned addresses.

(Phoenix++). Finally, we show the scalability of MrPhi using multiple Xeon Phi coprocessors as well as the capability of using the host. By default, the number of threads per core is set to the one that gives the best performance, unless specified otherwise.

5.1 Characterizing the Xeon Phi Coprocessor

SIMD and peak performance. We evaluate the performance of SIMD processing for computation-intensive workloads. We adopt the benchmarks from the previous study [29] to measure the arithmetic throughput for double precision numbers. Fig. 6 shows that the 5-11× speedups can be achieved by employing SIMD processing. On the other hand, the theoretical peak performance of around 1 Tflops is almost achieved by fused multiply-add when there are 240 threads.

SIMD scatter and gather. We evaluate the new SIMD scatter/gather instructions (*_mm512_i32scatter_ps* and *_mm512_i32gather_ps*) on Xeon Phi. We let each thread perform independent scatter or gather, and then measure the memory bandwidth. The scatter writes the data from a vector to given memory locations, while the gather reads the data from given memory locations to a vector. For each thread, we make the memory locations randomly distributed in a fixed-size memory space. We vary this memory space size per thread to study the performance. Note that each thread's memory space is not overlapped with another thread's.

Fig. 7 shows that when the memory size is small enough to fit into the local L2 cache, SIMD scatter and gather are up to 3.4× faster than their scalar versions and are able to achieve high memory bandwidth. However, when the data size becomes large, the SIMD scatter and gather do not help the performance since the performance is dominated by the memory latency due to cache misses. This suggests that it is worthwhile to exploit the SIMD scatter/gather when the memory accesses are distributed over a small range of addresses.

Atomic data types. We study the performance of atomic operations within the context of our usage pattern where there are many random memory accesses on a large array with a very low conflict rate. We design our experiment to randomly update elements in an array with 32 million integers. Fig. 8a shows that using the native and atomic data types do not have much performance difference. We consider the overhead of atomic operations hidden by the memory latency. This suggests that when the memory accesses are random with a low conflict rate, using atomic data types on the Xeon Phi is a reasonable choice. Our optimization of eliminating the large local arrays follows this suggestion.

Aligned data transfer between the host and Xeon Phi. We investigate the PCIe bandwidth between the host and Xeon Phi coprocessor with different memory address alignments. Fig. 8b shows that only 64-byte aligned memory addresses are able to fully utilize the PCIe bandwidth (around 6 GB/sec). For other misaligned memory addresses, the bandwidth is around 13× lower than that with 64-byte aligned address. This suggests that data transfer with aligned addresses is crucial to maximize the PCIe bandwidth.

Thread initialization overhead. We find the thread initialization overhead on the Xeon Phi is high compared to conventional CPU (0.75 vs 0.067 millisecond per thread). Therefore, we implement a thread pool and only initialize the threads once.

5.2 Evaluation of Optimization Techniques

In this section, we evaluate the performance impact of our proposed techniques on a single Xeon Phi coprocessor. When we evaluate a specific technique, we evaluate the optimized implementation (with all applicable techniques enabled, denoted as *Opt.*) and the other implementation without this specific technique. We will evaluate our four techniques one by one, which are vectorization friendly map (Fig. 9), SIMD hash computation (Fig. 11), pipelined map and reduce (Fig. 12) and eliminating local arrays (Figs. 13 and 14).

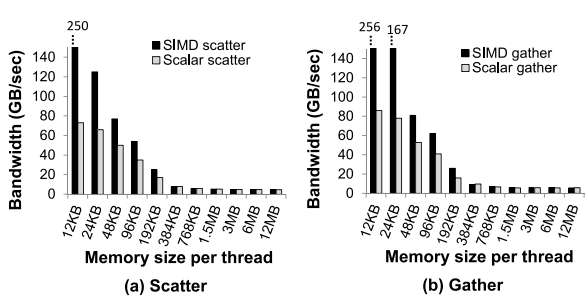


Fig. 7. Comparison between SIMD and scaler scatter/gather.

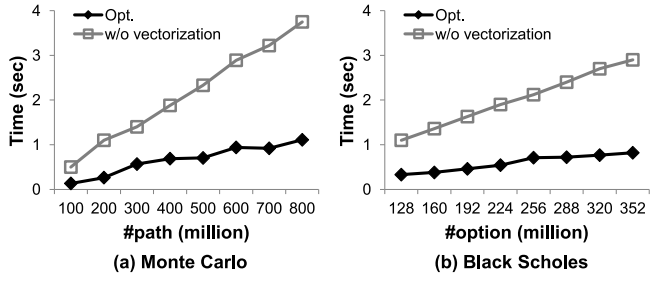


Fig. 9. Performance impact of vectorization friendly map for Monte Carlo and Black Scholes.

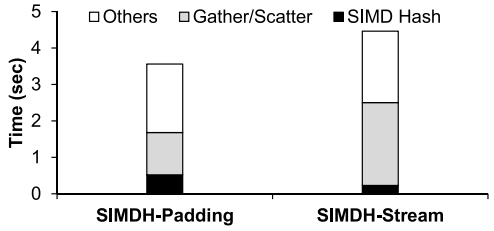


Fig. 10. Time breakdown of SIMD hash algorithms.

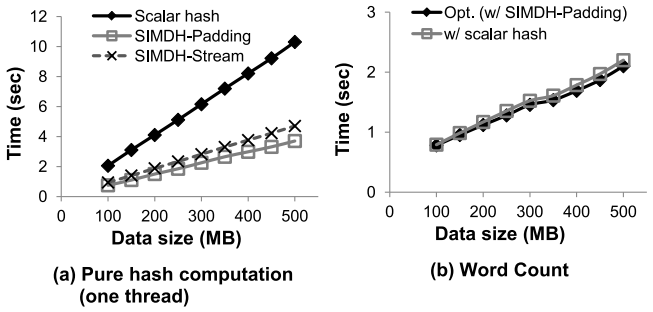


Fig. 11. The SIMD hash computation performance with the input data size varied. (a) The pure hash computation time (one thread). (b) The performance of Word Count with and without the SIMD hash computation.

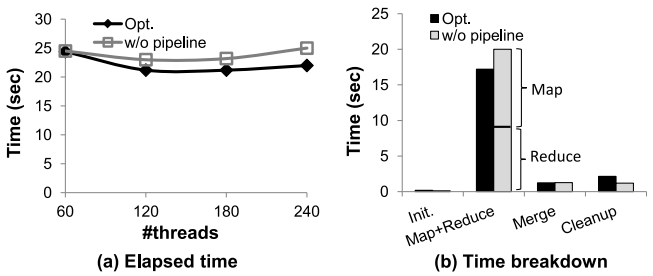


Fig. 12. Performance impact of pipelined map and reduce for Reverse Index. (a) Elapsed time (b) Time breakdown.

Vectorization friendly map. Fig. 9 shows the performance impact of vectorization friendly map for Monte Carlo and Black Scholes with data size varied. It shows that the vectorization friendly map can improve the performance by 2.5-4.2× and 3.0-3.6× for Monte Carlo and Black Scholes, respectively. For those two applications, the map phase dominates the overall performance (>99 percent). Therefore the vectorization for the map phase can greatly improve the overall performance.

SIMD parallelism for hash computation. We first evaluate the performance of hash computation separately using a single thread. Fig. 11a shows the performance result of pure hash computation with the input data size varied. We use the same data set as that used in the Word Count application. This shows that the SIMDH-Padding and SIMDH-Stream achieve the speedup of up to 2.8 and 2.2× over the scalar hash, respectively. Though SIMDH-Padding wastes computation resource due to the padding, it achieves better performance.

To investigate the performance difference between SIMDH-Padding and SIMDH-Stream, we study their time breakdown. Fig. 10 shows that their running time is decomposed to SIMD hash, Gather/Scatter and Others. SIMD hash is the SIMD intrinsics working on vectors. In our case

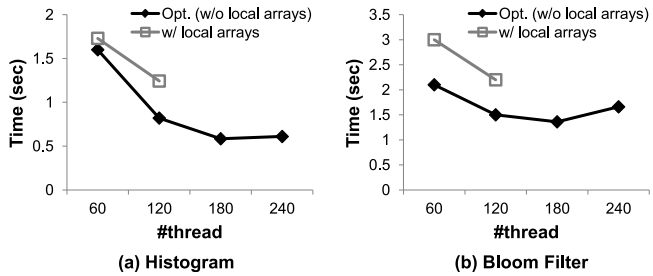


Fig. 13. Elapsed time of Histogram and Bloom Filter with the number of map threads varied. Histogram: 16 million unique keys and 256 million input elements; Bloom Filter: 30 million input elements, 300 million entries.

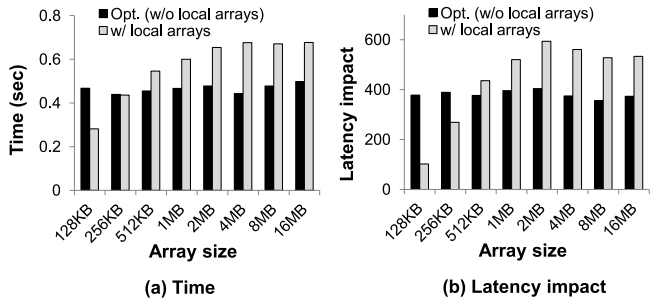


Fig. 14. Performance impact of eliminating local arrays for the map phase in Histogram. (a) Time. (b) Estimated memory latency impact.

of *bkdr* hash [24], there are two SIMD intrinsics per vector. Fig. 10 shows that SIMDH-Padding takes almost double the time to execute the SIMD hash compared to SIMDH-Stream. This is because SIMDH-Padding works on additional empty characters due to the padding. From our further investigation, this number is consistent to the padding efficiency (48 percent). However, SIMDH-Padding takes less time for Gather/Scatter than SIMDH-Stream. SIMDH-Padding has only one gather instruction per vector, while SIMDH-Stream has five gather/scatter instructions per vector. The more gather/scatter instructions come from the more complex control flow. The Others part represents the overhead of memory management and control flow operations (besides gather and scatter).

We further show the performance impact of using the SIMD hash for Word Count in Fig. 11b. It shows the overall performance is slightly improved (around 6 percent). The insignificant improvement is because the hash computation is not the performance bottleneck of the application. Instead, the random memory accesses from hash table building has poor cache efficiency [30], which dominates the overall performance.

Pipelined map and reduce. Word Count and Reverse Index are able to take advantage of pipelined map and reduce. We report the results of Reverse Index as Word Count has a similar conclusion. We use the data set in Phoenix++ for evaluations, which contains 78,355 files and 307,921 links in total. Fig. 12a shows the elapsed time with the number of threads varied. It shows that the overall performance is improved by around 8.5 percent. We further decompose the time as shown in Fig. 12b. This shows that for the map and reduce phases only, the pipelining technique improves the performance by around 14 percent. However, due to the storage overhead, the memory cleanup phase of the

TABLE 5
Data Sets for End-to-End Performance Comparison

Application	Data set
Word Count	Input data size: 500 MB
Reverse Index	#files; 78,355 ; #links: 307,921 ; size: 1 GB
Monte Carlo	#paths: 800 million
Black Scholes	#options: 352 million
Histogram	#unique keys: 16 million; #elements: 256 million
Bloom Filter	#elements: 30 million; #entries: 300 million (Arabidopsis chromosome 1)

pipelined map and reduce is more expensive and offsets the overall performance improvement.

Eliminating local arrays. By eliminating local arrays, the thread scalability for the map phases can be improved. Fig. 13 demonstrates such scenarios. The sizes of each local array are 64 and 40 MB for the Histogram and Bloom Filter, respectively. Fig. 13 shows that the largest numbers of threads when using local arrays are 120 for Histogram and Bloom Filter, due to the limited memory size (8 GB). On the contrary, if local arrays are eliminated, more threads can be used. As a result, by eliminating local arrays, it achieves a speedup of up to 2.1 and 1.6× for Histogram and Bloom Filter, respectively. Note that, with the different sizes of arrays, the available number of map threads for using local arrays is different. Therefore, the performance improvement from eliminating local arrays varies across different data sets.

In Fig. 13, we also observe that when using the same number of threads, our optimized implementation still outperforms the implementation using local arrays. This is because of the improved data locality. We further study this problem. We vary the data size of each array (note that the Opt. solution only has one global array). In this experiment, we exclude the impact from the thread scalability and make the two implementations (Opt. and w/o local arrays) be able to employ the same number of threads. Fig. 14a shows that when the array size is small enough to fit into the L2 cache, using local arrays is more efficient. This is because the global array has the overhead of cache coherence. However, when the array becomes larger, using global arrays outperforms local arrays by up to 34 percent. In such a case, both local and global arrays suffer from cache misses. However, the optimized solution can take advantage from the ring interconnection for better cache efficiency (Section 3.5).

To confirm the cache efficiency, Fig. 14b further shows the estimated memory latency impact. Measuring the memory latency impact was suggested by Intel to investigate the cache efficiency. It is an approximation of the number of clock cycles devoted to each L1 cache miss. Fig. 14b shows the consistent trend of increased memory latency impacting the elapsed time (Fig. 14a).

5.3 MrPhi vs. Phoenix++ on the Xeon Phi

Now we show the end-to-end performance comparison between our MrPhi and Phoenix++ [15], which is state-of-the-art MapReduce framework on multicore platforms. We compare their performance on a single Xeon Phi since Phoenix++ is unable to run on multiple Xeon Phi coprocessors. We use large data sets for evaluations, which are summarized in Table 5.

Fig. 15 shows the performance comparison between MrPhi and Phoenix++. Since the largest numbers of available threads for Histogram and Bloom Filter are less than 60 threads, we use dashed lines to represent their performance with more than 60 threads. Fig. 15 shows that, for Monte Carlo and Black Scholes, which take advantage of vectorization in MrPhi, they are up to 2.7 and 4.6× faster than their counterparts in Phoenix++. For Word Count and Reverse Index that are based on the hash table, MrPhi can achieve a speedup of up to 1.2×. Furthermore, by eliminating local arrays, MrPhi is able to achieve a speedup of up to 38 and 18× for Histogram and Bloom Filter, respectively. Note that these two speedup numbers are better than those reported in Fig. 13. This is because Phoenix++ has some implementation issues for large arrays, which lead to worse thread scalability as well as worse performance. In summary, our MrPhi can achieve a speedup of 1.2 to 38× over Phoenix++ on Xeon Phi.

5.4 Performance on the Heterogeneous Platform

In this section, we evaluate our MrPhi on the heterogeneous platform using all Xeon Phi coprocessors as well as the host. For each application, we further increase the input data size from the number listed in Table 5 until its speedup is stable for each experiment.

Performance on multiple Xeon Phi coprocessors. Fig. 16 shows the performance scalability with the number of Xeon Phi coprocessors increased. This shows that for most applications, including Word Count, Reverse Index, Histogram and Bloom Filter, their speedups are almost linearly proportional to the number of coprocessors, which show the excellent scalability. On the other hand, Monte Carlo and Black Scholes have the scalability factor of around 0.45. Recall that Monte Carlo and Black Scholes utilizes a common array container, which needs to send back their partial result to the host frequently (as described in Section 4.4). As a result,

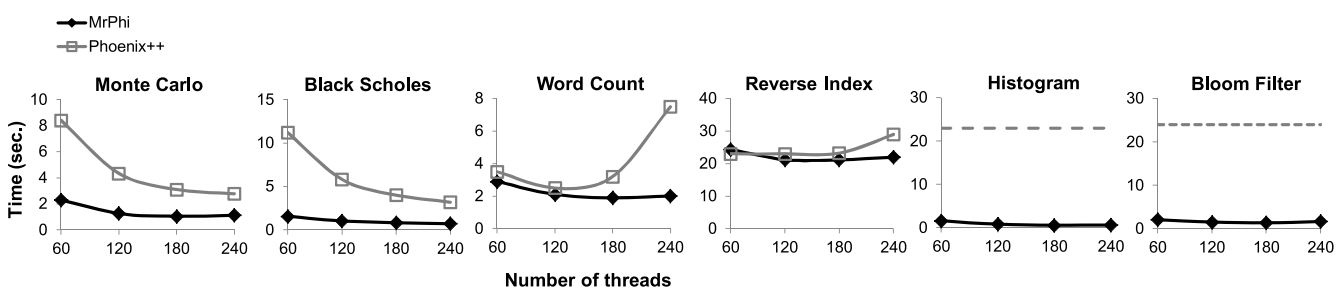


Fig. 15. Performance comparison between MrPhi and Phoenix++ on the Xeon Phi.

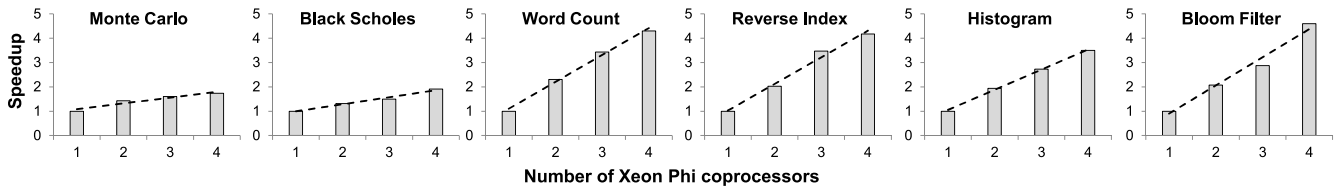


Fig. 16. Performance speedup with the number of Xeon Phi coprocessors varied.

the data transfer overhead is unable to be hidden and leads to poor performance scalability. Nevertheless, for Monte Carlo and Black Scholes, the performance speedup is still proportional to the number of Xeon Phi coprocessors with a sub-linear scalability. It achieves around 2× speedup when there are four Xeon Phi coprocessors used.

Performance improvement with the host. Our MrPhi framework is able to utilize multiple coprocessors as well as the host for the heterogeneity. Fig. 17 demonstrates the extra performance gained (16-67 percent) by using the host's resource. Note that due to the different hardware features, e.g., number of cores and cache size, the performance comparison between the Xeon Phi and host varies with applications. The purpose of Fig. 17 is to demonstrate the capability of our framework for utilizing the host's resource. Additionally, it also shows the applications such as Monte Carlo and Black, which suffer more from memory transfer overhead, will benefit less from the host's power. The reason is that data transfer overhead offsets the computation speedup for those applications.

5.5 Summary and Lessons Learnt

In this section, we summarize our major findings and lessons learnt from developing MrPhi.

First, direct code porting from existing programs is unlikely to result in high performance on Xeon Phi. The direct code porting eases the development burden, but it may not utilize Xeon Phi effectively. By comparing our result with directly porting Phoenix++, we find our optimized MrPhi framework is 1.2 to 38× faster.

Second, it is essential to exploit advanced hardware features on Xeon Phi. The major reason of the inefficiency for ported Phoenix++ on Xeon Phi is because their implementation is unaware of the unique hardware features on Xeon Phi, e.g., 512-bit SIMD vectors. Redesign of algorithms or data structures is required to take advantage of the hardware features.

Third, we do realize that Xeon Phi has several limitations, which may introduce difficulties for designing high-performance applications. For example, SIMD intrinsics are inflexible and difficult to use effectively. This greatly increases the coding complexity for large-scale programs. In addition, the small L2 cache on each core also makes Xeon Phi inefficient when handling random memory accesses.

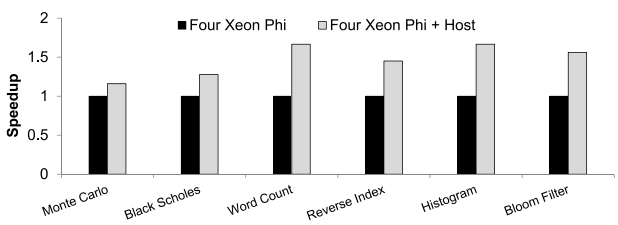


Fig. 17. Performance improvement with the host. Four Xeon Phi coprocessors are used.

Fourth, for applications employing multiple Xeon Phi coprocessors, hiding data transfer overhead is important. We believe our careful design of different kinds of processes employing non-blocking data transfer and aligned memory transfer techniques are applicable to other applications on multiple Xeon Phi.

6 Conclusion

In this work, we develop MrPhi, the first MapReduce framework optimized for the Intel Xeon Phi coprocessor. In MrPhi, in order to take advantage of VPUs, we develop a vectorization friendly technique for the map phase and SIMD hash computation. We also pipeline the map and reduce phases to better utilize the hardware resource. Furthermore, we eliminate local arrays to improve the thread scalability and data locality. Our framework is able to automatically identify suitable techniques to optimize a given application. Additionally, MrPhi is able to run on a heterogeneous platform to utilize the resource from all coprocessors as well as the host. Our experimental results show that MrPhi is able to achieve a speedup of 1.2 to 38× over Phoenix++ for different applications on a single Xeon Phi. Good performance scalability, e.g., linear scalability for four applications, is also demonstrated on a platform equipped with up to four Xeon Phi coprocessors.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. This work was partially supported by the National Natural Science Foundation of China (No. 61300005). Bingsheng He is partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

References

[1] H. P. Huynh, A. Hagiescu, O. Z. Liang, W.-F. Wong, and R. S. M. Goh, "Mapping streaming applications onto GPU systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 9, pp. 2374–2385, Sep. 2014.

[2] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.

[3] M. Lu, J. Zhao, Q. Luo, B. Wang, S. Fu, and Z. Lin, "GSNP: A DNA single-nucleotide polymorphism detection system with gpu acceleration," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 592–601.

[4] NVIDIA Compute Unified Device Architecture (CUDA) [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html, 2013.

[5] J. Dean, and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Oper. Syst. Design Implementation*, 2004, p. 10.

[6] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis, "Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon phi coprocessors," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 1085–1097.

[7] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on Intel Xeon phi coprocessor," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 126–137.

[8] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. 27th ACM Int. Conf. Supercomput.*, 2013, pp. 273–282.

[9] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim, "Tera-scale 1d FFT with low-communication algorithm and Intel Xeon phi coprocessors," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, p. 34.

[10] F. Wende and T. Steinke, "Swendsen-wang multi-cluster algorithm for the 2d/3d ising model on Xeon phi and GPU," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, p. 83.

[11] STAMPEDE: Dell PowerEdge C8220 culster with Intel Xeon Phi Coprocessors [Online]. Available: http://www.tacc.utexas.edu/resources/hpc/stampede, 2013.

[12] China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500 List [Online]. Available: http://www.top500.org/blog/lists/2013/06/press-release/, 2013.

[13] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proc. 13th Int. Conf. High Perform. Comput., Archit.*, 2007, pp. 13–24.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.

[15] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular mapreduce for shared-memory systems," in *Proc. 2nd Int. Workshop MapReduce Its Appl.*, 2011, pp. 9–16.

[16] R. Chen, H. Chen, and B. Zang, "Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 523–534.

[17] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing mapreduce for multicore architectures," Comput. Sci. Artif. Intell. Lab., Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2010-020, 2010.

[18] J. A. Stuart and J. D. Owens, "Multi-GPU mapreduce on GPU clusters," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2011, pp. 1068–1079.

[19] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: Writing parallel program portable between CPU and GPU," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 217–226.

[20] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled CPU-GPU architecture," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, p. 25.

[21] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Programm. Gate Array*, 2010, pp. 93–102.

[22] M. de Kruijf and K. Sankaralingam, "Mapreduce for the cell broadband engine architecture," *IBM J. Res. Develop.*, vol. 53, no. 5, pp. 747–758, 2009.

[23] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh, "Optimizing the mapreduce framework on Intel Xeon phi coprocessor," in *Proc. IEEE Conf. BigData*, 2013, pp. 125–130.

[24] B. W. Kernighan and D. M. Ritchie, *C Programming Language*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1978.

[25] FNV Hash [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/index.html, 2013.

[26] Hash Functions [Online]. Available: http://www.cse.yorku.ca/~oz/hash.html, 2013.

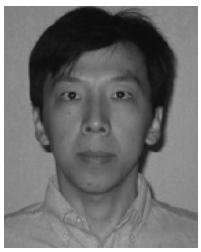
[27] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proc. 7th USENIX Conf. Netw. Syst. Design Implementation*, 2010, p. 21.

[28] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, "Bloom filter performance on graphics engines," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 522–531.

[29] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving Intel Xeon phi," in *Proc. 5th Int. Conf. Perform. Eng.*, 2014, pp. 137–148.

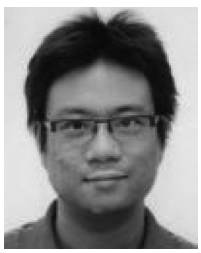
[30] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry, "Improving hash join performance through prefetching," in *Proc. Int. Conf. Data Eng.*, 2004, pp. 116–127.

Mian Lu received the bachelor's degree in software engineering from the Huazhong University of Science and Technology in 2003-2007, and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2007-2012. He is currently a scientist at the Institute of High Performance Computing, A*STAR, Singapore. His research interests are high performance computing and big data analytics.



Yun Liang received the BS degree in software engineering from Tongji University, Shanghai, in 2004, and the PhD degree in computer science from the National University of Singapore in 2010. He worked as a research scientist in ADSC, University of Illinois Urbana-Champaign between 2010 and 2012. He has been an assistant professor in the school of EECS, Peking University since 2012. His research interests include GPU architecture and optimization, heterogeneous computing, embedded system, and high level synthesis. He serves as a technical committee member for ASPDAC, DATE, CASES, etc. He is the TPC Subcommittee chair for ASPDAC13. He received the Best Paper Award in FCCM11 and Best Paper Award nominations in CODES+ISSS08 and DAC12.

Huynh Phung Huynh received the PhD degree in computer science from the National University of Singapore in 2010. His research interest focuses on High Performance Computing (HPC) research such as developing productivity tools for GPU/Many-core computing and Big data analytics. He is currently leading HPC group at the Institute of High Performance Computing, A*STAR, Singapore.



Zhongliang Ong received the BS degree in computing (specializing in computer engineering) from the National University of Singapore, in 2012. He is currently a research engineer at the Institute of High Performance Computing in Singapore. His research is centered on investigating parallel computing on heterogeneous systems.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science from the Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in the Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Rick Siow Mong Goh received the PhD degree in electrical and computer engineering from the National University of Singapore. He is the department director of the Computing Science Department at the A*STAR Institute of High Performance Computing. His research interests include high performance computing and big data analytics. He has a strong background in computer engineering and also has several years of experience in discrete event simulation, parallel and distributed computing, and performance optimization and tuning of applications on large-scale computing platforms.