SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

December 2017, Vol. 60 122106:1–122106:17
doi: 10.1007/s11432-015-0989-3

Characterizing and optimizing Java-based HPC applications on Intel many-core architecture

Yang YU[1,2], Tianyang LEI[2], Haibo CHEN[2] & Binyu ZANG[2*]

[1]*School of Computer Science, Fudan University, Shanghai* 200433*, China;*
[2]*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai* 200240*, China*

Received October 3, 2016; accepted December 13, 2016; published online May 8, 2017

Abstract The increasing demand for performance has stimulated the wide adoption of many-core accelerators like Intel® Xeon Phi™ Coprocessor, which is based on Intel's Many Integrated Core architecture. While many HPC applications running in native mode have been tuned to run efficiently on Xeon Phi, it is still unclear how a managed runtime like JVM performs on such an architecture. In this paper, we present the first measurement study of a set of Java HPC applications on Xeon Phi under JVM. One key obstacle to the study is that there is currently little support of Java for Xeon Phi. This paper presents the result based on the first porting of OpenJDK platform to Xeon Phi, in which the HotSpot virtual machine acts as the kernel execution engine. The main difficulty includes the incompatibility between Xeon Phi ISA and the assembly library of Hotspot VM. By evaluating the multithreaded Java Grande benchmark suite and our ported Java Phoenix benchmarks, we quantitatively study the performance and scalability issues of JVM on Xeon Phi and draw several conclusions from the study. To fully utilize the vector computing capability and hide the significant memory access latency on the coprocessor, we present a semi-automatic vectorization scheme and software prefetching model in HotSpot. Together with 60 physical cores and tuning, our optimized JVM achieves averagely 2.7x and 3.5x speedup compared to Xeon CPU processor by using vectorization and prefetching accordingly. Our study also indicates that it is viable and potentially performance-beneficial to run applications written for such a managed runtime like JVM on Xeon Phi.

Keywords many-core, Java, Xeon Phi, HPC, prefetching

Citation Yu Y, Lei T Y, Chen H B, et al. Characterizing and optimizing Java-based HPC applications on Intel many-core architecture. Sci China Inf Sci, 2017, 60(12): 122106, doi: 10.1007/s11432-015-0989-3

1 Introduction

As high-performance computing (HPC) continually evolves, the demand for computing resources has been steadily increasing in recent years. This also stimulates a new processor architecture, where more and more cores are integrated onto a single chip. The evolvement of many-core architectures has generated some remarkable outcomes. One of them is the Intel® Xeon Phi™ Many Integrated Core (MIC) Architecture, which is targeted for highly parallel and HPC workloads in a variety of fields [1]. Despite the similar features like SIMD and vectorization shared with general-purpose CPUs and GPUs, Xeon Phi has a completely independent OS and cache coherency support, providing massive computing power.

* Corresponding author (email: byzang@sjtu.edu.cn)

On the other hand, the pursuit for easy and portable programmability for HPC has never stopped. Java, as a very popular and widely-used language, is emerging as an appealing and competitive paradigm in HPC area, particularly because of its built-in multithreading mechanism and strong community/corporation support, as well as the continuous improvement in the performance of Java Virtual Machine (JVM) over the years. There are a lot of projects that focus on Java in HPC to improve program productivity [2–4], covering various fields such as computational physics, biochemistry, astronomy [5], and financial services. Java is shown not much slower than C/C++ for many numeric and scientific computing benchmarks [6, 7].

Despite the emergence of Java on HPC, we observe that there is still little study on the performance and scalability of Java HPC applications on Xeon Phi. Part of the reason is the lack of official support for Java in Intel's MIC architecture. To bridge this gap, this paper first presents a porting of OpenJDK platform to the Xeon Phi coprocessor, including the HotSpot VM as the kernel execution engine. This enables us to conduct a comprehensive study to gain more understanding on the performance characteristics of Java HPC applications on Xeon Phi.

Our study analyze the single-threaded throughput and multithreaded scalability of nine Java HPC programs which cover different kinds of algorithms and computing kernels. The single-threaded runs reveal quite inferior performance on Xeon Phi compared to that on CPU processors. Additionally, huge variations are observed among our selected benchmarks due to the different program characteristics, especially the memory access patterns. According to the result of multithreaded runs, we observe a satisfying scalability which is mainly brought by the large amount of hardware threads and the intrinsically supported multithreading mechanism in HotSpot VM. We thoroughly analyze the reasons for the performance gap between two architectures and the scalability issues. Accordingly, we provide a guidance for further optimization opportunities in HotSpot.

Among our proposed optimizing strategies, we implement a semi-automatic vectorization scheme in HotSpot to fully utilize the 512-bit vector processing unit on Xeon Phi as a first step. Our approach breaks the restrictions of HotSpot's original auto-vectorization, allowing programmers to indicate the computing loop that needs to be vectorized by using an annotation before the loop. Accordingly, our modified HotSpot VM will generate the corresponding native vector instructions. In a single-threaded run, an averagely 2.5x speedup is gained for five array-based benchmarks. By using our semi-automatic vectorization and an optimal number of threads, we can achieve averagely 2.7x and up to 3.4x speedup on MIC compared to that on the CPU processor.

Besides, Xeon Phi suffers significant memory access latency as a result of the design tradeoff for a much higher aggregate bandwidth, as well as more frequent on-chip cache misses due to the lack of traditional last-level cache. Hence, it relies heavily on software data prefetching to bring data into local caches ahead of need. In order to improve the poor memory performance, we implement a semi-automatic software prefetching model in HotSpot's JIT compiler based on an in-depth learning of the default customized prefetching policies of Intel *ICC* compiler. We apply various prefetching strategies to the vectorized Java benchmarks and analyze the performance under different combinations. The evaluation result shows that compared to the original version, our prefetching model could achieve up to 8.1x and 3.8x speedup under single thread and multi-threads accordingly. Moreover, an averagely 3.5x speedup for the best throughput with optimal thread count could be observed with respect to the CPU server. In summary, the main contributions of this paper include:

- The first porting of OpenJDK as a Java runtime environment (JRE) on Intel MIC architecture (Section 3).
- A comprehensive study on the performance issues such as the throughput and scalability of a set of high-performance computing applications on Xeon Phi (Section 4).
- A semi-automatic vectorization scheme in HotSpot, which fully utilizes the computing power of VPU on MIC and achieves averagely 2.7x speedup compared to the CPU server (Section 5).
- A semi-automatic prefetching model based on an in-depth learning of the prefetching strategies on Xeon Phi, which can effectively hide the high memory access latency and achieve averagely 3.5x speedup with respect to the CPU server (Section 6).

2 Background

This section presents an architectural overview of Intel® Xeon Phi™ coprocessor and a brief introduction to OpenJDK Java platform.

2.1 Intel Xeon Phi many-core architecture

The Intel Many Integrated Core (MIC) Architecture is designed by Intel for the purpose of high-performance computing. Intel announced "Xeon Phi" as the brand name for all their products based on the MIC architecture. The current generation of Xeon Phi product has a code name—Knights Corner (KNC), which is a coprocessor card that could be connected to a Xeon CPU host via a PCIe interface.

2.1.1 *Architectural overview*

The Xeon Phi coprocessor comprises up to 60 physical cores, each supporting 4 hardware thread contexts, which enables a total of 240 threads running simultaneously. The cores are based on a modified P54C design (used in the original Pentium) and comply with an in-order execution mode. The frequency of each core is around 1GHz, which is much lower than that on normal CPUs. This architecture provides a great opportunity for high parallelism but with a weak single-core processing ability.

Each coprocessor core employs a 512-bit wide vector processing unit (VPU) with 32 vector registers named *ZMM*. A new 512-bit SIMD ISA for Intel MIC architecture is designed to make use of the VPU. MIC does not support other SIMD ISAs, such as MMX, SSE, or AVX [8]. With the 512-bit VPU, 16 single-precision or 8 double-precision float-point operations can be performed in a single vector instruction. Therefore, compared to the traditional 128-bit SSE and 256-bit AVX, the new vector ISA can achieve a great improvement in the instruction-level parallelism.

The memory on the MIC is based on GDDR5 technique, which provides a theoretical maximum aggregate bandwidth of more than 300 GB/s, but suffers high memory access latency as a tradeoff. Each core has a 32 KB L1 data/instruction cache and a 512 KB L2 cache. The L2 caches are kept fully coherent and interconnected with each other as well as the memory controllers via a bidirectional ring bus. There is no traditional shared last-level cache on MIC.

2.1.2 *Programming for Xeon Phi*

Unlike other many-core platforms like GPU, there is a modified Linux μOS (version: 2.6.38.8) running on MIC, which is completely independent of the operating system on the host. An application has two ways to run on Xeon Phi—*offload mode* and *native mode*. Under the offload mode, the application starts on the host, and during execution it offloads highly parallel and computation-intensive regions to the coprocessor. As for the native mode, the application runs independently on Xeon Phi and never transfers data from/to the host. In this paper, we only focus on the native execution mode.

At present, Xeon Phi can run applications written in C/C++ and Fortran. There are various programming models and tools provided for Xeon Phi, e.g., OpenMP, Intel MPI, Intel Cilk Plus [9]. However, Xeon Phi does not support running Java.

2.2 OpenJDK platform

OpenJDK (Open Java Development Kit) is a widely-used and open-source implementation of the Java Platform.

The OpenJDK project consists of a number of components, including the Java compiler—*javac*, HotSpot virtual machine and the Java Class Library, etc.

Javac Compiler: OpenJDK provides a front-end Java compiler—*javac*, to parse Java source files into bytecodes that conform to the JVM specification [10] and store them in the class files in binary, which can be then executed by a Java virtual machine.

HotSpot VM: HotSpot virtual machine is the kernel execution engine of OpenJDK that can execute the Java bytecodes. HotSpot has a number of modules, including the class loader, Java interpreter, just-in-time compiler (JIT), garbage collector, etc., which cooperatively work together. In a running process, the class loader first dynamically loads the classes and interfaces into HotSpot. The interpreter then decodes and executes the bytecodes one by one, following the original instruction sequence stored in the binary file. Since *javac* does not perform any advanced optimization when compiling the source code, it is very slow to run a Java program solely using the interpreter. To improve the performance, HotSpot VM employs a just-in-time compiler (JIT) to translate Java bytecodes into native machine instructions for "hot codes" at runtime. The JIT compiler leverages a lot of extensive and multi-level optimizations while generating native instructions, which significantly reduces the execution time. HotSpot VM maintains an assembly library for each particular architecture such as x86 and Sparc. Either the bytecodes executed by interpreter or the native codes generated by JIT, they are essentially implemented by the platform-dependent instructions in the assembly library.

3 Methodology

In this section, we present our methodology in detail, including: (1) an introduction about our porting work of OpenJDK to Xeon Phi, (2) the multi-threaded Java HPC benchmarks, and (3) the experimental environment and measurement details.

3.1 Porting OpenJDK to Xeon Phi

In this work, we use OpenJDK 7u as our porting target. The primary challenges we face during the porting process and corresponding solutions are presented as below:

(1) First we need to cross-build OpenJDK for the MIC architecture. We use a customized GCC as the cross-compiler from the host machine, which is available as part of the Xeon Phi SDK. Moreover, building the Java class libraries in OpenJDK requires many dependent libraries, some of which are not provided by the μOS on MIC. Such libraries are related to graphics, fonts, etc., which are not essential for basic Java running. Therefore, we make a *headless* (i.e., no support for graphical applications) build of OpenJDK and modify the makefiles to omit such libraries when cross-compiling.

(2) The HotSpot VM can not be simply initialized after the cross-build. Most of our efforts focus on customizing HotSpot for the MIC architecture. As mentioned, Xeon Phi is not compatible with Intel's SSE or AVX instruction sets, which are required by HotSpot for floating-point operations on x86-based architectures. Instead, the coprocessor employs a new 512-bit SIMD ISA with brand new instructions and encodings. However, it does not cover all floating-point related instructions in HotSpot. Fortunately, Xeon Phi does support the legacy x87 FPU instruction set [11]. In Java, all floating-point arithmetic is basically IEEE 754 compliant [10]. Therefore, we leverage the new vector instructions to replace the SSE and AVX instructions in HotSpot, with some legacy x87 floating-point instructions as complement.

(3) Besides the incompatibility of SSE and AVX instructions, some others instructions, e.g., *mfence* and *clflush*, are not supported either. We modified these instructions based on the semantics in HotSpot as well as the instruction manual of MIC. Moreover, the encoding format of vector instructions on Xeon Phi is totally different from x86 instructions, which means adding vector instructions into the assembly library of HotSpot requires a redesign in the binary encoding. Therefore, we extend the assembly library with all indispensable vector instructions included.

3.2 Java benchmarks

In this paper, we study nine multi-threaded computing- and memory-intensive benchmarks. Five of them are derived from the Thread Version 1.0 of Java Grande benchmark suite [12]—*Crypt, Series, SOR, SparseMatmult* and *LUFact*. The Java Grande benchmarks are essentially dominated by scientific and numeric-intensive computation. For the other four Java benchmarks, we port them from the Phoenix benchmark suite [13] almost without distinction. Their key computations cover application domains such

Table 1 Hardware configuration

Parameter	Intel Xeon Phi^{TM} Coprocessor 5110P	Intel® Xeon® CPU E5-2620
Chips	1	1
Core type	In-order	Out-of-order
Physical cores	60	6
Threads per core	4	2
Frequency	1052.630 MHz	2.00 GHz
Data caches	32 KB L1, 512 KB L2 per core	32 KB L1d, 32 KB L1i 256 KB L2, per core 15 MB L3, shared
Memory capacity	7697 MB	32 GB
Memory technology	GDDR5	DDR3
Peak memory bandwidth	320 GB/s	42.6 GB/s
Vector length	512 bits	256 bits (Intel® AVX)
Memory access latency	340 cycles	140 cycles

as scientific computing (*MatrixMultiply*) and artificial intelligence (*KMeans, LinearRegression, PCA*). We emulate the parallel P-thread version of Phoenix and implement them using Java's built-in multi-threading mechanism, keeping their computing kernel logics unchanged. An arbitrary number of threads could be set to run each of the above benchmarks.

3.3 Experimental setup

Table 1 lists the architectural details of the MIC card used in this work. In addition, we provide performance comparison with an Intel Xeon processor, which acts as the host server of the coprocessor.

For multi-threaded executions on Xeon Phi, we assign the thread count as 1, 20, 40, 60 (physical core count), 120, 180 and 240. While for the Xeon CPU, the selected thread number varies among 1, 2, 4, 6 (physical core count), 9 and 12. In an implementation of HotSpot VM on Linux OS, each Java application thread is mapped to a native thread, belonging to the same JVM process. We invoke JNI (i.e., Java Native Interface) to make sure all the application threads will be distributed evenly onto each physical core in a multi-threaded run. Considering the inherent timing variations during benchmark runs, each benchmark-thread pair is executed five times. We report the average throughput results for the scalability and overall performance analysis.

4 Experimental analysis

This section summarizes the experimental results with the goal of correlating the performance metrics to the behaviors of applications, JVM components and the coprocessor features. Such a correlation allows us to identify the performance bottlenecks of computation- and memory-intensive Java programs running on Xeon Phi at multiple levels. Based on this, we further propose some approaches to improve the performance of JVM on Xeon Phi.

4.1 Performance with single thread

In order to have an intuitive impression of the performance differences between the two architectures, we first compare the single-threaded throughputs with Xeon CPU server as a baseline. The evaluation of all the benchmarks under single thread on both CPU and coprocessor is given in Figure 1. The left group contains the results of Java version and the right group is for the corresponding C-version. In each CPU-MIC pair, the throughputs are normalized to that on MIC.

As shown in Figure 1, there is a significant performance gap between CPU and MIC under single-threaded execution. This is actually caused by various reasons, including the in-order execution type and

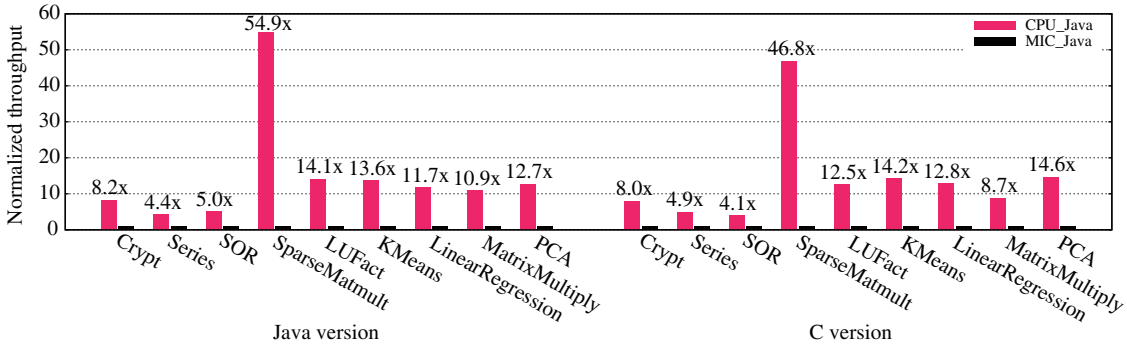

Figure 1 (Color onilne) Throughput of Java Grande benchmark suite with single thread.

Table 2 L1 cache hit rates of 7 memory-intensive benchmarks

Hardware events	SOR	SparseMatmult	LUFact	KMeans	LinearRegression	MatrixMultiply	PCA
L1 cache hit rate (%)	98.2	74.4	91.5	87.8	90.9	92.1	88.5

lower core frequency on Xeon Phi. Moreover, the two-cycle unit design in the instruction decoder makes it inferior to the 4 decoder units in Intel Xeon E5 processors.

Besides, a big variation among different applications can be observed. Actually, the nine programs can be divided into two categories based on the computing kernel loops: one is full of pure computations with little memory access, e.g., *Crypt* and *Series*; the other consists of the right seven benchmarks that are dominated by operations of double-precision floating-point arrays, with quite frequent memory accesses to the array elements, e.g., *SOR, SparseMatmult, LUFact, KMeans, LinearRegression, MatrixMultiply* and *PCA*. The degradation of the second category is generally greater than that of the first one. This is because that the GDDR5 memory of Xeon Phi suffers a much higher latency than the DDR3 memory, which is commonly used by normal Intel Xeon processors. However, there are two exceptional cases: one is *SOR* whose slowdown is much minor, the other is *SparseMatmult* which suffers a tremendous throughput degradation. Actually this is mainly caused by the memory access patterns leveraged in their computing kernels. *SOR* employs quite a sequential way to get array elements without nested loops, in which the hardware prefetcher could be triggered to retrieve data ahead of need on Xeon Phi [14]. As for *SparseMatmult*, its array elements are selected randomly, which leads to quite frequent off-chip memory accesses with the high latency totally exposed to the program.

To verify the explanations above, we measured the cache utility of the seven benchmarks with Intel® VTune™ Amplifier XE. Table 2 lists the L1 cache hit rate collected for the module of JIT generated code (VTune is not able to collect L2 hit rate on Xeon Phi). A very low miss rate could be observed for *SOR* because of its totally contiguous memory access pattern, while *SparseMatmult* suffers a rather high miss rate (up to 25.6%). For others five benchmarks, they reveal close hit rates which conform to the similar degradation times in Figure 1. The result is consistent with our analysis and explains the large variations in different kinds of benchmarks' performance gaps. Besides, there is no significant difference between the degradation times of Java- and C-versions, which confirms that the potential porting overhead is trivial, thus will not have an impact on our analysis of the performance.

4.2 Performance with multi-threads

One of the most important advantages of Xeon Phi is the large number of cores (up to 60) and hardware threads (up to 240). We use the speedup of throughput to study the performance scalability with various thread counts. Considering the massive number of threads, we additionally measured the throughputs of the nine benchmarks with a much bigger workload to make sure that each thread has enough work to do under a "many-thread" situation. The input data sizes are increased up to a magnitude of GB (while the original sizes are no more than tens of MB). The results with both workload sizes are presented in Figure 2. The graphs plot the speedup as a function of the number of threads. For each application, we

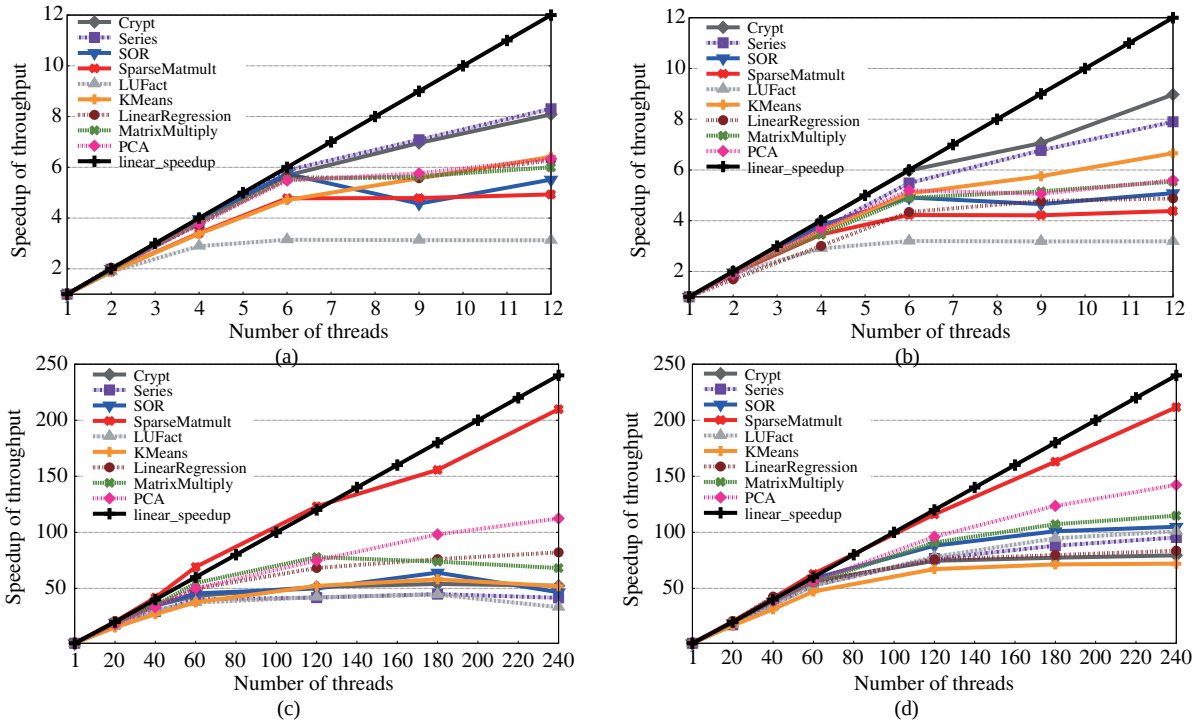

Figure 2 (Color online) Throughput speedup of multithreaded benchmarks on CPU and Xeon Phi with varying workload sizes. (a) Scalability on CPU with default size; (b) scalability on CPU with big size; (c) scalability on Xeon Phi with default size; (d) scalability on Xeon Phi with big size.

normalize the throughput with respect to a single-threaded configuration.

The scalability shows no significant fluctuation when the workload size varies on the CPU. However, the situation is rather different on the coprocessor. A better scalability for all the programs with a bigger workload size is observed, which could be explained by the Amdahl's law: since the thread count on Xeon Phi is far greater than that on CPU, a sufficiently large workload could make the running time of the parallel part remain non-neglected even under such a massive number of threads. The following analysis is all based on the results with big workload size.

Much better scalability for all the programs can be observed on Xeon Phi. In general, the performance keeps increasing with much more threads on Xeon Phi (more than 120) than CPU (no more than 12). An important reason is that the latencies caused by the in-order execution can be hidden by multi-threading. In another aspect, since the single-threaded performance is notably worse than that of CPU, it requires much more threads on Xeon Phi to boost the performance till the sequential part dominates the execution time. This also explains the prominent scalability of *SparseMatmult*: the high memory latency brings a much worse single-threaded throughput as well as a fairly high proportion of parallel-running part, which needs more than 240 threads to reach the critical point.

The throughputs increase before 120 *threads for all the programs on Xeon Phi.* As depicted in Figure 2(d), the 120-thread appears to be the critical point in throughput improvement for most of the benchmarks. Apart from Amdahl's law, another important reason is that the instruction decoder of each core is modified to be a two-cycle unit in the design of Xeon Phi [15]. For this reason the core is not able to issue instructions from the same hardware thread context in back-to-back cycle. Therefore, in order to hide this "issue latency" and fully utilize the core potential, programmers need to run at least two threads on each core.

Crypt is not able to scale even a little after exceeding two running threads per core. The throughput curve of *Crypt* keeps fairly flat when the number of running threads per core increases from 2 to 4. According to Intel's manual, most integer instructions (not including vector instructions for integer operations) have a 1-clock latency. For *Crypt*, when two threads are running on each core, the instruction

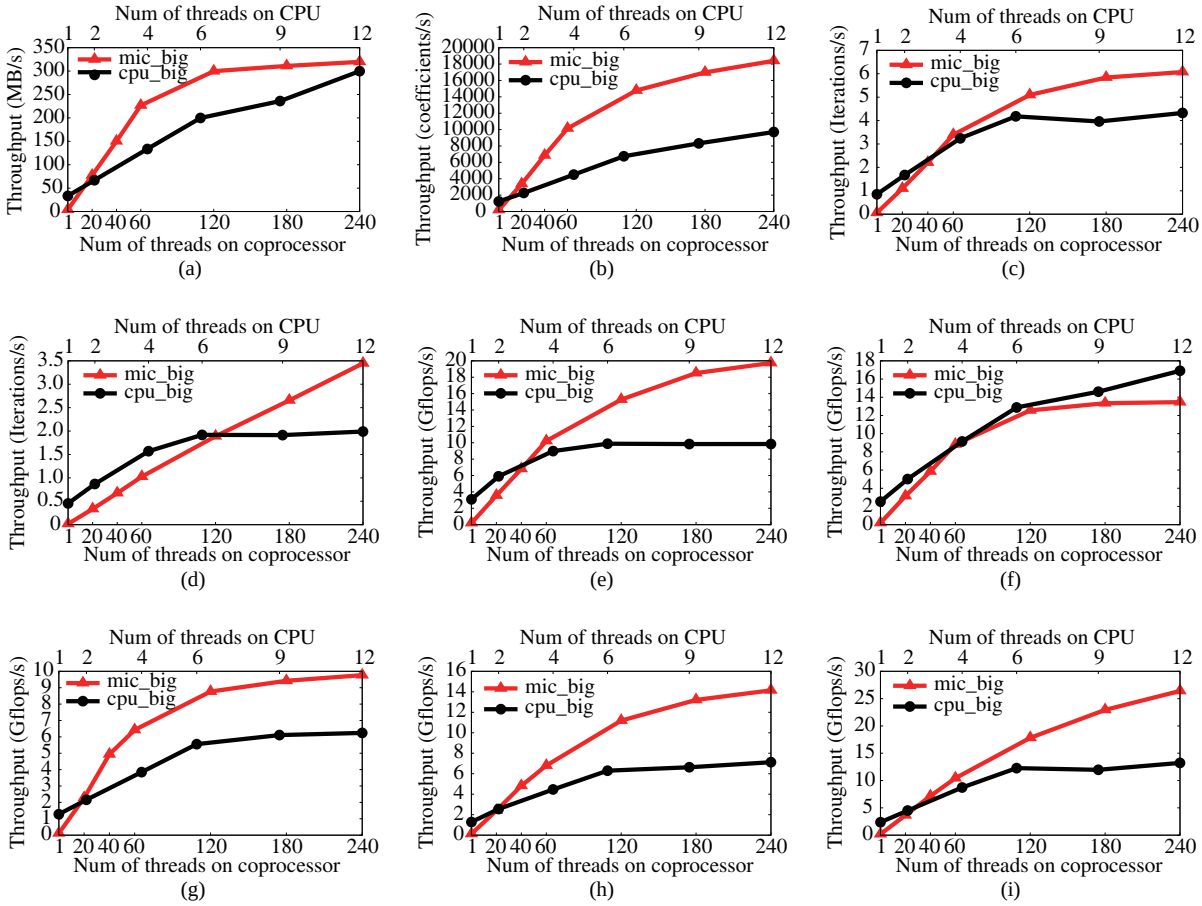

Figure 3 (Color online) Throughput of Java Grande benchmark suite with big workload size. (a) Crypt; (b) Series; (c) SOR; (d) SparseMatmult; (e) LUFact; (f) KMeans; (g) LinearRegression; (h) MatrixMultiply; (i) PCA.

stream can be fully pipelined. Since there is no more instruction latency (here is 1-clock) to hide, utilizing more than two threads per core will bring no extra benefit for throughput.

All the programs can be faster on MIC than on CPU with big enough running thread count except for KMeans. In Figure 3, we present a list of graphs to show the absolute throughput of each benchmark with varying thread counts on the two architectures. The x-axis on the top represents the number of running threads on CPU, while the bottom x-axis is for the coprocessor. We can observe a better performance for most benchmarks on Xeon Phi when running enough threads. Typically, the throughputs of *Series, LUFact, MatrixMultiply* and *PCA* with 240 threads nearly double the best throughputs on CPU accordingly. One exception here is the *KMeans* benchmark, which reveals inferior scalability and throughput under big thread count with respect to the processor. This is mainly because *KMeans* always performs a large number of computations for each element. When a lot of running threads are initialized, the application needs more space to cache previous data for later computations. The limited on-chip caches make Xeon Phi not able to fully parallelize the computing tasks, while normal CPU could scale well by using a sufficient last-level cache.

4.3 Summary of observations

Based on the observations, we can obtain some experience for the guidance of Java programming on Xeon Phi.

Memory latency. The lack of last-level cache generally causes more frequent off-chip memory accesses with much higher latency. The fact that different memory access patterns lead to tens of times' performance variation in Java HPC programs proves the importance of hiding off-chip memory latency on the

coprocessor. Therefore, it is much more critical to leverage on-chip caches for improving Java performance on Xeon Phi in comparison to normal CPU processors.

Multithreading utilization. Because of the relatively poor single-core performance and the modified P54C-based core design, SMT (Simultaneous Multi-Threading) plays a more important role on Xeon Phi. For well-parallelized Java programs, at least two hardware threads per core should be initialized to hide instruction-issuing latency. In general, the default multi-threading mechanism of HotSpot VM reveals a good scalability for Java HPC programs. Besides, in order to achieve better inter-threading cooperation and maximum chip utilization with more than 120 threads, there still remains further optimizing opportunities in JIT when the instruction streams are generated.

Optimizing solutions. Since HotSpot VM is not "brilliant" enough for MIC architecture so far, there remains much room for improvement. (a) One of the most important reasons for the poor single-threaded performance is not using the 512-bit VPU on Xeon Phi. Enabling vectorization in HotSpot VM would significantly improve the performance of Java HPC programs. (b) Since the high cache miss rate makes the memory latency totally exposed to the array-based Java applications, we can leverage a software prefetching method [14, 16, 17] when processing arrays in the computing kernel loops. (c) HotSpot's JIT does not do any optimization for the in-order execution type, making the instruction streams generated by JIT not well-pipelined on the coprocessor under single thread. Therefore, the corresponding optimization, e.g., to gather the same vector instructions together in unrolling loops, could be promising and expected.

5 Vectorization

5.1 Semi-automatic vectorization

The 512-bit vector unit is an important component to boost the performance for applications running on Xeon Phi. A vector instruction is able to operate 8 double-precision or 16 float-precision elements simultaneously, which brings a great opportunity for array-based Java HPC programs.

5.1.1 *Auto-vectorization in HotSpot*

Automatic vectorization has been enabled since the latest OpenJDK 7u version. The JIT compiler in HotSpot VM can recognize appropriate loops of array-based arithmetic operations. For example:

```
//Loop format that can be vectorized by JIT
for (int i = 0; i < n; i++) {
    a[i] = b[i] * c;
}
```

On traditional x86-based platform that supports Intel® AVX instructions, JIT will translate the above codes into native vector instructions like:

```
...
0xc90600a1: vmovdqu 0x10(%r14,%r13,8), %ymm2
0xc90600a8: vmulpd %ymm1, %ymm2, %ymm2
0xc90600ac: vmovdqu %ymm2, 0x10(%r8,%r13,8)
0xc90600c8: add $0x4, %r13d
0xc90600cc: cmp %edi, %r13d
0xc90600cf: jl 0xc90600a1
...
```

Nevertheless, the auto-vectorization in HotSpot has some restrictions. For example:

1. HotSpot's JIT requires the loop to comply with a very restricted format for vectorization. If the indexes of the array variables do not match the loop counter, JIT will not vectorize the loop:

```
// Constant times a vector, then plus a vector
void daxpy(int n, double da, double dx[], int dx_off, double dy[], int dy_off) {
    for (int i = 0; i < n; i++) {
        dy[i + dy_off] += da * dx[i + dx_off];
    }
}
```

However, this is a very good chance for vectorization from a programmer's standpoint.

2. For a widely-used operation—*reduction* idiom, JIT is not able to vectorize the loop because the left value of the assignment is not an array variable:

```
//Reduction idiom that can not be vectorized by JIT
double sum;
for (int i = 0; i < n; i++) {
    sum += a[i] * b;
}
```

5.1.2 *Implementation of semi-automatic vectorization*

The restrictions of the auto-vectorization in JIT reveal the importance to find a new way for vectorization, which can fully exploit developers' insights. Therefore, we implement a semi-automatic vectorization scheme in HotSpot VM, both for the interpreter and JIT compiler.

Our implementation allows users to specify an annotation before the loop that needs to be vectorized when writing their Java programs. We modify *javac* to identify the loop following the annotation and compile the operations for floating-point arrays in the loop into a list of new bytecodes, namely "vector bytecodes". Meanwhile, the bytecode that controls the loop counter is modified to add 8 (for double-precision, 16 for single-precision) across each iteration. When our optimized HotSpot VM receives the class file, it will parse the new "vector bytecodes" and store them into the code stream, which can be then executed by the interpreter or JIT.

Since the vector instructions for double-precision arithmetic operations on Xeon Phi have 64-byte alignment demand for memory operands, we take measures as below.

First, we need to ensure that the addresses of array objects are 64-byte aligned for the "vector bytecodes". The most convenient way is to indicate a command-line argument provided by HotSpot: "-XX:ObjectAlignmentInBytes=64", which makes all objects allocated in 64-byte aligned addresses.

Second, at the beginning of each Java object, there is a *header* followed by the object data, which contains information important for the JVM. We thus modify the length of array object's header to ensure that the first element of an array is 64-byte aligned.

For the *reduction* operation, we modify *javac* to recognize the loop and do a transformation:

```
//The loop after transformation by our modified javac
double sum;
double[] sum_a = new double[8]; // Initialize with 0.0
for (int i = 0; i < n; i++) {
    sum_a[0] += a[i] * b;
}
sum = sum_a[0] + sum_a[1] + ... + sum_a[7];
```

We use an 8-length double-precision array to replace the *sum* variable. The statement "sum_a[0] += a[i] * b;" will be parsed into "vector bytecodes", which can once operate 512-bit data. This means "(a[i] * b)–(a[i+7] * b)" will be stored into "sum_a[0]–sum_a[7]" accordingly (the stride of each iteration is modified to 8 by *javac*). As the loop ends, we add up the 8 elements in "sum_a" to get the result.

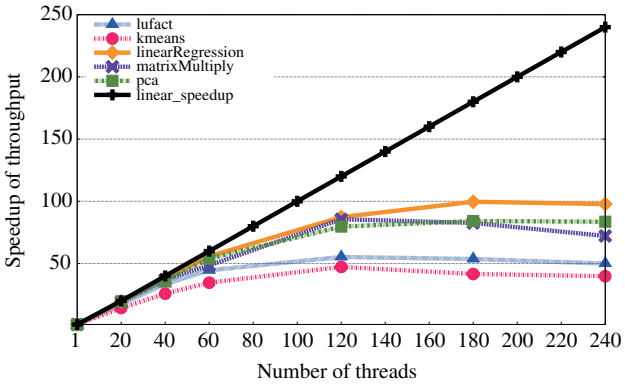

Figure 4 (Color online) Speedup of throughputs of five benchmarks with varying numbers of threads.

During the transformation of a *reduction* operation, we do not perform a memory boundary check for the last few elements in the array *a*. The correctness is guaranteed by the 64-byte alignment mechanism: For a double-precision array whose length is not divisible by 8, the memory space from the end of its last element to the next 64-byte aligned address does not contain any effective data, which is typically filled with zero. Therefore, we do not need to check whether the last iteration has enough elements because zero will be added into "sum_a" for those absences.

5.2 Evaluation

Since vectorization allows no dependency across the loop iterations, we choose *LUFact, KMeans, LinearRegression, MatrixMultiply* and *PCA* as the most appropriate benchmarks for vectorization. We make some transformation in the object placement for *LinearRegression* to make it meet the requirement for vectorization, while leaving the computing logic and result unchanged.

We vectorize the inner loops of these programs and evaluate their throughputs with varying numbers of threads. Figure 4 depicts the scalability curves of the five vectorized benchmarks. It is observed that most of them do not scale well after 60 running threads. This is mainly because that vectorization makes the proportion of parallel part's running time reduced, leaving the sequential part (like synchronization) dominant with big thread counts.

Figure 5 provides the throughput comparison among the three different versions of these array-based applications, including the vectorized version on Xeon Phi, and the original non-vec versions running on both Xeon Phi and CPU processor. In the left group, we compare the single-threaded throughputs on Xeon Phi. As for the right two groups, the best throughput is selected out of different thread counts to form the comparison. For each group, all the results are normalized to the non-vec version.

From the left part we can observe that on Xeon Phi, applying vectorization could achieve averagely 2.5x and maximumly 3.1x speedup of throughput compared to the original version under single thread. Moreover, we have measured the hardware events of the computing kernel modules for these applications on MIC (with and w/o vectorization) under a single-threaded run. As shown in Table 3, the number of retired instructions is significantly reduced, which is due to the fact that the essence of utilizing vector instructions is to reduce the loop iterations. Besides, an obvious decline in L1 cache hit rates is observed for all benchmarks with vectorization. This is reasonable because the 512-bit vector instruction makes the array traversed in a larger stride, e.g., 64 bytes, thus experiencing an L1 cache miss per access.

The right two parts of Figure 5 reveal significant performance improvement with the combination of multi-threading and vectorization for these array-based computing-intensive Java benchmarks on Xeon Phi. Compared to the peak throughput of the original version running on MIC, vectorization could bring up to 2.9x and averagely 1.8x speedup. With the best throughput on CPU as a baseline, averagely 2.7x speedup is achieved with the two key features on Xeon Phi, as shown in the third group. *KMeans* does not perform well on the coprocessor, which is mainly because its original multi-threaded performance is inferior to that of CPU server, and the improvement derived by vectorization is not able to make its

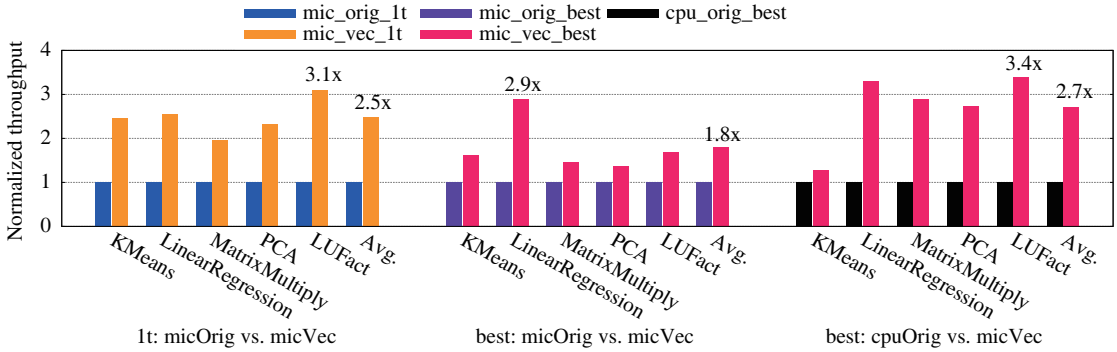

Figure 5 (Color online) Performance gains by vectorization compared to different versions on both architectures.

Table 3 Comparison of hardware events among different versions

Benchmarks	Hardware events	Origin	Vectorization	Prefetching
LUFact	Instructions retired	90440000000	17710000000	21630000000
	L1 hit ratio (%)	91.5	80.7	96.5
KMeans	Instructions retired	9660000000	6160000000	7070000000
	L1 hit ratio (%)	87.8	85.7	88.7
LinearRegression	Instructions retired	10290000000	6230000000	6720000000
(trans)	L1 hit ratio (%)	96.5	88.6	92.0
MatrixMultiply	Instructions retired	4690000000	3080000000	3290000000
	L1 hit ratio (%)	92.1	84.6	89.2
PCA	Instructions retired	18760000000	7280000000	8470000000
	L1 hit ratio (%)	88.5	79.3	91.9

throughput on Xeon Phi outperform CPU that much.

5.3 Lessons learnt

Our semi-automatic vectorization scheme provides a high programmability for users to fully utilize Xeon Phi's strong vector computing power in Java HPC. The significant performance improvement has demonstrated the feasibility of our proposed optimizing strategies, as well as the great potential of HotSpot's JIT on this state-of-the-art many-core architecture. However, the inefficiency in memory access is still a big challenge to the overall performance. Since Xeon Phi relies heavily on software prefetching technique to hide the high memory latency, we explore the prefetching mechanism of the coprocessor correlated with the HotSpot VM, which is introduced in Section 6.

6 Prefetching

Data prefetching is a prevalent memory optimizing technique to fetch data into caches in advance, which could help hide memory access latency and reduce processor stall time. It can be divided into two categories—*hardware prefetching* is implemented by the hardware mechanism inside the processor to determine the specific data prefetching way spontaneously; *software prefetching* is usually performed during compiling process, i.e., compilers automatically issue the prefetching instructions according to the analysis of program data and instruction streams. Since the hardware prefetching mechanism on Xeon Phi is rather limited with some restrictions in multiple scenarios [14], this work mainly focuses on software prefetching approaches to fully boost the memory performance.

In this section, we first implement a semi-automatic software prefetching model in HotSpot's JIT compiler according to an in-depth learning of the default prefetching policies from Intel® *ICC* compiler

that are customized for MIC. The prefetching model is based on our vectorization scheme, which allows users to indicate specific prefetching strategies via JVM command-line arguments. We apply the strategies to the vectorized Java benchmarks and analyze the performance under various combinations. Finally we provide the evaluation result of our prefetching model compared to the original and vectorized versions.

6.1 Prefetching model

By studying the assembly codes generated by Intel® *ICC* compiler with the "-opt-prefetch" option enabled, we found that several particular compiling policies are customized for this new many-core architecture. For example, a two-stage software prefetching algorithm performs best on the coprocessor. *Two-stage* means that data is first brought from memory to L2 cache with a relatively large prefetching distance, and then brought from L2 to L1 cache with a much smaller stride. Particularly, the prefetching distance is an extremely important factor which is correlated with loop structure and iteration times. Moreover, since the cache space on each core is very limited, *ICC* usually issues *clevict* instruction after store operations to manually evict the specific cache line that has just been used.

Based on the observations, we implement a semi-automatic software prefetching model in HotSpot VM, and further evaluate the performance of prefetching in Java HPC programs. When the added JVM argument "-XX:+UsePrefetch" is switched on, our modified HotSpot will recognize all the memory operations in the vectorized loops and generate two prefetching instructions around each corresponding load/store instruction—the first is from memory to L2, the second is from L2 to L1. It is worth noting that in a loop iteration, one array element may be used for multiple times, which means the same memory address may be accessed more than once in JIT's generated instructions. To solve this problem, we modify JIT to recognize and remove all the redundant prefetching nodes (HotSpot's JIT compiler uses an intermediate graph data structure called *ideal graph* to represent the program for most of its optimizations when compiling Java bytecodes to machine code [18]) during the loop transformation phase within its optimization process. Therefore, we can guarantee that all prefetching instructions are issued appropriately.

Meanwhile, our prefetching model provides command-line arguments for users to customize different prefetching strategies in JIT compiler, including the specific two-stage prefetching distances and the eviction of cache lines after memory store instructions. Moreover, for such memory load operation which is followed by a store operation for the same address, the user can indicate whether to prefetch the elements in the load operation exclusively[1)] or not.

6.2 Prefetching policy in Java

We conduct a set of experiments to evaluate the performance with various combinations of these prefetching policies using the five vectorized Java benchmarks. We select two groups of prefetching distances—one group is 64 cache lines (4096 bytes) for memory→L2, 16 lines (1024 bytes) for L2→L1; the other is 32 lines (2048 bytes) for memory→L2, 12 lines (768 bytes) for L2→L1. Other variables include the length of innermost loops and the number of running threads. Small loop length represents thousands of iterations, while big length means more than tens of thousands of iteration times. The thread counts are fixed as one single thread and 60 running threads. For each configuration and prefetching strategy, benchmarks are executed for five times. We report the average throughputs and normalize them to that of non-exclusive and non-clevict policy with (4096, 1024) stride pair.

The results are shown in Table 4. A very significant decline in throughput is observed for most benchmarks with the clevict-policy except *LUFact*. This is because in the other four programs, the *reduction* operation is frequently used in their innermost computing loops. Due to our implementation of the vectorization scheme in HotSpot mentioned in Section 5, the clevict-policy will evict the cache line that holds "sum_a[0]–sum_a[7]", resulting a miss in each store operation for "sum_a" array. While for

1) Exclusive prefetching means data is prefetched to the specific cache level with an invalidation broadcast to all other cores' cache lines that holds this memory data.

Table 4 Normalized throughputs of 5 Java vectorized benchmarks with different prefetching strategies

Benchmarks	Loop length	Threads	Prefetching Stride/bytes (MEM→L2, L2→L1)							
			(4096, 1024)				(2048, 768)			
			None (%)	Exclusive (%)	Clevict (%)	Both (%)	None (%)	Exclusive (%)	Clevict (%)	Both (%)
KMeans	Small	1T	100.00	100.13	77.81	77.30	106.16	105.75	79.78	79.74
		60T	100.00	100.27	90.00	88.86	106.41	107.75	92.88	90.87
	Big	1T	100.00	99.55	79.74	80.21	102.63	102.28	80.14	79.74
		60T	100.00	101.56	90.14	91.78	105.55	99.68	94.78	89.66
Linear-Regression (trans)	Small	1T	100.00	100.00	5.61	5.40	100.35	100.40	5.33	5.60
		60T	100.00	97.76	4.09	4.12	99.16	96.96	4.03	4.23
	Big	1T	100.00	99.91	5.45	5.73	100.39	100.13	5.63	5.09
		60T	100.00	101.52	4.32	4.18	102.74	101.83	4.17	4.04
Matrix-Multiply	Small	1T	100.00	100.48	31.80	32.75	97.64	98.57	31.23	32.73
		60T	100.00	104.02	42.49	42.39	101.78	101.57	42.55	43.17
	Big	1T	100.00	102.01	30.37	34.01	97.83	98.03	30.41	32.82
		60T	100.00	95.83	43.15	42.03	86.53	89.78	40.98	41.63
PCA	Small	1T	100.00	97.66	43.03	37.76	93.63	93.34	38.01	40.61
		60T	100.00	102.28	38.85	37.12	105.87	105.69	38.44	39.34
	Big	1T	100.00	98.81	35.81	38.15	97.50	97.58	36.96	41.67
		60T	100.00	102.92	35.86	34.98	90.94	95.19	34.03	33.39
LUFact	Small	1T	100.00	99.87	97.64	97.63	95.31	94.86	94.94	94.93
		60T	100.00	97.43	98.81	101.13	104.69	104.45	100.98	104.15
	Big	1T	100.00	100.30	96.64	96.43	100.13	100.49	96.65	96.69
		60T	100.00	99.26	99.39	99.09	93.00	92.90	100.92	100.19

Table 5 Correspondence between innermost loop length and prefetching distance

Loop length	Prefetching stride (in cache line)
$[0, 1000)$	No prefetch
$[1000, 2000)$	MEM→L2: 16 lines; L2→L1: 8 lines
$[2000, 5000)$	MEM→L2: 32 lines; L2→L1: 12 lines
$[5000, +\infty)$	MEM→L2: 64 lines; L2→L1: 16 lines

LUFact benchmark, store instructions followed by line eviction may not benefit the throughput either. As for the exclusive-policy, we can see that it has a very tiny impact for our chosen benchmarks.

Despite the strategies mentioned above, we find that the prefetching distance plays a more important role under all circumstances. Comparing the two columns of throughput of (4096, 1024) and (2048, 768) stride pairs with no exclusive- or clevict-policy, about -14.5%–6.4% difference could be observed. The influence of prefetching distance varies a lot with the loop length. An empirical mapping rule can be extracted from our substantial experimental results, which is shown in Table 5.

6.3 Evaluation

We evaluate the throughputs of the five applications using our prefetching model. Prefetching strides are determined according to Table 4. We provide the results with exclusive prefetching and clevict policies off. First, we compare the performance with that of the vectorized versions. Then we show the significant speedups with respect to the original versions on both Xeon Phi and CPU processor. As evaluations in

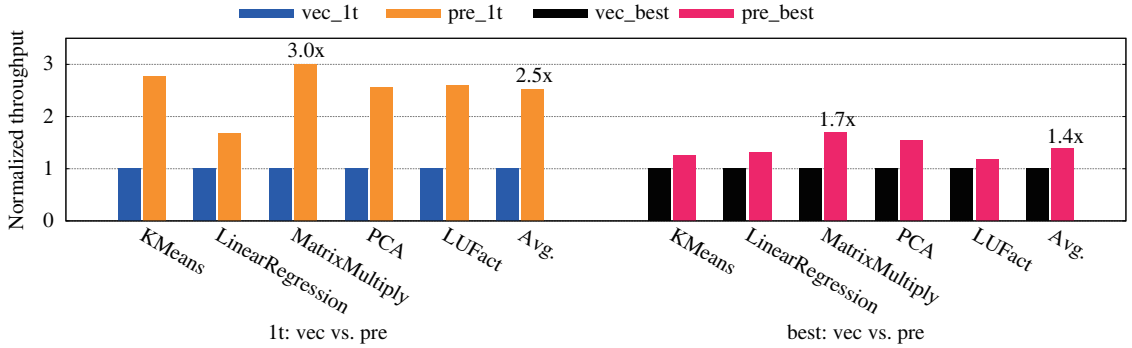

Figure 6 (Color online) Throughput comparison between prefetching and vectorization.

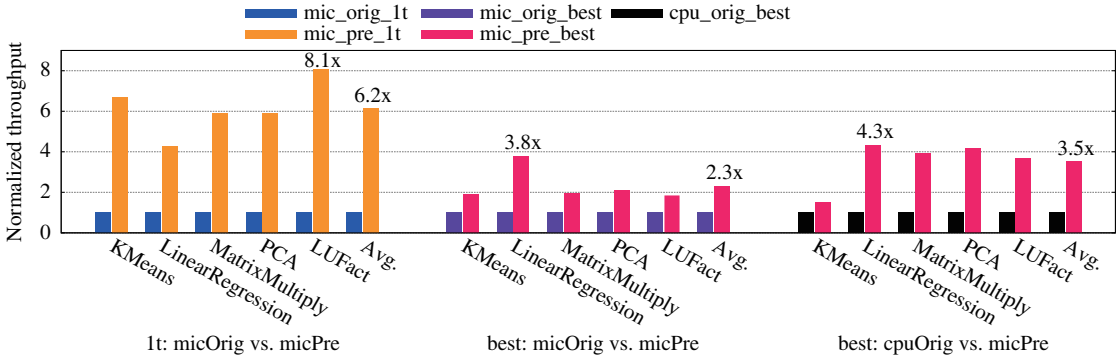

Figure 7 (Color online) Performance gains by prefetching compared to different versions on both architectures.

Section 5, we report the average throughput out of five iterative runs for each benchmark configuration, and normalize the results to that of non-prefetching versions. For the multi-threaded comparison, the best throughput that could be achieved across all executed thread counts is selected.

As illustrated in Figure 6, the left group is the single-threaded comparison with vectorized benchmarks while the right group is for multithreading. It is observed that based on vectorization, our prefetching scheme could bring averagely 2.5x and maximumly 3.0x speedup for single-threaded performance. For multi-threaded runs, the best throughput is averagely 40% more than that of the vectorized version. The performance improvement under multithreading is less notable than single thread, which is mainly caused by two reasons. The first is that a large number of threads manipulating prefetching operations simultaneously may fill up all the MSHRs[2)] on Xeon Phi, making the following outstanding prefetching requests postponed and keep waiting. The second is because of the cache contention that may be brought by data prefetching under SMT environment.

Figure 7 depicts the significant performance improvement derived by prefetching with respect to the original program versions on both architectures. The left group is the single-threaded throughput comparison between the prefetching version and original version on the coprocessor. On the basis of the great improvement with vectorization, the speedup of *LUFact* could reach up to 8.1 times. Besides, an averagely 6.2x speedup is observed. The right two groups compare the best performance improvement with prefetching on both Xeon Phi and CPU server. Averagely 2.3x and 3.5x speedups could be observed in terms of maximum throughput that could be reached on these two different architectures, respectively.

The hardware events profiled by VTune^{TM} are given in Table 3. With a slight increase in retired instructions due to the extra prefetches, the L1 cache miss rates for all applications are significantly reduced compared to the vectorized version, which explains the notable decrease in execution time and confirms the availability of our prefetching model for hiding memory access latency.

2) MSHR - Miss Status Handling Register [19], is a key hardware component that holds the cache miss requests and outstanding prefetches.

7 Related work

Our study is the first comprehensive study of JVM performance on Xeon Phi. This section discusses the most related work to our study, including research with respect to Intel's MIC architecture and Java performance analysis on multi-core and many-core platforms.

Fang et al. [20] perform an empirical study of Xeon Phi with a micro-benchmark suite, stressing both the potential and limits of the key performance factors that are not provided by Intel's data sheet. Some of our analysis is based on their findings. In [17], the authors focus on how to hide memory latencies and save bandwidth with software prefetching and special store instructions on the coprocessor. Fang [14] and Mehta [16] have studied different prefetching strategies and find a coordinated multi-stage prefetching policy that yields the best performance on Xeon Phi. Such studies give us hints to handle the performance problems of the memory system when running Java HPC benchmarks on the coprocessor. There are also studies that focus on the performance analysis and optimization for specific HPC applications on Xeon Phi, such as NAS parallel benchmarks [21] and Linpack benchmark [22].

For multi-core platforms, Eyerman et al. [23] give a comprehensive study about the performance of thread-level parallelism of different homo- or hetero-geneous multi-core designs, with or w/o SMT. Their results claim that many small cores are inferior to big multi-cores, some of which are in line with our results. But we additionally show that the weakness of small cores could be mitigated, or even to the point of getting better performance than big multi-cores via vectorization and massive threading.

Plenty of studies explore Java on general multi-core platforms. In [24], the authors focus on the performance and scalability issues of Java programs on modern multi-cores, including low-level hardware measurements and tuning techniques. However, their experiments do not cover large-enough thread count due to hardware limits. Gidra et al. [25] study the scalability problems of OpenJDK's default Parallel Scavenge garbage collector, and fix the bottlenecks with some well-established parallel programming techniques on NUMA multi-cores.

There are studies focusing on running Java applications on GPU. Yan et al. [26] present a programming interface called JCUDA, allowing Java programmers to invoke CUDA kernels. In [27], the authors provide an evaluation of Aparapi, which enables the data-parallel fragments in Java programs to be executed on GPGPUs. These studies convert Java bytecode into CUDA or OpenCL at runtime, but not run a managed runtime on many-core architectures like GPU.

8 Conclusion

This paper presents the first comprehensive study on the performance of JVM using high-performance computing applications on Intel MIC architecture. Based on our porting of OpenJDK to Xeon Phi that builds a complete Java environment, we analyze the performance issues using the metrics of throughput and scalability. To fully utilize the abundant computing power of Xeon Phi, we implement a semi-automatic vectorization scheme in HotSpot. Further, we propose a software prefetching model to hide the memory access latency. We show that, with the combination of the key features and our optimizing solutions, e.g., massive threading, vectorization and prefetching, it can bring significant performance improvement for the Java HPC applications that have abundant data parallelism and computing tasks.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Chrysos G. Intel® Xeon Phi™ Coprocessor-the Architecture. Intel Whitepaper, 2014
2 Shafi A, Carpenter B, Baker M. Nested parallelism for multi-core HPC systems using Java. J Parall Distrib Comput, 2009, 69: 532–545
3 Moreira J E, Midkiff S P, Gupta M, et al. NINJA: Java for high performance numerical computing. Sci Program, 2002, 10: 19–33
4 Amedro B, Bodnartchouk V, Caromel D, et al. Current state of Java for HPC. Technical Report RT-0353. INRIA, 2008

5 O'Mullane W, Luri X, Parsons P, et al. Using Java for distributed computing in the Gaia satellite data processing. Exp Astron, 2011, 31: 243–258
6 Taboada G L, Touriño J, Doallo R. Java for high performance computing: assessment of current research and practice. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. New York: ACM, 2009. 30–39
7 Boisvert R F, Moreira J, Philippsen M, et al. Java and numerical computing. Comput Sci Eng, 2001, 3: 18–24
8 Guide P. Intel® 64 and IA-32 Architectures Software Developer's Manual. 2010
9 Blumofe R D, Joerg C F, Kuszmaul B C, et al. Cilk: an efficient multithreaded runtime system. In: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 1995. 207–216
10 Lindholm T, Yellin F, Bracha G, et al. The Java Virtual Machine Specification. 8th ed. Redwood City: Pearson Education, 2014
11 Intel. Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual. 2012
12 Smith L A, Bull J M, Obdrizalek J. A parallel Java grande benchmark suite. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing. New York: ACM, 2001. 8
13 Ranger C, Raghuraman R, Penmetsa A, et al. Evaluating MapReduce for multi-core and multiprocessor systems. In: Proceedings of IEEE 13th International Symposium on High Performance Computer Architecture. Washington, DC: IEEE, 2007. 13–24
14 Fang Z, Mehta S, Yew P C, et al. Measuring microarchitectural details of multi-and many-core memory systems through microbenchmarking. ACM Trans Architect Code Optim, 2015, 11: 55
15 Intel. Intel® Xeon Phi™ Coprocessor System Software Developers Guide. 2013
16 Mehta S, Fang Z, Zhai A, et al. Multi-stage coordinated prefetching for present-day processors. In: Proceedings of the 28th ACM International Conference on Supercomputing. New York: ACM, 2014. 73–82
17 Krishnaiyer R, Kultursay E, Chawla P, et al. Compiler-based data prefetching and streaming non-temporal store generation for the Intel® Xeon Phi™ coprocessor. In: Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Cambridge, 2013. 1575–1586
18 Wurthinger T, Wimmer C, Mossenbock H. Visualization of program dependence graphs. In: Proceedings of the Joint European Conferences on Theory and Practice of Software and the 17th International Conference on Compiler Construction. Berlin/Heidelberg: Springer-Verlag, 2008. 193–196
19 Tuck J, Ceze L, Torrellas J. Scalable cache miss handling for high memory-level parallelism. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC: IEEE, 2006. 409–422
20 Fang J, Varbanescu A L, Sips H, et al. An empirical study of Intel Xeon Phi. arXiv:1310.5842
21 Ramachandran A, Vienne J, van der Wijngaart R, et al. Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi. In: Proceedings of the 42nd International Conference on Parallel Processing, Lyon, 2013. 736–743
22 Heinecke A, Vaidyanathan K, Smelyanskiy M, et al. Design and implementation of the linpack benchmark for single and multi-node systems based on Intel® Xeon Phi™ coprocessor. In: Proceedings of IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), Boston, 2013. 126–137
23 Eyerman S, Eeckhout L. The benefit of SMT in the multi-core era: flexibility towards degrees of thread-level parallelism. ACM SIGARCH Comput Architect News, 2014, 42: 591–606
24 Chen K Y, Chang J M, Hou T W. Multithreading in Java: performance and scalability on multicore systems. IEEE Trans Comput, 2011, 60: 1521–1534
25 Gidra L, Thomas G, Sopena J, et al. A study of the scalability of stop-the-world garbage collectors on multicores. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2013. 229–240
26 Yan Y H, Grossman M, Sarkar V. JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA. In: Proceedings of the 15th International Euro-Par Conference on Parallel Processing. Berlin/Heidelberg: Springer-Verlag, 2009. 887–899
27 Docampo J, Ramos S, Taboada G L, et al. Evaluation of Java for general purpose GPU computing. In: Proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshops. Washington, DC: IEEE, 2013. 1398–1404