OpenMP-based parallel implementation of matrix-matrix multiplication on the Intel Knights Landing

Roktaek Lim
Soongsil University
Seoul, Korea
rokt.lim@gmail.com

Yeongha Lee
Soongsil University
Seoul, Korea
yeongha22@gmail.com

Raehyun Kim
Seoul National University
Seoul, Korea
ab3169@snu.ac.kr

Jaeyoung Choi
Soongsil University
Seoul, Korea
choi@ssu.ac.kr

ABSTRACT

The second generation Intel Xeon Phi processor codenamed Knights Landing (KNL) have emerged with 2D tile mesh architecture. Implementing of the general matrix-matrix multiplication on a new architecture is an important practice. To date, there has not been a sufficient description on a parallel implementation of the general matrix-matrix multiplication. In this study, we describe the parallel implementation of the double-precision general matrix-matrix multiplication (DGEMM) with OpenMP on the KNL. The implementation is based on the blocked matrix-matrix multiplication. We propose a method for choosing the cache block sizes and discuss the parallelism within the implementation of DGEMM. We show that the performance of DGEMM varies by the thread affinity environment variables. We conducted the performance experiments with the Intel Xeon Phi 7210 and 7250. The performance experiments validate our method.

KEYWORDS

High-performance, Knights Landing, Matrix-matrix multiplication, Manycore, OpenMP Affinity

ACM Reference Format:
Roktaek Lim, Yeongha Lee, Raehyun Kim, and Jaeyoung Choi. 2018. OpenMP-based parallel implementation of matrix-matrix multiplication on the Intel Knights Landing. In *HPC Asia 2018 WS: Workshops of HPC Asia 2018, January 31, 2018, Chiyoda, Tokyo, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3176364.3176374

1 INTRODUCTION

Hardware technologies based on manycore architecture are common to most platforms. The Intel launched the second generation Intel Xeon Phi processor codenamed Knights Landing (KNL). It has several different features from the previous Xeon Phi processor codenamed Knights Corner (KNC). The KNL and the KNC may require different algorithms to solve the same problem because their microarchitectures are different. To achieve good performance of applications on the KNL, it is necessary to understand an efficient algorithm for applications as well as to utilize the KNL architecture to improve the performance of applications.

The present study aims to describe in detail the parallel implementation of the double-precision general matrix-matrix multiplication (DGEMM) with OpenMP on the KNL. The parallel implementation is based on the blocked matrix-matrix multiplication. Our study focuses on how to choose the proper block sizes, how to achieve parallelism in the blocked matrix-matrix multiplication, and how to set the thread affinity to get high-performance of DGEMM on the KNL. It is well-known that the block sizes are critical to the performance of DGEMM [1, 6, 8, 10, 11]. Marker et al. [7] showed that it is necessary to parallelize in multiple dimensions when many threads are used for the matrix-matrix multiplication. Since the KNL is made up of tiles and two cores on each tile share 1 MB L2 cache, it is important to pay attention to data placement in L2 cache and to optimized usage of shared L2 cache. The Intel OpenMP Runtime library and OpenMP Affinity are used to control allocation of hardware resources and to bind OpenMP threads to physical core.

The rest of the paper is organized as follows. Section 2 introduces the hardware and software used in this study. Section 3 describes implementation details of parallel DGEMM on the KNL. In Section 4, we present performance results. we conclude in Section 5.

2 HARDWARE AND SOFTWARE DESCRIPTIONS

We tested our experiments on the following two systems:

- The first system is equipped with intel Xeon Phi Processor 7210 has 64 cores, each running at 1.3 GHz. This delivers 41.6 GFLOPS (1.3 GHz × 2 VPUs × 8 doubles × 2 IPC) of double precision performance per core. The system is running CentOS version 7.2.1511 (kernel 3.10.0-327.13.1). The system is configured in the Flat/Quadrant mode. The Intel Parallel Studio XE 2017 Update 4 is installed on the system.
- The second system is equipped with intel Xeon Phi Processor 7250 has 68 cores, each running at 1.4 GHz. This delivers 44.8 GFLOPS (1.4 GHz × 2 VPUs × 8 doubles × 2 IPC) of double

HPC Asia 2018 WS, January 31, 2018, Chiyoda, Tokyo, Japan
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6347-1/18/01…$15.00
https://doi.org/10.1145/3176364.3176374

ALGORITHM 1: Matrix-matrix multiplication algorithm.

```
for $p = 0, \ldots, k-1$ in steps of $k_b$ do
    Pack $B_p$ into $\widetilde{B}$;
    for $i = 0, \ldots, m-1$ in steps of $m_b$ do
        Pack $A_{i,p}$ into $\widetilde{A}$;
        for $jr = 0, \ldots, n-1$ in steps of $n_r$ do
            for $ir = 0, \ldots, m_b - 1$ in steps of $m_r$ do
                $\widehat{A} = \widetilde{A}_{ir}$;
                $\widehat{B} = \widetilde{B}_{jr}$;
                $\widehat{C} \mathrel{+}= \widehat{A} \times \widehat{B}$;
                Update $C$ using $\widehat{C}$;
            end
        end
    end
end
```

precision performance per core. The system is configured in the Flat/Hemisphere mode. The system is running CentOS version 7.2.1511 (kernel 3.10.0-327.13.1). The Intel Parallel Studio XE 2017 Update 1 is installed on the system.

3 IMPLEMENTATION OF DGEMM

In this section, we briefly describe our implementation of the double-precision general matrix-matrix multiplication, $C = \alpha A \times B + \beta C$, where A, B, and C are $m \times k$, $k \times n$, and $m \times n$ matrices, respectively, while α and β are scalars. Our implementation assumes that the matrices are stored in row-major order. The matrix multiplication algorithm developed by [1] is used in this study. It is a blocking algorithm for matrix-matrix multiplication to amortize the cost of moving data between memory layers of an architecture with a multi-level memory. The algorithm that we implemented is presented in Algorithm 1 and the illustration of Algorithm 1 is provided in Figure 1.

3.1 Cache and register block sizes

The most compute intensive part in Algorithm 1 is the inner kernel that computes $\widehat{C} \mathrel{+}= \widehat{A} \times \widehat{B}$. The AVX-512 instructions and several optimization techniques such as cache blocking, prefetching, and loop unrolling are used to maximize the performance of the inner kernel. Our optimized implementation of the inner kernel for the KNL is presented in the previous study [5]. Our implementation on a single core achieved up to 99 percent of DGEMM using the Intel MKL, which is the current state-of-the-art library. For the register block sizes, we observed that good performance results were obtained with the full use of available vector registers when $(m_r, n_r) = (31, 8)$. In this study, we choose $m_r = 31$ and $n_r = 8$. For a single core on the KNL, our implementation requires $\widetilde{A}$, $\widehat{B}$, and $\widehat{C}$ to fit in the 1MB L2 cache. The best performance of our implementation is obtained, when $(m_b, k_b) = (31, 1620)$. When $(m_r, n_r, k_b, m_b) = (31, 8, 1620, 31)$, the size of $(\widetilde{A} + \widehat{B} + \widehat{C})$ is about 495.5 KB. And the performance experiments in [5] showed that good performance results were obtained when we used the half of the L2 cache to store $\widetilde{A}$, $\widehat{B}$, and $\widehat{C}$. Now we discuss the cache

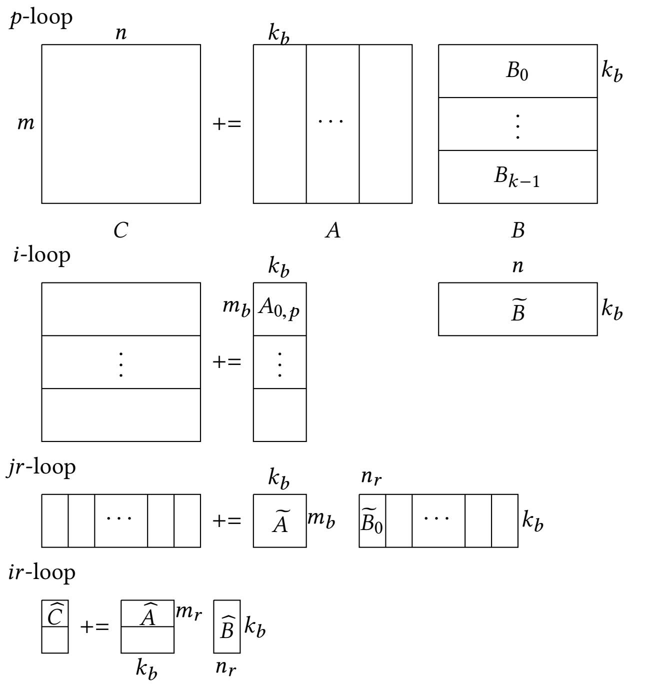

Figure 1: Illustration of Algorithm 1.

block size for the parallel implementation of DGEMM on the KNL. Conservatively, there must be room for storing $2(\widetilde{A} + \widehat{B} + \widehat{C})$ in the L2 cache because two cores on a tile shares the L2 cache. Then, we have to choose m_b and k_b so that

$$2 \times (m_b k_b + k_b n_r + m_r n_r) \times 8 \text{ bytes} \leq 512 \text{ KB}. \qquad (1)$$

If we parallelize the jr-loop only or carefully parallelize both the i- and jr-loop, two cores on the same tile require the same $\widetilde{A}$ to compute the inner kernel. If both cores on the same tile read the same cache line, there will only be a single copy of the cache line in the L2 cache.

$$[m_b k_b + 2 \times (k_b n_r + m_r n_r)] \times 8 \text{ bytes} \leq 512 \text{ KB}. \qquad (2)$$

3.2 Loop to parallelize

The i-loop and jr-loop in Algorithm 1 are appropriate to parallelize. Since the i-loop has steps of m_b, this loop has m/m_b iterations. If m is large, the i-loop provides a good opportunity for parallelism. Since n_r is 8, the jr-loop has n/n_r iterations and also provides a good opportunity for parallelism. The previous studies [3, 9] claimed that parallelizing both the i-loop and jr-loop can increase data sharing and reduce sychronization cost. If we parallelize the jr-loop only, the cost of packing the block of A from main memory into the L2 cache can not be amortized. If we parallelize the i-loop only, all cores will only work when m is a multiple of $m_b \times$ (64 or 68). Thus, we parallelize both the i- and jr-loop in this study. In the previous study [5], we obtained good performance when the i-loop is parallelized with 17 threads and jr-loop is parallelized with 4 threads for the Intel Xeon Phi Processor 7250. We parallelize the

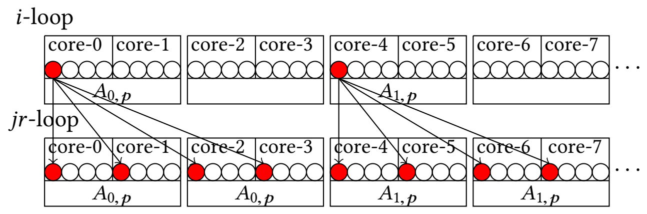

Figure 2: Thread placement with nested parallelism when OpenMP Affinity is used.

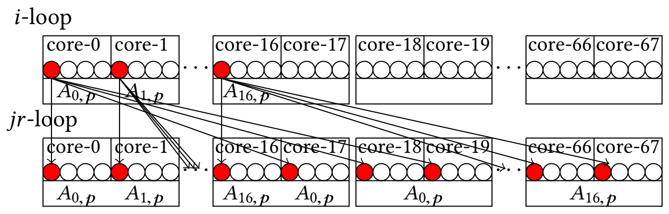

Figure 3: Thread placement with nested parallelism when KMP_AFFINITY=scatter is used.

i-loop with 16 threads and parallelize the *jr*-loop with 4 threads when the Intel Xeon Phi Processor 7210 is used.

3.3 Thread affinity

Our parallel implementation of DGEMM on the KNL contains two nested loops, *i*- and *jr*-loop. The thread affinity is crucial in order to achieve high performance with nested OpenMP. Nesting OpenMP parallel regions involve the repeated high overheads of creating and destroying the threads. Avoiding the repeated overhead of creating and destroying the threads can lead to efficient nesting and can improve application performance. The Intel OpenMP provides runtime environment variable, hot teams that is able to reduce these overheads, by keeping a pool of threads alive (but idle) during the execution of the nonnested parallel code [4]. The use of hot teams is controlled by two environment variables:

- KMP_HOT_TEAMS_MODE
- KMP_HOT_TEAMS_MAX_LEVEL

We set KMP_HOT_TEAMS_MODE=1 to keep unused team members alive and set KMP_HOT_TEAMS_MAX_LEVEL=2 since our implementation has two levels of parallelism.

According to [4], only one thread per core is enough to reach the maximum performance of the DGEMM for the KNL. OpenMP Affinity and the Intel OpenMP runtime library can be used to assign one thread per core on the KNL. The following settings are used to allocate hardware resources and pin OpenMP threads to hardware resources.

OpenMP Affinity
- OMP_PLACES: allocate hardware threads
- OMP_PROC_BIND: pin OpenMP threads to hardware threads

Intel OpenMP runtime library
- KMP_PLACES_THREADS: allocate hardware threads
- KMP_AFFINITY: pin OpenMP threads to hardware threads

OpenMP Affinity allocates hardware threads in nested parallel regions differently than the Intel OpenMP runtime library does. For assigning one thread per core with OpenMP Affinity, we set the environment variables as follows:

- OMP_PLACES = cores
- OMP_PROC_BIND = spread, spread

The thread placement in the *i*- and *jr*-loop are given in in Figure 2. Nearby two tiles are grouped together in teams of four threads. Thus, we can use (2) to choose m_b and k_b since it is necessary to store only one $\widetilde{A}$ in the L2 cache. For assigning one thread per core with the Intel OpenMP runtime library, we set the environment variables as follows:

- KMP_AFFINITY = scatter

When the Intel OpenMP runtime library is used, the thread placement in the *i*- and *jr*-loop are given in Figure 3. The Intel OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. However, it is known that the use of the high-level affinity interface, KMP_AFFINITY is limited in programs containing nested parallelism [2]. As a result, two cores on the same tile needs different $\widetilde{A}$ to compute the inner kernel when KMP_AFFINITY=scatter is used to assign one threads per core on the KNL. We have to use (1) to choose m_b and k_b since it is necessary to store two different $\widetilde{A}$ in the L2 cache for some tiles.

k_b should be picked to be as large as possible to amortize the cost of updating $m_r \times n_r$ elements of $\widehat{C}$. If m_b, m_r, and n_r are fixed, we can have larger k_b if we use OpenMP Affinity. Hence, we should set the environment variables with OpenMP Affinity to obtain high-performance of DGEMM.

4 EXPERIMENTS

The best performance of our implementation is obtained for a single core on the KNL, when $(m_b, k_b, m_r, n_r) = (31, 1620, 31, 8)$. However, we should consider the required memory bandwidth when we implement DGEMM on the KNL. The required memory bandwidth for computing $m_b \times n$ block of C is the ratio between the memory traffic and compute time. To update $m_b \times n$ block of C, $(m_b n + m_b k_b + k_b n) \times 8$ bytes of data moves from main memory to the L2 cache and $(m_b n) \times 8$ bytes of data moves from registers to main memory for DGEMM. Since the inner kernel use the fused-multiply add instruction and there are two VPUs in each core, a core on the KNL requires $(m_b \times n \times k_b)/16$ cycles to compute $m_b \times n$ block of C. Thus, the required memory bandwidth is given by

$$128\left(\frac{2}{k_b} + \frac{1}{n} + \frac{1}{m_b}\right) \text{ bytes/cycles.} \tag{3}$$

For large values of n, $\frac{1}{n}$ in (3) is negligible. The required memory bandwidth is 6 GB/sec (1.4 GHz × 4.3 bytes/cycles) when $m_r = 31$, $n_r = 8$, $k_b = 1620$, and $m_b = 31$. Since the DDR4 memory on the KNL is capable of delivering up to 90 GB/sec and MCDRAM is slightly higher latency than DDR4 memory, we do not need to use the MCDRAM in our experiments for a single core on the KNL. If we implement DGEMM on the KNL with the cache and register block sizes for a single core on the KNL, the required memory bandwidth will be 408 GB/sec. Thus, we should use the MCDRAM to

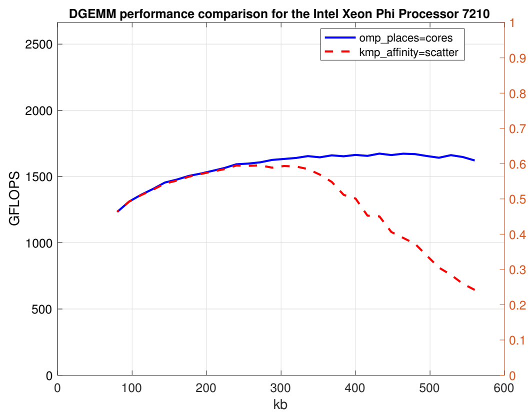

Figure 4: DGEMM performance comparison for the Intel Xeon Phi Processor 7210.

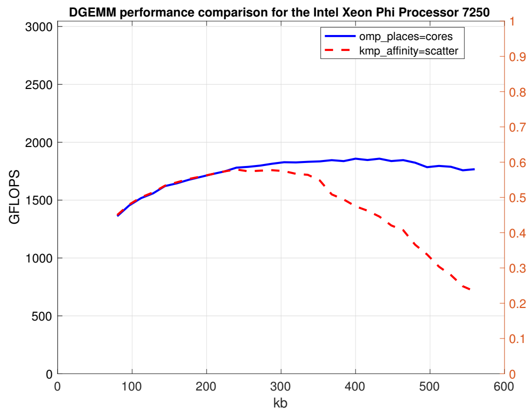

Figure 5: DGEMM performance comparison for the Intel Xeon Phi Processor 7250.

implement parallel DGEMM on the KNL. Memory latency increases with memory bandwidth for the MCDRAM [4]. The cache block sizes m_b and k_b should be resized to reduce memory latency. m_b is chosen to be 124. k_b will be determined using (1) or (2). If we control hardware resources with the Intel OpenMP runtime library, the optimal k_b is less than 256. If we control hardware resources with OpenMP, the optimal k_b is less than 480. If $k_b = 240$, the required memory bandwidth is about 200 GB/sec. Thus, we should put $\widetilde{A}$, $\widetilde{B}$, and $\widehat{C}$ in the MCDRAM, when we implement parallel DGEMM on the KNL.

We conduct performance experiments to choose k_b. In the performance experiments, we vary k_b from 80 to 560 and use $m = n = 20,000$ and $k = 5,000$. The performance results for the Intel Xeon Phi Processor 7210 and 7250 are given in Figure 4 and 5, respectively. When we use OpenMP Affinity, the optimal k_b value is 432 for both the Intel Xeon Phi Processor 7210 and 7250. When we use the Intel OpenMP runtime library, the optimal k_b values are 272 and 240 for the Intel Xeon Phi Processor 7210 and 7250, respectively. These results are in the line with our expectations.

5 CONCLUSIONS AND FUTURE DIRECTIONS

In this study, we described in detail the parallel implementation of the double-precision general matrix-matrix multiplication (DGEMM) with OpenMP on the KNL. We discussed the proper cache block sizes for the KNL and proposed the method for choosing the cache block sizes. We have presented the appropriate loops to be parallelized. It was shown that levels of parallelism in the DGEMM implementation on the KNL is the key to high-performance. We discussed that thread placement is important to achieve high-performance for the KNL. We improved parallel performance by using OpenMP Affinity. We conducted the performance experiments to validate our method for choosing the cache block sizes. They suggested that the method is reasonable and effective.

There are still chances for further studies on the precise choice of prefetch distances for the block matrices. We need to develop a highly optimized routine for packing B into $\widetilde{B}$. Since $\widetilde{B}$ is typically large, packing of $\widetilde{B}$ requires both reading from and writing to memory. A routine for the packing of $\widetilde{B}$ should be considered to improve the performance of packing of DGEMM.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. The work was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Korea Ministry of Science and ICT (MSIT) (2015M3C4A7065662).

REFERENCES

[1] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 12.

[2] Guide for Intel C++ Compiler 2015. User and reference Guide for the Intel C++ Compiler 15.0. (2015). https://software.intel.com/en-us/node/522691

[3] Murat Efe Guney, Kazushige Goto, Timothy B Costa, Sarah Knepper, Louise Huot, Arthur Mitrano, and Shane Story. 2017. Optimizing Matrix Multiplication on Intel® Xeon Phi x200 Architecture. In *Computer Arithmetic (ARITH), 2017 IEEE 24th Symposium on.* IEEE, 144–145.

[4] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition.* Morgan Kaufmann.

[5] Roktaek Lim, Yeongha Lee, Raehyun Kim, and Jaeyoung Choi. Submitted. An implementation of matrix-matrix multiplication on the Intel KNL processor with AVX-512. *Cluster Computing* (Submitted).

[6] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, 2 (2016), 12.

[7] Bryan Marker, Field G Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A Van De Geijn. 2007. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel Processing.* Springer, 748–757.

[8] Jonathan Lawrence Peyton. 2013. *Programming Dense Linear Algebra Kernels on Vectorized Architectures.* Master's thesis. The University of Tennessee, Knoxville.

[9] Tyler M Smith, Robert A Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, 1049–1059.

[10] R Clint Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 1–27.

[11] R Clint Whaley and Antoine Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35, 2 (2005), 101–121.