이학석사 학위논문

Implementing general matrix-matrix multiplication algorithm on the Intel Xeon Phi Knights Landing Processor

(Intel Xeon Phi Knights Landing 프로세서에서의 일반 행렬 곱셈 알고리즘 구현)

2018년 2월

서울대학교 대학원

수리과학부

김 래 현

Implementing general matrix-matrix multiplication algorithm on the Intel Xeon Phi Knights Landing Processor

A dissertation
submitted in partial fulfillment
of the requirements for the degree of
Master of Science
to the faculty of the Graduate School of
Seoul National University

by

Raehyun Kim

Dissertation Director : Professor Dongwoo Sheen

Department of Mathematical Sciences
Seoul National University

February 2018

Abstract

This paper presents the design and implementation of general matrix-matrix multiplication (GEMM) algorithm for the second generation Intel Xeon Phi processor codenamed Knights Landing (KNL). We illustrate several developing guidelines to achieve optimal performance with C programming language and the Advanced Vector Extensions (AVX-512) instruction set. Further, we present several environment variable issues associated with parallelization on the KNL. On a single core of the KNL, our double-precision GEMM (DGEMM) implementation achieves up to 99 percent of DGEMM performance using the Intel MKL, which is the current state-of-the-art library. Our parallel implementation for 68 cores of the KNL also achieves good scaling results, up to 93 percent of DGEMM performance using the Intel MKL.

Key words: Knights Landing, general matrix-matrix multiplication, vectorization, optimization
Student Number: 2016-20230

Contents

CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

In many numerical problems, matrix and vector operations take central part of algorithms. As computing power increases, effective implementation of these operations becomes an important task to achieve optimal performance on a target machine. Researchers clarified matrix and vector operations and developed Basic Linear Algebra Subprograms (BLAS). Lawson et al. [9] presented level 1 BLAS which are vector-vector operations and Dongarra et al. [2, 3, 4] developed level 2 BLAS and level 3 BLAS which are matrix-vector and matrix-matrix operations, respectively.

General matrix-matrix multiplication (GEMM) algorithm plays a fundamental role in effective implementation of level 3 BLAS. In level 3 BLAS, GEMM or modified GEMM such as SYMM takes the most compute intensive part of the operation. Optimal implementations of GEMM on various architectures have been studied in the papers [1, 5, 7, 14, 16, 17]. The key idea of effective GEMM implementation is blocking which divides matrices into blocks to utilize the hierarchical memory organization of the target machine. Systematic analysis on the blocked GEMM algorithm is presented in the literature [5, 6].

In this paper, we describe in detail the implementation of GEMM algorithm for the second generation Intel Xeon Phi processor codenamed Knights Landing (KNL). We illustrate several developing strategies to achieve good performance on the KNL. We carry out GEMM to analyze the structure of the KNL and find out restrictions and machine conditions to exploit the pow-

erful calculation performance of the KNL. Our implementation is coded with C programming language and the Advanced Vector Extensions(AVX-512) instruction set.

This research focuses on two main goals. The first one is the efficient implementation of GEMM algorithm on the KNL. Since GEMM is a key building block for other level 3 BLAS, the high performance of GEMM algorithm leads to the good performance of other level 3 BLAS. The second one is to suggest developing guidelines to achieve optimal performance on the KNL. The KNL is a brand-new many-core architecture which has a quite complicated structure. For other KNL users, we offer several restrictions for optimization and present environment variable issues.

To reach the optimal performance of applications on the KNL, it is essential to understand an efficient algorithm for applications as well as to utilize the KNL architecture to improve the performance of applications. From a mechanical point of view, the way to achieve high performance on the KNL is that the given application maximizes the data-reuse of the caches and exploits the powerful vector processing units (VPU) by vectorization. Reuse of data in the cache increases flops per memops to overcome the memory bandwidth limitation which is a major obstacle to fully utilize the computing power of the KNL. To maximize data reusing, the given algorithm has to divide requiring data into appropriate size of pieces so that they can fit in the caches.

On the other hand, vectorization helps the algorithm to efficiently use the 512-bit vector registers on the KNL. For effective vectorization, we need to properly align the working data for optimal vector load and store, and the size of data is large enough to make use of the vector capabilities. The storing order and proper size of working data are determined by the size of caches and registers. Our implementation includes individual packing kernel to reorder the requiring data.

The rest of the paper is organized as follows. In Chapter 2, we describe the hardware and software features of the target machine, the Intel KNL processor. In Chapter 3, GEMM algorithm based on register and cache blocking is presented. Also, we discuss about the parallelization potential of GEMM. The optimization process and performance results of double-precision GEMM al-

gorithm is presented in Chapter 4. We implemented double-precision GEMM algorithm for a single core first, and parallelized this algorithm for multiple cores in the KNL. Finally, Chapter 5 presents our conclusions and future directions.

This work is based on the collaborative research [11] with Roktaek Lim, Yeongha Lee and Jaeyoung Choi. Our 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)

Chapter 2

Hardware and software descriptions

2.1 Hardware description

We have tested our implementation of double-precision GEMM on the Intel Xeon Phi Processor 7250 with 68 cores, running at 1.4 GHz. The theoretical peak of the KNL is up to 44.8 GFLOPs (1.4 GHz $\times$ 2 VPUs $\times$ 8 doubles $\times$ 2 IPC) in double-precision computation per core.

On the KNL, 34 tiles are interconnected by 2D mesh. Each tile contains 2 cores and 4 VPUs (2 VPUs per core). In tile, both cores have 32 KB first level data cache (L1 cache) individually, and share 1 MB second level data cache (L2 cache), which is globally coherent by Caching/Home Agent (CHA) via directory-based MESIF protocol. Each core has 32 512-bit vector registers and its VPU executes 16-wide single-precision or 8-wide double-precision SIMD instructions in a single cycle.

One of the most important features of the KNL is Multi-Channel DRAM (MCDRAM). MCDRAM is a version of High Bandwidth Memory which has up to 450GB/s of bandwidth while DDR4 SDRAM has up to 90GB/s of bandwidth. On the KNL processor, 16GB of MCDRAM is integrated on package. Depending on memory usage, there are three kinds of memory modes, Cache Mode, Flat Mode, Hybrid Mode. In this paper, we select Flat Mode to utilize MCDRAM optimally. Since MCDRAM has slightly higher latency than

DDR4 SDRAM, it may not be beneficial to use MCDRAM for low bandwidth algorithm. On the other hand, MCDRAM helps to enhance performance for programs which need high bandwidth such as level 3 BLAS routines,

2.2 Software description

The Intel KNL processor is the first self-boot Intel Xeon Phi processor. The system is running CentOS version 7.2.1511(kernel 3.10.0-327.13.1). To compare the performance of GEMM algorithm, we use two of BLAS libraries, the Intel Math Kernel Library (MKL) and the BLAS-like Library Instantiation Software (BLIS). The version of the MKL is the Intel Parallel Studio XE 2017 Update 1 and the version of BLIS is 0.2.2.[1]

On the KNL, there are 32 vector registers, 512 bits-wide, per each core. To modify these vector registers, the KNL offers Advanced Vector Extensions (AVX-512) instruction set. Instruction is also called as intrinsic. The instruction set contains common arithmetic instructions, such as fused multiply-add(FMA), and register modifying instructions, such as load, store, prefetch. To utilize the vector registers, we use AVX-512 instruction set rather than straight assembler language.

The AVX-512 instructions are slightly higher-level macros which wrap around the assembly language constructs. Unlike assembly language, they allow common variable declarations rather than explicit allocation on the registers. The compiler takes care of precise adjustments for variable allocation. That is, the AVX-512 instruction set takes advantage of high efficiency of using assembly language without most time consuming part. Listing 2.1 and Listing 2.2 provide an example code of the AVX-512 instruction and an assembly code generated by the compiler Note that, the basic computation unit of the AVX-512 instruction is a 64 bytes wide cache line which consists of 8 double-precision or 16 single-precision floating point elements.

```
register __m512d a, b, c;
a = _mm512_load_pd(A);
b = _mm512_load_pd(B);
c = _mm512_load_pd(C);
c = _mm512_fmadd_pd(a, b, c);   // c += a * b
_mm512_store_pd(C, c);
```

Listing 2.1: AVX-512 instruction code example

[1] https://github.com/flame/blis

```
vmovaps          256(%rsp), %zmm1
vmovaps          320(%rsp), %zmm2
vmovaps          384(%rsp), %zmm3
vfmadd231pd      %zmm1, %zmm2, %zmm3
vmovaps          %zmm3, 384(%rsp)
```

Listing 2.2: Assembly code generated by code 2.1

Chapter 3

Implementing algorithms

In this chapter, we deal with the double-precision matrix-matrix multiplication (DGEMM) algorithm. Our DGEMM algorithm consists of blocked matrix multiplication, highly optimized inner kernel, and packing algorithm. We use AVX-512 instructions to build inner kernel, and C language to code blocked matrix multiplication part and packing algorithm.

We focus on the special case $C = A \times B + C$, where A, B, and C are $m \times k$, $k \times n$, and $m \times n$ matrices, respectively. It can be easily expanded to the general case $C = \alpha A \times B + \beta C$. Our implementation assumes that all matrices are stored in one dimensional array format with row-major order which is C standard. Row-major order GEMM algorithm can be easily expanded to Column-major order GEMM algorithm by exchanging the position of matrices A and B.

All through this paper we use several terms in specific meanings. First, we named matrices based on the shape. A "Matrix" is a matrix with both dimensions are large or unknown. A "Panel" is a matrix which one of the dimensions is small. A "Block" is a matrix which both dimensions are small. Moreover, we use the term "Source matrix" to refer the read-only matrices which are multiplied, and the term "Target matrix" to refer the read-write matrix which is updated. In Figure 3.1, we describe the dimension parameters for GEMM with appropriate symbol. Further explanation of each parameter is in Table 3.1. The parameters with subscript b mean that they are cache blocking parameters, and the parameters with subscript r are for the register

Parameter	Description
m	The number of rows in the matrices A and C.
k	The number of columns in the matrix A and rows in the matrix B.
n	The number of columns in the matrices B and C.
k_b	Blocking parameter for p-loop.
m_b	Blocking parameter for i-loop.
n_r	Blocking parameter for jr-loop.
m_r	Blocking parameter for ir-loop.

Table 3.1: Description of parameters for GEMM. $C = AB + C$

blocking.

ALGORITHM 1: Naïve matrix-matrix multiplication algorithm.

```
for i = 0, ..., m do
    {read i^th row of A into cache}
    for j = 0, ..., n do
        {read C(i, j) into cache}
        {read j^th column of B into cache}
        for p = 0, ..., k do
            C(i, j) += A(i, k) × B(k, j)
        end
    end
end
```

3.1 Blocked matrix multiplication

Highly optimized GEMM implementations adapt the blocking technique which divides the matrices into submatrices to exploit the nature of hierarchical memory organization. By partitioning matrices into block matrices which fits in each level of caches, they stay in the caches and can be reused during the computation with fast cache accesses, rather than slow memory accesses. Moreover, blocking technique minimizes data transportation between memory and caches. As a result, blocking reduces the bandwidth requirements of GEMM algorithm to be under the limit of the architecture can deliver.

On the case of $C = A \times B + C$, matrix multiplication of $m \times k$, $k \times n$, and $m \times n$ matrices, this computation consists of $2mnk$ times of floating point operation and operates on $(mk + kn + mn)$ words of memory. For large m, n, k, the ratio r_{ideal} of the number of memory operations to the number of floating point operations

$$r_{ideal} = \frac{1}{2n} + \frac{1}{2m} + \frac{1}{k} \quad \text{(memops/flops)} \tag{3.1.1}$$

becomes low. Therefore, GEMM algorithm has good potential to be a compute-bound algorithm with appropriate adjustment. However, this cannot be achieved with naïve matrix multiplication algorithm.

Consider the naïve matrix multiplication algorithm in Algorithm 1. The number of memory to cache references on naïve matrix multiply, denote

m_{naive}, is as follows:

$$m_{naive} = mnk + mk + 2mn \tag{3.1.2}$$

Algorithm 1 reads each column of B through i-loop and j-loop (mnk of $memops$), reads each row of A through i-loop (mk of $memops$), reads and writes each element of C ($2mn$ of $memops$). Therefore Algorithm 1 requires memory operations more than half the number of floating point operation. This ratio is consistent under the change of loop order in Algorithm 1 because of the mnk term.

Practically, the peak computing performance of the KNL is about 3046 Gflops for double-precision operation while the memory bandwidth is over 90 GB/s with DDR4RAM, and over 450 GB/s with MCDRAM. Then, Algorithm 1 requires over 1.5×10^{12} words of memory access. Since each word is double-precision floating point number, the memory bandwidth requirement is over 12 TB/s which clearly exceeds the bandwidth limit. Therefore, blocking is an essential technique for GEMM implementation on the KNL.

Gunnels et al. [6] categorized a family of block matrix multiplication algorithms and calculated the cost of moving data between memory layers of an architecture with a multi-level memory. Goto and van de Geijn [5] analyzed the cost of data transportation between memory layers focusing on the location of the working data. By calculating the memory distance and data accessing pattern for the algorithms suggested in the paper [6], they suggested that Algorithm 2 is preferable for the implementation of GEMM on the multi-level memory system.

We implement GEMM algorithm on the KNL using Algorithm 2. Figure 3.1 presents the illustration of Algorithm 2. Algorithm 2 consists of 5-nested loops which we group p-loop and i-loop as outer loop and jr-loop, ir-loop and the inner kernel as inner loop. We put a tilde on the submatrices in outer loop of Algorithm 2 and put a hat on the submatrices in inner loop of Algorithm 2

Looking at Figure 3.1, the first p-loop divides the computation into a series of panel-panel matrix multiplication in steps of k_b. Along the p-loop the entire panel $\widetilde{B}$ is packed into the MCDRAM. The second i-loop further breaks the problem into block-panel matrix multiplication in steps of m_b.

ALGORITHM 2: Blocked Matrix-matrix multiplication algorithm.

for $p = 0, \ldots, k-1$ *in steps of* k_b **do**
  Pack $B(p : p + k_b - 1, 0 : n - 1)$ into $\widetilde{B} \in \mathbb{R}^{k_b \times n}$;
  for $i = 0, \ldots, m-1$ *in steps of* m_b **do**
    Pack $A(i : i + m_b - 1, p : p + k_b - 1)$ into $\widetilde{A} \in \mathbb{R}^{m_b \times k_b}$;
    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**
        $\widehat{A} = \widetilde{A}(ir : ir + m_r - 1, :)$; // $\widehat{A} \in \mathbb{R}^{m_r \times k_b}$
        $\widehat{B} = \widetilde{B}(:, jr : jr + n_r - 1)$; // $\widehat{B} \in \mathbb{R}^{k_b \times n_r}$
        $\widehat{C} \mathrel{+}= \widehat{A} \times \widehat{B}$; // $\widehat{C} \in \mathbb{R}^{m_r \times n_r}$
        Update C using $\widehat{C}$;
      end
    end
  end
end

Along the i-loop the entire block $\widetilde{A}$ is packed into the L2 cache. The third jr-loop reduces the computation to the multiplication of a block $\widetilde{A}$ and a slice $\widehat{B}$ of panel $\widetilde{B}$ in steps of n_r. The fourth ir-loop divides the problem into the multiplication of a slice $\widehat{A}$ of block $\widetilde{A}$ and a slice $\widehat{B}$. In the inner kernel, the submatrix $\widehat{C}$ of the matrix C locates on the vector registers and is updated in the manner of the outer product with a column of $\widehat{A}$ and a row of $\widehat{B}$.

By blocking the matrices, we can reduce significant amount of the bandwidth requirement. Consider the blocked matrix multiplication algorithm in Algorithm 2. The number of memory to cache references on blocked matrix multiply, denote m_{block}, is as follows:

$$m_{block} \approx \frac{k}{k_b} \cdot \frac{m}{m_b} \cdot (m_b k_b + k_b n + 2 m_b n) \tag{3.1.3}$$

On the Algorithm 2, execution units read each panel $\widetilde{B}$ and each block $\widetilde{A}$ through i-loop($m_b k_b + k_b n$ of *memops*), read and write each panel of C($2 m_b n$

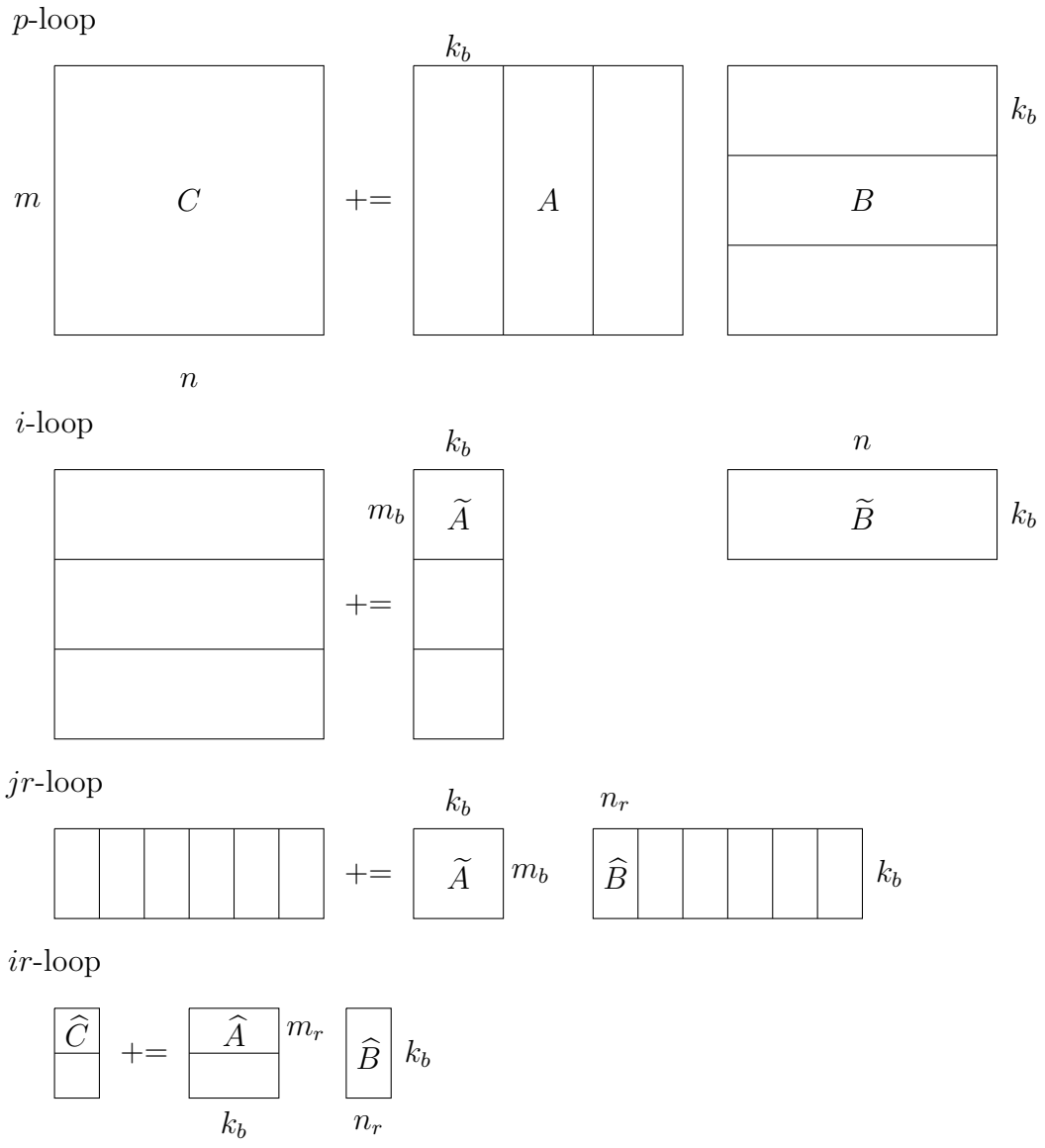

Figure 3.1: Illustration of Algorithm 2.

of *memops*). Since the size of panel $\widetilde{B}$ is too large to fit in the L2 cache, Algorithm 2 has to stream $\widetilde{B}$ from memort to cache for each iteration of i-loop. Therefore, the ratio r_{block} of the number of memory operation to the

number of floating point operation is as follows:

$$r_{block} \approx \frac{1}{2n} + \frac{1}{2m_b} + \frac{1}{k_b} \quad \text{(memops/flops)} \tag{3.1.4}$$

By choosing appropriate parameter m_b, k_b, the bandwidth requirement of Algorithm 2 can meet the limit of the KNL.

We discuss in detail about the determination of cache blocking parameters m_b, k_b in Subsection 4.1.2. And, Register blocking and determination of parameter m_r, n_r will be talked about in Section 3.2 and Subsection 4.1.1 respectively.

3.2 Inner kernel

For the GEMM implementation, the most important part is the inner kernel which handles the actual computation. To get the highest efficiency, the inner kernels of BLAS libraries are hand-coded in assembly language or similar low-level languages such as the Intel Intrinsics. In this paper, we code the inner kernel in the AVX-512 instruction set to achieve the optimal performance.

The inner kernel of the GEMM algorithm computes $\widehat{C} \mathrel{+}= \widehat{A} \times \widehat{B}$ by rank-1 update. Goto and Van de Gain [5] suggested a few general rules to build the inner kernel. And Heinecke et al [7], Peyton [14] presented more specific guidelines for the first generation of Intel Xeon Phi processor codenamed Knights Corner (KNC). In this paper, we build the inner kernel following the directions in the literature [5, 7, 14] and append a few specific guidelines for the KNL based on the performance experiments.

Algorithm 3 is the pseudocode of the inner kernel for an example of $(m_r, n_r) = (31, 8)$. By modifying variable declarations, we can easily build the inner kernel for any combination of (m_r, n_r). Recall that $\widehat{A}$, $\widehat{B}$, and $\widehat{C}$ are the submatrices of A, B, C, the each size is $m_r \times k_b, k_b \times n_r, m_r \times n_r$, respectively. Table 3.2 describes the AVX-512 instructions which are used in the inner kernel. Every instruction operates with data by the 512-bit wide vector register, named _m512d, which consists of eight double-precision floating point numbers.

In the inner kernel, vector registers hold entire $\widehat{C}$ and update $\widehat{C}$ by multiplying a column of $\widehat{A}$ and a row of $\widehat{B}$. Focus on the Algorithm 3, we load entire $\widehat{C}$ to vector registers R00 - R30, before starting the loop. Along the loop, the vector registers hold entire $\widehat{C}$ and updates each row of $\widehat{C}$ as follows:

$$\widehat{C}_i = \widehat{C}_i + \sum_{j=0}^{k_b-1} \widehat{A}_{i,j} \times \widehat{B}_j \tag{3.2.1}$$

where $\widehat{B}_j$ and $\widehat{C}_i$ are j-th row of $\widehat{B}$ and i-th row of $\widehat{C}$, respectively. For loading of $\widehat{B}$, _mm512_load_pd instruction is preferable to _mm512_loadu_pd since $\widehat{B}$ is packed contiguously in aligned space. Since the AVX-512 instruction set does not offer any kind of scalar-vector multiplication, we use _mm512_set1_pd to broadcast an element $\widehat{A}_{i,j}$ to all elements of a vector reg-

ALGORITHM 3: Inner kernel pseudocode for $(m_r, n_r) = (31, 8)$.

```
register __m512d R00,R01,···,R31;
R00 = _mm512_loadu_pd(&C[0]);
R01 = _mm512_loadu_pd(&C[8]);
    ⋮
R30 = _mm512_loadu_pd(&C[240]);
for k = 0, ..., k_b − 1 do
    _mm_prefetch(&A[DIST_A],_MM_HINT_T0);
    _mm_prefetch(&A[DIST_A+8],_MM_HINT_T0);
    _mm_prefetch(&A[DIST_A+16],_MM_HINT_T0);
    _mm_prefetch(&A[DIST_A+24],_MM_HINT_T0);
    _mm_prefetch(&B[DIST_B],_MM_HINT_T0);
    R31=_mm512_load_pd(&B[0]);
    R00=_mm512_fmadd_pd(_mm512_set1_pd(A[0]),R31,R00);
    R01=_mm512_fmadd_pd(_mm512_set1_pd(A[1]),R31,R01);
        ⋮
    R30=_mm512_fmadd_pd(_mm512_set1_pd(A[30]),R31,R30);
    A += 31;
    B += 8;
end
_mm512_storeu_pd(&C[0],R00);
_mm512_storeu_pd(&C[8],R01);
    ⋮
_mm512_storeu_pd(&C[240],R30);
```

ister. We employ FMA instruction rather than separate multiply and add instructions to update each row of $\widehat{C}$. At the end of the k-loop, the target matrix C is updated by $\widehat{C}$ using _mm512_storeu_pd instruction.

Prefetching data is an important technique for achieving high-performance [10]. Prefetching is mainly used to hide the memory latency for an algorithm. By fetching the requiring data before execution, we can minimize the amount of cache misses. The KNL provides both hardware and software prefetching.

Instruction	Description
_mm512_loadu_pd	load instruction for $\widehat{C}$
_mm512_storeu_pd	store instruction for $\widehat{C}$
_mm512_load_pd	load instruction for $\widehat{B}$
_mm512_set1_pd	broadcast instruction for $\widehat{A}$
_mm512_fmadd_pd	FMA instruction for $\widehat{C}$
_mm_prefetch	prefetch instruction for $\widehat{A}$, $\widehat{B}$

Table 3.2: Description of instructions for inner kernel

For hardware prefetching, each core has L1 hardware prefetcher and each tile has L2 hardware prefetcher unlike the KNC which has only L2 hardware prefetcher. Hardware prefetchers can be switched off and on.

Also, the KNL supports AVX-512 instructions for software prefetching, named AVX-512PF. Each prefetch instruction fetches one 64 bytes wide cache line which contains 8 double- or 16 single- precision floating point numbers. Hence, we need to call prefetching instruction for (size of data)/8 times for double-precision elements.

In Algorithm 2, $\widehat{A}$ and $\widehat{B}$ are streamed from the L2 cache to the L1 cache. Prior to their use in the inner kernel, we put them into the L1 cache by software prefetching. We insert L1 prefetch instructions for the next elements of $\widehat{A}$ and $\widehat{B}$ within the k-loop in Algorithm 3 .

According to Jeffers et al. [8], the hardware prefetchers for the KNL is more effective than that for the KNC. So, they recommended applying software prefetching less aggressively on the KNL. For the GEMM implementation, we observe that software prefetching is essential to achieve good performance. Without software prefetching, the performance decreases under 70%.

Loop unrolling also can increase the performance by reducing instructions that control the loop. Also, this technique help to operate on data using full vectors by duplicating the loop body. Which means that loop unrolling can enable vectorization and help to exploit the vector registers. According to Jeffers et al. [8], the Intel compiler can typically generate efficient vectorized code if a loop structure is not manually unrolled and it is recommended to

use #pragma unroll(N) to control unrolling. If N is specified, the optimizer unrolls the loop N times. If N is omitted, or it is outside the allowed range, the optimizer picks the number of times to unroll the loop [8], We unroll the i-loop in Algorithm 3 using #pragma unroll(N).

3.3 Packing algorithm

The fundamental problem for implementing blocked GEMM algorithm is that the submatrices $\widehat{A}$, $\widehat{B}$, $\widetilde{A}$ and $\widetilde{B}$ of large matrices A and B are not contiguously stored in memory. These non-contiguous accesses to large matrices A and B lead to two major problems.

First one is the Translation Lookaside Buffer (TLB) issues. The TLB is a memory cache which contains memory page table entries and segment table entries. By storing the recent translation of virtual memory to physical memory in the TLB, a machine can reduce the time to access memory. Without packing, the memory access pattern becomes scattered which cause the TLB misses. These TLB misses lead to the CPU stalls until the TLB has been updated. Hence, it is very expensive in terms of performance.

The second one is the spatial locality problem. On the KNL, the basic unit of memory access is 64 bytes-wide cache line. If the accessing location is scattered, only small portion of cache line holds the data of the source matrices. Moreover, the dispersed accessing pattern may hinder the spatial prediction by caches.

To solve these problems, we need to pack the sub-matrices $\widetilde{A}$, and $\widetilde{B}$ in temporary space which fits in the TLB to avoid the TLB misses. In addition, we rearrange the storing order of each submatrices to be able to access contiguously to exploit spatial locality from caches. Figure 3.2 and Figure 3.3 provide the illustrations of packed submatrices $\widetilde{A}$ and $\widetilde{B}$, respectively.

In our GEMM implementation, see Algorithm 2, we pack a panel of matrix B into auxiliary space $\widetilde{B}$ along the p-loop and pack a block of matrix A into auxiliary space $\widetilde{A}$ along the i-loop. Note that, $\widetilde{B}$ fits on the memory (MCDRAM) and $\widetilde{A}$ fits on the L2 cache. Along the k-loop in Algorithm 3, the inner kernel accesses a column of $\widehat{A}$ and a row of $\widehat{B}$. Therefore, we pack $\widehat{A}$ in column major order and $\widehat{B}$ in row major order. Along the ir-loop in Algorithm 2, $\widehat{A}$ in $\widetilde{A}$ is accessed in a row-wise manner. Thus $\widetilde{A}$ consists of $\widehat{A}$ in row major order with $\widehat{A}$ as a unit. Along the jr-loop in Algorithm 2, $\widehat{B}$ in $\widetilde{B}$ is accessed in a column-wise manner. Hence, $\widetilde{B}$ consists of $\widehat{B}$ in column major order with $\widehat{A}$ as a unit.

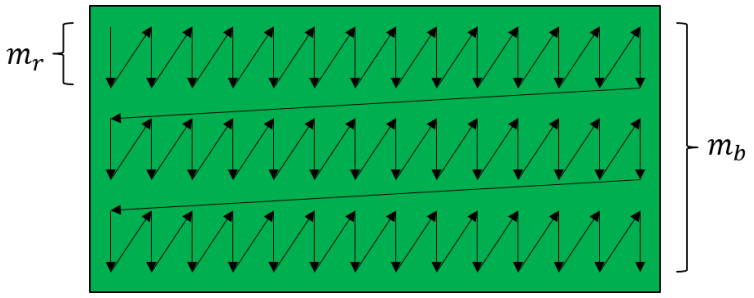

Figure 3.2: Storing order of packed block $\widetilde{A}$

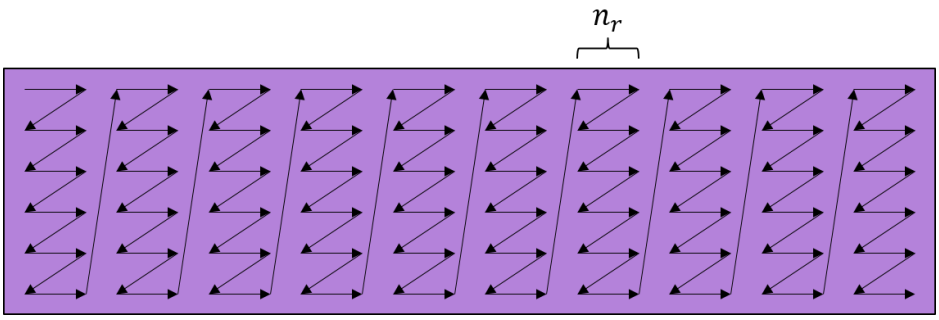

Figure 3.3: Storing order of packed panel $\widetilde{B}$

3.4 Parallelization

3.4.1 Parallelization of blocked matrix multiplication algorithm

Our GEMM algorithm has rich potential for parallelization. In Algorithm 2, the race condition occurs only on the p-loop which is the outermost loop. That is, we can parallelize the i-loop, jr-loop, and ir-loop. We analyze the potential of each loop in Algorithm 2.

First, consider the second i-loop in Algorithm 2. When we parallelize the i-loop, each thread will be assigned a different block $\widetilde{A}$ which stays in L2 cache and every thread will share the same panel $\widetilde{B}$. Afterwards, each thread will compute multiplication of their own block $\widetilde{A}$ and shared panel $\widetilde{B}$. Since L2 cache is shared by both cores in the same tile, we take two options about the size of $\widetilde{A}$. In a conservative manner, the size of $\widetilde{A}$ needs to be less than the half of L2 cache. On the other hand, we can take the full L2 cache to be a limit because of the sharing nature of L2 cache.

The iteration number of i-loop is $\frac{m}{m_b}$. The number is not limited by the blocking parameters, but highly depends on the size of m. Therefore, we can fully utilize the parallel potential of i-loop with sufficiently large m. On the other hand, if m is less the multiplication of the blocking parameter m_b and the degree of parallelization of the loop, we need to reduce the size of $\widetilde{A}$ or the degree of parallelization. Both of them may make a loss on the performance.

Next, consider the third jr-loop in Algorithm 2. When we parallelize the jr-loop, each thread will take a different slice $\widehat{B}$ of panel $\widetilde{B}$ which stays in individual L1 cache and every thread will share the same block $\widetilde{A}$. Subsequently, each thread will compute multiplication of their own slice $\widehat{B}$ and shared block $\widetilde{A}$.

The iteration number of jr-loop is $\frac{n}{n_r}$. Because the size of blocking parameter n_r is small, at most 32, the number of iterations is large enough to feed threads. Therefore, we are free on choosing the degree of parallelization of the jr-loop.

Now, consider the fourth ir-loop in Algorithm 2. When we parallelize the ir-loop, each thread will take a different slice $\widehat{A}$ of block $\widetilde{A}$ which is streamed

in individual L1 cache and every thread will share the same slice $\widehat{B}$. Then, each thread will compute multiplication of their own slice $\widehat{A}$ and shared slice $\widehat{B}$.

The iteration number of ir-loop is $\frac{m_b}{m_r}$. Note that this ir-loop has a fixed number of iteration regardless of the size of the target matrices A and B. Moreover, the amount of computation in each iteration of this loop is too small to amortize the cost of bringing $\widetilde{A}$ and $\widetilde{B}$ into L1 cache. Thus, the parallelization potential of the ir-loop is quite limited. Therefore, we do not parallelize the ir-loop on our GEMM implementation.

3.4.2 Parallelization of packing kernel

The packing kernel also has good parallelization potential. The race condition does not occur because each thread will access seperate position of source matrices and will store to different address of auxiliary spaces. Hence, we can parallelize both A and B packing kernels. We analyze the potential of each packing kernel in Algorithm 2.

First, consider the B packing kernel in the first p-loop in Algorithm 2. When we parallelize B packing kernel, the kernel will create an individual parallel region in the p-loop. Since we do not parallelize the p-loop because of the race condition, there is no pre-generated threads. Each thread will load a contiguous n_r number of elements of B and will store in temporary aligned space $\widetilde{B}$ in a certain order.

The iteration number of packing kernel is $\frac{n}{n_r}$. The number is not limited by the blocking parameters. Moreover, the number of iterations is big enough to take large degree of parallelization because of small size of the blocking parameter n_r. Hence, we can freely choose the degree of parallelization of B packing kernel. Notice that this individual parallel region may become a burden on thread generation, it requires attention to choose optimal degree of parallelization.

Next, consider the A packing kernel in the second i-loop in Algorithm 2. Like B packing kernel, this parallelization will create an individual parallel region in the i-loop. Since the i-loop is one of our parallelization target, we need to consider the pre-generated threads. Each thread will load a uniformly

scattered m_r number of elements of A and will store in auxiliary aligned space $\widetilde{A}$ in a contiguous order.

The iteration number of packing kernel is about $\frac{m_b}{m_r}$. Note that A packing kernel has a fixed number of iteration regardless of the size of the target matrix A. Moreover, we need to generate threads additionally for this parallel region. Since A packing kernel already locates in a parallelized loop, extra generation will make a loss on the performance. Thus, the parallelization potential of the A packing kernel is quite limited. Therefore, we do not parallelize the A packing kernel on our GEMM implementation.

3.4.3 Environment setting

Now, consider the environment variables of the KNL which are related to the thread affinity. Since we parallelize i-loop and jr-loop of Algorithm 2 which are nested, the thread binding is crucial to achieve good performance. Moreover, the tile-based structure of the KNL makes the environment setting more considerable.

On the KNL, following two environment variables bind threads to physical processing units.

- $ KMP_AFFINITY = SCATTER
- $ OMP_PLACES = CORES

KMP_AFFINITY variable has priority to OMP_PLACES variable. If two variables are declared concurrently, the machine binds threads according to the KMP_AFFINITY setting.

The Intel OpenMP runtime library provides KMP_AFFINITY environment variable to control thread placement. KMP_AFFINITY = SCATTER is a popular setting which scatters across cores before using multiple threads on a given core [8]. When we set OMP_NUM_THREADS = 68, the machine bind one thread to each core on the KNL under KMP_AFFINITY = SCATTER setting. OpenMP Affinity provides OMP_PLACES environment variable to set the thread placement. OMP_PLACES binds threads to physical processing units with general syntax location number stride. CORES option is a predefined value which set threads core by core. Which means is that

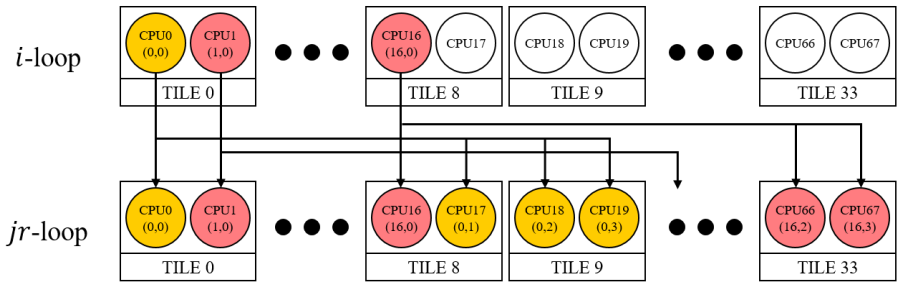

Figure 3.4: Task distribution in nested loop when KMP_AFFINITY = SCATTER. We indicate (0,0) task to (0,3) task in yellow

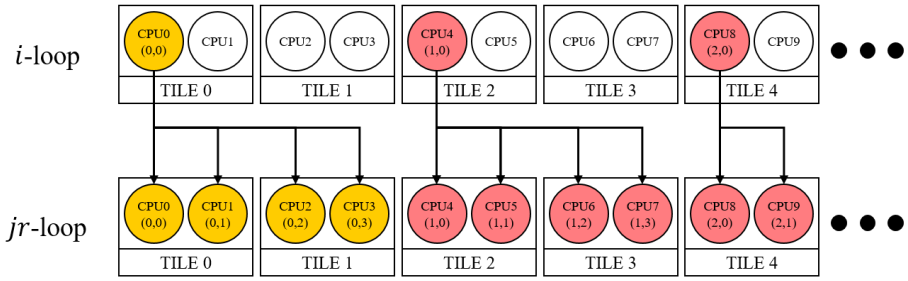

Figure 3.5: Task distribution in nested loop when OMP_PLACES = CORES. We indicate (0,0) task to (0,3) task in yellow

OMP_PLACES = CORES setting has same CPU map as KMP_AFFINITY = SCATTER. However, these two options differ for task distribution of nested loop.

Figure 3.4 shows the task distribution on the nesting OpenMP parallel region under the option KMP_AFFINITY = SCATTER. (m, n) denotes the task of mth iteration on i-loop and nth iteration on jr-loop. Firstly, 17 number of threads take the (n,0) job which are the fore part of inner loop. And other threads divide up the remains. Without losing of generality, we may assume that CPU0 takes (0,0) job and CPU17, CPU18, CPU19 take (0,1), (0,2), (0,3) respectively. Since CPU18, CPU19 locate in the same tile, they can share same block $\widetilde{A}$. However, CPU0 and CPU17 are in different tiles, sharing $\widetilde{A}$ is impossible. The number of tiles which requires different $\widetilde{A}$ to compute is $2 \times$ degree of parallelization on the i-loop. This makes hard to

exploit the sharing nature of L2 cache and leads the severe loss on performance.

On the other hand, OMP_PLACES = CORES setting provides different task distribution. Figure 3.5 shows the task distribution on the same nesting OpenMP parallel region under the option OMP_PLACES = CORES. The tasks are divided into four sequential units. That is, four number of threads bound nearby two tiles share same block $\widetilde{A}$. In this case, both cores in a same tile share same $\widetilde{A}$ which help to utilize the sharing nature of L2 cache. But, we need to be careful when the degree of parallelization on the jr-loop is odd. This makes at least four tiles to require different $\widetilde{A}$ to execute.

Consequently, we surmise that OMP_PLACES = CORES setting is more suitable for GEMM implementation. In Subsection 4.2.2, we discuss in detail with the performance results.

Chapter 4

Experiments

In this chapter, we deal with practical implementation of our GEMM algorithm on the KNL. We focus on the double-precision case. Our implementation can be easily expanded to single-precision case with same restrictions and environment setting. We conduct several experiments and present the optimization process based on the performance results.

Prior to the implementation, we need to check the optimal number of threads per core. Each core on the KNL generates up to 4 threads by hyper-threading. While the KNC requires more than one thread running per core to reach maximal performance, the KNL can reach maximum performance with one, two, or four threads per core [8]. The optimal number of threads per core highly depends on the individual routines. Therefore, we have tested the performance of cblas_dgemm kernel of the Intel MKL differing the number of threads.

We differ the number of threads from 1 to 4 by setting the environment variables as follows:

- $ export KMP_AFFINITY = compact
- $ export KMP_HW_SUBSET = '1C,nT' with n=1, 2, 3, 4

Figure 4.1 shows the performance results. The double-precision GEMM (DGEMM) kernel achieve the best performance when the number of thread is one and the worst performance when the number of thread is four. This result is consistent with the statement of Jeffers et al. [8] that the KNL can achieve good

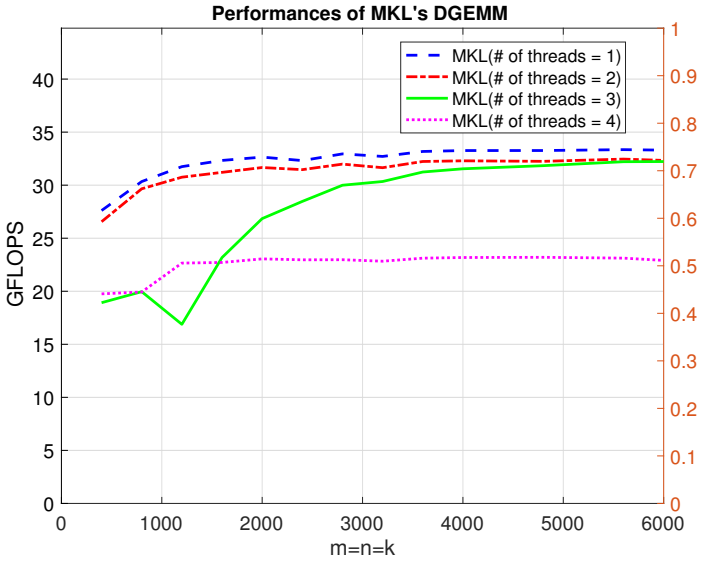

Figure 4.1: Performances of MKL's DGEMM with # of threads = 1, 2, 3, and 4.

performance with one thread per core. The DGEMM routine of the Intel MKL reaches 75 percent of the theoretical peak performance, 44.8 GFLOPS per core.

Based on this result, we implement our DGEMM algorithm with one thread per core.

4.1 Sequential double-precision GEMM

The sequential implementation of GEMM algorithm is the base of parallel implementation. In this section, we determine the blocking parameters m_r, n_r, m_b, k_b for register and cache blocking, and suggest a few guidelines to build BLAS routines on the KNL.

4.1.1 Register blocking

First, we determine the register blocking parameters m_r, n_r of the inner kernel of Algorithm 2. We use the term register block sizes to call the parameters m_r, n_r since their sizes are restricted by the number of vector registers.

Goto and van de Geijn [5] suggested to use half of the available registers for the storing $m_r \times n_r$ elements of $\widehat{C}$. This is because of the space for prefetching data of $\widetilde{A}$ and $\widetilde{B}$. Moreover, they claimed to take the parameters to be $m_r \approx n_r$ to amortize the cost of loading the registers optimally. This gives

$$\left\lceil \frac{m_r \times n_r}{\text{size of register}} \right\rceil \leq \frac{\text{number of register}}{2} \quad \text{with} \quad m_r \approx n_r \tag{4.1.1}$$

Heinecke et al. [7] picked $(m_r, n_r) = (30, 8)$ for register blocking on the KNC and Peyton [14] used $(m_r, n_r) = (24, 8)$ while each core on the KNL has 32 vector registers. Since each vector register is 512 bytes-wide, Heinecke et al. [7] used 31 vector registers and Peyton [14] used 25 vector registers for register blocking. Which means that both used more than half of available vector registers on the KNC.

According to Jeffers et al. [8], high performance can be achieved with the use of full vector registers. For choosing n_r, n_r should be a multiple of 8 because AVX-512 instructions offer 512-bit vector operations(8 double elements at once). The inner kernel uses $m_r \times \frac{n_r}{8}$ vector registers to store the multiplication result of $\widehat{A} \times \widehat{B}$ and dedicates $\frac{n_r}{8}$ vector registers to hold a row of $\widehat{B}$ for each rank-1 update. Like KNC, each core on the KNL has 32 vector registers. Thus, we can have the followings:

$$\frac{n_r}{8}(1 + m_r) \leq 32 \quad \text{and} \quad n_r \equiv 0 \pmod 8 \tag{4.1.2}$$

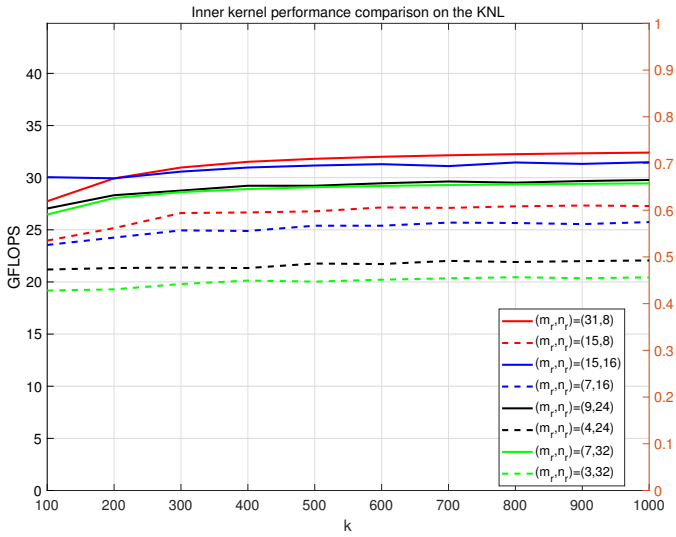

Figure 4.2: Performances of inner kernels at (4.1.3) and (4.1.4)

for register blocking on the KNL.

We consider the following pairs of register block sizes (m_r, n_r):

$$(m_r, n_r) = (31, 8),\ (15, 16),\ (9, 24),\ \text{or}\ (7, 32) \tag{4.1.3}$$

for making full use of available vector registers. Also we conduct performance test for half use of available vector registers.

$$(m_r, n_r) = (15, 8),\ (7, 16),\ (4, 24),\ \text{or}\ (3, 32) \tag{4.1.4}$$

We use the term full-use combination for cases of (4.1.3) and the term half-use combination for cases of (4.1.4).

Figure 4.2 presents the performance results of the inner kernel test with each pair of (m_r, n_r) in (4.1.3) and (4.1.4). On the experiment, the inner kernel computes $\widehat{C}$ += $\widehat{A} \times \widehat{B}$ under the assumption that $\widehat{A}$ and $\widehat{B}$ are already packed. We use solid lines to indicate full-use cases and dashed lines to indicate half-use cases. The performance of full-use combinations, (31, 8), (15, 16), (9, 24) and (7, 32), overcomes the result of every half-use combination. On the KNL, this result clearly supports the statement of Jeffers et al. [8] that

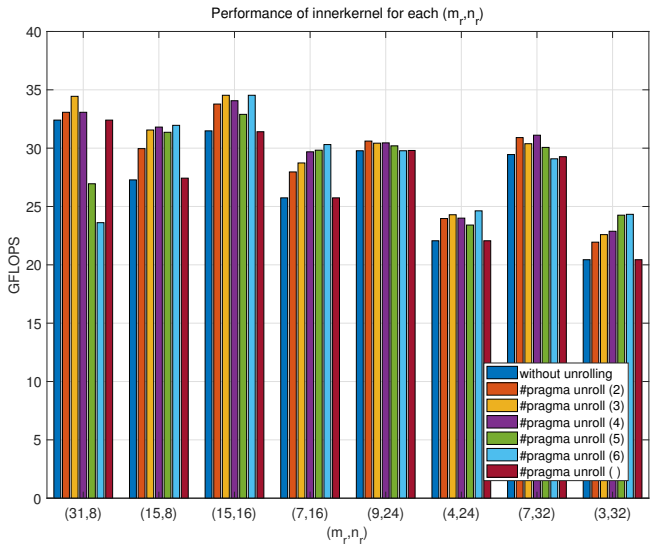

Figure 4.3: Performances of inner kernels with unrolling

we can achieve high performance with the use of full vector register. Unlike the cases of Goto and van de Geijn [5], the KNL does not store prefetched element on the vector registers. Therefore, the inner kernel needs to use the vector registers as many as possible to exploit the VPUs.

To increase the vector register usage of inner kernels, we consider applying loop unrolling technique to each inner kernel. Loop unrolling helps to fully exploit the vector registers by duplicating the loop body of inner kernels. We use #pragma unroll(N) option rather than manual unrolling as recommended.

Figure 4.3 provides the performance results of the inner kernel test with various unrolling factors. In all cases, unrolling of the k-loop enhances the performance of inner kernels. Note that half-use combinations require more unrolling than full-use combinations to get the optimal performance. Moreover, the effect of loop unrolling is more dramatic on the half-use cases. However, even with the performance enhancement by loop unrolling technique, the inner kernel achieves better result on the full-use cases.

Consequently, we focus on two best combinations:

$$(m_r, n_r) = (31, 8),\ (15, 16) \tag{4.1.5}$$

for the determination of the parameters on cache blocking.

4.1.2 Cache blocking

Highly optimized GEMM implementations use the block matrices which fit in multi-levels of cache on a given architecture. As we discussed at Section 3.1, blocking is essential for bandwidth requirement reduction and better cache reuse. The size of block matrices have been discussed in the literature [5, 6, 12, 14, 17, 18]. We first overview the strategies for block size determination introduced in the papers [5, 6, 7, 14].

Gunnels et al. [6] classified matrix-matrix multiplication into matrix-panel, panel-matrix, panel-panel multiplication. This classification is based on the shape of the matrices. They suggested the followings:

- In each matrix multiplication case, the largest matrix of the computation should take almost every space of cache .
- To amortize the cost of data movement between cache layers, the largest matrix need to be square-like shape.

Goto and van de Geijn [5] pointed out that the square blocking method suggested in the paper [6] is unrealistic for two reasons. One is lack of consideration about the inner kernel part, and the other is Translation Look-aside Buffer(TLB) issues. Goto and van de Geijn rather suggested the followings:

- k_b should be chosen as large as possible to amortize the cost of updating $m_r \times n_r$ elements of $\widehat{C}$.
- $\widehat{B}$ are reused at least m_b/m_r times, and must remain in the L1 cache. In practice, $\widehat{B}$ should occupy less than half of the L1 cache.
- m_b is typically chosen so that $\widetilde{A}$ occupies about half of smaller of the memory addressable by the TLB and the L2 cache.

This gives

$$k_b \times m_b \times 8 \text{ bytes} \leq \frac{\text{size of L2}}{2} \tag{4.1.6}$$

and

$$n_r \times k_b \times 8 \text{ bytes} \leq \frac{\text{size of L1}}{2}. \tag{4.1.7}$$

Heinecke et al. [7] designed a benchmark on the KNC including DGEMM routine. They used the blocking strategy on the L2 cache and claimed that it is sufficient to fit three submatrices, $\widetilde{A}$, $\widehat{B}$, and $\widehat{C}$ of dimensions $m_b \times k_b$, $k_b \times n_r$, and $m_r \times n_r$, respectively, into the L2 cache. This gives

$$(m_b k_b + k_b n_r + m_r n_r) \times 8 \text{ bytes} \leq \text{size of L2}. \tag{4.1.8}$$

Peyton [14] implemented a GEMM routine on the KNC. He claimed that the restrictions on blocking parameters, m_r, m_b, k_b, and n_b suggested by Goto and van de Geijn [5] may hinder the performance of the GEMM routine for the KNC. Since the multiple threads running on a core of the KNC share the same L1 and L2 cache, the number of threads per core should be considered for the implementation. Rather, Peyton suggested the restriction for the KNC as follows:

$$(m_b k_b + k_b n_r) \times 8 \text{ bytes} \times \#\text{ of threads} \leq \frac{\text{size of L2}}{2}. \tag{4.1.9}$$

We now discuss cache block sizes for the KNL. As we described at the start of this chapter, we assume that using one thread per core is enough to reach the maximum performance. First, we require $\widetilde{A}$, $\widehat{B}$, and $\widehat{C}$ to fit in the L2 cache. Since two cores on a tile share 1 MB L2 cache, we can obtain the following inequality:

$$(m_b k_b + k_b n_r + m_r n_r) \times 8 \text{ bytes} \times 2 \leq 1 \text{ MB}. \tag{4.1.10}$$

in a conservative manner. On the other hand, considering the sharing nature of L2 cache, we may assume that $\widetilde{A}$ is shared by both cores in the same tile. Thus, we can modify the inequality (4.1.10) as follows:

$$(m_b k_b + 2 \times (k_b n_r + m_r n_r)) \times 8 \text{ bytes} \leq 1 \text{ MB}. \tag{4.1.11}$$

Next, we consider the L1 cache blocking. On the KNL, the efficiency of L1 cache blocking is controversial. Goto and Ven da Gain [5] implied that the blocking for every cache level is essential to achieve optimal performance of the target machine. However, Heinecke et al. [7], Peyton [14], and Xianyi et al. [19] suggested that blocking for L2 cache is enough to get optimal performance. These researches on the KNC [7, 14] and the Loongsan processor [19] pointed out that the size of L1 cache is too small to take advantage of blocking. With small size L1 cache, the parameter k_b becomes too limited to be able to achieve the optimal performance of the inner kernel.

Since there does not exist any explicit method for cache control on the AVX-512 instruction set, the data flow depends on the caching algorithm. The cache replacement policy of the KNL is Pseudo-Least Recently Used(PLRU) which is a scheme that replaces one of the least recently used items based on approximate measures of age. Along the computation in the inner kernel, L1 cache streams the entire $\widehat{A}$, $\widehat{B}$, and $\widehat{C}$ to the vector registers. Since the cache replacement policy is PLRU, we need to consider not only the size of $\widehat{B}$ but also the size of $\widehat{A}$ and $\widehat{C}$ to hold $\widehat{B}$ in L1 cache through the ir-loop. Hence, we need $\widehat{A}$, $\widehat{B}$, and $\widehat{C}$ to dedicate in the L1 cache for L1 cache blocking. Since each core has 32 KB L1 cache, the parameters m_r, n_r, k_b satisfy the following inequality:

$$(m_r k_b + k_b n_r + m_r n_r) \times 8 \text{ bytes} \leq 32 \text{ KB}. \tag{4.1.12}$$

To meet the inequality (4.1.12), the parameter k_b is up to 124 with $(m_r, n_r) = (15, 16)$, and up to 98 with $(m_r, n_r) = (31, 8)$.

Since there are a lot of k_b and m_b satisfying (4.1.11) for each (m_r, n_r), we conduct performance experiments to choose k_b and m_b. To check the impact of L1 cache blocking, we vary the parameters k_b, m_b including both blocking and non-blocking cases while m, n, and k remain fixed at 2,400 in the performance experiments.

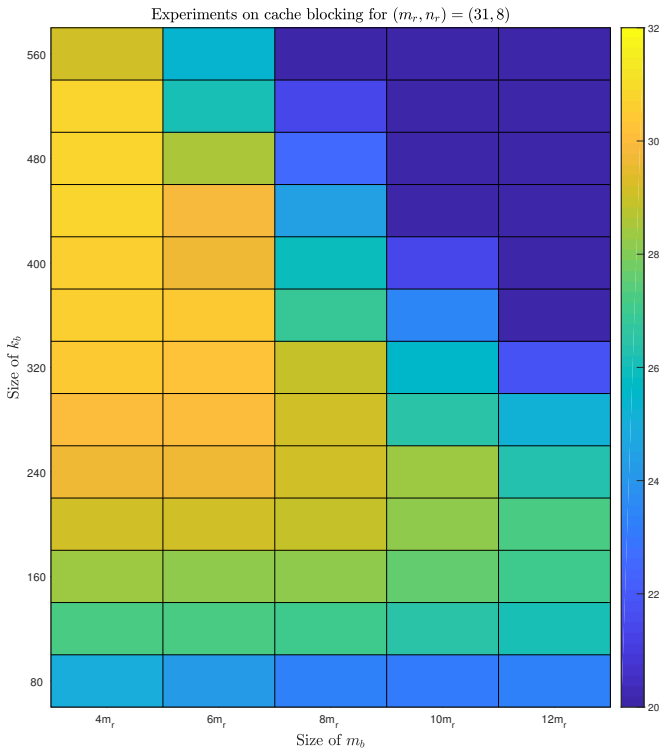

Figure 4.4: Performances of inner kernel with (m_r, n_r)=(31, 8)

Figure 4.4 and Figure 4.5 shows the performance results of inner kernel for (m_r, n_r)=(31, 8) and (m_r, n_r)=(15, 16) when $m = n = k = 2,400$. We vary the parameter k_b from 80 to 600 in steps of 40, m_b from 120 to 434 in steps of multiple of m_r. The optimal value of k_b for each m_b is provided in Table 4.1 and Table 4.2. Note that the inner kernel achieves better performance when $(m_r, n_r) = (31, 8)$. The best performance of our implementation is obtained when $(m_r, n_r, m_b, k_b) = (31, 8, 124, 520)$.

First, we can observe that the inner kernels obtain good performance results when the size of $(\widetilde{A} + \widehat{B} + \widehat{C})$ slightly exceeds the half of shared 1MB L2 cache. Since we use only a single core in a tile, it does not break the L2 cache blocking. The optimal L2 cache usage is consistent regardless of choice of the inner kernel or the size of parameter m_b.

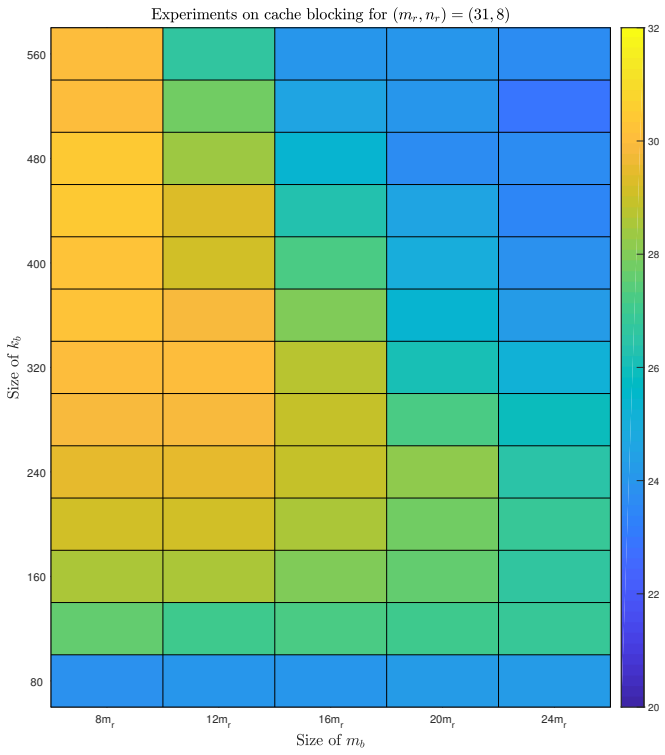

Figure 4.5: Performances of inner kernel with (m_r, n_r)=(15, 16)

Further, L1 cache blocking is not crucial for the implementation of DGEMM on the KNL. For example, the performance of DGEMM with $m_r = 15$, $n_r = 16$, $m_b = 120$, and $k_b = 480$ is better than that of DGEMM with $m_r = 15$, $n_r = 16$, $m_b = 300$, and $k_b = 240$, even though the size of $\widehat{B}$ with $n_r = 16$ and $k_b = 480$ exceeds the size of the L1 cache on the KNL while the size of $\widehat{B}$ with $n_r = 16$ and $k_b = 240$ fits into the L1 cache. These results show that fitting $\widehat{B}$ into the L1 cache is not essential to the performance of GEMM routine.

In addition, the inner kernels achieve better performance with smaller m_b. The size of m_b determines the number of data-reusing of $\widehat{B}$. If this data-reusing is essential to achieve better performance, larger m_b helps to get better performance with a fixed k_b. However, performance results show that

m_b	124	186	248	310	372	434
k_b	520	360	280	240	200	160
Size of $(\widetilde{A}+\widehat{B}+\widehat{C})$	538 KB	548 KB	562 KB	598 KB	596 KB	554 KB
Performance	30.86	30.39	29.17	28.43	27.30	26.22

Table 4.1: Optimal k_b for each m_b with (m_r, n_r)=(31, 8)

m_b	120	180	240	300	360	420
k_b	480	320	280	240	200	160
Size of $(\widetilde{A}+\widehat{B}+\widehat{C})$	512 KB	492 KB	562 KB	594 KB	589 KB	546 KB
Performance	30.38	29.98	28.97	28.24	26.84	26.60

Table 4.2: Optimal k_b for each m_b with (m_r, n_r)=(15, 16)

the performance decreases as the parameter m_b increases, even with small k_b which satisfies the inequality (4.1.12). This indicates that the data-reusing of $\widehat{B}$ in L1 cache is not crucial for the performance of inner kernel.

Moreover, small k_b may hinder the performance of GEMM routine crucially. When k_b is chosen to satisfy the inequality (4.1.12), k_b may not be large enough to amortize the cost of updating submatrix $\widehat{C}$ on the KNL. The amount of memops to update $m_r \times n_r$ elements of C is fixed to $m_r n_r$ regardless of k_b while the amount of flops is $2m_r n_r k_b$ which is proportional to k_b. Thus, larger k_b helps to amortize the cost of updating $m_r \times n_r$ elements of C.

We conduct further performance experiments to choose optimal parameters m_b and k_b for $(m_r, n_r) = (31, 8)$. We varied m_b from 31 to 124 and k_b from 480 to 1,920 while m, n and k remain fixed at 2,400 and the cache and register block sizes satisfy (4.1.11). Since preceding experiments implies that L1 cache blocking does not take an important role in the performance of inner kernel, we do not consider fitting $\widehat{B}$ in L1 cache. We also determine the optimal prefetch distances for $\widehat{A}$ and $\widehat{B}$ for each (k_b, m_b).

We obtain the best performance when $(m_b, k_b) = (31, 1620)$. Figure 4.6 shows the performance results of prefetching distance determination for $(m_b, k_b) = (31, 1620)$. We achieve the best performance when the prefetch distance for

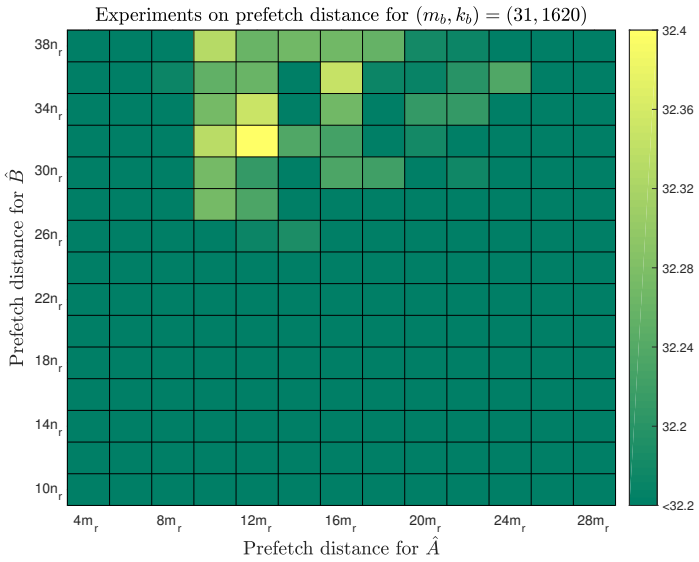

Figure 4.6: DGEMM experiments on prefetch distances for $(m_b, k_b) = (31, 1620)$, when $m = n = k = 2400$.

$\widehat{A}$ and $\widehat{B}$ are chosen to be $12m_r$ and $32n_r$, respectively.

Figure 4.7 presents the performance comparison between DGEMM kernel of the Intel MKL and our DGEMM implementation the different matrix sizes on the KNL. We measure the performance of DGEMM routine varying the size of the matrices $m = n = k$ from 400 to 9,600. Every point on the figure represents an average of 10 runs for the given conditions. Our implementation with $(m_r, n_r, m_b, k_b) = (31, 8, 31, 1620)$ achieves up to 99 percent of DGEMM performance using the Intel MKL when the size of $m = n = k$ is larger than 2,400.

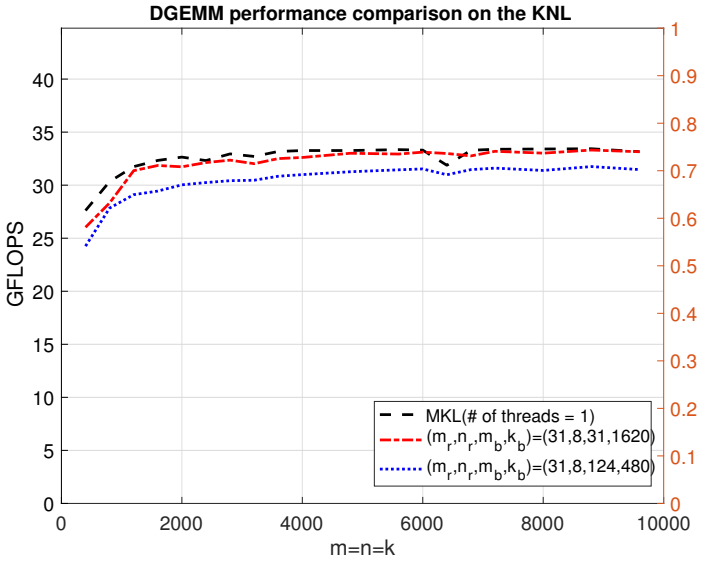

Figure 4.7: DGEMM performance comparison for different matrix sizes on the KNL.

4.2 Parallel double-precision GEMM

In this section, we parallelize the sequential implementation of double-precision GEMM on the KNL. First, we calculate the bandwidth requirement of our DGEMM implementation and adjust the cache blocking parameters m_b, k_b based on this calculation. Afterwards, we conduct several experiments to determine the degree of parallelization for i-loop and jr-loop of the Algorithm 2. We also present the performance results related to the environment variable issues.

4.2.1 Bandwidth requirement

Before parallelize the sequential implementation, we need to consider the bandwidth requirement of our DGEMM algorithm. As we described at Section 3.1, the ratio r_{block} of the number of memory operations to the number

of floating point operations is as follows:

$$r_{block} \approx \frac{1}{2n} + \frac{1}{2m_b} + \frac{1}{k_b} \quad \text{(memops/flops)} \tag{4.2.1}$$

The inner kernel uses the FMA instruction which computes 8 additions and 8 multiplications per cycle and there are two VPUs in each core. Hence, a core on the KNL carries out 32 floating point operation per cycle. Therefore, we can convert the term memops/flops to bytes/cycles as follows:

$$\frac{\text{memops}}{\text{flops}} = 8 \times 32 \times \frac{\text{bytes}}{\text{cycles}}. \tag{4.2.2}$$

Thus, the bandwidth requirement is given by

$$128 \left(\frac{2}{k_b} + \frac{1}{n} + \frac{1}{m_b} \right) \text{ bytes/cycles}. \tag{4.2.3}$$

$\frac{1}{n}$ in (4.2.3) is negligible for the large value n.

The bandwidth requirement per core is 6.02 GB/sec (1.4 GHz $\times$ 4.3 bytes/cycles) when $m_r = 31$, $n_r = 8$, $m_b = 31$, and $k_b = 1620$. Since DDR4 SDRAM meets the bandwidth requirement for sequential implementation, we do not need to use the MCDRAM for a single core on the KNL. On the other hand, if we run our DGEMM implementation on the whole cores of the KNL with above blocking parameters, the required memory bandwidth will be 409.36 GB/sec which clearly exceeds the capability of DDR4 SDRAM. Thus, we have to use the MCDRAM to meet the bandwidth requirement for parallel DGEMM implementation on the KNL. According to Jeffers et al. [8], memory latency increases with memory bandwidth in general and the latency rises rapidly near the bandwidth limitation. Therefore, we need to adjust the blocking parameter m_b which takes most part of bandwidth requirement to reduce memory latency. Also, we observe a serious loss on performance when the parameter m_b is bigger than 155. Hence, we choose the parameter m_b to be 124 and conduct experiments to find optimal parameter k_b. When we implement parallel DGEMM on the KNL, we put the submatrices $\widetilde{A}$, $\widetilde{B}$, and $\widehat{C}$ in the MCDRAM.

4.2.2 Degree of parallelization

Now, we determine the degree of parallelization for each parallel region. As we described at Section 3.4, we parallelize i-loop and jr-loop in Algorithm 2 and B packing kernel in p-loop. The KNL has 68 cores, we can have four different parallelization schemes on the KNL as follows:

- the jr-loop is parallelized with 68 threads.
- the i-loop is parallelized with 2 threads and the jr-loop is parallelized with 34 threads.
- the i-loop is parallelized with 4 threads and the jr-loop is parallelized with 17 threads.
- the i-loop is parallelized with 17 threads and the jr-loop is parallelized with 4 threads.

We do not consider the case that the i-loop is parallelized with 68 threads and the case that the i-loop is parallelized with 34 threads and jr-loop is parallelized with 2 threads because of limited parallelization potential. The tasks are evenly distributed across all cores only when m is a multiple of $m_b \times 68$ or $m_b \times 34$.

Also, we observe that the environment variable settings play an important role in the parallelization. Hence, we conduct several experiments to test the effect of environment variables for our DGEMM algorithm. We run our DGEMM program with the following conditions for each parallelization cases.

- KMP_AFFINITY = scatter
- OMP_PLACES = cores

To find optimal value of the blocking parameter k_b, we vary k_b from 80 to 560 in steps of 16 while fix the matrix size to m=n=20000, k=5000. The prefetch distance for $\widehat{A}$ and $\widehat{B}$ are chosen to be $18m_r$ and $20n_r$, respectively.

Figure 4.8 compares the performance results of different parallelization schemes on the KNL. There are four different parallelization schemes with two environment conditions presented. We use solid lines to indicate OMP_PLACES

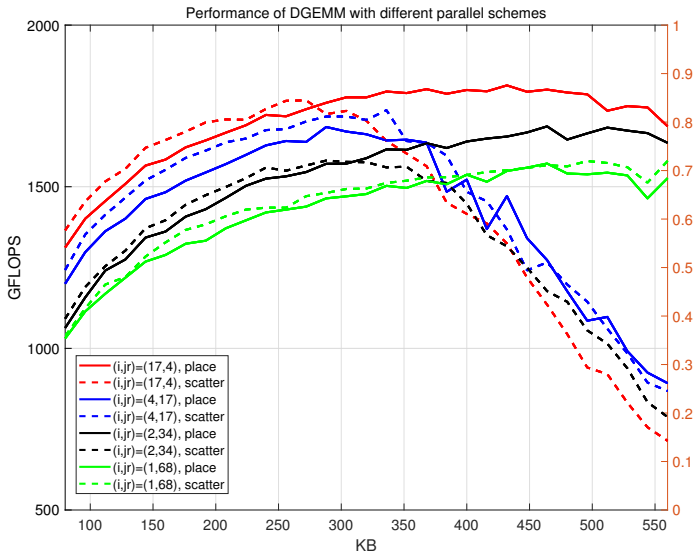

Figure 4.8: DGEMM performance comparison varying the parameter k_b while $m = n = 20000$, $k = 5000$

= CORES cases and dashed lines to indicate KMP_AFFINITY = SCATTER cases.

Notice that the performance results diminish after certain size of k_b under KMP_AFFINITY = SCATTER condition, except the $(i, jr) = (1, 68)$ case. When we set the degree of parallelization to be $(i, jr) = (1, 68)$, every core shares same block $\widetilde{A}$ and multiplies $\widetilde{A}$ with individual $\widehat{B}$. Hence, this combination exploits the sharing nature of L2 cache regardless of environment variable options. That is why, both cases achieve similar performance result varying the parameter k_b.

On the other hand, the performance of DGEMM kernel remains steady or increases as the size of k_b becomes bigger with OMP_PLACES = CORES option, except the $(i, jr) = (4, 17)$ case. When we set the degree of parallelization to be $(i, jr) = (4, 17)$, there exist two cores in same tile which hold different $\widetilde{A}$ because the degree of parallelization on jr-loop is odd. Therefore, $(i, jr) = (4, 17)$ combination cannot exploit the sharing nature of L2. This explains that the performance results under both environment settings are

similar.

Now, we compare the performance results between different (i, jr) combinations. We can achieve better performance when increase the number of threads parallelizing the i-loop. In Section 3.4, we examine that jr-loop has better parallel potential then i-loop. However, the experiment results are not consistent with our analysis. This discordance comes from the imbalance between packing time and computing time. As discussed in the literature [13, 15], there is not enough computing time to amortize the packing of $\widetilde{A}$. Inside the nested loop, the threads are grouped together in teams and pack their own block $\widetilde{A}$. If we distribute large number of threads to the jr-loop to exploit its rich parallelization potential, the degree of parallelization on the i-loop diminishes. Then, the time spent in packing $\widetilde{A}$ increases while the time spent in computing the jr-loop decreases. Hence, we need to increase the degree of parallelization for the i-loop to reduce packing time so that balance with the computing time. Consequently, $(i, jr) = (17, 4)$ parallelization scheme obtains the best performance with $(m_r, n_r, m_b, k_b) = (31, 8, 124, 432)$

There are not many open source BLAS libraries that support the KNL. BLIS is a portable software framework for instantiating high-performance BLAS-like dense linear algebra libraries and supports the Intel MIC architecture for level 3 BLAS [16]. In Figure 4.9, we compare performance reuslt of our implementation with BLIS and Intel MKL.

We observe that our parallel implementation makes a loss on performance when the sizes of matrices are small. To solve this problem, we employ the following environment variables.

- KMP_HOT_TEAMS_MODE=1
- KMP_HOT_TEAMS_MAX_LEVEL=2

According to Jeffers et al. [8], HOT_TEAMS environment variable can help to reduce high overheads of creating and destroying the threads in nesting OpenMP parallel regions. By keeping a pool of threads alive during the execution of the non-nested parallel region, our DGEMM implementation can reach its optimal performance earlier.

Consequently, the performance of our implementation with HOT_TEAMS option is better than that of BLIS DGEMM kernel regardless of the sizes of

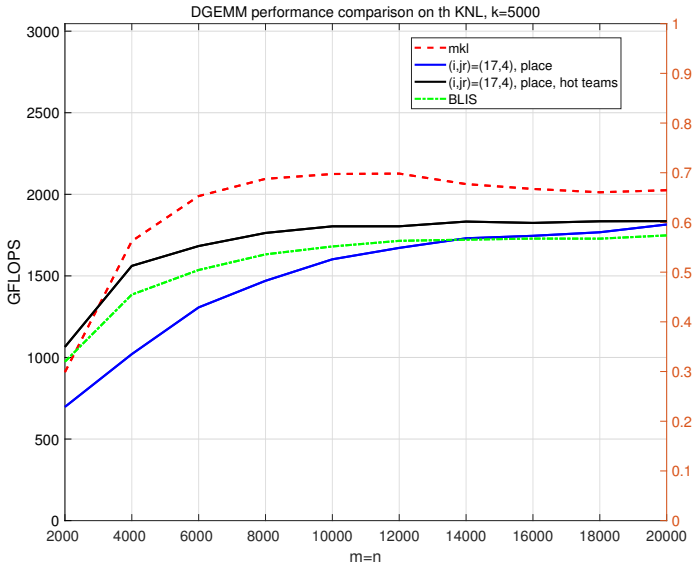

Figure 4.9: DGEMM performance comparison for different BLAS libraries on the KNL.

source matrices. Further, our DGEMM implementation achieves up to 93 percent of DGEMM performance using the Intel MKL.

Chapter 5

Conclusion and future works

In this thesis, we described in detail the implementation of the general matrix-matrix multiplication (GEMM) on the KNL. We developed double-precision GEMM (DGEMM) kernel for the KNL based on the blocked GEMM algorithm to reduce the bandwidth requirement. Based on the analysis of the machine structure, we derived several restrictions and suggestions to achieve high performance on the KNL. We determined optimal parameters such as blocking parameters, prefetch distances by performance experiments. Moreover, we refined the developing guidelines on the KNL. As a result, our implementation achieved up to 99 percent of DGEMM using the Intel MKL for a single core on the KNL. Also, we obtained up to 93 percent of DGEMM using the Intel MKL for all 68 cores of the KNL.

We still have several chances for further studies on the optimization of the DGEMM implementation. First, we did not control cache explicitly on our DGEMM implementation due to the lack of cache control intrinsic on the AVX-512 instruction set. We expect the performance enhancement of our DGEMM kernel with effective cache control.

Moreover, we observed that the HOT_TEAMS environment variables have differnet effect on the performance depending on the structure of nested loop. HOT_TEAMS environment variables cause bad effect to the performance when the degree of parallelization is $(2, 34)$. On the other hand, the same environment setting enhances the performance when the degree of parallelization is $(17, 4)$. The deeper investigation about HOT_TEAMS variables

would help to achieve better performance on the KNL.

While performance testing, we observed that L2 software prefetching is ineffective to the performance enhancement. L2 software prefetching became a pure burden when we insert L1 software prefetch on the kernel. This result is contradictory to the claim of Jeffers et al.[8] that L2 software prefetch is more important than L1 software prefetch. A convincing explanation would assist to apply optimal prefetching on the KNL.

Bibliography

[1] Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: ACM International Conference on Supercomputing 25th Anniversary Volume, pp. 253–260. ACM (2014)

[2] Dongarra, J. J., Du Croz, J., Hammarling, S., Hanson, R. J.: An extended set of FORTRAN basic linear algebra subprograms ACM Transactions on Mathematical Software (TOMS) **14**(1), pp. 1–17. (1988)

[3] Dongarra, J. J., Du Croz, J., Hammarling, S., Duff, I. S.: A set of level 3 basic linear algebra subprograms ACM Transactions on Mathematical Software (TOMS) **16**(1), pp. 1–17 (1988)

[4] Dongarra, J. J., Du Croz, J., Hammarling, S., Duff, I. S.: Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. ACM Transactions on Mathematical Software (TOMS) **16**(1), pp. 18-28 (1988)

[5] Goto, K., van de Geijn, R.A.: Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS) **34**(3), 12 (2008)

[6] Gunnels, J.A., Henry, G.M., Van De Geijn, R.A.: A family of high-performance matrix multiplication algorithms. In: International Conference on Computational Science, pp. 51–60. Springer (2001)

[7] Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A.G., Chrysos, G., Dubey, P.: Design and

implementation of the linpack benchmark for single and multi-node systems based on Intel® Xeon Phi Coprocessor. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 126–137. IEEE (2013)

[8] Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann (2016)

[9] Lawson, C. L., Hanson, R. j., Kincaid, D. R., Krogh, F. T.: Basic linear algebra subprograms for Fortran usage. ACM Transactions on Mathematical Software (TOMS) **5**(3), pp. 308–323 (1979)

[10] Lee, J., Kim, H., Vuduc, R.: When prefetching works, when it doesn't, and why. Architecture and Code Optimization (TACO) **9**(1), 2 (2012)

[11] Lim, R., Lee, Y., Kim, R., Choi, J.: An Implementation of matrix-matrix multiplication on the Intel KNL processor with AVX-512. Cluster Computing (Submitted)

[12] Low, T.M., Igual, F.D., Smith, T.M., Quintana-Orti, E.S.: Analytical modeling is enough for high-performance blis. ACM Transactions on Mathematical Software (TOMS) **43**(2), 12 (2016)

[13] Marker, B., Van Zee, F.G., Goto, K., Quintana-Ortí, G., Van De Geijn, R.A.: Toward scalable matrix multiply on multithreaded architectures. In: European Conference on Parallel Processing, pp. 748–757. Springer (2007)

[14] Peyton, J.L.: Programming dense linear algebra kernels on vectorized architectures. Master's thesis, The University of Tennessee, Knoxville (2013)

[15] Smith, T.M., Van De Geijn, R.A., Smelyanskiy, M., Hammond, J.R., Van Zee, F.G.: Anatomy of high-performance many-threaded matrix multiplication. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pp. 1049–1059. IEEE (2014)

[16] Van Zee, F.G., Van De Geijn, R.A.: BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software (TOMS) **41**(3), 14 (2015)

[17] Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing, pp. 1–27. IEEE Computer Society (1998)

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

[19] Xianyi, Z., Qian, W., Yunquan, Z. : Model-driven level 3 BLAS performance optimization on Loongson 3A processor In: Parallel and Distributed Systems, 2012 IEEE 18th International Conference, pp. 684–691. IEEE (2012)

국문초록

본 연구는 2세대 Intel Xeon Phi 프로세서 Knights Landing(KNL)에 대한 일반 행렬 곱셈 알고리즘의 설계 및 구현에 대해 다루고 있다. C 프로그래밍 언어와 Advanced Vector Extensions (AVX-512) 명령어 집합을 활용하여 일반 행렬 곱셈 알고리즘을 구현하였으며, 최적의 성능을 얻기 위한 몇가지 개발 가이드라인을 제시하였다. 또한 KNL 위에서의 병렬 프로그래밍과 관련된 여러가지 환경 변수에 대해 소개하였다. 그 결과 단일 코어 기준 Intel MKL 대비 99퍼센트의 성능을 이끌어낼 수 있었으며, 68 코어 기준 Intel MKL 대비 93 퍼센트의 성능을 얻을 수 있었다.

주요어휘: Knights Landing, 일반 행렬 곱셈, 벡터화, 최적화
학번: 2016-20230