Optimization of Matrix–Matrix Multiplication Algorithm for Matrix–Panel Multiplication on Intel KNL

Muhammad Rizwan
School of Computer Science & Eng.
Soongsil University
Seoul, South Korea
mrizwan@soongsil.ac.kr

Enoch Jung
School of Computer Science & Eng.
Soongsil University
Seoul, South Korea
enochjung@soongsil.ac.kr

Yoosang Park
School of Computer Science & Eng.
Soongsil University
Seoul, South Korea
yspark@soongsil.ac.kr

Jaeyoung Choi
School of Computer Science & Eng.
Soongsil University
Seoul, South Korea
choi@ssu.ac.kr

Yoonhee Kim
Department of Computer Science
Sookmyung Women's University
Seoul, South Korea
yulan@sookmyung.ac.kr

***Abstract*— The most scientific and numerical problems can be solved using the system of equations in linear algebra. Matrix–matrix multiplication is the foundation of linear algebra equations, and its optimization has an impact on the overall performance of a system. ScaLAPACK has established itself as the industry standard for dense linear algebraic computations, developed 30 years ago. Owing to advancements in microprocessor architectures, it is difficult to fully utilize the hardware capabilities of legacy software systems on modern architectures and achieve the maximum performance. In this study, we analyzed the effects of matrix size, register blocking parameters, and thread distribution on the performance, and improved our previously implemented matrix–matrix multiplication routine for matrix–panel multiplication, which performed well for large-sized square matrices. We also presented the ScaLAPACK QR factorization performance by replacing the double-precision general matrix–matrix multiplication routine (DGEMM) of ScaLAPACK with our matrix–matrix multiplication routine for a single node Intel Xeon Phi Knights Landing processor.**

Keywords*— *ScaLAPACK, Intel Knights Landing, QR factorization, matrix–matrix multiplication, AVX-512

I. Introduction

Generally linear algebra is used in numerous domains of science and engineering applications such as operations research and optimization studies, dynamical systems analysis and control, signal processing, computational chemistry, quantum mechanics, and even in big data analysis and machine learning. Because earlier computers had only a single processor, sequential programming was the basis for most software designs. In those days, the hardware performance has been improved with an increase of the clock speed, but hardware designers were faced with the "power wall" problem [1], and it was no longer possible to increase the performance by simply increasing the clock speed. Therefore the trend has shifts towards multi-core processors on a single chip, thereby shifting the processor design paradigm toward multi-core and manycore approaches. Software developers have therefore been required to rewrite their applications using a parallel programming model to achieve a higher performance by utilizing most of the hardware capabilities. This has given rise to parallel processing to improve the computing performance, perform complex calculations, and solve computationally expensive linear algebra equations.

Matrix multiplication is an essential component of linear algebra. Numerous applications of scientific and engineering problems have also used dense matrix–matrix multiplications. In this paper, we present a modified version of matrix multiplication algorithm on Intel Xeon Phi Knights Landing (KNL) processor, which is the second generation Intel Xeon Phi product, and supports vectorization and the AVX-512 SIMD instruction set to conduct extremely high-performance matrix computations [2].

A. Background

In earlier works [3, 4], a matrix–matrix multiplication algorithm was implemented using register and cache blocking, data prefetching, loop unrolling techniques, and the Intel AVX-512 instruction set. The register block sizes, cache block sizes, loop unrolling depth, and parallelization scheme all affect the performance of the matrix–matrix multiplication routine. We named our previously implemented matrix–matrix multiplication algorithm as USERDGEMM. We examined the effect of matrix size on the performance of USERDGEMM. The parameters for which this algorithm performed well for large square matrices, but the performance is lower when the resultant matrix C is not a square shape. In this study, we optimized the algorithm for matrix–panel multiplication operation.

The matrix multiplication operation is of the form $C = \alpha \cdot A \cdot B + \beta \cdot C$, where α and β are scalar, and A, B, and C are matrices of sizes $m \times k$, $k \times n$ and $m \times n$, respectively. When one of the dimensions is small, the matrix–matrix multiplication can be termed as panel–matrix, matrix–panel, or panel–panel, depending on whether m, n, or k is small, respectively. When n is small, the shape of the matrix–panel operation is illustrated in Fig. 1.

$$C = \alpha . A . B + \beta . C$$

Fig. 1. Matrix–panel multiplication operation

Data distribution is critical for the implementation of algorithms that uses cache blocking to improve the performance of cache-based hierarchical memory systems. To achieve a high performance and take advantage of the computational

979-8-3503-1008-5/22/$31.00 ©2022 IEEE

power of multiple computing units, the distribution and alignment of data require careful attention while moving data from one layout to another, as well as in the realignment of packing and unpacking during data transfer process, allowing each computing unit to access the required data in sequence from the memory. A regular array distribution can be distributed in block, cyclic, and block cyclic formats [5]. ScaLAPACK [6, 7] uses a block cyclic data distribution format.

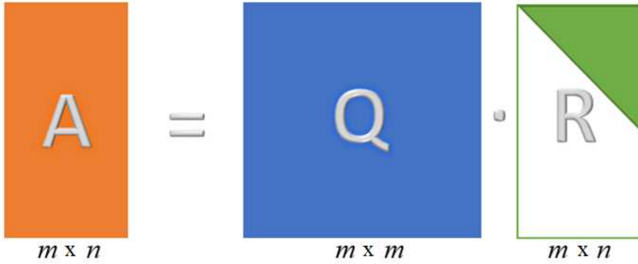

Fig. 2. QR decomposition

Given a matrix $A \in R_{m \times n}$ the QR decomposition of A is given by

$$A = Q \cdot R \quad (1)$$

where A, Q, and R are matrices of sizes $m \times n$, $m \times m$ and $m \times n$, respectively. In addition, Q is an orthogonal matrix, and R is an upper-triangular matrix. QR decomposition of matrix A is illustrated Fig. 2. A square matrix with real values is said to be orthogonal if it can be multiplied with its transposed matrix to produce an identity matrix and possesses commutative properties such as $Q^T \cdot Q = Q \cdot Q^T = I$ or $Q^T = Q^{-1}$, where Q^{-1} and Q^T are the inverse and transpose of Q, respectively.

QR factorization helps in solving linear least squares, eigenvalue, and singular value decomposition (SVD) related problems in linear algebra based computational methods and is also widely used in numerous applications including data processing, image processing, communication systems, and radar systems. Three of the most commonly used QR factorization methods are the Gram–Schmidt process, Householder transformations, and Givens rotations. The ScaLAPACK QR factorization algorithm relies on Householder reflectors because it is numerically more stable than the Gram–Schmidt algorithm [8]. QR factorization applies a series of Householder transformations of the following form:

$$H = I - \tau v v^T, \quad (2)$$

where I is the identity matrix, v is a column vector, and τ is a scalar.

The ScaLAPACK libraries are the *de facto* industry standard for dense linear algebraic computations, and the public version provides state-of-the-art algorithms for various problems. ScaLAPACK supports LU, QR, and Cholesky factorization. It solves singular value, eigenvalue, and linear least-square problems. ScaLAPACK accepts dense, tridiagonal, bidiagonal, banded, or packed symmetric or triangular matrices as the inputs. The ScaLAPACK source code is publicly available at www.netlib.org. ScaLAPACK is included in commercial packages from Apple, AMD, Compaq, Fujitsu, Hitachi, Hewlett-Packard, Intel, IBM, MathWorks, NEC, NAG, PGI, SUN, and Visual Numeric. Most Linux distributions, including Cygwin, Debian, and Fedora, also include it in their packages.

The implementation of a dense linear algebra system of equations is based on an open-source implementation of the BLAS [9] library, which is platform-dependent and targets the different architectures of AMD [10], IBM [11], Intel [12], and Nvidia [13]. Different variants of BLAS are available in libraries such as GotoBLAS [14], OpenBLAS [15], and BLIS [16]. The underlying base of the ScaLapack is the BLAS library. The ScaLAPACK has a modular architecture based on the HPC software packages BLAS, LAPACK [17], PBLAS [18], and BLACS [19]. ScaLAPACK is portable for multi-node systems that support MPI [20] and depends on PBLAS, similar to the dependence of LAPACK on BLAS as shown in Fig. 3.

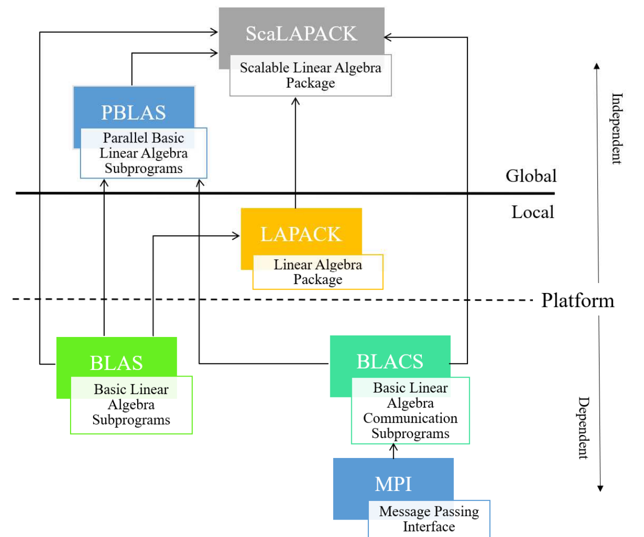

Fig. 3. The Calling stack and dependencies of ScaLAPACK

B. *Contribution*

The main contribution of this study:

- Evaluated the effect of matrix size, block size, register blocking parameters, and threads distribution on the performance of the matrix–matrix multiplication.
- We modified USERDGEMM, which performed well on square matrices; however, its performance remains poor when one of the dimensions is small. In this study, we improved the performance of USERDGEMM for matrix–panel like matrix–matrix multiplication.
- We replaced DGEMM, double-precision general matrix–matrix multiplication routine with our USERDGEMM routine in the ScaLAPACK QR factorization. We also evaluated the impact of this change on the QR factorization.

II. Related Work

LAPACK [17] is an open-source library that leverages an optimized BLAS [9] at the node level, and since its initial release approximately 30 years ago has been widely used on single nodes. By contrast, ScaLAPACK [6, 7] is built upon and having the same capabilities as LAPACK, and is designed for distributed memory systems. It uses both parallel BLAS (PBLAS) [18] and explicit distributed-memory parallelism to

extend LAPACK for multinode and distributed-memory structures.

Owing the evolution of the computer architecture, it is difficult to utilize modern heterogeneous high-performance computing machines to their full potential, and traditional and legacy libraries are rapidly becoming obsolete. For processors with hierarchical memory architectures, automatic kernel tuning software such as PHiPAC [21] and ATLAS [22] has been developed. The kernels automatically generated by the auto-tuned linear algebra softwares are not always optimal. Different algorithms may be optimal for different matrix dimensions depending on the shapes of the matrices involved, and hand-coded micro-kernels must be quantified in order to optimize performance. Intel AVX instructions were also used to improve the performance of kernels in published works [23, 24, 25, 26].

ScaLAPACK was designed and developed for computer systems with distributed memory. Current processor designs have evolved, and multi-core processors are now available on a single chip. Multi-core systems have shared memory architecture, whereas multinode systems have a distributed memory architecture. The hardware capabilities of multi-core computer systems have not been fully utilized by systems designed for distributed memory. Multinode computer systems using a multi-core computer as a node utilize a hybrid model of shared and distributed memories.

III. METHODOLOGY / PROPOSAL

We aim to improve the performance of QR factorization routine, using an AVX-512 instruction set-based matrix–matrix multiplication routine. ScaLAPACK QR factorization iteratively applies matrix–matrix multiplication and improving the performance of a matrix–matrix multiplication can yield a noticeable improvement in the QR factorization.

In this section, we discuss the existing implementation of the QR factorization routine, PDGEQRF in ScaLAPACK. We also discuss the changes made in its subroutine PDLARFB in subsection A, and then we discuss followed by a description of the modification to our USERDGEMM routine for improving the performance for matrix–panel multiplication operation in subsection B.

A. QR Factorization

The ScaLAPACK QR factorization routine, PDGEQRF, is a composite of the routines PDGEQR2, PDLARFT, and PDLARFB. PDGEQR2 further depends on the routines PDLARFG and PDLARF, which are responsible for generating elementary reflectors v_i and τ_i and updating the trailing submatrix, respectively. For the current column of the processes of Householder vector v in the process row, PDLARFT only computes the triangular matrix. In addition, PDLARFB is responsible for applying Q^T to the rest of the matrix from the left. We intend to improve only the PDLARFB routine, because it involves the DGEMM routine multiple times, as shown in Fig. 4.

To develop a distributed block form of this algorithm, it is necessary to represent the product of b elementary Householder matrices of order n as a block form of a Householder matrix. This is the most important step in creating the distributed block version of the algorithm,

$$Q = H_1 H_2 H_3 \ldots\ldots H_b = I - V\,T\,V^T. \qquad (3)$$

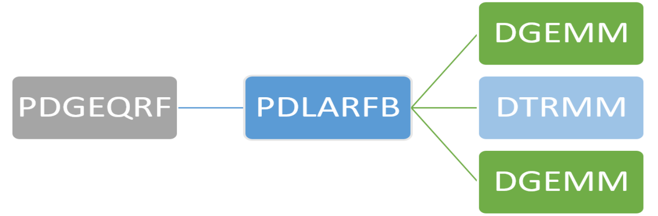

Fig. 4. ScaLAPACK QR Factorization routine PDGEQRF

In QR factorization, the primary purpose of the PDLARFB routine is to apply Q^T to the remainder of the matrix from the left: $\tilde{A} <= Q^T A$.

There are different ways to compute this routine, and the currently available version of ScaLAPACK computes this routine in the following manner:

This variant of PDLARFB computes as follows:

$$\begin{aligned} Q^T A &<= (I - V\,T\,V^T)^T A \\ &<= (I - V\,T^T\,V^T)\,A \\ &<= A - V\,T^T\,V^T A \\ &<= A - V\,(T^T\,V^T A) \\ &<= A - V\,(A^T\,V\,T)^T \\ &<= A - V\,(W\,T)^T \\ &<= A - V\widetilde{W}^T. \end{aligned}$$

In the current implementation of ScaLAPACK, the first DGEMM operation computes $W <= A^T\,V$ and conducts a (transpose)-(no transpose) operation of the matrix–matrix multiplication, as shown in Fig. 5 5. This operation is of a GEMP [27] type. DTRMM computes $\widetilde{W} <= W\,T$, which is a matrix–matrix operation, in which T is an upper triangular matrix. In addition, the second DGEMM operation shown in Fig. 4 computes and updates $\tilde{A} <= A - V\,\widetilde{W}^T$ and conducts a (no transpose)-(transpose) operation. An illustration of the second DGEMM operation is presented in Fig. 6.

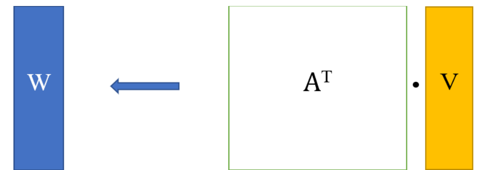

Fig. 5. Illustration of DGEMM operation for $W <= A^T\,V$

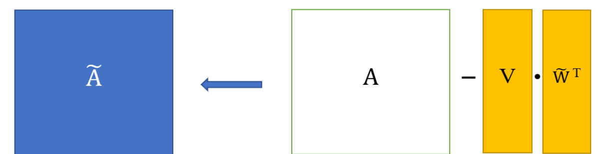

Fig. 6. Illustration of DGEMM operation for $\tilde{A} <= A - V\,\widetilde{W}^T$

The second variant of PDLARFB computes as follows:

$$\begin{aligned} Q^T A &<= (I - V\,T\,V^T)^T A \\ &<= (I - V\,T^T\,V^T)\,A \\ &<= A - V\,T^T\,V^T A \\ &<= A - V\,T^T\,W \\ &<= A - V\,\widetilde{W}. \end{aligned}$$

In this variant of PDLARFB, the first DGEMM operation computes $W <= V^T A$ and applies a (transpose)-(no transpose) operation, as shown in Fig. 7.

Fig. 7. Illustration of DGEMM operation for W <= V^TA

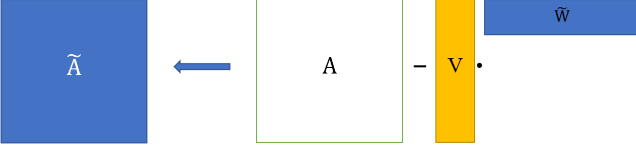

Fig. 8. Illustration of DGEMM operation for $\tilde{A}$ <= A – V$\tilde{W}$

In this implementation, DTRMM computes $\widetilde{W} <= T^T W$, which is a matrix–matrix operation in which T is an upper triangular matrix. In addition, the second DGEMM operation computes and updates $\tilde{A} <= A - V\widetilde{W}$, which conducts a (no transpose)-(no transpose) operation. An illustration of the second DGEMM operation is presented in Fig. 8.

The performance improvement of a matrix–matrix multiplication operation is critical for the overall performance improvement of the QR factorization routine. Matrix–matrix multiplication has been discussed extensively by Goto [27], and other works [3, 4], and double precision matrix–matrix multiplication operation in the row major order has been implemented and it works reasonably well for large matrices. The column major variant was also developed and used to improve the performance of the ScaLAPACK PDGEMM routine [4]. We modified the column major variant to improve the performance of the matrix–matrix multiplication routine for a matrix–panel operation.

Algorithm 1: Blocked matrix multiplication algorithm

for $i = 1, \ldots, m$ *in steps of* m_b ***do***
  for $p = 1, \ldots, k$ *in steps of* k_b ***do***
    Pack $A(i: i + m - 1, p{:}p + k_b\ 1)$ into $\tilde{A}$;
    for $j = 1, \ldots, n$ *in steps of* n_b ***do***
      Pack $B\ (p{:}\ k_b - 1, j{:}j + n_b - 1)$ into $\tilde{B}$;
      for $ir = 1, \ldots, m_b$ *in steps of* m_r ***do***
        for $jr = 1, \ldots, n_b$ *in steps of* n_r ***do***
          $\hat{A} = \tilde{A}\ (ir : ir + m_r - 1, :);$
          $\hat{B} = \tilde{B}(:, jr : jr + n_r - 1);$
          $\hat{C} += \hat{A} \cdot \hat{B};$
          Update C *using* $\hat{C}$;
        end
      end
    end
  end
End

Fig. 9. Blocked matrix–matrix multiplication algorithm

B. USERDGEMM MODIFICATION DETAIL

The blocked matrix–matrix multiplication is described in Fig. 9 as Algorithm 1. In this algorithm, m denotes the number of rows in matrices A and C; k represents the number of columns and rows in matrices A and B, respectively; and n represents the number of columns in matrices B and C.

The cache block parameters m_b, k_b, and n_b guide the algorithm on the sizes of the submatrix to copy, pack, and realign in sequential memory for the micro-kernel to compute a core matrix–matrix operation. The sizes of m_b, k_b, and n_b determines the data reuse size of the packed submatrices $\tilde{A}$ or $\tilde{B}$ and have a major impact on the performance of the routine because they are the key players in determining the effective usage of the cache. The matrices are stored in column-major format. In order to exploit the cache appropriately the matrices are blocked and packed in memory buffers. In the outer three loops matrices A and B are packed into $\tilde{A}$ and $\tilde{B}$, respectively; so that the submatrices are aligned in sequential manner.

The storage format of the A matrix is from row to column major in $\tilde{A}$ and, matrix B is from column to row major in $\tilde{B}$ and are illustrated in Fig. 10 and Fig. 11, respectively. This algorithm is for the BLAS Level 3 operation and thus it calculates:

$$C <= C - op(A) \times op(B) \qquad (4)$$

where op(X) = X or op(X) = X^T.

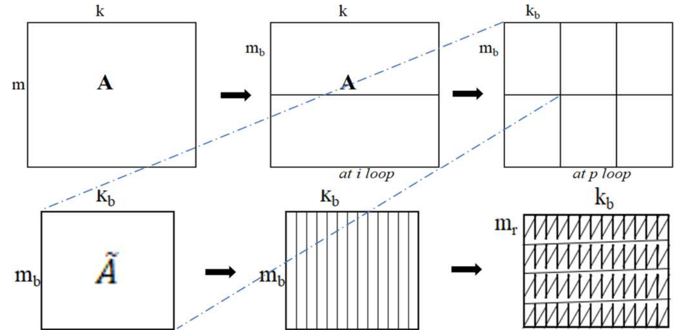

Fig. 10. Illustration of Pack A

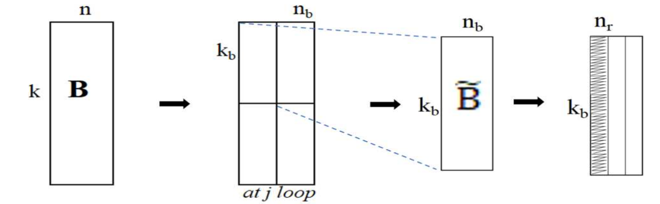

Fig. 11. Illustration of Pack B

For the same parameters for which this algorithm performed well for large square matrices, the performance is lower when the resultant matrix C is not a square shape. We considered a specific case of $C = C - A^T \cdot B$ with A as a square matrix of size m=k=10000, and B as a rectangular matrix of size k=10000 and n=300.

The Algorithm 1 was implemented in C language and has three main components, i.e., Pack A, Pack B and the micro-kernel routines. Micro-kernel is responsible for the core matrix–matrix computation and was implemented in AVX-

Algorithm 2: Micro-kernel pseudo code for (m_r, n_r) = (8,31)

```
register __m512d   _A0;
register __m512d   _C0, _C1, _C2, ……………, _C30;
_C0 = _mm512_loadu_pd(&C[0*ldc+0]);
_C1 = _mm512_loadu_pd(&C[1*ldc+0]);
:
:
_C30 = _mm512_loadu_pd(&C[30*ldc+0]);
for i = 0,…,kb-1 do
        _mm_prefetch(&A[L1_DIST_A+0], MM_HINT_T0);
        _A0 = _mm512_loadu_pd(&A[0]);
        _mm_prefetch(&B[L1_DIST_B+ 0..24], MM_HINT_T0);
        _C0 = _mm512_fnmadd_pd(_mm512_set1_pd(B[0]), _A0, _C0);
        _C1 = _mm512_fnmadd_pd(_mm512_set1_pd(B[1]), _A0, _C1);
        _C2 = _mm512_fnmadd_pd(_mm512_set1_pd(B[2]), _A0, _C2);
          :
          :
        _C30 = _mm512_fnmadd_pd(_mm512_set1_pd(B[30]), _A0, _C30);
        A += 8;
        B += 31;
end
_mm512_storeu_pd(&C[0*ldc+0], _C0);
_mm512_storeu_pd(&C[1*ldc+0], _C1);
:
:
_mm512_storeu_pd(&C[30*ldc+0], _C30);
```

Fig. 12. Micro-kernel implemented in AVX-512 pseudo code

Algorithm 3: Micro-kernel pseudo code for (m_r, n_r) = (8,15)

```
register __m512d   _A0;
register __m512d   _C0, _C1, _C2, ……………, _C29;
_C0 = _mm512_loadu_pd(&C[0*ldc+0]);
_C1 = _mm512_loadu_pd(&C[1*ldc+0]);
:
:
_C14 = _mm512_loadu_pd(&C[14*ldc+0]);
_C15 = _mm512_loadu_pd(&C [0*ldc+ nr*ldc]);
_C16 = _mm512_loadu_pd(&C [1*ldc+ nr*ldc]);
:
:
_C29 = _mm512_loadu_pd (&C [14*ldc+ nr*ldc]);
for i = 0,…,kb-1 do
        _mm_prefetch(&A[L1_DIST_A+0], MM_HINT_T0);
        _A0 = _mm512_loadu_pd(&A[0]);
        _mm_prefetch(&B[L1_DIST_B+ 0..24],_MM_HINT_T0);
        _C0 = _mm512_fnmadd_pd(_mm512_set1_pd(B[0]), _A0, _C0);
        _C1 = _mm512_fnmadd_pd(_mm512_set1_pd(B[1]), _A0, _C1);
        _C2 = _mm512_fnmadd_pd(_mm512_set1_pd(B[2]), _A0, _C2);
          :
          :
        _C14 = _mm512_fnmadd_pd(_mm512_set1_pd(B[14]), _A0, _C14);
        _C15 = _mm512_fnmadd_pd(_mm512_set1_pd(B[0+nr*kb]), _A0, _C15);
        _C16 = _mm512_fnmadd_pd(_mm512_set1_pd(B[1+nr*kb]), _A0, _C16);
        _C17 = _mm512_fnmadd_pd(_mm512_set1_pd(B[2+nr*kb]]), _A0, _C17);
        :
        :
        _C29 = _mm512_fnmadd_pd(_mm512_set1_pd(B[14+ nr*kb]), _A0, _C29);
        A += 8;
        B += 15;
end
_mm512_storeu_pd(&C[0*ldc+0], _C0);
_mm512_storeu_pd(&C[1*ldc+0], _C1);
:
:
_mm512_storeu_pd(&C[14*ldc+0], _C14);
:
:
_mm512_storeu_pd(&C[0*ldc+ nr*ldc], _C15);
_mm512_storeu_pd(&C[1*ldc+ nr*ldc], _C16);
:
:
_mm512_storeu_pd(&C[14*ldc+ nr*ldc], _C29);
```

Fig. 13. Modified micro-kernel pseudo code

512 intrinsic. Its pseudo code is described in Fig. 12 as Algorithm 2. For the same parameters but a different matrix size, the performance of USERDGEMM was less than 20% of the Intel MKL DGEMM. We changed the parameters m_b, k_b, and n_b and achieved a USERDGEMM performance of up to 50% that of MKL DGEMM.

We found that this degradation in the performance was caused by copying data from one memory location, realigning the data, and saving them to another memory location during packing. To enhance the parallelism and reduce the time required for copying and backend stalls, we then modified the thread distribution and applied all 68 threads instead of only 17, which improved the performance by up to 70% that of the MKL DGEMM for the specified case. We then implemented different variants of micro-kernel by modifying the original code for m_r=8 and n_r= 6, 7, 8, 10, 14, 15, 16, and 28. Using micro-kernel (m_r,n_r)=(8,15), we achieved maximum performance up to 90% that of the MKL DGEMM.

The main difference between the modified and the original implementation of USERDGEMM is of using two tiles of n_r=15 simultaneously, which are separated from each other by $n_r \times ldc$ instead of using a single tile of n_r=31, which reduces the number of iterations of the inner j_r loop. The pseudo code of micro-kernel mentioned in Fig. 13 handles only a full tile of size $m_r \times n_r$.

It handles the full block/tile and load data from C matrix. However, for handling partial tiles, there is another variant that does not read data from the resultant matrix memory location but initialize it in a buffer memory using instrinsic _mm512_setzero_pd. After the computation, the result is copied to the required position and updated at the corresponding memory location.

Matrix A was divided and packed into sub-matrix $\tilde{A}$ in column major order. Matrix B is packed into $\tilde{B}$ with row major order. This data realignment was performed using the packing routines. For the transpose matrix multiplication of $C = C - A^T \cdot B$, it is required to be packed into row to column major order. In addition, we modified the existing code for the packing routines using AVX-512 intrinsics.

IV. HARDWARE AND SOFTWARE

In this study, we used an Intel Xeon Phi 7250 machine, codenamed KNL, which has 68 cores and operates at a frequency of 1.4 GHz.

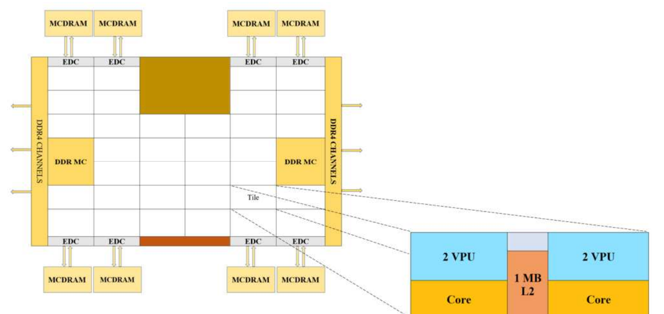

Fig. 14. Intel xeon phi 7250 (KNL) chip

It consists of a single socket and 34 tiles interconnected by a 2D mesh, each of which contains two cores, and the cores on each tile share a 1 MB L2 cache, as shown in Fig. 14. There are total 68 cores, and 4 threads per core. Each core has two vector processing units (VPU), 32 (AVX-512) vector registers, 8 mask registers, and an L1 cache of 32 KB. KNL equipped with 192 GB of DDR4 memory and 16 GB of high-bandwidth multi-channel DRAM (MCDRAM). KNL is capable of delivering performance of double-precision up to 44.8 GFLOPS per core [3].

CentOS version 8.5.2111 (kernel 4.18.0-348.7.1. el8-5.x86-64) and Intel oneAPI BaseKit and HPCKit for Linux versions 2021.3.0.3219 and 2021.3.0.3230, respectively, were installed on the machine. ScaLAPACK version 2.0.2 was used for this study. We used the Intel oneAPI module MKL library from the Base Kit, Intel Classic Compiler, and MPI library from the HPC Kit.

V. RESULTS AND EVALUATION

In this section, we discuss the results obtained and validated on an Intel Xeon Phi Processor 7250.

A. Performance of USERDGEMM

TABLE I lists the register block and cache block parameters for the original and modified versions of USERDGEMM. Fig. 15 shows a performance comparison of both variants of the original and modified USERDGEMM for $A^T \cdot B$ when A is a square matrix and matrix B has a panel shape for m=k=10000, and when varying the size of n from 40 to 5000. The graph demonstrates that the modified version of USERDGEMM performs significantly better than the original version for a value of n of less than or equal to 2000.

TABLE I. ALGORITHM PARAMETERS

Params	USERDGEMM (Original)	USERDGEMM (Modified)
m_r	8	8
n_r	31	15
m_b	6200	10000
n_b	124	150
k_b	336	500

The original code used 17 threads on Pack A, the same number of threads for the jth loop, and 4 threads for the inner *ir* loop. However, in the modified version, 68 threads were used for Pack A, the same number of threads were used for Pack B, and all 68 threads are used on the inner loop. We used the memkind library to pack and align submatrices $\tilde{A}$ and $\tilde{B}$ in the buffer on a high bandwidth memory (HBM).

We observed an improvement in performance for a small value of n. It was observed that when the size of n is close to m, the performance of the original USERDGEMM is higher than that of the modified variant of USERDGEMM.

The performance of the (transpose)-(no transpose) operation is shown in Fig. 15, and the performance of the (no transpose)-(no transpose) operation is almost identical for matrices of the same size. The sizes of m_b, k_b, and n_b impact on the performance of the routine, and k_b is crucial for the performance of all subroutines Pack A, Pack B, and the micro-kernel. Theoretically, a larger size of k_b can produce a better micro-kernel performance because it blocks data for a lengthy period of time and reuses the data; however, the larger size of the k_b affects the cache blocking, and the number of cache misses increases. The cache blocking parameters m_b and k_b, which produce a good performance for Pack A, whereas the same value of k_b with n_b limits the performance of Pack B because n is small. The cache and register blocking parameters for which we achieved a good performance are listed in TABLE I. In this study, we have improved the performance of USERDGEMM for matrix–panel operation when m=k and n is small.

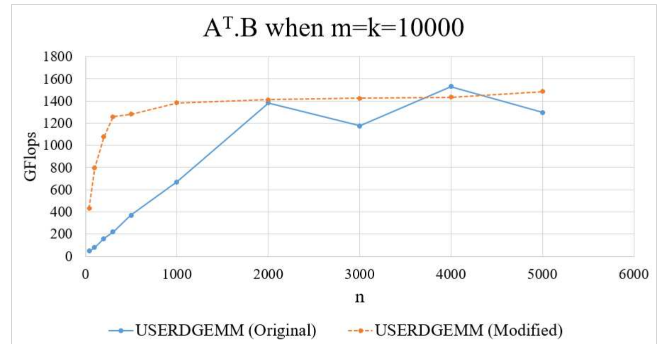

Fig. 15. Performance comparison of original and modified USERDGEMM

B. QR factorization performance

We tested the performance of the ScaLAPACK QR factorization routine with 2 USERGEMM routines, and the results are shown in Fig. 16. The matrix dimensions are ranged from 4000 to 36000, the solid line represents the QR factorization performance using modified USERDGEMM routine, and the dashed line represents the performance using original USERDGEMM routine. The performance using modified USERDGEMM is higher than using the original routine. The performance of the first variant of QR factorization using modified USERDGEMM is nearly double as compared to the performance using the original USERDGEMM routine for matrix of size 36000.

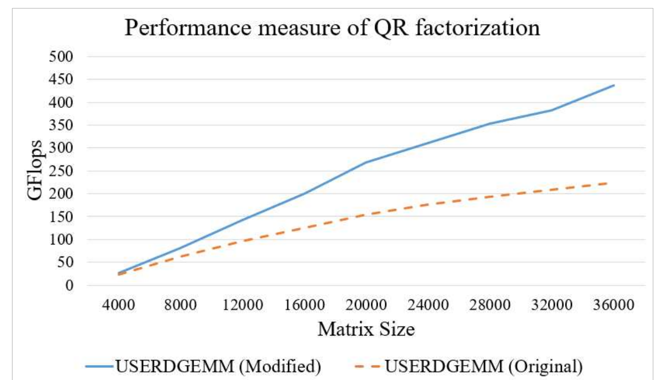

Fig. 16. Performance measure of QR factorization.

TABLE II lists the performance of QR factorization variants, for matrix of size 36000. The performance of the first variant of QR factorization using modified USERDGEMM is higher. However, the performance of second variant using modified USERDGEMM is lower than

the original routine, but better than the first variant using original USERDGEMM routine.

TABLE II. QR FACTORIZATION VARIANTS

Performance for matrix of size 36000				
Variants	**First**		**Second**	
USERDGEMM	Original	Modified	Original	Modified
GFlops	224.02033	436.12085	372.65709	240.97167

VI. CONCLUSION

This paper presented an improved version of matrix–matrix multiplication routine USERDGEMM for matrix–panel multiplication operation. Despite having the same memory structure, the optimal blocking parameters and thread distribution for matrices with different dimensions are not the same. When matrices are not in same size, the blocking parameters to amortize the cost of packing matrix A are not optimal for packing the data of matrix B, and vice versa. Therefore, the optimal blocking parameters for matrix–panel multiplication cannot yield the best performance for matrix–matrix or panel–matrix multiplication operations.

We also presented the performance of the ScaLAPACK QR factorization by replacing DGEMM with USERDGEMM routines. Using our modified USERDGEMM routine, a significant performance improvement was achieved in the first variant of the QR factorization as compared to the original matrix-matrix multiplication routine. This study is validated on a single node Intel Xeon Phi Knights Landing. In near future, we plan to test the routine on Intel next generation high-performance processor, Xeon Scalable processor (SKL) [29]. We will also extend our work to multi-node cluster environment of both Intel KNL and SKL processors.

ACKNOWLEGEMENT

This work was supported by the Supercomputer Development Leading Program of the National Research Foundation of Korea (NRF) funded by the Korean government (MSIT) (No. 2020M3H6A1084984). Also this work was supported by the National Supercomputing Center with supercomputing resources including technical support (No. KSC-2022-CRE-0202).

REFERENCES

[1] C. Meenderinck and B. Juurlink, "(When) Will CMPs Hit the Power Wall?" Euro-Par 2008 Workshops - Parallel Processing, pp. 184-193, 2009.

[2] C. Byun, "Optimizing Xeon Phi for Interactive Data Analysis," 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019.

[3] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512," Cluster Computing, vol. 21, no. 4, pp. 1785–1795, 2018.

[4] Y. Park, R. Kim, T.M.T. Nguyen, and J. Choi, "Improving blocked matrix-matrix multiplication routine by utilizing AVX-512 instructions on intel knights landing and xeon scalable processors" Cluster Computing, pp. 1-11, 2021.

[5] J. Choi, D. Walker, J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," Concurrency: Practice and Experience, vol. 6, no. 7, pp. 543-570, 1994.

[6] J. Choi, J. Dongarra, R. Pozo, and D. Walker, "ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers," Proceedings of the fourth symposium on the Frontiers of Massively Parallel Computation, 1994.

[7] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," Scientific Programming, vol. 5, no. 6, pp. 173-184, 1996.

[8] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langem, "QR factorization of tall and skinny matrices in a grid computing environment," 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010.

[9] "BLAS (Basic Linear Algebra Subprograms)," https://netlib.org/blas/ (accessed: July. 31, 2022).

[10] "Compiler Usage Guidelines for AMD64 Platforms Application Note," [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/32035.pdf.

[11] C. Gomez, "Engineering and Scientific Computing with Scilab," Springer Science & Business Media, 2012.

[12] E. Wang, "High-Performance Computing on the Intel Xeon Phi," 2014.

[13] "cuBLAS | NVIDIA Developer," https://developer.nvidia.com/cublas (accessed: July. 31, 2022).

[14] K. Goto and R. Van De Geijn, "High-performance implementation of the level 3 BLAS," ACM Transactions on Mathematical Software, vol. 35, no. 1, pp. 1-14, 2008.

[15] "OpenBLAS: An optimized BLAS library," https://www.openblas.net/ (accessed: Jul. 31, 2022).

[16] F. G. Van Zee and R. A. Van de Geijn, "BLIS: A Framework for Rapidly Instantiating BLAS Functionality," ACM Transactions on Mathematical Software, vol. 41, no. 3, pp. 1-33, 2015.

[17] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, D. Sorensen, "LAPACK: A portable linear algebra library for high-performance computers," Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, pp. 2-11, 1990.

[18] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, "A proposal for a set of parallel basic linear algebra sub-programs," Lecture Notes in Computer Science, pp. 107-114, 1996.

[19] C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage," ACM Trans. Math. Softw. vol. 5, no. 3, pp. 308–323, 1979.

[20] L. Clarke, I. Glendinning, and R. Hempel, "The MPI Message Passing Interface Standard," Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Switzerland, pp. 213-218, 1994.

[21] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Author retrospective for optimizing matrix multiply using PHiPAC," ACM International Conference on Supercomputing 25th Anniversary Volume, pp. 42-44, 2014.

[22] R. Whaley, A. PetitetJack, J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," Parallel Computing, vol. 27, no. 1–2, pp. 3-35, 2001.

[23] R. Kim, J. Choi, and M. Lee, "Optimizing parallel GEMM routines using auto-tuning with Intel AVX-512," Proceedings of the Inter. Conf. on HPC Asia, 2019.

[24] S. A. Hassan, A. Hemeida, and M. M. Mahmoud, "Performance Evaluation of Matrix-Matrix Multiplications Using Intel," Microprocessors and Microsystems, vol. 47, pp. 369-374, 2016.

[25] R. Lim, Y. Lee, R. Kim, and J. Choi, "OpenMP-based parallel implementation of matrix-matrix multiplication on the intel knights landing," Proceedings of Workshops of HPC Asia, 2018.

[26] M. E. Guney, "Optimizing Matrix Multiplication on Intel Xeon Phi TH x200 Architecture," 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH), 2017.

[27] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," ACM Transactions on Mathematical Software, vol. 34, no. 3, pp. 1-25, 2008.

[28] "Intel Xeon Scalable Processors - View Latest Generation Xeon," https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable.html (accessed: July. 31, 2022).