The Journal of Supercomputing (2024) 80:13813–13836
https://doi.org/10.1007/s11227-024-06002-2

Revisiting the performance optimization of QR factorization on Intel KNL and SKL multiprocessors

Muhammad Rizwan[1] · Enoch Jung[1] · Jongsun Choi[1] · Jaeyoung Choi[1]

Accepted: 16 February 2024 / Published online: 13 March 2024

Abstract

This study focused on the optimization of double-precision general matrix–matrix multiplication (DGEMM) routine to improve the QR factorization performance. By replacing the MKL DGEMM with our previously developed blocked matrix–matrix multiplication routine, we found that the QR factorization performance was suboptimal due to a bottleneck in the $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation. We present an investigation of the limitations of our matrix–matrix multiplication routine. It was found that the performance of the matrix multiplication routine depends on the shape and size of the matrices. Therefore, we recommend different kernels tailored to matrix shapes involved in QR factorization and developed a new routine for the $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation. We demonstrated the performance of the proposed kernels on the ScaLAPACK QR factorization routine by comparing them with the MKL, OPENBLAS, and BLIS libraries. Our proposed optimization demonstrates significant performance improvements in the multinode cluster environments of the Intel Xeon Phi Processor 7250 codenamed Knights Landing (KNL) and Intel Xeon Gold 6148 Scalable Skylake Processor (SKL).

Keywords QR factorization · matrix–panel multiplication · DGEMM · Intel Xeon Phi (Knights Landing) · Intel Xeon Scalable (Skylake) processor

✉ Jaeyoung Choi
choi@ssu.ac.kr

Muhammad Rizwan
mrizwan@soongsil.ac.kr

Enoch Jung
enochjung@soongsil.ac.kr

Jongsun Choi
jongsun.choi@ssu.ac.kr

[1] School of Computer Science and Engineering, Soongsil University, Seoul, Korea

1 Introduction

ScaLAPACK [1, 2] has been the industry standard for dense linear algebraic computation for the past 30 years. Our aim was to improve the performance of QR factorization that uses a basic linear algebra subprogram (BLAS) [3] double-precision general matrix–matrix multiplication (DGEMM) operation. The general matrix multiplication operation is of the form

$$C := aAB + \beta C, \tag{1}$$

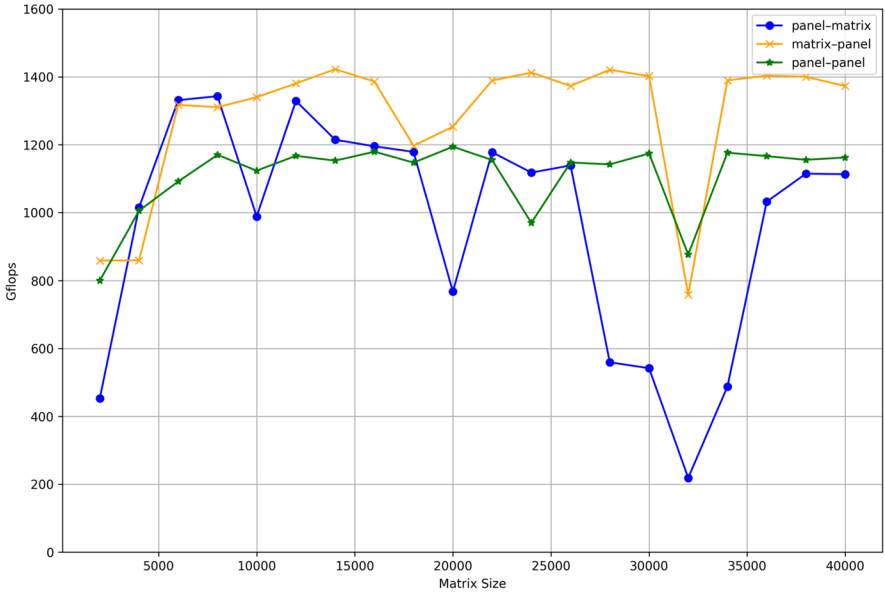
where α and β are scalars, and A, B, and C are matrices of sizes $m \times k$, $k \times n$, and $m \times n$, respectively.

In blocked matrix multiplication, the matrices are divided into submatrices. The dimensions of these submatrices are referred to as block sizes. When both the dimensions are small, it is referred to as a block. The term panel is used when only one dimension is small. Depending on whether m, n, or k is small, the matrix–matrix multiplication in Eq. (1) can be referred to as panel–matrix, matrix–panel, or panel–panel, respectively [4].

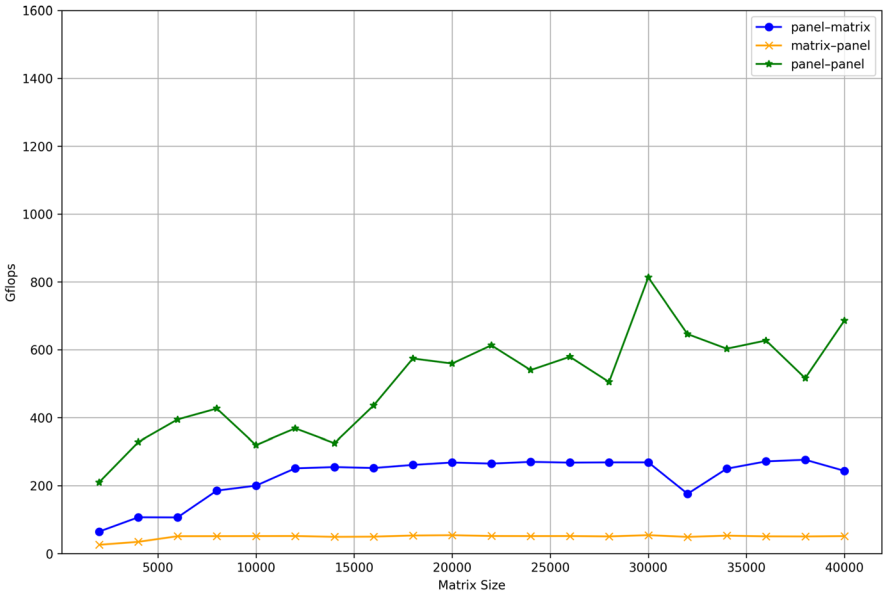
Gunnels et al. [5] presented a family of algorithms designed for matrix computations, and analyzed the cost of data movement between different memory layers in a multilevel memory architecture. Goto and Van De Geijn [6, 7] presented a layered approach for implementing general matrix–matrix multiplication (GEMM). In an earlier study [8], a blocked matrix–matrix multiplication operation was implemented for large matrices stored in the row-major. The aforementioned routine achieved up to 99 percent of the MKL DGEMM performance on a single core, and the parallel implementation achieved good scaling results that were more than 90 percent of the MKL DGEMM [8, 9]. The column major variant of the double-precision blocked matrix–matrix multiplication algorithm was also developed based on the conventional Goto algorithm in C using an Intel AVX-512 intrinsic and named OUR_DGEMM [4, 10]. Tuyen et al. [11] presented a method for optimizing ScaLAPACK LU factorization by optimizing some routines in the BLAS library, including the DGEMM routine, and replacing the MKL DGEMM routine with OUR_DGEMM, illustrating that the efficacy of the proposed LU factorization routine was comparable to that of the legacy ScaLAPACK LU factorization routine.

However, the performance of QR factorization was not optimal. The motivation behind this study was to identify the reasons for the performance bottleneck in QR factorization. We investigated the performance issues of OUR_DGEMM and conducted a detailed analysis focusing on optimizing the multiplication operations involved in QR factorization. QR factorization casts half of the computation involving panel–panel multiplication, and the other half requires either the matrix–panel or panel–matrix multiplication operation. We discovered that the suboptimal performance of $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation caused low performance of QR factorization.

A performance comparison of the matrix multiplication operation for different operations of the MKL DGEMM and OUR_DGEMM is shown in Fig. 1a, b, respectively. Figure 1 shows a comparative analysis of the performance of matrix



(a) MKL DGEMM



(b) OUR_DGEMM

Fig. 1 Performance comparison of matrix multiplication operations of **a** MKL DGEMM and **b** OUR_DGEMM for small dimension of size 40 on Intel KNL

multiplication operations when the matrices involved in the multiplication are of different shapes and sizes. The line with circular (•) data points presents the performance of panel–matrix multiplication. The line with cross-shaped data points (×) represents the performance of the matrix–panel multiplication. The line with star-shaped (⋆) data points presents the performance of panel–panel multiplication. For all the settings mentioned above, the small dimensions were set to 40.

The number of computations involved in the matrix multiplications is $2 \times m \times n \times k$ because the number of computational operations required is directly proportional to the number of elements in each matrix. However, in panel–matrix, matrix–panel, and panel–panel multiplication, the performances are different, even though the theoretical number of computations is the same. The performance was suboptimal, particularly for matrix–panel multiplication.

OUR_DGEMM performs nearly optimally on $m = n = k$ matrix multiplication but performs sub-optimally on matrices of different shapes. We discovered that our routine did not attain an optimal performance for different matrix shapes based on the operation and size of the matrices involved in multiplication.

QR factorization involves $A^{\mathrm{T}} \cdot B$ and $A \cdot B^{\mathrm{T}}$ multiplication operations. In our routine, the performance of $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation was very bad, so we developed a new routine for $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication, also optimized the existing routine for $A \cdot B^{\mathrm{T}}$ panel–panel multiplication operation, as discussed in Sect. 3.

In this study, we used the term "micro-kernel" as the core computational unit responsible for performing multiplication operations. The term "kernel" refers to the implementation of an algorithm with specific parameters and a micro-kernel. The term "routine" refers to the implementation of an algorithm without specific parameters. The routine can also be a kernel but has a different connotation. For example, an algorithm with different micro-kernels may be considered the same routine but treated as different kernels.

The main contribution of this study:

- The performance issues of OUR_DGEMM were analyzed when applied to matrices of different shapes. This highlights that a single kernel for matrix multiplication may not be universally optimal, and it is crucial to select different kernels based on the shapes and sizes of the matrices involved.
- Implemented a new routine for $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation and optimized the existing routine for $A \cdot B^{\mathrm{T}}$ panel–panel multiplication operation.
- The matrix multiplication performance using the new kernels in the ScaLAPACK QR factorization routine on Intel KNL and SKL multiprocessors was evaluated by comparing MKL [12], OPENBLAS [13], and BLIS [14].

2 Background

ScaLAPACK is a parallelized version of LAPACK [15, 16] that has a modular architecture based on HPC software packages and is portable for multinode systems that support MPI [17]. ScaLAPACK was built on top of the Basic Linear Algebra

Communication Subprograms (BLACS) [18], which provide a standardized API for communication between processes. ScaLAPACK uses both parallel BLAS (PBLAS) [19] and explicit distributed memory parallelism to extend LAPACK to multinode and distributed memory structures. ScaLAPACK solves systems of linear equations, least squares problems, eigenvalue problems, and singular value decompositions. ScaLAPACK was designed to operate with distributed memory systems that use a message-passing paradigm, such as clusters and supercomputers. ScaLAPACK provides several routines for factoring matrices, including QR factorization.

QR Factorization. A method for factoring a matrix into the product of an orthogonal matrix and an upper triangular matrix. The QR factorization of a matrix A is a technique used in linear algebra to decompose a matrix A into two matrices, Q and R, where $A := Q \cdot R$, A is an $m \times n$ matrix, Q is an $m \times m$ orthogonal matrix, and R is an $m \times n$ upper-triangular matrix, as shown in Fig. 2.

In QR factorization, the primary purpose of the routine is to apply Q^{T} to the remainder of the matrix from the left: $\tilde{A} := Q^{\mathrm{T}} \cdot A$. There are different methods to compute this routine, and the currently available version of ScaLAPACK computes it as follows:

$$\begin{aligned} Q^{\mathrm{T}} \cdot A &:= (I - VTV^{\mathrm{T}})^{\mathrm{T}} A \\ &:= (I - VT^{\mathrm{T}}V^{\mathrm{T}})A \\ &:= A - VT^{\mathrm{T}}V^{\mathrm{T}}A \\ &:= A - V(T^{\mathrm{T}}V^{\mathrm{T}}A) \\ &:= A - V(A^{\mathrm{T}}VT)^{\mathrm{T}} \\ &:= A - V(WT)^{\mathrm{T}} \\ &:= A - V\tilde{W}^{\mathrm{T}} \end{aligned}$$

In the current implementation of ScaLAPACK, the first DGEMM operation computes $W := A^{\mathrm{T}} \cdot V$ and conducts a (transpose)-(no-transpose) operation of the matrix multiplication, as shown in Fig. 3. This operation is a matrix–panel multiplication. In addition, the second DGEMM operation computes and updates

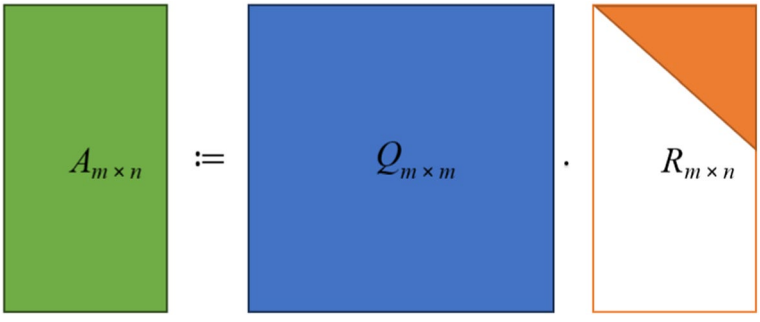


Fig. 2 QR factorization

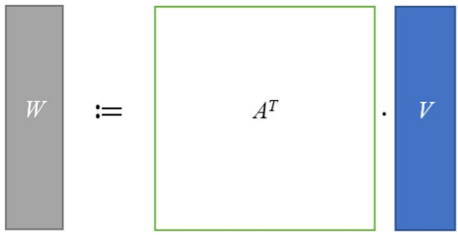


Fig. 3 Illustration of DGEMM operation for $W := A^{\mathrm{T}} \cdot V$

$\tilde{A} := A - V \cdot \tilde{W}^{\mathrm{T}}$ and conducts a (no-transpose)-(transpose) operation. The second DGEMM operation is illustrated in Fig. 4. This operation is called panel–panel multiplication.

3 Methodology

In this section, we analyze the performance issues of OUR_DGEMM and address them to improve QR factorization performance.

3.1 Revisiting OUR_DGEMM

The double-precision blocked matrix multiplication algorithm, OUR_DGEMM, was implemented in C with the AVX-512 intrinsic. A blocked matrix multiplication algorithm was used to improve the effectiveness of memory hierarchies. In addition to cache blocking, vectorization, loop un-rolling, parallelism, and data prefetching techniques have been utilized to optimize the performance of matrix–matrix multiplication. In this implementation, the matrices are stored in a column major order and packed into a contiguous array for the core computation component, the micro-kernel. The pseudocode for the micro-kernel can be found in this study [4]. The implementation of OUR_DGEMM adheres to Algorithm 1, as illustrated in Fig. 5.

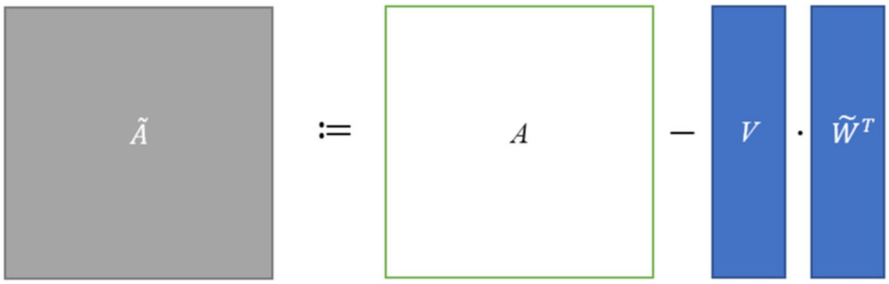


Fig. 4 Illustration of DGEMM operation for $\tilde{A} := A - V \cdot \tilde{W}^{\mathrm{T}}$

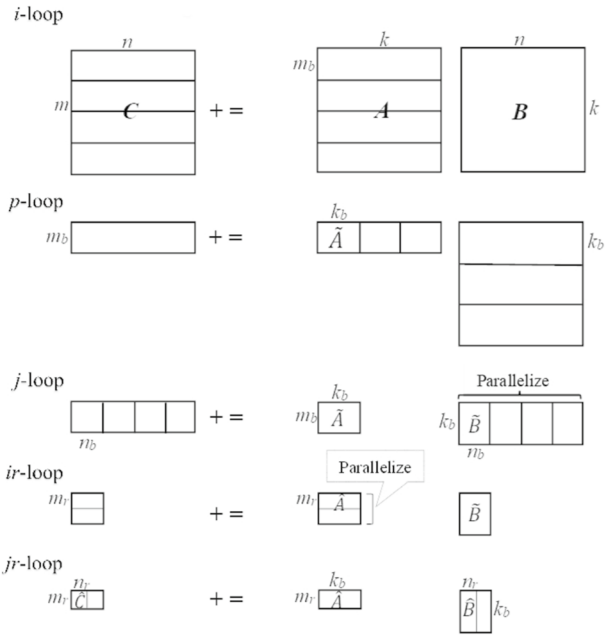


Fig. 5 Illustration of Algorithm 1

Algorithm 1 Blocked matrix–matrix multiplication algorithm

for $i := 1$ to m in steps of m_b **do**
  for $p := 1$ to k in steps of k_b **do**
    Pack $A(i : i + m_b - 1, p : p + k_b - 1)$ into $\tilde{A}$
    Parallel for t_j number of threads
    for $j := 1$ to n in steps of n_b **do**
      Pack $B(p : p + k_b - 1, j : j + n_b - 1)$ into $\tilde{B}$
      Parallel for t_{ir} number of threads
      for $ir := 1$ to m_b in steps of m_r **do**
        for $jr := 1$ to 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 for
      end for
    end for
  end for
end for

In these loops of Algorithm 1, the term $\hat{C}$ += $\hat{A} \cdot \hat{B}$ is corresponding to the micro-kernel. The micro-kernel performs a series of rank-1 updates on the $m_r \times n_r$

block of matrix C to load and store data in the registers. The choice of register-blocking parameters m_r, n_r, and cache-blocking parameter k_b is crucial for optimizing the micro-kernel performance. The choice of m_r, n_r is limited by the number of available registers on the target machine, and the size of k_b should be large enough to amortize the cost of updating the $m_r \times n_r$ block of matrix C. The methodology for choosing the best m_r, n_r, and k_b is discussed in Sects. 3.2 and 3.7.

The three outer loops i, p, and j are responsible for partitioning the matrices into submatrices, as shown in Fig. 5. The subdivision of the matrices into blocks or panels depends on the loop sequences and cache block sizes. The inner two loops ir and jr correspond to loop tiling, which is a technique used for cache blocking. These loops divide the submatrices into smaller units called slivers. These loops divide the matrix multiplication operation into smaller computing problems that can be processed by the micro-kernel. The algorithm performance relies on many factors, such as loop sequences, parallelization of appropriate loops, register and cache blocking parameters, vectorization, loop unrolling, and prefetch distances. The performance of the algorithm was determined by the coordination of these factors.

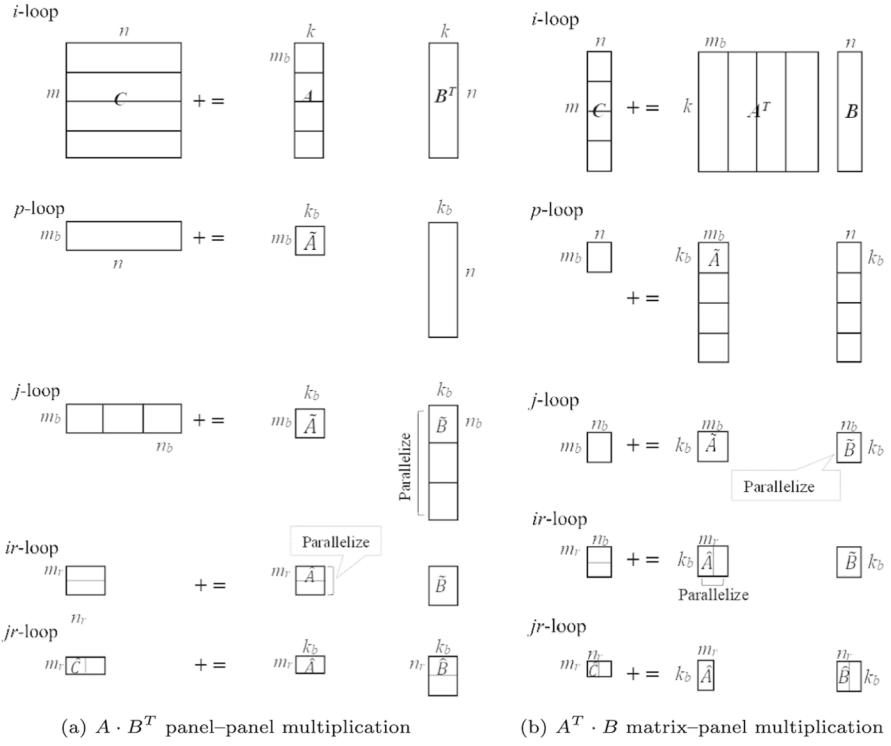


Fig. 6 Illustration of Algorithm 1 for **a** $A \cdot B^{\mathrm{T}}$ panel–panel multiplication on the left and **b** $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication on the right

For the computation of $A \cdot B^{\mathrm{T}}$ panel–panel multiplication and for $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication, the illustration of OUR_DGEMM in blocked diagrams is shown in Fig. 6.

To identify the performance issues of OUR_DGEMM for matrix–panel multiplication, it is necessary to analyze the data size for reuse. This analysis enables us to determine when and where each block is cached. Table 1 presents an analysis of the data reuse in the different loops of OUR_DGEMM implemented according to Algorithm 1.

Data are transferred across multiple levels of the memory hierarchy. Block $\hat{C}$ of size $m_{\mathrm{r}} \times n_{\mathrm{r}}$ resides in the registers and is reused k_{b} times in the innermost 6th loop. Block $\tilde{A}$ divided into sliver $\hat{A}$ of size $m_{\mathrm{r}} \times k_{\mathrm{b}}$ which resides in the L1 cache and is reused $n_{\mathrm{b}}/n_{\mathrm{r}}$ times in the 5th loop, *jr*. The panel of B is divided into blocks $\tilde{B}$ of size $n_{\mathrm{b}} \times k_{\mathrm{b}}$, residing in the L2 cache, and reused $m_{\mathrm{b}}/m_{\mathrm{r}}$ times in the 4th loop, *ir*. The submatrix of A divided into $\tilde{A}$ of size $m_{\mathrm{b}} \times k_{\mathrm{b}}$ that resides in the L3 cache or main memory when the L3 cache is not available, and is reused n/n_{b} times in the 3rd loop, *j*. Matrix C is divided into a submatrix of size $m_{\mathrm{b}} \times n$ because of loop *p*. Matrix A is divided into panels that are multiplied with matrix B of size $k \times n$, in the 1st loop, *i*.

3.2 Improving micro-kernel

For the choice of $m_{\mathrm{r}} \times n_{\mathrm{r}}$, Goto and Van De Geijn [6] argued that half of the available registers must be used to store the elements of $\hat{C}$; however, Lim et al. [8] demonstrated that the utilization of all registers achieves the optimal performance of micro-kernel. Both of our target multiprocessor machines, KNL and SKL, support AVX-512 intrinsic and have 32 vector registers per core. The possible combination of m_{r} and n_{r} can be determined using the following equation:

$$\frac{m_{\mathrm{r}}}{8}(1 + n_{\mathrm{r}}) \leq 32 \quad \text{and} \quad m_{\mathrm{r}} \mod 8 = 0. \tag{2}$$

In Eq. (2), $\frac{m_{\mathrm{r}}}{8}$ indicates the number of registers required to store the number of rows, and n_{r} indicates the number of registers required to store the number of columns of block $\hat{C}$, respectively. The term $m_{\mathrm{r}} \mod 8$ is important because AVX-512 operates on 512-bit registers, and each register can hold eight double-precision floating-point numbers. Parameters m_{r} and n_{r} determine the number of rows and columns of block $\hat{C}$ to be loaded and stored in the registers.

Table 1 Analytical review of data reuse in different loops of Algorithm 1

#	Loop	Data used	Size	Reuse factor
1	*i*	Matrix B	$k \times n$	m/m_{b}
2	*p*	Submatrix of C	$m_{\mathrm{b}} \times n$	k/k_{b}
3	*j*	$\tilde{A}$	$m_{\mathrm{b}} \times k_{\mathrm{b}}$	n/n_{b}
4	*ir*	$\tilde{B}$	$n_{\mathrm{b}} \times k_{\mathrm{b}}$	$m_{\mathrm{b}}/m_{\mathrm{r}}$
5	*jr*	$\hat{A}$	$m_{\mathrm{r}} \times k_{\mathrm{b}}$	$n_{\mathrm{b}}/n_{\mathrm{r}}$
6	k_{b}	$\hat{C}$	$m_{\mathrm{r}} \times n_{\mathrm{r}}$	k_{b}

Table 2 Optimal size of n_b and k_b for different (m_r, n_r) when $m = n = k = 6400$

(m_r, n_r)	(8, 31)	(16, 15)	(32, 7)	(24, 9)	(8, 20)
# of vector registers	32	32	32	30	21
n_b	124	135	126	135	120
k_b	438	400	400	438	400

The possible choices for m_r were 8, 16, 24, and 32. This allowed for the possible combination of (m_r, n_r) for the maximum possible utilization of the vector registers as (8, 31), (16, 15), (24, 9), and (32, 7). According to earlier research [8], the best combination for matrix–matrix multiplication in KNL is (8, 31). However, we observed that this combination did not work well for matrices with different dimensions, as shown in Fig. 1. We then evaluated the existing routine and performed experiments using different combinations of (m_r, n_r) on a single-core KNL to select the optimal micro-kernel. We found that (16, 15) is the second best option for matrix–matrix multiplication when all vector registers are used and $m_r \approx n_r$. In this analysis, we examined the performance on a single core with the assumption that the optimal performance routine on a single core could likewise achieve optimal performance in a multicore environment.

For the matrix multiplication operation, it was observed that the size of m_b is not crucial for cache blocking on KNL because of the loop order *ipj* and the absence of the L3 cache. Good performance is obtained when half of the L2 cache is used to store $\tilde{B}$, $\hat{A}$, and $\hat{C}$ [8, 9]. We observed that the performance was superior when the parameters were $(m_r, n_r, n_b, k_b) = (8, 31, 124, 438)$.

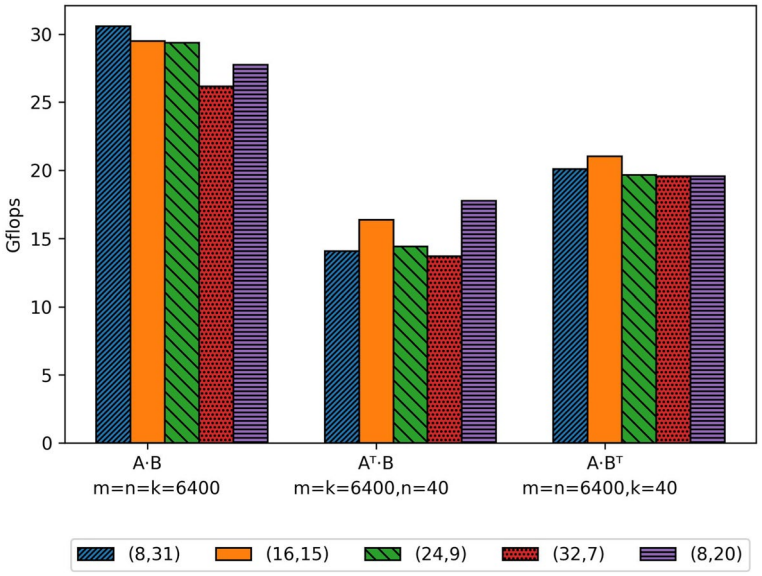


Fig. 7 Performance of OUR_DGEMM for different micro-kernels using (m_r, n_r) for $A \cdot B$ matrix–matrix multiplication when $m = n = k = 6400$, $A^T \cdot B$ matrix–panel multiplication when $m = k = 6400$ and $n = 40$, and $A \cdot B^T$ panel–panel multiplication when $m = n = 6400$ and $k = 40$ on KNL

KNL features two cores on the same tile that share 1MB of L2 cache. Accordingly, we considered Eq. (3) to calculate the size of n_b and k_b by taking the best-performing matrix–matrix multiplication operation parameters as benchmarks and adjusting them by varying the value of n_b and k_b experimentally for other combinations of (m_r, n_r) in Table 2.

$$n_b k_b + 2m_r k_b + 2m_r n_r \leq 64K \tag{3}$$

The performance of the different micro-kernels is shown in Fig. 7 for the matrix multiplication operations mentioned below:

- $A \cdot B$, matrix–matrix multiplication when $m = n = k = 6400$.
- $A^T \cdot B$, matrix–panel multiplication when $m = k = 6400$ and $n = 40$.
- $A \cdot B^T$, panel–panel multiplication when $m = n = 6400$ and $k = 40$.

In the case of matrix–panel multiplication operations, the n_b is limited by the size of n, and the size of $\tilde{B}$ cannot be sufficient for optimal performance using the same blocking parameters as for matrix–matrix multiplication. The most important parameter is k_b, which has the greatest impact on the performance. The size of k_b affects the sizes of $\hat{A}$, $\hat{B}$, $\tilde{A}$ and $\tilde{B}$ and eventually impacts the performance of the micro-kernel. In the case

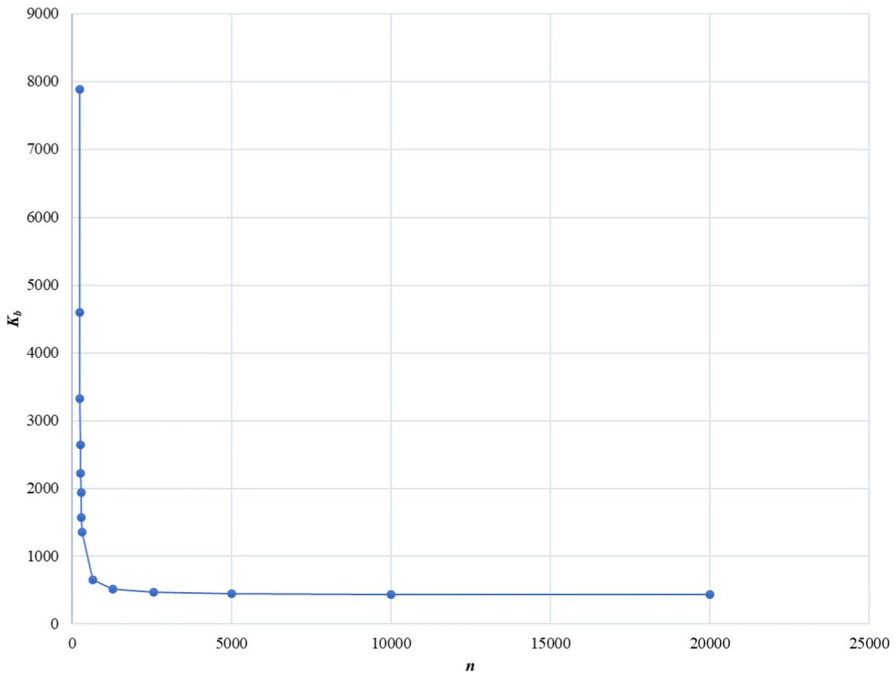


Fig. 8 Relationship between the size of n and the block size k_b for transferring data from memory to L2 cache in the computation of $m_b \times n$ elements of the submatrix of C on KNL

of a panel–panel multiplication operation, the block size of k_b is limited by the size of k. Accordingly, we adjusted the size of n_b and k_b in matrix–panel and panel–panel multiplication operations.

3.3 Analyzing the impact of *n* size on OUR_DGEMM

We analyzed the required memory bandwidth for OUR_DGEMM. The required bandwidth is the amount of data transferred divided by the time on computation. To compute the $m_b \times n$ elements of the submatrix of C, $2 \times m_b \times n \times k_b$ computations were performed, and at least $(m_b \times n + m_b \times k_b + k_b \times n)$ elements of data were transferred from the main memory to the L2 cache and $m_b \times n$ data were transferred back to the main memory.

- Data transfer: $(2 \times m_b \times n + m_b \times k_b + k_b \times n) \times (8 \text{ bytes})$
- KNL can compute 3,046.4 Gflops arithmetic operations per second: (1.4 GHz × 8 doubles × 2 VPUs × 2 IPC × 68 cores)
- Time on computation: $\frac{(2 \times m_b \times n \times k_b)}{(3{,}046.4 \times 10^9 \text{ s})}$

For KNL, the required bandwidth is given by

$$\left(\frac{2}{k_b} + \frac{1}{n} + \frac{1}{m_b}\right) \times 12185.6 \text{ GB/s} \tag{4}$$

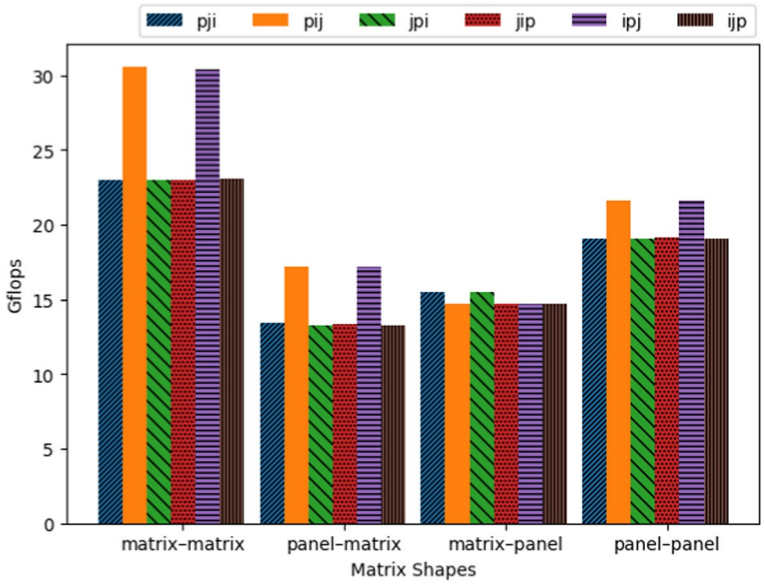


Fig. 9 Performance of different loop orders on single-core KNL for $A \cdot B$ matrix multiplication of different shapes of matrices when the largest dimension is 6400 and the smallest dimension is 40

Considering the optimal performing block sizes of matrix–matrix multiplication $(m_r, n_r, n_b, k_b) = (8, 31, 124, 438)$. Block $\tilde{B}$ of size $n_b \times k_b$ is 425 KB, and with that size of $\tilde{B}$, half of the L2 cache occupies with $\tilde{B}$, $\hat{A}$, and $\hat{C}$. But when $n = 40$, the block size n_b is limited by the size of n, and the size of $\tilde{B}$ reduced to 137 KB which is three times smaller. With that size of $\tilde{B}$ only $1/6$th of the portion of L2 covers with $\tilde{B}$, $\hat{A}$, and $\hat{C}$, which impacts the performance, and good performance can be obtained when half of the L2 cache is used to store the $\tilde{B}$, $\hat{A}$, and $\hat{C}$.

The size of m_b depends on the size of the L3 cache. On KNL, the L3 cache is not present, and for large values of m_b, $1/m_b$ is negligible in (4). When n is small, we need to change the size of k_b such that half of the L2 cache is used to store $\tilde{B}$, $\hat{A}$, and $\hat{C}$. Considering the case when n is large and $k_b = 438$, and varying the size of n the impact on k_b is presented in Fig. 8. The relationship between k_b and n follows a linear trend for n greater than 2000 and k_b remains almost consistent with a small difference. However, when n decreases, the required value of k_b increases exponentially, indicating an inverse relationship with the size of n. When n is less than 220, it is impossible to obtain a similar performance for matrix–panel multiplication as for matrix–matrix multiplication using OUR_DGEMM.

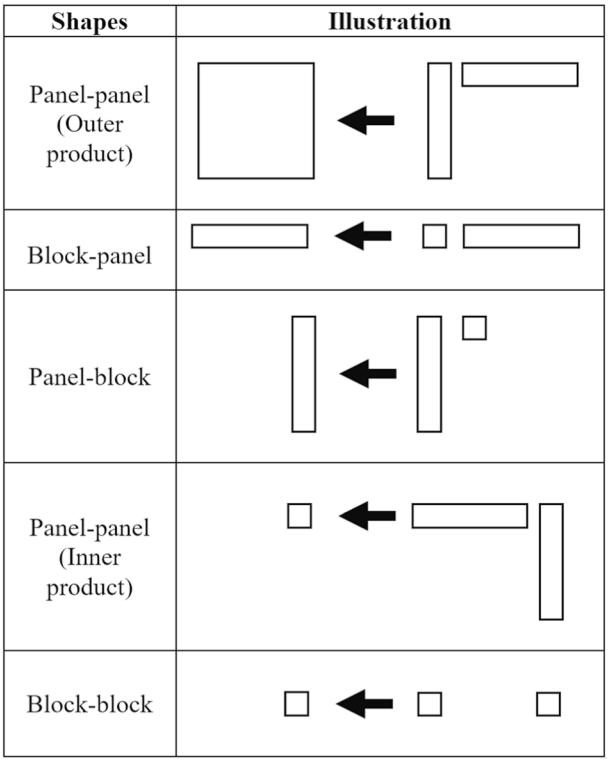


Fig. 10 Special shapes of submatrices

3.4 Parallelization scheme and loop orders

In OUR_DGEMM, the j loop was parallelized with t_j number of threads, and the ir loop was parallelized with t_{ir} number of threads. The detailed reasoning behind this parallelization scheme was elaborated in the paper [8]. We also observed that the performance of the matrix–matrix multiplication operation is superior when $(t_j, t_{ir}) = (17, 4)$. However, the performance of the panel–panel multiplication operation improved when $(t_j, t_{ir}) = (68, 1)$. However, in the case of matrix–panel multiplication operation, this parallelization scheme does not work well.

We analyzed the data reuse factors for Algorithm 1 by referring to Table 1. In the case of matrix–panel multiplication operation, the size of n is very small; therefore, the ratio of n/n_b is not sufficient to parallelize the j loop. However, the ratio of m_b/m_r is large enough to parallelize the ir loop. We found some performance improvements using $(t_j, t_{ir}) = (1, 68)$ with limited success. In the loop order of ipj, i loop is not a good candidate for parallelism, because each thread multiplies its own copy of panel of A with the shared data of matrix B which resides in the main memory. The ratio k/k_b is reasonable for parallelizing the p loop; however, parallelizing the p loop that iterates over the shared dimension k of the multiplying matrices may not yield significant benefits because of the synchronization overhead to ensure accurate result accumulation. The j loop is also not a good candidate for parallelization in matrix–panel multiplication because of its small ratio of n/n_b.

Subsequently, an analysis was conducted on different possible combinations of loop orders for matrices with different shapes, as shown in Fig. 9. The 3rd loop reveals better performance while traversing along the largest dimension. This also increases the potential for the parallelization of the operation by applying parallelization on 3rd loop. In this case, each thread allocates a different block of the submatrix stored in the L2 cache, and all threads access the same block of the multiplying matrix, depending on the loop order. Based on this observation, the loop structure outlined in Algorithm 1 is not appropriate for matrix–panel multiplication. Therefore, we developed a new algorithm for matrix–panel multiplication with the jpi loop order described in Sect. 3.6. In this case, i loop can be considered as a candidate for parallelization. This article [20] provides a more comprehensive description of the parallelization analysis.

For our analysis of loop orders, as shown in Fig. 9, we executed routines with different micro-kernels of (m_r, n_r), as listed in Table 2. The maximum performance achieved for each loop order was then recorded.

3.5 Shapes of submatrices

Large matrices can be divided into submatrices of different shapes as shown in Fig. 10, as a result of the execution of the outer three loops.

Effectively utilizing the principle of locality is critical for optimizing the algorithm performance in a hierarchical memory system. When the matrices are partitioned into panel–panel (inner product) shapes, a limited number of columns and

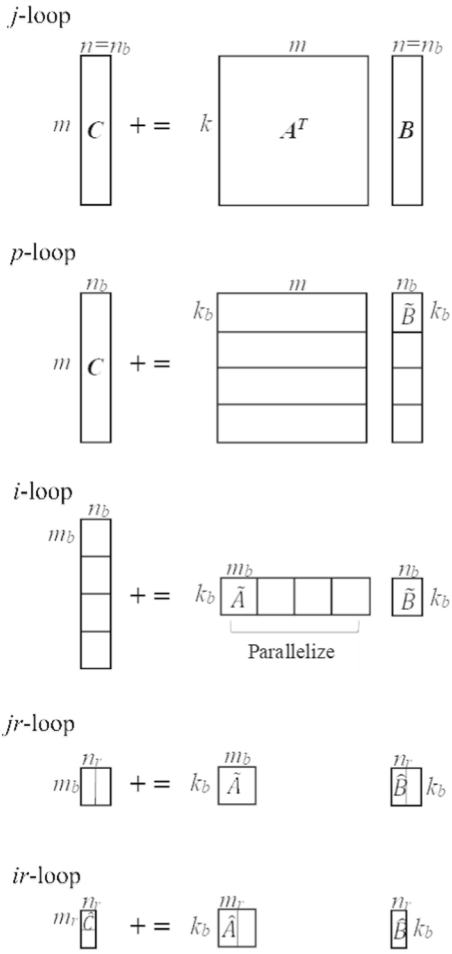
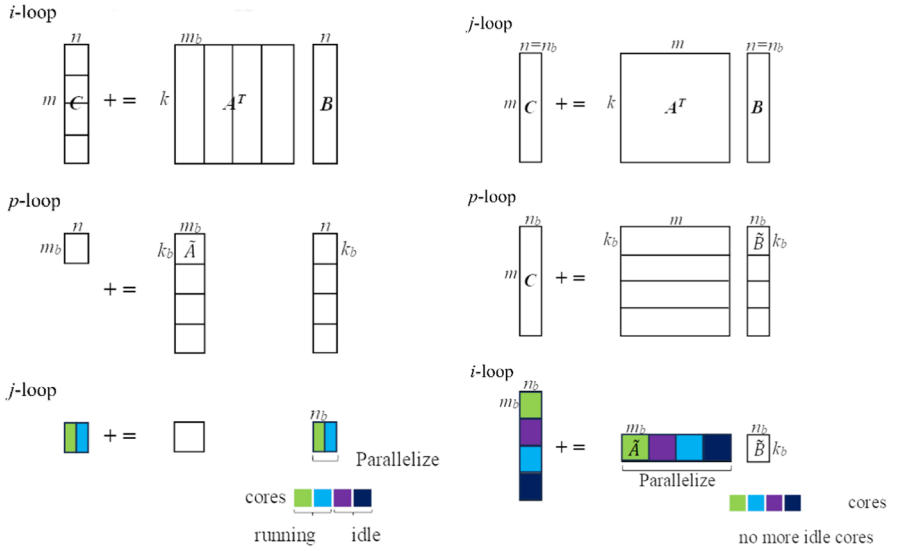


Fig. 11 Illustration of Algorithm 2 for $A^{\mathrm{T}} \cdot B$ matrix–panel Multiplication

rows from matrices A and B are streamed from the memory to the L2 cache. This requires the storage of partial products in memory and subsequent retrieval for the accumulation of the final result in matrix C. Consequently, this approach requires twice the panel–panel (outer product) bandwidth. The outer product eliminates the need for redundant reads of submatrices A or B, depending on the algorithm design. The inner product exhibited suboptimal performance compared to the outer product [6].

Moreover, the panel–panel (outer product) can be subdivided into block-panel or panel-block. This division of submatrices allows parallelization because there is no need for thread synchronization, and each thread processes different elements within the panel. This approach leverages both spatial and temporal localities with well-selected block sizes, and minimizes cache misses. The panel–panel (inner product) can be further divided into block–block. The block–block, effectively exploits the temporal locality as one block is used multiple times. However, it fails to utilize



(a) Algorithm 1: Parallelized j loop (b) Algorithm 2: Parallelized i loop

Fig. 12 Blocked diagram illustrating the $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation, emphasizing the challenge within Algorithm 1 and highlighting the necessity for the development of Algorithm 2

Table 3 Micro-kernels

Matrix operation	Dimensions	Micro-kernel ($m_{\mathrm{r}}, n_{\mathrm{r}}$)
KNL		
$A \cdot B$	$m = n \approx k$	(8, 31)
$A \cdot B^{\mathrm{T}}$	$m \approx n$ when $k = 40$	(16, 15)
$A^{\mathrm{T}} \cdot B$	$m \approx k$ when $n = 40$	(8, 20)
SKL		
$A \cdot B^{\mathrm{T}}$	$m \approx n$ when $k = 40$	(16, 14)
$A^{\mathrm{T}} \cdot B$	$m \approx k$ when $n = 40$	(8, 20)

spatial locality because the other block is stored in a buffer memory after packing, and its potential for parallelization is limited. When the result of matrix C is in block format after the third loop, its performance tends to be inferior compared to utilizing a panel shape for submatrix C. This difference can be observed in Figs. 6 and 12.

3.6 Matrix–panel multiplication

For matrix–panel multiplication, we developed an alternative routine based on Algorithm 2, illustrated in the block diagram in Fig. 11, which effectively enhances the performance. We refer to this revised routine as OUR_DGEMM2.

The illustrations of both algorithms in Fig. 12 visualizes the difference between the two algorithms for $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation. Algorithm 1 was parallelizing the j loop, and it becomes difficult to exploit the parallelism as shown in Fig. 12a, which proves to be unsuitable. On the other hand, Algorithm 2 addressed this difficulty by considering the *jpi* loop order and parallelizing the i loop, as shown in Fig. 12b.

Algorithm 2 Blocked matrix–panel Multiplication

```
1:  for j := 1, ..., n in steps of n_b do
2:      for p := 1, ..., k in steps of k_b do
3:          Pack B(p : p + k_b − 1, j : j + n_b − 1) into B̃;
4:          Parallel for ti number of threads
5:          for i := 1, ..., m in steps of m_b do
6:              Pack A(i : i + m_b − 1, p : p + k_b − 1) into Ã;
7:              for jr := 1, ..., n_b in steps of n_r do
8:                  for ir := 1, ..., m_b in steps of m_r do
9:                      Â := Ã(ir : ir + m_r − 1, :)
10:                     B̂ := B̃(:, jr : jr + n_r − 1)
11:                     Ĉ += Â · B̂
12:                     Update C using Ĉ
13:                 end for
14:             end for
15:         end for
16:     end for
17: end for
```

3.7 Choice of algorithm, micro-kernel, and cache blocking parameters

For the micro-kernel, we experimentally identified the kernel combinations mentioned in Table 3 that showed good performance on the KNL and SKL for our test cases.

For $A \cdot B^{\mathrm{T}}$ panel–panel multiplication, Algorithm 1 is preferable along with micro-kernel $(m_{\mathrm{r}}, n_{\mathrm{r}}) = (16, 15)$ for register blocks. As k is small, the size of k_{b} can be limited by the size of k. We conducted performance experiments to select n_{b} by varying its value, and the best performance was observed when $(m_{\mathrm{r}}, n_{\mathrm{r}}, n_{\mathrm{b}}) = (16, 15, 45)$ for KNL. The size of m_{b} is not crucial for L1 cache blocking on KNL; however, because of the presence of an L3 cache on the SKL system, the choice of m_{b} does not remain free to choose for cache blocking, and we obtained the best performance on SKL when choosing $(m_{\mathrm{r}}, n_{\mathrm{r}}, m_{\mathrm{b}}, n_{\mathrm{b}}) = (16, 14, 1024, 84)$.

For $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation, Algorithm 2 is preferable along with the micro-kernel $(m_{\mathrm{r}}, n_{\mathrm{r}}) = (8, 20)$. The combination of (8, 20) register blocks does not use full vector registers; however, we experimentally observed that this combination performs relatively better for matrix–panel multiplication, as shown in

Fig. 7. We conducted performance experiments and observed that the combination of $(m_r, n_r, m_b, n_b, k_b) = (8, 20, 8, 40, 876)$ and (8, 20, 16, 40, 1000) performed better for KNL and SKL, respectively.

The KNL system has one processor with 68 cores and the SKL system has two sockets, each with 20 cores, for a total of 40 cores. Any reference to 68 threads in this paper should be interpreted as 40 threads for the SKL system. Similarly, 17 threads on KNL were considered 10 threads on SKL.

4 Experiments and results

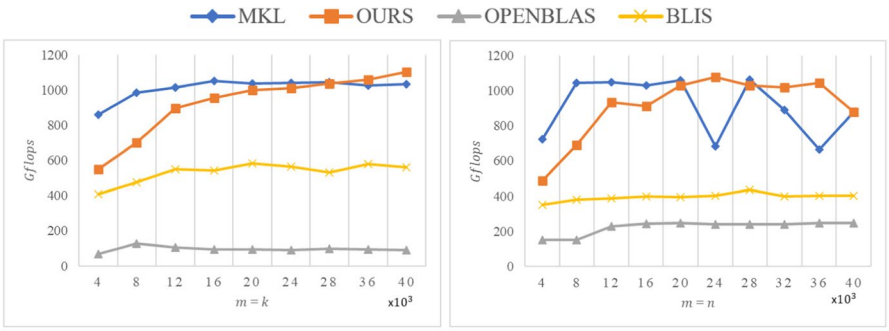
In this section, we discuss the results of the performance comparison. The KISTI Nurion system [21] was used in our experiments. We included the Intel/oneapi_21.2 and impi/oneapi_21.2 modules to use icc, mpicc, compilers, and MKL for comparison. We used the following libraries in our study:

- Memkind v1.14.0 [22]
- ScaLAPACK v2.2.0 [23]
- OPENBLAS v0.3.23 [13]
- BLIS v0.9.0 [24]

4.1 Settings

We used OUR_DGEMM2 to perform matrix–panel multiplication operation $C = A^T \cdot B$ when $m = k$ ranging from (4 to 40) $\times 10^3$ and $n = 40$. We used OUR_DGEMM for panel–panel multiplication operation $C = A \cdot B^T$ when $m = n$ ranging from (4 to 40) $\times 10^3$ and $k = 40$.

The DGEMM operation in ScaLAPACK QR factorization is linked to the MKL DGEMM. We have replaced our OUR_DGEMM2 routine with MKL DGEMM for $A^T \cdot B$ matrix–panel multiplication and OUR_DGEMM routine for



(a) $C = A^T \cdot B$ when $n = 40$ (b) $C = A \cdot B^T$ when $k = 40$

Fig. 13 Performance of **a** $A^T \cdot B$ when $n = 40$ and **b** $A \cdot B^T$ when $k = 40$ on KNL

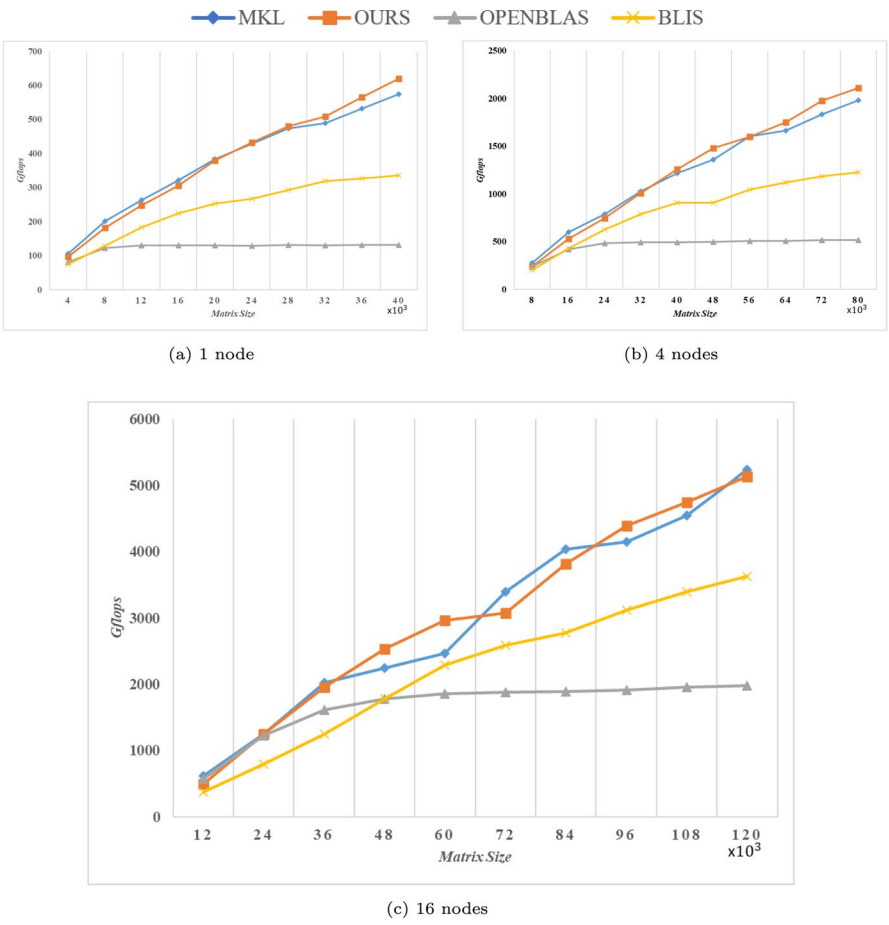
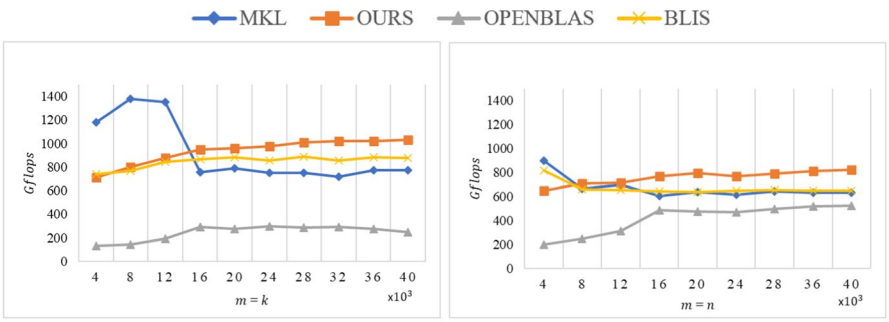


Fig. 14 Performance of QR factorization for $n_b = 40$ on **a** 1 node, **b** 4 nodes, and **c** 16 nodes KNL

$A \cdot B^T$ panel–panel multiplication. In a multinode environment, the performance of QR factorization is optimal for a range of matrix sizes and a block size of 40. Therefore, we examined our routines for block size $n_b = 40$. The sizes of the matrices were denoted in thousands with a scale factor of $\times 10^3$. We used OpenMP for parallelization and set the environment variables as follows:

```
$ export OMP_NUM_THREADS=<number of cores: 68 on KNL and 40 on SKL>
$ export OMP_PROC_BIND=close
$ export OMP_PLACES=cores
```

In our analysis, we adhered to a convention to represent the results. The lines with diamond-shaped (◆) data points represent the performance of the MKL, whereas the lines with square-shaped (■) data points represent the performance of OURS revised routines. We benchmarked our results against two more widely used



(a) $C = A^T \cdot B$ when $n = 40$ (b) $C = A \cdot B^T$ when $k = 40$

Fig. 15 Performance of **a** $A^{\mathrm{T}} \cdot B$ when $n = 40$ and **b** $A \cdot B^{\mathrm{T}}$ when $k = 40$ on SKL

open-source libraries, OPENBLAS and BLIS, for comparative purposes. The lines with cross-shaped (×) data points represent the performance of BLIS, while the lines with triangle-shaped (▲) data points represent the performance of OPENBLAS. We measured the performance of these routines in terms of Gflops. The largest evaluated matrix sizes are 40,000, 80,000, and 120,000 on one, four, and sixteen nodes, respectively.

4.2 Results on Knights Landing (KNL) cluster

This section presents the results conducted on the KNL cluster environment. First, we discuss the results of matrix–panel multiplication, as shown in Fig. 13a, and panel–panel multiplication, as shown in Fig. 13b. Then, we discuss the results of QR factorization.

For $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication, OURS routine achieved higher performance of MKL when $m = k = 36,000$ and 40,000. When $m = k = 40,000$, OURS routine performance is better than MKL, OPENBLAS and BLIS by the factor of 1.06× and 11.90× and 1.96×, respectively. While on a matrix size of $m = k = 20,000$ OURS routine significantly outperformed OPENBLAS and BLIS, showing 10.69× and 1.72× higher performance, respectively. But its performance lags behind of MKL. In every instance of $A \cdot B^{\mathrm{T}}$ panel–panel multiplication, OURS routine outperformed both BLIS and OPENBLAS, while delivering performance comparable to MKL when $m = n = 40,000$, but it surpassed MKL when $m = n = 24,000, 32,000$ and 36,000, respectively.

The performance results of QR factorization are illustrated in Fig. 14a–c for single, four, and sixteen nodes KNL, respectively. For a signle-node configuration, QR factorization with MKL performs the best among all settings for matrix sizes less than 20,000. For matrix sizes greater than that OURS implementation demonstrated better performance. Similar trends are observed for the configurations of the four nodes. MKL remained top-performing on a matrix size of less than 40,000, followed closely by OURS. The performance of QR factorization using the OURS implementation remains comparable to that of MKL for larger matrix sizes. In the case of

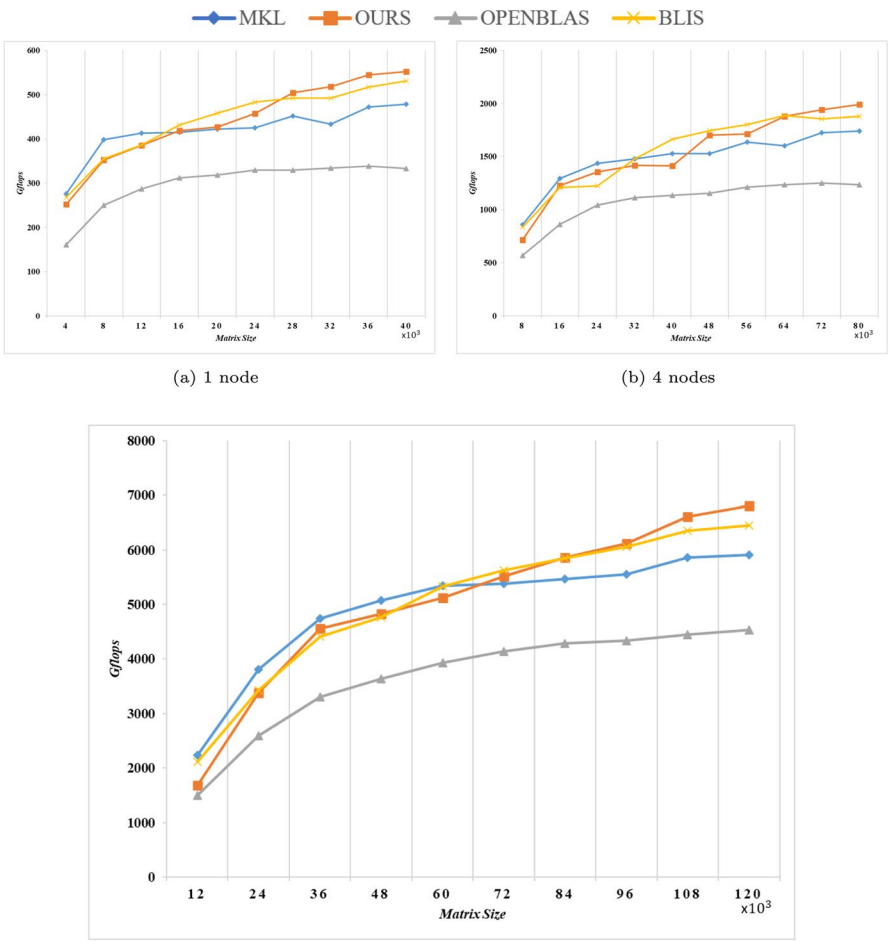


Fig. 16 Performance of QR factorization for $n_b = 40$ on **a** 1 node, **b** 4 nodes, and **c** 16 nodes SKL

16 nodes configurations, the MKL and OURS implementations again demonstrated better performance. These results highlight that the OURS routine exhibits competitive performance compared to the MKL routine. For the largest evaluated matrix sizes, OURS routine performed better than MKL, OPENBLAS, and BLIS. It outperformed MKL, OPENBLAS, and BLIS on a single node by 1.08×, 4.71×, and 1.84×, respectively. On four nodes OURS routine performed better than its competitors by 1.06×, 4.07×, and 1.72×, respectively. On 16 nodes, MKL outperformed OURS routine by 1.02×. However, OURS routine performed better than OPENBLAS and BLIS by 2.58× and 1.41×, respectively.

4.3 Results on Skylake Scalable Processor (SKL) cluster

On SKL the results of $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication operation using our routine are shown in the Fig. 15a. We compared the OURS routine with MKL, OPENBLAS and BLIS. The performance of MKL is higher until the matrix size reaches 14,000, and then degrades for large matrices. The OURS implemented routine demonstrated better performance than MKL, BLIS, and OPENBLAS when dealing with matrices larger than 16,000. OURS routine achieved higher performance of MKL, OPENBLAS, BLIS when $m = k = 40,000$ by the factor of 1.33×, 4.11×, and 1.17×, respectively. For $A \cdot B^{\mathrm{T}}$ panel–panel multiplication performance is presented in Fig. 15b. In this scenario, the OURS routine performed better than MKL, OPENBLAS, and BLIS when the matrix size was greater than 8,000. Specifically, when $m = n = 40,000$ OURS routine achieved higher performance of MKL, OPENBLAS, BLIS by the factor of 1.30×, 1.57×, and 1.26×, respectively. The performances of QR factorization using MKL, OURS, BLIS, and OPENBLAS on single, four, and sixteen nodes configuration on SKL are presented in Fig. 16a–c, respectively. The QR factorization performance of the OURS routine was better than that for other routines when the matrix sizes were greater than 28,000, 64,000, and 96,000 for 1, 4, and 16 nodes, respectively. Overall, our routine shows a significant improvement in the QR factorization performance, particularly for larger matrix sizes. For the largest evaluated matrix sizes, OURS routine emerged as the top performer. It was followed in performance by BLIS, MKL, and OPENBLAS, respectively.

5 Conclusion and future work

This paper presents an optimization of the matrix multiplication routine to enhance the performance of QR factorization. While our previously developed approach showed comparable results for LU factorization, the performance of QR factorization was found to be suboptimal due to limitations in the matrix multiplication routine, particularly for the $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication.

To address these performance issues, we investigated the behavior of the OUR_DGEMM and identified the need for different kernels tailored to the specific shapes of the matrices involved in the QR factorization. This insight led to the optimization of the existing routine for $A \cdot B^{\mathrm{T}}$ panel–panel multiplication and the development of a new routine for $A^{\mathrm{T}} \cdot B$ matrix–panel multiplication. Using our previous implementation, we determined that it is impossible to achieve comparable performance for matrix–panel multiplication when n is less than 220.

Our proposed approach demonstrated a significant improvement in the QR factorization performance of both KNL and SKL multiprocessors in multinode cluster environments. We also conducted a comparative analysis by benchmarking our routines against popular libraries, such as Intel MKL, OPENBLAS, and BLIS. The results showed better performance of our routines on the targeted architectures.

In future work, we intend to investigate the performance issues of matrices with different shapes and develop kernels to optimize their performance. This strategic

endeavor is consistent with our goal of increasing the efficiency of matrix multiplication operations. Furthermore, we intend to widen our investigation by developing an auto-tuner and extending it to a multinode cluster environment. This endeavor will allow us to optimize these kernels and examine their adaptability to the hardware characteristics on which they are running.

Author contributions MR was involved in the conceptualization, methodology, validation, writing—original draft, software, and data curation. EJ contributed to validation, methodology, software, and writing—review and editing. JC assisted in the methodology and writing—review and editing. JC contributed to the supervision.

Funding This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (RS-2023-00321688), and this work was also supported by the Korean National Supercomputing Center (KSC) with supercomputing resources (Nos. KSC-2022-CRE-0202 and TS-2023-RE-0036).

Availability of data and materials Not applicable.

Code availability Please visit GitHub Repository for the source code associated with this paper.

Declarations

Conflict of interest The authors declare no competing interests.

Ethical Approval Not applicable.

References

1. Choi J, Dongarra JJ, Ostrouchoy LS, Petitet AP, Whaley RC, Walker DW (1996) Design and Implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Sci Program. https://doi.org/10.1155/1996/483083
2. Choi J, Dongarra JJ, Pozo R, Walker DW (1992) ScaLAPACK: ascalable linear algebra library for distributed memory concurrent computers. In: The Fourth Symposium on the Frontiers of Massively Parallel Computation. IEEE Computer Society. pp 120–121. https://doi.org/10.1109/fmpc.1992.234898
3. Nassif N, Erhel J, Philippe B (2015) Basic linear algebra subprograms—BLAS. Introduction to computational linear algebra. https://doi.org/10.1201/b18662-7
4. Rizwan M, Jung E, Park Y, Choi J, Kim Y (2022) Optimization of matrix–matrix multiplication algorithm for matrix–panel multiplication on Intel KNL. In: 2022 IEEE/ACS 19th International Conference on Computer Systems and Applications (AICCSA). IEEE. pp 1–7. https://doi.org/10.1109/AICCSA56895.2022.10017947
5. Gunnels JA, Henry GM, van de Geijn RA (2001) A family of high-performance matrix multiplication algorithms. In: Computational Science—ICCS 2001. 2073. pp 51–60. https://doi.org/10.1007/3-540-45545-0_15
6. Goto K, Geijn RAVD (2008) Anatomy of high-performance matrix multiplication. ACM Trans Math Softw 10(1145/1356052):1356053
7. Goto K, Geijn RVD (2008) High-performance implementation of the level-3 BLAS. ACM Trans Math Softw 10(1145/1377603):1377607
8. Lim R, Lee Y, Kim R, Choi J (2018) An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512. Cluster Comput. https://doi.org/10.1007/s10586-018-2810-y
9. Lim R, Lee Y, Kim R, Choi J, Lee M (2019) Auto-tuning GEMM kernels on the Intel KNL and Intel Skylake-SP processors. J Supercomput. https://doi.org/10.1007/s11227-018-2702-1

10. Park Y, Kim R, Nguyen TMT, Choi J (2021) Improving blocked matrix–matrix multiplication routine by utilizing AVX-512 instructions on intel knights landing and Xeon scalable processors. Cluster Comput. https://doi.org/10.1007/s10586-021-03274-8
11. Thi N, Tuyen M (2020). Thesis for the Degree of Master. Improving Performance of LU Factorization Routine on Intel KNL and Xeon Scalable Processors. Dissertation, Soongsil University, Korea
12. Intel (2021) Intel oneAPI Math Kernel Library (oneMKL) Overview. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/overview.html. Accessed 21 Dec 2022
13. Xianyi Z, Qian W, Saar W (2023) OpenBLAS: an optimized BLAS library. https://www.openblas.net. Accessed 15 Apr 2023
14. Zee FGV, van de Geijn RA (2015) BLIS: a framework for rapidly instantiating BLAS functionality. ACM Trans Math Softw. https://doi.org/10.1145/2764454
15. Anderson E, Bai Z, Dongarra J, Greenbaum A, McKenney A, Croz J D, Hammarling S, Demmel J, Bischof C, Sorensen D (1990) LAPACK: A portable linear algebra library for high-performance computers. In: Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing. pp 2–11. https://doi.org/10.1109/superc.1990.129995
16. Demmel J (1991) LAPACK: a portable linear algebra library for high-performance computers. Concurr: Pract Exp. https://doi.org/10.1002/cpe.4330030610
17. Clarke L, Glendinning I, Hempel R (1994) The MPI message passing interface standard. Birkh, Basel. https://doi.org/10.1007/978-3-0348-8534-8_21
18. Anderson E, Dongarra J, Ostrouchov S, Benzoni A, Moulton S, Tourancheau B, Geijn RVD (1991) Basic linear Algebra communication subprograms. In: The Sixth Distributed Memory Computing Conference, 1991. Proceedings. pp 287–290. https://doi.org/10.1109/DMCC.1991.633146
19. Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, Whaley RC (1996) A proposal for a set of parallel basic linear algebra subprograms. In: Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science. Springer, Berlin. vol 1041, pp 107–114. https://doi.org/10.1007/3-540-60902-4_13
20. Smith TM, Geijn RVD, Smelyanskiy M, Hammond JR, Zee FG (2014) Anatomy of high-performance many-threaded matrix multiplication. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp 1049–1059. https://doi.org/10.1109/IPDPS.2014.110
21. KISTI (2018) National Supercomputing Center. https://www.ksc.re.kr/eng/resources/nurion. Accessed 5 Aug 1 2023
22. Cantalupo C, Venkatesan V, Hammond J, Czurlyo K, Hammond SD (2022) Memkind: an extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. https://github.com/memkind/memkind. Accessed 20 Dec 2022
23. Choi J, Dongarra JJ, Ostrouchoy LS, Petitet AP, Whaley RC, Walker DW (2022) ScaLAPACK—Scalable Linear Algebra PACKage. https://www.netlib.org/scalapack. Accessed 2 Dec 2022
24. Zee FGV, van de Geijn RA (2022) BLIS: BLAS-like Library Instantiation Software Framework. https://github.com/flame/blis. Accessed 28 Apr 2023