IATF: An Input-Aware Tuning Framework for Compact BLAS Based on ARMv8 CPUs

Cunyang Wei
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
School of Computer Science and Technology, University of Chinese Academy of Sciences
Beijing, China
weicunyang20g@ict.ac.cn

Haipeng Jia*
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
jiahaipeng@ict.ac.cn

Yunquan Zhang
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zyq@ict.ac.cn

Liusha Xu
Huawei Technologies Co., Ltd.
Beijing, China
xuliusha@huawei.com

Ji Qi
Huawei Technologies Co., Ltd.
Beijing, China
ryan.qiji@huawei.com

ABSTRACT

Recently the mainstream basic linear algebra libraries have delivered high performance on large scale General Matrix Multiplication(GEMM) and Triangular System Solve(TRSM). However, these libraries are still insufficient to provide sustained performance for batch operations on large groups of fixed-size small matrices on specific architectures, which are extensively used in various scientific computing applications. In this paper, we propose IATF, an input-aware tuning framework for optimizing large group of fixed-size small GEMM and TRSM to boost near-optimal performance on ARMv8 architecture. The IATF contains two stages: install-time stage and run-time stage. In the install-time stage, based on SIMD-friendly data layout, we propose computing kernel templates for high-performance GEMM and TRSM, analyze optimal kernel sizes to increase computational instruction ratio, and design kernel optimization strategies to improve kernel execution efficiency. Furthermore, an optimized data packing strategy is also presented for computing kernels to minimize the cost of memory accessing overhead. In the run-time stage, we present an input-aware tuning method to generate an efficient execution plan for large group of fixed-size small GEMM and TRSM, according to the input matrix properties. The experimental results show that IATF could achieve significant performance improvements in GEMM and TRSM compared with other mainstream BLAS libraries.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

KEYWORDS

Compact Batched BLAS, Auto-tune, Code Generation

ACM Reference Format:
Cunyang Wei, Haipeng Jia, Yunquan Zhang, Liusha Xu, and Ji Qi. 2022. IATF: An Input-Aware Tuning Framework for Compact BLAS Based on ARMv8 CPUs. In *51st International Conference on Parallel Processing (ICPP'22), August 29-September 1, 2022, Bordeaux,France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545032

1 INTRODUCTION

BLAS(Basic Linear Algebra Library) is one of the most basic and widely used libraries in scientific computing, machine learning [5] and other applications. Optimization of large-scale dense linear algebra libraries is already a well-studied field [15]. While many linear algebra libraries have been well designed and optimized, like Arm Performance Libraries (ARMPL) [1], Intel oneAPI Math Kernel Library(Intel MKL) [2], and OpenBLAS [3], delivering near processor peak performance. In addition to the widespread use of large-scale dense matrix operations, many applications apply BLAS routines to large group of small matrices, such as PDE based simulations [14], high-order Computational Fluid Dynamics(CFD) [20], machine learning [12], and image processing. Therefore, optimization of large group of small matrices operation is increasingly important.

A good way to deal with large group of small matrices operation is in batch form, which means that it processes many matrices simultaneously. In batch operations with small matrices, traditional optimization methods for large-scale dense matrix operations are inadequate to achieve optimal performance. The following four reasons limit the acceleration efficiency of traditional methods. First, the very small matrix is difficult to fully utilize the width of the SIMD register under traditional methods. Second, edge processing causes many overheads for small matrices. Third, the small matrix can be stored entirely in the L1 cache, making the original tiling methods meaningless. Fourth, traditional methods lack an input-aware tuning framework to generate high-performance execution plans for small matrices of different sizes. Although optimization

[*]*Corresponding authors

ICPP '22, August 29-September 1, 2022, Bordeaux, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9733-9/22/08.
https://doi.org/10.1145/3545008.3545032

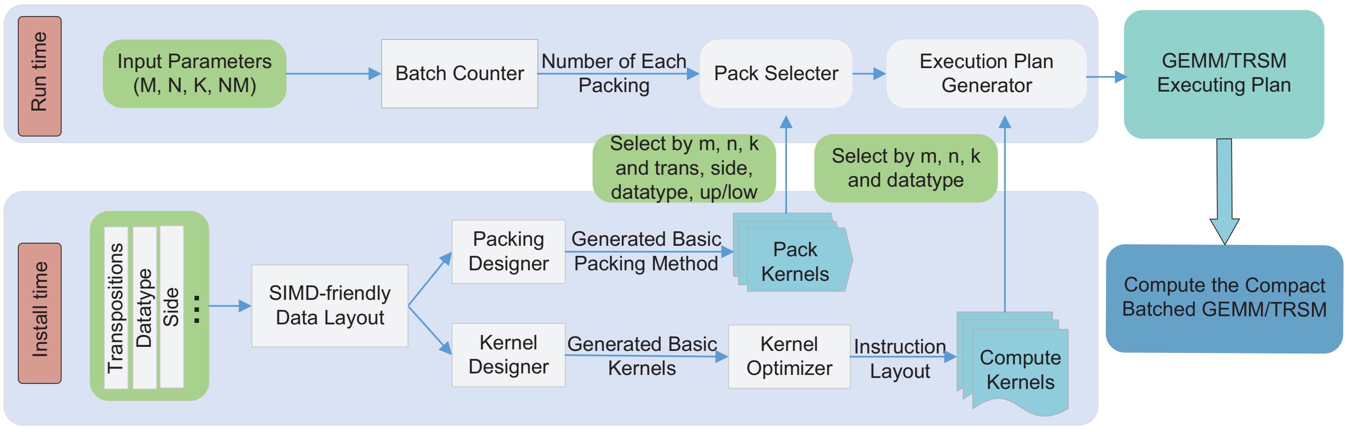


Figure 1: Overview of An Input-aware Tuning Framework.

studies for batch BLAS have demonstrated promising performance, optimization studies for a large group of small fixed-size matrix operations on ARMv8 CPUs are still insufficient.

This paper presents IATF, an input-aware tuning framework to optimize large group of small GEMM and TRSM on ARMv8 CPUs. It contains two stages, the install-time stage and the run-time stage. At the install-time stage, we convert the matrix to the SIMD-friendly data layout [14], which makes full use of the SIMD register width. It generates highly-optimized data packing kernels and computing kernels. For the computing kernel design, we avoid pipeline bubbles through careful instruction schedule and reduce edge processing problems by generating kernels of all possible sizes. We carefully design the data packing kernel so that the memory accesses of the computing kernel are contiguous. At the run-time stage, it chooses an appropriate number of matrices for batch forming each time according to L1 cache size and matrix size, and chooses the optimal data packing kernel and computing kernel according to the input matrix properties(Matrix Size, Transposed/Non-Transposed, Left/Right, Lower/Upper, Unit/NonUnit). Finally, it links the above strategies into execution plans for high-performance processing of large group of fixed-size small matrices GEMM and TRSM on ARMv8 architecture.

We apply the IATF to the Kunpeng 920 CPU [21] based on the ARMv8 architecture. For the sgemm, dgemm, cgemm, and zgemm, compared to looping calls to the OpenBLAS GEMM interface, our library IATF can provide up to 21x, 7x, 12x, and 6x speedups respectively. Compared to the ARMPL batched GEMM, the IATF achieves up to 8x, 4x, 8x, and 5x speedups for the four data types. Furthermore, compared with the LIBXSMM batched GEMM interface, IATF still provides up to 5x speedup for sgemm and up to 2x speedup for dgemm. For TRSM, compared with the ARMPL and OpenBLAS cyclically calling the TRSM interface, IATF is 28x, 12x, 10x, and 5x faster than looping calls to the OpenBLAS TRSM interface for the strsm, dtrsm, ctrsm, and ztrsm respectively. Compared to the loop around ARMPL TRSM calls, our implementation achieves up to 7x, 5x, 4x, and 3x speedups for the four data types. We also use the percentage of processor peak performance as a benchmark, which is still competitive with the Intel MKL compact BLAS interface on the Intel Xeon Gold 6240 CPU. These results demonstrate that our proposed methods are extremely competitive for fixed-size large group of small GEMM and TRSM.

The key contributions of this paper are summarized as follows:

- This paper proposes a high-performance input-aware tuning framework for large group of small matrices operations.
- Based on SIMD-friendly data layout, this paper presents a series of data packing methods and kernel design methods for large group of small matrices compact GEMM and compact TRSM, to dramatically improve performance.
- This paper applies our design methods to a high-performance BLAS library(IATF) for large group of fixed-size small GEMM and TRSM based on the ARMv8 architecture.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the overview of the framework. Section 4 and Section 5 elaborate on the design and implementation details of IATF. Section 6 presents the performance evaluation of our proposed framework. Finally, Section 7 concludes this paper with future work.

2 RELATED WORK

2.1 Traditional GEMM Algotithm

GEMM is used to solve the problem $C = \alpha AB + \beta C$. Where A is $M \times K$, B is $K \times N$, and C is $M \times N$. Traditional GEMM optimization usually uses the algorithm proposed by GOTO [9]. Its optimization direction mainly includes matrix tiling, data packing, and computing kernel design [17]. Efficient matrix multiplication tiles matrix to make full use of multi-level cache and improve data locality. Data packing makes compute kernel memory accesses contiguous. For the computing kernel, optimizing the instruction pipeline arrangement so that the computing instruction hides the memory access delay.

This article references these design ideas, but the implementation differs from them. For small matrices, each matrix can be completely stored in the L1 cache. The traditional tiling method is meaningless. We should consider simultaneously operating on

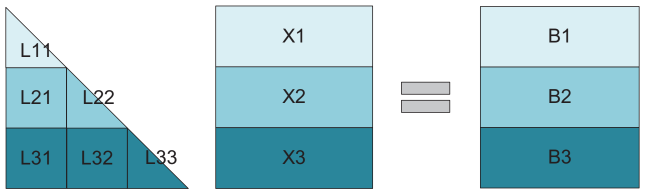


Figure 2: Tiling method of TRSM.

several matrices(not exceeding the L1 cache). For computing kernel design, the main idea is to reduce pipeline bubbles and edge processing. We refer to traditional optimization methods and the optimization methods for small-matrix [7, 8, 11, 18, 19, 22, 23].

2.2 Traditional TRSM Algorithm

TRSM is used to solve problems of the equation $AX = \alpha B$, where A is a triangular $M \times M$ matrix, X and B are $M \times N$ matrices. Or $XA = \alpha B$, where A is a triangular $N \times N$ matrix, X and B are $M \times N$ matrices. Finally, B is overwritten by the solution matrix X. Similar to the implementation of traditional GEMM, the main idea of TRSM is still to divide the matrix into blocks [6, 10], as shown in Figure 2. The large scale TRSM problem under this tiling idea can be calculated using the following formula, which transforms the problem into small triangular blocks solution and rectangular blocks calculation.

$$\begin{cases} X_1 = L_{11}^{-1} B_1 \\ X_2 = L_{22}^{-1}(B_2 - L_{21}X_1) \\ X_3 = L_{33}^{-1}(B_3 - L_{31}X_1 - L_{32}X_2) \end{cases} \tag{1}$$

The triangular part needs special treatment, and the rest can be converted into the GEBP operation in the GEMM function, where B stands for block and P stands for panel. The triangular part only accounts for a small part of the entire TRSM for the large-scale matrix, so the traditional TRSM algorithm usually does not vectorize this part. For data packing, the diagonal elements are converted to their inverses($\frac{1}{a_{ii}}$), so that in the calculation kernel, only vectorized multiplication and addition operations are performed.

In the kernel design, we focus on the calculation of the triangular part, and the rest can refer to the GEMM kernel design. For the triangular part, we design a vectorization solution method based on SIMD-friendly data to improve parallelism. In addition, we redesign the rectangular kernel to optimize performance further.

2.3 Batch BLAS and Small-Sized BLAS

There has been research in the community for GPU to specifically design GEMM kernels and automatic tuning methods for small-scale problems on optimizing batch matrices of variable size and fixed size [4]. For X86 architecture, the SIMD-friendly data layout adopted by Intel MKL [14] can make full use of the width of the vector register in modern processors, and by processing a group of matrices with the same operation, efficient vectorization of small matrices is possible. We evaluate this method and find that Intel MKL compact BLAS has obtained promising progress performance acceleration while using the SIMD-friendly data layout. For the ARM architecture, the community focus on batched BLAS operations, which is mainly for large group of small matrices of different sizes, such as the batch GEMM interface of ARMPL [1] and LIBXSMM [11]. These batch optimizations are parallelized between matrices and do not use SIMD-friendly data layout. In addition, there are some optimizations for small matrices [7, 23] that improve performance by reducing data packing and redesigning the computing kernel. LibShalom [23] is an open-source library for optimizing small and irregular-shaped GEMMs, based on ARMv8 architecture. It improves the performance of the GEMM kernel by improving the shortcomings of existing BLAS libraries, such as the overhead caused by data packing, inefficient boundary processing, and unreasonable parallelism methods.

There are no relevant further optimizations for small matrices of the same size for the ARM architecture. Our work aims to fill a gap in optimization research for compact batched BLAS on ARMv8 CPUs.

3 OVERVIEW OF INPUT-AWARE TUNING FRAMEWORK

This paper proposes an input-aware tuning framework. As shown in Figure 1, it is divided into the install-time stage and the run-time stage to achieve near-optimal performance for large group of fixed-size small GEMM and TRSM.

The install-time stage generates highly-optimized data packing kernels and computing kernels based on the abstracted computing kernel template described in section 4.2. The computing kernel template is called template in the following text for brevity. It contains the following components:

- **Packing Kernel Designer** generates data packing kernels for each data type and matrix properties.
- **Computing Kernel Designer** generates computing kernels for each data type and different kernel sizes based on the abstracted templates.
- **Kernel Optimizer** optimizes the computing kernels to achieve optimal performance by optimizing instruction placement.

The run-time stage chooses the optimal kernels from the install-time stage according to input matrix properties(Matrix Size, Transposed/Non-Transposed, Left/Right, Lower/Upper, Unit/NonUnit), to generate the optimal execution plan for high-performance large group of small GEMM and TRSM. It contains the following components:

- **Batch Counter** determines the number of matrices for each GEMM or TRSM operation based on L1 cache size and matrix size.
- **Pack Selecter** chooses the optimal data packing kernels or not data packing according to matrix properties.
- **Execution Plan Generator** selects the optimal computing kernels and combines the data packing kernels to generate an execution plan.

4 THE DESIGN OF INSTALL-TIME STAGE

In the install-time stage, several highly-optimized data packing kernels and computing kernels are generated according to the typical template described in 4.2, based on the SIMD-friendly data layout. The packing strategy depends on the computing kernel. We design the optimal computing kernel for every possible matrix size and datatype, and design the data packing kernels to match it.

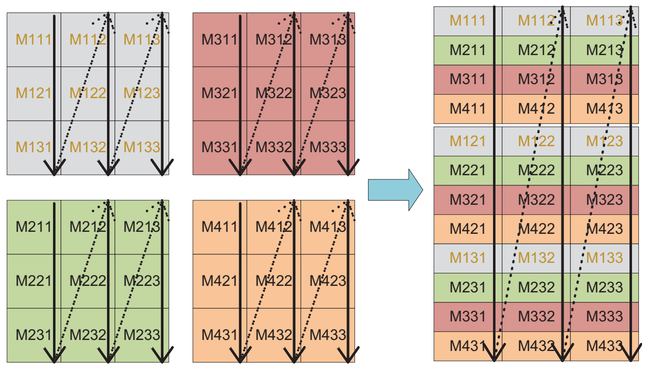


Figure 3: SIMD-friendly Data Layout: 3×3 **matrices on Kunpeng 920.**

Algorithm 1: Simplified compact GEMM

```
Input: A:$M \times K$; B:$K \times N$; C:$M \times N$;
       $NM$:Total number of matrices
Output: $C+ = A \times B$
1 for $v = 0 \to NM - 1$, step = $P$ do
2     for $j = 0 \to N - 1$ do
3         for $i = 0 \to M - 1$ do
4             for $l = 0 \to K - 1$ do
5                 $LOAD \quad V_c \leftarrow C[v \to v + p - 1](i, j)$
6                 $LOAD \quad V_a \leftarrow A[v \to v + p - 1](i, l)$
7                 $LOAD \quad V_b \leftarrow B[v \to v + p - 1](l, j)$
8                 $V_c \leftarrow FMA(V_a, V_b)$
9                 $STORE \quad V_c \to C[v \to v + p - 1](i, j)$
```

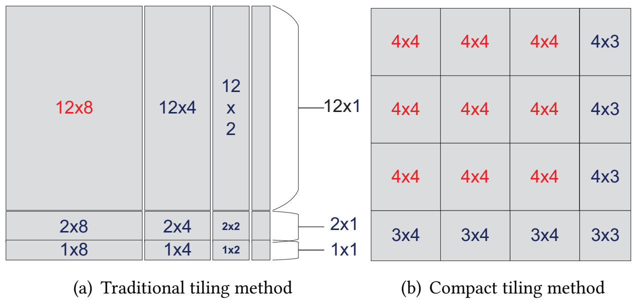


Figure 4: Tiling method of 15×15 **DGEMM.**

4.1 SIMD-friendly Data Layout

We apply the SIMD-friendly data layout [14] on the ARM architecture to make full use of SIMD registers width. As shown in Figure 3, for a group of matrices, SIMD-friendly data layout puts the same location of consecutive P matrices in a contiguous area in memory, with zero padding for the cases where there are not enough P matrices. The value of P is according to the data type and the length of the SIMD register. For example, for the 128-bit SIMD register of Kunpeng 920, single-precision floating-point data operation, P=4, the data just fills the length of the SIMD register.

Algorithm 1 describes a simplified SIMD-friendly data layout based GEMM. NM represents the number of batch matrices. As shown in lines 5-7 of the algorithm, under the SIMD-friendly data layout, we can use vectorized instructions to load the data of P matrices to the SIMD registers. And then calculate via SIMD FMA instruction in line 8. Finally in line 9 store in memory from SIMD registers. It shows that we can use vectorization instructions to process P matrices simultaneously with the SIMD-friendly data layout.

In addition, Figure 4(a) shows the traditional SGEMM block strategy under the ARMv8 architecture when the matrix C is 15×15 in size. A 128-bit register can store 4 single-precision data. Therefore, except for 12×8 blocks and 12×4 blocks, others can not fully utilize SIMD register data lengths. We can see that the cost of edge processing kernels is even higher than the main kernel. As shown in Figure 4(b), under the SIMD-friendly data layout, we can make the kernel smaller than the traditional kernel because the SIMD registers are filled with multiple matrices. For example, the main kernel of Figure 4(b) is 4×4 kernel that actually processes 4×4 blocks of 4 matrices at a time, based on SIMD-friendly data layout, for single-precision floating point. The reduction in kernel size can also reduce the occurrence of edge processing problems.

4.2 Kernel Design

We abstract the typical computing patterns of GEMM/TRSM as templates (Algorithm 2, 3). Based on the SIMD-friendly data layout, we abstract the computing kernel for GEMM, and design vectorized triangular and rectangular computing kernels for TRSM. We then automatically generate the assembly kernels through the templates for the upper-level functions to invoke. Automatic code generation can significantly reduce the workload of our approaches. Our main design idea is to avoid pipeline bubbles. In the following context, we assume that for GEMM, the matrix A is $M \times K$, the matrix B is $K \times N$, and the matrix C is $M \times N$. For TRSM, supposing the problem is $AX = B$, the matrix A is $M \times M$, the matrix B and the matrix X are $M \times N$. m_c, n_c represent the kernel block size.

4.2.1 GEMM kernel design. Algorithm 2 shows our abstracted GEMM template. It contains 6 templates(I, M1, M2, E, SAVE, SUB), and each template computes the matrix matrix multiplication (via $m_c \times n_c$ SIMD FMA/FMUL) of $P \times m_c \times 1$ of A matrices with $P \times 1 \times n_c$ of B matrices to obtain the $P \times m_c \times n_c$ of C matrices, as shown in line 5,10,14,16 and 20. Matrix C is store in the SIMD register $V_{2(m_c+n_c)} - V_{2(m_c+n_c)+m_c \times n_c - 1}$, and the value of P refers to the number of data that can fill the SIMD register. For each input matrix kernel size, it uses these 6 templates(I,M1,M2,E,SAVE,SUB) to implement the "ping-pong" operation, that is to load the data needed by the next template in the current template to avoid pipeline bubbles. Specifically, in the computing kernel, I is the entry of the kernel. As shown in lines 3-4, it loads the data it needs and loads the data required by $M2$. It completes the computing at line 5. $M1$ loads the data required by $M2$ at lines 8-9 and completes the computing at line 10. $M2$ loads the data required by $M1$ at lines 12-13 and completes the computing at line 14. E is the exit of the kernel that only contains calculation instructions, as shown in line 16. When

K is large enough, it will iterate in MI, $M2$. When $K < 4$, it is not enough to iterate $M1$, $M2$, so SUB is used, it only loads the data it needs at line 18-19, and complete the computing at line 20. Finally, use $SAVE$ to store the matrix in memory at lines 22-25.

Algorithm 2: Computing kernel templates of compact GEMM

```
Input: $pA_c : m_c \times K; pB_c : K \times n_c; C_c : m_c \times n_c$;
Output: Computing micro-kernel
1  $/*C_{[m_c \times n_c]} \leftarrow V_{2(m_c+n_c)} - V_{2(m_c+n_c)+m_c \times n_c - 1} */$
2  TEMPLATE_I
3      $LOAD \begin{cases} V_0 - V_{m_c-1} /*For \quad I*/ \\ V_{m_c} - V_{2m_c-1} /*For \quad M2*/ \end{cases} \leftarrow pA_{[2m_c]}$
4      $LOAD \begin{cases} V_{2m_c} - V_{2m_c+n_c-1} /*For \quad I*/ \\ V_{2m_c+n_c} - V_{2m_c+2n_c-1} /*For \quad M2*/ \end{cases} \leftarrow pB_{[2n_c]}$
5      $C_{[m_c \times n_c]} \leftarrow FMUL(V_0 - V_{m_c-1}, V_{2m_c} - V_{2m_c+n_c-1})$
6      $/*Vector - Vector \quad Multiply*/$
7  TEMPLATE_M1
8      $LOAD \quad V_{m_c} - V_{2m_c-1} \leftarrow pA_{[1 \times m_c]}$
9      $LOAD \quad V_{2m_c+n_c} - V_{2(m_c+n_c)-1} \leftarrow pB_{[1 \times n_c]}$
10     $C_{[m_c \times n_c]} \leftarrow FMA(V_0 - V_{m_c-1}, V_{2m_c} - V_{2m_c+n_c-1})$
11 TEMPLATE_M2
12     $LOAD \quad V_0 - V_{m_c-1} \leftarrow pA_{[1 \times m_c]}$
13     $LOAD \quad V_{2m_c} - V_{2m_c+n_c-1} \leftarrow pB_{[1 \times n_c]}$
14     $C_{[m_c \times n_c]} \leftarrow FMA(V_{m_c} - V_{2m_c-1}, V_{2m_c+n_c} - V_{2m_c+2n_c-1})$
15 TEMPLATE_E
16     $C_{[m_c \times n_c]} \leftarrow FMA(V_{m_c} - V_{2m_c-1}, V_{2m_c+n_c} - V_{2m_c+2n_c-1})$
17 TEMPLATE_SUB
18     $LOAD \quad V_0 - V_{m_c-1} \leftarrow pA_{[1 \times m_c]}$
19     $LOAD \quad V_{2m_c} - V_{2m_c+n_c-1} \leftarrow pB_{[1 \times n_c]}$
20     $C_{[m_c \times n_c]} \leftarrow FMA(V_0 - V_{m_c-1}, V_{2m_c} - V_{2m_c+n_c-1})$
21 TEMPLATE_SAVE
22     $LOAD \quad V_0 - V_{2(m_c+n_c)-1} \leftarrow originC_{[m_c \times n_c]}$
23     $V_0 - V_{2(m_c+n_c)-1} \leftarrow FMA(C_{[m_c \times n_c]}, Alpha)$
24     $/*Scalar - Vector \quad Multiply*/$
25     $STORE \quad V_0 - V_{2(m_c+n_c)-1} \longrightarrow originC_{[m_c \times n_c]}$
```

As shown in Algorithm 3, the computing kernel generator calls the above 6 templates to generate the assembly kernel according to the size of the input matrix kernel. It updates the $P \times m_c \times n_c$ of matrix C block by computing the matrix matrix multiplication of $P \times m_c \times K$ size matrix A block and $P \times K \times n_c$ size matrix B block. Especially when the entire matrix C block is calculated, it is stored in memory. Complex matrix matrix multiply kernels are similar. After defining the templates, we need to find the optimal kernel size to utilize the 32 SIMD registers fully.

The main idea of finding the optimal kernel size is to maximize the compute-to-memory-access ratio(CMAR) [16], which is important to effectively hide memory access latency for computational instructions in the micro-kernel. Notice that we need to reserve registers for "ping-pong" operations. Therefore, for DGEMM and SGEMM, the template needs $2m_c$ vector registers to store matrix

Algorithm 3: Computing kernel generator of compact GEMM

```
Input: $pA_c : m_c \times K; pB_c : K \times n_c; C_c : m_c \times n_c$;
Output: Compute kernel($C_c = A_c \times B_c + alpha \times C_c$)
1  if $K < 4$ then
2      if $K == 3$ then
3          $TEMPLATE\_I; TEMPLATE\_E; TEMPLATE\_SU$B;
4      else if $K == 2$ then
5          $TEMPLATE\_I; TEMPLATE\_E$;
6      else
7          $V_{2(m_c+n_c)} - V_{2(m_c+n_c)+m_c \times n_c - 1} \leftarrow Empty$
8          $TEMPLATE\_SUB$;
9  else
10     $TEMPLATE\_I; TEMPLATE\_M2; K- = 2$;
11     while $K > 2$ do
12         $TEMPLATE\_M1; TEMPLATE\_M2; K- = 2$;
13     if $K == 2$ then
14         $TEMPLATE\_M1; TEMPLATE\_E$;
15     else
16         $TEMPLATE\_SUB$;
17 $TEMPLATE\_SAVE$;
```

A, $2n_c$ vector registers to store matrix B, and $m_c \times n_c$ vector registers to store matrix C. We use vector-vector FMA instructions for matrix computation and store the matrix C after iteration in K dimensions. Thus, on average, each micro-kernel needs $(m_c + n_c)$ load operations and $m_c \times n_c$ computation operations. The average CMAR is:

$$CMAR_{real} = \frac{m_c \times n_c}{m_c + n_c} \tag{2}$$

We have to satisfy $2m_c + 2n_c + m_c n_c \leq 32$. It is easy to verify that $CMAR_{real}$ reaches its maximum value when $m_c = 4$ and $n_c = 4$. The total ratio of computational instructions reaches the maximum at this time. For DGEMM and SGEMM, the optimal kernel size is 4×4.

For complex data, we need $4m_c$ vector registers to store matrix A, $4n_c$ vector registers to store matrix B, and $2 \times m_c \times n_c$ vector registers to store matrix C. On average each micro-kernel needs $2(m_c + n_c)$ load operations and $4 \times m_c \times n_c$ computation operations. The average CMAR is:

$$CMAR_{complex} = \frac{4 \times m_c \times n_c}{2(m_c + n_c)} \tag{3}$$

Also, we have to satisfy $4m_c + 4n_c + 2m_c n_c \leq 32$. when $m_c = 3$ and $n_c = 2$ or $m_c = 2$ and $n_c = 3$, the $CMAR_{complex}$ reaches its maximum. For CGEMM and ZGEMM, the optimal kernel size is 3×2 or 2×3.

Based on the main computing kernel, we design kernels for every possible edge case for the problem of large proportion of edge processing in small matrices. These kernels can also be simply generated based on our abstracted templates. For compact GEMM, Table 1 shows all of our highly-optimized kernels in this paper. In this way, as shown in Figure 4, we can use 4×4, 4×3, 3×4, and 3×3 kernels to solve 15×15 compact GEMM, avoiding the generation

Table 1: All generated kernels

	SGEMM DGEMM	CGEMM ZGEMM	STRSM DTRSM	CTRSM ZTRSM
Main	4×4	3×2	4×4	2×2
Edge	$4 \times \{1,2,3\}$ $3 \times \{1,2,3,4\}$ $2 \times \{1,2,3,4\}$ $1 \times \{1,2,3,4\}$	3×1 $2 \times \{1,2\}$ $1 \times \{1,2\}$	3×4 2×4 1×4	1×2

of particularly small blocks, helping us make full use of 32 SIMD registers, and reducing pipeline bubbles.

4.2.2 TRSM kernel design. The TRSM kernel design is more complicated. We divide the TRSM kernel design into two cases. One is that matrix A can be put entirely into the register, and the other is that matrix A needs to be divided into blocks.

For a particularly small-scale TRSM problem, matrix A can all be placed in registers. In this case, it is possible to iterate through the columns of matrix B to reuse matrix A. In this case, $\frac{M(M+1)}{2}$ vector registers are needed to store matrix A. Similar to the GEMM optimization idea, considering the "ping-pong" operation, $2M$ vector registers are needed to store the B, and $2M + \frac{M(M+1)}{2} \leq 32$. So M is up to 5, that is, when $M \leq 5$, all matrix A can load into the SIMD register, and the column of matrix B is calculated in loops. Algorithm 4 shows the triangular kernel of TRSM. Depending on the matrix size, it can generate corresponding high-performance triangular kernels. Lines 1-3 show that it loads all matrix A into registers. Specifically, each row of A is placed in the register set Am_{regs}. The 4-10 lines calculate one column of B at a time. These operations are vectorized. Finally, it stores the result to X at line 10, which will replace matrix B. Similar to GEMM, we apply the "ping-pong" operation for lines 4-10 in Algorithm 4 in our implementation.

Algorithm 4: Simplified triangular kernel of compact TRSM

```
Input: $pA : M \times M, pB : M \times N$
Output: $X$
1  for $m = 0 \to M-1$ do
2      LOAD  $Am_{regs} \leftarrow$ pA[$row\ m$]
3      /*Load row m of $pA$ into register set $Am_{regs}$*/
4  for $l = 0 \to N-1$ do
5      LOAD  $B_{regs} \leftarrow pB[l \times M]$
6      for $i = 0 \to M-1$ do
7          for $j = 0 \to i-1$ do
8              $B_{regs[i]} \leftarrow$ FMS($B_{regs[j]}, Ai_{regs[j]}$)
9          $B_{regs[i]} \leftarrow$ FMUL($B_{regs[i]}, Ai_{regs[i]}$)
10     STORE  $B_{regs} \to$ X[$column\ l$]
```

For the case that the matrix A can not be stored in the register. It can be seen from Formula 1 that the TRSM computing kernel can be divided into two parts: the triangular computing kernel and the rectangular computing kernel. Triangular computing kernels can be generated directly using Algorithm 4. The rectangular computing kernel should be redesigned to achieve optimal performance.

We redesign the rectangular computing kernel instead of calling the GEMM kernel directly, since the GEMM kernel contains unnecessary computational overhead in this case. The rectangular block multiplication of TRSM is a fixed-format GEMM, which is equivalent to $Alpha = -1$ in GEMM. We replace the FMLA instruction with the FMLS instruction to reduce $M \times N$ multiplications.

$$X_i = -L_{ij}B_j + B_i \tag{4}$$

Direct calls to the GEMM kernel need $M \times M \times N + M \times N$ computation instructions. Therefore, the proportion of computing instructions we save is $\frac{M \times N}{M \times M \times N + M \times N}$. For small matrices, it will reduce the amount of computation by a large percentage.

Similar to GEMM, we also generate highly-optimized kernels for all possible edge processing problems for TRSM. Table 1 shows all TRSM rectangular kernels we defined in this paper. And all triangular cases have been generated in the case mentioned above when matrix A can all be placed in registers.

4.3 Kernel Optimization

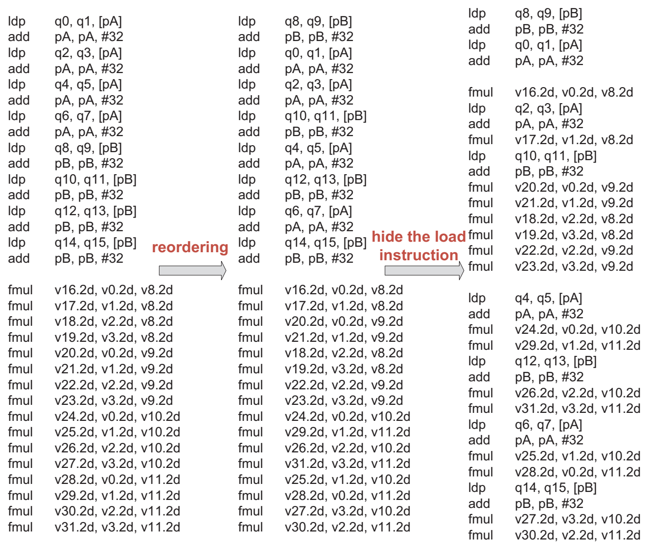
The kernel optimizer obtains high-performance computing kernels by optimizing instruction scheduling for all kernels. In the kernel designer, several kernels of different sizes can be generated according to the templates we abstract. However, a directly generated kernel instruction pipeline is not optimal. We take the GEMM kernel optimization as an example, and the TRSM kernel optimization method is similar.

Our kernel optimizer optimizes kernels to minimize pipeline blocking in instruction scheduling optimization. As shown in Figure 5, the original code shows TEMPLATE_I in a 4×4 DGEMM kernel generated by the kernel generator. The kernel optimizer will first reorder the instructions so that there is a large enough gap between two related instructions to element pipeline block. In the second step, the load instruction is then inserted between the calculation instructions so that the calculation instruction hides load instruction delays. In addition, at the computation phase, matrix A and B are already in the L1 cache, so no prefetching is needed. Comparatively, matrix C is still in the memory, thus we use the PRFM instruction of the ARM architecture to prefetch it at the beginning of the computing kernel to eliminate the risk of cache miss.

4.4 Packing Design

Under the SIMD-friendly data layout, the primary purpose of data packing is to enable the computing kernel to continuously access matrix A and matrix B. Similar to the mainstream BLAS library, we adopt the strategy of N-shaped or Z-shaped data packing for the matrix, as shown in Figure 6.

Compact batched GEMM packs matrix A and matrix B so that the GEMM computing kernel memory access is continuous. Take the NN mode as an example, we pack the matrix A into the N-shape and the matrix B into the Z-shape. Under the SIMD-friendly data layout, the data copied each time is at least the number of data that fills the length of the SIMD vector, so we use the memcpy function to minimize the overhead caused by data packing.

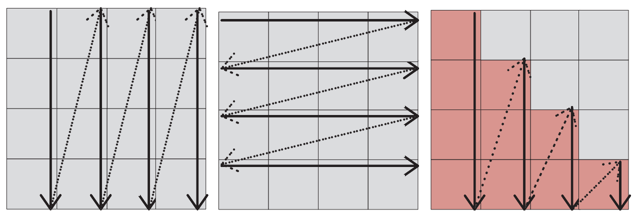


```
ldp     q0, q1, [pA]
add     pA, pA, #32
ldp     q2, q3, [pA]
add     pA, pA, #32
ldp     q4, q5, [pA]
add     pA, pA, #32
ldp     q6, q7, [pA]
add     pA, pA, #32
ldp     q8, q9, [pB]
add     pB, pB, #32
ldp     q10, q11, [pB]
add     pB, pB, #32
ldp     q12, q13, [pB]
add     pB, pB, #32
ldp     q14, q15, [pB]
add     pB, pB, #32

fmul    v16.2d, v0.2d, v8.2d
fmul    v17.2d, v1.2d, v8.2d
fmul    v18.2d, v2.2d, v8.2d
fmul    v19.2d, v3.2d, v8.2d
fmul    v20.2d, v0.2d, v9.2d
fmul    v21.2d, v1.2d, v9.2d
fmul    v22.2d, v2.2d, v9.2d
fmul    v23.2d, v3.2d, v9.2d
fmul    v24.2d, v0.2d, v10.2d
fmul    v25.2d, v1.2d, v10.2d
fmul    v26.2d, v2.2d, v10.2d
fmul    v27.2d, v3.2d, v10.2d
fmul    v28.2d, v0.2d, v11.2d
fmul    v29.2d, v1.2d, v11.2d
fmul    v30.2d, v2.2d, v11.2d
fmul    v31.2d, v3.2d, v11.2d
```

```
ldp     q8, q9, [pB]
add     pB, pB, #32
ldp     q0, q1, [pA]
add     pA, pA, #32
ldp     q2, q3, [pA]
add     pA, pA, #32
ldp     q10, q11, [pB]
add     pB, pB, #32
ldp     q4, q5, [pA]
add     pA, pA, #32
ldp     q12, q13, [pB]
add     pB, pB, #32
ldp     q6, q7, [pA]
add     pA, pA, #32
ldp     q14, q15, [pB]
add     pB, pB, #32

fmul    v16.2d, v0.2d, v8.2d
fmul    v17.2d, v1.2d, v8.2d
fmul    v20.2d, v0.2d, v9.2d
fmul    v21.2d, v1.2d, v9.2d
fmul    v18.2d, v2.2d, v8.2d
fmul    v19.2d, v3.2d, v8.2d
fmul    v22.2d, v2.2d, v9.2d
fmul    v23.2d, v3.2d, v9.2d
fmul    v24.2d, v0.2d, v10.2d
fmul    v29.2d, v1.2d, v11.2d
fmul    v26.2d, v2.2d, v10.2d
fmul    v31.2d, v3.2d, v11.2d
fmul    v25.2d, v1.2d, v10.2d
fmul    v28.2d, v0.2d, v11.2d
fmul    v27.2d, v3.2d, v10.2d
fmul    v30.2d, v2.2d, v11.2d
```

```
ldp     q8, q9, [pB]
add     pB, pB, #32
ldp     q0, q1, [pA]
add     pA, pA, #32

fmul    v16.2d, v0.2d, v8.2d
ldp     q2, q3, [pA]
add     pA, pA, #32
fmul    v17.2d, v1.2d, v8.2d
ldp     q10, q11, [pB]
add     pB, pB, #32
fmul    v20.2d, v0.2d, v9.2d
fmul    v21.2d, v1.2d, v9.2d
fmul    v18.2d, v2.2d, v8.2d
fmul    v19.2d, v3.2d, v8.2d
fmul    v22.2d, v2.2d, v9.2d
fmul    v23.2d, v3.2d, v9.2d

ldp     q4, q5, [pA]
add     pA, pA, #32
fmul    v24.2d, v0.2d, v10.2d
fmul    v29.2d, v1.2d, v11.2d
ldp     q12, q13, [pB]
add     pB, pB, #32
fmul    v26.2d, v2.2d, v10.2d
fmul    v31.2d, v3.2d, v11.2d
ldp     q6, q7, [pA]
add     pA, pA, #32
fmul    v25.2d, v1.2d, v10.2d
fmul    v28.2d, v0.2d, v11.2d
ldp     q14, q15, [pB]
add     pB, pB, #32
fmul    v27.2d, v3.2d, v10.2d
fmul    v30.2d, v2.2d, v11.2d
```

Figure 5: The kernel optimizer optimizes the DGEMM 4×4 TEMPLATE_I generated by the computing kernel generator into a better instruction placement.



(a) N-shape (b) Z-shape (c) Triangular matrix data packing

Figure 6: Data packing strategies.

Compact batched TRSM packs the triangular part of matrix A and the entire matrix B so that the TRSM computing kernel has continuous access to matrices A and B. Take the case of left, lower, no-transpose, non-unit diagonal as an example. Note that, the calculation of each row of matrix X depends on the results of all the rows above its column. So matrix A needs an N-shaped data pack, so that when the Nth row of the matrix A is read, the operations involved in the first N rows have been calculated. In addition, to store more data in the L1 cache, we only pack the data for the triangular part of the matrix A.

Note that the diagonal elements use division and the other use multiplication. Considering the long delay of division instructions under the ARM architecture, when packing matrix A, the diagonal part is stored as its reciprocal($\frac{1}{l_{ii}}$).

For matrix B, the calculation formula for each column is exactly the same. Therefore we consider N-shaped copy of B, which means that it iterates on the column, and A can be reused.

No-packing strategy is applied in cases where the expensive data packing overhead of small matrices can be saved, and the performance improvement of this strategy for small matrix operations is significant. Specifically, the purpose of data packing in this paper is to enable the computing kernel to access matrices continuously, so it is unnecessary to pack matrices that can be accessed sequentially even if they are not packed. For example, for GEMM under NN mode, when M does not exceed the size of the computing kernel design, matrix A is accessed rows by rows. For TRSM under LNLN mode, when M does not exceed the size of the computing kernel design, the packing of matrix B can be skipped.

The pack selector in the run-time stage of our auto-tuning framework looks for opportunities to choose a no-packing strategy as much as possible, to reduce the potentially expensive cost of data packing.

5 THE DESIGN OF RUN-TIME STAGE

The run-time stage is to select the optimal computing kernel and data packaging kernel generated at the install-time stage according to the input parameters, to form an optimal execution plan.

5.1 Batch Counter

The batch counter decides the number of the matrix for computing each time at the run-time stage according to matrix size(M, N, K) , datatype, and the L1 cache size. The batch processing problem we study is mainly aimed at the fact that it can be completely stored in the L1 cache, and the data between the matrices are not related. The batch counter needs to ensure the matrix is always in the L1 cache throughout the computation. For GEMM, pack matrices A and B up to the size of L1 cache at a time and reserve space for matrix C. For TRSM, pack matrices B and the triangle part of matrices A up to the size of L1 cache at a time.

5.2 Pack Selecter

At this stage, it chooses the optimal data packing kernel to match the computing kernel according to the data type and matrix properties. It matches appropriate data packing kernels for different modes to pack matrices into the same order, so that only one computational kernel is needed to handle all modes. Computation kernels are specially designed for matrices of different sizes, so it further selects data packing kernels based on the matrix size. In order to minimize memory access overhead, it only chooses data packing when the data cannot be continuously accessed in the computing core, otherwise it will choose no packing strategy.

5.3 Execution Plan Generator

The execution plan generator chooses the optimal computing kernel, and combines the data packing kernel provided by the data packing selector and the cache strategy of the batch counter to form the execution plan. In terms of computing kernel selection, it selects the most matching computing kernel according to the input scale, which is the optimal choice for instruction arrangement under this scale. Furthermore, this choice is strictly matched to the data packing kernel. Finally, it generates a high-performance execution plan, which means a set of command queues, for large groups of small GEMM and TRSM. By executing these command

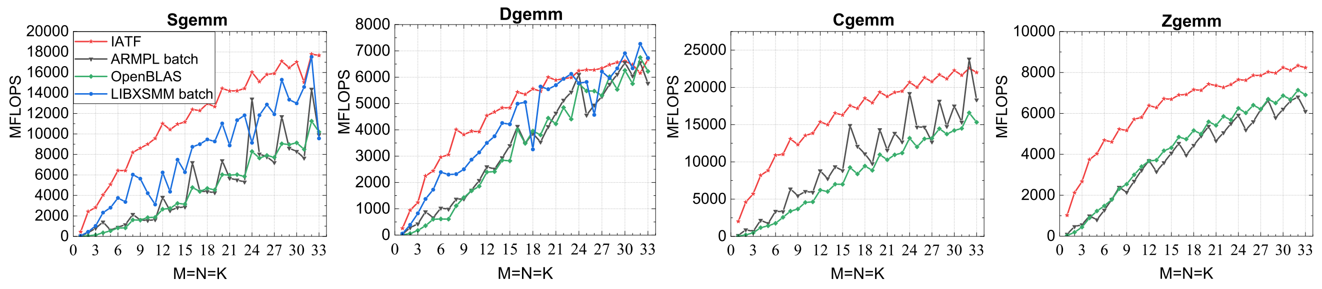


Figure 7: Performance of the IATF compact GEMM compared with ARMPL, LIBXSMM, and OpenBLAS under the NN mode .

Table 2: Experimental environments

CPU	Kunpeng 920	Intel Xeon Gold 6240
Peak perf. (FP64)	10.4GFLOPS	83.2GFLOPS
Peak perf. (FP32)	41.6GFLOPS	166.4GFLOPS
Arch.	ARMv8.2	Cascade Lake
Freq.	2.6GHz	2.6GHz
SIMD	128 bits	512 bits
L1D cache	64KB	32KB
L2 cache	512KB	1024KB
Compiler	GCC7.5	GCC7.5
Intel oneAPI MKL	-	2022.0.2
ARMPL	22.0	-
LIBXSMM	1.17	-
OpenBLAS	0.3.13	-

queues, we are able to compute compact batched GEMM/TRSM with high performance.

For large groups of matrix batch operations, the run-time stage overhead is not significant, since it only generates this execution plan at the beginning. Therefore these overheads are negligible when apportioned to each matrix.

6 EXPERIMENTS EVALUATION

We evaluate the IATF on the kunpeng 920 processor based on ARMv8 architecture. We also selected the Intel Xeon Gold 6240 CPU to test and evaluate the MKL compact BLAS. Our experiments found that the Intel Xeon Gold 6240 CPU was instable in performance when overclocked. Therefore, we adjusted the frequency to the processor's base frequency which is 2.6GHz. This adjustment has little effect on the experimental results, because we compare with Intel as a percentage of peak performance. Table 2 shows the specifications of the two processors. The performance tests used in this article are compiled with the GCC7.5 compiler with the "-O3 -g" option.

We compare the IATF with three BLAS libraries optimized for the ARMv8 architecture, among which OpenBLAS is the most widely used open-source BLAS library in the industry. ARMPL is the official performance library for ARM architecture. LIBXSMM is optimized for small matrices. Furthermore, ARMPL and LIBXSMM also support batch GEMM interface. We also evaluate the performance of the Intel MKL library, which is Intel's official performance library, which can provide excellent performance on Intel platform, and support compact BLAS interface. We run each kernel 100 times and take the geometric mean as the final result.

To fully demonstrate each kernel, we evaluate the performance of square matrices of sizes $1-33$ for each function. The batch size is 16384. We refer to the general testing scheme [13] to initialize the matrix by filling it with random floating-point numbers (0 to 1). We evaluate the performance under different datatype and different modes.

6.1 Compact GEMM

Figure 7 demonstrates our strong performance of the compact batched GEMM for real/complex numbers in single-precision or double-precision under the NN mode. We compared the IATF with the ARMPL batched GEMM(ARMPL batch), LIBXSMM batched GEMM, and loop around OpenBLAS library calls for GEMM. As shown in Figure 7, when the size of sgemm is less than 30 and the size of dgemm is less than 18, we can achieve great performance improvement compared with other libraries. In complex mode, the speedup is even more pronounced. For the sgemm, dgemm, cgemm, and zgemm, compared to looping calls to the OpenBLAS GEMM interface, the IATF can provide up to 21x, 7x, 12x, and 6x speedups respectively. Compared to the ARMPL batched GEMM, the IATF achieves up to 8x, 4x, 8x, and 5x speedups for the four data types. LIBXSMM is optimized for small matrix multiplication, but it does not support a complex interface. At some scales, it shows advantages, but at particularly small scales, the IATF still provides up to 5x speedup for sgemm and up to 2x speedup for dgemm.

Figure 8 shows the performance comparison in NN, NT, TN, and TT modes for sgemm, dgemm, cgemm, and zgemm. It demonstrates that we can provide excellent and stable performances in every mode. In addition, we also demonstrate significant performance advantages under NT, TN, and TT modes of sgemm, dgemm, cgemm, and zgemm, compared to ARMPL, LIBXSMM, and OpenBLAS.

6.2 Compact TRSM

Figure 9 demonstrates our very impressive performance of the TRSM for real/complex numbers in single-precision or double-precision under LNLN(Left, Non-Transpose, Lower, NonUnit) mode. We compared the IATF with the loop around ARMPL library calls for TRSM and the loop around OpenBLAS TRSM calls. In TRSM

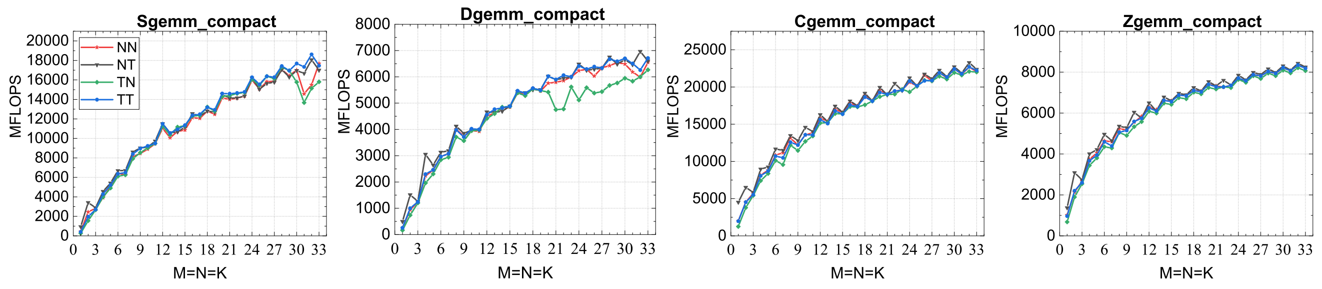


Figure 8: Performance of Compact GEMM under the NN, NT, TN, and TT modes.

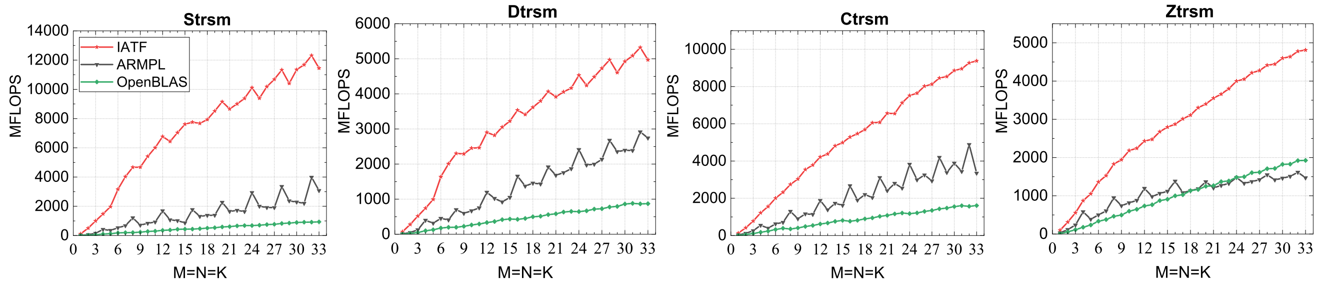


Figure 9: Performance of the IATF compact TRSM compared with ARMPL, and OpenBLAS under the LNLN mode.

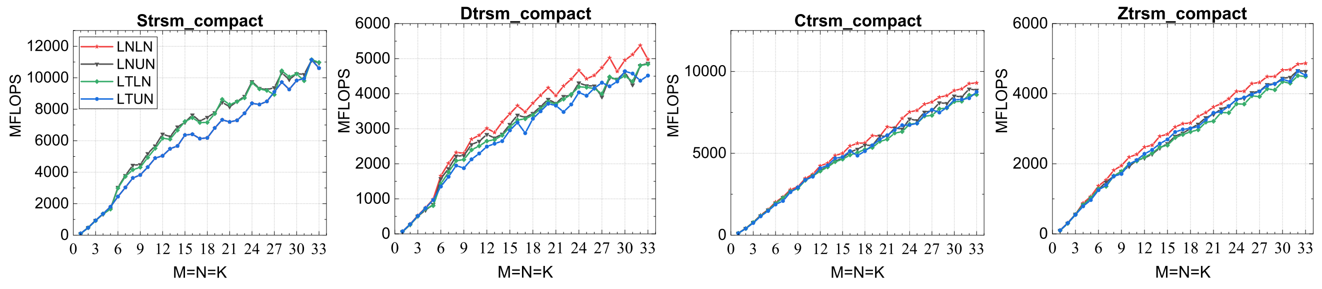


Figure 10: Performance of Compact TRSM under the LNLN, LNUN, LTLN, and LTUN modes.

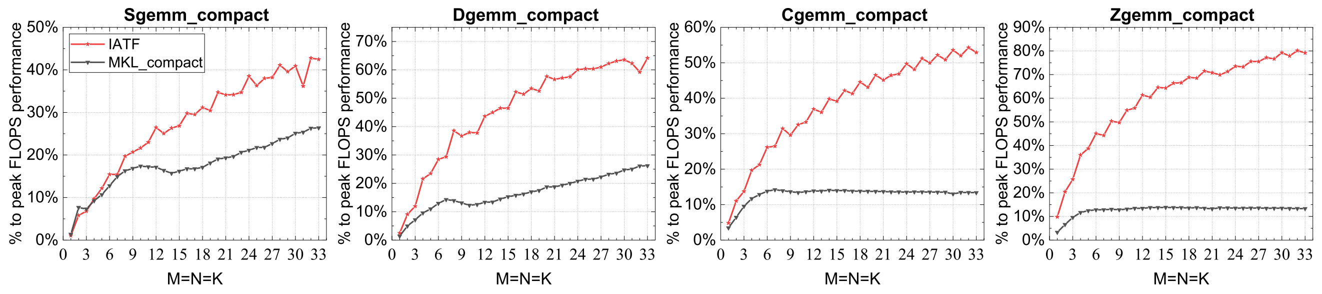


Figure 11: Performance of the IATF compact GEMM compared with the Intel MKL compact GEMM mode was evaluated using the percentage of peak processor performance as a benchmark under the NN.

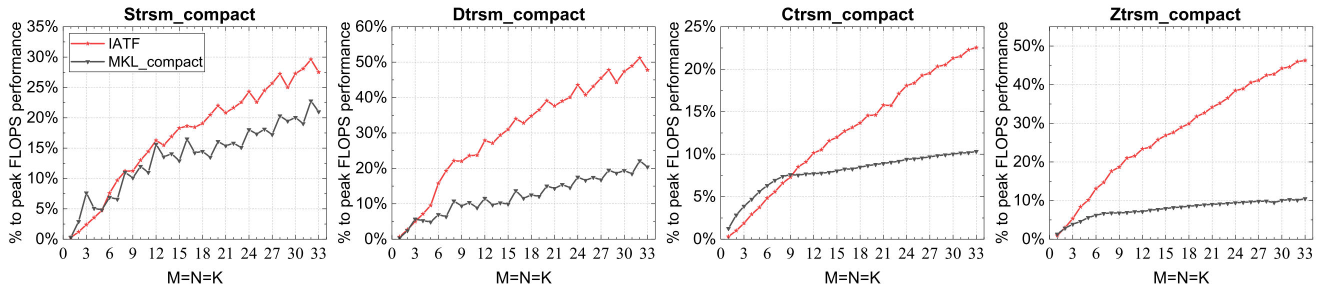


Figure 12: Performance of the IATF compact TRSM compared with the Intel MKL compact TRSM mode was evaluated using the percentage of peak processor performance as a benchmark under the LNLN.

case, we can see that the IATF achieves extremely large improvements for all sizes and all data types. The IATF is 28x, 12x, 10x, and 5x faster than looping calls to the OpenBLAS TRSM interface for the strsm, dtrsm, ctrsm, and ztrsm respectively. Compared to the loop around ARMPL TRSM calls, our implementation achieves up to 7x, 5x, 4x, and 3x speedups for the four data types. In this case, we do not compare with LIBXSMM as the TRSM is not available in the LIBXSMM library. Figure 10 shows that the IATF achieves nearly consistent high performance with the left side mode. The performance in other modes compared with OpenBLAS and ARMPL also shows that the IATF has significantly high performance.

6.3 Evaluation with Intel MKL

We use the percentage of processor peak performance as a benchmark to evaluate Intel MKL compact GEMM/TRSM for reference. As shown in Figure 11, We achieve significant advantages on double-precision floating-point numbers, both for real and complex. Note that Kunpeng 920 CPU can only issue one memory access instruction and one calculation instruction at the same time, or simultaneously issue two calculation instructions for single-precision floating-point numbers. In single precision, our advantage is not apparent. However, when the scale is larger, the computational instructions take up a higher percentage, and our optimization approach is reflected.

As shown in Figure 12, We show considerable advantages in double-precision floating-point numbers, both for real and complex. On the one hand, the division instruction cycle of the ARM architecture is more than that of the X86 architecture. On the other hand, as mentioned above, the dual-issue problem of the Kunpeng 920 platform we tested, combined to cause the poor performance of TRSM in single precision. However, we still get an advantage for the cgemm when the size is greater than 9 and for the sgemm when the size is greater than 12. This fully demonstrates the effectiveness of our kernel design and optimization.

7 CONCLUSION

This paper presents IATF, an input-aware tuning framework for compact batched GEMM/TRSM. It consists of two parts, the install-time stage and the run-time stage. The install-time stage generates and optimizes the computing kernel and data packing kernel based on SIMD-friendly data layout for ARMv8 architecture. The run-time stage chooses optimal kernels to generate an optimal execution plan according to the input matrix properties. Compared with other mainstream BLAS libraries, our implementation shows performance advantages in both GEMM and TRSM.

In the future, we will focus on the kernel design and optimization of other BLAS functions under the SIMD-friendly data layout. In addition, we would investigate and extend our approach to multi-core CPU and GPU in the future.

ACKNOWLEDGMENTS

The authors would like to thank all the reviewers for their insightful and valuable comments and suggestions. This work is supported by National Natural Science Foundation of China under Grant No. 61972376, No. 62072431, No. 62032023.

REFERENCES

[1] [n.d.]. *ARM PERFORMANCE LIBRARIES.* https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux/arm-performance-libraries
[2] [n.d.]. *Intel oneAPI Math Kernel Library.* https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
[3] [n.d.]. *OpenBLAS:An optimized BLAS library.* http://www.openblas.net/
[4] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, design, and autotuning of batched GEMM for GPUs. In *International Conference on High Performance Computing*. Springer, 21–38.
[5] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.
[6] Jean-Guillaume Dumas, Clément Pernet, and Jean-Louis Roch. 2006. Adaptive triangular system solving. In *Challenges in Symbolic Computation Software*. 770.
[7] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic linear algebra subroutines for embedded optimization. *ACM Transactions on Mathematical Software (TOMS)* 44, 4 (2018), 1–30.
[8] Gianluca Frison, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2020. The BLAS API of BLASFEO: Optimizing performance for small matrices. *ACM Transactions on Mathematical Software (TOMS)* 46, 2 (2020), 1–36.
[9] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 1–25.
[10] Kazushige Goto and Robert Van De Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)* 35, 1 (2008), 1–14.
[11] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.
[12] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. 2018. Matrix capsules with EM routing. In *International conference on learning representations*.
[13] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming. 109–123.

[14] Kyungjoo Kim, Timothy B Costa, Mehmet Deveci, Andrew M Bradley, Simon D Hammond, Murat E Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. 2017. Designing vector-friendly compact BLAS and LAPACK kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[15] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[16] Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt, Minwen Deng, Xiaoning Wang, Weiguo Liu, Yu Qiao, and Shengzhong Feng. 2019. FeatherCNN: Fast inference computation with TensorGEMM on ARM architectures. *IEEE Transactions on Parallel and Distributed Systems* 31, 3 (2019), 580–594.

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

[18] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–33.

[19] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[20] Bartosz D Wozniak, Freddie D Witherden, Francis P Russell, Peter E Vincent, and Paul HJ Kelly. 2016. GiMMiK—Generating bespoke matrix multiplication kernels for accelerators: Application to high-order Computational Fluid Dynamics. *Computer Physics Communications* 202 (2016), 12–22.

[21] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* 41, 5 (2021), 67–75.

[22] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th international conference on parallel and distributed systems*. IEEE, 684–691.

[23] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2021. LIBSHALOM: optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.