IrGEMM: An Input-Aware Tuning Framework for Irregular GEMM on ARM and X86 CPUs

Cunyang Wei, Haipeng Jia, Yunquan Zhang, *Senior Member, IEEE*, Jianyu Yao, Chendi Li , Wenxuan Cao

Abstract—The matrix multiplication algorithm is a fundamental numerical technique in linear algebra and plays a crucial role in many scientific computing applications. Despite the high performance of mainstream basic linear algebra libraries for large-scale dense matrix multiplications, they exhibit poor performance when applied to matrix multiplication with irregular input. This paper proposes an input-aware tuning framework that accounts for application scenarios and computer architectures to provide high-performance irregular matrix multiplication on ARMv8 and X86 CPUs. The framework comprises two stages: the install-time stage and the run-time stage. The install-time stage utilizes our proposed computational template to generate high-performance kernels for general data layout and SIMD-friendly data layout. The run-time stage utilizes a tiling algorithm suitable for irregular GEMM to select the optimal kernel and link as an execution plan. Additionally, load-balanced multi-threaded optimization algorithms are defined to exploit the multi-threading capability of modern processors. Experiments demonstrate that the proposed IrGEMM framework can achieve significant performance improvements for irregular GEMM on both ARMv8 and X86 CPUs compared to other mainstream BLAS libraries.

Index Terms—batch GEMM, compact GEMM, TSMM, code generation.

1 Introduction

Recently the mainstream basic linear algebra libraries (BLAS) have delivered near-peak high performance on large-scale General Matrix Multiplication (GEMM). The definition of GEMM is given in Equation 1:

$$\alpha AB + \beta C = C \tag{1}$$

where A, B, and C are $M \times K$, $K \times N$ and $M \times N$ matrices, respectively. Traditional approaches have proven inadequate in achieving optimal performance for a number of modern applications, such as metabolic networks [1], PDE-based simulations [2], tensor shrinkage for finite element simulations [3], and image processing [4]. The high computational demands of these applications necessitate efficient methods for irregular matrix multiplication. In this regard, We summarize the irregular matrix multiplication into three types including compact GEMM, batch GEMM, and TSMM (Tall-and-Skinny Matrix-Matrix Multiplication). A typical example of batch GEMM and compact GEMM is given in Equation 2:

$$\alpha_i A_i B_i + \beta_i C_i = C_i, i = 0 \rightarrow L \tag{2}$$

where L is large, but A_i, B_i, and C_i are small matrices that $\sqrt[3]{MNK} \leq 80$. Compact GEMM, which based on SIMD-friendly data layout, refers to the case where all matrices are the same size, and Batch GEMM refers to the case of dealing with several groups of matrices in which the matrices within the group have the same properties while each group of matrices has a different properties. Besides, TSMM refers to the case where one of the input matrices (either A or B) is a tall-and-skinny matrix (i.e., one dimension is significantly smaller than the other).

Many previous studies have delivered near-peak high performance on dense GEMM by using the three-step process: tiling, packing, and computing. The tiling phase divides the matrix into smaller blocks based on computer architecture characteristics, such as TLB and cache size, in order to maximize cache locality. The packing phase then arranges the matrix data into small blocks in linear buffers, allowing the computational kernel to access memory continuously and reducing memory access latency. Additionally, different transpose patterns can be achieved simply by adjusting the packing order, rather than rewriting the kernel, which reduces the workload. Data packing is especially important for large-scale dense matrices as it can significantly reduce cache miss and TLB miss overhead. The computing step uses high-performance kernels with boundary processing to perform matrix multiplication.

However, the traditional method is not effective in optimizing irregular matrix multiplication. In terms of the commonality of irregular matrix multiplication, there are several main reasons. First, tiling based on L2 Cache size is not applicable to small-scale matrices, since small matrices can all be placed entirely in the L2 cache. Besides, the traditional tiling approach will introduce many boundaries on irregular GEMM problems and the overhead from boundary handling is not negligible. Second, the data packing overhead accounts for a large proportion of the small-scale matrix multiplication problem, which makes data packing no longer advantageous. On the contrary, it brings more performance loss in terms of access storage overhead. Thirdly, traditional methods can often attain remarkable perfor-

- *C. Wei, J. Yao, C. Li, and W. Cao is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China, the University of Chinese Academy of Sciences, Beijing 100049, China. E-mail: weicunyang20g@ict.ac.cn, {yaojianyu89, lichendi.cs}@gmail.com, caowenxuan22@mails.ucas.ac.cn.*
- *H. Jia and Y. Zhang are with the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China. E-mail: {jiahaipeng, zyq}@ict.ac.cn.*

Manuscript received April 19, 2005; revised August 26, 2015. (Corresponding author: Haipeng Jia.)

mance on large-scale dense matrix multiplication problems by utilizing a handful of meticulously optimized master kernels, since the majority of computations can be executed via these kernels. Conversely, in the case of irregular matrix multiplication, the bounds of computational scale occupy a substantial portion of the overall computational scale, necessitating a multitude of kernels with various sizes to tackle the possible boundary cases and attain high performance across different scales that irregular GEMMs may encounter. Forth, traditional methods lack an input-aware tuning framework to generate high-performance execution plans for irregular GEMM of different sizes.

Additionally, for each specific irregular GEMM type, there are additional problems that limit the performance of irregular matrix multiplication. For batch GEMM, different groups contain different size of matrices, which makes load-balanced task scheduling necessary to exploiting the multi-core performance of modern processors. For compact GEMM, Traditional approach lacks a data layout that can fully utilize the width of the SIMD registers. For TSMM, matrix data reuse is often required in deep learning which is an important application scenario of TSMM. But conventional GEMM implementations cannot reuse matrices because the packing operation and computing operation are coupled.

To summarize, the efficacy of the implementation of irregular GEMM is not only limited by the computer architecture but also limited its diverse application scenarios. This paper presents IrGEMM, an input-aware tuning framework to optimize irregular GEMM on ARMv8 and X86 CPUs. It consists of two stages, the install-time stage and the run-time stage.

In the install-time phase, we propose a template based code generation method, which to the best of our knowledge is the first comprehensive code generation method for irregular matrix multiplication. We analyze the GEMM computational characteristics for each irregular GEMM type and carefully analyze the benefits and overheads of data packaging. We extracted these typical computational patterns as computational templates. These computation templates were then used by the kernel generator to generate kernels containing "ping-pong" operations. Subsequently, the instruction mapping rules and the assembly template optimizer convert the generated kernels into high-performance assembly GEMM kernels for different architectures.

In the run-time stage, We propose an input-aware tiling algorithm to minimize memory accesses in irregular matrix multiplication and to prevent the generation of very small blocks. Based on this tiling algorithm, we selects the optimal kernels and combines them into an execution plan, which is saved in the form of a command queue. For batch GEMM, we also propose a load-balanced multithreaded optimization framework which divides the large matrix group into smaller task blocks and achieves optimal multithreaded performance through dynamic mapping between threads and command queues.

In this study, we demonstrate the effectiveness of our proposed irregular GEMM optimization techniques on the ARMv8 and X86 AVX512 architectures. The results of our experiments indicate that IrGEMM outperforms existing state-of-the-art approaches, including LIBXSMM [5], ARM Performance Library (ARMPL) [6], Intel OneAPI Math Kernel Library (MKL) [7], BLIS [8], and OpenBLAS [9].

The key contributions and innovations of this paper are summarized as follows:

- We present a performance tuning framework that considers both the characteristics of the target application scenarios and the features of the computer architecture. The framework also includes a load-balanced multithreaded scheduling strategy for batch processing problems, which allows it to fully utilize the capabilities of multicore processors. We demonstrate the effectiveness of the framework through three irregular GEMM types involving irregular matrix multiplication, showing that it can generate efficient execution plans that consistently achieve high performance across various scales and matrix characteristics.
- We present a template-based code generation method that generates highly optimized assembly kernels for diverse irregular GEMM types based on ARMv8 and X86 CPUs. Furthermore, we propose an input-aware tiling algorithm to generate optimal kernel combination strategies, thereby ensuring consistent high performance at any matrix scale.
- Based on our proposed framework, we have implemented a high-performance irregular matrix multiplication library for ARMv8 and Intel cascade Lake architectures. The experimental results demonstrate that the performance of IrGEMM surpasses that of the mainstream BLAS libraries in the three types of computations that encompass irregular matrix multiplication, includes Batch GEMM, Compact GEMM, and TSMM, based on the ARMv8 and X86 architectures. The aforementioned improvements of IrGEMM's performance signify a noteworthy advancement in the field of irregular matrix multiplication, as IrGEMM offers superior computational efficiency in comparison to the commonly employed BLAS library.

This paper builds upon the conference versions [10], [11], [12], [13] by: 1) enhancing the generality of the framework to handle a wider range of irregular matrix multiplication application scenarios; 2) adding experimental results for the port of IrGEMM to Intel cascade Lake CPUs; 3) refining the tiling algorithm to minimize memory access while considering the algorithm's complexity; and 4) providing a detailed description of the implementation and optimization techniques used in irregular GEMM code generation. These additions serve to further elaborate upon and strengthen the findings presented in the conference versions.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the overview of the IrGEMM. Section 4 elaborates on the design of the install-time stage. Section 5 describes the design and implementation details of the run-time stage. Section 6 presents the performance evaluation of our methods. Finally, Section 7 concludes this paper with future work.

2 RELATED WORK

Many GEMM algorithms [5], [14], [15], [16], [17], [18] have been proposed to compute the dense GEMM, small GEMM,

compact GEMM and batch GEMM. As the most popular dense GEMM optimize method prposed by GOTO [14], [19], is supported by most mainstream GEMM libraries. Batch GEMM as a new subroutine of BLAS has been discussed in a more comprehensive interface, and chased for multi-threaded load balancing optimization. And since the compact interface was proposed by intel, it is mainly aimed at optimizing under SIMD-friendly data layout.

Nowadays, many libraries are optimized by vendors or researchers to achieve high performance on specific hardware architectures. Vendor-supplied high-performance GEMM libraries (including ARMPL [6], Intel MKL [7]) are properly optimized for specific architectures using (SIMD) techniques, and have been fine-tuned for optimal performance on specific microarchitectures. However, these vendor-specific libraries cannot be easily ported to processors from other vendors. To address this issue, several auto-tuning systems, including OpenBLAS, BLIS, and LIBXSMM, have been developed by academia to provide comparable performance to vendor-specific libraries.

In terms of the application scenarios for which irregular matrix multiplication is oriented, there already have many studies in the academic community, as follows:

2.1 Small GEMM

Small matrices pose a challenge for HPC systems because modern processors are often designed to handle large-scale data. This can make it difficult to fully utilize multi-level cache structures and vector registers, which are key features of modern SIMD architectures. In addition, the overhead associated with data packing and boundary processing can significantly impact the performance of GEMM operations. These small GEMM characteristics prevent conventional approaches from achieving optimal performance on small GEMM. Designing a library without data packing steps and boundary processing is necessary to achieve high performance for small GEMM. LibShalom [17] proposes to overlap the packing and computation of GEMM, which is implemented by handwritten assembly code and distributes the overall GEMM load rationally to each computing kernel of the processor. LIBXSMM [5] uses the Just-in-time (JIT) code compilation technique to generate assembly code for small GEMMs. LIBXSMM uses code caching to reuse compilation results to reduce the JIT overhead. BLIS treats small-scale GEMM as part of a thin GEMM (skinny GEMM). BLIS defines a thin GEMM as a GEMM as long as one dimension of the input matrix is smaller. however, due to the imperfection of BLIS's tiling method and edge processing, and the small-scale matrix, BLIS is unable to obtain the near-optimal performance of a small-scale GEMM. However, due to the imperfection of BLIS's tiling method and edge processing, and the small-scale matrix, BLIS cannot obtain the near-optimal small-scale GEMM performance. These approaches give us great inspiration. However, we still need to consider how to load-balance the scheduling of these small GEMM kernels on batch GEMM.

2.2 Batch GEMM and Compact GEMM

Batch operations are an effective means of handling a large number of small matrices, as they allow for the efficient processing of multiple matrices simultaneously. To optimize batch GEMM operations, several approaches have been proposed, including compact GEMM and batch GEMM. For fixed sizes, the compact GEMM of the Intel MKL [16] uses a SIMD-friendly data layout that fully uses SIMD registers. IATF [11] proposes automatic tuning algorithms and code generation methods for ARM architectures based on SIMD-friendly data layouts. For variable size, the community has proposed a standard interface for Batch BLAS [20]. Intel MKL, ARMPL, BLIS, and other mainstream linear algebra libraries support this interface. The main optimization of batch GEMM focus on multi-thread load balancing. There has been research comparing the performance of OpenMP on batch processing problems with different strategies. The results show that the group-based approach is an effective way to handle variable batch GEMM [21]. In addition, there is also a large amount of GPU-based batch GEMM optimization research in the community [22], [23], which provides ideas for the work in this paper.

2.3 TSMM

Highly thin matrix-matrix multiplication is one of the most important irregular matrix-matrix multiplication methods, which indicates that one of the input matrices (A or B) is a highly thin matrix (one dimension is significantly smaller than the other). Its widely used in applications such as deep learning [24], [25] and Neural networks (NN) [26], [27].

Many deep learning frameworks, such as TensorFlow [28] and OneDNN, rely heavily on convolutional layers, which are implemented using GEMM. One popular technique for optimizing convolutional computation is the image-to-column algorithm, which converts the convolution operation into a GEMM operation in order to take advantage of optimized linear algebra libraries such as BLAS. However, the input sizes for GEMM operations can vary greatly depending on the application, and are often irregular in shape. In particular, convolutional kernels are often small while the input images are large, leading to a highly thin matrix-matrix multiplication problem. In some cases, the input matrix may need to be reused multiple times.

There has been researches on TSMM, with various vendors and researchers developing optimized implementations for different platforms. For example, Intel MKL provides highly optimized TSMM on X86 platforms, and Facebook has optimized TSMM [29] on its X86 processors in its data center on NVIDIA GPUs. However, TSMM optimization methods have not been fully discussed. While traditional optimization methods are also informative, the most important issue is that traditional matrix-matrix multiplication does not support data reuse, which as mentioned above is key in optimizing TSMMs for this class of applications. In addition, due to the small size of tall and thin matrices in some dimensions, traditional matrix-matrix multiplication implementations often only enable suboptimal tiling algorithms

3 THE IRGEMM FRAMEWORK

This paper presents an input-aware tuning framework for optimizing irregular GEMM on ARMv8 and X86 CPUs. Although we summarize the irregular GEMM into three types,

the framework we proposed is generic and can be tailored to specific patterns through implementation. The framework consists of the install-time stage and the run-time stage, as illustrated in Figure 1, and aims to achieve near-optimal performance for irregular GEMM. IrGEMM supports four modes: N, T, R, and C, representing non-transposed, transposed, conjugate non-transposed, and conjugate transposed matrices, respectively. For instance, GEMM in TN mode indicates that matrix A is transposed (T) and matrix B is not (N). In terms of data types, we support S, D, C, and Z for single precision floating point numbers, double precision floating point numbers, single precision complex numbers, and double precision complex numbers, respectively. The specific application scenarios for each irregular GEMM type are outlined in the table 1.

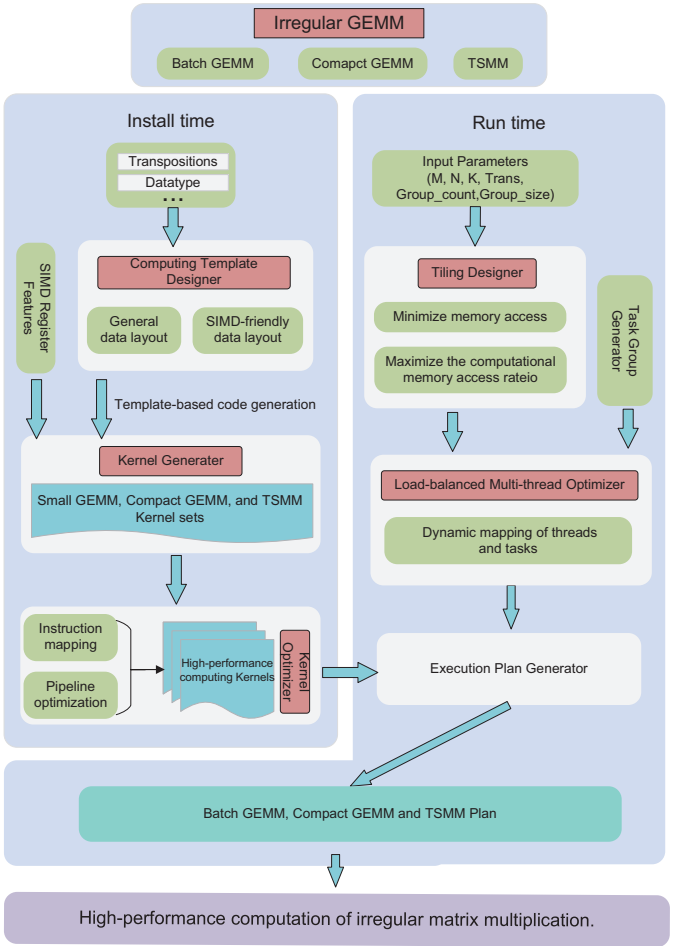


Fig. 1. Input-aware tuning framework to optimize irregular GEMM on ARMv8 and X86 CPUs

In the install-time stage, we present a code generation method that is based on the typical computational pattern of irregular GEMM, which can automatically generate a vast number of computational kernels of varying sizes. In addition, by leveraging our proposed instruction mapping method and pipeline optimization approach, we achieve superior performance on ARMv8 and X86 architectures. This stage contains the following components:

- **Computational Template Designer** designs kernels for the NN, NT, TN, and TT modes for three irregular GEMM types. For small GEMM, these kernels do not include packing operations. For compact matrices, the kernels are based on a SIMD-friendly data layout to take advantage of SIMD registers.
- **Kernel Generator** employs a computational template based algorithm to generate hundreds of kernels for every possible boundary case for each mode, and using a "ping-pong" strategy to minimize the memory access overhead.
- **Kernel Optimizer** optimizes the generated kernels through careful selection and scheduling of instructions to achieve performance that is as close as possible to the hardware optimal performance.

TABLE 1
Three classifications of irregular GEMM

Irregular GEMM Types	Scenario
Batch	There are many groups of small matrices ($\sqrt[3]{MNK} \leq 80$) and when looking for the ultimate multi-threaded performance.
Compact	When there are many matrices of the same size, and $M, N, K < 33$
TSMM	One of the matrices A or B is a tall-and-skinny matrix (one dimension of the matrix is much larger than the other), and the tall-and-skinny matrix needs to be reused by many times.

In the run-time stage, we present an input-aware tiling algorithm to generate an execution plan by selecting the optimal kernel for any input scale. This execution plan will be saved in the form of the command queues. We abstract the tiling problem as a boxing problem and use a dynamic programming algorithm to minimize the amount of memory access and to maximize the computational memory access ratio [30]. Furthermore, we propose a load-balanced multi-threaded optimizer to enable the Batch GEMM to achieve the optimal multi-threaded performance. This optimizer assigns tasks to threads dynamically by employing a dynamic mapping between threads and tasks. It contains the following components:

- **Tiling Designer** uses an input-aware tiling algorithm to generate an optimal kernel selection strategy for any possible input scale.
- **Load-balanced Multi-thread Optimizer**, designed specifically for Batch GEMM, generates task groups based on computer architecture, such as L1 cache and matrix size, using the small matrix multiplication execution plan and dynamically assigns tasks to threads through the dynamic mapping of threads and tasks.
- **Execution plan Generator** selects the optimal computation kernel based on the tiling strategy and generates the execution plan separately in the form of a command queue for each irregular GEMM types.

4 THE DESIGN OF INSTALL-TIME STAGE

In this section, we present the template-based code generation method which is built on typical GEMM computational patterns and obtains high performance by designing

code mapping and pipeline optimization strategies based on the characteristics of computer architecture. To take full advantage of these patterns and implement code generation, we introduce the computational template designer. Computational templates form the foundation of the code generation method and consist of meta-templates and hybrid templates. Meta-templates are sets of predefined C preprocessor macros that represent basic arithmetic operations. In contrast, hybrid templates are high-performance assembly codes that are designed for a specific processor architecture. By carefully designing the mapping between the meta-templates to the highly optimized assembly instructions, the install-time stage allows for the generation of high-performance computing kernels for three irregular GEMM types of irregular matrix multiplication, on both ARMv8 and X86 CPUs. The install-time stage employs three components to implement this process.

4.1 GEMM Kernel

This section outlines design strategies of GEMM kernel size, to take advantage of the limited number of vector register resources in modern CPUs. ARMv8 processors have 32 128-bit vector registers, while Intel Cascade Lake processors have 32 512-bit vector registers. When generating GEMM kernels in assembly, it is crucial to use these registers effectively to increase data reuse and reduce memory access overhead. One efficient method is to divide GEMM operations into smaller blocks. We refer to these blocks as microkernels, which compute the matrix multiplication between the matrix A of $m_r \times 1$ and the matrix B of $1 \times n_r$. The microkernel is then looped in K dimensions to compute the block of matrix C of size $m_r \times n_r$. In order to maximize data reuse, it is important to ensure that the matrix A of $m_r \times 1$, the matrix B of $1 \times n_r$, and the matrix C of $m_r \times n_r$ can be fully stored in the SIMD registers. Additionally, we should reserve registers for "ping-pong" operations.

Conventional approaches typically only require a main kernel to be designed to achieve high performance. This is because most of the computation can be tiled with the main kernel, and the boundary portion of the computation typically constitutes only a small fraction. However, in the case of irregular matrix multiplication, it is essential to account for the impact of boundary issues on performance. To achieve consistently high performance at any input matrix sizes, we need to design computing kernels for all possible boundary scales. In addition, to make full use of the width of the SIMD registers, this paper classifies irregular matrix multiplication into general and SIMD-friendly data layouts based on the application scenarios.

4.1.1 General data layout

Irregular GEMM types based on general data layout include batch GEMM and TSMM. Traditional methods often use data packing operations, which enable the computational kernel to have continuous access to memory and thus achieve high performance. Such methods are not suitable for small matrix multiplication, which results in significant memory access overhead. Hence, in this paper, we design separate small GEMM kernels for each transpose mode.

For TSMM, we design the data packing kernel considering that it is large enough to bring performance gains. This way only one computation kernel needs to be designed for each data type and it is able to handle all transpose patterns. In contrast to the conventional approach, we also designed all other boundary kernels as shown in Table 2.

TABLE 2
All generated kernels for TSMM

	ARM NEON	AVX512
S	$\{1,..,12\} \times \{1,..,8\}$	$\{1,..,48\} \times \{1,..,8\}$
D	$\{1,..,8\} \times \{1,..,4\}$	$\{1,..,16\} \times \{1,2\}$
C	$\{1,..,8\} \times \{1,..,4\}$	$\{1,..,8\} \times \{1,2\}$
Z	$\{1,..,4\} \times \{1,..,4\}$	$\{1,..,4\} \times \{1,2\}$

Tables 3 and 4 depict the kernels which we plan to generate for ARMv8 and X86 architectures, respectively. The ARM Neon instructions' support for vector-to-element multiplication enables the clever use of loop unrolling to handle the TN mode which is not conducive to SIMD operations. On the contrary, the AVX512 only supports vector-to-vector multiplication, rendering it incapable of making effective use of the SIMD registers in TN mode. To address this issue, we transpose the matrix A and then use the kernel of NN mode, as shown in Table 4. We point out that the performance impact of this transposition operation is acceptable.

TABLE 3
All generated kernels for small GEMM on ARMv8

	NN	NT	TN	TT
S	$16 \times \{1,..,4\}$ $12 \times \{1,..,6\}$ $8 \times \{1,..,8\}$ $4 \times \{1,..,13\}$ $3 \times \{1,..,13\}$ $2 \times \{1,..,13\}$ $1 \times \{1,..,13\}$	$16 \times \{1,..,4\}$ $12 \times \{1,..,8\}$ $8 \times \{1,..,8\}$ $4 \times \{1,..,20\}$ $3 \times \{1,..,24\}$ $2 \times \{1,..,28\}$ $1 \times \{1,..,32\}$	$4 \times \{1,..,4\}$ $3 \times \{1,..,5\}$ $2 \times \{1,..,7\}$ $1 \times \{1,..,10\}$	$\{1,..,4\} \times 16$ $\{1,..,6\} \times 12$ $\{1,..,8\} \times 8$ $\{1,..,13\} \times 4$ $\{1,..,13\} \times 3$ $\{1,..,13\} \times 2$ $\{1,..,13\} \times 1$
D	$8 \times \{1,..,4\}$ $4 \times \{1,..,8\}$ $3 \times \{1,..,8\}$ $2 \times \{1,..,15\}$ $1 \times \{1,..,15\}$	$8 \times \{1,..,4\}$ $4 \times \{1,..,8\}$ $3 \times \{1,..,8\}$ $2 \times \{1,..,20\}$ $1 \times \{1,..,20\}$	$4 \times \{1,..,4\}$ $3 \times \{1,..,5\}$ $2 \times \{1,..,7\}$ $1 \times \{1,..,10\}$	$\{1,..,4\} \times 8$ $\{1,..,8\} \times 4$ $\{1,..,8\} \times 3$ $\{1,..,15\} \times 2$ $\{1,..,15\} \times 1$
C	$8 \times \{1,..,4\}$ $4 \times \{1,..,9\}$ $3 \times \{1,..,9\}$ $2 \times \{1,..,12\}$ $1 \times \{1,..,20\}$	$8 \times \{1,..,4\}$ $4 \times \{1,..,8\}$ $3 \times \{1,..,8\}$ $2 \times \{1,..,12\}$ $1 \times \{1,..,20\}$	$4 \times \{1,..,9\}$ $3 \times \{1,..,9\}$ $2 \times \{1,..,12\}$ $1 \times \{1,..,20\}$	$\{1,..,4\} \times 8$ $\{1,..,9\} \times 4$ $\{1,..,9\} \times 3$ $\{1,..,12\} \times 2$ $\{1,..,20\} \times 1$
Z	$4 \times \{1,..,4\}$ $3 \times \{1,..,4\}$ $2 \times \{1,..,7\}$ $1 \times \{1,..,10\}$	$4 \times \{1,..,4\}$ $3 \times \{1,..,4\}$ $2 \times \{1,..,7\}$ $1 \times \{1,..,10\}$	$4 \times \{1,..,4\}$ $3 \times \{1,..,4\}$ $2 \times \{1,..,7\}$ $1 \times \{1,..,10\}$	$\{1,..,4\} \times 4$ $\{1,..,4\} \times 3$ $\{1,..,7\} \times 2$ $\{1,..,10\} \times 1$

These kernels, which cover almost all possible boundary cases, allow us to minimize the generation of small blocks as well as reduce the amount of memory accesses. This is crucial for the performance improvement of irregular matrix multiplication, which we will discuss in detail in Section 5.1.

4.1.2 SIMD-friendly data layout

For application scenarios with a large number of GEMMs of the same size, we utilize SIMD-friendly data layout to fully utilize the vector processing capability of modern processors. We would like to emphasize that some applications are inherently the SIMD-friendly data layout. In addition,

TABLE 4
All generated kernels for small GEMM on AVX512

	NN/TN	NT	TT
S	$\{1,.,16\} \times \{1,.,30\}$ $\{17,.,32\} \times \{1,.,13\}$ $\{33,.,48\} \times \{1,.,8\}$ $\{49,.,64\} \times \{1,.,5\}$	$\{1,.,16\} \times \{1,.,30\}$ $\{17,.,32\} \times \{1,.,13\}$ $\{33,.,48\} \times \{1,.,8\}$ $\{49,.,64\} \times \{1,.,5\}$	$\{1,.,5\} \times \{1,.,64\}$ $\{6,.,8\} \times \{1,.,48\}$ $\{9,.,13\} \times \{1,.,32\}$ $\{14,.,30\} \times \{1,.,16\}$
D	$\{1,.,8\} \times \{1,.,30\}$ $\{9,.,16\} \times \{1,.,13\}$ $\{17,.,24\} \times \{1,.,8\}$ $\{25,.,32\} \times \{1,.,5\}$	$\{1,.,8\} \times \{1,.,30\}$ $\{9,.,16\} \times \{1,.,13\}$ $\{17,.,24\} \times \{1,.,8\}$ $\{25,.,32\} \times \{1,.,5\}$	$\{1,.,5\} \times \{1,.,32\}$ $\{6,.,8\} \times \{1,.,24\}$ $\{9,.,13\} \times \{1,.,16\}$ $\{14,.,30\} \times \{1,.,8\}$
C	$\{1,.,8\} \times \{1,.,15\}$ $\{9,.,16\} \times \{1,.,6\}$ $\{17,.,24\} \times \{1,.,4\}$ $\{25,.,32\} \times \{1,.,2\}$	$\{1,.,8\} \times \{1,.,15\}$ $\{9,.,16\} \times \{1,.,6\}$ $\{17,.,24\} \times \{1,.,4\}$ $\{25,.,32\} \times \{1,.,2\}$	$\{1,2\} \times \{1,.,32\}$ $\{3,.,4\} \times \{1,.,24\}$ $\{5,.,6\} \times \{1,.,16\}$ $\{7,.,15\} \times \{1,.,8\}$
Z	$\{1,.,4\} \times \{1,.,15\}$ $\{5,.,8\} \times \{1,.,6\}$ $\{9,.,12\} \times \{1,.,4\}$ $\{13,.,16\} \times \{1,.,2\}$	$\{1,.,4\} \times \{1,.,15\}$ $\{5,.,8\} \times \{1,.,6\}$ $\{9,.,12\} \times \{1,.,4\}$ $\{13,.,16\} \times \{1,.,2\}$	$\{1,.,2\} \times \{1,.,16\}$ $\{3,.,4\} \times \{1,.,12\}$ $\{5,.,6\} \times \{1,.,8\}$ $\{7,.,15\} \times \{1,.,4\}$

for some applications, the cost of converting to the SIMD-friendly data layout is the same as converting to the generic data layout [16]. Therefore the SIMD-friendly data layout used in this paper does not add any additional overhead.

As depicted in Figure 2, when applied to a group of matrices, this layout arranges the corresponding locations of consecutive P matrices in a contiguous region of memory, with padding added as necessary when there are insufficient P matrices. The value of P depends on the data type and the length of the SIMD register. For example, when working with single-precision floating-point data on the Kunpeng 920 processor with the 128-bit SIMD register, P=4, and on the x86 architecture with AVX512 support, P=16. The data is designed to fit the length of the SIMD register.

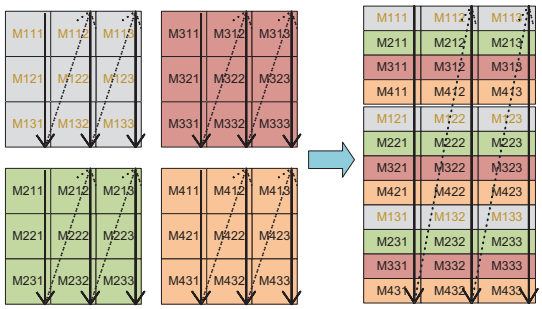


Fig. 2. SIMD-friendly Data Layout:3×3 matrices on Kunpeng 920.

In this layout, vectorization is a natural consequence. We only need to address offsets to accommodate all transpose patterns. Table 5 shows all the kernel sizes we have designed for compact GEMM.

TABLE 5
All generated kernels for compact GEMM

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

4.2 Computing Template Designer

We conducted a thorough analysis of computational patterns of irregular matrix multiplication and identified them as computational templates. As illustrated in Template 1, the meta-template takes the kernel scale as input and generates a high-level computational template as output. For the purposes of the following discussion, we assume that the size of the input kernel is $M_r \times N_r$. For each input kernel size, these six templates (I, M1, M2, E, SAVE, SUB) are utilized to implement the "ping-pong" operation which means the data needed by the next template is loaded in the current template to avoid pipeline bubbles.

Template 1: Meta templates supported in IrGEMM

Input: m_r, n_r: the size of input kernel
Output: Computing micro-kernel

1 /*Matrix C is stored in $C_{[m_r \times n_r]}$ */
2 **TEMPLATE_I**
3 $LOAD \begin{cases} V_{\{A1\}} /*For \quad I*/ \\ V_{\{A2\}} /*For \quad M2*/ \end{cases} \leftarrow A_{[2m_r]}$
4 $LOAD \begin{cases} V_{\{B1\}} /*For \quad I*/ \\ V_{\{B2\}} /*For \quad M2*/ \end{cases} \leftarrow B_{[2n_r]}$
5 $C_{[m_r \times n_r]} \leftarrow FMUL(V_{\{A1\}}, V_{\{B1\}})$
6 $/*SIMD \quad Multiply*/$
7 **TEMPLATE_M1**
8 $LOAD \quad V_{\{A2\}} \leftarrow A_{[m_r \times 1]}$
9 $LOAD \quad V_{\{B2\}} \leftarrow B_{[1 \times n_r]}$
10 $C_{[m_r \times n_r]} \leftarrow FMA(V_{\{A1\}}, V_{\{B1\}})$
11 **TEMPLATE_M2**
12 $LOAD \quad V_{\{A1\}} \leftarrow A_{[m_r \times 1]}$
13 $LOAD \quad V_{\{B1\}} \leftarrow B_{[1 \times n_r]}$
14 $C_{[m_r \times n_r]} \leftarrow FMA(V_{\{A2\}}, V_{\{B2\}})$
15 **TEMPLATE_E**
16 $C_{[m_r \times n_r]} \leftarrow FMA(V_{\{A2\}}, V_{\{B2\}})$
17 **TEMPLATE_SUB**
18 $LOAD \quad V_{\{A1\}} \leftarrow A_{[m_r \times 1]}$
19 $LOAD \quad V_{\{B1\}} \leftarrow B_{[1 \times n_r]}$
20 $C_{[m_r \times n_r]} \leftarrow FMA(V_{\{A1\}}, V_{\{B1\}})$
21 **TEMPLATE_SAVE**
22 $LOAD \quad V_{\{orignC\}} \leftarrow originC_{[m_r \times n_r]}$
23 $V_{\{orignC\}} \leftarrow FMA(C_{[m_r \times n_r]}, Alpha)$
24 $STORE \quad V_{\{orignC\}} \rightarrow originC_{[m_r \times n_r]}$

Template 1 presents our abstracted GEMM templates, which include I, M1, M2, E, SAVE, and SUB. Each template computes the matrix-matrix multiplication (via SIMD FMA/FMUL) of $m_r \times 1$ of matrix A with $1 \times n_r$ of matrix B to obtain $m_r \times n_r$ of matrix C, as shown in lines 5, 10, 14, 16, and 20. Matrix C is stored in the SIMD register $C_{[m_r \times n_r]}$.

- $TEMPLATE-I$ and $TEMPLATE-E$ represent the entry and the exit of "ping-pong" operation, respectively.$TEMPLATE-I$ loads the data it needs and the data required by $M2$ in lines 3-4 and completes the computation at line 5. $TEMPLATE-E$ only contains calculation instructions, as shown in line 16.

- $TEMPLATE-M1$ and $TEMPLATE-M2$ represent the two phases of the "ping-pong" operation. The $M1$ template loads the data needed by $M2$ in lines 8-9 and completes the computation at line 10. The $M2$ template loads the data needed by $M1$ in lines 12-13 and completes the computation at line 14. The "ping-pong" operation will loop between $M1$ and $M2$, which we will detail in Section 4.3.
- $TEMPLATE-SUB$ only loads the data it needs in lines 18-19 and completes the computation at line 20. This template mainly deals with the case when the kernel scale is small and "ping-pong" operations are not profitable.
- $TEMPLATE-SAVE$ multiplies the parameter $ALPHA$ with the matrix C and stores the matrix in memory, as shown in lines 22-24.

It should be emphasized that the prefetching of the next stage in the "ping-pong" operation does not strictly follow this process, but also takes into account whether the matrix is transposed and the choice of assembly instructions. In some cases, we load all the data needed for $M1$ and $M2$ at template $M1$ to take full advantage of the SIMD registers. We will describe this in detail in Section 4.4.

These templates are built on the typical computational patterns of GEMM. Further fine-tuning is necessary for irregular GEMM-oriented applications in different specific types. We will provide detailed descriptions in the following sections.

Algorithm 1: Computing kernel generator of irregular GEMM

Input: $A_r : m_r \times K$; $B_r : K \times n_r$; $C_r : m_r \times n_r$;
Output: Compute kernel($C_r = A_r \times B_r + alpha \times C_r$)

```
1  if K < 4 then
2      if K == 3 then
3          TEMPLATE_I; TEMPLATE_E; TEMPLATE_SUB;
4      else if K == 2 then
5          TEMPLATE_I; TEMPLATE_E;
6      else
7          V_{2(m_r+n_r)} - V_{2(m_r+n_r)+m_r×n_r-1} ← 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;
```

4.3 Kernel Generator

In this section, We utilize the templates to automatically generate the computing kernels with ping-pong operations, as outlined in Algorithm 1. The use of automatic code generation significantly reduces the workload of our methods. The generator updates the $m_r \times n_r$ block of matrix C by computing the matrix multiplication of a matrix block A of $m_r \times K$ and a matrix block B of $K \times n_r$. When the computation of the entire matrix C block is completed, it is stored in memory. The generator takes a specified block size and the parameter K as input and invokes the necessary computational templates to generate efficient irregular GEMM kernels for that scale. These kernels are combined at run-time stage into an overall irregular GEMM execution plan, making their implementation and optimization crucial to the overall performance of the algorithm.

The algorithm 1 addresses the case where $K < 4$, as shown in lines 1-8, by utilizing a set of templates. When $K >= 4$, the ping-pong operation is initiated using TEMPLATE_I (line 10) and terminated using TEMPLATE_E (line 14). The loop between TEMPLATE_M1 and TEMPLATE_M2, which constitutes the core of the ping-pong operation, is used to minimize pipeline bubbles. In the case where $K\%2 == 1$, a SUB processing step is performed. It is worth noting that similar techniques can be applied to complex-number GEMM kernels.

4.4 Kernel Optimizer

We define instruction mapping rules to translate the high level computational templates into architecture-specific hardcoded optimization templates by selecting and scheduling efficient assembly instructions. IrGEMM currently focuses on the ARMv8 ISA and the x86-64 ISA. Benefiting from the optimization templates, when new architectures emerge, we only need to implement the corresponding optimization templates.

4.4.1 General data layout

For standard matrix column (row) main-order storage, we carefully analyze the impact of data packing on performance in irregular GEMM. This paper only employs the data packing operation when it is advantageous to do so.

For the small matrix multiplication kernel that will be used in batch GEMM, we designed computational kernels for each of the four transposition modes.

In NN mode, we use the SIMD instruction to load matrix A one column at a time. For AVX512, when the elements of matrix B can be reused, that is, when m_r is large, we broadcast the B matrix into the register. And when m_r is small and there is no need to reuse B, matrix B can be loaded using the AVX512 instruction with embedded broadcast, which fuses the load instruction with the compute instruction. While there is no equivalent instruction in the ARMv8 architecture, this paper proposes the use of ping-pong operation to reduce memory access overhead by directly loading adjacent rows. The instructions corresponding to the mate templates in NN mode under the AVX512 and ARMv8 architecture are detailed in the Figure 3.

The loading process for matrix A in NT mode is similar to that in NN mode. The difference lies in the loading direction of matrix B. For AVX512 instructions, we also use

	Small GEMM on NN mode and TSMM		Compact GEMM	
Hybrid templates	**AVX512**	**ARM NEON**	**AVX512**	**ARM NEON**

LOAD A(addr, offset, FA, V_a{1, .., m})	/*The offset based on computational types*/
	mov FA, 0
	vmovupd V_a1, [addr + FA]
	add FA, offset
	vmovupd V_a2, [addr + FA]
	…
	add FA, offset
	vmovupd V_an, [addr + FA]
	add FA, offset
		/*The offset based on computational types*/
		mov FA, 0
		ldr V_a1, [addr + FA]
		add FA, offset
		ldr V_a2, [addr + FA]
		…
		add FA, offset
		ldr V_am, [addr + FA]
		add FA, offset
			/*if A transpose, offset = LDA*/
			/*if A non-transpose, offset = 64*/
			mov FA, 0
			vmovupd V_a1, [addr + FA]
			add FA, offset
			vmovupd V_a2, [addr + FA]
			…
			add FA, offset
			vmovupd V_am, [addr + FA]
			add FA, offset
				/*if A transpose, offset = LDA*/
				/*if A non-transpose, offset = 16*/
				mov FA, 0
				ldr V_a1, [addr + FA]
				add FA, offset
				ldr V_a2, [addr + FA]
				…
				add FA, offset
				ldr V_am, [addr + FA]
				add FA, offset

LOAD B(addr, offset, FB, V_b{1, .., n})	/*The offset based on computational types*/
	mov FB, 0
	vbroadcastsd V_b1, QWORD PTR [addr + FB]
	add FB,offset
	vbroadcastsd V_b2, QWORD PTR [addr + FB]
	add FB, offset
	...
	vbroadcastsd V_bn, QWORD PTR [addr + FB]
	add FB, offset
		/*For Small GEMM, this only load on M1*/
		mov FB, 0
		ldr V_b1, [addr + FB]
		add FB, offset
		ldr V_b2, [addr + FB]
		add FB, offset
		...
		ldr V_bn, [addr + FB]
		add FB, offset
			/*if B transpose, offset = 64*/
			/*if B non-transpose, offset = LDB*/
			mov FB, 0
			vmovupd V_b1, [addr + FB]
			add FB, offset
			vmovupd V_b2, [addr + FB]
			add FB, offset
			...
			vmovupd V_bn, [addr + FB]
				/*if B transpose, offset = 16*/
				/*if B non-transpose, offset = LDB*/
				mov FB, 0
				ldr V_b1, [addr + FB]
				add FB, offset
				ldr V_b2, [addr + FB]
				add FB, offset
				...
				ldr V_bn, [addr + FB]

MUL(V_c{1, .., mxn}, V_a{1, .., m}, V_b{1, .., n})	vmulpd V_c1, V_a1, V_b1 vmulpd V_c2, V_a2, V_b1 … vmulpd V_cm, V_am, V_b1 … vmulpd V_c{m*(n-1)+1}, V_a1, V_bn vmulpd V_c{m*(n-1)+2}, V_a2, V_bn … vmulpd V_c{m*n}, V_am, V_bn	fmul V_c1.2d, V_a1.2d, V_b1.d[0] fmul V_c2.2d, V_a2.2d, V_b1.d[0] … fmul V_cm.2d, V_am.2d, V_b1.d[0] … fmul V_c{m*(n-1)+1}.2d, V_a1.2d, V_bn.d[0] fmul V_c{m*(n-1)+2}.2d, V_a2.2d, V_bn.d[0] … fmul V_c{m*n}.2d, V_am.2d, V_bn.d[0]	vmulpd V_c1, V_a1, V_b1 vmulpd V_c2, V_a2, V_b1 … vmulpd V_cm, V_am, V_b1 … vmulpd V_c{m*(n-1)+1}, V_a1, V_bn vmulpd V_c{m*(n-1)+2}, V_a2, V_bn … vmulpd V_c{m*n}, V_am, V_bn	fmul V_c1.2d, V_a1.2d, V_b1.2d fmul V_c2.2d, V_a2.2d, V_b1.2d … fmul V_cm.2d, V_am.2d, V_b1.2d … fmul V_c{m*(n-1)+1}.2d, V_a1.2d, V_bn.2d fmul V_c{m*(n-1)+2}.2d, V_a2.2d, V_bn.2d … fmul V_c{m*n}.2d, V_am.2d, V_bn.2d
/*TEMPLATE-M1*/ FMA(V_c{1, .., mxn}, V_a{1, .., m}, V_b{1, .., n})	vfmadd231pd V_c1, V_a1, V_b1 vfmadd231pd V_c2, V_a2, V_b1 … vfmadd231pd V_cm, V_am, V_b1 … vfmadd231pd V_c{m*(n-1)+1}, V_a1, V_bn vfmadd231pd V_c{m*(n-1)+2}, V_a2, V_bn … vfmadd231pd V_c{m*n}, V_am, V_bn	/*K=0 with TEMPLATE-M1*/ /*K=1 with TEMPLATE-M2 and TEMPLATE-E*/ fmla V_c1.2d, V_a1.2d, V_b1.d[K] fmla V_c2.2d, V_a2.2d, V_b1.d[K] … fmla V_cm.2d, V_am.2d, V_b1.d[K] … fmla V_c{m*(n-1)+1}.2d, V_a1.2d, V_bn.d[K] fmla V_c{m*(n-1)+2}.2d, V_a2.2d, V_bn.d[K] … fmla V_c{m*n}.2d, V_am.2d, V_bn.d[K]	vfmadd231pd V_c1, V_a1, V_b1 vfmadd231pd V_c2, V_a2, V_b1 … vfmadd231pd V_cm, V_am, V_b1 … vfmadd231pd V_c{m*(n-1)+1}, V_a1, V_bn vfmadd231pd V_c{m*(n-1)+2}, V_a2, V_bn … vfmadd231pd V_c{m*n}, V_am, V_bn	fmla V_c1.2d, V_a1.2d, V_b1.2d fmla V_c2.2d, V_a2.2d, V_b1.2d … fmla V_cm.2d, V_am.2d, V_b1.2d … fmla V_c{m*(n-1)+1}.2d, V_a1.2d, V_bn.2d fmla V_c{m*(n-1)+2}.2d, V_a2.2d, V_bn.2d … fmla V_c{m*n}.2d, V_am.2d, V_bn.2d

STORE(dst, FC, LDC, V_c{1, .., mxn})	mov FC, 0
	vmovupd [dst], V_c1
	vmovupd [dst + 64], V_c2
	…
	vmovupd [dst + (m-1)*64], V_cm
	…
	add FC, LDC
	vmovupd [dst+FC], V_c{m*(n-1)+1}
	vmovupd [dst+FC + 64], V_c{m*(n-1)+2}
	…
	vmovupd [dst+FC + (m-1)*64], V_c{m*n}
		mov FC, 0
		str [dst], V_c1
		str [dst, #16], V_c2
		…
		str [dst, #((m-1)*16)], V_cm
		…
		add dst, LDC
		str [ds], V_c{m*(n-1)+1}
		str [dst, #16], V_c{m*(n-1)+2}
		…
		str [dst, #((m-1)*16)], V_c{m*n}
			mov FC, 0
			vmovupd [dst], V_c1
			vmovupd [dst + 64], V_c2
			…
			vmovupd [dst + (m-1)*64], V_cm
			…
			add, FC, LDC
			vmovupd [dst+FC], V_c{m*(n-1)+1}
			vmovupd [dst+FC + 64], V_c{m*(n-1)+2}
			…
			vmovupd [dst+FC + (m-1)*64], V_c{m*n}
				str [dst], V_c1.2d
				str [dst, #16], V_c2.2d
				…
				str [dst, #((m-1)*16)], V_cm.2d
				…
				add, dst, LDC
				str [dst], V_c{m*(n-1)+1}.2d
				str [dst, #64], V_c{m*(n-1)+2}.2d
				…
				str [dst, #((m-1)*16)], V_c{m*n}.2d

Fig. 3. Instruction mapping rules between hybrid templates and optimization templates for DGEMM based on AVX512 and ARM NEON.

broadcast instructions. For the ARM architecture, NT mode is favorable for SIMD operations as it allows for the direct loading of a row of elements into the vector register. And then, by using of the ARMv8 FMA instruction's vector outer product multiplication (i.e. multiply a vector with each lane of another vector separately) to complete the matrix multiplication.

Unlike NT mode, TN mode is not conducive to SIMD operations due to the reversed data fetch order. To fully utilize modern processor SIMD instructions, we transpose matrix A and apply the same execution strategy as in NN mode. Traditional methods handle the problem of TN mode using data packing, which packs the matrices based on the size of the computational kernel. However, data packing is typically done for both matrices A and B, leading to a significant memory access overhead. In contrast, this paper proposes a transpose operation for only matrix A. It is worth noting that for small matrices, the transposed data can be completely stored in the L2 cache and will not be swapped out during the entire operation. This transposition strategy does not incur excessive access memory overhead, but rather the performance benefits of vectorization are significant.

TT mode is the inverse of NN mode in terms of data order and requires only the opposite loading strategy of NN mode to complete the computation.

The instruction mapping for TSMM is similar to that for Small GEMM because they are both based on the general data layout. We only need to adjust the address offset when loading the data to generate the assembly kernel.

4.4.2 SIMD-friendly data layout

In SIMD-friendly data layout, we only need to design address offsets to be able to handle all transpose patterns. Figure 3 presents the conversion of the matrix template to AVX512 assembly instructions. The efficient design of this kernel template significantly reduces our workload. When porting to different platforms, it is only necessary to match the corresponding multiplication and access instructions in order to achieve a high-performance compact GEMM kernel.

In addition, after conducting experiments, we discovered that the packing operation improves performance on our implementation of the Kunpeng 920 platform. The Kunpeng 920's efficient memory access reduces the impact of packing overhead on performance, while the resulting

sequential accesses yield significant improvements. We also evaluated the packing strategy on an Intel platform and found that, in most cases, direct loading without data packing yields higher performance. We would like to stress that the decision to use data packing for compact GEMM should be carefully evaluated in light of the specific characteristics of the targeted computer architecture.

4.4.3 Pipeline optimization

In the kernel designer, various kernels of different sizes can be generated based on the templates abstracted by us. However, the directly generated kernel instruction pipeline is not optimal. To ensure that each execution unit of the processor is fully utilized, as shown in Figure 4, we employ the following two methods for rescheduling: 1) separate instructions that have data associated with them to the greatest extent possible., and 2) inserting access instructions between arithmetic instructions in order to hide memory access latency.

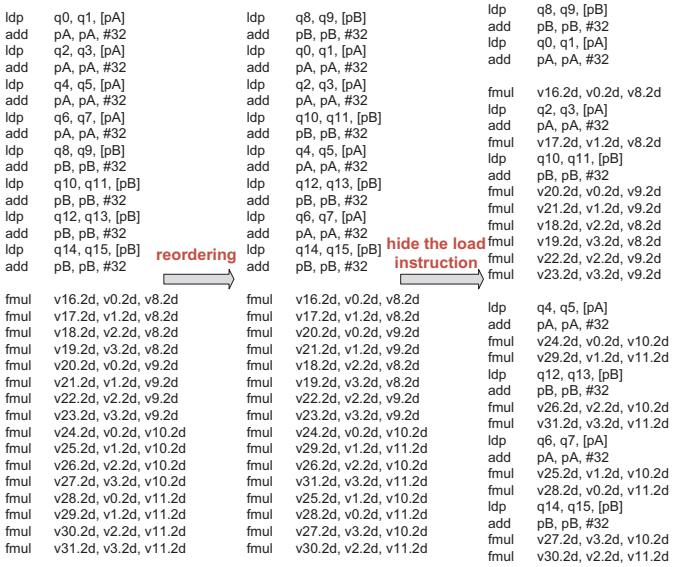


Fig. 4. Kernel optimization strategy based on ARMv8

In particular, for the batch processing problem, we also insert a prefetch instruction between two GEMM operations to minimize the occurrence of cache misses as much as possible.

5 THE DESIGN OF RUN-TIME STAGE

In the run-time stage, we propose an input-aware tiling algorithm that selects the optimal kernels, which are generated during the install-time stage, for any input size. Additionally, we propose load-balanced multi-thread scheduling algorithms to fully utilize the performance of modern multi-core processors. In the tiling strategy, we carefully analyze the factors that impact performance and abstract the tiling problem into the boxing problem. We use the dynamic programming approach to minimize the amount of memory access and to maximize the computational memory access ratio. The kernels selected by the tiling algorithm are linked as execution plans and saved as command queues, which significantly reduces the number of branch commands and enhances performance.

5.1 Tiling Designer

The tiling designer selects the optimal kernels from among hundreds of kernels based on the user input matrix size, and saves it as an execution plan in the form of a command queue. This approach enables generating the plan once and calling it multiple times to reduce overhead.

The tiling algorithm discussed in this section is concerned with optimizing block tiling at the microkernel level for both ARMv8 and Intel x86 architectures. In large-scale GEMM problems, there is often no need to consider tiling in particular. This is because 1) most of the computation can be done using a specially optimized kernel that can hide the memory access overhead almost completely using computational instructions, and 2) the bounding part accounts for a small percentage of the total computation size, so high performance can be achieved overall even without special optimization. However, the boundary regions of irregular matrix multiplications cannot be ignored as they may contribute a significant portion of the computation, and thus a reasonable tiling strategy is necessary to ensure nearly optimal performance. In this paper, we consider the tiling strategy from two perspectives:

- From the perspective of memory access size, a reasonable tiling algorithm can reduce the amount of accessed data. As shown in the figure 5, each block is loaded with the matrix A of $m_r \times k$ and the matrix B of $K \times n_r$, and they are multiplied to obtain the matrix C block of $m_r \times n_r$.

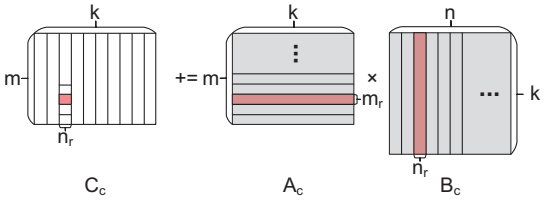


Fig. 5. The inner kernel perform a slice-times-slice matrix-matrix multiplication(m_r and n_r are the sizes suit for register locking).

We can easily conclude that the whole GEMM problem needs to load a total of $\Sigma(m_r + n_r) \times K$ data. And the number of computations is the same for different tiling methods. Therefore, how to minimize the Equation 3 is an important design direction.

$$Minimize(\Sigma(m_r + n_r)) \tag{3}$$

- From the point of view of hiding memory access overhead using computational instructions, we should maximize the computational memory access ratio (CMAR) [30], which is defined as the ratio of computational operations to memory accesses and is important for efficiently hiding memory access overhead. The computational memory access ratio for the GEMM problem is given by the expression $\frac{2\times m_r \times n_r}{m_r + n_r}$. Our goal is to make sure that the CMAR of each kernel is not too small, and we can do this by minimizing Equation 4.

$$Minimize(\Sigma(\frac{1}{m_r} + \frac{1}{n_r})) \tag{4}$$

This paper abstracts the tiling problem as a two-dimensional boxing problem , which is known to be an NP problem, aiming to fill the large box with a number of small boxes exactly while satisfying Equations 3 and 4. Where the large boxes represent the size of the input matrix C and the small boxes represent the size of the hundreds of kernels we generate. In addition, there is no limit to the number of each small box that can be used.

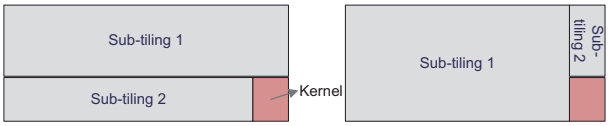


Fig. 6. Decomposition of the tiling problem

This paper presents a dynamic programming approach to design a tiling algorithm for the this problem. The goal is to divide the problem into smaller subproblems, or "tiles," and find the optimal tiling strategy among them to iterate the optimal execution plan. Regarding the subproblem decomposition, we emphasize that 1) when the size of a generated kernel is equal to the size of the subproblem, it is optimal to directly use with this kernel to avoid additional memory access overhead. 2) In this paper, we divide the tiling problem into the two cases depicted in Figure 6, where the block in the lower right corner represents the size of a certain kernel in the kernel set. We can obtain the optimal strategy for these two subproblems by several iterations. We have to emphasis that, although other partitioning methods exist [31], they require algorithms with higher algorithm complexity to implement (e.g., DFS algorithm) and are not suitable for the problem addressed in this paper.

As demonstrated in Algorithm 2, this algorithm takes the size of the matrix C and the generated kernel as inputs, and produces the optimal tiling strategy as output. The notation dp[i][j] is used to represent $\Sigma(M_r + N_r)$ (Eq. 3) at the current scale, dp_v represents $\Sigma\frac{1}{m_r} + \frac{1}{n_r}$ (Eq. 4), and dp_back denotes the current optimal tiling policy at the current scale, in the form of a command queue containing several kernels.

As shown in lines 2-3 in Algorithm 2, the dynamic programming method traverses the entire two-dimensional space. We use K to represent the current traversed kernel, as shown in line 4. When the current size can be exactly filled by a single kernel, the optimal strategy is to directly call this kernel, as shown in line 6. The current state is then saved in lines 7-9. Otherwise when there is a strategy with fewer memory accesses, as depicted in Equation 3, update the dp array as shown in lines 10-11. The corresponding state is saved, as shown in lines 12-17. Furthermore, in case of strategies with equal memory accesses but larger CMAR, as depicted in Equation 4, the strategy with the larger total CMAR is selected and saved, as shown in lines 19-24.

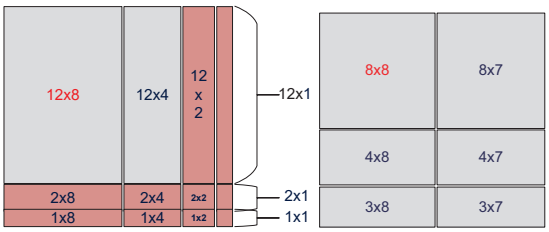
Figure 7 shows the difference between the traditional tiling method and our proposed tiling method under the ARMv8 architecture. Our approach achieves a significant reduction in the amount of memory accesses. Moreover, it effectively reduces the formation of smaller blocks, which possess a low CMAR and present challenges for optimizing high-performance computations.

Algorithm 2: Tiling algorithm

Input: $M \times N$: Matrix size of C;
$kernels$: List with all generated kernels
Output: Execution plan for irregular GEMM

1 $init\_dp\_matrix(dp\_matrix, M, N, dp\_back)$;
2 **for** $i = 1 \rightarrow M$ **do**
3 **for** $j = 1 \rightarrow N$ **do**
4 **for** *k in kernels* **do**
5 m = k.m, n = k.n;
6 **if** *i - m == 0 and j - n == 0* **then**
7 $dp[i][j] = m + n$;
8 $dp\_v[i][j] = \frac{1}{m} + \frac{1}{n}$;
9 $dp\_back[i][j] = k.kernel$;
10 **else if** *dp[i][j] > min(dp[i-m][j]+dp[m][j-n]+m+n, dp[i][j-n]+dp[i-m][n]+m+n)* **then**
11 $dp[i][j] = min(dp[i-m][j] + dp[m][j-n] + m + n, dp[i][j-n] + dp[i-m][n] + m + n)$;
12 **if** *dp[i-m][j]+dp[m][j-n] > dp[i][j-n]+dp[i-m][n]* **then**
13 $dp\_back[i][j] = dp\_back[i][j-n] + dp\_back[i-m][n] + k.kernel$;
14 $dp\_v[i][j] = dp\_v[i][j-n] + dp\_v[i-m][n] + \frac{1}{m} + \frac{1}{n}$;
15 **else**
16 $dp\_back[i][j] = dp\_back[i-m][j] + dp\_back[m][j-n] + k.kernel$;
17 $dp\_v[i][j] = dp\_v[m][j-n] + dp\_v[i-m][j] + \frac{1}{m} + \frac{1}{n}$;
18 **else if** *dp[i][j] == min(dp[i-m][j]+dp[m][j-n]+m+n, dp[i][j-n]+dp[i-m][n]+m+n)* **then**
19 **if** *dp_v[i-m][j]+dp_v[m][j-n] > dp_v[i][j-n]+dp[i-m][n]* **then**
20 $dp\_back[i][j] = dp\_back[i][j-n] + dp\_back[i-m][n] + k.kernel$;
21 $dp\_v[i][j] = dp\_v[i][j-n] + dp\_v[i-m][n] + \frac{1}{m} + \frac{1}{n}$;
22 **else**
23 $dp\_back[i][j] = dp\_back[i-m][j] + dp\_back[k.m][j-n] + k.kernel$;
24 $dp\_v[i][j] = dp\_v[m][j-n] + dp\_v[i-m][j] + \frac{1}{m} + \frac{1}{n}$;

In addition, the batch GEMM and compact GRMM only need to consider the blocking at the micro-kernel level since their application scenarios allow the matrix to be fully stored in the L2 cache. In contrast, TSMM cannot store entirely into the L2 cache, requiring us to consider the blocking strategy at the cache level. Typically, the traditional GEMM algorithm packs a block of the A or B matrices into a linear buffer, enabling it to be stored in the last level of the data cache [32]. Similarly, another matrix block is packed into a linear buffer adapted to the L2 data cache. Data packing is



(a) Traditional tiling method (b) Small GEMM tiling method

Fig. 7. Tiling method of 15×15 SGEMM.

essential to achieve high-performance GEMM by reducing memory access latency through cache locality optimization [33], [34]. However, we note that the blocking parameter is usually fixed which is not suitable for tall-and-skinny matrices. This is because the valuable cache resource are wasted when the dimensions of the tall-and-skinny matrix are smaller than the blocking size. In this paper, we proposed a dynamically tuned blocking method based on cache size and input matrix properties for TSMM to fully utilize the cache resource. Additionally, considering that the application scenario of TSMM often requires reusing the tall-and-skinny matrix multiple times, we extract the packing operation for the tall-and-skinny matrix to further reduce the overhead of data packaging operations.

5.2 Load-balanced Multi-thread Optimizer

In this section, we present a load-balanced multithreaded optimization approach designed for the batch GEMM. This highly effective batch GEMM optimization strategy consists of two steps. First, a number of large matrix groups are divided into smaller task groups. Second, these task groups are dynamically assigned to threads using a dynamic mapping between threads and command queues proposed in this paper. This strategy allows us to achieve significant performance improvements in multi-threaded acceleration.

5.2.1 Task Group Generator

The task group generator divides each matrix group into smaller groups, called task groups, which are then assigned to threads for processing. As shown in Figure 8, the tiling designer develops an optimal computing strategy based on matrix properties in order to maximize performance without using data packing. This strategy is only applied once per matrix group in order to minimize overhead. However, directly assigning task groups to threads can lead to an imbalance in workload, as the number and size of matrices can vary significantly between groups. On the other hand, assigning individual matrices to threads can result in a high overhead for thread scheduling, particularly when the matrix size is small and the number of threads is large, causing some threads to wait for task assignment. Thus, it is important to group task groups in a way that matches the hardware specifications in order to ensure optimal performance.

$$\sum(m_i \times k_i + m_i \times n_i + n_i \times k_i) \leq L1 \ \ cache \qquad (5)$$

In accordance with Equation 5, we present an upper bound on the matrix contained within each task group. This bound

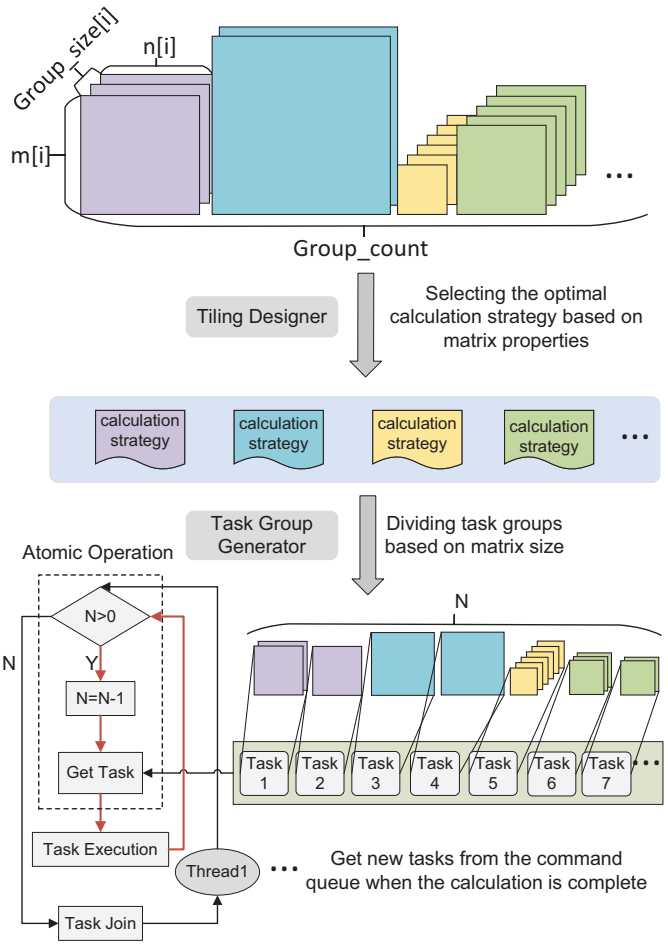


Fig. 8. Overview of multi-thread scheduling.

is determined by the matrix size and the size of the L1 cache. As illustrated in Figure 8, we store small GEMM groups that do not exceed this bound in the form of a command queue, allowing threads to execute these commands directly without invoking the small GEMM interface.

5.2.2 Dynamic mapping between threads and tasks

To achieve the optimal multi-thread speed-up ratio, we employ dynamic assignment of task groups to threads. While the pre-grouping strategy results in minimal variance in the computational workload among the task groups, it is worth noting that larger matrices tend to achieve higher processor performance, while particularly small GEMMs may only attain approximately 10% of peak processor performance according to [10]. As a result, even when the computational workload is equivalent, the task group with larger matrices will execute significantly faster than the group with smaller matrices, potentially causing a load imbalance. To address this issue, we propose a dynamic mapping of threads and tasks to enhance multi-thread execution efficiency.

This dynamic scheduling algorithm designs for load balancing, which is an improvement over static scheduling. The kernel selection and tiling design will be placed in command queues, which reduces the resource allocation overhead for individual threads. As shown in Figure 8, the simplified task assignment process is depicted. The number

Algorithm 3: Load-balanced Multi-thread scheduling algorithm

Input: A, B, C; /*array*/
$M\_array, N\_array, K\_array$;
$transa\_array, transb\_array$;
$group\_rount, group\_size$;
Output: Dynamic scheduling strategy

```
1  idx = 0, T = 0;
2  for P = 0 → group_count − 1 do
3      tra = transa_array[p];
4      trb = transb_array[p];
5      m = m_array[p];
6      n = n_array[p];
7      k = k_array[p];
8      kernel = tiling_designer(tra, trb, m, n, k);
9      Ntask = L1_cache_size/(m×n+n×k+k×m);
10     for i = 0 → group_size[p] − 1; i+ = Ntask do
11         compute_n = min(Ntask, group_size[p] − i);
12         task_group[T] ← (kernel, compute_n, idx);
13         T+ = 1; idx+ = compute_n;
14 mutex.init();
15 total_num = T;
16 create_threads(thread_array, THREAD_NUM);
17 /* Multi-thread execution */
18 while T > 0 do
19     mutex.lock(), T− = 1;
20     if T < 0 then
21         mutex.unlock(), break;
22     idx = total_num − T − 1, mutex.unlock();
23     task_group[idx].run();
24 join_threads(thread_array, THREAD_NUM);
```

of matrices assigned to each thread is based on the size of each group of matrices. The global variable N is used to indicate the current number of remaining tasks. When a thread finishes its current task, it will check if there are any remaining tasks that can be executed. When $N > 0$, a new group of tasks is obtained from the command queue. This process is atomic. The dynamic scheduling method, which is based on the pre-grouping of tasks, allows LBBGEMM to achieve optimal multi-thread acceleration.

Algorithm 3 presents a pseudo-code depiction of the aforementioned optimization process. First, the algorithm determines the optimal tiling strategy and computation kernels based on the characteristics of the input matrix, as illustrated in lines 3-8. This selection only needs to be performed once per group of matrices. Second, the algorithm calculates the most suitable number of matrices for the current task group, as shown in line 9. Third, it divides the large set of matrices into several smaller task groups for the multi-thread optimizer to allocate and schedule, as demonstrated in lines 10-13. These task groups are represented in the form of command queues, as indicated in line 12. Our implementation ensures that the chosen computation strategy is optimal for the arrangement of these commands given the matrix properties. Finally, a load-balanced multi-thread optimizer dynamically assigns these task groups to each thread to ensure maximum load balancing, as shown in lines 14-24. A mutex lock is initialized in line 14, and the multi-thread execution in lines 18-23 allows each thread to access the current state of the command queue by atomically accessing the global variable T. If a thread is idle and the task queue is not empty, it will execute a new task. These methods enable the efficient computation of batch GEMMs on state-of-the-art hardware platforms.

5.3 Execution Plan Generator

The execution plan generator selects the optimal compute kernel based on the tiling designer and links it as an execution plan, which is saved as a command queue. By executing this command queue, we are able to compute the irregular GEMM with high performance. We ensure that this is the optimal choice for instruction scheduling at these input parameters.

For Compact GEMM, the tiling designer utilizes high-performance kernels based on a SIMD-friendly data layout to generate a high-performance execution plan. As discussed in Section 4.4, data packing operations can yield performance gains on the Kunpeng 920, and thus the execution plan based on Kunpeng 920 includes data packing operations that match the computational kernel.

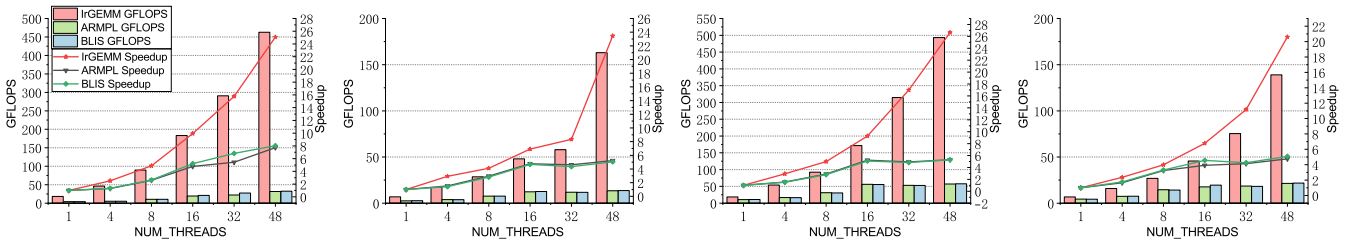
For batch GEMM, it leverages the small GEMM kernel and combines it with the load-balanced multithreaded scheduling framework to achieve high performance for all threaded modes. It is important to note that each group in batch GEMM usually contains multiple matrices of the same size, and the small GEMM plan is generated at the beginning of each group. As a result, when allocated to each matrix, these overheads are negligible.

For TSMM, we designs a blocking strategy based on the properties of the matrices, as described in section 5.1, to maximize cache utilization. Subsequently, the computing kernels are chosen based on the result the tiling designer. Finally, the above policies are linked into the execution plan.

6 PERFORMANCE EVALUATION

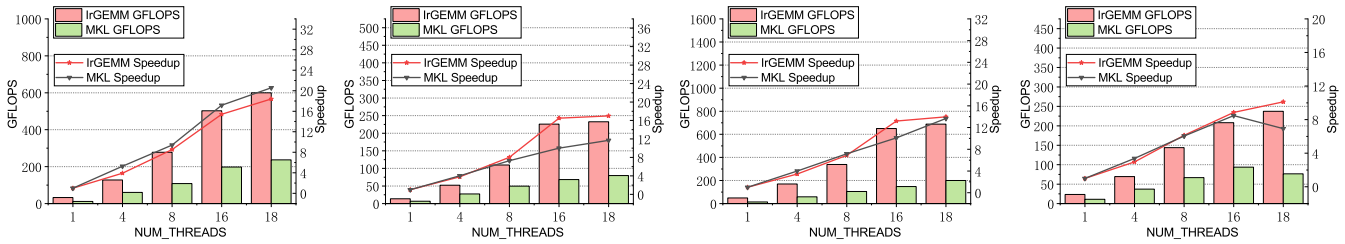
We presents an evaluation of the performance of IrGEMM on server-grade ARMv8 (Kunpeng 920) and x86-64 (Intel Cascade Lake) CPUs. Our experiments found that the Intel Xeon Gold 6240 CPU was instable in performance when overclocked. Therefore, we adjusted the frequency to the processor's base frequency which is 2.6GHz. To evaluate the effectiveness of IrGEMM, we compared it with five BLAS libraries, including Intel OneAPI MKL - the official performance library for Intel X86 architecture, ARMPL - the official performance library for ARM architecture, BLIS - a widely used open-source BLAS library in the industry, LIBXSMM - which is optimized for small matrices, and OpenBLAS - with is the most widely used open-source BLAS library in the industry. Table 6 outlines the experimental conditions.

The performance tests used in this paper were compiled using the GCC7.5 compiler with the "-O3 -g" option. We initialize the matrix by filling it with random floating point numbers (0 to 1) with reference to the generic test scheme [35]. We run each core 100 times and take the geometric mean as the final result.



(a) SGEMM Batch (b) DGEMM Batch (c) CGEMM Batch (d) ZGEMM Batch

Fig. 9. Performance of the multi-threaded IrGEMM batch GEMM compared with ARMPL, BLIS, and LIBXSMM based on ARMv8 architecture.



(a) SGEMM Batch (b) DGEMM Batch (c) CGEMM Batch (d) ZGEMM Batch

Fig. 10. Performance of the single-threaded IrGEMM batch GEMM compared with Intel MKL, BLIS, and LIBXSMM based on X86 architecture.

TABLE 6
Experimental environments

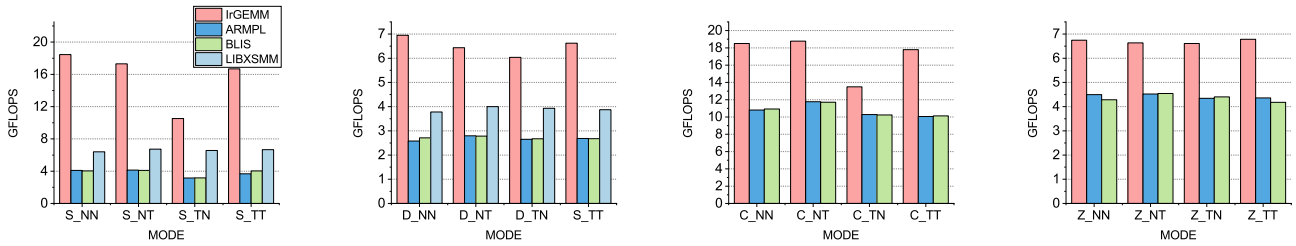
CPU	Kunpeng 920	Intel Xeon Gold 6240
Peak perf. (FP64)	10.4GFLOPS	83.2GFLOPS
Peak perf. (FP32)	41.6GFLOPS	166.4GFLOPS
Number of Cores	48	18
Arch.	ARMv8.2	Cascade Lake
Freq.	2.6GHz	2.6GHz
SIMD	128 bits	512 bits
L1D cache	64KB	32KB
L2 cache	512KB	1024KB
Compiler	GCC7.5	GCC7.5
Intel OneAPI MKL	-	23.0
ARMPL	22.1.0	-
BLIS	0.9.0	0.9.0
Libxsmm	1.17	1.17
OpenBLAS	0.3.21	0.3.21

6.1 Batch GEMM

In this study, we conduct a detailed analysis of the performance of IrGEMM in comparison with high-performance BLAS libraries supplied by vendors and the industry, on both ARM and X86 platforms. We evaluate the effectiveness of our proposed code generation method and tiling method using single-threaded performance comparisons. Moreover, we test our multi-threaded load balancing capability through a multi-threaded performance comparison. To fully test the load balancing capabilities of batch GEMM, we set group_count = 4, group_size = {10000, 1000, 100, 100}, m = n = k = {10, 20, 30, 40}.

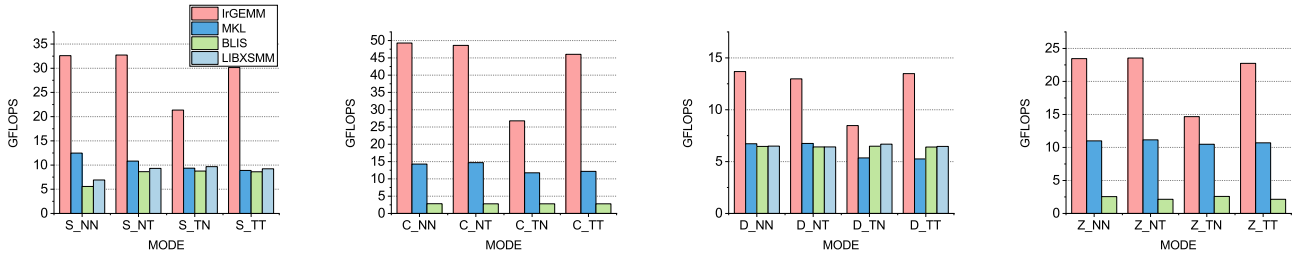
Figure 11 illustrates the single-threaded batch GEMM performance of IrGEMM, ARMPL, BLIS, and LIBXSMM on ARMv8 architecture, while Figure 12 presents the performance of single-threaded batch GEMM for IrGEMM, Intel MKL, BLIS, and LIBXSMM on Intel Cascade Lake architecture. Note that LIBXSMM does not support complex data types. Based on the results, we can make the following observations: 1) In terms of single-threaded performance, IrGEMM outperforms the other libraries on all four data types. For example, in ARM architecture, IrGEMM provides speedup ratios up to 2.7×, 2.5×, and 1.8× over ARMPL, BLIS, and LIBXSMM, respectively, for DGEMM. In X86 architecture, IrGEMM provides speedups up to 2.5×, 2.1×, and 2.1× over MKL, BLIS, and LIBXSMM, respectively. 2) IrGEMM performs better on real number problems than other libraries because traditional methods struggle to handle boundary problems, which constitute a significant proportion of the total computation for small matrices in batch problems. IrGEMM takes advantage of the SIMD features of modern processors by optimizing all possible boundary modes, and we proposed a dynamic tiling method to select the optimal computing kernels. 3) IrGEMM performs the worst in TN mode compared to other transpose modes, as TN mode is not SIMD friendly. Although we describe the treatment for TN mode in Section 4.4, there is still additional access overhead compared to other transposition modes. However, IrGEMM still outperforms other library implementations.

For multi-threaded tests, we have conducted a comprehensive analysis of performance at varying thread counts. It should be noted that LIBXSMM lacks support for a multi-threaded batch interface, while BLIS has only been evaluated to have multi-threaded acceleration on ARM platforms. Figure 9 presents the multithreaded performance of IrGEMM and ARMPL, BLIS on ARMv8 architecture. Figure 10 illustrates the multithreaded performance of IrGEMM and Intel MKL under the Cascade Lake architecture. To compare the load balancing capability of these three libraries, we utilize the ratio of multi-threaded performance to single-threaded performance as the multi-threaded speedup ratio.



(a) SGEMM Batch (b) DGEMM Batch (c) CGEMM Batch (d) ZGEMM Batch

Fig. 11. Performance of the single-threaded IrGEMM batch GEMM compared with ARMPL, BLIS, and LIBXSMM under the NN mode, based on ARMv8 architecture.



(a) SGEMM Batch (b) DGEMM Batch (c) CGEMM Batch (d) ZGEMM Batch

Fig. 12. Performance of the single-threaded IrGEMM batch GEMM compared with Intel MKL, BLIS, and LIBXSMM under the NN mode based on X86 architecture.

Based on the observations from Figure 9 and Figure 10, we draw the following conclusions: 1) Regarding absolute performance, the IrGEMM implementation in this study outperforms other BLAS library implementations on ARM and X86 platforms by a significant margin. It displays considerable performance advantages across all four data types. 2) On ARM platform, with regards to multi-threaded speedup ratio, ARMPL and BLIS perform comparably under multi-threaded acceleration, but the speedup is not evident beyond 16 threads. As a comparison, IrGEMM outperforms the other libraries significantly in terms of multi-threaded acceleration. Specifically, it achieves speedups of up to 4.5×, 8.7×, 8.4×, 9.3×, 13.1×, and 14.6× for 1, 4, 8, 16, 32, and 48 threads, respectively, compared to ARMPL. Similarly, compared to BLIS, we can achieve up to 4.6×, 8.6×, 8.4×, 8.7×, 10.5×, and 14.3× with 1, 4, 8, 16, 32, and 48 threads, respectively. 3) Additionally, we demonstrate extreme acceleration ratios on the X86 platform. Despite IrGEMM's multithreaded acceleration ratios on SGEMM being almost identical to Intel MKL, we still achieve a significant absolute performance advantage. IrGEMM's multithreading performance is superior to the MKL implementation for all other data types. Specifically, it achieves speedups of up to 3.3×, 2.9×, 3.2×, 4.4×, and 3.4× for 1, 4, 8, 16, and 18 threads, respectively, compared to Intel MKL. This indicates that the proposed load balancing optimization method is effective.

6.2 Compact GEMM

Of the mainstream BLAS libraries currently available, only Intel MKL supports the compact interface, while other libraries require the batch interface to conduct tests. To fully demonstrate the performance of each kernel, we evaluate the performance of square matrices of sizes 1-33 for each function with a batch size of 16384.

Figures 13 and 14 demonstrate the high performance of our Compact GEMM in NN mode for real/complex numbers with single or double precision. On the ARM platform, we compare IrGEMM with ARMPL batch, BLIS batch, and LIBXSMM batch interfaces, while on the X86 platform, we compare it to the MKL Compact interface, the BLIS batch, and the LIBXSMM batch interface. As shown in Figures 13 and 14, we can conclude that 1) IrGEMM demonstrates significant performance advantages on both the ARM and X86 platforms. Specifically, for DGEMM on the ARM platform, compared to ARMPL, BLIS, and LIBXSMM, IrGEMM provides up to 3.4×, 3.2×, and 2.4× speedups, respectively. On the X86 platform, compared to MKL, BLIS, and LIBXSMM, IrGEMM provides up to 1.4×, 7.6×, and 4.8× speedups for DGEMM, respectively. 2) Both IrGEMM and MKL's compact interfaces exhibit strong performance when the matrix size is small. This is due to the fact that both implementations use a SIMD-friendly data layout. The conventional data layouts used by other libraries have difficulty taking full advantage of the width of SIMD registers when the matrix size is small. 3) As the matrix size increases, the performance gap gradually decreases. Although the traditional data layout performs poorly in small-scale matrix multiplication due to its inability to take full advantage of the SIMD features of modern processors, we point out that when the matrix size is large enough, SIMD-friendly data layouts will no longer be advantageous. This is because the computational access ratio of the traditional data layout is larger than that of the SIMD-friendly data layout. As described in section 5.1, for real numbers, the computational access ratio of the traditional data layout is $\frac{2\times m_r\times n_r}{m_r+n_r}$, while the computational access ratio of the SIMD-friendly data layout is $\frac{m_r\times n_r}{m_r+n_r}$ [11]. A large computational access ratio means that we can use compute instructions to

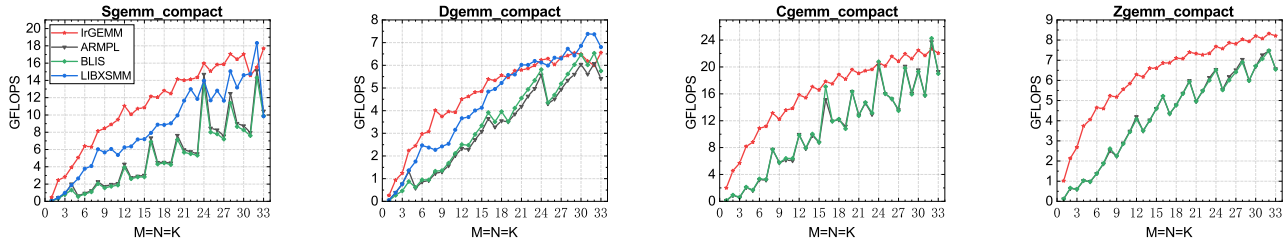


Fig. 13. Performance of the IrGEMM compact GEMM compared with ARMPL, BLIS, and LIBXSMM based on ARMv8 architecture.

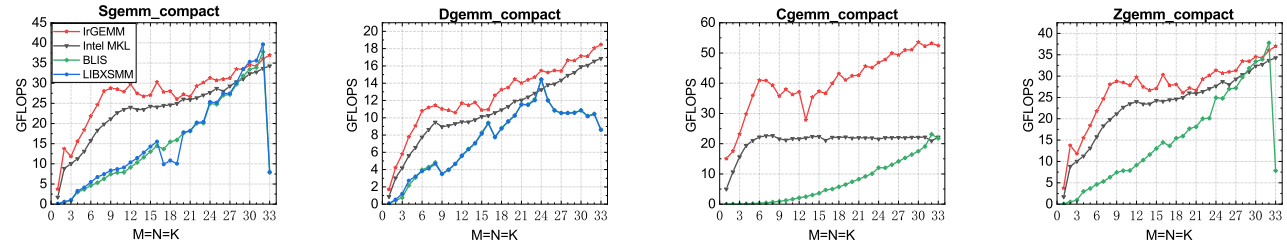


Fig. 14. Performance of the IrGEMM compact GEMM compared with Intel MKL, BLIS, and LIBXSMM based on X86 architecture.

hide memory access instruction latency. When the matrix size is large enough, the conventional data layout can also make great use of the width of the SIMD registers. Nevertheless, the disadvantage of memory access overhead in the computing kernel makes SIMD-friendly data layout no longer advantageous for large-scale matrix multiplication.

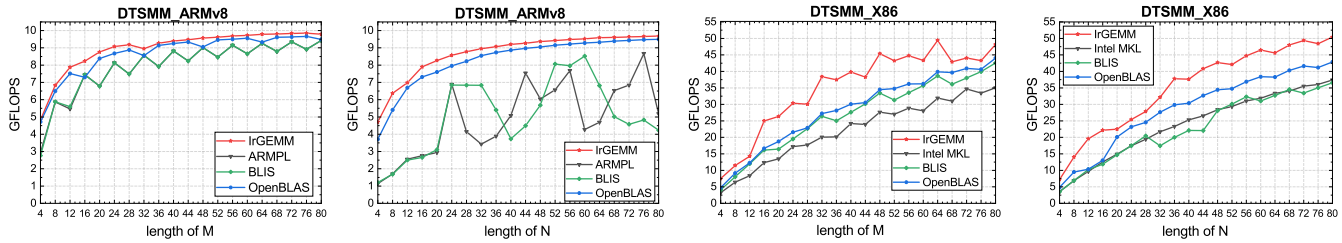
6.3 TSMM

This paper evaluates the performance of TSMM on the ARMv8 and X86 architectures. On the ARM platform, we compare IrGEMM with ARMPL, BLIS, and OpenBLAS, while on the X86 platform, we compare it with MKL, BLIS, and OpenBLAS. These libraries are known to deliver high performance in large-scale matrix multiplication, with Intel MKL library supporting the TSMM interface (gemm_compute) and BLIS having special optimization for tall-and-skinny matrices. Our tests compare the performance of two cases where matrix A and matrix B are tall-and-skinny matrices, respectively. In the case where matrix A is tall-and-skinny, we assign its dimensions to be {4, 8, ..., 80} $\times$ 10240, while matrix B is set to have dimensions 10240 $\times$ 10240. Conversely, if matrix B is tall-and-skinny, we set its dimensions to be 10240$\times$ {4, 8, ..., 80}, and assign the dimensions of matrix A to be 10240 $\times$ 10240. In addition to the single-thread test, this article also compares the performance between 48 threads on the ARM platform and 16 threads on the X86 platform.

Figure 15 and Figure 16 present the results of TSMM tests in NN mode for double precision real numbers in single and multi-threaded mode. The performance comparison leads to the following conclusions: 1) IrGEMM outperforms ARMPL, BLIS, and OpenBLAS on both ARM and X86 platforms. Specifically, on the ARM platform, for DTSMM, IrGEMM provides up to 2.9$\times$, 3.0$\times$, 1.1$\times$ speedups compared to ARMPL, BLIS, and OpenBLAS, respectively, and in the 48-thread test, it provides up to 9.6$\times$, 9.5$\times$, 4.5$\times$ performance improvements, respectively. On the X86 platform, compared to MKL, BLIS, and OPENBLAS, it provides up to 2.0$\times$, 2.0$\times$, 1.9$\times$ speedups, and in tests with 16 threads, up to 2.6$\times$, 2.9$\times$, 3.4$\times$ performance improvements, respectively. 2) While the traditional implementation also exhibits higher performance than IrGEMM in tests against the ARM platform, IrGEMM still has a performance advantage. In contrast, IrGEMM shows a considerable performance advantage in the test on the X86 platform, indicating the effectiveness of the code generation method and input-aware tiling algorithm designed in this paper in handling the TSMM problem. 3) Although the implementation of this paper does not show a significant advantage over other libraries in single-core tests based on ARMv8 CPUs, IrGEMM shows a considerable performance advantage in the multi-threaded test. We argue that this is because other libraries are not optimized for TSMM for multi-threading. Moreover, BLIS on the X86 platform demonstrates strong performance in the case of matrix A as a high skinny matrix and poor performance in the case of matrix B as a high thin matrix. We believe the blocking process of BLIS may not explicitly consider whether matrix A or B is tall-and-skinny, leading to the cache not being fully utilized.

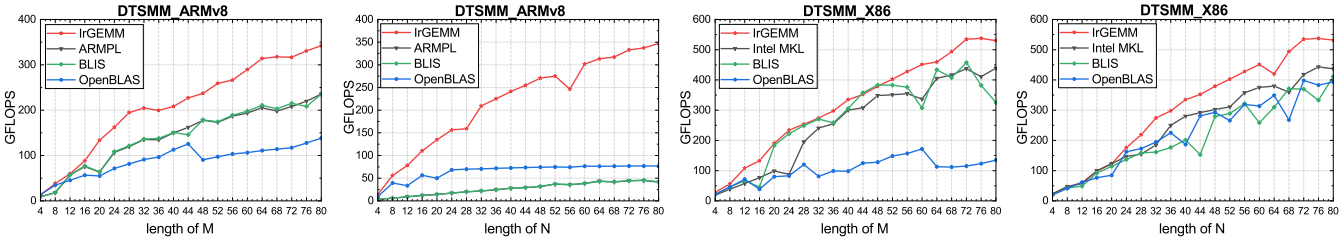
7 Conclusion

This paper presents IrGEMM, an input-aware tuning framework for irregular GEMM based on ARMv8 and X86 CPUs. It includes the install-time stage and the run-time phase. In the install-time stage, we focus on designing and optimizing the computing kernel based on the computer architecture's characteristics. We proposed a code generation method to efficiently generate computational kernels using kernel computation templates and instruction mapping. In the run-time stage, this paper proposes an input-aware tiling algorithm to achieve a comprehensive automatic tuning process. For multi-threaded optimization for batch GEMM, the paper divides the large matrix group into task groups, dynamically assigning them to threads through the dynamic map-



(a) A is a tall-and-skinny matrix (b) B is a tall-and-skinny matrix (c) A is a tall-and-skinny matrix (d) B is a tall-and-skinny matrix

Fig. 15. Performance of the single-threaded IrGEMM TSMM compared with ARMPL, BLIS, and OpenBLAS based on ARMv8 architecture, and compared with Intel MKL, BLIS, and OpenBLAS based on X86 architecture.



(a) A is a tall-and-skinny matrix (b) B is a tall-and-skinny matrix (c) A is a tall-and-skinny matrix (d) B is a tall-and-skinny matrix

Fig. 16. Performance of the multi-threaded IrGEMM TSMM compared with ARMPL, BLIS, and OpenBLAS based on ARMv8 architecture, and compared with Intel MKL, BLIS, and OpenBLAS based on X86 architecture.

ping between threads and tasks. The experimental results demonstrate that IrGEMM is highly competitive on both ARMv8 and X86 architectures, which signifies a notable progression in the area of irregular GEMM.

In the future, we aims to optimize other BLAS routines and investigate and extend the approach to state-of-the-art CPUs and GPUs.

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] A. Khodayari, A. R. Zomorrodi, J. C. Liao, and C. D. Maranas, "A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data," *Metabolic engineering*, vol. 25, pp. 50–62, 2014.
[2] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse qr factorization on the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 2, pp. 1–29, 2017.
[3] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, *et al.*, "High-performance tensor contractions for gpus," *Procedia Computer Science*, vol. 80, pp. 108–118, 2016.
[4] J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza, "Poster: A batched cholesky solver for local rx anomaly detection on gpus," *PUMPS: Moscow, Russia*, 2013.
[5] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "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*, no. 1, pp. 981–991, IEEE, 2016.
[6] "Arm performance libraries," [n.d.].
[7] "Intel oneapi math kernel library," [n.d.].
[8] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, pp. 1–33, 2015.
[9] "Openblas:an optimized blas library," [n.d.].
[10] J. Yao, B. Shi, C. Xiang, H. Jia, C. Li, H. Cao, and Y. Zhang, "Iaat: A input-aware adaptive tuning framework for small gemm," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 899–906, IEEE, 2021.
[11] C. Wei, H. Jia, Y. Zhang, L. Xu, and J. Qi, "Iatf: An input-aware tuning framework for compact blas based on armv8 cpus," in *Proceedings of the 51st International Conference on Parallel Processing*, ICPP '22, (New York, NY, USA), Association for Computing Machinery, 2023.
[12] C. Li, H. Jia, H. Cao, J. Yao, B. Shi, C. Xiang, J. Sun, P. Lu, and Y. Zhang, "Autotsmm: An auto-tuning framework for building high-performance tall-and-skinny matrix-matrix multiplication on cpus," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 159–166, IEEE, 2021.
[13] C. Wei, H. Jia, Y. Zhang, K. Li, and L. Wang, "Lbbgemm: A load-balanced batch gemm framework on arm cpus," in *2022 IEEE 24rd Int Conf on High Performance Computing & Communications*, pp. 1–8, 2022.
[14] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008.
[15] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl, "Blasfeo: Basic linear algebra subroutines for embedded optimization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 4, pp. 1–30, 2018.
[16] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing vector-friendly compact blas and lapack kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.
[17] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "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*, pp. 1–14, 2021.
[18] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *2012*

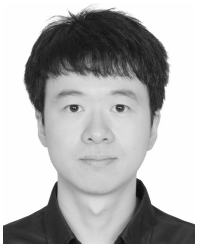
IEEE 18th international conference on parallel and distributed systems, pp. 684–691, IEEE, 2012.
[19] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 1, pp. 1–14, 2008.
[20] A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Kurzak, P. Luszczek, S. Tomov, *et al.*, "A set of batched basic linear algebra subprograms and lapack routines," *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 3, pp. 1–23, 2021.
[21] P. Valero-Lara, I. Martinez-Perez, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, and J. Labarta, "Variable batched dgemm," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 363–367, IEEE, 2018.
[22] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for gpus," in *International Conference on High Performance Computing*, pp. 21–38, Springer, 2016.
[23] R. Wang, Z. Yang, H. Xu, and L. Lu, "A high-performance batched matrix multiplication framework for gpus under unbalanced input distribution," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 1741–1758, 2022.
[24] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 830–841, IEEE, 2018.
[25] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, *et al.*, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *arXiv preprint arXiv:1811.09886*, 2018.
[26] W. Cao, X. Wang, Z. Ming, and J. Gao, "A review on neural networks with random weights," *Neurocomputing*, vol. 275, pp. 278–287, 2018.
[27] M. Anthony and P. Bartlett, *Neural network learning: Theoretical foundations*. cambridge university press, 1999.
[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.
[29] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, *et al.*, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *arXiv preprint arXiv:1811.09886*, 2018.
[30] H. Lan, J. Meng, C. Hundt, B. Schmidt, M. Deng, X. Wang, W. Liu, Y. Qiao, and S. Feng, "Feathercnn: Fast inference computation with tensorgemm on arm architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 580–594, 2019.
[31] M. Monaci and A. G. dos Santos, "Minimum tiling of a rectangle by squares," *Annals of Operations Research*, vol. 271, pp. 831–851, 2018.
[32] F. Wang, H. Jiang, K. Zuo, X. Su, J. Xue, and C. Yang, "Design and implementation of a highly efficient dgemm for 64-bit armv8 multi-core processors," in *2015 44th International Conference on Parallel Processing*, pp. 200–209, 2015.
[33] Q. Han, Y. Hu, F. Yu, H. Yang, B. Liu, P. Hu, R. Gong, Y. Wang, R. Wang, Z. Luan, *et al.*, "Extremely low-bit convolution optimization for quantized neural network on modern computer architectures," in *Proceedings of the 49th International Conference on Parallel Processing*, pp. 1–12, 2020.
[34] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, pp. 1–18, 2016.
[35] Z. Jia, A. Zlateski, F. Durand, and K. Li, "Optimizing n-dimensional, winograd-based convolution for manycore cpus," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 109–123, 2018.

Cunyang Wei received the master's degree from the Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing, China, in 2023. His main research interest is high performance computing.

Haipeng Jia received the PhD degree from the Ocean University of China, Qingdao, China, in 2012. He was a visiting PhD student with the Institute of Software, Chinese Academy of Sciences from 2010 to 2012. He is currently an associate professor with the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences. His research interests include heterogeneous computing and manycore parallel programming method.

Yunquan Zhang (Senior Member, IEEE) received the Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2000. He is a Full Professor of computer science with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include high-performance parallel computing, with particular emphasis on large scale parallel computation and programming models, high-performance parallel numerical algorithms, and performance modeling and evaluation for parallel programs.

Jianyu Yao received the master's degree from the Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing, China, in 2022. His main research interest is high performance computing.



Chendi Li received the master's degree from the Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing, China, in 2022. He is currently working toward the PhD degree with the The University of Utah. His main research interest is high performance computing.

Wenxuan Cao is a current master's student at the Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing, China. His research is in high performance computing and he has participated in projects related to architecture and compilers.