LBBGEMM: A Load-balanced Batch GEMM Framework on ARM CPUs

Cunyang Wei*†, Haipeng Jia*‡, Yunquan Zhang*, Kun Li§, Luhan Wang*†
*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
†School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
§Microsoft Research
{weicunyang20g, jiahaipeng, zyq, wangluhan21s}@ict.ac.cn, kunli@microsoft.com

***Abstract*—The trend in modern high performance computing is to decompose a large linear algebraic problem into many small problems that can be solved independently. Although there are many studies to obtain near-peak performance for large-scale dense matrix operations, it is not sufficient for batch operations with small matrices. The study of small GEMM kernel optimization and load balanced scheduling of batch operations on ARM processors is not enough. In this paper, we present LBBGEMM, a load-balanced batch GEMM framework for optimizing large groups of variable-size small GEMM to boost near-optimal performance based on ARMv8 architecture. The LBBGEMM is divided into the install-time stage and the run-time stage. At the install-time stage, we analyze the characteristics of each transposition mode to design the high-performance small GEMM kernel without data packing. This strategy greatly reduces the memory access overhead. In addition, we optimize instruction scheduling and instruction selection carefully to achieve optimal performance. The run-time stage provides a comprehensive auto-tuning process for batch GEMM by using a tiling designer and a pre-grouped dynamic scheduling algorithm. The tiling designer generates high-performance execution plans for each group of matrices with different sizes. Then we divide the large group of GEMM operations into small task groups. These task groups are assigned to threads for execution in the form of command queues through our proposed dynamic mapping between threads and tasks. This pre-grouped multi-thread task scheduling algorithm greatly improves the speedup of multi-thread. The experiments show that LBBGEMM could achieve significant performance improvements in batch GEMM compared with other mainstream BLAS libraries.**

***Index Terms*—GEMM, Batch GEMM, Small matrices**

I. Introduction

Recently the mainstream basic linear algebra libraries (BLAS) have delivered near-peak high performance on large-scale General Matrix Multiplication (GEMM). However, many modern applications are based on solutions with a large number of small matrix operations, such as metabolic networks [1], PDE-based simulation [2], tensor shrinkage for finite element simulation [3], and image [4] processing. A typical example might be to perform

$$\alpha_i A_i B_i + \beta_i C_i = C_i, i = 0 \rightarrow L \tag{1}$$

where L is large, but A_i, B_i, and C_i are small matrices. The community has proposed Batch_GEMM as an important extension to the traditional BLAS library [5]. In addition to the traditional matrix parameters of the GEMM interface, Batch_size is added to indicate the number of matrices of the same size in each matrix group, and batch_count indicates the number of matrix groups. It is difficult to deliver extremely high performance on modern processors when using traditional methods to compute these problems. Therefore, it is important to design batch GEMM optimization methods on state-of-the-art hardware platforms.

Mainstream BLAS libraries such as Arm Performance Libraries (ARMPL) [6], Intel oneAPI Math Kernel Library (Intel MKL) [7], and BLIS [8] have added support for Batch_GEMM routines. However, there are still opportunities to achieve more extreme performance for batch GEMM based on the ARMv8 architecture. First, high performance on a single small GEMM is the basis for multi-core parallel optimization of batch GEMMs. Small GEMM kernels with consistently high performance at all possible sizes are insufficiently researched on the ARM architecture. Second, different groups contain different size of matrices, which makes load-balanced task scheduling necessary to exploiting the multi-core performance of modern processors. Through experiments, we believe that batch GEMM on the ARM platform still has optimization opportunities for multi-thread acceleration.

This paper presents LBBGEMM, a high-performance batch GEMM framework on ARMv8 CPUs, to meet the needs of new applications on batch GEMM. It contains the install-time stage and the run-time stage. The install-time stage generates a series of high-performance computing kernels for small GEMM. We design computing kernels for each transposition mode without data packing to reduce the expensive memory access overhead. In addition, we optimize the kernel for each possible boundary case to minimize the boundary processing overhead. The run-time stage chooses the optimal kernels and tiling strategy, combined with the load-balanced multi-thread scheduling strategy, to generate a high-performance batch GEMM executing plan. Firstly, Our proposed tiling designer generates the execution plan for each matrix group according to input matrix properties (Matrix Size, Transposition). Secondly, we divide the large group of GEMM operations into smaller task groups for thread calls. Then the multi-thread optimizer assigns tasks to threads through a dynamic mapping of threads and tasks that we proposed. With the pre-

‡Corresponding authors.

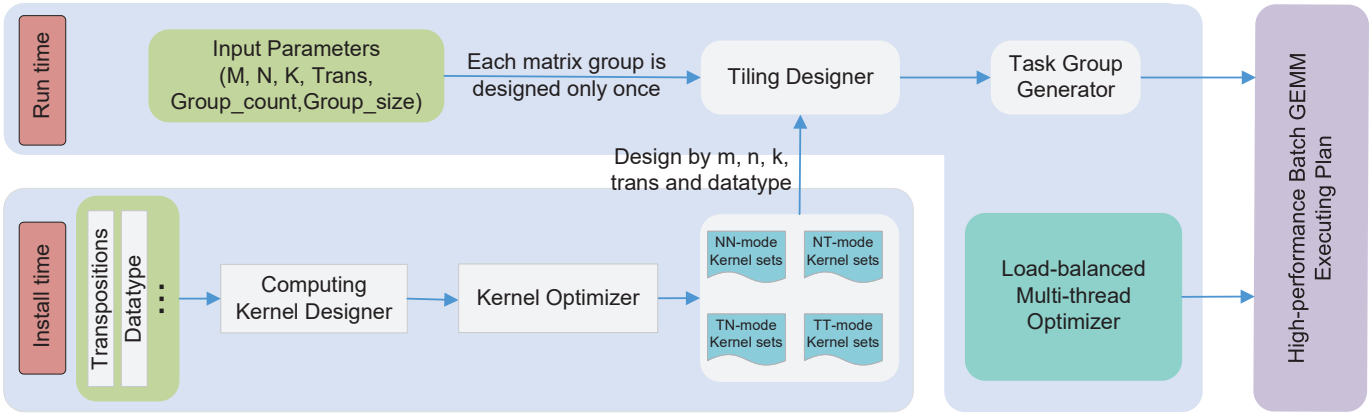


Fig. 1. Overview of Load-balanced Batch GEMM Framework.

grouping of task group generators, this scheduling method can obtain load balancing with less thread scheduling overhead. Finally, the LBBGEMM links the above strategies into a multi-thread execution plan for high-performance processing of batch GEMM on ARMv8 architecture.

We apply these proposed batch GEMM optimization methods to the Kunpeng 920 CPU [9] based on the ARMv8 architecture. Regarding single-core performance, LBBGEMM can provide up to 2.3× speedup compared to the ARMPL batch GEMM interface and achieve up to 2.4× speedup compared with the BLIS batch GEMM interface. These demonstrate that the small GEMM kernel design optimization method proposed in this paper is highly competitive. In terms of multi-core performance, LBBGEMM shows a superior speedup ratio than ARMPL and BLIS for all thread modes. Specifically, In the 48-thread test, LBBGEMM can obtain up to 4.2× performance improvement compared with ARMPL. Moreover, we can achieve 4.1× speedup compared to the BLIS. These show that our proposed group-based load balancing dynamic scheduling algorithm is extremely competitive for batch GEMM.

The key contributions of this paper are summarized as follows:

- We propose an auto-tuning algorithm for small GEMM to obtain the optimal performance for any possible input matrix property.
- We present a load-balanced multi-thread task scheduling strategy for batch GEMM to improve multi-core performance dramatically.
- We apply our design methods to a high-performance library (LBBGEMM) for batch GEMM based on the ARMv8 architecture.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the overview of the LBBGEMM. 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.

II. Related Work

A. General Matrix Multiply

In the past decades, the community has put great efforts into designing and implementing efficient BLAS libraries, mainly to obtain the ultimate high performance in large-scale computations [8], [10]. In dealing with dense matrix operations, the approach proposed by GOTO [11] has been widely adopted by mainstream linear algebra libraries such as Intel MKL, OpenBLAS [12], and ARMPL. The traditional implementation and optimization method of GEMM has three main steps: tiling, packing, and calculation. First, the matrix is divided into small blocks based on hardware features such as TLB and cache size. The tiling algorithm reduces the expensive memory access overhead by taking advantage of the multi-level cache architecture of modern computer architectures. Secondly, the matrix blocks are packed to make the memory continuously accessible to the computing kernels. Finally, the high-performance assembly kernels with careful instruction scheduling will perform the computation. However, this approach cannot utilize the full performance of the processor for small GEMM. The reasons for this are described in the next subsection.

B. Small GEMM

Small matrices pose a challenge for HPC systems because modern processors are often designed to handle large-scale data. It is difficult to take full advantage of multi-level cache structures. In extreme cases, small GEMM may not fully use vector registers, which limits the effectiveness of modern SIMD architectures. In addition, the data packing overhead and boundary processing costs cannot be neglected in small GEMM. 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 [13] 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 [14] 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. IAAT [15] removes packing operations by automatically generating hundreds of differently sized kernels during the install-time stage. The input-aware tiling algorithm is used in the run-time stage to play the role of auto-tuning. 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.

C. Batch GEMM

An efficient way to handle large numbers of small matrices is to utilize batch operations. For fixed sizes, the compact interface of the Intel MKL [16] uses a SIMD-friendly data layout that fully uses SIMD registers. IATF [17] proposes automatic tuning algorithms and code generation models for ARM architectures based on SIMD-friendly data layouts. For variable size, the community has proposed a standard interface for Batch BLAS [5]. 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 [18]. In addition, there is also a large amount of GPU-based batch GEMM optimization research in the community [19], [20], which provides ideas for the work in this paper.

Based on past work by the community, we believe that efficient batch GEMM requires high-performance small GEMM kernels and a load-balanced task scheduling method.

III. Overview of the LBBGEMM

This paper proposes a load-balanced batch GEMM framework. As shown in Figure 1, it is divided into the install-time stage and the run-time stage to achieve near-optimal performance for batch GEMM. Our implementation provides C/C++ APIs for applications, and the computing kernels use ARM assembly for ultimate high performance. In terms of interface, we refer to the mainstream batch GEMM, supporting four modes N, T, R, and C, where T and N stand for transposed and non-transposed matrices, and R and C stand for conjugate transposed and conjugate non-transposed matrices. For example, GEMM in TN mode indicates that matrix A is transposed (T), but matrix B is not (N). For the 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. In the batch interface, we refer to the latest standard. As shown in Algorithm 1, the matrices within each group have the same properties. The batch_size indicates the number of matrices in each matrix group of the same size, and the batch_count indicates the number of matrices groups.

Algorithm 1: Simplified batch GEMM

Input: A:$M\_array \times K\_array$;
B:$K\_array \times N\_array$;
C:$M\_array \times N\_array$;
$Alpha\_array, Beta\_array$;
$group\_count$:Number of groups;
$group\_size$:Number of each group;
Output: $C+ = A \times B$

```
p = 0;
for i = 0 → group_count − 1 do
    α = Alpha_array[i]
    β = Beta_array[i]
    for j = 0 → group_size[i] − 1 do
        small_gemm(A[p], B[p], C[p], α, β);
        p = p + 1;
```

The install-time stage generates highly-optimized computing kernels for the run-time stage to call. It contains the following components:

- **Computing Kernel Designer** designs the NN, NT, TN, and TT mode kernel sets without data packing. In addition, it designed kernels for every possible boundary case for each mode.
- **Kernel Optimizer** optimizes kernels from the computing kernel designer to achieve extreme performance.

The run-time stage chooses the optimal kernels according to input matrix properties. It applies tiling methods combined with the load-balanced multi-thread schedule method to generate the optimal execution plan for high-performance batch GEMM. It contains the following components:

- **Tiling Designer** divides the matrix into blocks with a minimum number of boundary processing blocks according to the input properties to generate the high-performance small GEMM execution plan.
- **Task Group Generator** divides the large group of GEMMs into small task groups for threads to compute.
- **Load-balanced Multi-thread Optimizer** dynamically assigns task groups to threads for execution through our proposed dynamic mapping of threads and tasks.

IV. The Design of Install-time Stage

The install-time stage generates small GEMM kernel sets without data-packing for each mode to obtain optimal high-performance. We point out that the high-performance small GEMM execution strategy is the key to getting the ultimate performance on batch GEMM. In our previous work [15], we introduced the design and optimization method of the small GEMM kernel in detail. This paper briefly summarizes the key design ideas. In this paper, we default the matrix to be stored in column-major.

A. Computing Kernel Designer

The computing kernel designer designs high-performance assembly kernels for small GEMM without data packing.

TABLE I
ALL GENERATED KERNELS

	NN/TT/TN	TN
SGEMM_Batch	$8 \times \{8,4,3,2,1\}$ $4 \times \{8,4,3,2,1\}$ $3 \times \{8,7,..,1\}$ $2 \times \{8,7,..,1\}$ $1 \times \{8,7,..,1\}$	$4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$
DGEMM_Batch	$8 \times \{4,3,2,1\}$ $4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$	$4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$
CGEMM_Batch	$8 \times \{4,3,2,1\}$ $4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$	$4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$
ZGEMM_Batch	$4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$	$4 \times \{4,3,2,1\}$ $3 \times \{4,3,2,1\}$ $2 \times \{4,3,2,1\}$ $1 \times \{4,3,2,1\}$

Although the mainstream BLAS libraries use the data packing strategy that allows the computing kernel to access matrices A and B consecutively, the data packing overhead is expensive when the matrix size is small [13], [15]. It is necessary to consider an efficient way to eliminate the impact of data packing. We design computing kernels without data packing by careful data loading analysis to get consistent high performance for each mode. We carefully analyzed the characteristics of the GEMM calculations in each mode to maximize the compute-to-memory-access ratio (CMAR) [21], which is important to effectively hide memory access latency for computational instructions in the micro-kernel. In addition, As shown in Table I, we design and optimize all possible kernel sizes for each transposition mode. These kernel sets are automatically selected at the run-time stage based on input parameters for the optimal execution Plan.

B. Kernel Optimizer

We have proposed a series of optimization methods to obtain the ultimate high performance in small GEMM. Our main design idea is to avoid pipeline bubbles.

An efficient way to avoid pipeline bubbles is to use the "ping-pong" operation, which divides the computation into two phases, M_1 and M_2. In phase M_1, the rows required for matrices B_c and A_c in phase M_2 are prefetched into the registers. Moreover, in phase M_2, the rows required for matrices B_c and A_c in phase M_1 are prefetched into the registers. This method provides enough space for the load instruction. In addition, within the M_1 and M_2 stages, we place the Load instruction between the compute instructions and minimize the correlation between the two compute instructions to avoid pipeline blocking.

Appropriate data prefetching operations can significantly improve performance. Since this paper uses the no data-packing strategy, the matrix A, B, and C are still in memory when the kernel loads the data. In addition, the batch GEMM studied in this paper makes each thread compute several consecutive small GEMMs. Therefore, data prefetching is necessary. We use the ARM assembly PRFM instruction at the beginning of the computing kernel for data prefetching to minimize memory access latency.

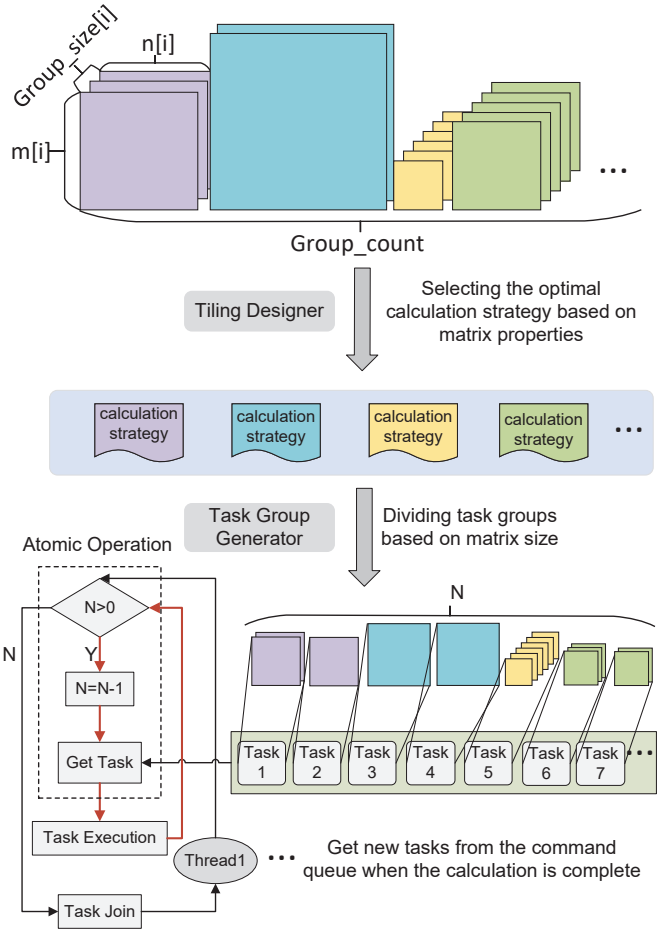


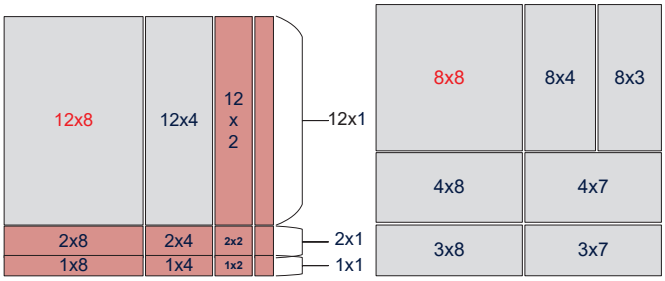
Fig. 2. Overview of multi-thread scheduling.

V. THE DESIGN OF RUN-TIME STAGE

The run-time stage presents a comprehensive auto-tuning process for batch GEMM that provides consistently high performance for every mode and size of matrices, as shown in Algorithm 2 and Figure 2. The tiling designer generates optimal execution strategies for small GEMM. The task group generator and load-balanced multi-thread optimizer dynamically allocate task groups for the ultimate multi-thread speedup ratio.

A. Tiling Designer

Appropriate tiling methods can greatly improve the efficiency of the calculation. As shown in Figure 3(a), The traditional edge processing kernel is often not fully utilized to the vector registers due to the small scale of the blocks. In addition, particularly small blocks make it difficult to hide



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

Fig. 3. Tiling method of 15×15 SGEMM.

memory access latency by using computation instructions. Therefore, we design a new tiling algorithm to reduce the generation of particularly small blocks. In addition to the main kernel, we have designed and optimized all possible boundary cases to avoid particularly small-sized blocks. As shown in Figure 3(b), we have greatly reduced the number of boundary blocks compared to Figure 3(a) and also avoid the generation of particularly small scales such as 1×1, 1×2, and 2×2. This greatly improves the performance of small GEMM processing.

For batch GEMM, although the transpose patterns of the matrices in each group are different, the patterns within the groups are the same. Therefore our tiling designer only needs to design once in each group to reduce the overhead.

B. Task Group Generator

The task group generator divides each matrix group into several smaller task groups, which would be assigned to threads. As shown in Figure 2, the tiling designer generates the optimal computing strategy based on matrix properties. These strategies ensure that the ultimate performance can be obtained without data packing. This option is run only once in each matrix group to reduce the overhead. However, directly assigning these groups to threads can cause severe load imbalance. This is because the size and number of matrices are different between the groups. This makes the amount of computation differ significantly between the groups. On the other hand, assigning individual matrices to threads incurs a huge thread scheduling overhead. When the matrix size is particularly small, and the number of threads is large, the execution time of a single thread is short, which can result in multiple threads waiting for task assignment. Based on the above analysis, we point out that the optimal grouping needs to make the computation of each task group match the hardware specifications.

$$\sum(m_i \times k_i + m_i \times n_i + n_i \times k_i) \leq L1 \ \ cache \qquad (2)$$

As shown in Equation 2, we present an upper limit on the matrix contained in each task group, and this limit is based on the matrix size and the L1 cache size. As shown in Figure 2, we store the small GEMM groups that do not exceed this limit as a task group in the form of a command queue. This allows threads to execute these commands directly without calling the small GEMM interface.

C. Load-balanced Multi-thread Optimizer

We dynamically assign task groups to threads to obtain the ultimate multi-thread speed-up ratio. The pre-grouping strategy results in little difference in the amount of computation between the task groups. However, we should note that larger matrices can achieve higher processor performance, while particularly small GEMMs can achieve only about 10% of the peak processor performance [15]. Thus, even with the same amount of computation, the task group with larger matrices will run significantly less time than the matrix group with smaller matrices. This produces a potential load imbalance problem. In this paper, we design a dynamic mapping of threads and tasks to improve multi-thread execution efficiency.

This dynamic scheduling algorithm further allows for load balancing compared to static scheduling. The kernel selection and tiling design will be allocated in command queues, reducing the resource allocation overhead of individual threads. As shown in Figure 2, it illustrates a simplified task assignment process. The number of matrices assigned to each thread is determined based on the size of each group of matrices. We indicate the current number of remaining tasks by setting the global variable N. When a thread finishes its current task, it will determine if there are any remaining executable tasks. When $N > 0$, a new group of tasks is obtained from the command queue. This process is atomic. LBBGEMM can achieve optimal multi-thread acceleration with the dynamic scheduling method based on the pre-grouping of tasks .

D. Implementation Of The Run-time Stage

Algorithm 2 shows a pseudo-code implementation of the above optimization. First, it selects the tiling strategy and computation kernels based on the input matrix properties. This selection needs to be performed only once per matrices group, as shown in lines 3-8. These kernels ensure that the ultimate performance can be obtained without data packing. Second, we compute the most appropriate 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 shown in lines 10-14. These task groups are in the form of command queues, as shown in line 13. Our implementation ensures that the final computation strategy is the best choice for command arrangement at that 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 15-28. We initialize the mutex lock, as shown in line 15. Lines 18-27 are multi-thread executions where each thread will get 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 allow us to compute batch GEMMs at high performance on state-of-the-art hardware platforms.

Algorithm 2: Auto-tuning and load-balanced of the run-time stage

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

$idx = 0, T = 0$;
for $P = 0 \rightarrow group\_count - 1$ **do**
  $tra = transa\_array[p]$;
  $trb = transb\_array[p]$;
  $m = m\_array[p]$;
  $n = n\_array[p]$;
  $k = k\_array[p]$;
  $kernel = tiling\_designer(tra, trb, m, n, k)$;
  $Ntask = L1\_cache\_size/(m \times n + n \times k + k \times m)$;
  for $i = 0 \rightarrow group\_size[p] - 1; i+ = Ntask$ **do**
    $compute\_n = min(Ntask, group\_size[p] - i)$;
    $task\_group[T] \leftarrow (kernel, compute\_n, idx)$;
    $T+ = 1$; $idx+ = compute\_n$;
$mutex.init()$;
$total\_num = T$;
$create\_threads(thread\_array, THREAD\_NUM)$;
/* Multi-thread execution */
while $T > 0$ **do**
  $mutex.lock()$;
  $T- = 1$;
  if $T < 0$ **then**
    $mutex.unlock()$;
    $break$;
  $idx = total\_num - T - 1$
  $mutex.unlock()$;
  $task\_group[idx].run()$;
$join\_threads(thread\_array, THREAD\_NUM)$;

VI. PERFORMANCE EVALUATION

We evaluate the LBBGEMM proposed in this paper on the Kunpeng 920 processor based on the ARMv8 architecture. We compared LBBGEMM with two BLAS libraries optimized for the ARMv8 architecture, of which BLIS is a widely used open-source BLAS library in the industry. ARMPL is the official performance library for ARM architecture. These libraries support the multi-thread batch GEMM interface. The evaluation includes four data types: single-precision, double-precision, single-precision complex, and double-precision complex. Each data type supports four transpositions: NN, NT, TN, and TT.

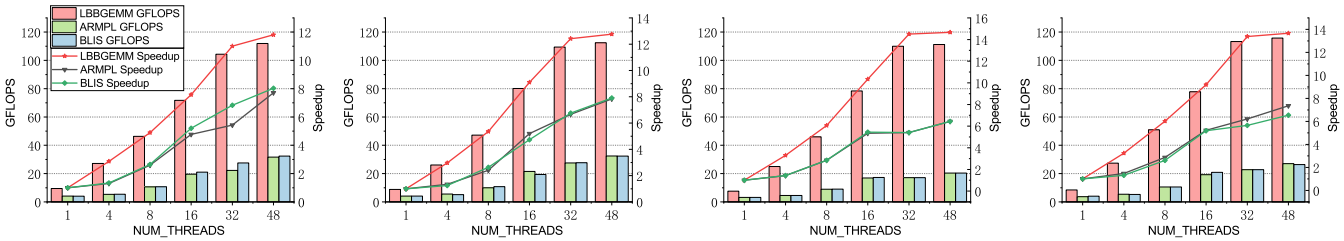
For batch GEMM, we set group_count = 4, group_size = {10000, 1000, 100, 100}, m = n = k = {10, 20, 30, 40}. We run each core 100 times and take the geometric mean as the final result. Table II shows the key specifications of the processor. The performance tests used in this paper were compiled using 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 [22]. In the multi-thread test, we compare the performance of LBBGEMM, ARMPL, and BLIS with 1, 4, 8, 16, 32, and 48 threads. In addition, the ratio of multi-thread performance to single-threaded performance is used as the multi-thread speedup ratio to compare the load balance of these three libraries.

TABLE II
EXPERIMENTAL ENVIRONMENTS

Hardware	CPU	Kunpeng 920
	Single-Core Peak perf. (FP64)	10.4GFLOPS
	Single-Core Peak perf. (FP32)	41.6GFLOPS
	Number of Cores	96
	Arch.	ARMv8.2
	Freq.	2.6GHz
	SIMD	128 bits
	L1D cache	64KB
	L2 cache	512KB
Software	Compiler	GCC7.5
	ARMPL	22.0
	BLIS	0.9.0

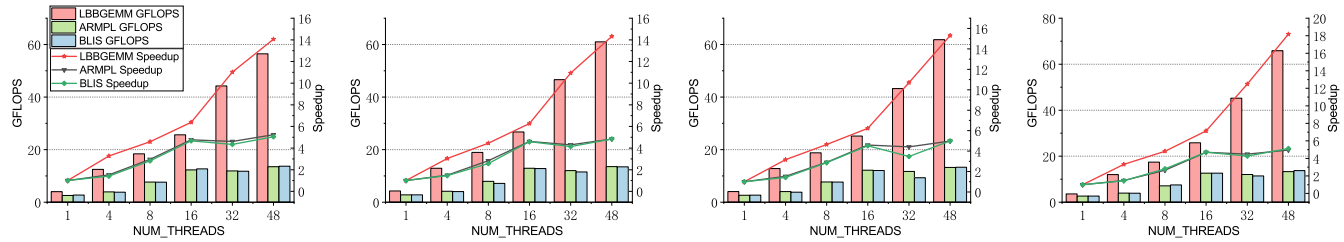
We take DGEMM as an example to analyze the multi-thread performance and speedup ratio of LBBGEMM, ARMPL, and BLIS under different threads in each transposition mode. The results show that LBBGEMM achieves a huge performance advantage at all threads. Figure 5 demonstrates our strong performance of the batch GEMM for double-precision real numbers under the NN, NT, TN, and TT modes. We compared the LBBGEMM with the ARMPL batch GEMM (ARMPL batch) and BLIS batch GEMM. As shown in Figure 5(a), in NN mode, LBBGEMM can provide 1.5×, 3.2×, 2.4×, 2.1×, 3.7× and 4.1× speedup in 1, 4, 8, 16, 32 and 48 threads, respectively, compared to ARMPL. And providing 1.5×, 3.2×, 2.4×, 2.0×, 3.7× and 4.1× speedup for the six thread modes, respectively, compared to BLIS. Compared to the ARMPL, the LBBGEMM achieves 1.5×, 3.1×, 2.4×, 2.1×, 3.9×, and 4.5× speedups for the six thread modes under NT mode. Moreover compared to the BLIS, the LBBGEMM achieves 1.5×, 3.1×, 2.4×, 2.1×, 3.9×, and 4.5× speedups for the six thread modes under NT mode, as shown in Figure 5(b). As shown in Figure 5(c), in TN mode, LBBGEMM can provide 1.5×, 3.1×, 2.4×, 2.1×, 3.7× and 4.7× speedup for the six thread modes, respectively, compared to ARMPL. And providing 1.5×, 3.4×, 2.5×, 2.1×, 4.7× and 4.4× speedup for the six thread modes, respectively, compared to BLIS. Compared to the ARMPL, the LBBGEMM achieves 1.3×, 3.0×, 2.4×, 2.0×, 3.8×, and 5.0× speedups for the six thread modes under TT mode. And compared to the BLIS, the LBBGEMM achieves 1.3×, 3.1×, 2.3×, 2.0×, 4.0×, and 4.8× speedups respectively for the six thread modes, under TT mode, as shown in Figure 5(d).

In terms of the multi-thread speedup ratio, the grouping-based dynamic scheduling algorithm proposed by LB-



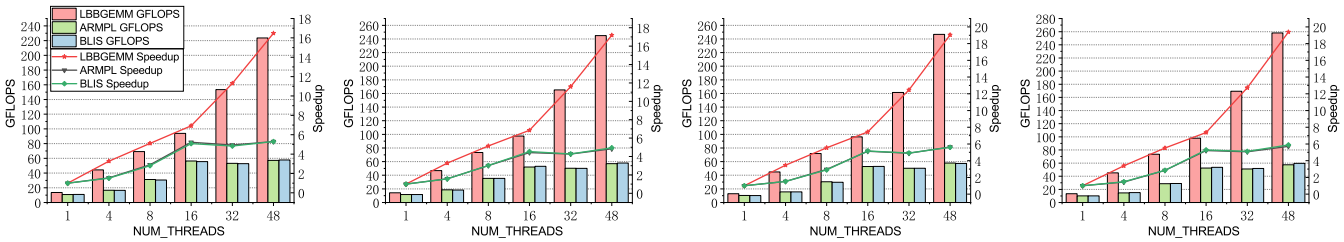
(a) SGEMM with NN mode (b) SGEMM with NT mode (c) SGEMM with TN mode (d) SGEMM with TT mode

Fig. 4. Performance of the LBBGEMM batch GEMM compared with ARMPL and BLIS under the single-precision real number.



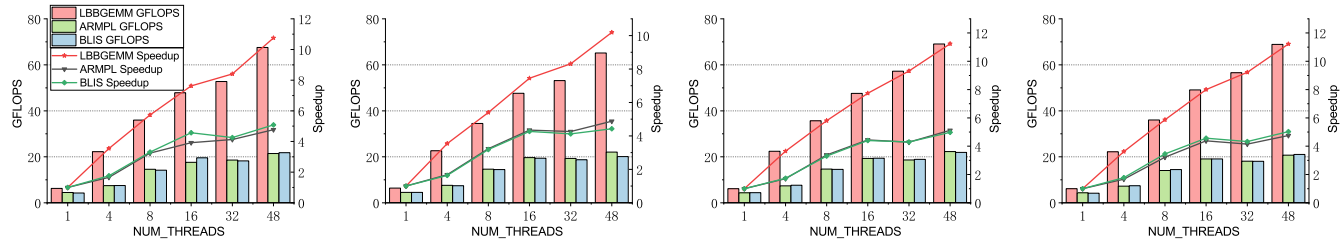
(a) DGEMM with NN mode (b) DGEMM with NT mode (c) DGEMM with TN mode (d) DGEMM with TT mode

Fig. 5. Performance of the LBBGEMM batch GEMM compared with ARMPL and BLIS under the double-precision real number.



(a) CGEMM with NN mode (b) CGEMM with NT mode (c) CGEMM with TN mode (d) CGEMM with TT mode

Fig. 6. Performance of the LBBGEMM batch GEMM compared with ARMPL and BLIS under the single-precision complex number.



(a) ZGEMM with NN mode (b) ZGEMM with NT mode (c) ZGEMM with TN mode (d) ZGEMM with TT mode

Fig. 7. Performance of the LBBGEMM batch GEMM compared with ARMPL and BLIS under the double-precision complex number.

BGEMM outperforms ARMPL and BLIS substantially in each thread. For the NN, NT, TN, and TT, LBBGEMM can provide 14.1×, 14.3×, 15.3×, and 18.2× speedups, respectively, under 48 threads. As comparisons, ARMPL achieves 5.2×, 4.8×, 5.0×, and 5.0× speedups, respectively, under 48 threads for the four modes, and BLIS achieves 5.2×, 4.8×, 5.0×, and 5.0× speedups respectively under 48 threads for the four modes.

As shown in Fig. 4, 6, and 7, the performance of SGEMM, CGEMM, ZGEMM is similar to that of DGEMM batch, LBBGEMM all achieved a great advantage. The huge advantage of LBBGEMM on a single core shows that this paper's small GEMM auto-tuning and kernel design algorithm is competitive. The LBBGEMM still has a huge advantage on the multi-core speedup ratio, which shows that the combination of group design and dynamic scheduling in this paper is highly effective in dealing with the batch GEMM problem.

VII. CONCLUSION

This paper presents LBBGEMM, a load-balanced batch GEMM framework based on the ARMv8 CPUs. We divide it into the install-time stage and the run-time stage. The install-time stage focus on the optimization of small GEMM. We designed no-packing strategies and optimized all possible boundary kernels. At the run-time stage, we implement a comprehensive auto-tuning process for batch GEMM by using the tiling designer. We divide the large matrix group into task groups, stored as command queues and dynamically assigned to threads through our proposed dynamic mapping between threads and tasks. The experimental results show that the small GEMM kernel designed in this paper is highly competitive. In terms of multi-thread speed-up ratios, LBBGEMM demonstrates significantly improved performance compared to the performance of ARMPL and BLIS.

In the future, we will focus on optimizing other BLAS routines for small-scale batch processing problems to get the ultimate performance. In addition, we will investigate and extend our approach to multi-core CPUs and GPUs.

ACKNOWLEDGMENT

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. 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.

[6] "Arm performance libraries." Website. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux/arm-performance-libraries.

[7] "Intel oneAPI Math Kernel Library." Website. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.

[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] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun, "Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services," *IEEE Micro*, vol. 41, no. 5, pp. 67–75, 2021.

[10] 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.

[11] 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.

[12] "OpenBLAS:An optimized BLAS library." Website. http://www.openblas.net/.

[13] 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.

[14] 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*, pp. 981–991, IEEE, 2016.

[15] 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.

[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] 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 *51th International Conference on Parallel Processing*, pp. 1–11, 2022.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.