2024 Design, Automation & Test in Europe Conference (DATE 2024)

Performance Analysis and Optimizations of Matrix Multiplications on ARMv8 Processors

Hucheng Liu[†], Shaohuai Shi[†‡*], Xuan Wang[†‡*], Zoe L. Jiang[†‡] and Qian Chen[†]
[†]School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
[‡]Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Shenzhen, China
Emails: liuhucheng@stu.hit.edu.cn, shaohuais@hit.edu.cn, wangxuan@cs.hitsz.edu.cn
zoeljiang@hit.edu.cn, qianchen@stu.hit.edu.cn,

***Abstract*—General matrix multiplication (GEMM) as a fundamental subroutine has been widely used in many applications like scientific computing, machine learning, etc. Although many studies are dedicated to optimizing its performance, they mainly focus on matrices with regular shapes or x86 platforms. The irregularly shaped matrices on GEMM running on modern ARMv8 processors are under-explored. In this paper, we provide a thorough performance analysis of the general block-panel multiplication (GEBP) kernel of GEMM that has irregular shapes. Based on our analysis, we propose a new GEMM algorithm named EPPA with three novel schemes to improve GEMM performance on ARMv8 processors: i) eliminating packing to reduce L1 cache contention, ii) avoiding data eviction and pre-fetching data to reduce the L1 cache miss penalty, and iii) an adaptive selection strategy of the above two and original schemes. We conduct extensive experiments with a large range of irregular matrices on three popular ARMv8 processors compared to seven state-of-the-art GEMM libraries. The experimental results show that our EPPA algorithm outperforms existing ones across workloads and processors and accelerates real-world applications.**

***Index Terms*—GEMM, ARMv8, Cache analysis, Performance Optimization**

I. Introduction

General matrix multiplication (GEMM) is a fundamental linear algebra operation widely used in scientific computing, machine learning, and many other fields [1]–[3]. It takes the form of $\mathbf{C} = \alpha\mathbf{A} \times \mathbf{B} + \beta\mathbf{C}$, where $\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{K\times N}$, and $\mathbf{C} \in \mathbb{R}^{M\times N}$ are matrices, and α and β are scalars. While GEMM has been highly optimized in popular BLAS libraries (e.g., Eigen [4], OpenBlas [5], Libxsmm [6], Blis [7], Blasfeo [8]) for high-performance computing on both x86 and ARM architectures, variant workloads from different applications that have irregular matrix shapes (e.g., one dimension is significantly smaller than the other two) may make them inefficient on ARM processors [6,9]. ARM processors have different implementation architectures (such as Mars [10], TaiShan [11], Vulcan [12], etc.) with different configurations in terms of instructions and memory.

Many real-world applications require extensive computations with small and irregularly shaped matrix-multiplications. For example, in DeepStack [13] using search-based algorithms for machine games, GEMM operations with shapes of $M \in Z^+, K = 1326, N = 1326$ dominate the overall workload of determining a strategy at each round in the Heads-Up No-Limit poker. However, calculating a matrix multiplication with $M = 5, K = 1326, N = 1326$ on a 4-core processor whose peak performance is 166.4 GFlops only achieves 17.6 GFlops, which indicates the processor computing power is significantly under-utilized. When N is smaller, it runs slower with two threads than the single-thread version on the multi-core processor KunPeng920. Such matrices are challenging to be accelerated by multiple cores and significantly under-utilize the computer power of a single core. The inefficient use of cache memory in state-of-the-art BLAS libraries causes inefficiency in irregularly shaped matrix multiplications.

In this paper, we first analyze six different libraries to determine the cause of GEMM performance degradation on irregular matrix shapes in a KunPeng920 processor. Then, we conduct a cache modeling analysis on the GEBP kernel-based algorithm to determine the causes of the high cache miss rates. After that, we propose an efficient GEMM algorithm named EPPA, which provides three novel schemes: 1) eliminating unnecessary packing operations to reduce the L1 cache contention, 2) avoiding data eviction and pre-fetching data to reduce the L1 cache miss penalty, and 3) adaptively selecting the above two schemes and the original scheme to fit for different cases of matrix shapes. We conduct extensive experiments using a wide range of input matrix shapes to demonstrate the effectiveness of our proposed schemes on three popular ARMv8 processors (i.e., KunPeng920, Graviton2, and Phytium 2000+) compared with seven state-of-the-art GEMM libraries. The experimental results demonstrate that our EPPA outperforms other libraries by 1.108× to 3.605× on a KunPeng920 processor.

Our main contributions of the paper can be summarized as (1) we propose a cache utilization analysis model to understand the performance of GEMM with irregular shapes, (2) we propose three optimization schemes, including optimizing L1 data cache contention, reducing the cost of L1 data cache miss penalty, and their adaptive selection to improve the performance on ARMv8 processors, and (3) we conduct extensive experiments to verify the effectiveness of our proposed analysis model and optimizations.

II. Background and Related Work

The highly optimized GEMM implementation can be divided into three processes: partitioning, packing, and computation [14]. (1) Partitioning: the matrix is partitioned into smaller blocks of sub-matrices in M, K, N dimensions according to

*Corresponding author.

979-8-3503-4859-0/DATE24/© 2024 EDAA

the cache size of the target hardware, such as the L1 data cache and the translation lookaside buffer (TLB). (2) Packing: the partitioned sub-matrices are stored in a contiguous form to be efficiently accessed. (3) Computation: the sub-matrices are computed by an assembly kernel that is highly optimized for the physical structure of the computational unit to produce partial results.

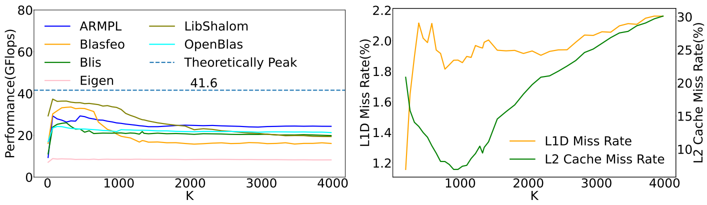
The purpose of partitioning is to split the matrix into sizes (say m_c, k_c, and n_c from the dimensions of M, K, and N respectively. They can be adaptively adjusted) that are relatively friendly to the multilevel cache structure of target processors [14,15]. The packing process is responsible for storing the data involved in the computation in a continuous form that is helpful for accessing, thus reducing the performance degradation of the computation kernel due to unready data. Boundary cases occur during partitioning when any dimension of the matrix is not an integer multiple of the kernel used in the computation. Padding the boundary case [7] or using special computational cores [6,8] are the two dominant solutions. The number of processor registers and floating-point computation units varies greatly from architecture to architecture, and the computation kernel during computation needs to be tuned according to these characteristics.

Particularly on ARMv8 processors, Wang et al., [16] propose a performance analysis model based on the architecture of the multilevel cache and register of the processor and designed a new partitioning scheme under its guidance. LibShalom redesigns partitioning, packing, and computational kernels based on the ARMv8's cache structure and the number of registers to significantly improve the performance of small-scale and irregular matrices [9]. Jiang et al., [17] propose IAAT by using LibShalom's packing and kernel parameters to perform further arithmetic optimization, which removes the packing process by generating a large number of computational kernels at installation time and improves the performance. Kung et al. [18] and Wei et al. [19] research optimizing the computation process to enhance cache utilization, specifically emphasizing tiling. However, their study fails to comprehensively explore the performance impact of data eviction triggered by matrix shape. In this work, we provide a more thorough analysis of the cache utilization and new algorithms based on the cache configurations.

III. Performance Analysis

A. The Inefficiency of Existing GEMM Implementations

We first note that specific matrix shapes may cause inefficient performance in existing GEMM libraries. We use a real-world case($M = 30$, $N = 1328$, and varying K) from DeepStack and measure the performance of GEMM on ARMv8 processors. The results (detailed configurations can be found in §V) on a KunPeng920 processor are shown in Fig. 1a. It has been observed that the performance of all six libraries decreases as the value of K increases, achieving only 10% to 60% of peak performance when K exceeds 1800. This pattern is also observed when M or N varies or runs on other ARMv8 processors like Graviton2 and Phytium 2000+.



(a) (b)

Fig. 1. GEMM on KunPeng920 with $M = 30, N = 1328$ and varying K. (a) The performance degradation; (b) The L1 and L2 cache miss rates.

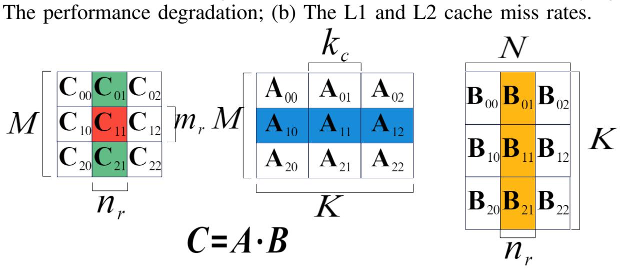


Fig. 2. A GEBP kernel-based matrix multiplication algorithm example.

To investigate the reasons for the performance degradation, we profile the cache performance on KunPeng920 using LibShalom [9] as shown in Fig. 1b, where we observe the usage of processor cache resources (i.e., L1 and L2 cache miss). It is seen that L1 and L2 cache miss rates are relevant to the performance of GEMM, which are the two main factors we need to optimize in this work.

B. Optimization Opportunities

From a single processor core point of view, the average memory access time (AMAT) [20] can be formulated as $AMAT = HT + MR \times MP$, where HT denotes the number of cycles required to reach the nearest L1 cache from the register, MR represents the L1 cache miss rate and MP indicates the number of cycles necessary to access data from other storage without hitting. For a given algorithm, the AMAT can be optimized by decreasing the L2 cache miss rate. Reducing MP of L1 cache hitlessness is a crucial factor in improving computational performance(i.e., reducing the L2 cache miss rate).

C. The GEBP Kernel based GEMM

High-performance GEMM algorithms are based on the GEBP, GEPB, or GEPDOT kernel [14]. Although kernels have different data access modes, the matrix **C** must be traversed. Fig. 2 provides a brief outline of how the GEBP kernel operates. First, **A** is divided into multiple $(m_r K)$-sized row panels along the M-dimensional direction (the blue parts in Fig. 2). Second, **B** is divided into multiple $(K n_r)$-sized column panels along the N-dimensional direction (the orange parts in Fig.2). Third, the algorithm will compute the sub-result matrix (the red parts in Fig. 2). Finally, the above process is repeated until all the sub-result matrices are computed.

D. Caching Analysis for GEBP Kernel-based GEMM

From the computation procedure introduced in §III-C, it is found that by setting a specific traversal order, the data of **A** or **B** can be reused during computation, which reduces the cache contention between matrices under the limited cache size. We analyze the caching requirements of LibShalom [9].

1) Panel Level Data Prefetching: Existing algorithms like LibShalom select to reuse the data of **B** to alleviate the L1 cache contention. Their data prefetching scheme mainly prefetches **B** and **C**. Since the access to **A** is sequential and computing each column of the resultant sub-matrices (C_{00}, C_{10}, C_{20} in Fig. 2) need to traverse the entire matrix, the prefetching of **A** is block-level (i.e., when computing C_{00}, the data of A_{01} and A_{02} will be prefetched). After each column of the result sub-matrices (e.g., C_{00}, C_{10}, C_{20}) is computed, the data of the following panel of **B** and **C** are prefetched (e.g., B_{01}, B_{11}, B_{21} and C_{01}, C_{11}, C_{21} in Fig. 2). Such a scheme may be inefficient as the L1 cache fails to hit when M or K is large, where the prefetched and computed blocks exceed the size of the L1 cache. If the required data is not in the L2 cache, it takes a long time to wait for the data to be computed. For example, computing C_{10} on a Phytium 2000+ processor takes 64.71% more time when the data is in memory than in the L2 cache.

2) L1 Data Cache Utilization: LibShalom computes **C** by columns. To reuse the data of **B**, LibShalom reorganizes the layout of **B** when computing the first submatrix block of each column of **C**. This results in different cache block requirements for computing C_{00} and C_{10} in Fig. 2. We analyze the use of the L1 data cache using C_{00} and C_{10} as examples.

Computing C_{00}. For **A**, it needs to read $m_r K$ numbers (e.g., A_{00}, A_{01}, A_{02}). For **B**, B_{i0} and B_{i1} ($i \in \{0, 1, 2\}$) are required to be read, resulting in a total of $2n_r K$ numbers. When $N > 96$, there exists packing, it is also necessary to read and write Kn_r numbers from and to the packing buffer. For **C**, it is necessary to read C_{00} and C_{01} with a total of $2m_r n_r$ numbers, and write C_{00} with a total of $m_r n_r$ numbers.

Computing C_{10}. For **A**, the amount of reading numbers is similar to that of computing C_{00}, which requires reading $m_r K$ numbers. For **B**, it only needs to read the data after packing, the size of which is Kn_r numbers; For **C**, it is necessary to read C_{00}, C_{01} with a total size of $2m_r n_r$ numbers, and write C_{00} matrices with a total size of $m_r n_r$ numbers.

In summary, the capacity of the L1 data cache used after computing a column of **C** can be theoretically expressed by

$$RB = (MK + 3Kn_r + 2Mn_r) \cdot BS_{Float}, \tag{1}$$

where n_r represents the result matrix's columns of the kernel, $N \geq 2n_r$, BS_{Float} represents the number of bytes of single-precision floating-point numbers (typically 4 bytes).

Taking the KunPeng920 processor as an example, according to Eq. (1), when $M = 30, K = 200, N = 1328$, computing one column of the result matrix fills the L1 cache. When K gradually increases, the miss rate of the L1 cache would increase, and when $K = 1672$ (If the L2 cache is unified, K will be less than this value.), the L2 cache miss rate would increase. The empirical studies are shown in Fig. 1b. It is seen that as the value of K increases, the L1 and L2 cache miss rates of computing one column of the result matrix both increase.

IV. Performance Optimizations

A. Reducing L1 Data Cache Contention

When computing the first submatrix of the result matrix by columns, LibShalom stores the panel of **B** continuously. As K increases, the matrices **A**, **B**, and **C** compete more intensively for the L1 cache. However, packing **B** does not introduce enough data reuse when M is small, which is expected to improve performance. Instead, it takes some overheads to pack the data and intensifies the competition for the L1 data cache. Therefore, when M is small (e.g., $M = m_r$), packing **B** is eliminated, significantly alleviating the competition of the L1 cache. This scheme is named 'EPPA_NP' in Algorithm 1.

B. Reducing the L1 Data Cache Missing Penalty

Based on the analysis presented in section §III-D, the prefetched data of **B** and **C** will be replaced first, along with the computed data of **A**. This means that some L1 cache misses cannot be avoided, resulting in slow data access from other memory and a performance sacrifice.

To reduce the missing penalty of an L1 cache, data can be accessed from the L2 cache instead of the main memory. This can be achieved by keeping the data in the L2 cache. There are two ways to do this - prevent cached data from being evicted from the L2 cache or prefetch evicted data. This scheme is named 'EPPA_P_L2P' in Algorithm 1.

1) Avoiding Data Eviction: LibShalom requires RB bytes of data to compute one block of **C**, as stated in Eq. (1). However, if RB exceeds the size of the L2 cache, the prefetched data of **B** and **C** is replaced by the data of **A**. It will result in an increase of the L1 data cache missing penalty.

To reduce the L2 cache miss rate, submatrices of matrices **B** and **C** for subsequent rounds should not be evicted in the current computation. But the problem is *when data eviction happens*. This issue is analyzed in two aspects by taking Fig. 2 as an example, assuming data eviction occurs when computing $C_{x0} (x \in 0, 1, 2)$.

When $x = 0$. A panel of **A**, two panels of **B**, packing buffer and two blocks of **C** with total size of $RB_{FB} = [K \cdot (m_r + 3n_r) + 2m_r n_r] \cdot BS_{Float}$ are needed to compute C_{00}. Assuming that the data fills the L2 cache, data eviction will occur when the formula $K > \frac{BS_{L2} - 2m_r n_r \cdot BS_{Float}}{(m_r + 3n_r) \cdot BS_{Float}}$ is satisfied.

When $x > 0$. A new panel of **A** and two new blocks of **C** with a total size of $RB_{OB} = m_r \cdot (K + 2n_r) \cdot BS_{Float}$ will load into the L2 cache to compute C_{x0}. When $RB_{FB} + x \cdot RB_{OB} > BS_{L2}$, data eviction will take place.

Eviction at $x = 0$ means the first block of results hasn't been computed yet. As computation proceeds, cached data gets replaced, leaving little room for optimization. Therefore, we need to focus on optimizing the case when $x > 0$, which means that the lower bound of K satisfies the constraint of Eq. (2).

$$K > \frac{\alpha \cdot BS_{L2} - 2(x + 1)m_r n_r \cdot BS_{Float}}{[(x + 1)m_r + 3n_r] \cdot BS_{Float}}, \alpha \in (0, 1], \tag{2}$$

where α indicates the intensity of competition with instruction streams and other programs in the current system, BS_{L2} denotes the size of the L2 cache.

When K is around the lower bound ($(x+1)m_r = M$), we can place a prefetch instruction in the current block computation (e.g., $A_{00} \cdot B_{00}$ in Fig. 2) to keep the next block of **B** and **C** (e.g., B_{10} and C_{10} in Fig. 2) from being evicted.

2) Data Prefetching From the Next Level of Storage: Assuming the L3 cache is inclusive mode with size BS_{L3}. When K is increased but still less than the upper bound $\frac{BS_{L3}-2Mn_r \cdot BS_{Float}}{(M+3n_r)\cdot BS_{Float}}$, data remains in L3 cache even if evicted from L2 cache. Improving performance through data prefetching requires answering the question of *when data should be prefetched.* Assuming that the target processor requires t_c cycles to finish the current computation, the problem can be analyzed using the following equations.

$$\begin{aligned} t_{preftch}^{L3\to L2} &= [k_c + m_r] \cdot \lceil \frac{n_r \cdot BS_{Float}}{BS_{CL}} \rceil \cdot \frac{BS_{CL}}{BW_{L3}}, \\ \beta &= \frac{t_c}{T_{L3}^{d} + t_{preftch}^{L3\to L2}}, \end{aligned} \tag{3}$$

where the latency and bandwidth of the L3 cache, along with the bytes number of the cache line, are represented by T_{L3}^{d}, BW_{L3} and BS_{CL} respectively. When $\beta < 1$, it is necessary to prefetch data in advance of $\lceil \frac{1}{\beta} \rceil$ rounds. When $\beta \geq 1$, one round of prefetching is sufficient.

Taking KunPeng920 as an example, when all data is in the L2 cache, it takes about 524 clock cycles to compute $A_{00} \times B_{00}$ as shown Fig. 2. Since performing random access to the L3 cache takes about 37 clock cycles [21], the number of cache lines needed to prefetch for the following computation is 13, 8 of which can be read consecutively. Thus, the total required cycles to prefetch data blocks is less than 481, much less than the computation time. Consequently, it is sufficient to prefetch one round of data to the L2 cache in advance.

C. Optimization Strategies

The optimization strategies aim to reduce competition for the L1 data cache and lower the cost of the L1 data cache non-hit. However, their applicability may vary for different matrix shapes. An in-depth analysis of the strategies is provided in the following section.

1) Adaptive Reducing L1 Data Cache Contention Strategy: The original algorithm sometimes fails to provide positive gains despite reusing the data of **B** through packing. When two panels of **B** occupy most of the space in the L1 cache, it is difficult for the gain in computation to compensate for the extra overhead introduced by packing. This case can be described by using Eq. (4).

$$\begin{aligned} MK + 2(K+M)n_r &< \frac{BS_{L1}}{BS_{Float}}, \\ MK + (K+M)N &< \frac{\gamma \cdot BS_{L2}}{BS_{Float}}, \gamma \in (0, 2], \\ K &\geq \frac{\delta \cdot BS_{L1}}{2n_r \cdot BS_{Float}}, \delta \in (0, 1], \end{aligned} \tag{4}$$

where γ is a parameter that controls the total size of the three matrices and is closely related to the replacement policy of the L1 cache, the size and bandwidth of the L2 cache; δ represents the minimum percentage of L1 cache required to store two panels of **B**. The number of float point numbers necessary to compute a result block (e.g., C_{00} in Fig. 2) without packing is $MK + 2(K+M)n_r$. It is necessary for **B** to have a large K to amortize the overhead of loading and storing the result block.

2) Reducing the L1 Data Cache Missing Penalty: When M is large, the advantages of packing outweigh its additional overhead. At this point, the main reason for the decline in computational performance is the cache miss of accessing **B** and **C**. Assuming that the target processor generally has prefetching guesses functions, we only consider applying this strategy to improve performance when $RB \geq BS_{L1}$ in Eq. (1).

As matrices have various shapes, prefetching instructions must be arranged according to the shape of the matrix. This is described in §IV-B. We will only consider the situation where the used and prefetched data use up the L2 cache when a panel of the result matrix **C** is computed. If the matrices meet the constraints shown in Eq. (5), we can arrange the prefetch instruction in the current block computation.

$$\begin{aligned} MK + (3K + 2M)n_r &> \frac{\theta \cdot BS_{L2}}{BS_{Float}}, \theta \in (0, 1], \\ \frac{KN}{MK + KN + MN} &> \varepsilon, \varepsilon \in (0, 1), \end{aligned} \tag{5}$$

where θ represents the point at which stronger competition for cache occurs, usually taking the 1; ε as the ratio of **B** in three matrices. Its value is affected by the replacement strategy used by the L2 cache, and it directly impacts the likelihood of cache data eviction. Processors typically have a value of 0.75 for ε. Larger values of K can improve performance by hiding data prefetching time.

In summary, we have integrated the two schemes above explicated in Eq. (4) and Eq. (5), and formulated our EPPA algorithm, detailed in Algorithm 1.

Algorithm 1 EPPA

Require: **A**, **B**, **C**, M, K, N, BS_{L1}, BS_{L2}, γ, δ, θ, ε
$n_r \leftarrow 16$, $TBS \leftarrow (MK + KN + MN) \cdot BS_{Float}$
$NPRB \leftarrow (MK + 2(K+M)n_r) \cdot BS_{Float}$
if $NPRB < BS_{L1}$ **then**
    $threshold \leftarrow 2Kn_r \cdot BS_{Float} - \delta BS_{L1}$
    if $TBS < \gamma \cdot BS_{L2}$ and $threshold \geq 0$ **then**
        return EPPA_NP(**C**, **A**, **B**, M, N, K)
    else
        return sgemm(**C**, **A**, **B**, M, N, K)
else
    $PRB \leftarrow NPRB + Kn_r \cdot BS_{Float}$
    if $PRB > \theta \cdot BS_{L2}$ **then**
        $threshold \leftarrow KN - \varepsilon \cdot (MK + KN + MN)$
        if $threshold > 0$ **then**
            return EPPA_P_L2P(**C**, **A**, **B**, M, N, K)
        else
            return sgemm(**C**, **A**, **B**, M, N, K)
    else
        return sgemm(**C**, **A**, **B**, M, N, K) {Scheme of LibShalom}

V. Evaluation

A. Evaluation Platforms

We choose three representative ARMv8 processors, including Phytium 2000+, KunPeng 920, and Graviton2, to evaluate the

TABLE I
THE EVALUATION PLATFORMS

	KunPeng920	Graviton2	Phytium 2000+
ISA	ARMv8.2	ARMv8.2	ARMv8.0
Core Peak. Perf	41.6 GFlops	40 GFlops	17.6 GFlops
Cores	128	2	64
Frequency(GHz)	2.6	2.5	2.2
L1 Cache	64KB	64KB	32KB
L2 Cache	512KB	1MB	2MB
L3 Cache	64MB	32MB	-
RAM	128GB	8GB	64GB
Kernel Ver.	4.19.90	5.19.0	4.19.90
OS	Kylin V10	Ubuntu 22.04	Kylin V10

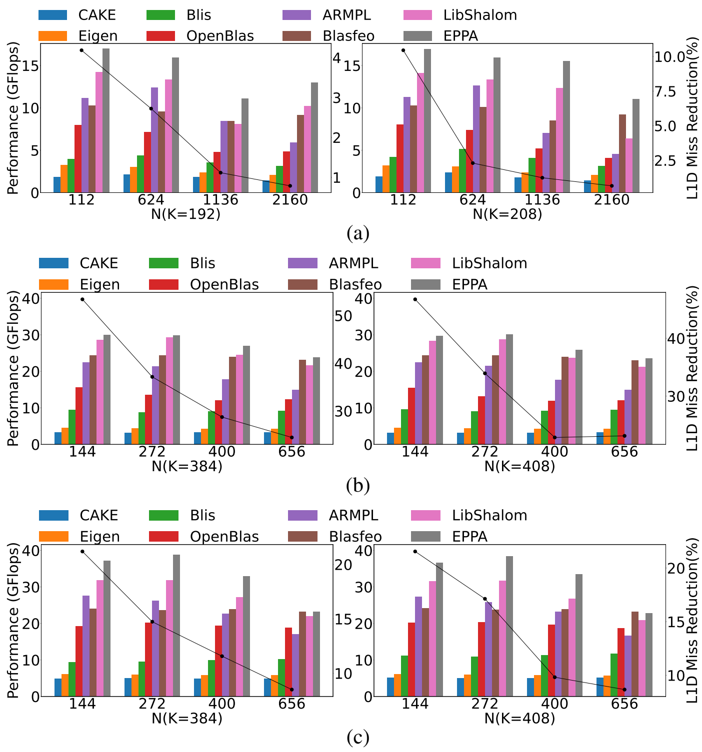


Fig. 3. GEMM Performance with irregular-shaped matrices on (a) Phytium 2000+, (b) KunPeng920 and (c) Graviton2 under reducing L1 data cache contention ($M = 5$)

performance of different algorithms, and the details of the hardware are shown in Table I. GCC optimization is set to "-O3", and performance events are recorded with "perf".

B. Performance in Reducing L1 Data Cache Contention

Experimental Setup: Our scheme performs better when fewer accesses are made to the same panel of **B** and more computation is needed to calculate a single block of the result matrix. We set parameters $\gamma = 2$ for KunPeng920 and $\gamma = 1$ for Graviton2 and Phytium 2000+ in equation (4) with $\delta = 0.75$ for all three platforms. We compare our optimization scheme with seven state-of-the-art libraries: LibShalom, OpenBlas, Blis, Eigen, Blasfeo, ARMPL, and CAKE [18].

Results: Our scheme achieves 12.317 GFlops, 27.470 GFlops, and 25.617 GFlops on Phytium 2000+, KunPeng920, and Graviton2 platforms, respectively, and the comparative results with other libraries are shown in Table II. Our scheme delivers significant speedups over other libraries on three platforms. Specifically, on the Phytium 2000+, Graviton2, and KunPeng920 platforms, the average speedup over LibShalom is 1.157×, 1.097×, and 1.074×, respectively. Additionally, the average L1 data cache miss number is reduced by 3.0%, 17.8%, and 26.7%, respectively. We speculate that the reason for the lower L1 data cache miss number reduction rate on Phytium 2000+ is related to the processor's special prefetch stride guessing mechanism. As shown in Fig. 3a, Fig. 3b and Fig. 3c, our approach EPPA effectively reduces L1 data cache misses and performs better than other libraries. When M, K is certain, as N increases, the probability of being assigned to the same cache group between the same columns inside **B** increases, leading to a lower reduction rate of L1 data cache misses.

TABLE II
THE SPEEDUPS OF REDUCING L1 DATA CACHE CONTENTION SCHEME OVER OTHER LIBRARIES ON THREE PLATFORMS

	KunPeng920	Graviton2	Phytium 2000+
CAKE	8.395×	5.161×	6.913×
Eigen	6.426×	4.408×	4.785×
Blis	3.009×	2.414×	3.287×
OpenBlas	2.256×	1.346×	2.218×
ARMPL	1.694×	1.362×	1.468×
Blasfeo	1.170×	1.136×	1.375×
LibShalom	1.074×	1.097×	1.157×

C. Performance in Reducing the L1 Cache Missing Penalty

Experimental Setup: To increase the competitiveness of test shapes, we set $\varepsilon = 0.75$ and $\theta = 1$ for KunPeng920 and Graviton2 platforms in (5). For the Phytium 2000+ platform, we observe that when the L2 cache usage exceeded 25%, there is a noticeable performance loss. Therefore, we set $\theta = 0.25$ for this platform. To limit the amount of data to be tested, we set the total size of the three matrices to $4MB$. After that, we randomly select 2000 shapes from the set of shapes that satisfy $M \ mod\ 5 = 0, K \ mod\ 8 = 0, N \ mod\ 16 = 0$ using uniform sampling.

Results: Our scheme achieves comparable or better performance than other libraries, with the performance of 26.360 GFlops, 27.978 GFlops, and 9.948 GFlops on KunPeng920, Graviton2, and Phytium 2000+ platforms, respectively, as shown in Table III. Specifically, on the KunPeng920 platform, our scheme outperforms LibShalom, Blis, OpenBlas, and ARMPL by over 20% and achieves a 71.58% L2 cache hit rate, almost four percentage points better than LibShalom. Additionally, the number of stalled-cycles-backend(the cycles wait for data readiness during instruction execution) is reduced by 69.58% compared to LibShalom. On the Graviton2 platform, our scheme has a 94.87% L2 cache hit rate, 10% higher than LibShalom's. Backend stalled cycles are reduced by 61.37% compared to LibShalom. On the Phytium 2000+ platform, our scheme shows no significant change in the L2 hit rate compared to LibShalom but reduces L1 data cache misses by 3.78%. This indicates a flawed packing selection strategy when $BS_{L1} < RB < BS_{L2}$ in the original scheme. In summary, our scheme improves the L2 cache hit rate and reduces the L1 data cache missing penalty.

TABLE III
THE SPEEDUPS OF REDUCING L1 DATA CACHE CACHE MISSING PENALTY STRATEGY OVER OTHER LIBRARIES ON THREE PLATFORM

	KunPeng920	Graviton2	Phytium 2000+
CAKE	$2.778\times$	$3.475\times$	$2.038\times$
Eigen	$3.467\times$	$2.710\times$	$3.167\times$
Blis	$1.317\times$	$1.685\times$	$1.116\times$
OpenBlas	$1.248\times$	$1.049\times$	$1.056\times$
ARMPL	$1.209\times$	$1.262\times$	$1.193\times$
Blasfeo	$1.045\times$	$0.981\times$	$1.165\times$
LibShalom	$1.269\times$	$1.217\times$	$1.128\times$

TABLE IV
THE SPEEDUPS OF EPPA OVER OTHER LIBRARIES IN TWO APPLICATIONS

	KunPeng920		Graviton2	
	DeepStack	GROMACS	DeepStack	GROMACS
CAKE	$2.637\times$	$2.843\times$	$2.203\times$	$2.558\times$
Eigen	$3.605\times$	$4.737\times$	$2.314\times$	$2.980\times$
Blis	$1.470\times$	$1.373\times$	$1.207\times$	$1.655\times$
OpenBlas	$1.314\times$	$1.634\times$	$0.930\times$	$1.578\times$
ARMPL	$1.289\times$	$1.550\times$	$1.121\times$	$1.375\times$
Blasfeo	$1.696\times$	$1.301\times$	$1.069\times$	$1.044\times$
LibShalom	$1.108\times$	$1.101\times$	$1.100\times$	$1.050\times$

D. Evaluation on Applications

Experimental Setup: We conduct performance tests on EPPA using DeepStack and GROMACS [22]. Our test data included 1814 GEMM computations from 500 games of DeepStack with Slumbot [23]. These matrices have shapes like $[x, 1326] \times [1326, 1326], x \in Z^+$. We pad up K and N to 1328 to ensure program stability. GROMACS is a free, open-source software suite used for molecular dynamics and output analysis, often in drug screening. Drug researchers typically utilize GROMACS to simulate protein systems containing around one thousand particles. They analyze the movement patterns of amino acids through the covariance moment calculation function. GROMACS covariance moment calculation is dominated by GEMM, taking up more than half of the time, and the percentage increases with the scale of the problem. Because the matrices involved in this test require much more storage than the L2 cache size of Phytium 2000+, and the Phytium 2000+ does not have an L3 cache, the Phytium 2000+ is not involved. We set $\varepsilon = 0.75$ and $\theta = 1$ in (5), set $\gamma = 2$ for KunPeng920, $\gamma = 1$ for Graviton2 and $\delta = 0.75$ in (4).

Results: The results of our algorithm's speedups over other libraries are presented in Table IV. In DeepStack, our algorithm achieves 28.975 GFlops and 29.490 GFlops on KunPeng920 and Graviton2 platforms, respectively. In GROMACS, our algorithm achieves 34.131 GFlops and 34.958 GFlops on KunPeng920 and Graviton2 platforms. Our algorithm outperforms other libraries in terms of computational performance by more than 10% on average. Some test shapes on the Graviton2 platform require more cache than the size of the L3 cache during computation, which results in fewer speedups.

VI. CONCLUSION

In this paper, we thoroughly analyzed the reasons behind the performance degradation experienced by the GEBP kernel-based matrix multiplication algorithm (e.g., LibShalom) when computing matrices of specific shapes on ARMv8 processors. We then proposed a cache optimization strategy named EPPA by combining three novel techniques: reducing data contention in the L1 data cache, reducing the cost of L1 data cache misses, and their adaptive combination. We conducted experiments on three popular ARMv8 processors compared with seven state-of-the-art BLAS libraries. The results show that our strategy improves the performance of GEMM with irregular shapes. Our EPPA applied in real-world applications, DeepStack and GROMACS, improves their performance by $1.108\times$-$3.605\times$ over existing BLAS libraries on a KunPeng920 processor.

ACKNOWLEDGMENT

This research was supported by NSFC No. 62302123 and 62376073, Guangdong Key Laboratory No. 2022B1212010005, Shenzhen Research Funding No. JCYJ20220818102414030, and CCF-Ant Research Funding (CCF-AFSG RF20220015).

REFERENCES

[1] M. Calderara et al., "Pushing back the limit of ab-initio quantum transport simulations on hybrid supercomputers," in *SC*, 2015, pp. 1–12.
[2] C. Szegedy et al., "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI*, vol. 31, no. 1, 2017.
[3] R. Mulder et al., "Fast optimisation of convolutional neural network inference using system performance models," in *Workshop on EuroMLSys*, 2021, pp. 104–110.
[4] G. Guennebaud et al., "Eigen," *URl: http://eigen.tuxfamily.org*, vol. 3, 2010.
[5] Z. Xianyi et al., "Openblas: An optimized blas library," *Accedido: Agosto*, 2016.
[6] A. Heinecke et al., "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *SC*. IEEE, 2016, pp. 981–991.
[7] F. G. Van Zee et al., "Blis: A framework for rapidly instantiating blas functionality," *TOMS*, vol. 41, no. 3, pp. 1–33, 2015.
[8] G. Frison et al., "Blasfeo: Basic linear algebra subroutines for embedded optimization," *TOMS*, vol. 44, no. 4, pp. 1–30, 2018.
[9] W. Yang et al., "Libshalom: optimizing small and irregular-shaped matrix multiplications on armv8 multi-cores," in *SC*, 2021, pp. 1–14.
[10] C. Zhang, "Mars: A 64-core armv8 processor," in *HCS*. IEEE, 2015, pp. 1–23.
[11] J. Xia et al., "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.
[12] F. Mantovani et al., "Performance and energy consumption of hpc workloads on a cluster based on arm thunderx2 cpu," *FGCS*, vol. 112, pp. 800–818, 2020.
[13] M. Moravčík et al., "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017.
[14] K. Goto et al., "Anatomy of high-performance matrix multiplication," *TOMS*, vol. 34, no. 3, pp. 1–25, 2008.
[15] A. Castelló et al., "High performance and energy efficient inference for deep learning on multicore arm processors using general optimization techniques and blis," *Journal of Systems Architecture*, vol. 125, p. 102459, 2022.
[16] F. Wang et al., "Design and implementation of a highly efficient dgemm for 64-bit armv8 multi-core processors," in *ICPP*. IEEE, 2015, pp. 200–209.
[17] J. Jiang et al., "Characterizing and optimizing transformer inference on arm many-core processor," in *ICPP*, 2022, pp. 1–11.
[18] H. Kung et al., "Cake: matrix multiplication using constant-bandwidth blocks," in *SC*, 2021, pp. 1–14.
[19] C. Wei et al., "Iatf: An input-aware tuning framework for compact blas based on armv8 cpus," in *ICPP*, 2022, pp. 1–11.
[20] W. A. Wulf et al., "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.
[21] W.-R. Gao et al., "wrbench: Comparing cache architectures and coherency protocols on armv8 many-core systems," *JCST*, p. 1, 2018.
[22] D. Van Der Spoel et al., "Gromacs: fast, flexible, and free," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1701–1718, 2005.
[23] E. G. Jackson, "Slumbot nl: Solving large games with counterfactual regret minimization using sampling and distributed processing," in *Workshops at AAAI on Artificial Intelligence*, 2013.