KirbyMM: Outer-Product Based Matrix Multiplication on ARMv9 Processor

Lanshu Huang*, Han Huang*, Zhiguang Chen†, Yutong Lu
School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
Guangdong Province Key Laboratory of Computational Science, Guangzhou, China
{huanglsh25, huangh367}@mail2.sysu.edu.cn {chenzhg29, luyutong}@mail.sysu.edu.cn

***Abstract*—General Matrix Multiplication (GEMM) serves as a cornerstone of high-performance computing and has been extensively optimized across diverse architectures. With the increasing prevalence of ARM processors in embedded systems and high-end servers, ARMv9 introduces the Scalable Matrix Extensions (SME), delivering substantially higher computational throughput compared to conventional vector SIMD units like Neon and SVE. However, existing GEMM libraries on ARMv9 still encounter critical challenges, including the lack of analytical modelling, under-utilization of SVE's capabilities, and suboptimal cache efficiency. To address these issues, we propose *KirbyMM*, a general and portable implementation for GEMM optimization on ARMv9 architecture. *KirbyMM* presents three key contributions: 1) *BiReg-CMR*, an analytical model that fully exploits SME's potential; 2) SME-SVE hybrid routine tailored for edge cases; and 3) cache-friendly data packing and partitioning strategies that enhance data locality. Experimental results demonstrate that *KirbyMM* achieves speedups of 1.11x - 1.75x on general matrix sizes compared to vendor libraries across different CPU platforms, and up to 3.59x on edge cases.**

***Index Terms*—Matrix Multiplication, ARMv9, Scalable Matrix Extension**

I. Introduction

General Matrix Multiplication (GEMM) is a fundamental computing kernel in a plethora of applications, including classical numerical simulations [1–3], machine learning [4–7], as well as the emerging Large Language Models [8, 9]. Therefore, numerous works have proposed varied methods to optimize GEMM on mainstream architectures [10–14].

The ARM architecture has gained significant prevalence across various computing platforms, from embedded devices (e.g., Apple M4 [15]) to high-performance servers (e.g., LX2 and NVIDIA GB200 [16]). This widespread adoption, coupled with the growing demands of machine learning and artificial intelligence, has made the optimization of GEMM on ARM architectures a critical research focus. Previous works [10, 17–20] accelerating GEMM on ARM architectures have mostly focused on the conventional one-dimensional (1D) Single Instruction Multiple Data (SIMD) paradigms, such as Neon [21] and SVE [22]. 1D-SIMD applies a single instruction to a 1D vector of data, and improves the performance of GEMM.

Besides vector units, ARMv9 introduces the Scalable Matrix Extension (SME) [23], which has been adopted by cutting-edge processors including LX2 and Apple M4 [15]. SME generalizes SIMD by extending 1D vector operations into two-dimensional matrix operations, thus improves the performance by as many as 4×. Taking full advantages of the SME is a promising way to optimize GEMM, but exploiting the performance of SME is non-trivial.

Existing GEMM libraries such as LibXSMM [18], Kunpeng-BLAS (KBLAS) [24] and Apple Accelerate [25] have provided initial support for SME in their latest versions. However, these libraries either lack performance portability or fail to exploit SME's potential. Specifically, the following three issues remain unaddressed. **1) Lack of analytical modelling.** Studies on heterogeneous ARMv9 with both vector and matrix units remain scarce, and feasible solutions for its diverse implementations are still lacking. **2) Under-utilization of the SVE capabilities.** Existing libraries rely solely on SME for GEMM optimization, overlooking the potential of the cooperation of SVE and SME, especially in edge cases. **3) Low cache hit rates.** L1 cache hit rates drop sharply as the matrix size increases, leading to performance degradation.

Building on these observations, we present ***KirbyMM***, a general and portable implementation for optimizing GEMM on the ARMv9 architecture. At the **MircoKernel** level, we introduce an analytical model and instruction scheduling techniques, incorporating both SME and SVE units efficiently. At the **MacroKernel** level, we apply ARMv9-optimized data packing and partitioning strategies to enhance cache locality. Our main contributions can be summarized as follows:

1) We propose *KirbyMM*, a novel GEMM implementation that efficiently leverages SME. It introduces an analytical model considering instruction throughput, latency, and issue width to better explore both SME and SVE.
2) We further design a dedicated SME-SVE hybrid routine that optimizes edge cases by integrating matrix and vector instructions, together with cache-friendly data packing and partitioning strategies.
3) Extensive experiments confirm that *KirbyMM* substantially outperforms vendor libraries across the Apple M4 and LX2 CPUs, and improve cache efficiency significantly.

II. Background and Related Work

A. Matrix Unit in ARMv9

The Scalable Matrix Extensions in ARMv9 architecture introduce dedicated two-dimensional *ZA* (Z Array) [26] tiles

* Both authors contribute equally.
† Corresponding author.

978-3-9826741-1-7/DATE26/© 2026 EDAA

as matrix registers. ZA tiles support various precisions from INT8 to FP64, and are configured into four 16×16 tiles in FP32 computation. These tiles, together with vector registers, provide a substantial register file of 6144 bytes, which is 3 times larger than the 2048-byte register file in ARMv8 architecture. The expanded capacity enables more in-place data accesses, thus reducing the loads/stores to memory significantly.

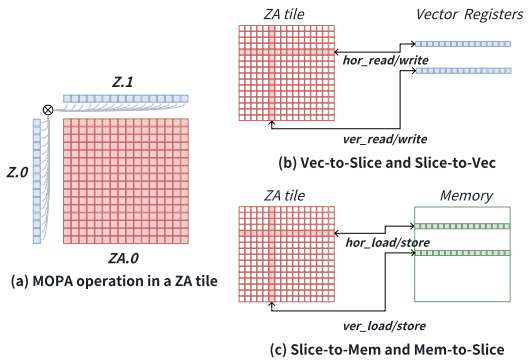


Fig. 1: The features of ZA tiles in ARMv9.

Based on *ZA* tiles, SME introduces the Matrix Outer Product Add (MOPA) instruction, which computes the outer product of two vectors and accumulates the result into a *ZA* tile (see Fig.1 (a)). Aided by the MOPA instructions, the LX2 CPU delivers up to 0.512 TFLOPS (FP32) per core, which is four times higher than the 0.128 TFLOPS achieved by SVE Multiply–Add (MLA) instruction. Besides computation, SME also provides efficient data movement between matrix registers, vector registers, and memory (see Fig. 1(b),(c)), through two mechanisms: **Slice-to-Vec/Vec-to-Slice** for moving data between *ZA* slices and vector registers, and **Mem-to-Slice/Slice-to-Mem** for exchanging data between memory and *ZA* slices. Both support horizontal or vertical forms and have identical latency and throughput.

B. General Matrix Multiplication Algorithm and Method

Goto's algorithm [27] is a classical and widely adopted approach to conduct matrix multiplication, forming the foundation of high-performance libraries such as BLIS [28] and OpenBLAS [14]. As shown in Algorithm 1 and Fig. 2, this approach can be divided into two main phases:

- **MacroKernel**(Loops *L1-L3*)
  Partition the matrices A, B, C into sub-matrices, and pack them into buffers for contiguous access and better memory locality [29, 30].
- **MicroKernel**(Loops *L4-L5*)
  Compute the k-update of the $blk_n \times blk_m$ sub-matrices. Leverage optimal register blocking strategy to maximize compute throughput and minimize memory access overhead.

Building upon Goto's algorithm, numerous efforts have sought to optimize matrix multiplication. Wang [10] et al. introduce a computation–memory ratio (CMR) performance model targeting GEMM. Following this, Yang et al. propose

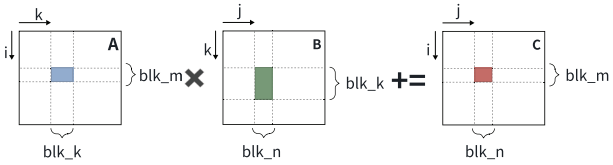


Fig. 2: Illustration of GOTO's algorithm.

Algorithm 1: GOTO's Algorithm

Input: Matrices $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$
Output: Matrix $C \in \mathbb{R}^{M \times N}$

1 **for** $j \leftarrow 1$ ***to*** N ***by*** blk_n **do** // L1
2 **for** $k \leftarrow 1$ ***to*** K ***by*** blk_k **do** // L2
3 $b_B = B(k : k + blk_k - 1, j : j + blk_n - 1)$ **for** $i \leftarrow 1$ ***to*** M ***by*** blk_m **do** // L3
4 $b_A = A(i : i + blk_m - 1, k : k + blk_k - 1)$
  for $jj \leftarrow 1$ ***to*** blk_n ***by*** s_j **do** // L4
5 **for** $ii \leftarrow 1$ ***to*** blk_m ***by*** s_i **do** // L5
6 $b_C[i + ii : i + ii + s_i,\ j + jj : j + jj + s_j] += b_A[ii : ii + s_i,\ :] \cdot b_B[:,\ jj : jj + s_j]$

LibShaLom [18], which dynamically adapts data packing and exploits FMA-SIMD scheduling to reduce overhead. For edge cases, methods include padding [28], dedicated routines [14], automatic framework [19], and partitioning [11, 12].

C. ARMv9 architecture implementation

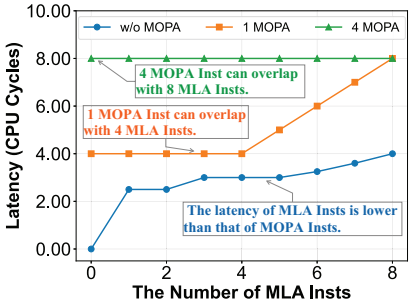


Fig. 3: Instruction-level parallelism tests on LX2 CPU.

Currently, the LX2 and Apple M4 CPUs are the newly-released ARMv9 CPUs with SME, but with distinct designs in both instruction sets and micro-architecture. LX2 implements SME, while M4 adopts SME2 [31] to enable more features such as contiguous multi-vector loads. For the micro-architecture, LX2 separates SVE and SME units and enables SME-SVE overlap (see Fig. 3), whereas M4 lacks efficient SVE units and leverages SME units for vector processing. Aided by the separation on the LX2 CPU, the low-latency SVE instructions can overlap with SME execution, a behavior absent on M4.

III. Motivation and Overview

Although prior works have provided support for ARMv9 architecture, fully utilizing its capabilities presents unique challenges. LIBXSMM [32, 33] performs well on small matrices, but falls short on large ones (see Fig. 4(b)). KBLAS [24] and Accelerate [25], as vendor libraries, deliver only mediocre performance across different sizes of matrices, primarily due to: 1) Lack of analytical modelling; 2) Ignorance of the SVE capabilities; 3) Low cache hit rates.

A. Lack of analytical modeling

Prior studies [10, 17–20] have extensively explored the ARMv8 architecture. However, research on the cutting-edge ARMv9 architecture featuring both vector and matrix units remains limited. In particular, there is a lack of feasible solutions for handling the diverse implementations of ARMv9 architectures, such as LX2 and M4 CPUs. Moreover, on ARMv9 architecture, factors such as instruction throughput, latency, issue width are critical, as they directly influence the utilization of SME and ultimately dictate achievable performance. Consequently, there is a clear need for an analytical model that integrates these factors to provide a more accurate and comprehensive characterization.

B. Under-utilization of the SVE capabilities

Existing libraries implement GEMM using SME alone, under-utilize SVE. Although the throughput of SME is up to four times higher than that of SVE, in certain scenarios such as edge cases, SME may underperform. Such limitations suggest the necessity of hybrid strategies, and some works have explored: for instance, Ktransformer [34] combines AVX [35] and AMX [36] in attention kernel, leveraging AMX for compute-bound prefill and AVX for memory-bound decode. Likewise, HStencil [37] proposes a hybrid SME-SVE approach for stencil kernels. These examples underscore the importance of exploring how SME can be effectively optimized in conjunction with SVE capabilities.

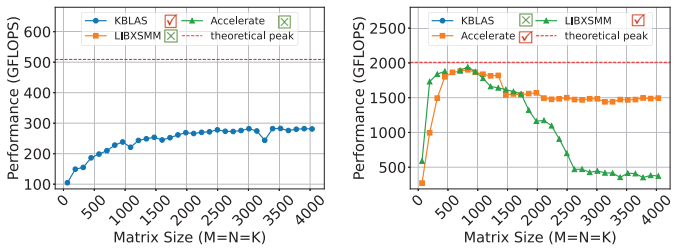
TABLE I: Cache metrics of vendor libraries.

Method	Size	L1 cache load times	L1 cache load hit rate	L1 cache load miss times
KBLAS	1024^3	1.06×10^8	81.79%	1.93×10^7
	2048^3	4.66×10^8	71.82%	1.31×10^8
	4096^3	2.59×10^9	62.00%	9.84×10^8
Accelerate[1]	1024^3	N/A	N/A	6.39×10^2
	2048^3	N/A	N/A	2.64×10^6
	4096^3	N/A	N/A	9.49×10^6

[1]The Xcode profiling tool (Instrument) on Apple platforms does not support measurements of L1 cache load times and hit rates.

C. Low cache hit rates

KBLAS and Accelerate are vendor libraries respectively on LX2 and M4 CPUs, yet they both face significant cache misses. As presented in Table I, KBLAS exhibits a sharp decline in L1 cache hit rate as the matrix size increases, decreasing from 81.79% to 62%. For Accelerate, a similar trend is observed where the L1 cache misses increase significantly as the matrix size exceeds 1024^3. This phenomenon accounts for the results observed in Fig. 4, where the two libraries demonstrate substantial potential to further approach their peak performance. The performance gap can be attributed to the absence of macrokernel design explicitly optimized to enhance cache utilization.



(a) On LX2 CPU (b) On Apple's M4 CPU

Fig. 4: Performance of existing ARMv9 GEMM libraries.

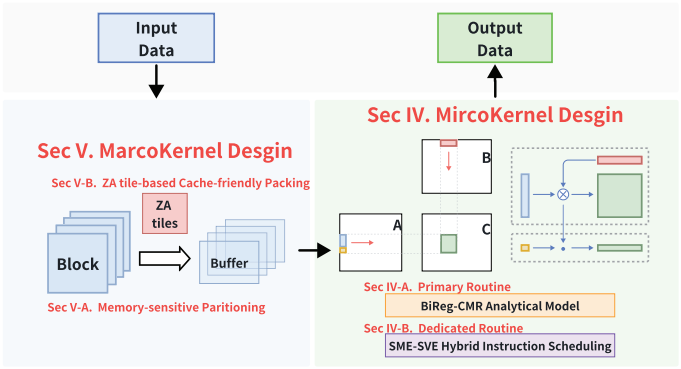


Fig. 5: Overview of the *KirbyMM* implementation.

To overcome these challenges, we present ***KirbyMM***, a general and portable implementation for GEMM. As shown in Fig. 5, *KirbyMM* comprises two main components: 1) a MacroKernel with memory-sensitive partitioning and ZA tile-based cache-friendly packing, and 2) a MicroKernel with both primary routine and dedicated routine.

IV. MicroKernel Design

KirbyMM employs two complementary MicroKernels. The first (primary routine) handles matrix multiplication in general matrix sizes, while the second (dedicated routine) is optimized for small matrices and edge cases. To design these kernels, we introduce ***BiReg CMR***, a novel analytical model that incorporates register allocation for both matrix and vector units.

A. Primary Routine

1) Constraint on register resources: To conduct register allocation for both matrix and vector units, we first analyze their behaviors: **SME units** employ the *mopa* instruction to directly compute outer-products between operands from matrices A and B, and accumulates intermediate results in ZA tiles (see Fig. 1). **SVE units** rely on the *mla* instruction to emulate the outer-product through "broadcasting", where a scalar element from

matrix A is broadcast across vector lanes and multiplied with corresponding elements form matrix B in parallel (see Fig. 6).

Therefore, for **SME-*mopa* operands**, we allocate $\frac{mr_1}{vl}$[1] vector registers to hold elements from matrix A and $\frac{nr}{vl}$ vector registers for B. Each *mopa* operand accumulates partial products into a $\frac{mr_1}{vl} \times \frac{nr}{vl}$ ZA tile block for C. For **SVE-*mla* operands**, we allocate mr_2 vector registers to hold elements from matrix A, and reuse the same B-elements loaded for SME-*mopa* operands. These *mla* operations accumulate into a $mr_2 \times \frac{nr}{vl}$ array of vector registers, completing the remains of the output matrix C.

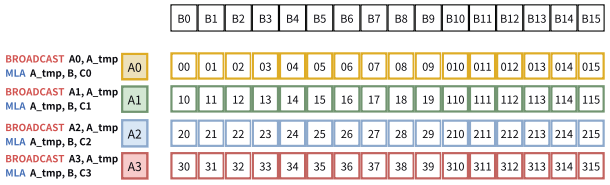


Fig. 6: SVE k-update with broadcast. An example to compute 4×16 outer-product of C.

To ensure that all operands fit within the available vector (32 in total) and matrix (4 for FP32) registers, the parameters mr_1, mr_2, and nr must satisfy the following constraints:

$$\begin{cases} \frac{mr_1}{vl} + mr_2 + \frac{nr}{vl} + \frac{mr_2 \times nr}{vl} \leq 32 \\ \frac{mr_1}{vl} \times \frac{nr}{vl} \leq 4 \\ mr_1 \bmod vl = 0 \\ nr \bmod vl = 0 \end{cases} \tag{1}$$

The primary routine contains no edge cases and thus can fully utilize the matrix units. Therefore, both mr_1 and nr must be exact multiples of vl (i.e. $mr_1 \% vl = 0$ and $nr \% vl = 0$).

2) Optimization Objective: For each iteration, we require $\frac{mr_1}{vl}$ load instructions to fetch elements from matrix A and $\frac{nr}{vl}$ load instructions to fetch elements from matrix B for the *mopa* operation. In addition, mr_2 load instructions are needed to fetch elements from matrix A for the *mla* operation.

In terms of computation, the *mopa* instruction contributes to $\frac{mr_1}{vl} \times \frac{nr}{vl} \times vl^2$ matrix elements, and the *mla* instruction contributes to $m_2 \times \frac{nr}{vl} \times vl$ matrix elements. Since the *mopa* and *mla* instructions perform fused multiplication-accumulation, there are a total of $2 \times (\frac{mr_1}{vl} \times \frac{nr}{vl} \times vl^2 + mr_2 \times \frac{nr}{vl} \times vl)$ floating-point operations per iteration. Combining memory access and computation, the BiReg-CMR of our MicroKernel is given by:

$$CMR = \frac{2 \times (\frac{mr_1}{vl} \times \frac{nr}{vl} \times vl^2 + mr_2 \times \frac{nr}{vl} \times vl)}{\frac{mr_1}{vl} + mr_2 + \frac{nr}{vl}} \tag{2}$$

3) Solving the equations: To determine the values for mr_1, mr_2 and nr that maximize the CMR, we solve the objective of maximizing the CMR defined in Eq. 2, with the constraints defined in Eq. 1. This process yields optimal values of $mr_1 = 32$, $mr_2 = 32$ and $nr = 0$, which are used in our primary MicroKernel implementation for the ARMv9 architecture.

[1] vl denotes the vector length (16 for FP32)

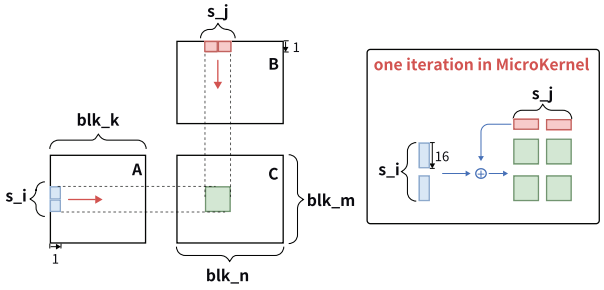


Fig. 7: The primary routine of MicroKernel for FP32.

Fig. 7 illustrates the primary routine of the MicroKernel for FP32. In each iteration, two vector registers are used to load elements from matrix A, and another two vector registers load elements from matrix B to perform an outer product. The results are then accumulated into a 2×2 block of ZA tiles, and so forth.

B. Dedicated Routine

The primary routine targets large, out-of-cache matrices by maximizing CMR through reduced memory accesses and higher computational intensity. For small or edge-case matrices that fit in cache, memory overhead is negligible, and performance instead depends on instruction scheduling and execution overlap. To handle these, we introduce the dedicated routine.

In the dedicated routine, we add an optimization objective (Eq. 3) to minimize T_{compute}, the total CPU cycles of SME and SVE execution. By leveraging the interleaved execution of SME and SVE instructions (Section II-C), this design improves pipeline utilization and computation intensity.

$$\begin{aligned} T &= \max(T_{\text{compute}}, T_{\text{load}}) \\ T_{\text{compute}} &= T_{non\_overlap} + T_{overlap} \end{aligned} \tag{3}$$

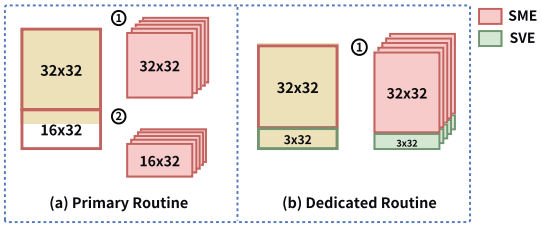


Fig. 8: Comparison of the tiling strategy for matrix C with dimensions $M = 35$ and $N = 32$.

Fig. 8 compares primary and dedicated routines. In the primary routine (Fig. 8(a)), tiling with 32×32 and 16×32 blocks yields a theoretical CMR of 512, but only 304 is effectively achieved because the 16×32 blocks underutilized vector lanes. In contrast, the dedicated routine achieves 203 CMR, but fully exploits vector-lane capacity. As shown in Fig. 8(b), workloads are aligned with SME and SVE unit widths.

Overhead Analysis. Since SVE and SME instructions can execute concurrently, only step ① in Fig. 8(b) is required,

whereas the primary routine performs both ① and ②. The overhead is:

$$\begin{aligned} T_a &= \max(T_{4\times mopa+2\times mopa}, T_{load_a}) \\ T_b &= \max(T_{4\times mopa+6\times mla}, T_{load_b}) \\ T_{4\times mopa+6\times mla} &= T_{4\times mopa} < T_{4\times mopa+2\times mopa} \end{aligned} \tag{4}$$

As shown in Fig. 3, SME and SVE operations interleave, so $T_{4\times mopa+6\times mla} = T_{4\times mopa}$. Thus, the dedicated routine achieves lower overhead than the primary routine in edge cases.

V. MacroKernel Design

KirbyMM adopts partitioning and packing techniques to optimize memory access efficiency. Building upon the ARMv9 characteristics, we further introduce memory-sensitive partitioning along with cache-friendly packing strategies.

A. Memory-sensitive Partitioning

Partitioning aims to identify the optimal values of blk_m, blk_n, and blk_k that maximize data locality. We determine these values guided by the roofline model[38] and cache hierarchy analysis.

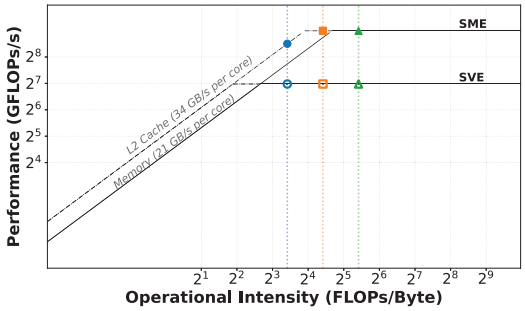


Fig. 9: Roofline model on LX2 CPU.

Roofline Model Analysis. SME offers significantly higher throughput than SVE, which may shift the performance bottleneck to memory bandwidth. As the roofline model illustrates (Fig. 9), the elevated throughput of SME renders dense GEMM operations memory-bound. Consequently, selecting a partitioning size whose computational intensity exceeds the machine's maximum attainable operational intensity is critical.

Operational intensity is defined as $I = \frac{\text{total\_arithmetic\_op}}{\text{total\_data\_movement}}$. Multiplying a $blk_m \times blk_k$ block of A by a $blk_k \times blk_n$ block of B requires $2 \times blk_m \times blk_k \times blk_n$ floating-point operations, while loading $(blk_m \times blk_k + blk_k \times blk_n) \times 4$ bytes of single-precision data. To reach peak performance rather than remain in memory-bound region, the operational intensity must satisfy[2]:

$$\begin{aligned} I_{ours} = \frac{2 \cdot blk_m \cdot blk_k \cdot blk_n}{4 \cdot (blk_m + blk_n) \cdot blk_k} = \frac{blk_m \cdot blk_n}{blk_m + blk_n} \geq I_{max} \\ I_{max} = \frac{P_{peak}}{B_{max}} \end{aligned} \tag{5}$$

which in return constraints the choices of blk_m, blk_n, blk_k.

Cache Hierarchy Analysis. Partitioning sizes also depend on the cache hierarchy and must allow all working sets to fit in cache. For efficient L2 cache utilization, a $blk_m \times blk_k$ block of matrix A is staged in each iteration of Loop $L4$, while the cache must also hold two $blk_k \times s_j$ panels of B and one $blk_m \times s_j$ panel of C under LRU replacement policy. This leads to the L2 cache constraints:

$$\begin{aligned} \lceil \frac{blk_m \cdot blk_k \cdot bpe}{S_{L2} \cdot L_{L2}} \rceil + 2 \cdot \lceil \frac{blk_k \cdot s_j \cdot bpe}{S_{L2} \cdot L_{L2}} \rceil \\ + \lceil \frac{blk_m \cdot blk_j \cdot bpe}{S_{L2} \cdot L_{L2}} \rceil \leq A_{L2} - 1 \end{aligned} \tag{6}$$

Here, S_{L2}, L_{L2}, and A_{L2} denote the number of cache sets, cacheline size, and associativity of the L2 cache, respectively. The term bpe refers to the number of bytes per element. By reserving one way per set, we ensure that unpacked C elements never evict the packed A and B tiles.

For L1 cache, in each iteration of L_5, a $blk_k \cdot s_j$ tile of B must remain resident in the L1 cache. To avoid eviction due to conflicting accesses to A and non-contiguous C, the L1 cache constraint becomes:

$$\lceil \frac{blk_k \cdot s_j \cdot bpe}{S_{L1} \cdot L_{L1}} \rceil + 2 \cdot \lceil \frac{s_i \cdot blk_k \cdot bpe}{S_{L1} \cdot L_{L1}} \rceil + \lceil \frac{s_i \cdot s_j \cdot bpe}{S_{L1} \cdot L_{L1}} \rceil \leq A_{L1} - 1 \tag{7}$$

Solving these three constraints jointly (Eq. 5, 7 and 6) yields the feasible values for blk_m, blk_n, and blk_k.

B. ZA-tile-based Cache-friendly Packing

During packing, matrices A, B, and C may be stored in either row-major or column-major format. In some cases, a row-major matrix block should be packed into a column-major buffer for contiguous access in MicroKernel. Although the SVE *gather* instruction (Fig.10(a)) can load strided elements with a single operation, its inherently non-contiguous pattern incurs significant CPU latency and cache misses. To mitigate these overheads, we leverage the ZA tile as a programmable two-dimensional cache (Fig.10(b)). Specifically, the block is first loaded into the ZA tile row by row, ensuring contiguous memory accesses. Then, by reading in a column-wise manner, we perform an implicit transpose and extract contiguous column-wise data into the destination buffer.

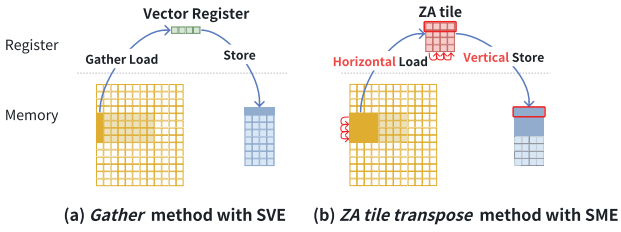


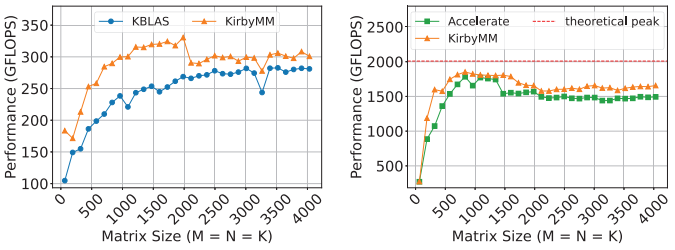
Fig. 10: Illustration of packing a matrix block stored in row-major order into a column-wise linear buffer.

VI. Evaluation

In this section, we evaluates *KirbyMM* on two newly-released ARMv9 processors, the LX2 and Apple M4. Binaries are compiled with `-O3` using *Clang v17.0* on LX2 and *Clang v16.2* on M4. Performance metrics are collected with `perf` on LX2 and *Xcode Instruments* on M4.

[2] P_{peak} is the peak compute throughput and B_{max} is the peak memory bandwidth.

A. Performance Improvement



(a) On LX2 CPU (b) On Apple's M4 CPU

Fig. 11: Performance comparison of *KirbyMM* versus vendor implementations.

The performance of general matrix sizes ranging from 64 to 4096 is presented in Fig. 11. *KirbyMM* achieves up to 330.49 GFlops and 1851.76 GFlops on LX2 CPU and M4 CPU, delivering speedups of approximately 1.17x-1.75x and 1.11x-1.49x speedup over the vendor-provided LX2's KBLAS and M4's Accelerate.

Besides GEMM performance, Table II reports cache metrics. On the LX2 CPU, *KirbyMM* consistently achieves approximately 90% L1 hit rates, while KBLAS drops sharply on larger matrices. On the M4 CPU, although Accelerate has fewer misses for small cases (e.g., 1024^3), its misses grow from 6.39×10^2 to 9.49×10^6 as size increases. In contrast, *KirbyMM* maintains stable cache behavior and reduces L1 misses by nearly **10x** at large sizes, demonstrating superior cache efficiency.

TABLE II: Cache metrics comparison.

Method	Size	L1 cache load times	L1 cache load hit rate	L1 cache load miss times
KirbyMM	1024^3	1.36×10^8	92.11%	1.07×10^7
	2048^3	5.80×10^8	90.87%	5.3×10^7
	4096^3	2.41×10^8	89.70%	2.48×10^7
KBLAS	1024^3	1.06×10^8	81.79%	1.93×10^7
	2048^3	4.66×10^8	71.82%	1.31×10^8
	4096^3	2.59×10^9	62.00%	9.84×10^8
Accelerate	1024^3	N/A	N/A	6.39×10^2
	2048^3	N/A	N/A	2.64×10^6
	4096^3	N/A	N/A	9.49×10^6

Apart from the primary routine, we also conduct an evaluation of the dedicated routine in several edge cases. (see Fig. 12) The dedicated routine is compared against both the vendor implementation and the primary routine of *KirbyMM*. The results demonstrate that *KirbyMM* achieves an average speedup of 3.59x over KBLAS and 1.36x over the primary routine.

B. Performance Breakdown

Fig. 13 presents a component-level breakdown for the optimizations. The MircoKernel delivers baseline performance of 32.82-1425.13 GFLOPS on M4 and 105.24-250.63 GFLOPS on LX2. Packing optimization provides a speedup of 1.26x

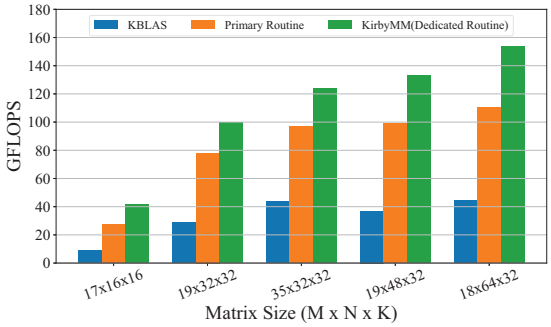
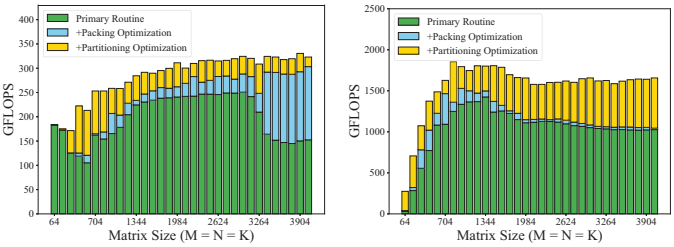


Fig. 12: Edge-case performance on the LX2 CPU.



(a) On LX2 CPU (b) On Apple's M4 CPU

Fig. 13: Performance breakdown across CPU platforms.

on LX2 and 1.09x on M4, while partitioning optimization achieves 1.21x on LX2 and 1.59x on M4. On LX2 CPU, most of the performance gain comes from packing optimization, indicating that memory bandwidth and data layout are the primary bottlenecks of this architecture. This also explains why the contribution of the MircoKernel diminishes as matrix size increases, since the execution becomes memory-bound. For the M4 CPU, the major performance improvements are attributed to partitioning strategies. Overall, these results highlight that different architectures benefit from different optimizations, yet our approach consistently delivers substantial speedups, showcasing its effectiveness and performance portability.

VII. Conclusion

This paper presents *KirbyMM*, a novel GEMM implementation for ARMv9. *KirbyMM* introduces the *BiReg-CMR* analytical model to design a high-performance MircoKernel, imultaneously leverages both SME and SVE units to accelerate small and edge-case matrices. It also employs cache-friendly packing and partitioning strategies to improve data locality. Extensive experiments confirm that *KirbyMM* achieves superior cache efficiency and outperforms state-of-the-art libraries.

VIII. Acknowledgement

This work is Supported by Guangdong S&T Program underGrant NO.2025B0101080001, the National Natural Science Foundation of China (NSFC) under Grant NO.62272499, the Guangdong Province Special Support Program for Cultivating High-Level Talents under Grant NO.2021T006X160, and PazhouLab under Grant NO.PZL2023KF0001.

REFERENCES

[1] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros, "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[2] D. Wu, P. Chen, X. Wang, I. Lyngaas, T. Miyajima, T. Endo, S. Matsuoka, and M. Wahib, "Real-time high-resolution x-ray computed tomography," in *ICS '24*. New York, NY, USA: Association for Computing Machinery, 2024, p. 110–123. [Online]. Available: https://doi.org/10.1145/3650200.3656634

[3] P. Rostami, M. Sharifi, and M. Dejam, "Shape factor for regular and irregular matrix blocks in fractured porous media," *Petroleum Science*, vol. 17, pp. 136–152, 2020.

[4] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018.

[5] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:206594692

[7] E. Georganas, K. Banerjee, D. Kalamkar, S. Avancha, A. Venkat, M. Anderson, G. Henry, H. Pabst, and A. Heinecke, "Harnessing deep learning via a single building block," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 222–233.

[8] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: fast and memory-efficient exact attention with io-awareness," in *NIPS '22*. Red Hook, NY, USA: Curran Associates Inc., 2022.

[9] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, Y. Dong, and Y. Wang, "Flashdecoding++: Faster large language model inference on gpus," 2024. [Online]. Available: https://arxiv.org/abs/2311.01282

[10] 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 *ICPP '15*, 2015, pp. 200–209.

[11] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 634–643.

[12] K. Yu, X. Qi, P. Zhang, J. Fang, D. Dong, R. Wang, T. Tang, C. Huang, Y. Che, and Z. Wang, "Optimizing general matrix multiplications on modern multi-core dsps," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 964–975.

[13] J. Chen, N. Xiong, X. Liang, D. Tao, S. Li, K. Ouyang, K. Zhao, N. DeBardeleben, Q. Guan, and Z. Chen, "Tsm2: optimizing tall-and-skinny matrix-matrix multiplication on gpus," in *ICS '19*. New York, NY, USA: Association for Computing Machinery, 2019, p. 106–116. [Online]. Available: https://doi.org/10.1145/3330345.3330355

[14] 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*, 2012, pp. 684–691.

[15] Apple, "Apple introduces m4 pro and m4 max," October 2024, apple Newsroom. [Online]. Available: https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/

[16] "Nvidia gb200 nvl72: Hpc & ai gpu for data centers," NVIDIA website, 2025, accessed: 2025-08-18. [Online]. Available: https://www.nvidia.com/en-in/data-center/gb200-nvl72/

[17] H. Liu, S. Shi, X. Wang, Z. L. Jiang, and Q. Chen, "Performance analysis and optimizations of matrix multiplications on armv8 processors," in *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2024, pp. 1–6.

[18] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "Libshalom: optimizing small and irregular-shaped matrix multiplications on armv8 multi-cores," in *SC '21*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476217

[19] D. Wu, J. Meng, W. Zhu, M. Deng, X. Wang, T. Luo, M. Wahib, and Y. Wei, "autogemm: Pushing the limits of irregular matrix multiplication on arm architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024. [Online]. Available: https://doi.org/10.1109/SC41406.2024.00027

[20] 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*, ser. ICPP '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3545008.3545032

[21] Arm Developer, "Introducing-neon," https://developer.arm.com/documentation/dht0002/a/Introducing-NEON, accessed: 27 August 2025.

[22] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, p. 26–39, Mar. 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.35

[23] F. Wilkinson and S. McIntosh-Smith, "An initial evaluation of arm's scalable matrix extension," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 135–140.

[24] Hikunpeng, "Hpc kit download," 2024, hikunpeng Developer. [Online]. Available: https://www.hikunpeng.cn/developer/hpc/hpckit-download

[25] Apple Inc., "Accelerate," https://developer.apple.com/documentation/accelerate, 2025, accessed: 2025-07-11.

[26] ARM, "Streaming sve mode and za storage," https://developer.arm.com/documentation/109246/0100/Introduction/The-Scalable-Matrix-Extensions/Streaming-SVE-mode-and-ZA-storage, 2023, accessed: 2023-10-24.

[27] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, May 2008. [Online]. Available: https://doi.org/10.1145/1356052.1356053

[28] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2764454

[29] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, no. 2, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2925987

[30] Q. Han, Y. Hu, F. Yu, H. Yang, B. Liu, P. Hu, R. Gong, Y. Wang, R. Wang, Z. Luan, and D. Qian, "Extremely low-bit convolution optimization for quantized neural network on modern computer architectures," in *ICPP '20*. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3404397.3404407

[31] Arm Limited, *SME and SME2 - SME Overview*, 2023, accessed: 2025-08-22. [Online]. Available: https://developer.arm.com/documentation/109246/0100/SME-Overview/SME-and-SME2

[32] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: Accelerating small matrix multiplications by runtime code generation," in *SC '16*, 2016, pp. 981–991.

[33] S. Remke and A. Breuer, "Hello sme! generating fast matrix multiplication kernels using the scalable matrix extension," 2024. [Online]. Available: https://arxiv.org/abs/2409.18779

[34] kvcache-ai, "kvcache-ai/ktransformers: A flexible framework for experiencing cutting-edge llm inference optimizations," GitHub repository, 2025, latest release: v0.3.2 (July 1, 2025); accessed on 2025-08-18. [Online]. Available: https://github.com/kvcache-ai/ktransformers

[35] Intel Corporation. (2023) Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Overview. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html

[36] Intel Corporations. (2021) Advanced matrix extensions overview. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html

[37] H. Huang, J. Xie, G. Feng, X. Zhang, D. Huang, Z. Chen, and Y. Lu, "Hstencil: Matrix-vector stencil computation with interleaved outer product and mla," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1816–1829. [Online]. Available: https://doi.org/10.1145/3712285.3759769

[38] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785