SMEAtten: Fast and Memory-Efficient Outer Product-based Attention on ARMv9 CPUs with SME

Tengyang Zheng*, Han Huang*, Junru Chen, Xianwei Zhang, and Yutong Lu

School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
{zhengty26,huangh367,chenjr97}@mail2.sysu.edu.cn,
{zhangxw79,luyutong}@mail.sysu.edu.cn

Abstract. Transformer-based models are widely used in modern artificial intelligence, and the attention mechanism is a major determinant of their runtime efficiency. To accelerate matrix-intensive workloads, such as attention, ARMv9 introduces the Scalable Matrix Extension (SME) to enhance matrix processing capabilities on ARM CPUs. However, efficiently accelerating attention with SME remains challenging because of suboptimal compute-unit utilization, inefficient memory access, and limited task-level parallelism. To address these challenges, we present SMEAtten, a fast and memory-efficient attention design for ARMv9 CPUs with SME. SMEAtten incorporates three key techniques: throughput-driven interleaved matrix-vector attention kernels, an SME-adapted data layout and access scheme based on blocking, packing, buffering, and access-compute overlap, and inter-task parallelism exploitation. Experimental results show that SMEAtten delivers an average speedup of 13.62× over state-of-the-art baselines and achieves up to 56.09× acceleration in PyTorch integration evaluations. SMEAtten also supports efficient execution on different ARMv9 platforms, including LX2 and Apple M4.

Keywords: Attention, ARMv9, Scalable Matrix Extension

1 INTRODUCTION

Transformers have driven significant advances in natural language processing (NLP), powering highly effective models such as BERT [9] and GPT [21]. At the core of these models is the self-attention mechanism, which captures dependencies among tokens in a sequence [24,10]. A single attention computation over a **QKV** (query–key–value) triplet forms an **individual task**, and practical deployments typically execute many such tasks concurrently. Each task further consists of several **kernel-level operations**, such as the $QK^\top$ matrix multiplication and softmax normalization, which together contribute substantially to the

* Both authors contributed equally.

overall execution cost [17]. As a result, attention computation is highly resource-intensive, accounting for approximately 70% of the total runtime on average [10] and becoming the primary performance bottleneck in Transformer-based systems [19,7,23,18].

As Transformer workloads continue to spread across both everyday applications and large-scale services, ARMv9 is becoming an increasingly attractive platform for efficient Transformer execution [17]. ARM-based processors are now widely deployed across diverse systems, ranging from edge devices such as Apple M4 [2] and ARM C1 [5] to high-performance servers such as LX2 [14] and NVIDIA Grace [6]. To improve matrix-processing capability on ARM CPUs, ARMv9 introduces the Scalable Matrix Extension (SME) [25], which has been adopted by modern processors including LX2 and Apple M4. The fundamental outer-product instruction in SME extends the traditional one-dimensional SIMD computation model into a two-dimensional computation paradigm, improving performance by as much as 4× [22]. These features make ARMv9 with SME a promising substrate for accelerating Transformer attention, and SME-based acceleration has already attracted growing interest [14][15].

Although ARMv9 CPUs with SME provide substantial matrix-computation capability, efficiently accelerating attention on them remains challenging. SME follows an outer-product-based execution model that differs fundamentally from both traditional vector units and inner-product-based matrix units [20]. Existing library-style optimizations [13,3,4] often introduce extra overheads and cannot efficiently coordinate the mixed matrix-dominant and light-weight vector operations in attention. Moreover, SME-based execution must adapt to different hardware organizations across ARMv9 CPUs, which further complicates efficient and scalable deployment. As a result, existing attention optimization techniques [11,16,12,8] do not fully match SME, which calls for a dedicated architectural design.

To address these issues, efficient attention on CPUs with SME must resolve three key challenges: 1) Suboptimal kernel performance, due to pipeline stalls and insufficient exploitation of SME-specific computational features. 2) Inefficient memory access, often caused by fixed partitioning or platform-specific memory layouts [11]. 3) Limited task parallelism, as the task distribution is typically based only on input size rather than available compute capacity or system topology [1,13,26]. To overcome these limitations, we propose **SMEAtten**, an efficient attention design optimized for modern ARMv9 CPUs equipped with Scalable Matrix Extension [25]. At the **kernel** level, we implement attention kernels using interleaved SVE/SME instructions under a careful instruction scheduling strategy. Within **a single task**, we employ an SME-adapted data layout and access scheme to improve data locality and reduce memory overhead. Across **multiple tasks**, we account for the underlying core topology and dynamically distribute tasks to maximize parallel efficiency.

In summary, our contributions are as follows:

- We identify three key limitations that hinder attention execution on SME-enabled CPUs: unnecessary memory waste, low utilization of SME units, and

poor task-level parallelism. These limitations explain why existing attention implementations fail to effectively leverage SME architectures.

– We propose an SME-oriented attention design. Our design incorporates interleaved matrix-vector micro kernels and an SME-adapted data layout to improve matrix units utilization and reduce memory overhead. It also introduces an inter-task parallelism strategy that accounts for the core clusters in modern ARMv9 processors.
– We implement SMEAtten in a lightweight and practical way using *ARM C Language Extensions* for SME/SVE kernels and *Pthreads* for the parallel execution runtime. Evaluation shows that SMEAtten delivers substantial performance improvements across both LX2 and Apple M4 CPUs, demonstrating high efficiency and portability.

2 BACKGROUND AND MOTIVATION

2.1 ARMv9 Architecture

With the growing interest in AI workloads, ARMv9 introduces the Scalable Matrix Extension (SME) to enhance matrix-processing capability. Centered around the ZA (Z-Array) tiles, SME provides the Matrix Outer Product Add (MOPA) instruction, which takes two SVE vectors as input and accumulates the results into the ZA tiles. Compared with the traditional one-dimensional vector computation model, SME exposes a two-dimensional matrix-oriented paradigm with higher throughput for matrix-intensive workloads.

SME also introduces streaming execution and specialized load/store behaviors that affect instruction scheduling, data access, and cache utilization. Although different ARMv9 processors may expose different SME implementations, they share the same matrix-oriented execution foundation, which makes SME promising for attention acceleration while introducing new optimization challenges compared with conventional SIMD-based execution.

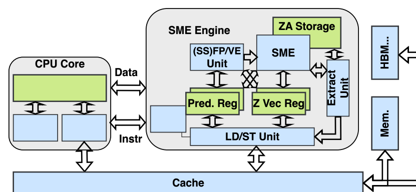


Fig. 1: ARMv9 micro architecture.

2.2 Outer Product-based Attention

Mainstream attention implementations [23,11] typically follow a GEMM-centric workflow, where the QK^T and SV multiplications dominate the computation. When mapped onto CPUs with SME, this workflow leads to the structure shown in Figure 2. It consists of **intra-MM**, which includes two batched matrix multiplications and three point-wise operations (*scale*, *mask*, and *softmax*), and

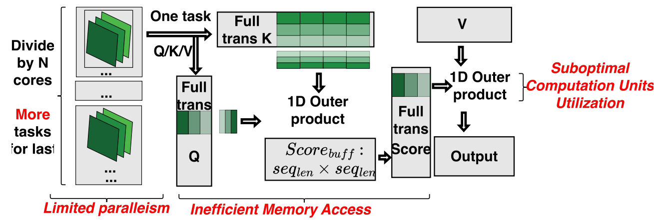


Fig. 2: Typical outer product-based attention.

inter-MM, which repeats this process $batch_{size} \times h$ times. To exploit high-performance GEMM libraries such as KBLAS and Accelerate, previous works partition the input matrices into blocks. Under this design, the SME engine executes $S \leftarrow Q \times K^T$ and $O \leftarrow S \times V$, while SVE vectors handle the point-wise operations.

2.3 Motivation and Related Work

Although SME opens new opportunities for accelerating attention computation, it also creates several performance challenges when transitioning from a vector-centric to a matrix-centric compute paradigm. These challenges manifest primarily as: a) suboptimal utilization of computational units, b) inefficient memory access, and c) limited task parallelism. Prior attention optimizations on GPUs, conventional ARM CPUs, and general matrix or SDPA libraries address some of these issues, but they do not directly align with SME's outer-product-based computation and mixed SVE/SME execution.

Suboptimal Computation Units Utilization: Existing attention kernels are typically designed for either vector units or inner-product-based matrix units [11,12], and therefore cannot fully exploit SME's execution model. On SME-enabled CPUs, attention combines matrix-dominant and light-weight vector operations, which require different instruction mappings on SME and SME2. Moreover, the interaction between SVE and SME execution introduces additional opportunities for instruction-level overlap, calling for a tailored kernel and scheduling design. A recent ARM CPU attention implementation, MEATTEN [11], exploits data reuse with NEON-based kernels. As shown in Figure 3, MEATTEN achieves only a small fraction of the compute capability available on SME-enabled CPUs across sequence lengths. This large utilization gap suggests that a vector-centric implementation cannot effectively exploit SME, and leaves substantial room for architecture-aware kernel mapping and instruction scheduling.

Inefficient Memory Access: Existing attention implementations often optimize locality for vector or inner-product-based matrix units [23,11,16,8]. However, these designs do not match SME's outer-product execution model. Directly migrating them either increases memory overhead, such as storing fully transposed intermediates in Figure 2, or leads to non-contiguous accesses within blocks. Therefore, attention on SME requires a dedicated data layout and access scheme. The runtime breakdown in Figure 3 further indicates that the dominant

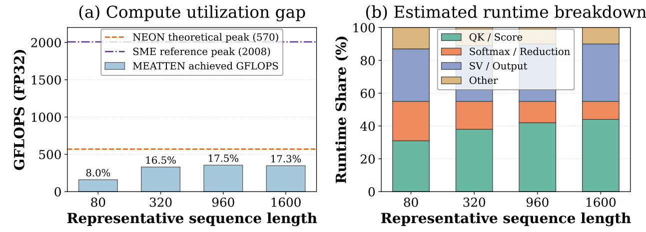


Fig. 3: Compute utilization and runtime breakdown of MEATTEN on ARMv9.

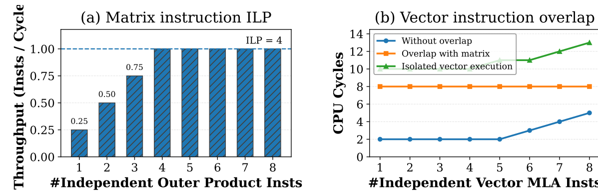


Fig. 4: Illustration of matrix instruction ILP and vector instruction overlap on ARMv9.

attention phases remain the major contributors to execution time, which makes memory access efficiency within these phases particularly important.

Limited Task Parallelism: The parallelism of attention varies with both seq_{len} and $batch_{size}$ [11], while existing task-partition strategies are usually designed for vector-centric or inner-product-based implementations [1,13,26]. As a result, they cannot effectively adapt to SME-based execution or to the topology differences across ARMv9 CPUs, such as the shared SME engine on Apple M4 and the per-core SME engines on LX2. This calls for an architecture-aware task allocation mechanism.

These challenges motivate an architecture-aware design that jointly improves compute utilization, memory access, and task scheduling on SME-enabled CPUs.

3 Design

In this section, we present *SMEAtten*, a novel design for accelerating attention computations on CPUs with SME. As shown in Figure 5, *SMEAtten* comprises three core components:

- Integrated matrix-vector **micro kernels** in §3.1 that leverage matrix and vector units and schedule instructions based on attention characteristics to enhance computation throughput.
- SME-adapted **data layout and access scheme for a single task** in subsection 3.2, which organizes blocking, packing, buffering, and access-compute overlap to improve locality and reduce memory overhead.
- **Inter-task parallelism** exploitation in §3.3 to execute in different micro architectures with SME, which partitions and distributes tasks, and identifies the architecture features for task-thread mapping.

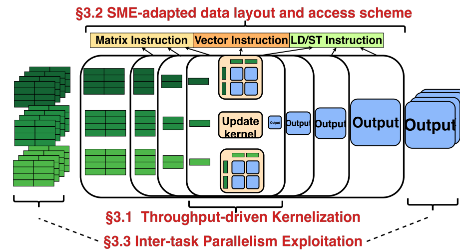


Fig. 5: Overview of SMEAtten. SMEAtten improves attention execution on ARMv9 CPUs with SME through kernelization, data layout optimization, and inter-task parallelism.

3.1 Throughput-driven Kernelization

Efficient attention on SME-enabled CPUs requires high utilization of both matrix and vector resources. At the block level, SMEAtten organizes attention into three kernels, namely **Score**, **Update**, and **Output**, where **Score** computes block scores $S = Q \times K^T$, **Update** maintains the row-wise softmax states across blocks, and **Output** accumulates the output from the normalized scores and V. Among them, **Score** and **Output** are matrix-dominant and compute-intensive, while **Update** mainly performs light-weight vector operations for softmax-state maintenance. Accordingly, we focus on the compute-centric design first, while the memory-centric optimizations are presented in §3.2.

2D Outer-product Kernel. SMEAtten selects two vectors from Q_b and two vectors from K_b, arranges them into a 2×2 grid, and accumulates all pairwise combinations directly into four ZA tiles. In FP32 mode, SME naturally exposes four parallel accumulation regions, and the 2×2 mapping aligns with this four-way structure, allowing four outer products to proceed concurrently. Compared with asymmetric groupings such as 1×4 and 4×1, the 2×2 mapping requires fewer live operand registers and yields a better compute-to-memory ratio. Overall, this mapping better matches SME's execution model, increases tile-level parallelism, and reduces instruction count by up to 20%.

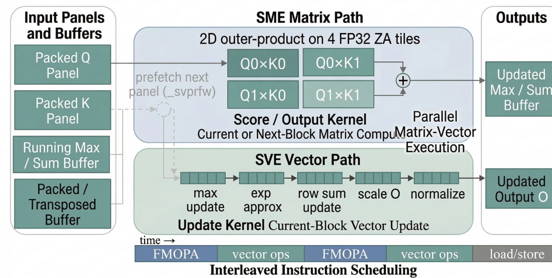


Fig. 6: Block-level cooperative SME/SVE execution in SMEAtten.

Vector Operations. Besides the matrix-dominant **Score** and **Output** kernels, attention also requires several light-weight vector operations, as shown by the SVE vector path in Figure 6. These operations include maximum update,

exponential approximation, row-sum update, partial-output scaling, and normalization. Since they are dominated by element-wise transformation and row-wise state maintenance rather than outer-product accumulation, SMEAtten maps them to the **Update** kernel on the SVE path. This vector-side organization efficiently maintains the running softmax states across blocks while leaving the SME path to focus on the matrix-heavy computation.

For architecture-specific tuning, SMEAtten further selects different instruction variants on SME and SME2 for these vector-side operations. In the **Update** kernel, we use higher-throughput SME2 instructions such as *svtbl2* and *svsel* to accelerate vector lookup and selection during state update. In addition, SME2 also allows us to further optimize the matrix-side pipeline by merging the *max* and *store* operations in the **Score** kernel through multi-vector instructions.

Matrix-vector Cooperative Execution. In the SME architecture, the SME instructions can only take up to one of the two issue ports of the SME units. Thus, there leaves a chance to overlap the matrix instructions with vector instructions, as described in (ILP tests in sec2). Since a 2D outer-product update uses only part of the architectural vector registers as SME inputs, the unused SVE registers are assigned to an independent vector path for extra inner-product-style accumulation on independent fragments. In this way, SME handles the main matrix-dominant computation, while SVE opportunistically processes additional vector work in parallel. This cooperation also guides execution across kernels, as shown in algorithm 1. In the **Update** kernel, the vector path becomes dominant, while SME can simultaneously start the matrix multiplication of the next block.

Algorithm 1: Interleaved SME/SVE execution in SMEAtten

Input: current-block states $m_{j-1}, m_j, s_{j-1}, O_{old}$; next-block packed panels Q_{b+1}, K_{b+1}

1 initialize ZA tiles for next-block Score;
2 **for** $t \leftarrow 0$ **to** i_{end} ***step*** *vl* **do**
3 $m_{new} \leftarrow \max(m_{j-1}[t], m_j[t])$, $\Delta \leftarrow m_{j-1}[t] - m_{new}$;
4 update s using SME/SME2 vector lookup and select operations;
5 $s \leftarrow poly_corr(s, \Delta)$;
6 $s_{j-1}[t] \leftarrow s_{j-1}[t] \times s$, $O_{old}[t : t + vl] \leftarrow O_{old}[t : t + vl] \times s$;
7 **if** *next block exists* **then**
8 prefetch next packed panels with *_svprfw*;
9 load q_0, q_1 from Q_{b+1} and k_0, k_1 from K_{b+1};
10 issue $FMOPA(ZA_0 \dots ZA_3, \{q_0, q_1\}, \{k_0, k_1\})$;

3.2 SME-adapted Data Layout and Access Scheme

Attention execution on SME-enabled CPUs is highly sensitive to data movement due to repeated accesses to tiled $Q/K/V$ data and intermediate *Score* buffers. To improve locality and reduce data access overhead, SMEAtten adopts an SME-adapted data layout and access scheme based on blocking, packing, buffering, and

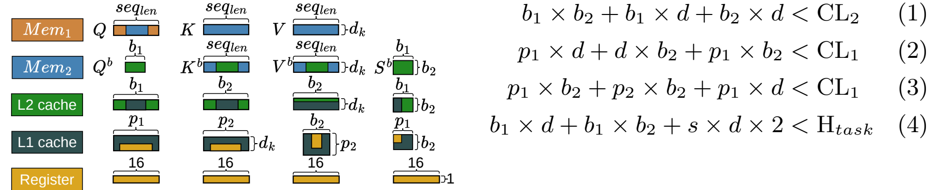
access-compute overlap, as shown in Figure 7 and constrained by Equation 1–Equation 4.

L2 Cache Blocking. SMEAtten applies **blocking** at the L2-cache level to bound the working set of each attention task. We divide $Q/K/V/O$ into blocks and process one Q block together with the corresponding K and V blocks. This loop ordering improves temporal locality across the *Score* and *Output* kernels. We constrain the block sizes using Equation 1 so that the main working set remains cache-resident.

L1 Cache Packing. Within each block, SMEAtten performs **packing** to reconstruct the accessed panels into contiguous layouts that are friendly to SME outer-product execution. The packed panels are streamed into registers for the micro kernels, reducing irregular accesses and L1 cache misses during both the *Score* and *Output* kernels. We use Equation 2 and Equation 3 to constrain the panel sizes within the L1 cache, and set p_1 and p_2 to 32 according to the micro-kernel configuration.

Memory Buffer. SMEAtten preallocates a fixed-size **buffer pool** for packed and transposed operands consumed by the compute kernels. The total buffering footprint is bounded by Equation 4, where H_{task} denotes the memory capacity assigned to each attention task. Since the buffering memory can differ from the original input location, SMEAtten places the pool in high-bandwidth memory when available, so that packed panels and transposed operands can be supplied with lower access cost.

Finally, SMEAtten overlaps memory access with computation through **instruction reordering**. While the current panel is being consumed by the SME/SVE micro kernels, the next packed or transposed panel is prefetched into cache using *_svprfw* in line 8. This access-compute overlap partially hides data movement behind ongoing computation and helps sustain throughput across blocks and panels.



$$b_1 \times b_2 + b_1 \times d + b_2 \times d < \mathrm{CL}_2 \quad (1)$$

$$p_1 \times d + d \times b_2 + p_1 \times b_2 < \mathrm{CL}_1 \quad (2)$$

$$p_1 \times b_2 + p_2 \times b_2 + p_1 \times d < \mathrm{CL}_1 \quad (3)$$

$$b_1 \times d + b_1 \times b_2 + s \times d \times 2 < \mathrm{H}_{task} \quad (4)$$

Fig. 7: ARMv9-aware packing, blocking, buffering, and prefetching strategy. Mem_1/Mem_2 denote low/high speed memory on ARMv9 platforms.

3.3 Inter-task Parallelism Exploitation

The designs above could only enhance performance within a single task. To execute efficiently on **multicore**, SMEAtten partitions and distributes tasks carefully, and then maps threads to cores depending on microarchitecture, to exploit parallelism.

To adapt to different sizes of inputs for a single task, we utilize data padding and then migrate our SME-adapted data layout and access scheme. Matrices

smaller than a block are padded to match the block size, whereas for matrices exceeding one block, padding is applied solely to the last panel to complete the final block. During computing at the panel level, we utilize the *mask* registers in SME to avoid wrong calculations.

To reasonably divide and distribute tasks, SMEAtten follows the algorithm 2 that identifies architecture features and computation characteristics. Considering the expensive transposition operation, we partition $batch_{size} \times h$ tasks at the task level rather than within tasks. Depending on the number of threads T, SMEAtten follows the line 1 in algorithm 2 to divide the tasks. Based on the computing capabilities of the SME and SVE units, we aggregate the core performance within each cluster and sort the clusters in descending order, as shown from line 2 to line 4 in algorithm 2. Then, according to the thread identifier, we determine the mapped core and the range of task indices assigned to it, as shown at line 5. We bind threads to the cores in a one-to-one manner. As for the remaining tasks, we dynamically push them to free threads, each time a single task. Based on our mapping method, SMEAtten maintains the continuity of data among adjacent cores, ensuring efficient parallelism.

Algorithm 2: TASKALLOC function with thread-core mapping.

Input: Number of threads T, thread identifier $thread_{id}$, total number of tasks N, SME/SVE GFLOPS G^{SME}/G^{SVE}

Output: Assigned task indices $task_s$, $task_e$ and corresponding CPU cores

1 **Step 0:** Compute $q = \lfloor N/T \rfloor$ and remainder tasks $R = N \bmod T$. Store the R extra tasks into an array $\mathcal{R}$.

2 **Step 1:** Collect SME/SVE units' GFLOPS (G^{SME}/G^{SVE}), and the SME–SVE mapping relation $\mathcal{M}$.

3 **Step 2:** Construct an array $\mathcal{A}$ of size $|\mathcal{M}|$, where each element is

$$\mathcal{A}[i] = G_i^{SME} + 2 \cdot \sum_{j \in \mathcal{M}(i)} G_j^{SVE},$$

and another array $\mathcal{B}$ where $\mathcal{B}[i] = |\mathcal{M}(i)|$.

4 **Step 3:** Sort $\mathcal{A}$ and $\mathcal{B}$ jointly in descending order of $\mathcal{A}$.

5 **Step 4:** For a given $thread_id$:

- Determine the SME group index by subtracting $\mathcal{B}[i]$ sequentially until $thread_id < \mathcal{B}[i]$.
- Within that group, map the residual index to the corresponding SVE core, and get $Core_{map}$.
- Compute the task range as $[thread_id \times q,\ (thread_id + 1) \times q)$

return $task_s$, $task_e$ *and* $Core_{map}$.

3.4 Portable Implementation

SMEAtten also remains portable across ARMv9 CPUs with different SME organizations. Using LX2 and Apple M4 as examples, at the **kernel** level, SMEAtten implements both SME and SME2 kernels through ARM C Language Extensions and selects the appropriate version according to the target microarchitecture. Within **a single task**, SMEAtten places inputs and buffers according to the memory hierarchy, e.g., using unified LPDDR5x on Apple M4 and DDR/HBM on LX2. Across **multiple tasks**, portability is supported by the `TASKALLOC` function in algorithm 2, which adapts to both the P/E-core clusters of Apple M4 and the single-core clusters of LX2 based on the SME-SVE mapping relation.

4 EVALUATION

4.1 Experiment Setups

We conduct tests on the LX2 CPU and Apple M4 CPU. The hardware configurations and comparison methods are summarized in Table 1. We set the batch size to 64, the number of attention heads to 12, and the head dimension to 64. We vary the sequence length from 160 to 1600, use FP32, and keep the same input configurations and thread counts across methods on each platform. We report performance in GFLOPS, computed as the total floating-point operations of the two matrix multiplications in attention divided by execution time. All methods are evaluated with the same input configurations, precision, and thread counts on each platform, while execution is measured through each implementation's available operator path.

Table 1: Experimental setup.

Hardware platforms				
CPU	L1 Cache	L2 Cache	Memory	Core Cluster
LX2	64 KB	512 KB	32/4 GB DDR/HBM	32 cores/NUMA node
Apple M4	320 KB	4096 KB	16 GB LPDDR5x	4P+6E cores
Comparison methods				
XNNPACK fused SDPA operator[1]				XNN_F
XNNPACK non-fused SDPA (individual routines)[1]				XNN_NF
MEATTEN (ARMv8, NEON-optimized)[11]				MEATTEN
SMEAtten (Our work)				SMEAtten

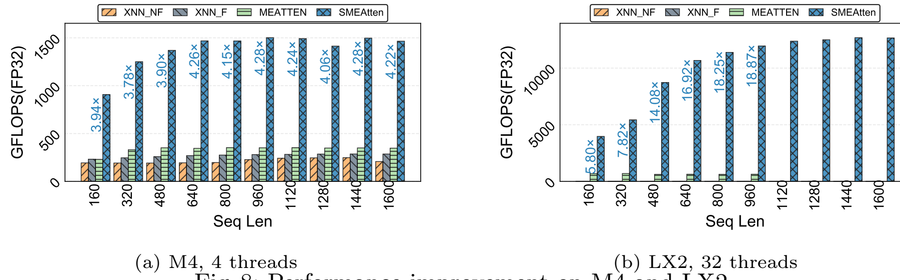
4.2 Performance Improvement

In this experiment, we evaluate the parallel performance of the SDPA operator using 32 cores on LX2 and 4 performance cores on M4. As shown in Figure 8, SMEAtten outperforms XNN_NF, XNN_F and MEATTEN across sequence lengths 160–1600. Compared to MEATTEN, SMEAtten achieves average speedups of **4.11**× on M4 and **13.62**× on LX2. The highest speedup appears at $seq_{len} = 960$, where dynamic parallelism mitigates load imbalance for large inputs; at $seq_{len} = 160$, SMEAtten still reaches **3.94**× on M4 and **5.80**× on LX2.

Moreover, all competing methods encounter out-of-memory errors when $seq_{len} > 960$. These OOM cases mainly come from intermediate score and other temporary buffers whose memory footprint grows rapidly with sequence length. In contrast, SMEAtten still maintains strong performance and scales well at larger sequence lengths by reusing bounded packed panels and buffer space, as discussed in §3.2.

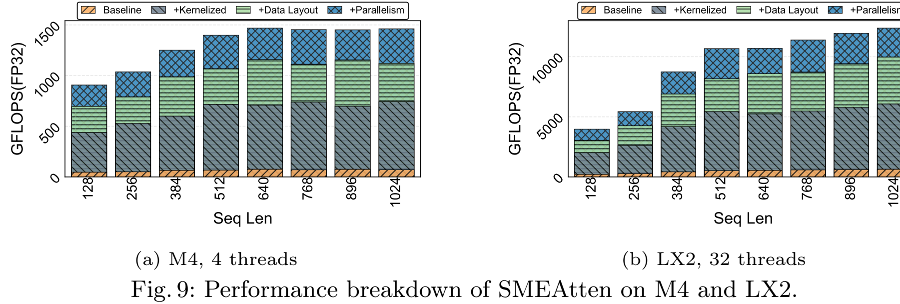
4.3 Performance Breakdown

To evaluate the effectiveness of SMEAtten, we analyze the contribution of each design component in Figure 9. Starting from a baseline implementation without



(a) M4, 4 threads (b) LX2, 32 threads

Fig. 8: Performance improvement on M4 and LX2.



(a) M4, 4 threads (b) LX2, 32 threads

Fig. 9: Performance breakdown of SMEAtten on M4 and LX2.

SMEAtten-specific optimizations, we incrementally evaluate the kernelized implementation, the kernelized implementation with the SME-adapted data layout and access scheme, and the full SMEAtten system with inter-task parallelism.

- Enabling the integrated matrix–vector micro-kernels directly improves SME throughput, reaching up to **10.63**× on M4 and **10.47**× on LX2 at $seq_{len} = 512$.
- Adding the SME-adapted data layout and access scheme improves memory management and the L1 cache hit rate, yielding average speedups of **1.57**× on M4 and **1.60**× on LX2 over the kernelized implementation.
- Finally, adding the inter-task parallelism mechanism further improves long-sequence performance, with additional average speedups of **1.29**× on M4 and **1.28**× on LX2.

4.4 Cache and Memory Utilization

To demonstrate the feasibility of our SME-adapted data layout and access scheme, we evaluate MEATTEN and SMEAtten with different seq_{len} to collect their L1 cache hit rates and HBM usage in Table 2 on LX2. Our strategy is tailor-made for outer-product matrix units, deserving a higher cache hit rate, which is discussed in detail in §3.2. The results, shown in Table 2, are consistent match our expectations: SMEAtten delivers an average of **7.64%** hit rate improvement over MEATTEN, up to **9.78%**. The larger improvement of L1 cache hit rates for larger inputs indicates the advantages of our data layout and tiling strategy. MEATTEN fails to execute when the sequence length is 1024 and 2048, resulting in an out-of-memory error. On the contrast, SMEAtten still maintains low HBM Usage, saving **3.9** GB HBM at least. Additionally, SMEAtten increases the HBM

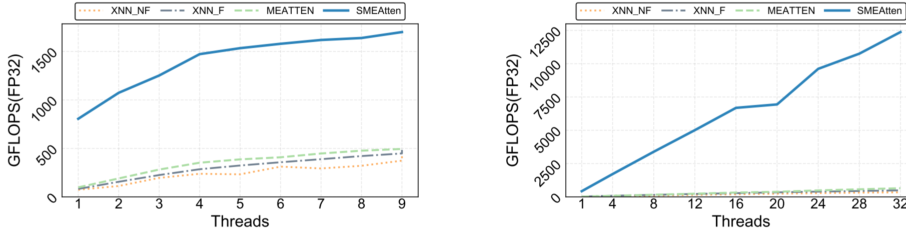
Table 2: L1 hit rates and HBM usage on LX2. MA denotes MEATTEN, and SA denotes SMEAtten.

Seq.	L1 Hit (%)		HBM (MB)		Seq.	L1 Hit (%)		HBM (MB)	
Len.	MA	SA	MA	SA	Len.	MA	SA	MA	SA
256	66.76	71.73	405.01	17.55	1024	48.21	57.12	OOM	32.84
512	63.44	70.35	1212.68	22.65	2048	45.90	55.68	OOM	53.23

usage at a rate lower than linear, starting from a modest baseline footprint and scaling gradually with sequence length.

4.5 Scalability and Portability Analysis

We conduct the scaling and portability experiments on LX2 and Apple M4, as shown in Figure 10. SMEAtten reaches average performance of **111.82** GFLOPS on M4 and **385.48** GFLOPS on LX2, corresponding to **36.77**×, **25.59**×, and **19.00**× the performance of XNN_NF, XNN_F, and MEATTEN on LX2. SMEAtten also exhibits a steeper growth curve than other methods, indicating that our parallel mechanism scales well across SME-enabled architectures. Owing to the subnuma design within a NUMA node, SMEAtten experiences a sudden decrease in scalability around 20 cores on LX2.



(a) M4 scalability

(b) LX2 scalability

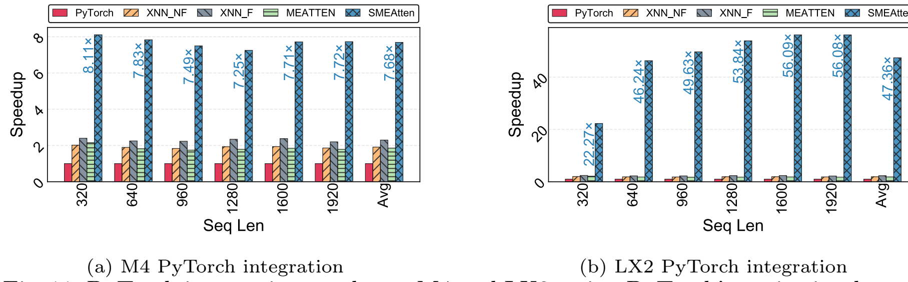
Fig. 10: Thread scalability on M4 and LX2 ($seq_{len} = 1120$).

4.6 PyTorch Integration Case Studies

In Figure 11, we replace only the SDPA computation in PyTorch with SMEAtten and evaluate the resulting attention module against PyTorch's native implementation. SMEAtten reaches up to **8.11**× speedup on M4 and **56.09**× on LX2, while MEATTEN remains slower because of less effective memory allocation and task distribution in its PyTorch integration.

5 CONCLUSION

Attention computation serves as a critical component in Transformer-based models and a key determinant of overall performance. In this study, we present SMEAtten, the first practical attention computation framework designed to fully exploit the capabilities of the Scalable Matrix Extension on modern CPUs. Experiments on both LX2 and Apple M4 show that SMEAtten achieves substantial performance gains while remaining portable across SME-enabled CPUs.



(a) M4 PyTorch integration

(b) LX2 PyTorch integration

Fig. 11: PyTorch integration results on M4 and LX2, using PyTorch's native implementation as the baseline.

Acknowledgments. We sincerely thank the anonymous reviewers for their constructive comments and suggestions. This work is supported by the Guangdong S&T Program under Grant NO. 2024B0101040005, NSFC Grant #62461146204 / #62472462. Xianwei Zhang and Yutong Lu are the corresponding authors.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Google xnnpack. https://github.com/google/xnnpack. (2023)
2. Apple introduces m4 pro and m4 max. https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/ (2024), [Online; accessed 12-Feb-2025]
3. Hpc kit download (2024), https://www.hikunpeng.cn/developer/hpc/hpckit-download, [Online; accessed 12-Feb-2025]
4. Accelerate (2025), https://developer.apple.com/documentation/accelerate, [Online; accessed 12-Feb-2025]
5. Arm c1-premium. https://www.arm.com/products/silicon-ip-cpu/c1-premium (2025), [Online; accessed 12-Feb-2025]
6. Nvidia gb200 nvl72: Hpc & ai gpu for data centers. https://www.nvidia.com/en-in/data-center/gb200-nvl72/ (2025), [Online; accessed 12-Feb-2025]
7. Abdelfattah, A., Costa, T., Dongarra, J., Gates, M., Haidar, A., Hammarling, S., Higham, N.J., Kurzak, J., Luszczek, P., Tomov, S., et al.: A set of batched basic linear algebra subprograms and lapack routines. ACM Transactions on Mathematical Software (TOMS) **47**(3), 1–23 (2021)
8. Chen, H., Xie, W., Zhang, B., Tang, J., Wang, J., Dong, J., Chen, S., Yuan, Z., Lin, C., Qiu, C., Zhu, Y., Ou, Q., Liao, J., Chen, X., Ai, Z., Wu, Y., Zhang, M.: Ktransformers: Unleashing the full potential of cpu/gpu hybrid inference for moe models. In: Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (2025)
9. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers). pp. 4171–4186 (2019)
10. Du, J., Jiang, J., You, Y., Huang, D., Lu, Y.: Handling heavy-tailed input of transformer inference on gpus. In: Proceedings of the 36th ACM International Conference on Supercomputing. pp. 1–11 (2022)

11. Fu, X., Yang, W., Dong, D., Su, X.: Optimizing attention by exploiting data reuse on arm multi-core cpus. In: Proceedings of the 38th ACM International Conference on Supercomputing. pp. 137–149 (2024)
12. Gao, X., Lin, X., Liu, R.: Comparable gpu: Optimizing the bert model with amx feature. In: 2023 IEEE 3rd International Conference on Computer Communication and Artificial Intelligence (CCAI). pp. 158–162. IEEE (2023)
13. Heinecke, A., Henry, G., Hutchinson, M., Pabst, H.: 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)
14. Huang, H., Xie, J., Feng, G., Zhang, X., Huang, D., Chen, Z., Lu, Y.: Hstencil: Matrix-vector stencil computation with interleaved outer product and mla. In: The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), St. louis, MO, United States (2025)
15. Huang, L., Huang, H., Chen, Z., Lu, Y.: Kirbymm: Outer-product based matrix multiplication on armv9 processor
16. Intel Corporation: intel-extension-for-pytorch (2025), https://deepwiki.com/intel/intel-extension-for-pytorch/5.2-attention-mechanism-optimizations
17. Jiang, J., Du, J., Huang, D., Li, D., Zheng, J., Lu, Y.: Characterizing and optimizing transformer inference on arm many-core processor. In: Proceedings of the 51st International Conference on Parallel Processing. pp. 1–11 (2022)
18. Kim, H., Ye, G., Wang, N., Yazdanbakhsh, A., Kim, N.S.: Exploiting intel® advanced matrix extensions (amx) for large language model inference. IEEE Computer Architecture Letters (2024)
19. Li, X., Liang, Y., Yan, S., Jia, L., Li, Y.: A coordinated tiling and batching framework for efficient gemm on gpus. In: Proceedings of the 24th symposium on principles and practice of parallel programming. pp. 229–241 (2019)
20. Markidis, S., Der Chien, S.W., Laure, E., Peng, I.B., Vetter, J.S.: Nvidia tensor core programmability, performance & precision. In: IEEE IPDPSW. pp. 522–531 (2018)
21. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training (2018)
22. Remke, S., Breuer, A.: Hello sme! generating fast matrix multiplication kernels using the scalable matrix extension. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1443–1454. IEEE (2024)
23. Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., Dao, T.: Flashattention-3: Fast and accurate attention with asynchrony and low-precision. Advances in Neural Information Processing Systems **37**, 68658–68685 (2024)
24. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
25. Wilkinson, F., McIntosh-Smith, S.: 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). pp. 135–140. IEEE (2022)
26. Yang, W., Fang, J., Dong, D., Su, X., Wang, Z.: Optimizing full-spectrum matrix multiplications on armv8 multi-core cpus. IEEE Transactions on Parallel and Distributed Systems **35**(3), 439–454 (2024)