Optimizing Attention by Exploiting Data Reuse on ARM Multi-core CPUs

Xiao Fu[*]
National University of Defence Technology
China
fuxiao21@nudt.edu.cn

Weiling Yang[*]
National University of Defence Technology
China
w.yang@nudt.edu.cn

Dezun Dong[†]
National University of Defence Technology
China
dong@nudt.edu.cn

Xing Su[†]
National University of Defence Technology
China
xingsu@nudt.edu.cn

ABSTRACT

Transformers reign supreme in natural language processing, representing a milestone innovation in deep learning. For high-performance model inference, optimizing the time-consuming attention module is crucial. Owing to the irregular-shaped matrix workloads and intricate data access patterns, the attention operator is bounded by memory bandwidth. Existing works utilize kernel fusion to reduce memory access overhead, resulting in promising performance enhancements. However, these efforts primarily focus on GPU or X86 architectures, leaving ARM multi-cores, commonly encountered in emerging HPC systems, insufficiently explored. We present MEATTEN, a memory-efficient attention fusion scheme and batched approach to exploit ARM multi-core CPUs effectively. It builds on fused micro-kernels and a new data layout suitable for SIMD vectorization. An analytic model is used to guide loop permutation, tiling, and batched parallelization according to the on-chip hierarchical memory architecture and workload characterization. We apply MEATTEN to three representative ARM multi-cores against state-of-the-art libraries and compilers. Experimental results demonstrate that our approach consistently outperforms prior approaches across various evaluation scenarios and platforms.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Theory of computation → Shared memory algorithms**.

KEYWORDS

Attention module, ARM multi-cores, Fusion, Optimization

[*]Equal contribution.
[†]Corresponding author.

ACM Reference Format:
Xiao Fu, Weiling Yang, Dezun Dong, and Xing Su. 2024. Optimizing Attention by Exploiting Data Reuse on ARM Multi-core CPUs. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656620

1 INTRODUCTION

Recently, the field of deep learning (DL) has achieved significant success in natural language processing with Transformer-based models [40] such as Bert [14] and GPT [36]. Transformers use a self-attention mechanism that simultaneously computes the dependencies between any two tokens in a sequence [15]. This feature endows the models with exceptional algorithmic parallelism and accuracy while mitigating gradient vanishing and exploding issues [15, 17].

As shown in Figure 1, the execution time of the attention module accounts for 70% of the overall time when inferencing the Bert-base model. This demonstrates that the module is a significant performance bottleneck of the Transformer-based models. It mainly comprises two batched matrix multiplication[1](MM) operators and a softmax positioned between them [25]. In real model deployment, the performance optimization of the attention poses three considerable challenges. Firstly, the MM workload in attention is typically small and irregular-shaped, with the second MM having the opportunity to benefit from the first MM when fusing them [55]. This is because the sequence lengths of Transformers are variable and small [15, 41]. For instance, the MM has $M = N = 512, K = 64$ in the Bert-base model, which is considered a non-common matrix shape and hard to optimize [22, 47]. Secondly, the softmax operator is memory-intensive and exhibits complex data dependencies, necessitating the meticulous design to fuse it into the two MM while ensuring accuracy. Lastly, the current optimization support for batched MM remains rudimentary on CPUs [4, 30].

Existing dense linear algebra libraries have adopted highly optimized strategies for large-scale MM and can achieve about 80% of a processor's peak performance [44, 50]. However, these libraries exhibit poor performance for the matrix sizes commonly used in Transformer models. To accelerate small and irregular-shaped MM,

[1]Batched MM can be expressed as $\mathbf{C_i} = \alpha_i \mathbf{A_i} \mathbf{B_i} + \beta_i \mathbf{C_i}$, $i = 1, 2, \ldots$, where the matrices $\mathbf{A_i}, \mathbf{B_i}$, and $\mathbf{C_i}$ have sizes $M \times K$, $K \times N$, and $M \times N$, respectively.

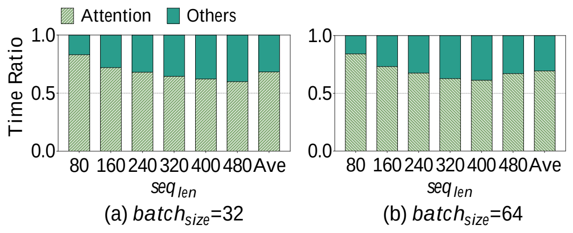

Figure 1: Time breakdown of the Bert-base model on a 64-core ThunderX2 server. We utilize PyTorch 2.0.0 with BLIS as the backend.

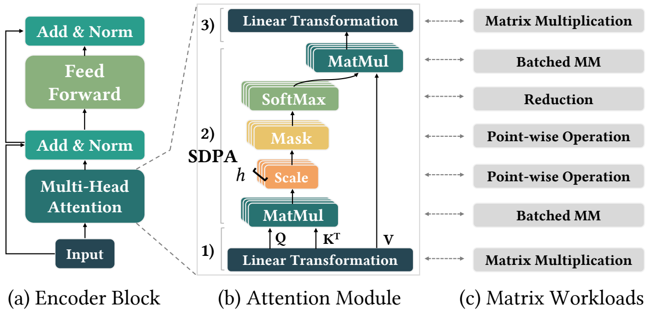

Figure 2: The architecture of an encoder block and the workflow of the attention module.

LIBXSMM [22] and LIBSHALOM [47, 48] have designed new computation micro-kernels targeted at X86 and ARM platforms, respectively. LIBXSMM utilizes cache and vector-friendly data layouts, which makes it unable to be directly integrated into DL frameworks [42] (e.g., PyTorch [35]). These works focus solely on optimizing MM operations and do not consider operator fusion across operations. To overcome the limitation, XNNPACK [2] fuses the MM and softmax operations within the attention module to reduce the overhead imposed by the softmax operation. Nevertheless, it does not carefully design the micro-kernel and batched parallelization strategy for the MM operation, resulting in its inability to utilize the processor's cores and SIMD units fully. Ansor [54] employs an automated tuning approach to optimize the attention module and is unable to perceive the details of the underlying hardware, thus delivering inferior performance compared to a hand-written library [45]. The library FLASHATTENTION [13] achieves highly promising acceleration on GPUs. GPUs exhibit numerous architectural differences, and further exploration is still needed to optimize the attention module on CPUs.

ARM multi-cores are widely used in high-performance clusters and data centers [1, 26, 38, 46, 49]. For instance, Japan's Fugaku supercomputer [38] is a homogeneous system ranking at the top 500 [3] list and entirely constructed based on the Fujitsu A64FX ARM CPU [34]. Recent research also indicates that utilizing ARM multi-cores for DL workloads is highly suitable [5, 6, 24, 25, 28, 42]. This motivates us to develop an efficient attention library on the ARM architecture.

To this end, the paper presents MEATTEN[2], an open-source library dedicated to optimizing the self-attention module on ARM multi-core CPUs. Novel computation micro-kernel, data format, and parallelization strategies have been implemented. Specifically, we develop micro-kernels based on outer product and vector instructions while utilizing an online strategy for the register-level fusion pattern of "softmax to MM". We design a data format suitable for matrix multiplication and judicious loop layouts to reduce memory access overhead and improve data locality. Informed by an analytical model, our decisions on loop permutation, tiling, and parallelization are tailored to the characteristics of workloads and the hierarchical memory system. Intra- and inter-MM parallelism are exploited while achieving load balance. We share our experience that the core insight of the proposed holistic optimizations lies in leveraging data reuse to mitigate the expensive overhead associated with memory read and write operations.

We demonstrate the advantages of MEATTEN by applying it to three modern ARM multi-core CPUs, Phytium 2000+[18], Kunpeng 920 (KP920) [43], and ThunderX2 [32]. We compare it with the state-of-the-art solutions, including both library [2, 31] and compilation methods [54]. Experimental results indicate that our approach delivers optimal performance across various scenarios, including batch size, sequence length, and thread number. To further show the benefits of MEATTEN on end-to-end inference, we integrate it into the DL framework PyTorch [35]. Our integration shows that MEATTEN achieves a speedup of over 3× on the representative Bert-base model.

The contributions of the paper can be summarized as follows:

- It provides an analytical model that guides data reuse and derives the algorithmic parameters for different architectures and workloads (Section 4).
- It presents a new method to develop fused micro-kernels, reducing memory access overhead (Section 5).
- It designs a bathed parallelization algorithm, enabling intra- and inter-MM parallelism (Section 6).

2 BACKGROUND

2.1 Transformer-based Models

Transformer-based language models [8, 40] typically consist of four modules: embedding, encoder, decoder, and output. The encoder and decoder dominate the execution time of a model and are a key optimization focus. The encoder is formed by stacking multiple encoder blocks with the same structure but independently trained parameters such as weights and biases. The construction of the decoder follows a similar approach. Different networks exhibit distinct structures; for instance, Bert-base [14] comprises only an encoder of 12 blocks but without a decoder. Additionally, the number of blocks is configurable. For simplicity, in Figure 2a, we only illustrate the architecture of a single encoder block, which mainly includes two sub-layers: a multi-head self-attention and a fully connected feed-forward network. The simple feed-forward sub-layer consists of two linear transformation operators and an activation operator between them, while the multi-head self-attention sub-layer involves a series of operators with a more intricate structure.

[2]The data and code associated with this paper are openly available at https://github.com/FuncJ.

Table 1: Terminology used throughout the paper

Variable	Description
$batch_{size}$	number of the input sequences in a batch
seq_{len}	length (tokens) of an input request sequence
d_{model}	vector length of a token (hidden dimension)
h	number of attention heads
d_k	vector length of each attention head after splitting

2.2 Attention Module

The self-attention mechanism serves as a core building block in Transformers [14, 40]. It computes the representation of a sequence by capturing relationships between tokens at different positions. Table 1 summarizes the parameters used in the paper. Given a sequence that undergoes tokenization and embedding, a matrix of shape $seq_{len} \times d_{model}$ is produced, where seq_{len} is the number of tokens, and d_{model} is the length of a word vector. This matrix is then fed into the attention module, undergoing a three-step calculation, as illustrated in Figure 2b. 1) It requires the multiplication of three weight matrices, $\mathbf{W_Q}$, $\mathbf{W_K}$, and $\mathbf{W_V}$, to generate queries ($\mathbf{Q}$), keys ($\mathbf{K}$), and values ($\mathbf{V}$). 2) Following Formula 1 , the process involves: ① computing the dot product of matrices $\mathbf{Q}$ and $\mathbf{K^T}$ to obtain the attention matrix [19], ② applying a mask to the attention matrix to conceal certain token-to-token relationships, ③ softmax normalization along the rows of the matrix to derive the attention probability matrix [19], and ④ multiplying the matrix by $\mathbf{V}$ to obtain the output. This process is conducted in parallel with h attention heads, where $d_k = d_{model} \ / \ h$ after splitting. 3) The data from multiple attention heads is concatenated and linearly transformed to yield the attention module's output.

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{QK^T}}{\sqrt{d_k}})\mathbf{V} \tag{1}$$

2.3 Characteristic of Workloads

Modern software systems commonly adopt a modular and hierarchical design approach to reduce code coupling and development costs [8]. They typically decompose the workflow of the attention module into several segments and seek support from operator libraries that are highly optimized for specific architectures. As depicted in Figure 2c, we categorize the matrix operations involved in the module into four classes and aim to analyze the characteristics of the workloads.

2.3.1 Matrix Multiplication. The linear transformation illustrated in steps 1) and 3) essentially involves a GEMM (GEneral Matrix Multiplication) routine. The optimization of GEMM is a well-explored area, with classical dense linear algebra libraries such as OpenBLAS [44] and BLIS [31] providing high-performance implementations. Small values of $batch_{size}$ and seq_{len} may lead to small or irregular-shaped GEMM [7, 47], as shown in Figure 3a, and recent research, including works like LIBSHALOM [47] and LIBXSMM [22], has delved into this issue thoroughly. Therefore, the paper's focus is not the acceleration of a single large-scale or irregular matrix multiplication operator.

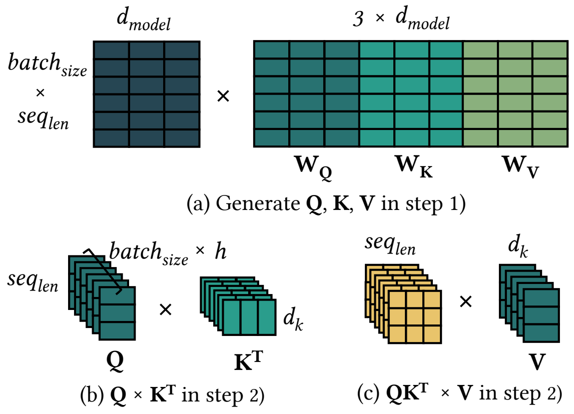
d_{model}
$3 \times d_{model}$
$batch_{size} \times seq_{len}$
×
$\mathbf{W_Q}$
$\mathbf{W_K}$
$\mathbf{W_V}$
(a) Generate **Q**, **K**, **V** in step 1)
seq_{len}
$batch_{size} \times h$
×
d_k
Q
$\mathbf{K^T}$
(b) $\mathbf{Q} \times \mathbf{K^T}$ in step 2)
seq_{len}
d_k
×
V
(c) $\mathbf{QK^T} \times \mathbf{V}$ in step 2)

Figure 3: Key workloads in the attention module.

2.3.2 Batched MM. Transformers use h attention heads, allowing the model to learn representations from different subspaces [40], as seen in models like GPT-2, where h=12 [37]. Considering the batch processing technique during model inference, the process represented by Formula 1 is computed $batch_{size} \times h$ times. Batched matrix multiplication dominates the overall computation, with the shape being illustrated in Figures 3b and 3c. Unlike the matrix workload in linear transformation, developing parallelism solely within a single MM (intra-MM) is insufficient for utilizing multi-core CPUs. Particularly in natural language processing tasks, the range of seq_{len} variation can be substantial [15, 16], and small-scale MMs generated by short sequences result in very low computational efficiency. Therefore, batch parallelization of MMs (inter-MM) is highly essential. Moreover, in the case of variable sequence lengths, the on-chip cache requirements for MMs vary, posing a challenge in kernel fusion and devising parallel algorithms.

2.3.3 Point-wise Operation. Mask and Scale are typically implemented as matrix addition and scalar multiplication, respectively. They are both memory-intensive point-wise operations and lack opportunities for data reuse. Previous work has optimized them through straightforward fusion strategies [53].

$$y_i = \frac{e^{x_i - max}}{sum} = \frac{e^{x_i - max}}{\sum_{j=1}^{seq_{len}} e^{x_j - max}} \tag{2}$$

2.3.4 Reduction. Softmax reduces the values of the attention matrix to the interval $[0, 1]$, where the shape of each attention matrix is $seq_{len} \times seq_{len}$. For each row vector $\mathbf{X}$ of length seq_{len} in the matrix, softmax [11, 33] is performed according to Formula 2. Here, max is the maximum element in the vector, and sum is the row sum after taking the exponentials of the vector elements. Four iterations are required on $\mathbf{X}$ to obtain each output y_i: find the row maximum, exponentiate each element, accumulate the row sum, and normalize by dividing by sum. It is evident that the softmax operator features significant memory access overhead, and its serial workflow introduces complex and strict data dependencies.

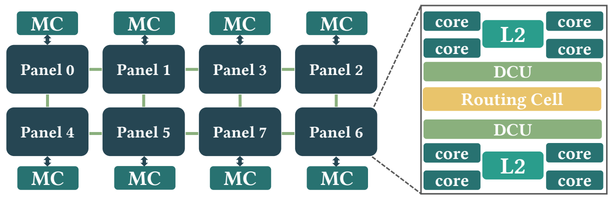

Figure 4: A high-level view of the Phytium 2000+ architecture.

2.4 Hierarchical Memory System

High-performance multi-core processors typically adopt a hierarchical on-chip cache architecture, leading to inconsistent access latency. We illustrate this design using the Phytium 2000+ [18] processor, which serves as a building block for China's next-generation exascale computer [49] as an example. As depicted in Figure 4, this processor organizes every four cores into a core group, where each core has an L1 cache size of 32KB, and the four cores share a 2MB L2 cache [20, 52]. Eight cores form a panel, and the whole chip is composed of eight panels. Accessing the local L1 and L2 caches yields the fastest speeds, with a latency difference of 4 to 5 times between them [20, 52]. When data is not in the local L2 cache, the access latency depends on the distance between the data and the CPU core. For instance, accessing data in panel 6 from the CPU core in panel 0 incurs the highest on-chip data access overhead, approximately 10 times that of accessing the local L2 cache [20]. When data is off-chip, the memory access latency becomes significantly higher.

3 CHALLENGES AND OVERVIEW

3.1 Problem Scope

Given the insights from the preceding analysis, we concentrate our optimization efforts on the procedure illustrated in step 2) of Figure 2. Termed scaled dot-product attention (SDPA) [40], this process is encapsulated as a function primitive in deep learning frameworks like PyTorch [35], serving developers in constructing transformer-based neural networks. Based on our comprehension of existing research, we summarize three technical perspectives for optimizing and enhancing the performance of the SDPA operator.

3.2 Optimization Challenges

3.2.1 How do we devise computation kernels for SDPA with operator fusion techniques. A fundamental direction is using the outer product formula to construct a GEMM kernel [44] as a starting point. Subsequently, while ensuring the correctness of data dependencies, we gradually integrate functionalities such as softmax into the GEMM kernel. Specifically, we split these steps into sub-steps based on their memory access characteristics and fuse them into the GEMM kernel, generating multiple computational kernels with different functionalities. This allows us to use these customized kernels to describe the computational logic of SDPA. Additionally, we typically use data packing techniques to rearrange data into contiguous buffers to ensure continuous data access during kernel computation. A combination of on-the-fly packing techniques [47] and thoughtful data format design is necessary to mitigate associated costs. Therefore, we face multiple challenges, such as data formats, complex memory access patterns of operators, and data dependencies.

3.2.2 How do we exploit intra- and inter-MM parallelism based on the characteristics of the workload. The computational complexity of the SDPA varies with different sequence lengths and batch sizes. This results in varying intra- and inter-MM parallelism. Recent research has initiated preliminary attempts at optimizing batched GEMM on CPUs, often imposing constraints on the matrix shape. For instance, LIBXSMM [22] requires the three dimensions of an MM to satisfy $\sqrt[3]{MNK} \leq 32$, a condition rarely met in practical scenarios. Other approaches either lack the capability to dynamically adjust task partitioning strategies for varying sequence lengths [2], thus failing to effectively utilize on-chip caches, or resort to table lookup methods that cannot cover all cases [25]. This indicates the need for an adaptive parallel algorithm capable of allocating tasks and parallelizing threads in real-time based on workload characteristics. Simultaneously, leveraging the hierarchical memory system for locality-aware task mapping is crucial.

3.2.3 How can we derive algorithmic parameters using analytical models to achieve portability. Deducing parameters rationally based on architectural disparities and workload variations is foundational for ensuring high algorithmic performance. Additionally, even though the kernel is coded in assembly language on a specific instruction set architecture, it is essential to abstract critical parameters of the algorithm and guide its design using analytical models to avoid binding parallel algorithms to specific platforms.

3.3 Overview

3.3.1 High-level view. Figure 5 illustrates a high-level perspective of MEATTEN's algorithm design. The algorithm's inputs **Q**, **K**, **V**, and output **O** are all four-dimensional tensors with a shape of $batch_{size} \times h \times seq_{len} \times d_k$. We assume that each tensor is stored in row-major order, aligning with the data format conventions of mainstream deep learning frameworks such as PyTorch [35]. MEATTEN draws inspiration from Goto's blocked algorithm [21, 39] and is primarily composed of a seven-layer nested loop, with the innermost loop, known as the micro-kernel, being written in assembly instructions. The crucial optimizations of the algorithm encompass micro-kernel, loop layouts, and parallelization, aiming to enhance data reuse while improving parallelism.

3.3.2 Roadmap. In the subsequent sections, we will elaborate on the technical details of MEATTEN. We begin by elucidating how data reuse is achieved on the hierarchical cache architecture through loop tiling and permutation (Section 4). An analytical model is provided to guide the optimization of the loop layout. Next, we introduce the design principles of micro-kernels within the loop, complemented by the enhancement in data format (Section 5). Finally, we discuss the parallelization techniques, encompassing task partition, distribution, and thread mapping (Section 6). We describe the algorithm's design using single floating-point data (fp32), but the methodology can be applied to other precisions such as fp16, fp64, and int16.

4 LOOP OPTIMIZATION

Loop tiling and permutation are two paramount loop optimization techniques with crucial implications for improving data reuse [29]. For conciseness, Figure 6 exclusively presents one head of SDPA, with its workflow and components mirroring Figure 5, excluding the L1 loop. Our analysis initiates from the two micro-kernels of loop L6 and L7 in Figure 5, progressively elucidating how we determine the order of the nested loop. Subsequently, we deduce the tiling parameters of the algorithm based on an analytical model.

4.1 Loop Order

4.1.1 $\mathbf{Q} \times \mathbf{K}^{\mathbf{T}}$. Kernel one primarily computes the product of $\mathbf{Q}$ and $\mathbf{K}^{\mathbf{T}}$, simultaneously fusing two point-wise operations, namely scale and mask, along with a substep of the softmax involving finding the row-wise maximum (max). Detailed design considerations regarding the functionality of kernels will be explained in Section 5. Here, our focus is solely on how to manipulate data reuse.

In the L6 loop, we access two panels from $\mathbf{Q}$ and $\mathbf{K}^{\mathbf{T}}$, each of size $mr \times d_k$ and $d_k \times nr$, as depicted by the dashed box in Figure 6. During the loop, we load data from main memory for $\mathbf{Q}$ and $\mathbf{K}^{\mathbf{T}}$, compute their product, and store the product to a tile of size $mr \times nr$ directly in registers. Along the reduction dimension d_k, this tile accumulates partial results [27] generated by outer-product computations for each $mr \times 1$ column slice and $1 \times nr$ row slice, achieving register-level data reuse. Transitioning to the L5 loop, it is noteworthy that in the first iteration of the L5 loop, we opt to pack the elements of the $d_k \times nr$ panel from $\mathbf{K}^{\mathbf{T}}$ to a linear buffer. Employing an online packing method [42, 47], we achieve instruction overlap between data packing and micro-kernel computation, thereby amortizing the overhead of memory access instructions of data packing. d_k is typically a small value (64 in Bert [14]), and nr is not large due to register constraints. Therefore, the $d_k \times nr$ buffer is much smaller than the capacity of the L1 cache. Hence, in subsequent iterations of the L5 loop, we can reuse this buffer along the $b1$ dimension, fetching each $mr \times d_k$ panel of $\mathbf{Q}$ from main memory until the $b1 \times d_k$ block $\mathbf{Q}$ fills the L1 cache. Following this, in the L4 loop, we reuse block $\mathbf{Q}$ from the L1 cache along the $b2$ dimension until

```
Input Q, K, V and Output O: [$batch_{size}$, $h$, $seq_{len}$, $d_k$]
L1 : parallel for $b = 0$; $b < batch_{size} \times h$; $b$ += 1 do
L2 :   parallel for $i = 0$; $i < seq_{len}$; $i$ += $b1$ do
L3 :     for $j = 0$; $j < seq_{len}$; $j$ += $b2$ do
L4 :       for $jj = j$; $jj < j+b2$; $jj$ += $nr$ do
L5 :         for $ii = i$; $ii < i+b1$; $ii$ += $mr$ do
               pack $\mathbf{K}^{\mathrm{T}}[0{:}d_k, jj{:}jj+nr]$ if $ii == i$
L6 :           for $kk = 0$; $kk < d_k$; $kk$ += 4 do            ▲ micro-kernel
                 $\mathbf{S}[ii{:}ii+mr, jj{:}jj+nr]$ += $\mathbf{Q}[ii{:}ii+mr, kk{:}kk+4]$ ×
Kernel-One             $\mathbf{K}^{\mathrm{T}}[kk{:}kk+4, jj{:}jj+nr]$ (Fused Scale, Mask, Max)
L4':       for $p = j$; $p < j+b2$; $p$ += $b3$ do
L5':         for $ii = i$; $ii < i+b1$; $ii$ += $mr$ do
L6':           for $jj = 0$; $jj < d_k$; $jj$ += $nr$ do
L7 :             for $kk = p$; $kk < p+b3$; $kk$ += 4 do          ▲ micro-kernel
                   $\mathbf{O}[ii{:}ii+mr, jj{:}jj+nr]$ += $\mathbf{S}[ii{:}ii+mr, kk{:}kk+4]$ ×
Kernel-Two               $\mathbf{V}[kk{:}kk+4, jj{:}jj+nr]$ (Fused Exp, Sum, Norm)
```

Figure 5: Algorithm design of MEATTEN.

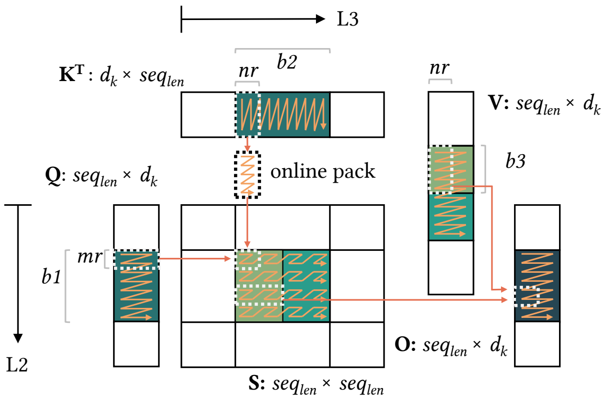
L3
$\mathbf{K}^{\mathbf{T}}$: $d_k \times seq_{len}$
nr
b2
nr
V: $seq_{len} \times d_k$
online pack
Q: $seq_{len} \times d_k$
b3
mr
b1
L2
O: $seq_{len} \times d_k$
S: $seq_{len} \times seq_{len}$

Figure 6: A high-level view of MEATTEN

data of the $d_k \times b2$ block $\mathbf{K}^{\mathbf{T}}$ and $b1 \times b2$ block $\mathbf{S}$ can fit into the L2 cache.

In iterations of the L4 loop, we pack each $d_k \times nr$ panel of $\mathbf{K}^{\mathbf{T}}$ into the previously allocated buffer to achieve low memory space consumption. Notably, the old buffer can reside in the L1 cache. It significantly reduces memory write overhead, as there is no need to pack data into a newly allocated cold buffer outside the cache. Meanwhile, we allocate a contiguous memory space of $b1 \times b2$ for the runtime-generated data block $\mathbf{S}$. The space serves as the output for kernel one and is subsequently fed into kernel two as input. It is expected to be held in the L2 cache and reused in each iteration of the L3 loop. Moreover, this buffer can also be reused along the $batch_{size}$ and h dimensions for a thread when dealing with different attention heads. We empirically demonstrate that buffer reuse has a highly positive impact on performance.

4.1.2 $\mathbf{S} \times \mathbf{V}$. Kernel two computes the product of $\mathbf{S}$ and $\mathbf{V}$, concurrently fusing the remaining three substeps of softmax, namely exponentiation (exp), the sum of rows (sum), and normalization (norm). At the L7 loop, we access panels of $\mathbf{S}$ and $\mathbf{V}$, each of size $mr \times b3$ and $b3 \times nr$, respectively. The $mr \times b3$ panel is respected to reside in the L2 cache, while the $b3 \times nr$ panel needs to be loaded from main memory. At the L6' loop, we traverse the block $\mathbf{V}$ of size $b3 \times d_k$, ensuring that the L1 cache can hold the block. Unlike Goto's algorithm, we do not pack each discontinuous $b3 \times nr$ panel of $\mathbf{V}$. Although we cannot continuously read data at the L7 and L6' loops, we can reuse the data block $\mathbf{V}$ residing in the L1 cache during each L5' loop iteration. Our approach avoids any performance degradation due to discontinuous memory access. Instead, it leverages the L1 cache efficiently without introducing any memory write overhead.

4.2 Tiling Size

The preceding research [21] indicates that the key to selecting the block algorithm's parameters lies in the effective utilization of L2 and L1 cache. We will provide the choices for the parameters mr and nr in Section 5, which are related to the micro-kernel design and the register capacity. At the L2 loop, $\mathbf{Q}$ is partitioned into blocks of $b1 \times d_k$, which can be accommodated in the L1 cache when iterating loop L4. Meanwhile, space needs to be reserved for $\mathbf{K}^{\mathbf{T}}$ and $\mathbf{S}$ of size $d_k \times nr$ and $b1 \times nr$, respectively. Thus, we have inequality 3.

Similarly, at the L3 loop, $\mathbf{K^T}$ and $\mathbf{S}$ are divided into blocks of size $d_k \times b2$ and $b1 \times b2$. We expect the L2 cache to hold these blocks, thus imposing constraint 4. The L4' loop partitions $\mathbf{V}$ into blocks of size $b3 \times d_k$ that can not exceed the size of the L1 cache and be reused during the iteration of loop L5', satisfying constraint 5. The constraint inequalities 3-5 derive $b1$, $b2$, and $b3$ boundary values, respectively. We need minor empirical fine-tuning of them, which is consistent with BLIS [31].

$$b1 \times d_k + d_k \times nr + b1 \times nr \;<\; C_{L1} \tag{3}$$
$$b1 \times d_k + d_k \times b2 + b1 \times b2 \;<\; C_{L2} \tag{4}$$
$$b3 \times d_k + mr \times b3 + mr \times d_k \;<\; C_{L1} \tag{5}$$

5 MICRO-KERNEL DESIGN

MEATTEN has two types of micro-kernels designed for computing $\mathbf{Q} \times \mathbf{K^T}$ and $\mathbf{S} \times \mathbf{V}$ while fusing non-matmul steps such as softmax. Micro-kernels are built upon the ARMv8 architecture, which provides 32 128-bit vector registers (V0-V31) and fused multiply-accumulate instructions (FMA). The micro-kernels aim to fully exploit the modern CPU's out-of-order instruction execution ability by employing classic instruction-level parallelism [23] techniques such as loop unrolling and instruction scheduling.

5.1 Online Softmax

The softmax introduces intricate data dependencies because it can only be performed once all elements in a row of $\mathbf{S}$ are computed. We introduce an online softmax [12] method and then decompose its process into four substeps, seamlessly integrating them forward and backward into the computation of $\mathbf{Q} \times \mathbf{K^T}$ and $\mathbf{S} \times \mathbf{V}$, as illustrated in Figure 7. Considering a simple scenario with five matrix blocks: $\mathbf{Q}$, $(\mathbf{K}^{(1)})^\mathbf{T}$, $(\mathbf{K}^{(2)})^\mathbf{T}$, $\mathbf{V}^{(1)}$, and $\mathbf{V}^{(2)}$, we seek to calculate softmax($\mathbf{Q} \times \left[\, (\mathbf{K}^{(1)})^\mathbf{T} \; (\mathbf{K}^{(2)})^\mathbf{T} \right]) \times \begin{bmatrix} \mathbf{V}^{(1)} \\ \mathbf{V}^{(2)} \end{bmatrix}$ to obtain the output $\mathbf{O}$.

This process can be decomposed into two stages. In the first stage, we calculate the product $\mathbf{S}^{(1)}$ of $\mathbf{Q}$ and $(\mathbf{K}^{(1)})^\mathbf{T}$, simultaneously identifying the local maximum $\mathbf{m}^{(1)}$ for each row in $\mathbf{S}^{(1)}$. Then, we exponentiate the matrix $\mathbf{S}^{(1)}$ in place and obtain the sum of each row, denoted as $\mathbf{sum}^{(1)}$. Finally, the matrix multiplication of $\mathbf{S}^{(1)}$ and $\mathbf{V}^{(1)}$ yields $\mathbf{O}^{(1)}$. It is notable that we do not normalize each row of $\mathbf{O}^{(1)}$ by dividing by the corresponding elements of $\mathbf{sum}^{(1)}$. Therefore, the first stage is formulated as follows:

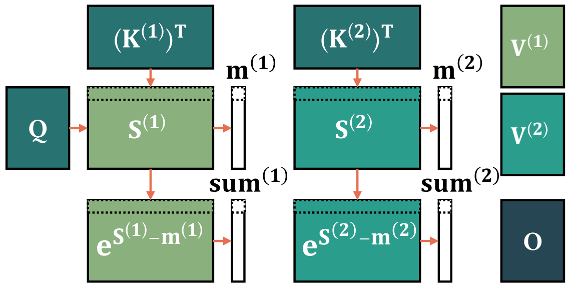

Figure 7: The workflow of the online softmax

$$\mathbf{S}^{(1)}_{i,\,j} = \sum_k \mathbf{Q}_{i,\,k} \times (\mathbf{K}^{(1)})^\mathbf{T}_{k,\,j},\;\; \mathbf{m}^{(1)}_i = \max_j \mathbf{S}^{(1)}_{i,\,j}$$
$$\mathbf{S}^{(1)}_{i,\,j} = e^{\mathbf{S}^{(1)}_{i,\,j} - \mathbf{m}^{(1)}_i},\;\; \mathbf{sum}^{(1)}_i = \sum_j \mathbf{S}^{(1)}_{i,\,j}$$
$$\mathbf{O}^{(1)}_{i,\,j} = \sum_k \mathbf{S}^{(1)}_{i,\,k} \times \mathbf{V}^{(1)}_{k,\,j}$$

In the second stage, a similar computation is performed for $\mathbf{Q}$ and $(\mathbf{K}^{(2)})^\mathbf{T}$, resulting in the product $\mathbf{S}^{(2)}$. While reducing the value $\mathbf{m}^{(2)}$ for each row of $\mathbf{S}^{(2)}$, a comparison is made with the values of $\mathbf{m}^{(1)}$ to determine the global maximum. The exponentiated result of $\mathbf{S}^{(2)}$ is then multiplied by $\mathbf{V}^{(2)}$ to obtain $\mathbf{O}^{(2)}$. Note that each element of $\mathbf{sum}^{(2)}$ is the sum of each row of $\mathbf{S}^{(2)}$, augmented by the updated $\mathbf{sum}^{(1)}$. Thus, the final output $\mathbf{O}$ is computed as $\mathbf{O}^{(1)}$ multiplied by $\mathbf{e}^{\mathbf{m}^{(1)}-\mathbf{m}^{(2)}}$ plus $\mathbf{O}^{(2)}$, then divided by $\mathbf{sum}^{(2)}$. Therefore, we have:

$$\mathbf{S}^{(2)}_{i,\,j} = \sum_k \mathbf{Q}_{i,\,k} \times (\mathbf{K}^{(2)})^\mathbf{T}_{k,\,j},\;\; \mathbf{m}^{(2)}_i = \max(\mathbf{m}^{(1)}_i,\, \max_j \mathbf{S}^{(2)}_{i,\,j})$$
$$\mathbf{S}^{(2)}_{i,\,j} = e^{\mathbf{S}^{(2)}_{i,\,j} - \mathbf{m}^{(2)}_i},\;\; \mathbf{sum}^{(2)}_i = \mathbf{sum}^{(1)}_i \times e^{\mathbf{m}^{(1)}_i - \mathbf{m}^{(2)}_i} + \sum_j \mathbf{S}^{(2)}_{i,\,j}$$
$$\mathbf{O}^{(2)}_{i,\,j} = \sum_k \mathbf{S}^{(2)}_{i,\,k} \times \mathbf{V}^{(2)}_{k,\,j}$$
$$\mathbf{O}_{i,\,j} = (\mathbf{O}^{(1)}_{i,\,j} \times e^{\mathbf{m}^{(1)}_i - \mathbf{m}^{(2)}_i} + \mathbf{O}^{(2)}_{i,\,j})/\mathbf{sum}^{(2)}_i$$

5.2 Micro Kernels

We have designed two types of micro-kernels based on the functionalities and memory access characteristics of loops L6 and L7, as indicated by the dashed boxes in Figures 5. They can be regarded as the minimal computational unit that partitions the computation space constructed by the nested loop. As shown in Figure 8, we interpret the design with a 32-bit floating-point data type (FP32), with each vector register capable of storing four FP32 data. The micro-kernels are coded with the ARMv8 ISA with NEON extension [32].

5.2.1 $\mathbf{Q} \times \mathbf{K^T}$. We allocate mr and $nr/4$ registers for reading $\mathbf{Q}$ and $\mathbf{K^T}$, requiring a total of $mr \times nr/4$ registers to store the product results. Notably, the data of matrix $\mathbf{K^T}$ is packed into a contiguous buffer to facilitate vectorization. Additionally, three registers should be reserved for holding the value of $\frac{1}{d_k}$, loading the value of the mask matrix, and storing the row maximum value. It is desirable for the value of mr to be as close as possible to nr to maximize the computation-to-memory ratio (CMR) of the micro-kernels [31]. Ultimately, we set $mr = 5$, $nr = 16$, utilizing all 32 vector registers. In summary, the usage of registers needs to satisfy:

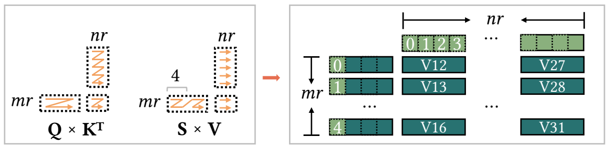

Figure 8: Design of micro-kernel

$$mr + \frac{nr}{4} + \frac{mr \times nr}{4} \leq (32 - 3) \tag{6}$$

$$nr \mod 4 == 0 \tag{7}$$

$$CMR = \frac{2 \times mr \times nr}{mr + nr} \tag{8}$$

The detailed computation process is outlined in Algorithm 1. We calculate the product of $\mathbf{Q}$ and $\mathbf{K}^\mathbf{T}$ using the outer product formula and FMA instructions. Before writing the product back to memory, we scale it by $\frac{1}{d_k}$ and perform addition with the loaded mask matrix data. Crucially, we also use FMAX instructions to derive the maximum value in a row-wise manner (max), which is the first step of softmax. Our micro-kernel allows for a series of memory-intensive operations performed on the product stored in registers, thus can significantly reduce memory access overhead.

The results are typically written back to memory in a row-major or column-major fashion in classical dense linear algebra libraries. To facilitate the computation of $\mathbf{S} \times \mathbf{V}$ in the subsequent micro-kernel, we store the result data as tiles of size $mr \times 4$. The advantage of this data storage format lies in the fact that when the subsequent micro-kernel uses mr registers to read the matrix $\mathbf{S}$, the data will be accessed continuously. This eliminates the need for data packing and facilitates data access using vector instructions.

5.2.2 **S×V**. The second micro-kernel exponentiates the values of $\mathbf{S}$ with the base e prior to performing matrix multiplication, as shown in Algorithm 2. Here, $\mathbf{S}$ represents the output of the preceding $\mathbf{Q} \times \mathbf{K}^\mathbf{T}$, expected to reside in the L2 cache. Exponential operations impose high register demands, preventing their integration into the matrix multiplication computation. However, when computing the exponentiation (exp) of $\mathbf{S}$ by loading data into vector registers, we can fuse the computation of row sums (sum) before writing the exponential values back to memory. Subsequently, we calculate the product of $\mathbf{S}$ and $\mathbf{V}$, denoted as $\mathbf{O}$. A condition must be introduced to ensure normalization is performed only on the last iteration of loop L3. As scalar multiplication (norm) is a memory-intensive point-wise operation, we similarly fuse this step before writing the values of $\mathbf{O}$ stored in registers back to memory. Therefore, the four steps of softmax, except for exp, which can only achieve cache-level fusion, max, sum, and norm, can be fused at the register level, mitigating expensive memory access overhead.

Algorithm 1: Micro-kernel for $\mathbf{Q} \times \mathbf{K}^\mathbf{T}$

```
for $kk = 0 \rightarrow d_k$ $step$ $4$ do
    (V0 - V4) ← $\mathbf{Q}(ii : ii{+}5,\ kk : kk{+}4)$
    (V8 - V11) ← $\mathbf{K}^\mathbf{T}(kk,\ jj : jj{+}16)$
    (V12 - V31) ← FMA((V0 - V4)[0], (V8 - V11))        ▷ Outer Product
    · · ·                                               ▷ Loop unroll
    (V8 - V11) ← $\mathbf{K}^\mathbf{T}(kk{+}3,\ jj : jj{+}16)$
    (V12 - V31) ← FMA((V0 - V4)[3], (V8 - V11))
V5 ← BCAST(1 / $d_k$)
V12 ← FMUL(V12, V5)                                     ▷ Scale
V6 ← LDR(mask($ii$, $jj : jj{+}4$)), V12 ← FADD(V12, V6)    ▷ Mask
V7 ← BCAST(max[$ii$]), V7 ← FMAX(V12, V7)                 ▷ Max
    · · ·
V31 ← FMUL(V31, V5)
V6 ← LDR(mask($ii{+}4$, $jj{+}12 : jj{+}16$)), V31 ← FADD(V31, V6)
V7 ← BCAST(max[$ii{+}4$]), V7 ← FMAX(V31, V7)
```

Algorithm 2: Micro-kernel for $\mathbf{S} \times \mathbf{V}$

```
Exponentiate $\mathbf{S}[ii : ii{+}5,\ p : p{+}b3]$ if $jj == 0$      ▷ Exp and Sum
for $kk = p \rightarrow p + b3$ $step$ $4$ do
    (V0 - V4) ← $\mathbf{S}(ii : ii{+}5,\ kk : kk{+}4)$
    (V8 - V11) ← $\mathbf{V}(kk,\ jj : jj{+}16)$
    (V12 - V31) ← FMA((V0 - V4)[0], (V8 - V11))        ▷ Outer Product
    · · ·                                               ▷ Loop unroll
    (V8 - V11) ← $\mathbf{V}(kk{+}3,\ jj : jj{+}16)$
    (V12 - V31) ← FMA((V0 - V4)[3], (V8 - V11))
if $j == seq_{len} - b2$ and $p == j + b2 - b3$ then
    V5 ← BCAST(1 / sum[$ii$])
    V12 ← FMUL(V12, V5)                                 ▷ Norm
    · · ·
    V5 ← BCAST(1 / sum[$ii{+}4$])
    V31 ← FMUL(V31, V5)
```

6 PARALLELIZATION SCHEME

Our adaptive parallel algorithm aims to exploit intra- and inter-MM parallelism while maintaining load balance and data locality. We employ the widely used shared-memory programming model OpenMP for parallelization. Users can set the thread count T through `OMP_NUM_THREADS`, and our algorithm evenly distributes tasks among T threads to harness hardware parallelism. Simultaneously, the environment variable `GOMP_CPU_AFFINITY` is utilized to establish thread-to-core affinity, which is a crucial step for performance enhancement. Below, we will elucidate the parallelization methodology from three perspectives.

6.1 Task Partition

We parallelize the dimensions $batch_{size}$, h, and seq_{len}, as depicted in loops L1 and L2 of Figure 5. Loops L3 and L4' are not parallelized because the reduction step of matrix multiplication would introduce write-after-write data dependencies at these two loops. It necessitates costly synchronization operations to ensure consistency. We partition the computation space of each SDPA head into $\lceil seq_{len} \ / \ b1 \rceil$ partitions. Therefore, in the case of batch processing, we generate $parts$ partitions, where $parts$ equals $\lceil batch_{size} \times h \times seq_{len} \ / \ b1 \rceil$. When $parts$ is not divisible by T, we iteratively reduce the $b1$ by a step of 1, ensuring an even workload distribution among T threads. This means our parallel algorithm adaptively adjusts the number of parallel threads for each dimension based on $batch_{size}$, seq_{len}, and h. Therefore, we have:

$$parts = \lceil \frac{batch_{size} \times h \times seq_{len}}{b1} \rceil \tag{9}$$

$$parts \bmod T == 0 \tag{10}$$

6.2 Task Distribution

We assign sequential indices starting from 0 to all partitions, resulting in a $part_{index}$ of range $[0, parts - 1]$. Based on the index of each partition, we determine the corresponding thread index number $thread_{index}$ of the thread that is responsible for the partition's execution. The calculation of the $thread_{index}$ is as follows:

$$thread_{index} = part_{index} \bmod T \tag{11}$$

Next, we need to determine the positional information for each partition to facilitate locating the corresponding matrix data during task execution. We introduce two variables: $intra_{index}$ and $inter_{index}$. Here, $intra_{index}$ signifies the task index within a attention head, while $inter_{index}$ indicates which attention head the task belongs. Therefore, we have:

$$intra_{index} = part_{index} \bmod \lceil \frac{seq_{len}}{b1} \rceil \tag{12}$$

$$inter_{index} = \frac{part_{index}}{\lceil \frac{seq_{len}}{b1} \rceil} \tag{13}$$

Consider a simplified scenario with parameters set as $batch_{size}$=1, h=2, and seq_{len}=200. Assuming $b1$=100, then two attention heads will generate four partitions designated by the index range $[0, 3]$. The partition with index 2 is assigned to thread 2 by our method when using four parallel threads. The $inter_{index}$ for this partition is 1, signifying its affiliation with the second attention head. Meanwhile, the $intra_{index}$ is 0, reflecting that this partition is the first partition within the attention head.

6.3 Thread Mapping

Configuring thread-to-core affinity helps mitigate the performance degradation caused by thread migration across CPU cores. This is because when a CPU core undergoes a thread switch, there is a need to reload the data required by the current thread into the on-chip cache. Moreover, mapping multiple threads that share data to multiple cores with a shared cache can reduce the redundancy of data copying and the overhead of remote data access.

Within a single attention head, multiple partitions can share $\mathbf{K^T}$ and $\mathbf{V}$. Therefore, we map the parallel threads within a single attention head to multiple closer CPU cores, such as the CPU cores within the same panel of a Phytium 2000+. There is no opportunity for data sharing among multiple attention heads in SDPA, requiring no special treatment.

7 EXPERIMENTAL SETUP

7.1 Platform Details

We have conducted a comprehensive performance evaluation on three representative ARMv8 multi-core processors: Phytium 2000+ [18], KP920 [43], and ThunderX2 [32]. Table 2 details the hardware specifications for each platform. Notably, Phytium 2000+ employs a shared 2MB L2 cache for every four cores. In contrast, the L2 cache in the other two platforms is individually assigned to each core, with a collective utilization of a large-capacity L3 cache shared among 64 cores. We measure the performance of the SDPA operator and conduct end-to-end inference.

7.2 Competitive Works

We evaluate MEATTEN by comparing it with four existing works, namely:

XNN_F. XNNPACK is a high-performance and prevailing hand-tuned library offered by Google [2]. It utilizes techniques such as thread pooling to exploit hardware parallelism and achieve load balancing. XNNPACK includes a fused SDPA operator, referred to here as XNN_F.

Table 2: Main characteristics of each platform

	Phytium 2000+	KP920	ThunderX2
Peak FP32 GFLOPS	1126.4	2662.4	2559.4
Cores	64	64	64
Frequency	2.2GHz	2.6GHz	2.5GHz
L1 cache	32KB	64KB	32KB
L2 cache	2MB	512KB	256KB
L3 cache	None	64MB	64MB

XNN_NF. Since SDPA is a complex operator formed by stacking multiple sub-steps, we have constructed an SDPA operator using various routines provided by XNNPACK, denoted as XNN_NF here. Each sub-step is implemented by calling an individual parallel routine without fusion.

Ansor. Ansor [54], a submodule of the deep learning compiler TVM [9], is responsible for generating schedules for operators automatically. We run auto-tuning up to 1000 measurement trails per test case, as per the default configuration of Ansor. For an entire deep neural network, we set the number of measurement trials to 1000×n, where n is the number of subgraphs. It is typically sufficient for the search or auto-tuning to converge. We use TVM 0.12.0.

BLIS. BLIS [31] is a classical dense linear algebra library that provides high-performance GEMM routines. As BLIS does not offer softmax routines, we invoke relevant functions from XNNPACK. We parallelize the $batch_{size}$ and h dimensions using OpenMP, with individual GEMM relying on BLIS as the backend.

PyTorch. We use PyTorch [35] version 2.0.0 with BLIS as the backend. This serves as the baseline for our end-to-end experiments.

7.3 Workloads

The number of attention heads h and the hidden dimension d_{model} serve as hyperparameters for transformers. In the Bert-base model [14], with h=12 and d_{model}=768, the resulting d_k is 64. This parameter configuration remains fixed during our evaluation. The work [15] has statistically analyzed sequence lengths in 8 corpora of the GLUE benchmark [41], revealing that, excluding certain outliers, sequence lengths predominantly concentrate within the interval $[0, 1024]$. We have conducted tests on sequences with lengths distributed in the range $[160, 1600]$ to ensure good coverage. We mainly utilize batch sizes of 32 and 64.

7.4 Evaluation Methodology

We perform a warm-up to load the code into the instruction cache for each experimental case; the time taken for this warm-up is not considered. We conduct three repeated tests for each case and calculate the arithmetic mean of the three results to obtain stable performance. We measure performance using GFLOPS as the metric and only consider the floating-point operations of the two batched matrix multiplication in SDPA. Therefore, we have:

$$GFLOPS = \frac{batch_{size} \times h \times (4 \times seq_{len} \times seq_{len} \times d_k)}{time_{cost} \times 1.0e9} \tag{14}$$

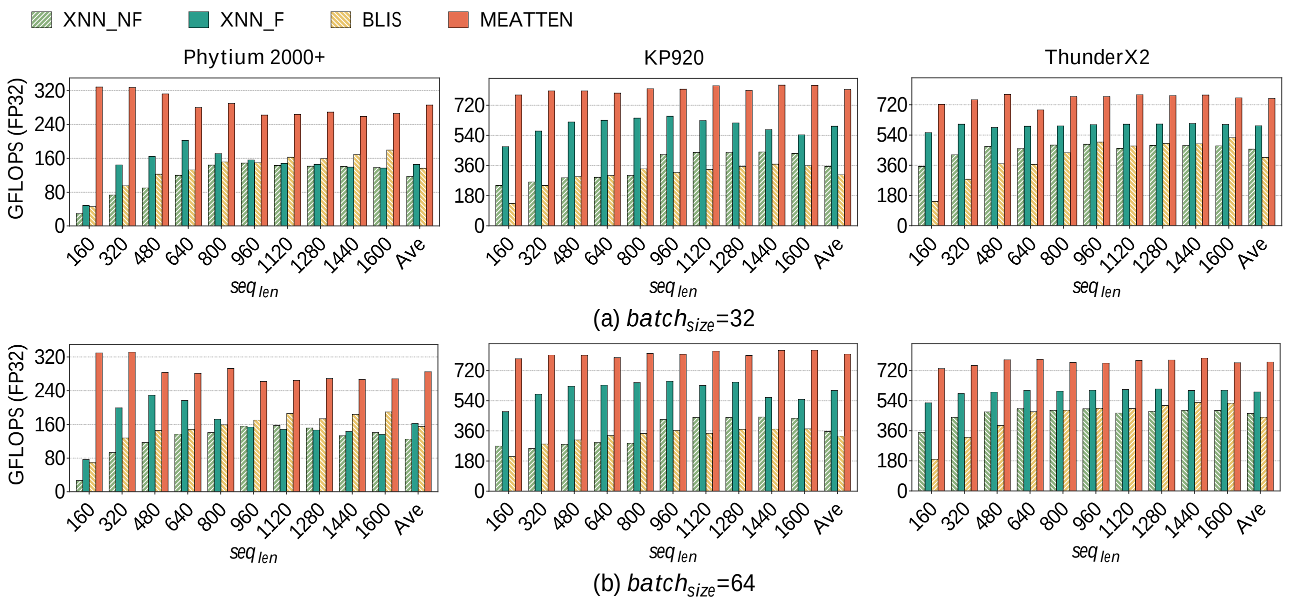

Figure 9: Performance of the SDPA operator using 32 threads. Higher is better. ($h = 12, d_k = 64$)

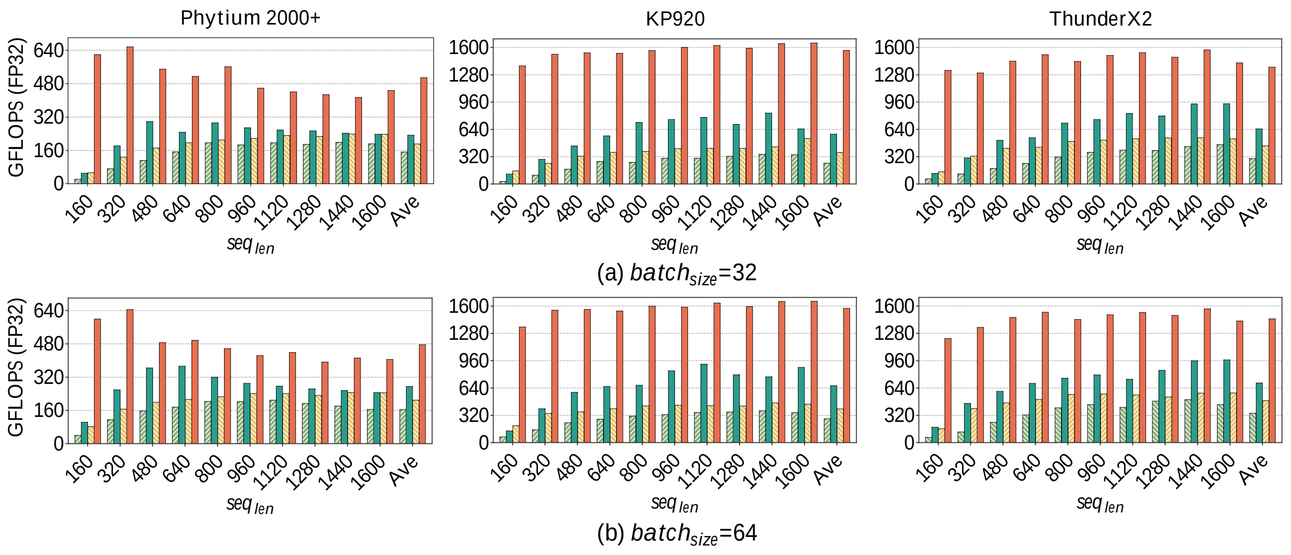

Figure 10: Performance of the SDPA operator using 64 threads. Higher is better. ($h = 12, d_k = 64$)

8 EVALUATION RESULTS

8.1 32 Cores

In this experiment, we measure the parallel performance of the SDPA operator using 32 cores, with each core spawning a thread. We consider two commonly used batch sizes, 32 and 64, and the experimental results indicate that both are sufficient for the tested method to harness the multi-core CPUs effectively.

Figure 9a presents the experimental results of a batch size 32. Compared to the optimal baseline, MEATTEN achieves average speedups of 1.96×, 1.37×, and 1.27× on Phytium 2000+, KP920, and ThunderX2, respectively. The arithmetic mean is used for the average performance measurement and demonstrates the effectiveness of our approach.

XNN_F achieves much superior performance in most scenarios compared to XNN_NF, revealing the importance of fusion. However, notable performance improvement potential remains for XNN_F. Through a thorough study of the XNNPACK codebase, we have discovered the following reasons: Firstly, it uses an offline data pack approach, packing all input data into a contiguous buffer before kernel computation to facilitate vectorization and realize data continuity. This packing approach introduces significant overhead, especially in scenarios involving small MMs generated by short

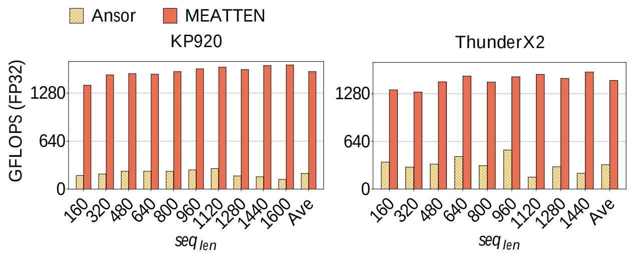

Figure 11: Ansor's performance. Higher is better. ($batch_{size}$ = 32, h = 12, d_k = 64, T = 64)

sequences. We reduce data packing overhead by using optimized loop layouts and a newly designed data format. Secondly, unlike MEATTEN, it does not adopt multidimensional loop tiling, failing to leverage on-chip hierarchical cache storage architecture fully. BLIS achieves performance comparable to that of XNN_NF, and on Phytium 2000+, BLIS even slightly outperforms XNN_NF. However, BLIS and XNN_NF do not exploit fusion strategies and fail to achieve optimal performance. Additionally, compared to the other two processors, MEATTEN achieves the highest performance speedup on Phytium 2000+, which is attributed to our thread mapping strategy that can efficiently utilize the shared L2 cache.

Figure 9b shows the throughput of the SDPA with a batch size of 64. MEATTEN outperforms the optimal method and delivers average speedups of 1.75×, 1.36×, and 1.30× on Phytium 2000+, KP920, and ThunderX2, respectively. The performance results presented in Figure 9b closely align with those in Figure 9a, which demonstrates the excellent performance of our approach across various batch sizes.

8.2 64 Cores

All 64 cores are employed in this experiment to leverage hardware parallelism fully. This implies that KP920 and ThunderX2 utilize two NUMA nodes, posing a challenge to the scalability of the tested methods.

As shown in Figure 10, with a batch size of 32, MEATTEN improves computational throughput by 2.17×, 2.67×, and 2.12× on Phytium 2000+, KP920, and ThunderX2, respectively. For a batch size of 64, MEATTEN achieves acceleration ratios of 1.72×, 2.35×, and 2.07×. The performance of the best-performing XNN_F does not significantly grow with an increase in the number of cores, indicating a notable impact on its performance due to cross-NUMA-node memory access. We emphasize that all methods use the `numactl` command to specify identical memory allocation strategies. MEATTEN, utilizing fusion and thread mapping techniques, effectively reduces the demand for memory bandwidth and remote memory access, achieving excellent scalability.

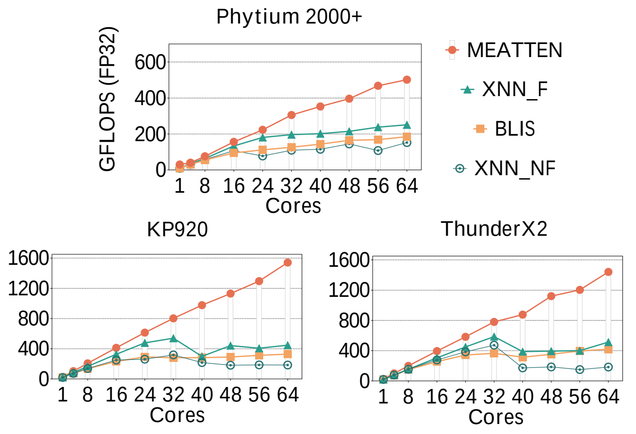

Figure 12: Scalability study of different works. Higher is better. ($batch_{size}$ = 32, h = 12, seq_{len} = 480, d_k = 64)

8.3 Comparison with Ansor

To our knowledge, Ansor currently does not support dynamic shapes. Therefore, we perform 1000 tuning trials for each instance of the SDPA operator with different $batch_{size}$ and seq_{len}. This indicates that we have allocated sufficient budget for Ansor's tuning. In these experiments, we utilize all 64 cores. Ansor does not achieve optimal performance because its search method cannot generate efficient fusion strategies, as shown in Figure 11.

8.4 Scalability

We assess the scalability performance of each method using varying CPU cores, with a single thread assigned to each core. As depicted in Figure 12, MEATTEN and XNN_F exhibit significant performance gains with increasing cores, and MEATTEN demonstrates better performance than XNN_F. We observe sub-optimal scalability of XNNPACK on Phytium 2000+, primarily due to this processor having 8 NUMA nodes, posing significant challenges for memory access optimization. On the other two processors, XNNPACK's performance demonstrates almost linear growth with increased cores when using only one NUMA node. It is when the cores exceeds 32, leading to cross-NUMA-node memory access, that XNNPACK's performance even starts to decline with more parallelism.

8.5 Ablation Study

In this experiment, we analyze the impact of each optimization step on performance, as shown in Figure 13. We focus on analyzing four steps: softmax fusion (V2), MM fusion (V3), tuning of the parameter $b2$ (V4), and buffer **S** reuse (MEATTEN). V1 refers to not using any optimization, while MEATTEN uses all four mentioned steps.

Compared to V1, V2 decomposes the softmax into several sub-steps and fuses them forward and backward into the two matrix multiplications. This significantly reduces memory access overhead, as sub-steps can be fused at the register level. V3 further combines the two matrix multiplications into the same loop, as illustrated in the L3 loop of Figure 5, referred to as MM fusion. This practice effectively exploits the locality between producers and consumers, thus significantly improving performance. V4 fine-tunes the parameter $b2$. By adjusting the value of $b2$, we aim to keep the temporarily generated block **S** in the L2 cache. Finally, the buffer reuse's impact has been evaluated. Experimental results demonstrate that buffer reuse reduces memory consumption and yields a noticeable performance improvement.

8.6 End-to-end Performance

We use Bert-base as a study case and perform end-to-end performance benchmarks on ThunderX2, as shown in Figure 14. We have

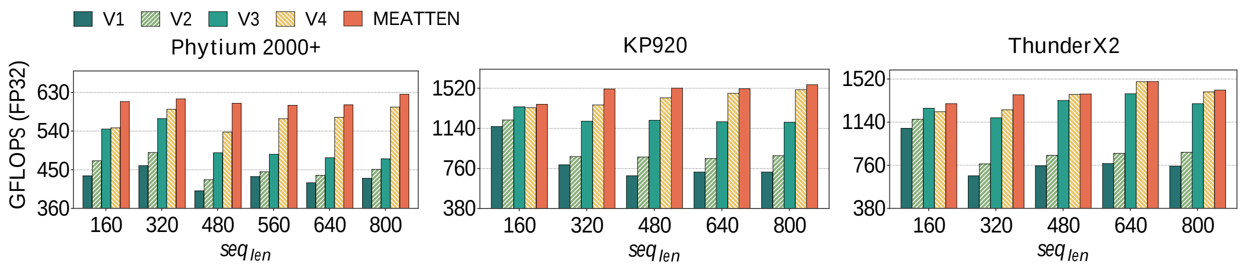

Figure 13: Ablation study on the three platforms. Higher is better. ($batch_{size} = 16, h = 12, d_k = 64, T = 64$)

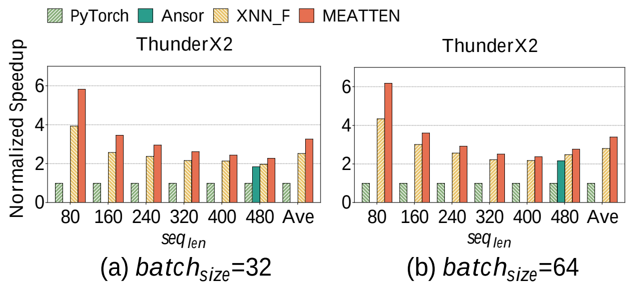

Figure 14: End-to-end performance of Bert-base. Higher is better. ($h = 12, d_k = 64, T = 64$)

Table 3: The normalized inference performance of Vision Transformer. ($h = 12, d_k = 64, T = 64$)

$batch_{size}$	PyTorch	XNN_F	MEATTEN
32	1.0	2.25 ×	2.76 ×
64	1.0	2.49 ×	2.85 ×

observed similar results on the other two platforms. The maximum allowed sequence length for the Bert-base model is 512. Under a batch size of 32, XNN_F and MEATTEN achieve average speedup ratios of 2.52× and 3.26× compared to PyTorch, respectively. With a batch size of 64, XNN_F and MEATTEN improve the throughput by 2.80× and 3.39×, respectively.

We load the Bert-base model from PyTorch and subsequently optimize the entire network using Ansor. We currently consider two batch sizes, 32 and 64, with a sequence length set to 480. Our approach outperforms Ansor by 1.27× and 1.23×, respectively.

We have also benchmarked the performance of Vision Transformer [10]. As shown in Table 3, our method achieved a 2.76 × and 2.85 × acceleration compared to PyTorch.

9 RELATED WORK

The self-attention mechanism [40], serving as a core building block in Transformer-based models [14, 36, 37], is commonly identified as a bottleneck during model inference. The DL stack usually seeks hand-tuned libraries or compilers for low-level optimizations [8, 9, 12, 13, 15, 17, 25, 51, 53–55], aiming to harness hardware capabilities efficiently. This section delineates prior efforts related to this paper from three technical perspectives.

Loop optimization. Loop permutation and tiling are crucial loop optimization techniques with significant implications for enhancing data locality. Chimera [55] and MOpt [29] formalize the data movement volume of a nested loop through analytic models and employ constraint-solving methods to derive the optimal optimization strategy under constraints. Ansor [54] employs an automatic tuning approach, generating extensive schedules for an operator and then evaluating their performance using a cost model to select the most efficient one. Inspired by Goto's blocked algorithm [21], MEATTEN analyzes how to organize the loop layout for data reuse on the on-chip memory hierarchy, from registers to L1 and L2 caches.

Micro-kernel design. Classical dense linear algebra libraries such as OpenBLAS [44] and BLIS [31] typically adopt outer-product-based micro-kernels that can achieve exceptional performance for large-scale matrices. LIBXSMM [22] and LIBSHALOM [47, 48] have undertaken aggressive optimizations for small or irregular-shaped matrices, including key techniques like data packing hiding and instruction scheduling. However, these approaches are limited to MMs. By dissecting the memory access patterns of each step in the attention module, MEATTEN has devised highly efficient micro-kernels capable of fusing multiple memory-intensive steps. Its micro-kernels reduce the need for data packing by coupling it with innovative data formats, thus significantly enhancing performance.

Parallel strategy. XNNPACK [2] utilizes thread pooling to exploit multi-cores. However, its task partition strategy overlooks the hierarchical cache structure and leads to sub-optimal locality. The work [25] employs a table-based method to determine the thread allocation strategy for different workloads, making a trade-off between performance and portability. While LIBXSMM [22] achieves batched parallelism for operators, it imposes restrictions on the shape of operators. MEATTEN's adaptive parallel algorithm can dynamically explore intra- and inter-MM parallelism based on workload characteristics while maintaining data locality.

10 DISCUSSION

MEATTEN has currently implemented support for 128-bit NEON on ARMv8 platforms. However, its technologies can be easily ported to other platforms, including the ARM Scalable Vector Extension (SVE) and the Intel AVX-512 SIMD extension. This entails deriving tiling parameters, such as *b1*, *b2*, and *b3*, specific to the architecture using our analytical model and rewriting micro-kernels with assembly. Our parallelism approach is applicable for accelerating the commonly used batched matrix-multiplication workload in deep learning. Furthermore, MEATTEN is advantageous for compiler frameworks like TVM, as the fused SDPA operator provided by MEATTEN can be utilized as a backend for aggressive low-level optimizations.

11 CONCLUSION

This paper introduces MEATTEN, an open-source library optimizing self-attention performance on ARMv8 multi-cores. MEATTEN presents innovations in loop optimization, micro-kernel design, and parallel algorithms for the attention operator. We conduct performance evaluations for single operators and end-to-end inference. The results indicate that MEATTEN achieves highly competitive performance across various platforms, thread counts, batch sizes, and sequence lengths.

ACKNOWLEDGMENTS

We extend our sincere gratitude to the anonymous reviewers for their constructive feedback, which has been instrumental in enhancing the quality of our work. Additionally, we are indebted to Pengyu Wang for his invaluable assistance and to Xiaoyun Zhang, Jian Wang, and Yanghai Wang for their insightful suggestions. This work is supported by the National Key Research and Development Program of China under Grant No.2022YFB4501702.

REFERENCES

[1] 2023. AWS Gravition. https://aws.amazon.com/cn/ec2/graviton/.
[2] 2023. Google XNNPACK. https://github.com/google/XNNPACK.
[3] 2023. The Top500 list. https://top500.org.
[4] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, et al. 2021. A set of batched basic linear algebra subprograms and LAPACK routines. *ACM Transactions on Mathematical Software (TOMS)* 47, 3 (2021), 1–23.
[5] Sergio Barrachina, Adrián Castelló, Manuel F Dolz, Tze Meng Low, Héctor Martínez, Enrique S Quintana-Ortí, Upasana Sridhar, and Andrés E Tomás. 2023. Reformulating the direct convolution for high-performance deep learning inference on ARM processors. *Journal of Systems Architecture* 135 (2023), 102806.
[6] Hung-Yang Chang, Seyyed Hasan Mozafari, Cheng Chen, James J Clark, Brett H Meyer, and Warren J Gross. 2023. PipeBERT: high-throughput BERT inference for arm big. Little multi-core processors. *Journal of Signal Processing Systems* 95, 7 (2023), 877–894.
[7] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*. 106–116.
[8] Shiyang Chen, Shaoyi Huang, Santosh Pandey, Bingbing Li, Guang R Gao, Long Zheng, Caiwen Ding, and Hang Liu. 2021. Et: re-thinking self-attention for transformer models on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.
[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[10] Zhengsu Chen, Lingxi Xie, Jianwei Niu, Xuefeng Liu, Longhui Wei, and Qi Tian. 2021. Visformer: The vision-friendly transformer. In *Proceedings of the IEEE/CVF international conference on computer vision*. 589–598.
[11] Jaewan Choi, Hailong Li, Byeongho Kim, Seunghwan Hwang, and Jung Ho Ahn. 2022. Accelerating transformer networks through recomposing softmax layers. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 92–103.
[12] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).
[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[15] Jiangsu Du, Jiazhi Jiang, Yang You, Dan Huang, and Yutong Lu. 2022. Handling heavy-tailed input of transformer inference on GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–11.
[16] Jiangsu Du, Jiazhi Jiang, Jiang Zheng, Hongbin Zhang, Dan Huang, and Yutong Lu. 2023. Improving Computation and Memory Efficiency for Real-world Transformer Inference on GPUs. *ACM Transactions on Architecture and Code Optimization* 20, 4 (2023), 1–22.
[17] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.
[18] Jian-Bin Fang, Xiang-Ke Liao, Chun Huang, and De-Zun Dong. 2021. Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+. *Journal of Computer Science and Technology* 36 (2021), 33–43.
[19] Yuan Feng, Hyeran Jeon, Filip Blagojevic, Cyril Guyot, Qing Li, and Dong Li. 2023. MEMO: Accelerating Transformers with Memoization on Big Memory Systems. *arXiv preprint arXiv:2301.09262* (2023).
[20] Wan-Rong Gao, Jian-Bin Fang, Chun Huang, Chuan-Fu Xu, and Zheng Wang. 2023. wrBench: Comparing Cache Architectures and Coherency Protocols on ARMv8 Many-Core Systems. *Journal of Computer Science and Technology* (2023), 1. https://doi.org/10.1007/s11390-021-1251-x
[21] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 1–25.
[22] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. 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*. IEEE, 981–991.
[23] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[24] Jiazhi Jiang, Jiangsu Du, Dan Huang, Zhiguang Chen, Yutong Lu, and Xiangke Liao. 2023. Full-stack Optimizing Transformer Inference on ARM Many-core CPU. *IEEE Transactions on Parallel and Distributed Systems* (2023).
[25] Jiazhi Jiang, Jiangsu Du, Dan Huang, Dongsheng Li, Jiang Zheng, and Yutong Lu. 2022. Characterizing and optimizing transformer inference on arm many-core processor. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.
[26] Qingye Jiang, Young Choon Lee, and Albert Y Zomaya. 2020. The power of ARM64 in public clouds. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 459–468.
[27] HT Kung, Vikas Natesh, and Andrew Sabot. 2021. Cake: matrix multiplication using constant-bandwidth blocks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[28] Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt, Minwen Deng, Xiaoning Wang, Weiguo Liu, Yu Qiao, and Shengzhong Feng. 2019. FeatherCNN: Fast inference computation with TensorGEMM on ARM architectures. *IEEE Transactions on Parallel and Distributed Systems* 31, 3 (2019), 580–594.
[29] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. 2021. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 928–942.
[30] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 229–241.
[31] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software (TOMS)* 43, 2 (2016), 1–18.
[32] Filippo Mantovani, Marta Garcia-Gasulla, José Gracia, Esteban Stafford, Fabio Banchelli, Marc Josep-Fabrego, Joel Criado-Ledesma, and Mathias Nachtmann. 2020. Performance and energy consumption of HPC workloads on a cluster based on Arm ThunderX2 CPU. *Future generation computer systems* 112 (2020), 800–818.
[33] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867* (2018).
[34] Tetsuya Odajima, Yuetsu Kodama, Miwako Tsuji, Motohiko Matsuda, Yutaka Maruyama, and Mitsuhisa Sato. 2020. Preliminary performance evaluation of the Fujitsu A64FX using HPC applications. In *2020 IEEE international conference on cluster computing (cluster)*. IEEE, 523–530.
[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[36] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[38] Mitsuhisa Sato, Yuetsu Kodama, Miwako Tsuji, and Tesuya Odajima. 2021. Co-design and system for the supercomputer "Fugaku". *IEEE micro* 42, 2 (2021), 26–34.
[39] Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic generation of fast BLAS3-GEMM: A portable compiler approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 122–133.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[41] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[42] Pengyu Wang, Weiling Yang, Jianbin Fang, Dezun Dong, Chun Huang, Peng Zhang, Tao Tang, and Zheng Wang. 2023. Optimizing Direct Convolutions on ARM Multi-Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[43] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services. *IEEE Micro* 41, 5 (2021), 67–75.

[44] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th international conference on parallel and distributed systems*. 684–691.

[45] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org.

[46] Shulei Xu, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2022. Arm meets Cloud: A Case Study of MPI Library Performance on AWS Arm-based HPC Cloud with Elastic Fabric Adapter. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 449–456.

[47] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2021. Libshalom: optimizing small and irregular-shaped matrix multiplications on armv8 multi-cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[48] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2024. Optimizing Full-Spectrum Matrix Multiplications on ARMv8 Multi-Core CPUs. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[49] Xin You, Hailong Yang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2019. Performance evaluation and analysis of linear algebra kernels in the prototype tianhe-3 cluster. In *Asian Conference on Supercomputing Frontiers*. Springer, 86–105.

[50] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3 (2015), 14:1–14:33.

[51] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. 2023. ByteTransformer: A high-performance transformer boosted for variable-length inputs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 344–355.

[52] Charles Zhang. 2015. Mars: A 64-core ARMv8 processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–23.

[53] Zining Zhang, Yao Chen, Bingsheng He, and Zhenjie Zhang. 2023. NIOT: A Novel Inference Optimization of Transformers on Modern CPUs. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[55] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. 2023. Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1113–1126.