The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05890-8

SWattention: designing fast and memory-efficient attention for a new Sunway Supercomputer

Ruohan Wu[1] · Xianyu Zhu[1] · Junshi Chen[1] · Sha Liu[1] · Tianyu Zheng[2] · Xin Liu[3] · Hong An[1]

Accepted: 4 January 2024

Abstract
In the past few years, Transformer-based large language models (LLM) have become the dominant technology in a series of applications. To scale up the sequence length of the Transformer, FlashAttention is proposed to compute exact attention with reduced memory requirements and faster execution. However, implementing the FlashAttention algorithm on the new generation Sunway Supercomputer faces many constraints such as the unique heterogeneous architecture and the limited memory bandwidth. This work proposes SWattention, a highly efficient method for computing the exact attention on the SW26010pro processor. To fully utilize the 6 core groups (CG) and 64 cores per CG on the processor, we design a two-level parallel task partition strategy. Asynchronous memory access is employed to ensure that memory access overlaps with computation. Additionally, a tiling strategy is introduced to determine optimal SRAM block sizes. Compared with the standard attention, SWattention achieves around 2.0x speedup for FP32 training and 2.5x speedup for mixed-precision training. The sequence lengths range from 1k to 8k and scale up to 16k without being out of memory. As for the end-to-end performance, SWattention achieves up to 1.26x speedup for training GPT-style models, which demonstrates that SWattention enables longer sequence length for LLM training.

Keywords Artificial intelligence · Attention optimization · Parallel programs · Sunway architecture

1 Introduction

As innovation in deep learning and the number of parameters continues to grow, large language models (LLM) act as promising tools in many natural language processing (NLP) applications such as BERT, GPT, and Llama [1–4], and the Transformer [5] architecture is wildly adopted in many LLMs due to its state-of-the-art performance. A notable trend in recent years is the expansion of the

Extended author information available on the last page of the article sequence length in Transformers. However, scaling up the sequence length poses a great challenge [6], as both the memory and computation complexity of the self-attention layer in the Transformer increase quadratically with the sequence length.

Given the observation that Transformer is bottlenecked by memory access [7] on modern GPUs, FlashAttention [8] has been proposed to address the above issues. The author designed a novel tiling algorithm that computes the softmax operator without requiring access to the whole input of the attention layer. Consequently, FlashAttention eliminates the need to store the large intermediate attention matrix in high bandwidth memory (HBM), and its memory complexity is linear in sequence length. Besides, FlashAttention reduces the data movement between HBM and fast on-chip SRAM by fusing all the attention operations into one kernel, which brings significant speedup over the standard attention. Therefore, FlashAttention has become a foundational technology widely employed in LLM training [9].

Recently, the new generation Sunway Supercomputer that consists of numerous SW26010pro processors has shown great potential in supporting AI-based workloads [10–13]. However, it is non-trivial to apply the FlashAttention algorithm on the new generation Sunway Supercomputer. First, the distinctive architecture of the SW26010pro processor requires a redesign of the FlashAttention algorithm for GPUs. With 6 core groups (CG) on a SW26010pro processor and 64 computing processing elements (CPE) in each CG, it is necessary to design a parallel strategy that evenly distributes the attention computation among all CGs and CPEs. Second, different from the memory hierarchy of GPUs that consists of HBM and fast on-chip SRAM, each SW26010pro processor is equipped with 96GB DRAM and on-chip SRAM called LDM. The limited DRAM bandwidth necessitates the design of a highly efficient memory access strategy.

In this work, we design and implement SWattention to optimize attention computation on the new generation Sunway Supercomputer. First, to maximize the computational capabilities of the SW26010pro processor, we develop a two-level parallel attention algorithm. For the CG level parallelism, we evenly distribute all 2D attention heads among 6 CGs, and each CG computes independently. For the CPE level parallelism, all 64 CPEs within a CG collaboratively compute these attention heads using remote memory access (RMA) between CPEs. Second, to fully utilize the limited DRAM bandwidth, SWattention is implemented with asynchronous direct memory access (DMA) for the overlapping of communication and computation. Third, we propose a tiling strategy to help search the optimal block sizes for block attention computation on LDM. Additionally, to leverage the FP16 performance of the SW26010pro, SWattention is designed with a mixed-precision method using both FP32 and FP16.

The evaluation results show that SWattention computes exact attention with a smaller memory footprint and runs faster. Compared with the standard attention, SWattention achieves around 2.0x speedup for FP32 training and 2.5x speedup for mixed-precision training. When applied to training GPT-style models [2], SWattention offers up to 1.26x speedup for the end-to-end performance. The main contributions of this work can be summarized as follows:

- We design and implement SWattention on the SW26010pro processor and demonstrate its effectiveness.
- We propose a two-level parallel algorithm to evenly distribute attention computation on the SW26010pro processor.
- We design a tiling strategy to find the optimal block sizes on LDM and employ the asynchronous DMA to fully utilize the limited DRAM bandwidth.
- We successfully integrate mixed-precision training technology with SWattention, which further accelerates the computation of attention on the new generation Sunway Supercomputer.

The rest of this paper is organized as follows. Section 2 introduces the background of the Transformer, FlashAttention, and SW26010pro processor. Section 3 describes the detailed implementation of SWattention. Section 4 presents the evaluation results. Section 5 concludes this paper.

2 Background

2.1 Transformer architecture

From here, we use the following variables to describe the shape of attention inputs: B, batch size; N, number of attention heads, S, sequence length; H, head dimension. Figure 1 shows an overview of Transformer-based models. The model

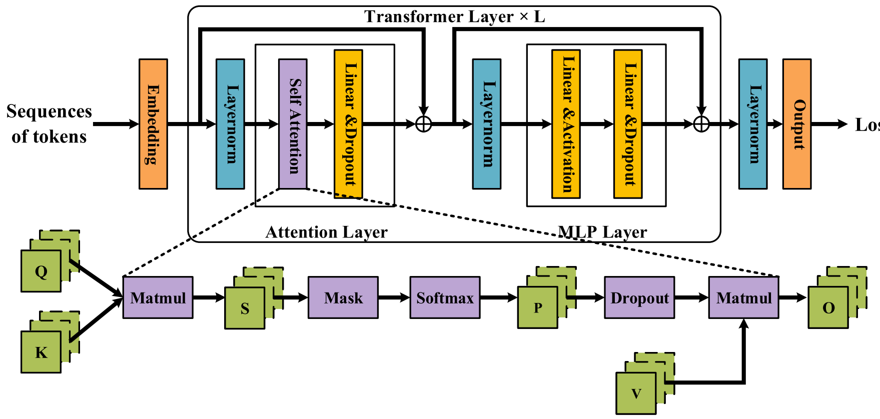


Fig. 1 The architecture of Transformer-based models

mainly consists of L homogeneous Transformer layers. The inputs of the model are sequences of tokens, and the embedding layer converts each token into a vector of shape $[N \cdot H]$. Therefore, the input of the Transformer layers are tensors of shape $[B, S, N \cdot H]$. Each Transformer layer consists of one attention layer and one MLP layer. We can observe that the self-attention layer only accounts for a small portion of the model. However, as the computation complexity of self-attention increases quadratically with S, optimizing the self-attention is of great significance for long sequence lengths.

Before the computation of the self-attention layer, the input tensors **Q, K, V** are reshaped to $[B, N, S, H]$. Each attention head is a matrix of shape $[S, H]$. In the forward pass of self-attention, an intermediate matrix $\mathbf{S}$ of shape $[S, S]$ is generated with:

$$\mathbf{S} = \frac{\mathbf{QK}^T}{\sqrt{H}} \tag{1}$$

Matrix $\mathbf{S}$ then undergoes mask and softmax operators. The other attention matrix $\mathbf{P}$ is the result of the softmax operator. Finally, each attention head multiples the output of the dropout with $\mathbf{V}$, and the shape of the output tensor $\mathbf{O}$ is restored to $[B, N, S, H]$.

2.2 FlashAttention

According to the number of arithmetic operations per byte of memory access, operators in the self-attention can be classified as either compute-bound or memory-bound [14]. The matrix-multiply (GEMM) operation with a large inner dimension can be compute-bound, while softmax, mask, and dropout operators are memory-bound. As the computational speed of modern GPUs such as A100 surpasses memory bandwidth, memory access has become a significant bottleneck for attention computation.

In detail, the attention computation on GPUs involves a massive number of threads executing operations known as kernels. An A100 GPU [15] has 40-80GB HBM and 108 streaming multiprocessors (SM). Each SM has 192KB SRAM whose memory bandwidth is an order of magnitude faster than HBM. Taking the mask operation in Fig. 1 as an example, these kernels load $\mathbf{S}$ from HBM to SRAM and registers, compute, and write the masked output back to HBM. The masked output in HBM then in turn would be loaded as the input of the softmax operator, which brings substantial memory access costs. Although kernel fusion [16–18] can be used to reduce memory access, the intermediate results still need to be saved for backward pass.

Therefore, FlashAttention is proposed to compute exact attention with reduced memory accesses and without the need to store the intermediate results. To compute softmax without access to the whole input, FlashAttention computes attention by blocks with a novel tiling algorithm. This enables kernel fusion for all operations in the self-attention, which significantly accelerates the attention computation. FlashAttention also saves softmax normalization statistics in the forward pass and recomputes attention matrices **S** and **P** in the backward pass; thus, no intermediate results are needed to be written to HBM. Implemented as a fused kernel, FlashAttention has been integrated into many LLM training frameworks, including Megatron-LM [19] and DeepSpeed [20].

2.3 Architecture of the SW26010pro processor

The new generation Sunway Supercomputer is based on the SW26010pro processor. As shown in Fig. 2, the SW26010pro processor consists of 6 CGs. Each CG has one management processing element (MPE), 16GB DRAM and one CPE array with 64 CPEs. The MPE is deployed for controlling CPE array and communication. The CPE supports 512-bit SIMD instruction for FP16, FP32 and FP64 computations. The SW26010pro processor provides 14 TFLOPS FP64 and FP32 performance, and 55 TFLOPS FP16 performance.

These CGs are connected with a network on chip (NoC). There are two ways for launching parallel MPI processes on the new generation Sunway Supercomputer. One is called single-CG mode; another is called all-shared mode. For the single-CG mode, each MPE controls a single CG. As for the all-shared mode, all 6 CPE arrays and 96GB DRAM can be managed by one MPI process.

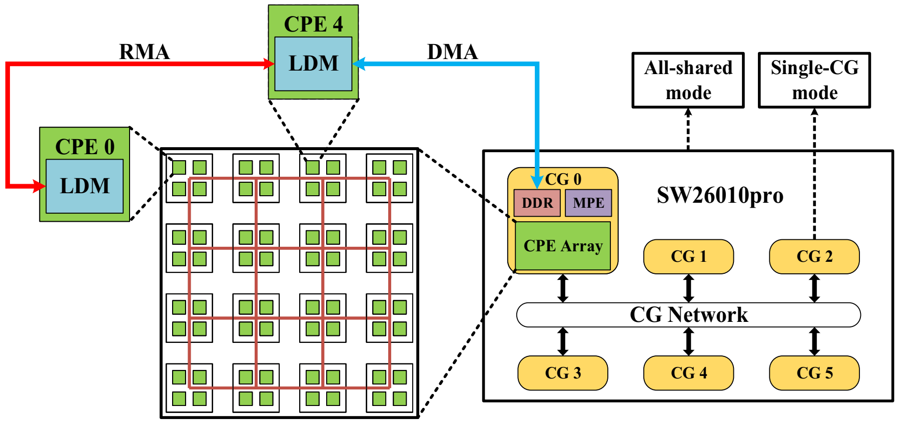


Fig. 2 The architecture of the SW26010pro processor

Considering the large memory consumption of LLM training, SWattention should be designed to run in the all-shared mode.

As depicted in Fig. 2, each CPE is equipped with 256KB SRAM that can be configured into LDM. DMA provides high memory bandwidth for data transfer between DRAM and LDM. CPEs are connected with an intra-CG network, which enables RMA communication between any two CPEs. The global load/store operations can only reach a bandwidth of 0.24 GB/s on the SW26010pro processor, while the theoretical bandwidth of DMA and RMA is 307 GB/s and 460 GB/s, respectively. The unique memory hierarchy [21, 22] of the SW26010pro processor highlights the significance of fully utilizing the memory bandwidth of both DMA and RMA to design highly efficient AI operators.

3 Implementation

3.1 Task partition on SW26010pro

Based on the principles of the FlashAttention algorithm on the CPU-GPU platform, and incorporating the unique computation units and memory hierarchy of

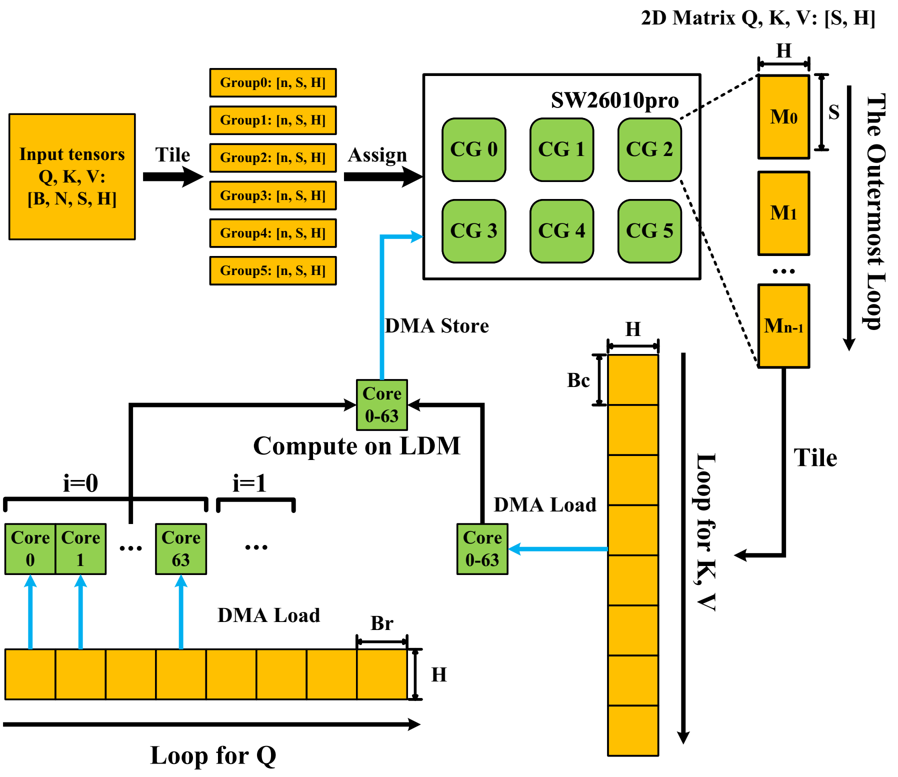


Fig. 3 The overview of SWattention on SW26010pro

the SW26010pro processor, we design and implement SWattention on the new generation Sunway Supercomputer.

As depicted in the upper left of Fig. 3, the input of SWattention is three tensors **Q, K, V** of shape $[B, N, S, H]$. To train LLMs on the new generation Sunway Supercomputer, it is necessary to choose the all-shared mode, since the memory space of single-CG mode is only 16GB. Currently, there are two possible task partitioning strategies. First, each CPE independently computes a matrix of shape $[S, H]$. Second, each CG processes a matrix of shape $[S, H]$. Due to the prevalent use of tensor parallelism technology [19, 23, 24] in current LLM training, the number of attention heads assigned to each processor is typically very small. For instance, if $N = 64$ and the tensor parallelism size is 8, each processor is responsible for processing $\frac{64}{8} = 8$ heads. Besides, due to the limited memory space, the batch size B is often set to a very small value (eg. $B = 1$).

To fully utilize 6 CGs and 384 CPEs of the SW26010pro processor in the all-shared mode, we design a two-level parallel strategy that each CG processes a matrix of shape $[S, H]$. For the CG level parallelism, we divide an input tensor of shape $[B, N, S, H]$ into 6 groups of 2D matrices of shape $[S, H]$ and assign each group of 2D matrices to its corresponding CG. The outermost loop for the task partition is shown in the top-right corner of Fig. 3. At this point, each CG has a batch of **Q, K, V** matrices with a shape of $[n, S, H]$. The outermost loop needs to iterate n times, each time extracting a 2D **Q, K, V** matrix with a shape of $[S, H]$. Therefore, the performance of SWattention can be maximized when $B \times N$ is a multiple of 6 on each processor. For other circumstances in which the attention heads cannot be evenly divided among the 6 CGs, we left the multi-CG collaborative computation tasks for future work.

The computation within each CG is shown at the bottom of Fig. 3. For the CPE level parallelism, SWattention parallelizes over the sequence length dimension of matrix **Q** with shape $[S, H]$ using all 64 CPEs. As we will introduce in Sect. 3.2, we refer to the loop over matrix **Q** as the inner loop and the loop over matrices **K, V** as the outer loop. Matrix **Q** is divided into multiple blocks of shape $[B_r, H]$, while matrices **K** and **V** are divided into multiple blocks of shape $[B_c, H]$. Since the global load/store operations on SW26010pro are extremely inefficient, data transfer between DRAM and LDM is implemented with DMA load/store operations. In the outer loop, each CPE loads the same block of matrices **K, V** with shape $[B_c, H]$. In the outer loop, each CPE loads different blocks of matrices **Q** with shape $[B_r, H]$. Therefore, the outer loop needs to iterate $\lceil \frac{S}{B_c} \rceil$ times, while the inner loop needs to iterate $\lceil \frac{S}{64 \times B_r} \rceil$ times. By loading blocks of **Q, K, V** matrices to LDM, SWattention does not need to store intermediate results with shape $[S, S]$, which significantly reduces memory consumption. Besides, this block-based algorithm minimizes frequent load and store operations between the LDM and DRAM, achieving acceleration by reducing memory access.

3.2 Algorithm design

Algorithm 1 Standard Attention Forward Pass

```
Input: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{S\times H}$ in DRAM, masking function $\mathbf{MASK}$.
Output: $\mathbf{O}$.
1: Load $\mathbf{Q},\mathbf{K}$ by blocks from DRAM, compute $\mathbf{S} = \frac{\mathbf{QK}^T}{\sqrt{H}}$, write $\mathbf{S}$ to DRAM.
2: Load $\mathbf{S}$ from DRAM, compute $\mathbf{S}^{\text{masked}} = \mathbf{MASK}(\mathbf{S})$, write $\mathbf{S}^{\text{masked}}$ to DRAM.
3: Load $\mathbf{S}^{\text{masked}}$ from DRAM, compute $\mathbf{P} = \text{softmax}(\mathbf{S}^{\text{masked}})$, write $\mathbf{P}$ to DRAM.
4: Load $\mathbf{P}$ from DRAM, compute $\mathbf{P}^{\text{dropped}} = \text{dropout}(\mathbf{P})$, write $\mathbf{P}^{\text{dropped}}$ to DRAM.
5: Load $\mathbf{P}^{\text{dropped}}$ and $\mathbf{V}$ from DRAM, compute $\mathbf{O} = \mathbf{P}^{\text{dropped}}\mathbf{V}$, write $\mathbf{O}$ to DRAM.
6: Return $\mathbf{O}$.
```

In this section, we describe the detailed implementation of SWattention forward pass and backward pass. In order to facilitate comparison, we use the same algorithm description format as in FlashAttention [8]. Before we introduce the SWattention algorithm, we describe the implementation of the standard attention in Algorithm 1. It is evident that multiple read and write operations to DRAM are required for the attention computation, resulting in significant memory access overhead.

The SWattention forward pass is shown in Algorithm 2. The input is **Q, K, V** matrices of shape [S, H], and the output of the forward pass is matrix **O** of shape [S, H]. The softmax statistics l and m are initialized in DRAM and saved for backward pass. For the softmax scaling constant, we use $\tau = \frac{1}{\sqrt{H}}$. We also use the causal masking function in SWattention. Each CPE should calculate the row index and column index of each entry of block S_{ij} in S. If the column index is larger than the row index, this entry is set to $-\infty$. The random number generate state R is generated using MPE and is also saved for backward pass. Then, each CPE generates its own random state number using R, CG index, and CPE index. The Br and Bc are row and column block sizes, and the tiling strategy for choosing optimal Br and Bc on SW26010pro will be discussed in Sect. 3.3.

As shown in Fig. 3, SWattention parallelizes over the dimension S of matrices **Q,O** and softmax statistics l, m using all 64 CPEs numbered from 0 to 63. In the inner loop, CPE k computes its own index i and loads the corresponding blocks of **Q,O**,l, m from DRAM to LDM. For example, if $S = 2048$ and $B_r = 16$, there are only $\frac{2048}{16\times64} = 2$ iterations in the inner loop. At the end of the I-th iteration, each CPE stores its own block of l, m and matrix **O** to DRAM. For the outer loop, all CPEs load the same block of **K, V** matrices in the j-th iteration.

Algorithm 2 SWattention Forward Pass

```
Input: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{S\times H}$ in DRAM, free LDM of size $M$, softmax scaling
    constant $\tau \in \mathbb{R}$, masking function $\mathbf{MASK}$, dropout probability $p_{\text{drop}}$.
Output: $\mathbf{O}$, $l$, $m$, $R$.
1: Initialize the random number generate state $R$ using MPE and save to DRAM.
2: Set block sizes $B_c$ and $B_r$ based on the tiling strategy for forward pass.
3: Initialize $\mathbf{O}=(0)_{S\times H} \in \mathbb{R}^{S\times H}$, $l=(0)_S \in \mathbb{R}^S$, $m=(-\infty)_S \in \mathbb{R}^S$ in DRAM.
4: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{S}{B_r} \rceil$ blocks $\mathbf{Q}_0,\cdots,\mathbf{Q}_{T_r-1}$ of size $B_r \times H$ each, and divide
   $\mathbf{K},\mathbf{V}$ into $T_c = \lceil \frac{S}{B_c} \rceil$ blocks $\mathbf{K}_0,\cdots,\mathbf{K}_{T_c-1}$ and $\mathbf{V}_0,\cdots,\mathbf{V}_{T_c-1}$ of size $B_c \times H$ each.
5: Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_0,\cdots,\mathbf{O}_{T_r-1}$ of size $B_r \times H$ each, divide $l$ into $T_r$ blocks
   $l_0,\cdots,l_{T_r-1}$ of size $B_r$ each, divide $m$ into $T_r$ blocks $m_0,\cdots,m_{T_r-1}$ of size $B_r$
   each.
6: for $0 \le j \le T_c - 1$ do
7:     Load $\mathbf{K}_j$, $\mathbf{V}_j$ from DRAM to LDM.
8:     for $0 \le I \le \lceil \frac{S}{B_r \times 64} \rceil - 1$ do
9:         In CPE $k$, calculate index $i = k + I \times 64$.
10:        In CPE $k$, Load $\mathbf{Q}_i, \mathbf{O}_i, l_i, m_i$ from DRAM to LDM.
11:        In CPE $k$, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
12:        In CPE $k$, compute $\mathbf{S}_{ij}^{\text{masked}} = \mathbf{MASK}(\mathbf{S}_{ij})$.
13:        In CPE $k$, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}^{\text{masked}}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij}^{\text{masked}} -$
   $\tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$, $\tilde{l}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
14:        In CPE $k$, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $l_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} l_i +$
   $e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{l}_{ij} \in \mathbb{R}^{B_r}$.
15:        In CPE $k$, compute $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$.
16:        In CPE $k$, compute $\mathbf{O}_i = \text{diag}(l_i^{\text{new}})^{-1}(\text{diag}(l_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i +$
   $e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij}^{\text{dropped}} \mathbf{V}_j)$, and write $\mathbf{O}_i$ to DRAM.
17:        In CPE $k$, $l_i = l_i^{\text{new}}$, $m_i = m_i^{\text{new}}$, and write $l_i, m_i$ to DRAM.
18:    end for
19: end for
20: Return $\mathbf{O}$, $l$, $m$, $R$.
```

The SWattention backward pass is described in Algorithm 3. The **Q, K, V, O** matrices are from the forward pass, and **dO** is the backward pass gradient. The output **dQ, dK, dV** are gradients of input **Q, K, V** matrices. The parallel strategy of the SWattention backward pass is almost the same as that of the forward pass. However, all CPEs load the same block of **K, V** matrices, and the intermediate result $\tilde{\mathbf{dK}}_j$ and $\tilde{\mathbf{dV}}_j$ on LDM cannot be directly written to $\mathbf{dK}_j$ and $\mathbf{dV}_j$ in DRAM. Therefore, we need to sum the gradients $\tilde{\mathbf{dK}}_j$ and $\tilde{\mathbf{dV}}_j$ in all CPEs. As shown in Fig. 4, instead of performing RMA for all data to CPE 0 and then conducting reduction operations, we design a tree-based CPE reduction operation using RMA. This operation includes $log_2 64 = 6$ iterations. In the i-th iteration ($i = 0, 1, 2, 3, 4, 5$), we define $loop = 2^{5-i}$. For CPE k, if $k < loop$, we use RMA to load $\tilde{\mathbf{dK}}_j$ and $\tilde{\mathbf{dV}}_j$ in CPE $k + loop$, and reduce on CPE k. After all 6 iterations,

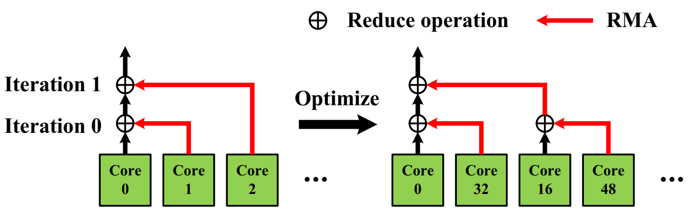


Fig. 4 Reduction of dV and dK using RMA on SW26010pro

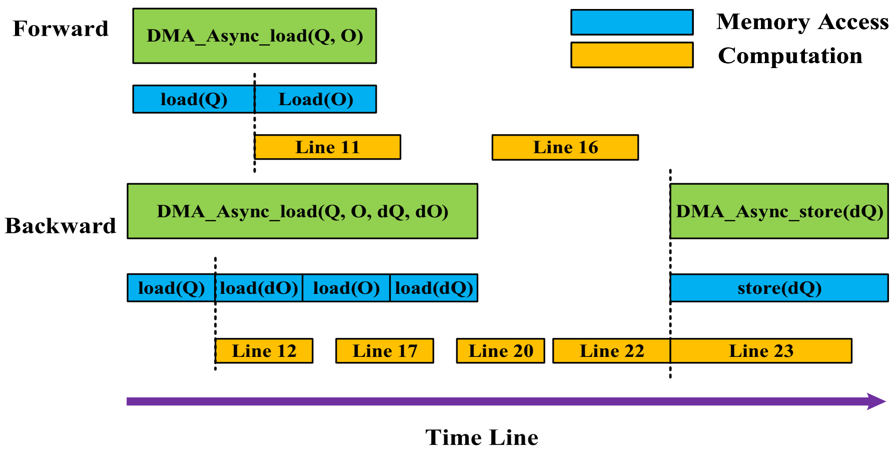


Fig. 5 Computation overlaps with asynchronous memory access

all gradients are reduced on CPE 0. Then, we use DMA to store $\mathbf{d\tilde{K}}_j$ and $\mathbf{d\tilde{V}}_j$ to DRAM on CPE 0.

Moreover, the computation in both forward pass and backward pass can overlap with asynchronous DMA. As shown in Fig. 5, once the matrix **Q** has been loaded into LDM, the computation in line 11 of Algorithm 2 can overlap with loading matrix **O**. In the backward pass algorithm, we find that line 12 needs matrix **Q**, line 17 needs matrix **dO**, line 20 needs matrix **O**, and line 22 needs matrix **dQ**. Therefore, these four matrices are loaded asynchronously in the same order. When line 22 finishes, we need to write **dQ** to DRAM, and the asynchronous DMA store operation can also overlap with the computation in line 23.

Algorithm 3 SWattention Backward Pass

```
Input: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V},\mathbf{O},\mathbf{dO} \in \mathbb{R}^{S\times H}$ in DRAM, vectors $l, m \in \mathbb{R}^S$ in DRAM,
    free LDM of size $M$, softmax scaling constant $\tau \in \mathbb{R}$, masking function $\mathbf{MASK}$,
    dropout probability $p_{\text{drop}}$, random number generate state $R$ from the forward pass.
Output: $\mathbf{dQ}$, $\mathbf{dK}$, $\mathbf{dV}$.
 1: Set the random number generate state to $R$.
 2: Set block sizes $B_c$ and $B_r$ based on the tiling strategy for backward pass.
 3: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{S}{B_r} \rceil$ blocks $\mathbf{Q}_0, \cdots, \mathbf{Q}_{T_r-1}$ of size $B_r \times H$ each, and divide
    $\mathbf{K},\mathbf{V}$ into $T_c = \lceil \frac{S}{B_c} \rceil$ blocks $\mathbf{K}_0, \cdots, \mathbf{K}_{T_c-1}$ and $\mathbf{V}_0, \cdots, \mathbf{V}_{T_c-1}$ of size $B_c \times H$ each.
 4: Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_0, \cdots, \mathbf{O}_{T_r-1}$ of size $B_r \times H$ each, Divide $\mathbf{dO}$ into $T_r$
    blocks $\mathbf{dO}_0, \cdots, \mathbf{dO}_{T_r-1}$ of size $B_r \times H$ each, divide $l$ into $T_r$ blocks $l_0, \cdots, l_{T_r-1}$
    of size $B_r$ each, divide $m$ into $T_r$ blocks $m_0, \cdots, m_{T_r-1}$ of size $B_r$ each.
 5: Initialize $\mathbf{dQ} = (0)_{S\times H}$ in DRAM and divide it into $T_r$ blocks $\mathbf{dQ}_0, \cdots, \mathbf{dQ}_{T_r-1}$
    of size $B_r \times H$ each. Initialize $\mathbf{dK} = (0)_{S\times H}, \mathbf{dV} = (0)_{S\times H}$ in DRAM and divide
    $\mathbf{dK},\mathbf{dV}$ into $T_c$ blocks $\mathbf{dK}_0, \cdots, \mathbf{dK}_{T_c-1}$ and $\mathbf{dV}_0, \cdots, \mathbf{dV}_{T_c-1}$ of size $B_c \times H$
    each.
 6: for $0 \le j \le T_c - 1$ do
 7:     Load $\mathbf{K}_j$, $\mathbf{V}_j$ from DRAM to LDM.
 8:     Initialize $\tilde{\mathbf{dK}}_j = (0)_{B_c\times H}, \tilde{\mathbf{dV}}_j = (0)_{B_c\times H}$ on LDM.
 9:     for $0 \le I \le \lceil \frac{S}{B_r \times 64} \rceil - 1$ do
10:         In CPE $k$, calculate index $i = k + I \times 64$.
11:         In CPE $k$, Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dQ}_i, \mathbf{dO}_i, l_i, m_i$ from DRAM to LDM.
12:         In CPE $k$, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
13:         In CPE $k$, compute $\mathbf{S}_{ij}^{\text{masked}} = \mathbf{MASK}(\mathbf{S}_{ij})$.
14:         In CPE $k$, compute $\mathbf{P}_{ij} = \text{diag}(l_i)^{-1}\exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{Br\times Bc}$.
15:         In CPE $k$, compute dropout mask $\mathbf{Z}_{ij} \in \mathbb{R}^{Br\times Bc}$.
16:         In CPE $k$, compute $\mathbf{P}_{ij}^{\text{dropped}} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
17:         In CPE $k$, compute $\tilde{\mathbf{dV}}_j = \tilde{\mathbf{dV}}_j + (\mathbf{P}_{ij}^{\text{dropped}})^T \mathbf{dO}_i \in \mathbb{R}^{Bc\times H}$.
18:         In CPE $k$, compute $\mathbf{dP}_{ij}^{\text{dropped}} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{Br\times Bc}$.
19:         In CPE $k$, compute $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{dropped}} \circ \mathbf{Z}_{ij}$.
20:         In CPE $k$, compute $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{Br}$.
21:         In CPE $k$, compute $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{Br\times Bc}$.
22:         In CPE $k$, compute $\mathbf{dQ}_i = \mathbf{dQ}_i + \tau \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{Br\times H}$ and write $\mathbf{dQ}_i$ to
    DRAM.
23:         In CPE $k$, compute $\tilde{\mathbf{dK}}_j = \tilde{\mathbf{dK}}_j + \tau \mathbf{dS}_{ij}^T \mathbf{Q}_i \in \mathbb{R}^{Bc\times H}$.
24:     end for
25:     Reduce all the $\tilde{\mathbf{dK}}_j, \tilde{\mathbf{dV}}_j$ in 64 CPEs to $\tilde{\mathbf{dK}}_j, \tilde{\mathbf{dV}}_j$ in CPE 0.
26:     In CPE 0, write $\tilde{\mathbf{dK}}_j, \tilde{\mathbf{dV}}_j$ to $\mathbf{dK}_j, \mathbf{dV}_j$ in DRAM.
27: end for
28: Return $\mathbf{dQ}$, $\mathbf{dK}$, $\mathbf{dV}$.
```

3.3 Tiling strategy and mixed-precision training

Although LDM is much faster than DRAM, each CPE only has 256KB of LDM. Therefore, we need a tiling strategy to determine the optimal block sizes B_r and B_c, ensuring that the memory requirement of our algorithm does not exceed 256KB while maximizing the performance of SWattention. The detailed tiling strategy is as follows. First, there are many transposed GEMM operations in both forward pass and backward pass. To transpose matrices on LDM, we need another auxiliary matrix of the same shape. In Algorithm 2, matrix $\mathbf{K}_j$ of shape $[B_c, H]$ in line 11 needs to be transposed. In Algorithm 3, we need to transpose matrix $\mathbf{K}_j$ of shape $[B_c, H]$ in line 12, matrix $\mathbf{P}_{ij}^{\text{dropped}}$ of shape $[B_r, B_c]$ in line 17, matrix $\mathbf{V}$ of shape $[B_c, H]$ in line 18, and matrix $\mathbf{dS}_{ij}^T$ of shape $[B_r, B_c]$ in line 23. Therefore, we need to allocate a matrix of shape $[B_c, H]$ for forward pass and a matrix of shape $[B_c, \max(B_r, H)]$ for backward pass. Second, the backward pass requires both matrices **Q, K, V, O** and their gradients **dQ, dK, dV, dO**. Therefore, the block sizes B_r and B_c are smaller in the backward pass. Third, we also need to allocate LDM for many intermediate matrices such as $\mathbf{S}_{ij}$ in the forward pass. To save the limited LDM, some of the intermediate results can be stored using the same LDM space, and matrices for memory reuse are listed in Table 1.

Based on the above tiling strategy, the LDM required for forward pass and backward pass (using FP32) can be calculated as:

$$\text{LDM}_{\text{forward}} = \frac{(2B_rH + 3B_cH + B_rB_c) \times 4}{1024}\text{KB} \tag{2}$$

$$\text{LDM}_{\text{backward}} = \frac{(4B_rH + 4B_cH + 3B_rB_c + B_c \cdot \max(B_r, H)) \times 4}{1024}\text{KB} \tag{3}$$

Here, we omit other computation data with trivial memory cost, including l and m. The $3B_rB_c$ in $\text{LDM}_{\text{backward}}$ includes two matrices of shape $[B_r, B_c]$ in Table 1 and a matrix of shape $[B_r, B_c]$ for storing dropout mask $\mathbf{Z}_{ij}$. When we use FP16 for mixed-precision training, the required LDM is reduced by half. In general, larger block sizes B_r and B_c bring better performance. Although the total amount of computation remains unchanged, larger block sizes reduce the number of DMA calls and loops in algorithms, resulting in improved performance. Moreover, we should notice that

Table 1 Memory reuse on LDM

	Matrices	Shape
Forward pass	$\mathbf{S}_{ij}, \mathbf{S}_{ij}^{\text{masked}}, \tilde{\mathbf{P}}_{ij}, \tilde{\mathbf{P}}_{ij}^{\text{dropped}}$	$[B_r, B_c]$
	$\mathbf{Q}_i, \tilde{\mathbf{P}}_{ij}^{\text{dropped}}\mathbf{V}_j$	$[B_r, H]$
Backward pass	$\mathbf{S}_{ij}, \mathbf{S}_{ij}^{\text{masked}}, \mathbf{P}_{ij}, \mathbf{dS}_{ij}$	$[B_r, B_c]$
	$\mathbf{P}_{ij}^{\text{dropped}}, \mathbf{dP}_{ij}^{\text{dropped}}, \mathbf{dP}_{ij}$	$[B_r, B_c]$

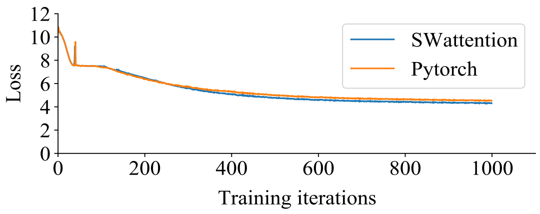


Fig. 6 The loss curves for training a 7B GPT-style model using SWattention and PyTorch attention

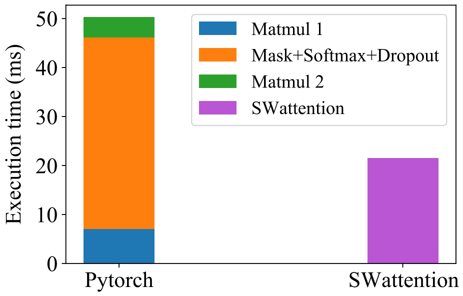


Fig. 7 The forward pass speedup of SWattention over PyTorch attention

SWattention parallelizes over the dimension S using 64 CPEs, which means that S should be a multiple of $64 \times B_r$.

Since the throughput of FP16 on SW26010pro is nearly four times that of FP32, it is crucial to use FP16 to implement SWattention. Based on the mixed-precision training strategy in [12], we use FP16 for GEMM in SWattention and use FP32 for **exp** operator. However, the precision conversion from FP16 to FP32 and from FP32 to FP16 introduces additional overhead, preventing FP16 from achieving the expected speedup.

4 Evaluation

4.1 Training validation and PyTorch attention

We validate the correctness of SWattention by training a GPT-style model with 7 billion parameters. The training is implemented with the PyTorch framework [25]. Figure 6 shows the loss curves of using SWattention and the standard attention that we refer to as PyTorch attention. We use 512 SW26010pro processors on the new Sunway Supercomputer and a training dataset with 1 billion tokens. The model is trained with tensor parallel [19] size 8, global batch size 512, FP32, sequence length $S = 2048$, and 1000 iterations. We can see that the training curves of SWattention and PyTorch attention are almost identical, which validates the correctness of SWattention.

In detail, the PyTorch attention is implemented with matmul, mask, softmax, and dropout operators. All of these operators have been parallelized across all CPEs and optimized using SIMD instructions on the SW26010pro processor. As illustrated in Fig. 7, the two matmul operators constitute only a small portion of the overall

PyTorch attention time. Due to the necessity of transposing the matrix **K** in the first matmul operator, the execution time is relatively longer. In this experiment, we use $B = 3$, $N = 8$, $S = 2048$, $H = 128$, and FP32. The performance of SWattention is 2.4 TFLOPS, while the performance of the second matmul operator is 6.3 TFLOPS, which reaches 45% of the peak performance of the SW26010pro processor. By fusing all these operators into one kernel, SWattention achieves speedup by reducing memory access between DRAM and LDM.

4.2 Speedup

We measure the speedup and the memory-saving function of SWattention in this section. We use the causal mask and vary the sequence length from 1k to 16k. The floating point operations (FLOPs) of attention can be calculated with the same formula in Megatron-LM [24]. When the shape of input **Q,K,V** is [B, N, S, H], the formula for the FLOPs of the forward pass is:

$$\text{FLOPs} = 4BNS^2H \tag{4}$$

For the FLOPs of backward pass, the FLOPs are doubled, and the FLOPs of recomputation are not included. Different from the skipping strategy of causal mask in FlashAttention-2 [26], we do not skip the blocks to be masked, so we do not reduce the FLOPs by half.

The performance of FP32 SWattention forward pass is shown in Fig. 8. We conducted experiments with head dimension $H = 64$ and $H = 128$. To evenly distribute all the attention heads across 6 CGs, we use $B = 3$ and $N = 8$. Generally, a larger head dimension H brings higher speed for both PyTorch attention and SWattention. Compared with PyTorch attention, the SWattention forward pass achieves an average speedup of 2.21x for $H = 128$ and 3.42x for $H = 64$. With data type FP32 and $H = 128$, we use block sizes $B_r = 32, B_c = 128$ for forward pass. Based on Eq. 2,

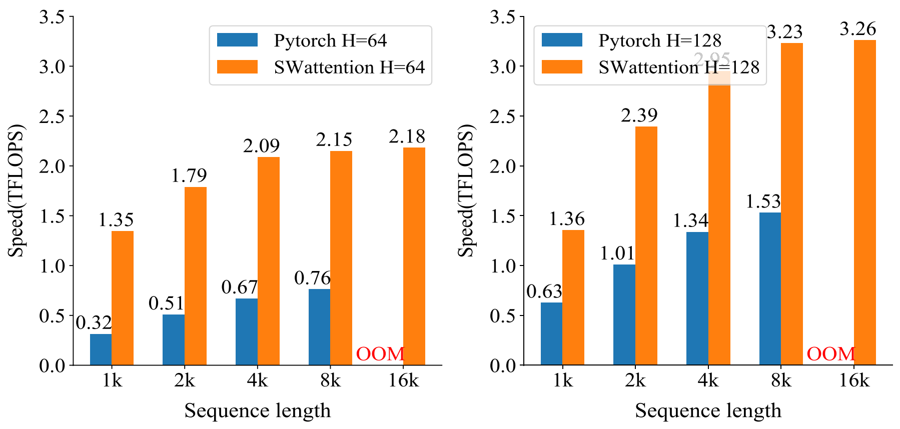


Fig. 8 Attention forward speed on SW26010pro processor with FP32

the LDM needed for this tiling strategy is 240KB. However, it is more efficient to set $B_r = 16$ and $B_c = 128$ with 216KB LDM for sequence length $S = 1024$, since SWattention requires that S is a multiple of $64 \times B_r$ to fully utilize all 64 CPEs. For the SWattention forward pass with $H = 64$, larger block sizes $B_r = 32$ and $B_c = 256$ can be employed (we continue to apply $B_r = 16$ for $S = 1024$), which brings higher speedup. When the sequence length reaches 16k, PyTorch attention runs out of memory (OOM), while SWattention continues to yield correct results. This demonstrates the memory-saving function of SWattention.

The performance of FP32 SWattention backward pass is shown in Fig. 9. SWattention backward pass achieves 1.33x speedup for $H = 128$ and 1.94x speedup for $H = 64$. Obviously, the speedup of the SWattention backward pass is not as significant as that observed in the forward pass, and this can be caused by the following reasons. First, the speed of PyTorch attention backward pass has been improved compared with the forward pass. Second, SWattention recomputes $\mathbf{QK}^T$ in line 12 of Algorithm 3, and this portion of FLOPs is not included. Third, according to Eq. 3, the block sizes are smaller in the backward pass. In practice, we use block sizes $B_r = 16, B_c = 64$ for $H = 128$ and $B_r = 32, B_c = 128$ for $H = 64$. We do not employ $B_r = 32, B_c = 64$ for $H = 128$ with 248KB LDM, as more than 8KB of LDM need to be reserved for other auxiliary arrays.

The overall speedup of FP32 SWattention is depicted in Fig. 10. Compared with the standard PyTorch attention, SWattention achieves 2.35x-2.44x speedup for $H = 64$ and 1.51x-1.65x speedup for $H = 128$. As the sequence length increases from 1k to 8k, the speed increases from 1.18 TFLOPS to 2.76 TFLOPS, and the speedup continues to grow. When the sequence length reaches 16k, the speed of SWattention exceeds 21% of the FP32 theoretical peak performance of the SW26010pro processor, which demonstrates the effectiveness of our implementation.

With the same experimental configurations, we evaluate the performance of FP16 SWattention using the mixed-precision training strategy. As shown in Fig. 11, the

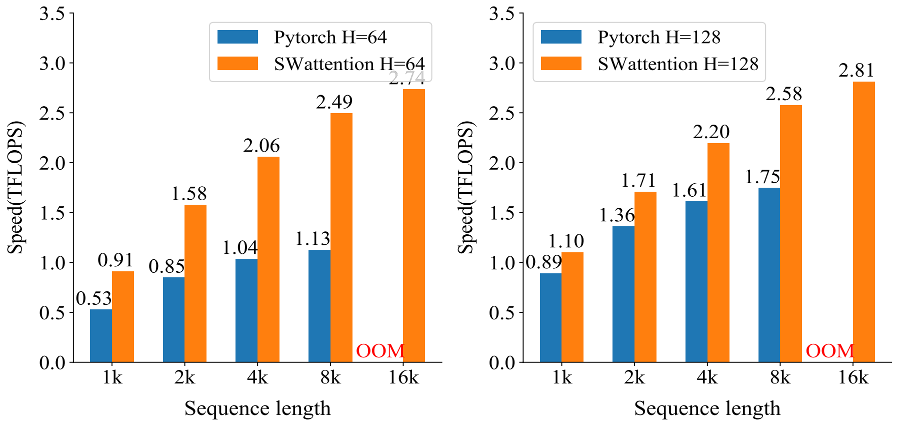


Fig. 9 Attention backward speed on SW26010pro processor with FP32

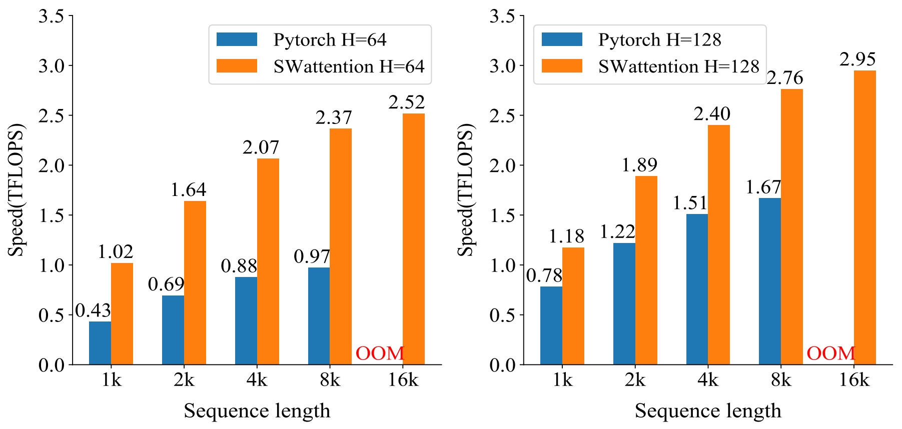


Fig. 10 Attention forward+backward speed on SW26010pro processor with FP32

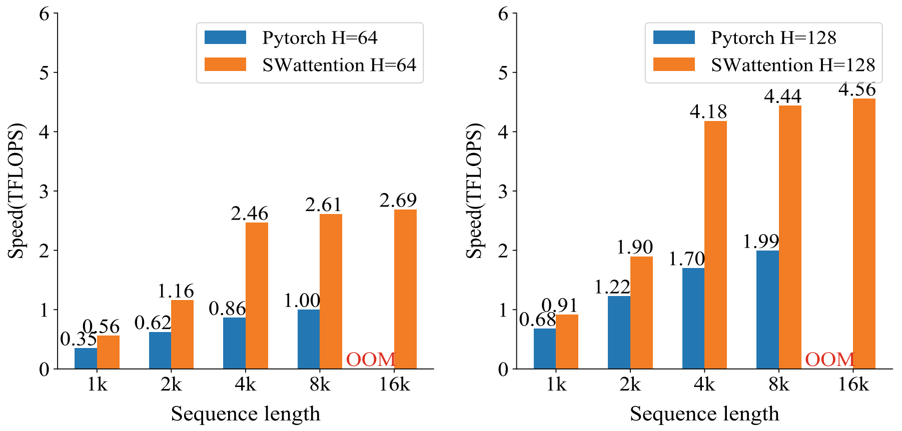


Fig. 11 Attention forward speed on SW26010pro processor with FP16

average FP16 SWattention forward pass speedup of $S = 4\text{k}$ and $S = 8\text{k}$ is 2.74x for $H = 64$ and 2.35x for $H = 128$. The performance of $S = 1\text{k}$ and $S = 2\text{k}$ in FP16 SWattention is primarily affected by the lack of relevant functions in the swBLAS library on the SW26010pro processor, which prevents the adoption of optimal block sizes. In practice, we use $B_r = 64, B_c = 128$ for all sequence lengths, resulting in inefficient utilization of CPEs for $S = 1\text{k}$ and $S = 2\text{k}$.

As shown in Fig. 12, the suboptimal performance of $S = 1\text{k}$ in the FP16 backward pass is also attributed to incapable of using the optimal block sizes B_r and B_c. For other sequence lengths, SWattention backward pass achieves 2.81x-3.22x speedup for $H = 64$ and 1.85x-2.32x speedup for $H = 128$. Besides, we observed that the FP16 backward pass is faster than the FP16 forward pass, which is different from FP32 SWattention. This may be caused by the larger block sizes used in FP16. For

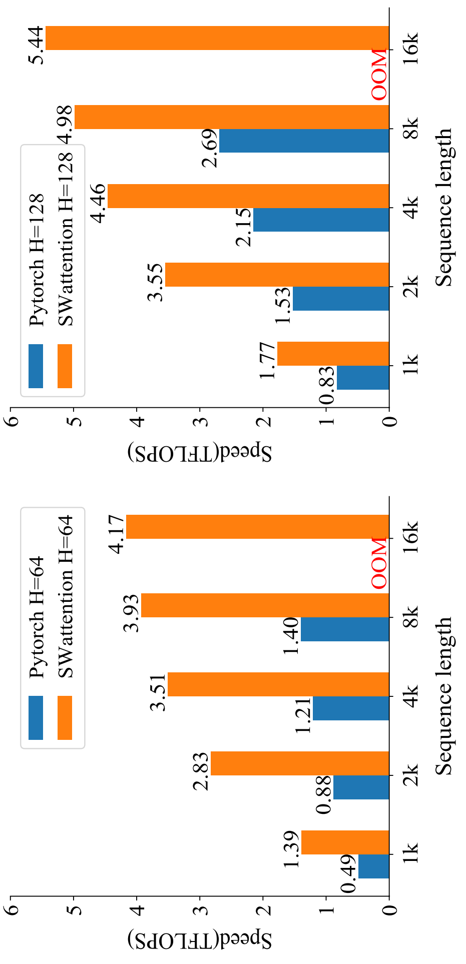


Fig. 12 Attention backward speed on SW26010pro processor with FP16

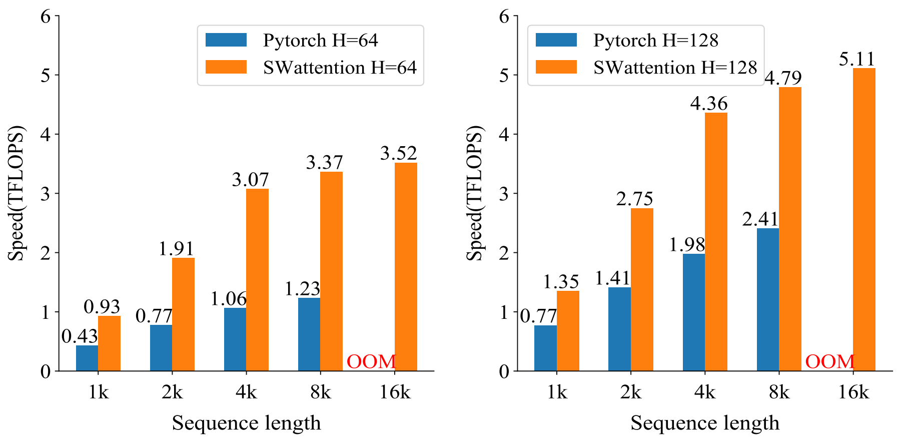
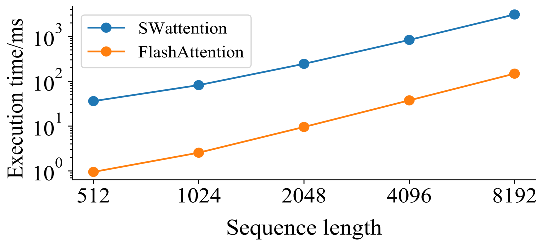


Fig. 13 Attention forward+backward speed on SW26010pro processor with FP16

Fig. 14 The execution time of SWattention and FlashAttention with the same workloads



example, we use $B_r = 64, B_c = 128$ for $H = 64$ and $B_r = 64, B_c = 64$ for $H = 128$. With larger block sizes, the speed of FP16 SWattention backward pass with $S = 16$k and $H = 128$ reaches 5.44 TFLOPS. Moreover, Fig. 13 shows the overall speedup of FP16 SWattention. Due to the cost of data type casting, the performance of mixed-precision training is degraded. Compared with FP32 training, mixed-precision SWattention achieves 1.73x-1.86x speedup with $H = 128$ and $S =$4k, 8k, 16k. Although the speedup achieved by mixed-precision training is far less than the ideal 4.0x, it still significantly accelerates the attention computation on the SW26010pro processor.

Finally, we compare the execution time of SWattention and FlashAttention with the same workloads. We measure the forward + backward runtimes of SWattention with $B = 16$, $N = 8$, and $H = 64$. The experiment is performed using FP32, and the sequence lengths vary from 512 to 8192. We then compare the results with the benchmarks evaluated on A100 GPU in the FlashAttention [8]. As shown in Fig. 14, both SWattention and FlashAttention exhibit a quadratic increase in execution time with the sequence length. Although SWattention is an order of magnitude slower than FlashAttention, the strength of the new Sunway Supercomputer is the world-top class amount of processors. With data and tensor parallelism technology, SWattention can be useful for training LLMs with longer sequence lengths and more parameters.

4.3 Memory access

We evaluate the memory bandwidth and the speedup of asynchronous memory access in this section. The experiment is performed with $B = 3$, $N = 4$, $H = 128$, and FP32. We use two sequence lengths 2048 and 4096. As depicted in the left part of Fig. 15, the DMA bandwidth is tested with loading matrix $\mathbf{O}$ from DRAM to LDM and storing matrix $\mathbf{O}$ from LDM to DRAM. Compared with the very limited memory bandwidth of the global load and store operations that can only reach 0.24 GB/s and 0.024 GB/s [12], the DMA load and store bandwidth are 211 GB/s and 122 GB/s, respectively. Therefore, DMA is well-suited for transferring contiguous memory blocks between DRAM and LDM, and the DMA bandwidth is close to the theoretical 307 GB/s bandwidth on the SW26010pro processor.

The speedup of asynchronous memory access is shown in the right part of Fig. 15. The asynchronous memory access is implemented with non-blocking DMA instructions, and we also record the runtimes of the implementation with blocking DMA instructions. Generally, the asynchronous memory access provides 1.1x speedup for both forward pass and backward pass. Additionally, we show the performance of the RMA reduction operation. The runtime of RMA reduction consists of both RMA communication between CPEs and the SIMD reduction operation on CPE. When the sequence length is 2048, the RMA reduction operation constitutes 24% of the total backward time. As the sequence length increases to 4096, this proportion decreases to 16%.

4.4 End-to-end performance

To measure the performance of SWattention for training real-world DNN models, we choose 2 GPT-style models for evaluation. We use hidden dimension 3072, head dimension $H = 128$, and number of heads $N = 24$. The 3.9B model is made up of 32 attention layers, while the 2.1B model contains 16 attention layers. We train these

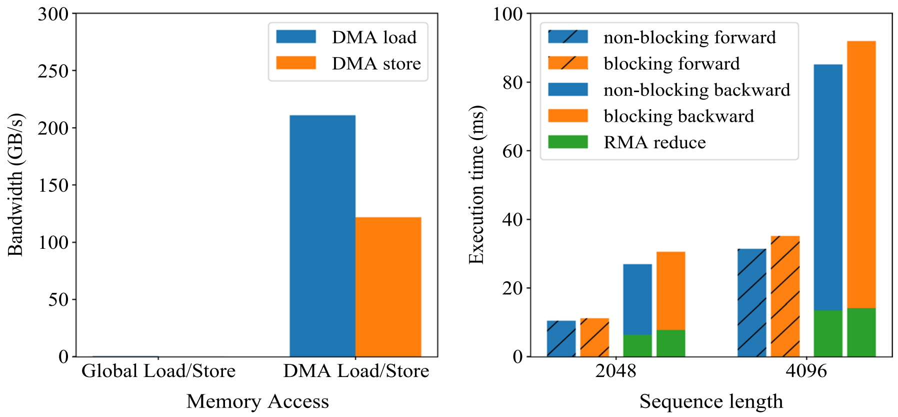


Fig. 15 Memory access optimization of SWattention

two models with 16 SW26010pro processors, global batch size 4, and tensor parallel size 4. Therefore, all processors run with batch size 1 and 6 attention heads.

As shown in Fig. 16, we train the 2.1B model and 3.9B model with/without FP32 SWattention. When the sequence length is relatively short (e.g., $S = 1$k, 2k), the attention computation constitutes a small fraction of the overall model execution time. Therefore, SWattention only achieves limited speedup. As the sequence length increases, the execution time of other parts of the model linearly increases with S, while the execution time of attention increases quadratically with S. Consequently, the proportion of attention computation in the overall model execution time becomes more significant. When the sequence length reaches 8k, the training of the 3.9B model without SWattention failed with OOM error, whereas the model with SWattention can still be trained successfully. Additionally, for the 2.1B model, we can observe that SWattention provides a 1.261x speedup for the whole model training.

To evaluate the scalability of SWattention, we compare the per-batch training time of four GPT-style models with SWattention and PyTorch attention. We scale the number of SW26010pro processors with the hidden dimension. In particular, we use $S = 4096$, $H = 128$, FP32, 16 attention layers, and global batch size 64 for all models. To ensure that the total amount of attention computation on each processor remains constant, the tensor parallel size doubles alongside the doubling of the number of processors. As shown in Table 2, SWattention achieves 1.15x speedup with 64 processors. In GPT-style models, the complexity of attention increases linearly with the hidden dimension, and the complexity of other modules such as MLP layers increases quadratically with the hidden dimension. When the number of processors increases to 512 and the model parameter size reaches 7.9 billion, SWattention continues to provide a 1.08x acceleration. Consequently, SWattention effectively accelerates LLM training with longer sequence lengths and more parameters.

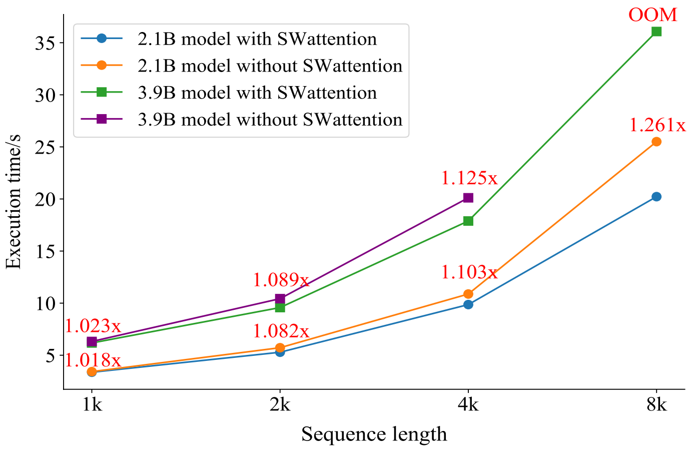


Fig. 16 Per-batch training time of 2 GPT-style models with 2.1 billion and 3.9 billion parameters with/without SWattention

Table 2 Scalability performance with different model parameters

Parameters	Heads	Hidden dimension	Tensor parallel	Number of processors	SWattention time (s)	PyTorch time (s)	Speedup
0.2B	6	768	1	64	8.93	10.27	1.15
0.6B	12	1536	2	128	8.78	9.87	1.13
2.1B	24	3072	4	256	10.21	11.21	1.10
7.9B	48	6144	8	512	14.28	15.35	1.08

The hidden dimension increases with the number of processors, and we record the per-batch training time

5 Conclusion

In this paper, we propose SWattention, an algorithm for computing exact attention with faster speed and reduced memory consumption on the new generation Sunway Supercomputer. Similar to the implementation of the FlashAttention, SWattention requires writing the attention algorithm with low-level master–slave style programming approach on the SW26010pro processor, which needs significant engineering effort. When training LLMs, the batch sizes B and the number of heads N are usually very small. Our two-level parallel strategy can effectively meet the computation requirements for task partition in the all-shared mode. Besides, our memory access optimizations and the utilization of FP16 further improve the performance of SWattention.

In general, the algorithm design of SWattention is based on both FlashAttention and FlashAttention-2 [26]. Following the approach in FlashAttention, for the double-loop iteration over the sequence length, we use B_c for the outer loop and B_r for the inner loop. One innovation in FlashAttention-2 is the reversal of the order of the two aforementioned loops. However, this optimization requires logarithmic operations, and currently, there is a lack of efficient SIMD optimization for logarithmic operations in PyTorch on the new generation Sunway Supercomputer. SWattention also draws inspiration from the method of parallelization along the S dimension in FlashAttention-2. SWattention parallelizes over the dimension S using 64 CPEs and reduces $\mathbf{dV}$ and $\mathbf{dK}$ in the backward pass using RMA.

Our future work mainly focuses on the following three aspects. First, when $B \times N$ is not a multiple of 6, our two-level parallel strategy could not fully utilize the performance of all 6 CGs. We are looking forward to calculating one attention head with more than one CG, and even with more than one processor. Second, due to the limitation of the swBLAS library, some optimal block sizes B_r and B_c could not be used. We hope to collaborate with engineers to optimize the swBLAS library and further optimize the SWattention with methods in the FlashAttention-2. Third, apart from the exact attention algorithms, we plan to implement other attention algorithms such as sparse attention [27–29] to develop the AI ecosystem on the new generation Sunway Supercomputer.

Acknowledgements We would like to thank other people in the Advanced Computer System Architecture (ACSA) Laboratory of University of Science and Technology of China for helping in this work. This work has been supported by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDB0500102) and the National Natural Science Foundation of China (Grant No. 62102389).

Author Contributions Our paper is applicable for submissions with multiple authors.

Funding This is not applicable.

Data availability The data generated during and/or analyzed during the current study are available from the corresponding author or the first author on reasonable request.

Declarations

Conflict of interest All authors declare that they have no conflict of interest.

References

1. Devlin J, Chang M-W, Lee K, Toutanova K (2018) Bert: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805
2. Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901
3. OpenAI R (2023) GPT-4 technical report. arXiv:2303.08774. View in Article 2
4. Touvron H, Lavril T, Izacard G, Martinet X, Lachaux M-A, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F et al (2023) Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971
5. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems 30
6. Tay Y, Dehghani M, Abnar S, Shen Y, Bahri D, Pham P, Rao J, Yang L, Ruder S, Metzler D (2020) Long range arena: a benchmark for efficient transformers. arXiv preprint arXiv:2011.04006
7. Ivanov A, Dryden N, Ben-Nun T, Li S, Hoefler T (2021) Data movement is all you need: A case study on optimizing transformers. Proc Mach Learn Syst 3:711–732
8. Dao T, Fu D, Ermon S, Rudra A, Ré C (2022) Flashattention: Fast and memory-efficient exact attention with io-awareness. Adv Neural Inf Process Syst 35:16344–16359
9. Zhao WX, Zhou K, Li J, Tang T, Wang X, Hou Y, Min Y, Zhang B, Zhang J, Dong Z et al (2023) A survey of large language models. arXiv preprint arXiv:2303.18223
10. Liu S, Gao J, Liu X, Huang Z, Zheng T (2021) Establishing high performance AI ecosystem on Sunway platform. CCF Trans High Perform Comput 3:224–241
11. Li M, Chen J, Xiao Q, Wang F, Jiang Q, Zhao X, Lin R, An H, Liang X, He L (2022) Bridging the gap between deep learning and frustrated quantum spin system for extreme-scale simulations on new generation of sunway supercomputer. IEEE Trans Parallel Distrib Syst 33(11):2846–2859
12. Ma Z, He J, Qiu J, Cao H, Wang Y, Sun Z, Zheng L, Wang H, Tang S, Zheng T et al (2022) BaGualu: targeting brain scale pretrained models with over 37 million cores. In: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp 192–204

13. Zhao Y, Zheng J, Fu H, Wu W, Gao J, Chen M, Zhang J, Zhang L, Dong R., Du Z et al (2023) SW-LCM: a scalable and weakly-supervised land cover mapping method on a new Sunway supercomputer. In: 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp 657–667
14. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76
15. Choquette J, Gandhi W, Giroux O, Stam N, Krashinsky R (2021) Nvidia a100 tensor core GPU: performance and innovation. IEEE Micro 41(2):29–35
16. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L 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), pp 578–594
17. Sivathanu M, Chugh T, Singapuram SS, Zhou L (2019) Astra: Exploiting predictability to optimize deep learning. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 909–923
18. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Gan L, Yang G, Qian D (2020) The deep learning compiler: a comprehensive survey. IEEE Trans Parallel Distrib Syst 32(3):708–727
19. Shoeybi M, Patwary M, Puri R, LeGresley P, Casper J, Catanzaro B (2019) Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053
20. Rasley J, Rajbhandari S, Ruwase O, He Y (2020) DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp 3505–3506
21. Gao J, Zheng F, Qi F, Ding Y, Li H, Lu H, He W, Wei H, Jin L, Liu X et al (2021) Sunway supercomputer architecture towards exascale computing: analysis and practice. SCIENCE CHINA Inf Sci 64(4):141101
22. Asad A, Kaur R, Mohammadi F (2022) A survey on memory subsystems for deep neural network accelerators. Future Internet 14(5):146
23. Shazeer N, Cheng Y, Parmar N, Tran D, Vaswani A, Koanantakool P, Hawkins P, Lee H, Hong M, Young C et al (2018) Mesh-tensorflow: Deep learning for supercomputers. In: Advances in neural information processing systems, vol 31
24. Narayanan D, Shoeybi M, Casper J, LeGresley P, Patwary M, Korthikanti V, Vainbrand D, Kashinkunti P, Bernauer J, Catanzaro B et al (2021) Efficient large-scale language model training on GPU clusters using megatron-LM. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–15
25. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al (2019) Pytorch: an imperative style, high-performance deep learning library. In: Advances in neural information processing systems, vol 32
26. Dao T (2023) Flashattention-2: Faster attention with better parallelism and work partitioning. arXiv preprint arXiv:2307.08691
27. Kitaev N, Kaiser Ł, Levskaya A (2020) Reformer: the efficient transformer. arXiv preprint arXiv:2001.04451
28. Roy A, Saffar M, Vaswani A, Grangier D (2021) Efficient content-based sparse attention with routing transformers. Trans Assoc Comput Linguist 9:53–68
29. Chen B, Dao T, Winsor E, Song Z, Rudra A, Ré C (2021) Scatterbrain: unifying sparse and low-rank attention. Adv Neural Inf Process Syst 34:17413–17426

Authors and Affiliations

Ruohan Wu[1] · Xianyu Zhu[1] · Junshi Chen[1] · Sha Liu[1] · Tianyu Zheng[2] · Xin Liu[3] · Hong An[1]

✉ Hong An
han@ustc.edu.cn

Ruohan Wu
ruohanwu@mail.ustc.edu.cn

Xianyu Zhu
zhuxy@mail.ustc.edu.cn

Junshi Chen
cjuns@ustc.edu.cn

Sha Liu
iliusha@mail.ustc.edu.cn

Tianyu Zheng
ashofcat@163.com

Xin Liu
yyylx@263.net

1 School of Computer Science and Technology, University of Science and Technology of China, Hefei, China

2 Zhejiang Lab, Hangzhou, China

3 National Supercomputing Center in Wuxi, Wuxi, China