

HAIMA: A Hybrid SRAM and DRAM Accelerator-in-Memory Architecture for Transformer

Yan Ding*, Chubo Liu*, Mingxing Duan*, Wanli Chang*[†], Keqin Li*[‡], Kenli Li*

^{*}College of Information Science and Engineering, Hunan University, China

[†]Department of Computer Science, University of York, UK

[‡]Department of Computer Science, State University of New York, New Paltz, USA

{ding, liuchubo, duanmingxing, likl}@hnu.edu.cn, wanli.chang@york.ac.uk, and lik@newpaltz.edu

Abstract—Through the attention mechanism, Transformer-based large-scale deep neural networks (LSDNNs) have demonstrated remarkable achievements in artificial intelligence applications such as natural language processing and computer vision. The matrix-matrix multiplication operation (MMMO) in Transformer makes data movement dominate the inference overhead over computation. A solution for efficient data movement during Transformer inference is to embed arithmetic logic units (ALUs) into the memory array, hence an accelerator-in-memory architecture (AIMA). Existing work along this direction has not considered the heterogeneity of parallelism and resource requirements among Transformer layers. This increases the inference latency and lowers the resource utilization, which is critical for the embedded systems domain. To this end, we propose HAIMA, a hybrid AIMA and the parallel dataflow for Transformer, which exploit the cooperation between SRAM and DRAM to accelerate different MMOs. Compared to the state-of-the-art Newton and TransPIM, our proposed hardware-software co-design achieves 1.4x-1.5x speedup, and solves the problem of resource under-utilization when DRAM-based AIMA performs the light-weight MMOs.

I. INTRODUCTION

In recent years, the large-scale deep neural networks (LSDNNs) based on Transformer have made remarkable achievements in artificial intelligence applications such as natural language processing [1] and computer vision [2]. However, the continuous improvement of LSDNN performance implies a continuous increase in the number of network layers and the scale of parameters. LSDNN not only requires more computing and memory resources, but also creates a new challenge, that the matrix-matrix multiplication operation (MMMO) in Transformer makes data movement dominate the inference overhead over computation. To address the issue, academia and industry have paid great attention to embedding arithmetic logic units (ALUs) into the memory array to reduce the data movement. This technique is called accelerator-in-memory architecture (AIMA).

Static random access memory (SRAM) and dynamic random access memory (DRAM) are important representatives of modern very large scale integration (VLSI) memory systems, and are 2 main research substrates for AIMAs. The fabrication process of SRAM is compatible with the standard CMOS

process and is embedded on the chip as a cache, so SRAM-based AIMA can realize the computing-in-memory (CIM) at bit-cell level [3]. For example, Neural Cache [4] enabled bit-level addition and multiplication without significant hardware changes. Although SRAM does not require to refresh its circuit to retain the stored data, a typical SRAM 1-bit memory cell requires 6 or 8 transistors, so SRAM-based AIMA is characterized by lower power consumption and memory density. Compared to SRAM, a 1-bit memory cell of DRAM consists of a transistor and a capacitor, and its circuit needs to be refreshed periodically to retain the stored data. Thus, DRAM-based AIMA is characterized by higher power consumption and memory density. Meanwhile, since the fabrication process of DRAM is not compatible with the standard CMOS process, DRAM-based AIMA can often realize the computing-near-memory (CNM) at bank level [3]. For example, Newton [5] designed the multiply-accumulate (MAC) unit and buffer in DRAM banks to reduce data movement while maintaining memory capacity. Combining the characteristics of SRAM and DRAM, SRAM-based AIMA focuses more on accelerating binary neural networks [6] or other small-scale neural networks [7], while DRAM-based AIMA is more frequently used for accelerating LSDNN [8].

But the above work ignores the following 3 facts. First, for Transformer, its layers exhibit significant heterogeneity in terms of parallelism as well as the resource requirements of computing and memory [9]. There are H independent heads in a multi-head attention layer (MHAL), so MMOs of each head are processed in parallel. However, the feed forward network layer (FFNL) does not have that kind of parallelism. Hence, it is necessary to propose different parallel computing mechanisms according to the characteristics of different layers. Second, in addition to the area and power, we should also consider the resource utilization of entire system and the gain of AIMA. If a light-weight MMO is still performed on the AIMA with sufficient resources such as 64GB DRAM [8], it will significantly reduce the resource utilization and acceleration gain. Finally, we must carefully design ALUs in the memory array to ensure the memory capacity, so AIMA cannot have the same computing power as the host such

as CPU and GPU. The data movement between the off-chip memory and the host cannot be completely eliminated in a system. Thus, there is the possibility and necessity of cooperation between SRAM- and DRAM-based AIMA.

Based on the above discussions, this paper aims to improve the parallel efficiency of the computation of MMMOs, and to reduce Transformer inference latency from an end-to-end perspective. To this end, we propose a **hybrid accelerator-in-memory architecture (HAIMA)** in which SRAM and DRAM cooperate to complete the computation of MMMOs in Transformer. The contributions of the paper are as follows:

- A hybrid AIMA named HAIMA is proposed, which exploits the hardware characteristics of SRAM and DRAM. DRAM-based AIMA performs large-scale MMMOs. SRAM-based AIMA acts as a filter between the DRAM-based AIMA and the host, and handles light-weight MMMOs during data movement.
- A parallel dataflow for Transformer inference is proposed, which takes into account the different parallelism and resource requirements among different layers. Meanwhile, the dataflow considers the computation process of Transformer in the system.
- The CIM unit of SRAM and the CNM unit of DRAM are developed to support the proposed architecture and dataflow for efficient computation of MMMOs.

Compared to the state-of-the-art Newton and TransPIM, our proposed hardware-software co-design achieves 1.4x-1.5x speedup. The efficient collaboration between the host, SRAM-based AIMA, and DRAM-based AIMA not only enables AIMA to be better integrated with the traditional von Neumann architecture, but also reduces inference latency while improving the resource utilization of the whole system. For convenience, we use SRAM-CIM and DRAM-CNM to represent SRAM-based AIMA and DRAM-based AIMA, respectively.

II. BACKGROUND AND MOTIVATION

A. Transformer

All Transformer-based LSDNNs are built on a stack of encoder blocks and decoder blocks. Figure 1(a) shows the block structure of the encoder and decoder. Figures 1(b) and 1(c) show the runtime breakdown of 2 block types in BERT [1] and BART [10]. As shown in the figures, the MHAL and FFNL are 2 critical modules of Transformer, and dominate the computation latency. The main reason for this phenomenon is that there are a lot of MMMOs (i.e., MAC operations) in the 2 modules. As shown in Figure 1(a), for a sequence with N tokens and M hidden dimensions, 3 fully connected layers (FCLs) with 3 weight matrices (i.e., $\mathcal{W}^Q \in \mathbb{R}^{M \times M}$, $\mathcal{W}^K \in \mathbb{R}^{M \times M}$, $\mathcal{W}^V \in \mathbb{R}^{M \times M}$) obtain a positional embedded matrix ($\mathcal{X} \in \mathbb{R}^{N \times M}$) and generate Query ($\mathcal{Q} \in \mathbb{R}^{N \times M}$), Key ($\mathcal{K} \in \mathbb{R}^{N \times M}$), and Value ($\mathcal{V} \in \mathbb{R}^{N \times M}$) matrices respectively. Assuming that there are H heads in Transformer, then $\mathcal{Q}, \mathcal{K}, \mathcal{V}$ are divided into H parts by column, i.e., the weight sub-matrices of each head are generated. We use $\mathcal{Q}_h, \mathcal{K}_h, \mathcal{V}_h$, where $h \in [0, H-1]$, to represent the weight sub-matrix of $\mathcal{Q}, \mathcal{K}, \mathcal{V}$ in each head, respectively. The scaled dot product

attention layer (SDPAL) is responsible for calculating the attention scores between tokens. SDPAL mainly consists of 3 MMMOs, i.e., $\mathcal{Y}_h = \mathcal{Q}_h \mathcal{K}_h^\top$, $\mathcal{S}_h = \text{Softmax}(\mathcal{Q}_h \mathcal{K}_h^\top / \sqrt{M})$, and $\mathcal{A}_h = \mathcal{S}_h \mathcal{V}_h$. Then, an attention score weight matrix of all heads, i.e., $\mathcal{A} = \text{Concat}(\mathcal{A}_0, \dots, \mathcal{A}_{H-1})$, is fed into the last FCL of MHAL with weight matrix $\mathcal{W}^L \in \mathbb{R}^{M \times M}$ to generate the output of MHAL, i.e., $\mathcal{Z} = \mathcal{A} \mathcal{W}^L + B^L$, where $B^L \in \mathbb{R}^{1 \times M}$ is bias vector. Finally, the output matrix $\mathcal{O} \in \mathbb{R}^{N \times M}$ of an encoder block will be obtained after $\mathcal{Z}$ is fed into the following 3 layers, i.e., Add & Norm, FFNL, and Add & Norm. $\mathcal{O}$ will be the input for the next block.

Although a decoder has more computations than an encoder, its core operation is no different from the encoder. A decoder block is also composed of MHAL, FFNL, and Add & Norm layer. It should be noted that, in a decoder block, the output of its previous encoder needs to be used as the input to obtain the key and value matrices of its second MHAL, so this joint structure is called encoder-decoder block.

B. Motivation

The data movement takes about 60% of the inference latency [8]. In addition, the operation type of single-instruction-multiple-data (SIMD) like MMMO is exactly the computation type that AIMA can effectively handle. Therefore, co-designing AIMA and parallel dataflow of MMMO is a key method to speed up Transformer inference. As mentioned above, all MMMOs in Transformer exhibit significant heterogeneity in terms of parallelism as well as the resource requirements of computing and memory. Next, we characterize 4 MMMO types in Transformer, and describe their corresponding parallel strategies. The 4 MMMO types include (i) the MMMO to obtain $\mathcal{Q}, \mathcal{K}$, and $\mathcal{V}$ (marked as MMMO-Q/K/V respectively); (ii) the MMMO to obtain $\mathcal{Y}$ and $\mathcal{A}$ (marked as MMMO-Y/A respectively); (iii) the MMMO to obtain $\mathcal{Z}$ (marked as MMMO-Z); (iv) the MMMO in FFNL (marked as MMMO-FFNL).

1) *MMMO-Q/K/V*: The parallel strategy adopted for MMMO-Q/K/V is matrix partitioning multiplication (MPM). Specifically, $\mathcal{W}^Q, \mathcal{W}^K, \mathcal{W}^V$ are equally split into H sub-matrices by column, respectively. The linearly projected weight sub-matrices for each head can be denoted as $\mathcal{W}_h^Q \in \mathbb{R}^{M \times (M/H)}$, $\mathcal{W}_h^K \in \mathbb{R}^{M \times (M/H)}$, and $\mathcal{W}_h^V \in \mathbb{R}^{M \times (M/H)}$, respectively. Hence, $\mathcal{Q}_h, \mathcal{K}_h$, and $\mathcal{V}_h$ can be obtained in parallel through $\mathcal{Q}_h = \mathcal{X} \mathcal{W}_h^Q + B_h^Q$, $\mathcal{K}_h = \mathcal{X} \mathcal{W}_h^K + B_h^K$, and $\mathcal{V}_h = \mathcal{X} \mathcal{W}_h^V + B_h^V$, where $B_h^Q \in \mathbb{R}^{1 \times (M/H)}$, $B_h^K \in \mathbb{R}^{1 \times (M/H)}$, and $B_h^V \in \mathbb{R}^{1 \times (M/H)}$ are bias vectors.

2) *MMMO-Y/A*: The parallel strategy adopted by most of current work for MMMO-Y/A is just to independently compute $\mathcal{Y}_h$ and $\mathcal{A}_h$ in H heads, and does not further explore the parallelism of MMMOs within a head. However, we observe that since AIMA can not handle relatively complex operations like $\text{Softmax}(\cdot)$ in a cost-effective way. Therefore, in order to get $\mathcal{A}_h$, $\mathcal{Y}_h$ will first be sent to the host to complete the computation of $\mathcal{S}_h = \text{Softmax}(\mathcal{Q}_h \mathcal{K}_h^\top)$, and then the result matrix $\mathcal{S}_h$ will be sent back to AIMA to complete the computation of $\mathcal{A}_h = \mathcal{S}_h \mathcal{V}_h$. However, $\mathcal{V}_h$ can only be in

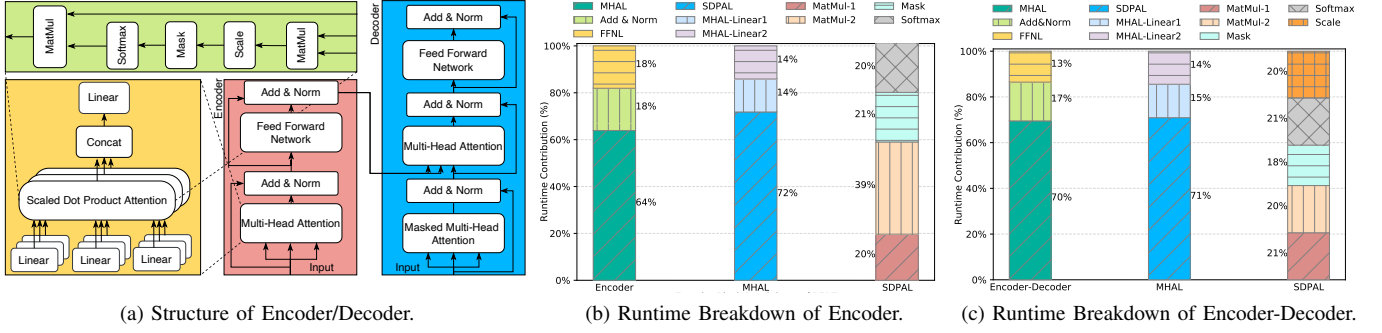


Fig. 1: Preliminary of Transformer.

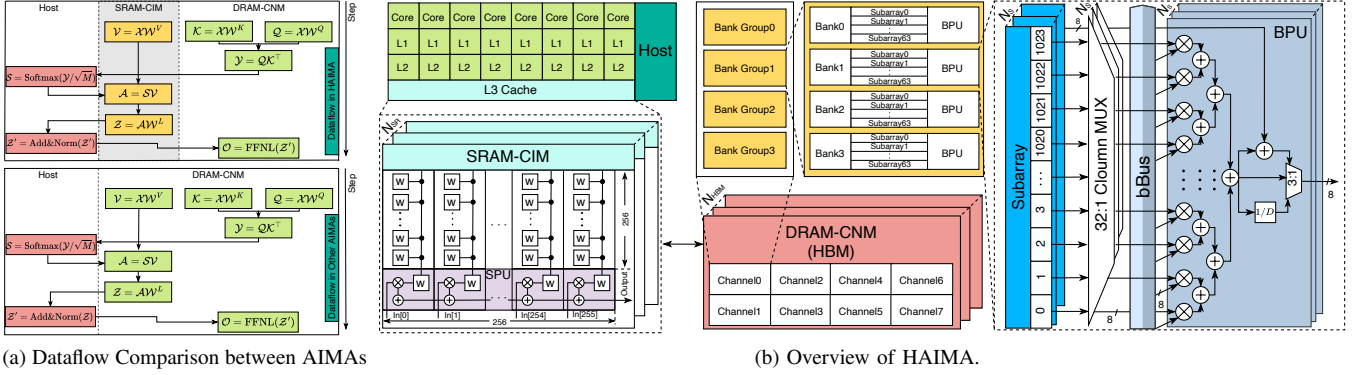


Fig. 2: Dataflow and Architecture of HAIMA.

a waiting state during the computation of S_h and the data communication between the host and the off-chip memory, thereby increasing the unnecessary waiting latency.

3) *MMMO-Z*: Compared to *MMMO-Q/K/V*, although *MMMO-Z* does not have the parallelism between multiple heads, its computation scale is only one third of theirs. In other words, *MMMO-Z* requires less computing and memory resources. Thus, if *MMMO-Z* is still performed on the off-chip memory with sufficient resources, it will significantly reduce resource utilization and cause unnecessary energy consumption of the system. The computation of *MMMO-Z* can be accelerated by the MPM parallel strategy.

4) *MMMO-FFNL*: *MMMO-FFNL* has no parallelism among multiple heads, but its computational load scale is large, so it requires more computing and memory resources than other *MMMOs*. The existing work generally adopts the MPM parallel strategy and DRAM-based AIMA to speed up the computation of *MMMO-FFNL*.

Unlike existing work, this paper exploits both the inter-head parallelism and the intra-head parallelism in different *MMMOs* to accelerate Transformer inference. Moreover, the resource requirements of different *MMMOs* are considered in the proposed parallel dataflow. Specifically, SRAM-CIM handles small-scale *MMMOs*, while DRAM-CNM handles large-scale *MMMOs*. Figure 2(a) shows a dataflow comparison between HAIMA and other DRAM-based AIMAs. As shown in the top of Figure 2(a), the HAIMA dataflow utilizes the limited computing power of SRAM-CIM to complete the computation of *MMMO-V/A/Z*. Thus, the attention scores $\mathcal{A}$ can be obtained during the movement of $\mathcal{S}$. This mechanism

can not only realize the overlap of communication and computation to reduce the unnecessary waiting latency of $\mathcal{V}$, but also solve the problems of low resource utilization and high energy consumption when DRAM-CNM performs the light-weight *MMMOs*. Next, we detail the dataflow of HAIMA.

III. HAIMA DATAFLOW

The underlying architecture of dataflow is based on HAIMA as shown in Figure 2(b). To take full advantage of the parallel characteristics between banks on DRAM-CNM, we adopt a joint strategy of data parallelism and token-based data mapping to improve the locality of data within a bank [5], [8]. Specifically, $\mathcal{W}^Q$ and $\mathcal{W}^K$ are stored in each bank of DRAM-CNM. For a DRAM-CNM with N_B banks, each bank can receive N/N_B tokens. In this way, for N_B groups of tokens, their Q and K can be computed independently on N_B banks. Without loss of generality, in the following content, we assume that each bank receives one token to facilitate the introduction of dataflow. In addition, we use the idea of systolic array to construct an efficient light-weight *MMMO* architecture on SRAM-CIM. Specifically, $\mathcal{W}^V$ is evenly divided into H sub-matrices by column and stored in H SRAM-CIMs respectively. In this way, each SRAM-CIM can independently performed $\mathcal{V}_h = \mathcal{X}\mathcal{W}_h^V$. Moreover, $\mathcal{A}_h$ can also be obtained in the SRAM-CIM when $\mathcal{S}_h$ is transferred back to DRAM-CNM from the host. Based on the above, we know that DRAM-CNM performs *MMMO-Q/K/Y/FFNL*, while SRAM-CIM performs *MMMO-V/A/Z*. Next, we detail the parallel dataflow of SRAM-CIM and DRAM-CNM for different *MMMOs*, respectively.

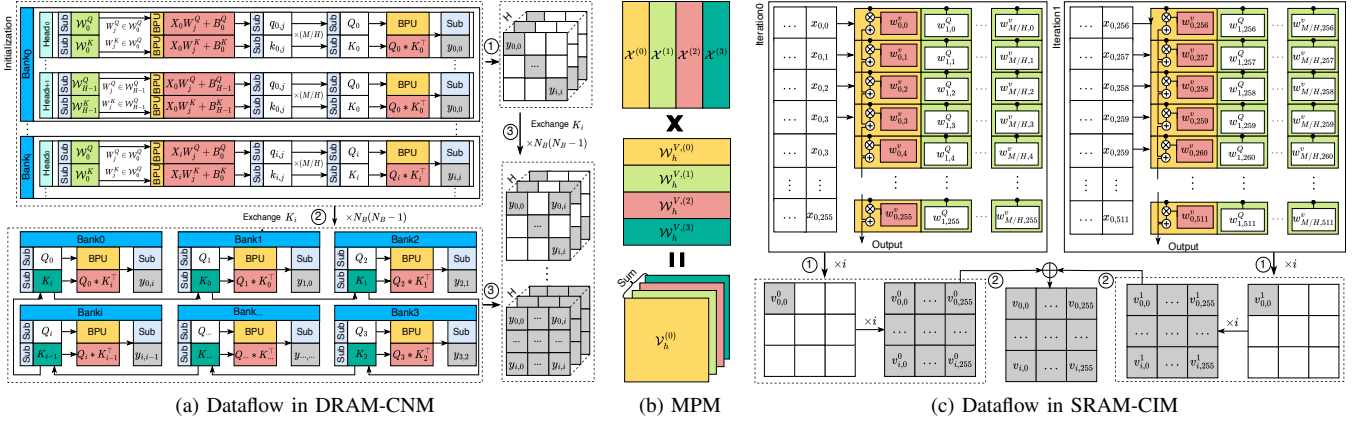


Fig. 3: Dataflow in DRAM-CNM and SRAM-CIM

A. Dataflow in DRAM-CNM

1) *MMMO-Q/K/Y*: As shown in Figure 3(a), the weight matrices $\mathcal{W}^Q$ and $\mathcal{W}^K$ are loaded in each bank before the computation of MMMO-Q/K. Moreover, $\mathcal{W}^Q$ and $\mathcal{W}^K$ are evenly divided into H sub-matrices by column. The weight sub-matrices for each head (i.e., $\mathcal{W}_h^Q$ and $\mathcal{W}_h^K$) are stored separately in a subarray of the bank. Because accessing a subarray once only retrieves one row of data in the subarray, thus, for a head, vectors $Q_i \in \mathbb{R}^{1 \times M/H}$ and $K_i \in \mathbb{R}^{1 \times M/H}$ related to X_i ($i \in [0, N-1]$) can be obtained through M/H row accesses, respectively. After getting Q_i and K_i , the computation of $y_{i,i} = Q_i K_i^\top$ can be performed immediately. Benefit from the parallel processing capability between banks, as shown in the upper right part of Figure 3(a), the attention scores between Q_i and K_i in all heads can be obtained at the same time. Furthermore, to obtain the scores between all tokens, banks exchange their stored K_i in a ring fashion. At each exchange iteration, Q_i originally stored in the bank is multiplied by the exchanged $K_{i'}$ ($i' \neq i$), thus resulting in an attention score between X_i and $X_{i'}$. As shown in the bottom of Figure 3(a), after $N_B(N_B - 1)$ exchange iterations, we can finally get the attention score matrices between N tokens in all heads, i.e., $\mathcal{Y}$. Next, $\mathcal{Y}$ is sent to the host and the computation of $\mathcal{S} = \text{Softmax}(\mathcal{Y})$ will be performed by the host.

2) *MMMO-FFNL*: As shown in Figure 3(b), matrix multiplication enables parallel computing in the form of MPM. After obtaining the matrix $\mathcal{Z}' \in \mathbb{R}^{N \times M}$ sent by the host/SRAM-CIM (Section III.B), DRAM-CNM completes the computation of MMMO-FFNL in the way of MPM. $\mathcal{Z}'$ and the first weight matrix of FFNL are evenly divided into N_B sub-matrices by row and column respectively, and stored in N_B banks. Thus, the computation between the sub-matrices are completed in parallel, and N_B intermediate result matrices are generated. The result weight matrix of FFNL's first layer can be obtained by summing all the intermediate result matrices. Repeat the above process twice to get $\mathcal{O}$.

B. Dataflow in SRAM-CIM

1) *MMMO-V*: The weight matrix $\mathcal{W}^V$ is loaded in SRAM-CIM before the computation of MMMO-V. It should be noted that due to the limited memory capacity of SRAM (256×256

8-bit memory cells in this paper), parallel efficiency and resource utilization must be taken into account when $\mathcal{W}_h^V$ is stored in SRAM-CIM. Thus, the computation strategy is MPM. The storage format of $\mathcal{W}_h^V$ in SRAM-CIM is shown in the middle of Figure 3(b). Figure 3(c) shows an example of MMVO-V in SRAM-CIM when $M = 512$. We first divide $\mathcal{W}^V$ into H sub-matrices by column, and then further divide a sub-matrix $\mathcal{W}_h^V$ into $M/256$ sub-matrices by row, i.e., $\mathcal{W}_h^{V,(0)}$ and $\mathcal{W}_h^{V,(1)}$. Input matrix $\mathcal{X}$ is divided into $M/256$ sub-matrices by column, i.e., $\mathcal{X}_h^{(0)}$ and $\mathcal{X}_h^{(1)}$. Then, both inputs and partial-sums pass through a 2-dimension SRAM processing unit (SPU) while pipelined parallel MMVOs are performed. Although the latency for one SPU to complete $\mathcal{V}_h = \mathcal{X} \mathcal{W}_h^V$ is $M/256$ times that for $M/256$ SPUs to complete $\mathcal{V}_h = \mathcal{X} \mathcal{W}_h^V$, this does not affect the performance of HAIMA to complete $\mathcal{A} = \mathcal{S} \mathcal{V}$. As shown in Figures 1(a) and 2(a), this is because that the computation of MMVO-V has been completed before $\mathcal{A}$ reaches SRAM-CIM. Moreover, due to area and power constraints, we does not add ALUs that supports matrix addition in SRAM-CIM. Therefore, the intermediate result matrices $\mathcal{V}_h^{(0)}, \mathcal{V}_h^{(1)}$ will be transferred to the host for summation, and then the final result matrix $\mathcal{V}_h$ will be written to SRAM-CIM. The storage format of $\mathcal{V}_h$ is the same as that of $\mathcal{W}_h^V$ to facilitate subsequent computation of MMMO-A.

2) *MMMO-A/Z*: After the computation of $\mathcal{S}_h$ is completed, the host will send $\mathcal{S}_h$ to the SRAM-CIM that retained $\mathcal{V}_h$. Similarly, $\mathcal{S}_h$ is divided into $M/256$ sub-matrices by column, and then each sub-matrix is sent to the SPU row by row to complete the computation of MMMO-A. During the computation of $\mathcal{A}_h$, $\mathcal{W}^L$ is loaded in SRAM-CIMs. Note that in order to reduce the delay caused by SRAM-CIM switching between CIM mode and memory mode, HAIMA always ensures that if there is an SRAM-CIM in CIM state, there must be an SRAM-CIM in memory state. We first divide $\mathcal{W}^L$ into H sub-matrices by column, and then divide a sub-matrix $\mathcal{W}_h^L$ into $M/256$ sub-matrices by row. Thus, MMMO-Z can be performed in the same way as MMMO-V/A. The host takes $\mathcal{Z}$ as input to Add & Norm layer to generate a result matrix $\mathcal{Z}'$. Then, $\mathcal{Z}'$ is sent back to DRAM-CNM to complete the computation of

TABLE I: Configuration Parameters of HAIMA

Memory	Item	Parameter	Value
DRAM	Configuration	Number of HBM ranks	1
		Number of bank groups per rank	4
		Number of banks per bank group	4
		Number of subarrays per bank	64
		Number of rows per bank	32,768
		Number of rows per subarray	512
		Number of bit per row	$1,024 \times 8$
SRAM	Timing (ns)	$t_{RAS} = 29$, $t_{RP} = 14$, $t_{RCD} = 16$, $t_{CL} = 16$, $t_{CCD} = 2$, $t_{WR} = 16$, $t_{RC} = 45$, $t_{RRD} = 2$	
	Configuration	Number of SRAM-CIMs	32
		Number of rows per SRAM	256
		Number of bit per row	256×8
	Timing (ns)	The write recovery time $t_{SWR} = 0.163$	

MMMO-FFNL (Section III.A).

IV. HAIMA HARDWARE

A. DRAM-CNM Design

Figure 2(b) shows the overview of the proposed architecture HAIMA. We design the bank processing unit (BPU) to support the computation of MMOs. The BPU is composed of a multiplier-adder tree, a full-adder, a divider, and a 3-to-1 multiplexer (MUX). HAIMA supports simultaneous activation of N_S subarrays in each bank to increase the parallelism of computations. Hence, HAIMA can employ N_S BPUs per bank to match the bank's access bandwidth. We reserve a computation row (CR) connected to a BPU in each subarray, and the data that need to be performed must first be copied to the row. In this way, the stored data in CR can be performed when it is accessed. To avoid expensive area and power overhead, we add a 32-to-1 MUX between the BPU and the CR. That is, a multiplier-adder tree of BPU consists of 32 multipliers and 31 full-adders. The BPU is designed to support the dot product between 2 vectors, and each vector consists of 32 8-bit elements. It should be noted that the 8-bit precision is sufficient for Transformer inference. The BPU requires 32 iterations to complete the MAC operations between 2 1KB data. In each iteration, 32 8-bit elements are accessed from the CR, and these accessed data are then combined with the 32 8-bit elements in the input bus to complete the computations.

B. SRAM-CIM Design

As shown in Figure 2(b), each SPU has 256 computing units, and each computing unit is composed of 8 6-transistor SRAM cells, a multiplier, and a full-adder. To avoid noise interference, this paper adopts digital circuit technology to realize the CIM and CNM [7]. Similarly, to avoid expensive area and power overhead, the SPU is designed to support the dot product between 2 vectors, and each vector consists of 256 8-bit elements. Moreover, the stored data in SRAM-CIM that need to be performed are first copied to the computation array of SPU. Then, both inputs and partial-sums pass through the SPU while pipelined parallel MMOs are completed.

V. EVALUATION

The off-chip memory, i.e., DRAM, used by HAIMA is standard HBM2 architecture. The simulation parameters are obtained from the previous work [4], [11], [12]. The parameters that characterize the hardware are shown in Table I. In addition, the host-HBM bandwidth is set to 256GB/s,

TABLE II: Overhead Comparison Between AIMAs

Unit	Area/Unit (mm ²)	Power/Unit (mW)
CNM Unit of TransPIM	0.016	800.6
CNM Unit of Newton	0.044	3,110.7
BPU of HAIMA	0.014	148.0
32-to-1 MUX of HAIMA	0.033	2,990
SPU of HAIMA	0.125	1,554.3

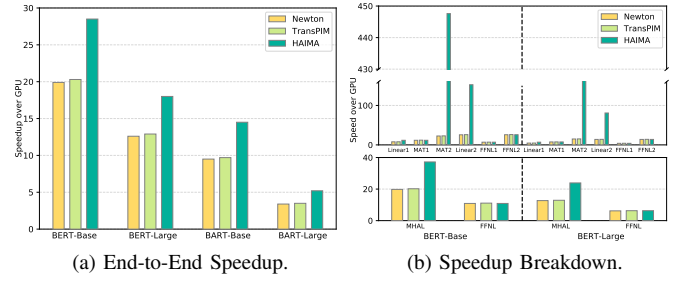


Fig. 4: Performance Comparison Between AIMAs.

the host-SRAM bandwidth is set to 1TB/s. The frequency of DRAM-CNM and SRAM-CIM are set to 500 MHz. The area of DRAM-CNM is evaluated by CACTI 7.0. Moreover, we implement HAIMA using Verilog HDL, synthesize the proposed design in Synopsys Design Compiler, and use 45nm library to measure its area and power. The implementation results of HAIMA are shown in Table II.

We use the following DRAM-based AIMAs or hardware as baselines: (i) Newton [5], (ii) TransPIM [8], and (iii) Nvidia Tesla V100. To make the comparison between different AIMAs more rigorous, and to bring the simulation closer to reality, the baseline TransPIM is not equipped with CIM capability [13]. Meanwhile, since the number of processing units in a bank can be increased according to certain rules, in order to compare the performance of different architectures better and more fairly, the experiments set only one processing unit in each bank for all AIMAs. In addition, 2 representative Transformer-based LSDNNs, i.e., BERT [1] and BART [10], are used to evaluate the different AIMAs. BERT consists of multiple encoder blocks, while BART consists of multiple encoder-decoder blocks. The 2 models are used for question-answering task (on the dataset of TriviaQA [14]) and summary generation task (on the dataset of Pubmed [15]), respectively.

A. Performance Comparison

Figure 4 shows the speedup of different AIMAs over the Nvidia Tesla V100. As shown in Figure 4(a), compared to other DRAM-based AIMAs, HAIMA achieves 1.4x-1.5x speedup. As shown in Figures 2(a) and 4(b), since HAIMA takes into account the heterogeneity of MMOs in terms of parallelism and resource requirements, it uses SRAM-CIM with limited resources to perform light-weight MMO-V/A/Z, which overlaps data communication and computation, thereby accelerating inference. As for FFNL, none of the 3 AIMAs has a distinct advantage. The main reason is the hardware characteristics of DRAM. Because the process of DRAM is different from that of CMOS, DRAM can only realize the CNM at bank level. Therefore, this leads to frequent read and write operations between the CNM unit and the memory cell, thus limiting the acceleration gain of DRAM-based AIMAs. It should be noted that, since the acceleration

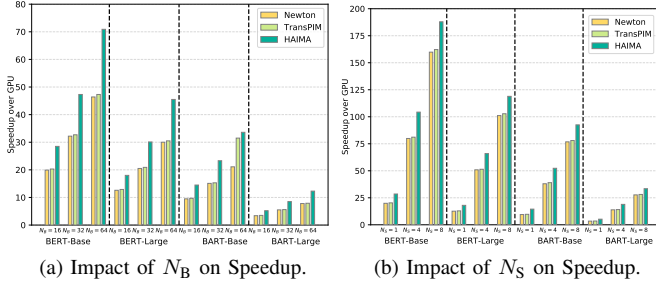


Fig. 5: Scalability Comparison Between AIMAs.

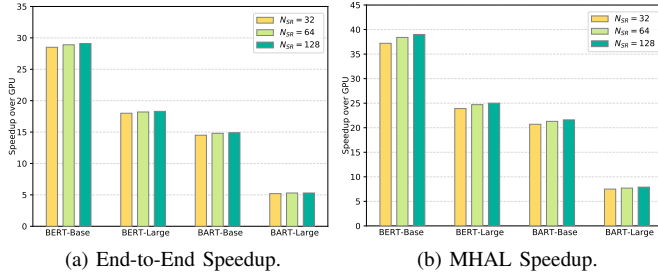


Fig. 6: Impact of N_{SR} on Speedup.

gain of SRAM-CIM is based on the 48.1% area overhead of SRAM (Section V.B), and considering the limited memory capacity of SRAM, we only use SRAM-CIM to perform light-weight MMMOs instead of all MMMOs in Transformer.

B. Area and Power Comparison

Compared to the original HBM2 with 8GB memory (53.15 mm²), in HAIMA, each CNM unit has one BPU and one 32-to-1 MUX, thus causing about 1.4% area overhead. Meanwhile, compared to the original SRAM with 64KB memory (0.26 mm²), there is a SPU in each SRAM-CIM, incurring about 48.1% area overhead. To avoid the influence of process and precision, we synthesize Newton and TransPIM in Synopsys Design Compiler and use 45nm library to measure their area and power. The minimum data processing unit of all AIMAs is set to 8-bit precision in the experiments. Meanwhile, since Newton and TransPIM have no SRAM-CIM, we only compare the area and power between the CNM units of the 3 AIMAs. As shown in Table II, the area of our CNM unit (0.047 mm²) is 1.07x and 2.94x that of Newton and TransPIM, respectively. The power of our CNM unit (3,138 mW) is 1.01x and 3.92x that of Newton and TransPIM, respectively.

C. Scalability Comparison

As shown in Figure 5, with the increase of N_B and N_S , the speedup of the 3 AIMAs is significantly improved. It can also be seen that HAIMA is the best performer. The reason for the difference in the acceleration gain on the 2 models is that the parameter scale of BART is larger than that of BERT. This demonstrates the advantage of DRAM-CNM in accelerating LSDNN. As shown in the figures, the speedup of increasing N_S is larger than that of increasing N_B . The reason is that increasing N_B will increase the communication between banks, so the speedup is affected. As shown in Figure 6, although increasing the number of SRAM-CIMs

(i.e., N_{SR}) can also improve the performance of HAIMA, the improvement is very limited. This is because the workload of MMIO-V/A/Z is a relatively small part of the total workload of Transformer, thus limiting the gain acceleration of SRAM-CIM. In addition, the delay $\mathcal{V}$ waits for $\mathcal{S}$ also affects the performance of HAIMA. The fundamental reason, of course, is that SRAM resources are very limited. The experimental results prove the necessity of collaboration between SRAM-CIM and DRAM-CNM to accelerate LSDNN.

VI. CONCLUSION

This paper proposes the HAIMA and parallel dataflow for Transformer, which exploits the cooperation between SRAM and DRAM to accelerate different MMMOs. Compared to other DRAM-based AIMAs, our proposed hardware-software co-design achieves 1.4x-1.5x speedup, and solves the problem of resource under-utilization when DRAM-based AIMA performs the light-weight MMMOs. Our work not only demonstrates the respective characteristics of SRAM- and DRAM-based AIMAs, but also opens opportunities for exploiting the hybrid AIMA for LSDNN acceleration.

Acknowledgement: This work was supported in part by NSFC (Nos. 62202154, 62072165, and U19A2058.), in part by the National Key Research and Development Program of China (No. 2021YFB0300300), in part by the Project funded by China Postdoctoral Science Foundation (No. 2022TQ0102), and in part by the Fundamental Research Funds for the Central Universities.

REFERENCES

- [1] J. Devlin et al., "BERT: pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.
- [2] Z. Liu et al., "Swin transformer: Hierarchical vision transformer using shifted windows," in ICCV, 2021, pp. 9992–10002.
- [3] D. Kim et al., "An overview of processing-in-memory circuits for artificial intelligence and machine learning," IEEE J. Emerg. Sel. Topics Circuits Syst., vol. 12, no. 2, pp. 338–353, 2022.
- [4] C. Eckert et al., "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in ISCA, 2018, pp. 383–396.
- [5] M. He et al., "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in MICRO, 2020, pp. 372–385.
- [6] M. E. Sinangil et al., "A 7-nm compute-in-memory SRAM macro supporting multi-bit input, weight and output and achieving 351 TOPS/W and 372.4 GOPS," IEEE J. Solid State Circuits, vol. 56, no. 1, pp. 188–198, 2021.
- [7] H. Kim et al., "Colonnade: A reconfigurable sram-based digital bit-serial compute-in-memory macro for processing neural networks," IEEE J. Solid State Circuits, vol. 56, no. 7, pp. 2221–2233, 2021.
- [8] M. Zhou et al., "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in HPCA, 2022, pp. 1071–1085.
- [9] S. Pati et al., "Demystifying BERT: implications for accelerator design," arXiv preprint arXiv:2104.08335, 2021.
- [10] M. Lewis et al., "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in ACL, 2020, pp. 7871–7880.
- [11] M. O'Connor et al., "Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems," in MICRO, 2017, pp. 41–54.
- [12] J. Zhang et al., "Sealer: In-sram AES for high-performance and low-overhead memory encryption," in ISLPED, 2022, pp. 14:1–14:6.
- [13] V. Seshadri et al., "Ambic: in-memory accelerator for bulk bitwise operations using commodity DRAM technology," in MICRO, 2017, pp. 273–287.
- [14] M. Joshi et al., "Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension," in ACL, 2017, pp. 1601–1611.
- [15] A. Cohan et al., "A discourse-aware attention model for abstractive summarization of long documents," in NAACL-HLT, 2018, pp. 615–621.