ReTransformer: ReRAM-based Processing-in-Memory Architecture for Transformer Acceleration

Xiaoxuan Yang
Duke University
Durham, NC, USA
xy92@duke.edu

Bonan Yan
Duke University
Durham, NC, USA
bonan.yan@duke.edu

Hai Li
Duke University
Durham, NC, USA
hai.li@duke.edu

Yiran Chen
Duke University
Durham, NC, USA
yiran.chen@duke.edu

ABSTRACT

Transformer has emerged as a popular deep neural network (DNN) model for Neural Language Processing (NLP) applications and demonstrated excellent performance in neural machine translation, entity recognition, etc. However, its scaled dot-product attention mechanism in auto-regressive decoder brings a performance bottleneck during inference. *Transformer* is also computationally and memory intensive and demands for a hardware acceleration solution. Although researchers have successfully applied ReRAM-based Processing-in-Memory (PIM) to accelerate convolutional neural networks (CNNs) and recurrent neural networks (RNNs), the unique computation process of the scaled dot-product attention in *Transformer* makes it difficult to directly apply these designs. Besides, how to handle intermediate results in Matrix-matrix Multiplication (*MatMul*) and how to design a pipeline at a finer granularity of *Transformer* remain unsolved. In this work, we propose ReTransformer – a ReRAM-based PIM architecture for *Transformer* acceleration. ReTransformer can not only accelerate the scaled dot-product attention of *Transformer* using ReRAM-based PIM but also eliminate some data dependency by avoiding writing the intermediate results using the proposed matrix decomposition technique. Moreover, we propose a new sub-matrix pipeline design for multi-head self-attention. Experimental results show that compared to GPU and Pipelayer, ReTransformer improves computing efficiency by 23.21× and 3.25×, respectively. The corresponding overall power is reduced by 1086× and 2.82×, respectively.

CCS CONCEPTS

• **Hardware → Analysis and design of emerging devices and systems**.

KEYWORDS

Transformer, processing-in-memory, ReRAM

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICCAD '20, November 2–5, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415640

ACM Reference Format:
Xiaoxuan Yang, Bonan Yan, Hai Li, and Yiran Chen. 2020. ReTransformer: ReRAM-based Processing-in-Memory Architecture for Transformer Acceleration. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415640

1 INTRODUCTION

Natural Language Processing (NLP) is an important sector of Artificial Intelligence (AI) that enables computers to process human languages. Today, the resounding success of deep learning has advanced NLP research by introducing different architectures of deep neural network (DNN) models, such as LSTM [13], RNN [21], GRU [5]. Among the DNN models for sequence-based NLP tasks, attention mechanism is a popular and powerful tool for its ability to handle various lengths and focus on the most relevant parts in the sequence. Very recently, an outstanding self-attention based model – *Transformer* [28] significantly reduce the path length between long-range dependencies and thus achieve state-of-the-art performance in many transduction tasks.

Nevertheless, *Transformer* models that have already optimized for simplicity still require considerable computational resources owing to the complicated nature of the NLP sequences. For instance, the original *Transformer* model proposed in [28], which is the backbone of many designs [1, 30, 34], has 65M parameters. As another example, the popular pre-trained model *BERT* [6] has 108M parameters. *ALBERT* [16] proposes to prune the parameters and reduce the total parameter number down to 12M with a 2.2% accuracy degradation. Structure-level optimizations have also been developed, such as replacing the sequential model with an average attention model [34], exploring non-autoregressive decoders [9], and sharing the weights between adjacent layers [30]. However, the structural adjustments incur either the loss of intrinsic word dependencies in the model or a more complicated training process. Moreover, these acceleration methods do not consider hardware characteristics.

In order to boost *Transformer* inference performance without forfeiting accuracy, mobile and embedded systems start to adopt energy-efficient domain-specific hardware accelerators. A major performance bottleneck – the heavy use of dot-product attention in auto-regressive decoders [30], requires special handling during the acceleration. Frequent moving intermediate data back and forth between memories and processing units should also be reduced to improve energy efficiency.

We identify processing-in-memory (PIM) as an effective architecture to accelerate the execution of *Transformer* through performing the computation within memory macros. PIM dot-product engines can be efficiently implemented using resistive random-access memory (ReRAM), a type of emerging nonvolatile memory [4, 29]. Prior ReRAM-based PIM works [20, 24, 27] have demonstrated their potentials in performing vector-matrix multiplications (VMM) compared with pure CMOS accelerator architectures. However, the existing designs only support general CNNs and RNNs, and cannot be directly applied to *Transformer* due to the following reasons:

- *Transformer* has to perform many matrix-matrix multiplication (*MatMul*) operations, in which both matrices are intermediate results from the previous layers. Prior works require writing these intermediate results onto the computing device. Such operations may pause the computation process and degrade speed and energy efficiency.
- The existing CNN accelerator designs mainly focus on fully-connect (FC) and convolutional (CONV) computations as these two layers dominate the computational cost. However, *Transformer* involves different computations that are introduced by scaled dot-product attention.
- Most of the previous accelerator pipeline designs are at the layer granularity. The pipeline design of *Transformer* accelerator, however, needs to focus on a finer granularity.

To combat these challenges, in this paper, we propose ReTransformer, a ReRAM-based PIM architecture to accelerate *Transformer* for NLP sequence tasks. ReTransformer uses matrix decomposition to avoid writing the intermediate results and remove the data dependency. Besides, ReTransformer is particularly designed to support the scaled dot-product attention in *Transformer*. ReTransformer also designs a sub-matrix pipeline to further improve the throughput.

The main contributions of our works can be summarized as follows:

(1) We propose ReTransformer, a ReRAM-based PIM architecture for *Transformer* inference acceleration. We also evaluate the power and energy efficiency of ReTransformer w.r.t. GPU platform and existing ReRAM-based designs.
(2) We propose optimized *MatMul*that uses matrix decomposition in scaled dot-product attention to eliminate the data dependency and reduce the computation latency.
(3) We exploit in-memory logic techniques to implement ReRAM-based hybrid *softmax* in order to save the system power consumption.
(4) We propose a sub-matrix pipeline design at a fine granularity for *Transformer* inference.

The remainder of this paper is organized as follows: Section 2 introduces the backgrounds about *Transformer* model and ReRAM-based PIM designs; Section 3 explains the motivations of ReTransformer design; Section 4 presents the overall architecture and implementation of ReTransformer; Section 5 presents our experimental setup, results, and discussion; Section 6 concludes our work.

2 BACKGROUND

2.1 Transformer Model

Self-attention based neural models [28] are under intense investigation in the field of NLP thanks to its excellent performance in capturing semantic dependencies between words. With the help of transfer learning, pre-trained language models, such as *BERT* [6] and *XLNet* [31], have largely improved state-of-the-art performance in various NLP tasks: the multi-head self-attention of *Transformer* can connect the input and output sequences of length n with $O(1)$ operations. As a comparison, traditional sequential models (e.g., RNN, LSTM, etc.) require $O(n)$ operations to complete the same function. In this part, we introduce the structure and methodology of *Transformer* model.

2.1.1 Transformer Model Structure. *Transformer* is composed of an encoder stack and a decoder stack, which share a similar module structure. Take the structure in [28] as an example, the encoder has a stack of 6 identical blocks, as shown in Fig. 1(a). Each block consists of two major functional modules: multi-head self-attention and feed forward, as respectively shown by the light-red block and blue blocks in Fig. 1(b). After each of these two major functional modules, there is a residual block to add the input and the output and perform layer norm calculation, as depicted by the green blocks in Fig. 1(b). The multi-head attention layer takes input from the input embedding or previous encoder block. The decoder stack also consists of 6 identical blocks but each decoder block consists of three major functional layers, including two multi-head self-attention layers and a feed forward layer, as shown in Fig. 1(c). Compared with the encoder block, the decoder block has an extra encoder-decoder self-attention module which has one input connected with the output from the encoder stack.

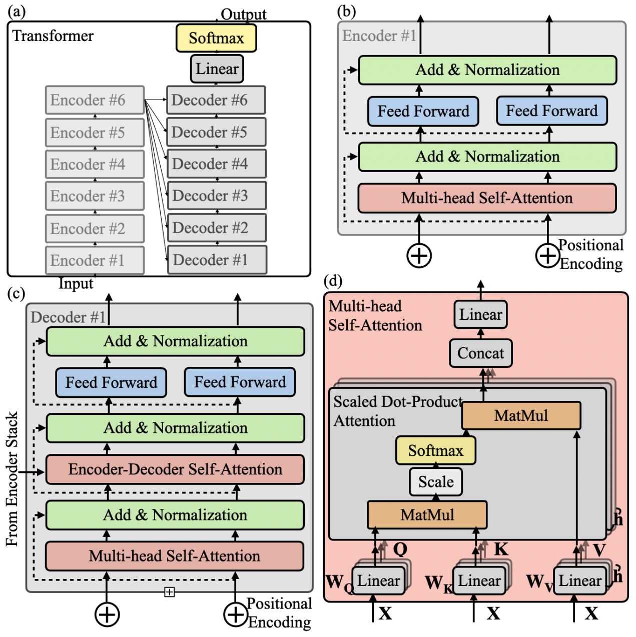


Figure 1: Transformer model structure: (a) encoder and decoder stacks, (b) encoder structure, (c) decoder structure, (d) calculations in multi-head self-attention module.

2.1.2 Transformer Model Methodology. As shown in Fig. 1(d), each head in the multi-head self-attention layer contains three trainable matrices, namely, Query matrix $\mathbf{W_Q}$, Key matrix $\mathbf{W_K}$,and Value matrix $\mathbf{W_V}$ in linear layers. Queries $\mathbf{Q}$, Keys $\mathbf{K}$ and Values $\mathbf{V}$ are generated by multiplying input sequence $\mathbf{X} = [X_0, X_1, ...X_n]$ with the Query, Key and Value matrices correspondingly as:

$$\mathbf{Q}, \mathbf{K}, \mathbf{V} = \mathbf{X} \cdot \mathbf{W_Q}, \mathbf{X} \cdot \mathbf{W_K}, \mathbf{X} \cdot \mathbf{W_V}. \tag{1}$$

Here $\mathbf{X} \in \mathbb{R}^{n \times d_{model}}$, $\mathbf{W_Q}, \mathbf{W_K} \in \mathbb{R}^{d_{model} \times d_k}$, $\mathbf{W_V} \in \mathbb{R}^{d_{model} \times d_v}$, $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{n \times d_k}$, $\mathbf{V} \in \mathbb{R}^{n \times d_v}$. d_{model}, d_k, d_v are the dimensions of model, key, and value, respectively. These layers are linear layers as shown in Fig. 1(d). Here multi-head self-attention mechanism is applied to improve performance by enabling the model to focus on different positions. The attention score is calculated using scaled dot-product attention layer, i.e.,

$$Attention(\mathbf{K}, \mathbf{Q}, \mathbf{V}) = softmax(\frac{\mathbf{Q} \cdot \mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}) \cdot \mathbf{V}, \tag{2}$$

where the result $\in \mathbb{R}^{d_k \times d_v}$. This scaled dot-product attention layer contains *MatMul*, scale, and *softmax* functions. The first *MatMul* multiplies matrix $\mathbf{Q}$ with matrix $\mathbf{K}^{\mathrm{T}}$. Scale multiplies the *MatMul* result with a scaling factor $\frac{1}{\sqrt{d_k}}$. *Softmax* for a d_k-dimensional vector is calculated as:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{d_k} e^{x_j}}. \tag{3}$$

The second *MatMul* multiplies *softmax* result with matrix $\mathbf{V}$. Multi-head attention result is achieved by multiplying the concatenation over the attention results of every head and weight matrix $\mathbf{W_O}$ as:

$$\begin{aligned} \mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head}^1, \ldots, \mathrm{head}^h\Big) \cdot \mathbf{W_O}, \\ \text{where } \mathrm{head}^i &= \mathrm{Attention}\left(X \cdot \mathbf{W_Q}^i, X \cdot \mathbf{W_K}^i, X \cdot \mathbf{W_V}^i\right). \end{aligned} \tag{4}$$

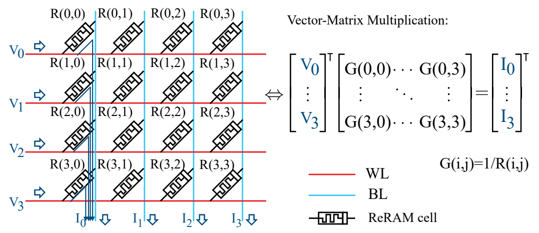
Here h is the number of heads in multi-head attention. $\mathbf{W_Q}^i, \mathbf{W_K}^i \in \mathbb{R}^{d_{model} \times d_k}$, $\mathbf{W_V}^i \in \mathbb{R}^{d_{model} \times d_v}$, $\mathbf{W_O} \in \mathbb{R}^{hd_v \times d_{model}}$. The outputs of self-attention are then passed through the position-wise feed forward layer according to

$$FFN(\mathbf{X}) = max(0, \mathbf{X} \cdot \mathbf{W_1} + b_1) \cdot \mathbf{W_2} + b_2, \tag{5}$$

where $\mathbf{W_1} \in \mathbb{R}^{d_{model} \times d_{ff}}$, $\mathbf{W_2} \in \mathbb{R}^{d_{ff} \times d_{model}}$, and the result $FFN(x) \in \mathbb{R}^{d_{model}}$. Here d_{ff} is the dimension of the hidden layer.

2.2 ReRAM-based PIM Designs

ReRAM is one of the most promising emerging nonvolatile memories featuring high density, low access energy, and feasibility of realizing multi-level cell and 3D integration [17]. A single ReRAM device stores binary or multi-bit information in the form of programmable conductance. A crossbar array of ReRAM has multiple word lines (WLs) and bit lines (BLs) that are orthogonal to each other. With the help of peripheral circuits, ReRAM crossbar arrays support PIM operations including ReRAM-based VMM and in-memory logic. In this section, we provide the basics of these operations as well as the existing architectures of ReRAM-based PIM designs.



$$\Leftrightarrow \begin{bmatrix} V_0 \\ \vdots \\ V_3 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} G(0,0) \cdots & G(0,3) \\ \vdots \quad \ddots & \vdots \\ G(3,0) \cdots & G(3,3) \end{bmatrix} = \begin{bmatrix} I_0 \\ \vdots \\ I_3 \end{bmatrix}^{\mathrm{T}}$$

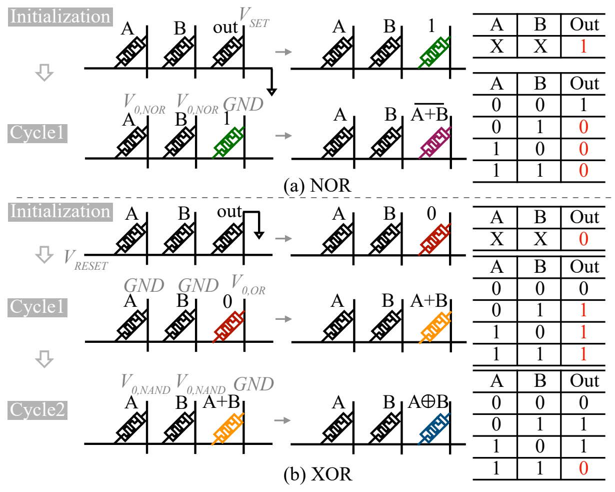
Figure 2: ReRAM-based vector-matrix multiplication.

2.2.1 ReRAM-based Vector-Matrix & Matrix-Matrix Multiplication. Fig. 2 depicts a VMM operation conducted in a heuristic example of a 4×4 ReRAM crossbar array. The conductance of each ReRAM cell represents an element in a matrix [14]. An input voltage vector $V_I = [v_0, v_1, v_2, v_3]$ is fed to the WLs. According to Kirchhoff's law, the output current generated through the j^{th} BL i_j is

$$i_j = \sum_{i=0}^{3} \frac{v_i}{R(i,j)} = \sum_{i=0}^{3} v_i G(i,j), \tag{6}$$

where $R(i,j)$ and $G(i,j)$ are respectively the cell resistance and conductance at the intersection of i^{th} WL and j^{th} BL. By organizing all of $G(i,j)$ and I_j in a matrix (or vector) as $\mathbf{G}$ and I, VMM can be expressed as $I = V_I \cdot \mathbf{G}$. The above VMM operation can be completed in one read cycle. To execute *MatMul*, the input matrix can be separated into input vectors, and then multiple VMMs are performed to obtain the *MatMul* results.

2.2.2 ReRAM-based In-memory Logic. Binary ReRAM can be used to implement in-memory logic [10, 15]. Fig. 3(a) depicts the operation principle of ReRAM-based in-memory NOR logics. The conductance values of cell "A" and "B" represent the inputs. In the tables of Fig. 3, "1" refers to high conductance, "0" refers to low conductance, and "X" refers to unknown status. The output cell is initialized to "1" first. In cycle 1, the p terminals of the input cells are set to an execution voltage $V_{0,NOR}$, while the p terminal of the output cell is grounded. As a result, the output cell will be programmed to "0" if there exists at least one "1" among cell "A" and "B". Similarly, the XOR implementation is shown in Fig. 3(b). The



A	B	Out
X	X	1

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Out
X	X	0

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3: ReRAM-based in-memory logic: (a) NOR, (b) XOR.

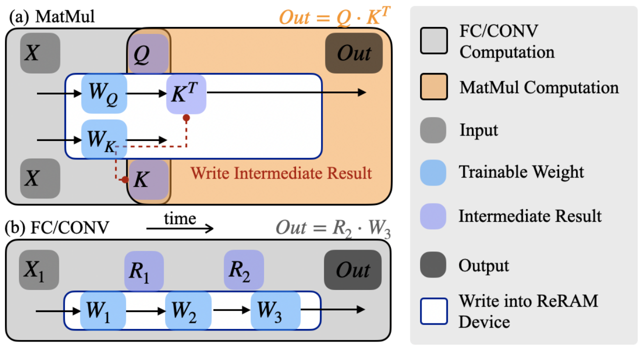


Figure 4: Comparison between FC/CONV layers and MatMul layer in ReRAM implementation: (a) FC/CONV layers, (b) MatMul layer.

execution voltages used in cycle 1 and 2 are $V_{0,OR}$ and $V_{0,NAND}$, respectively. Additionally, ReRAM in-memory logic implementation for INV, OR takes one and two cycles, respectively [10].

2.2.3 Prior ReRAM-based PIM designs. ReRAM-based PIM designs accelerate the execution of neural network models with high power efficiency. PRIME [4] and ISAAC [24] designs utilize ReRAM to accelerate the inference of CNNs. Pipelayer [27] enables neural network training with ReRAM, and further optimizes the computation latency by using pipelined stages and balancing computation resources. A ReRAM-based PIM design for RNN [20] extends to RNN acceleration with multiplier arrays and special function units to handle element-wise multiplication and nonlinear functions. Even though the above ReRAM-based PIM designs have explored CNN/RNN accelerations, none of these works can directly accelerate *Transformer*.

3 MOTIVATIONS

In this part, we discuss the intrinsic challenges in designing a ReRAM-based PIM architecture for *Transformer* acceleration with enhanced efficiency.

Intermediate Result Challenge: *Transformer* needs to keep many intermediate results during the execution. As shown in Fig. 4(a), the two operands $\mathbf{Q}$ and $\mathbf{K}^\mathrm{T}$ of *MatMul* in Fig. 1(d) are both obtained as intermediate results generated from previous layers. To calculate the subsequent $\mathbf{Q} \cdot \mathbf{K}^\mathrm{T}$, one of these two matrices has to be loaded to the ReRAM array as a conductance matrix to perform PIM *MatMul*. However, constrained by the programming driver circuits, programming a matrix into a ReRAM array is often done column by column with a typical latency of $7.2ns$/column [25]. $\mathbf{Q}$ and $\mathbf{K}^\mathrm{T}$ matrices with multiple columns would cause considerable long latency to load intermediate results into ReRAM arrays and stall the computation. Moreover, the iterative calculation in a multi-head self-attention module inevitably causes frequent rewriting of ReRAM cells, leading to a high write energy consumption.

We further analyze the calculation steps of the FC/CONV layers and *MatMul* layer in ReRAM-based PIM implementations. As shown in Fig. 4(b), the computations of the FC/CONV layers are mainly VMMs, the operands of which are input and weight matrix. The trained weight matrices, such as $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{W}_3$, are retained in the ReRAM crossbars during the inference. Therefore, the total number of the write accesses to the ReRAM crossbars is determined by the number of the initialization and normally much less than the total number of the read accesses to (or the computations performed on) the ReRAM crossbars. On the contrary, the intermediate result $\mathbf{K}^\mathrm{T}$ in *MatMul* operations are frequently being written into the ReRAM crossbars during the computation. Storing the intermediate results in other medium such as on-chip SRAM or off-chip DRAM may mitigate the large overheads of the write accesses to the ReRAM crossbars, but introduces either extra hardware cost or performance overhead.

The writing process of the intermediate result ($\mathbf{K}^\mathrm{T}$) cannot start until the computation of the previous linear layer ($\mathbf{K} = \mathbf{X} \cdot \mathbf{W}_\mathbf{K}$) completes. Note that there must exist a mapping scheme to map the output of the previous linear layer $\mathbf{K}$ onto the ReRAM crossbar in the transposed format, i.e., $\mathbf{K}^\mathrm{T}$. Obviously, the computation of *MatMul* ($\mathbf{Out} = \mathbf{Q} \cdot \mathbf{K}^\mathrm{T}$) cannot be performed until the writing of $\mathbf{K}^\mathrm{T}$ completes. This compute-write-compute (CWC) dependency is inherent between the linear layer and the *MatMul* layer and will be resolved in ReTransformer by decomposing the *MatMul* between $\mathbf{Q}$ and $\mathbf{K}^T$ to two multiplication steps (see Section 4.2).

Arithmetic Operation Challenge: The other operations in scaled dot-product attention, such as scale and *softmax* which require division and exponential calculation that are not supported by the existing ReRAM-based PIM designs. Efficient arithmetic processing units are necessary to complete them.

Low Crossbar Utilization Challenge: The traditional pipelining of DNN models at layer granularity [27] will cause a low utilization rate of the ReRAM crossbars when *Transformer* is running on ReRAM-based PIM designs (details will be provided in Section 4.4). A delicate pipeline design should be put forward to avoid such low utilization of ReRAM crossbars.

4 METHODS

To address these challenges, we propose ReTransformer, an improved architecture with novel methodologies for ReRAM-based PIM design that are particularly suited for *Transformer* alike applications involving the self-attention mechanism.

4.1 Overall Architecture

Fig. 5 presents the overall architecture. A ReRAM-based PIM module is divided into three types of functional components:

- **Processing subarrays** are the data processing engines to execute the computations of scaled dot-product attention and feed forward. More specifically, they are composed of ReRAM crossbar arrays, computing controller circuits as well as analog/digital interface circuits [4].
- **Buffer subarrays** serve as the cache for computing units. They receive data from memory subarrays and pass the data to WLs of processing subarrays. Besides, they collect results from BLs and store the data. The size of buffer subarrays is determined by the application requirements.
- **Memory subarrays** are dedicated memories to store the original input data and final calculation results.

As described in Section 2.1.2, *Transformer* encoder and decoder have feed forward and multi-head self-attention modules; and the key

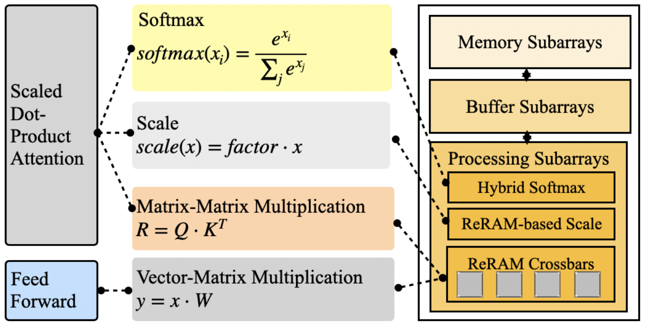

Scaled Dot-Product Attention
Feed Forward
Softmax
$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
Scale
$scale(x) = factor \cdot x$
Matrix-Matrix Multiplication
$R = Q \cdot K^T$
Vector-Matrix Multiplication
$y = x \cdot W$
Memory Subarrays
Buffer Subarrays
Processing Subarrays
Hybrid Softmax
ReRAM-based Scale
ReRAM Crossbars

Figure 5: Overview of the proposed ReRAM-based PIM design for Transformer.

computation part of scaled dot-product attention includes *MatMul*, scale and *softmax*.

Feed forward computation can be efficiently completed by PIM VMM, which has been well explored in prior literature [4, 24]. For this type of operations, ReTransformer follows the same route.

MatMul operations involve multiple VMMs in a sequential manner. Due to the recurrent nature of the self-attention layer, *MatMul* operations are often accompanied with recursive data dependency.

Scale operations refer to the scale factor $1/\sqrt{d_k}$ in scaled dot-product attention in Eq. (2). d_k is the width of the trained matrices $\mathbf{W_Q}$ and $\mathbf{W_K}$. A common setting of d_k is 64 in [28]. The corresponding scaling operation, hence, is achieved by using a 3-bit left-shift, and thereby the results are stored to the ReRAM crossbar for further processing. When the square root of d_k is not a power of 2, we can combine the bit shifting with a constant multiplication (storing the constant value in the ReRAM crossbar array and multiplying the constant with the concerned matrices) to construct the scaling operation.

Softmax operations appear at every multi-head self-attention module to attain a high NLP accuracy. In contrast, *softmax* operations only exist in the last output layer of many other DNNs, such as ResNet [12].

Among all the above computations, *MatMul* and *softmax* dominate the latency and power consumption. Therefore, we propose three major techniques around these two dominant operations in order to optimize the overall performance and power consumption of executing multi-head self-attention modules.

4.2 Optimized *MatMul*

In this section, we present the implementation details of *MatMul* in ReTransformer. Our primary goal is to reduce the number of frequent reloadings of intermediate results.

In Section 3, we discussed the CWC dependency that exists between the linear layer and *MatMul* in *Transformer*. Fig. 6(a) illustrates the data generation and computation process of scaled dot-product attention layer in the inference of *Transformer* between $\mathbf{W_Q}$ and $\mathbf{W_K}$, and shows the CWC dependency caused by the intermediate result $\mathbf{K}$. At beginning of the inference, W_Q and W_K are initialized and written into two ReRAM crossbars, respectively. To perform the *MalMul* between $\mathbf{Q}$ and $\mathbf{K}$ (i.e., $\mathbf{Out} = \mathbf{Q} \cdot \mathbf{K}^\mathrm{T}$) in the ReRAM-based PIM module, one of the intermediate results $\mathbf{Q}$ and $\mathbf{K}$, must be stored in a ReRAM crossbar. Without loss of generality,

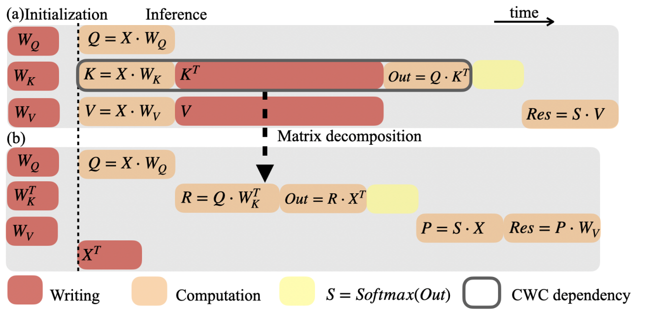


Figure 6: Remove data dependency in scaled dot-product attention layer: (a) a CWC dependency caused by the intermediate result K. (b) The optimized *MatMul* eliminates the CWC dependency by decomposing the computation into two cascaded multiplications.

here we assume $\mathbf{K}$ (indeed $\mathbf{K}^\mathrm{T}$) is stored in the ReRAM crossbar. As demonstrated in the memory designs [4, 24], the ReRAM crossbar needs to be programmed row-by-row; the total latency of programming a $N \times N$ ReRAM crossbar is N write cycles by assuming every cycle we can program a whole crossbar row. As we have illustrated in Section 2.2, the computation latency of VMM on a ReRAM crossbar is one read cycle.

We find that *MalMul* between $\mathbf{Q}$ and $\mathbf{K}$ can be decomposed into two cascaded multiplication steps as:

$$\mathbf{Out} = \mathbf{Q} \cdot \mathbf{K}^\mathrm{T} = \mathbf{Q} \cdot (\mathbf{X} \cdot \mathbf{W_K})^\mathrm{T} = (\mathbf{Q} \cdot \mathbf{W_K}^\mathrm{T}) \cdot \mathbf{X}^\mathrm{T}. \tag{7}$$

Hence, a possible way to avoid writing the intermediate results into a ReRAM crossbar can be the follows: at the beginning of the process, we initialize two ReRAM crossbars with $\mathbf{W_Q}$ and $\mathbf{W_K}^\mathrm{T}$ (instead of $\mathbf{W_K}$), respectively. Then compute $\mathbf{Q} = \mathbf{X} \cdot \mathbf{W_Q}$ on the ReRAM crossbar storing $\mathbf{W_Q}$ and $\mathbf{R} = \mathbf{Q} \cdot \mathbf{W_K}^\mathrm{T}$ on the ReRAM crossbar storing $\mathbf{W_K}^\mathrm{T}$. In the meanwhile, we initialize another ReRAM crossbar with $\mathbf{X}^\mathrm{T}$. After we obtain both $\mathbf{Q}$ and $\mathbf{R}$, we perform $\mathbf{Out} = \mathbf{R} \cdot \mathbf{X}^\mathrm{T}$ on the ReRAM crossbar storing $\mathbf{X}^\mathrm{T}$. The whole process is illustrated in Fig. 6(b). The CWC dependency caused by the intermediate result $\mathbf{K}^\mathrm{T}$ is resolved and the long latency of writing $\mathbf{K}^\mathrm{T}$ into the ReRAM crossbar is eliminated.

The *MatMul* computation process between $\mathbf{S}$ and $\mathbf{V}$ can be optimized in a similar way. However, the computation threads of $\mathbf{W_K}$ and $\mathbf{W_V}$ must respectively retain a copy of $\mathbf{X}^\mathrm{T}$ and $\mathbf{X}$ if both threads need to be performed simultaneously. To avoid the large overhead of write accesses to ReRAM crossbars, we propose to use only one copy of $\mathbf{X}$ to perform $\mathbf{Out} = \mathbf{Q} \cdot \mathbf{X}^\mathrm{T}$ and $\mathbf{P} = \mathbf{S} \cdot \mathbf{X}$ in a sequence. Here, a dual-access design of ReRAM crossbars [18] must be implemented to support the multiplications with $\mathbf{X}$ and $\mathbf{X}^\mathrm{T}$ on the same crossbar. Besides, the computation of $\mathbf{P} = \mathbf{S} \cdot \mathbf{X}$ cannot be performed until the computation of $\mathbf{Out} = \mathbf{Q} \cdot \mathbf{X}^\mathrm{T}$ completes.

4.3 ReRAM-Based Hybrid *Softmax*

This section describes another technique, ReRAM-based hybrid *softmax*, to efficiently compute *softmax* in the scaled dot-produce attention with specialized hardware. The fundamental idea is incorporating in-memory logic to maximally exploit on-chip ReRAM crossbar arrays in the processing subarrays. We base our choice of

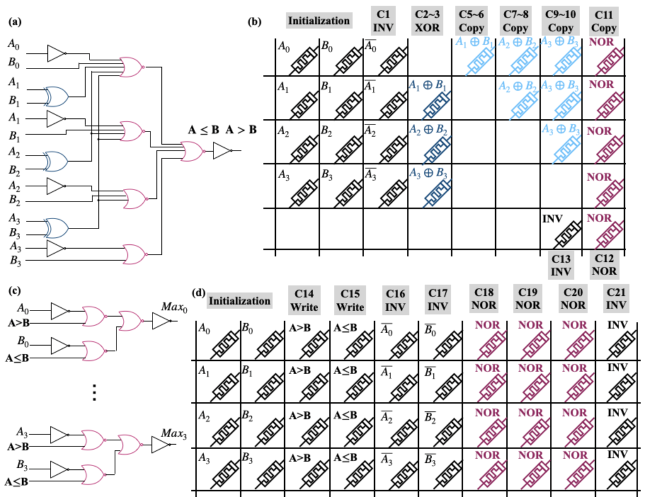


Figure 7: Compare and select logics CMOS and ReRAM implementations: (a) compare CMOS implementation, (b) compare ReRAM implementation, (c) select CMOS implementation (d) select ReRAM implementation.

in-memory logic technology on the consideration of moving post-processing closer to the processing subarrays and lowering the power consumption. Note that device endurance remains an open issue in contemporary in-memory logic designs that using emerging ReRAM devices [11, 26]. Fortunately, the rapidly advancing fabrication and compilation [26] technologies offer a good promise to significantly enhance the reliability.

First, we adapt the *softmax* expression in order to avoid the overflow problem of *softmax* calculation and circumvent the usage of *division* operation, which would result in costly digital circuit implementation with very high design complexity [32]. In particular, we use negative numbers in *exponential* step and substitute *division* with *logarithm*. Every element x_i $(i = 0, ..., d_k - 1)$ in the input vector X of length d_k is subtracted by the largest element in this vector, i.e., $x_{max} := max(x_0, ..., x_{d_k - 1})$. Therefore, in our design, *softmax* is calculated as:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{d_k} e^{x_j}} = \frac{e^{x_i - x_{max}}}{\sum_{j=1}^{d_k} e^{x_j - x_{max}}} = \exp[x_i - x_{max} - \log(\sum_{j=1}^{d_k} e^{x_j - x_{max}})]. \quad (8)$$

Next, in-memory logic is introduced to search x_{max} with minimum power consumption. In particular, we build ReRAM-based compare and select logics to find the x_{max} and use look-up tables to perform the *exponential* and *logarithm* functions in Eq. (8). Take two 4-bit number $\mathbf{a} = A_3A_2A_1A_0$ and $\mathbf{b} = B_3B_2B_1B_0$ as an example, the CMOS implementations of compare and select logics are depicted in Fig. 7(a) and (c), respectively. Iterative compare and select logic are fed with the multi-bit elements sequentially in sequences X to reach the final maximum element. Chances are that the dynamic power consumption is high considering the large dimension of the possible *softmax* input dimension (e.g., typical value is 64 for *Transformer* [28]).

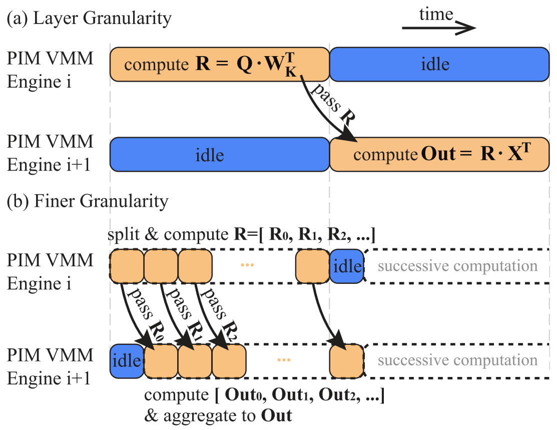

(a) Layer Granularity
time
PIM VMM Engine i
compute $\mathbf{R} = \mathbf{Q} \cdot \mathbf{W_K^T}$
idle
pass R
PIM VMM Engine i+1
idle
compute $\mathbf{Out} = \mathbf{R} \cdot \mathbf{X^T}$
(b) Finer Granularity
split & compute R=[R₀, R₁, R₂, ...]
idle
successive computation
pass R₀
pass R₁
pass R₂
compute [Out₀, Out₁, Out₂, ...]
& aggregate to Out

Figure 8: Pipeline designs: (a) at layer granularity, (b) at finer granularity.

Fig. 7(b) and (d) illustrate the corresponding ReRAM-based in-memory logic implementations of compare and select. As shown in Fig. 7(b), the computation steps of the compare block are as follows:

Step 1: Initialization. Cells $A_0 \sim A_3$ and $B_0 \sim B_3$ are written into the ReRAM crossbars;

Step 2: Logical Inference. The information propagates in the ReRAM crossbar array in the sequence of $INV \rightarrow XOR \rightarrow Copy \rightarrow Copy \rightarrow Copy \rightarrow NOR \rightarrow NOR \rightarrow INV$ at the pace of successive clock cycles[1], where *Copy* is the operation that takes two cycles to duplicate the single-bit information stored in one cell to another.

Therefore, the ReRAM implementation of compare logic consumes 13 cycles. Similarly, ReRAM-based select logic takes 8 cycles to perform *Write* (i.e., programming a ReRAM cell), *INV*, and *NOR*. The operational cycle numbers of each basic logic operation can be found in Section 2.2.2. After x_{max} is found, we adopt developed look-up table circuits to calculate *exponential* and *logarithm* as [2] due to the maturity of the related circuits.

4.4 Sub-Matrix Pipeline

In this section, we will design pipeline stages of ReTransformer for a higher throughput at a finer granularity than the layer granularity adopted in existing PIM-based DNN accelerators [27].

First of all, we need to understand why the conventional layer granularity causes low utilization rate of processing subarrays. Based on the discussion in Section 4.2, we have adjusted the computation sequence in *MatMul* such that two multiplications should proceed sequentially, as shown in Fig. 8(a). If the pipeline stage is arranged at the layer granularity, the intra-layer computation details cannot be handled efficiently. For this *MatMul* layer, two computation resources (denoted by $\mathbf{W_K}^T$ and $\mathbf{X}^T$) are prepared. The input matrix $\mathbf{Q}$ is first fed into resource $\mathbf{W_K}^T$ to compute $\mathbf{R}$ (Step 1), and then $\mathbf{R}$ is fed into resource $\mathbf{X}^T$ to compute **Out** (Step 2). Both $\mathbf{W_K}^T$ and $\mathbf{X}^T$ are released only after both steps 1 and 2 complete. However, $\mathbf{X}^T$ is idle during step 1 and $\mathbf{W_K}^T$ is idle during step 2, leading to an inefficient utilization of these two resources (i.e., the processing subarrays storing $\mathbf{W_K}^T$ and $\mathbf{X}^T$).

[1]According to the measurement in [25], we assume the read and write cycle are the same and defined as unit clock cycle. Clock cycle is identified in "Ck" (where $k = 1, 2, 3, ...$) in Fig. 7.

Table 1: Experimental setup details.

(a) GPU Configurations.

GPU	NVIDIA TITAN RTX
Memory	24 GB
Memory Bandwidth	672 GB/s
CUDA Cores	4,608
CUDA Version	10.1
Power Consumption	280W

(b) ReTransformer Configurations.

ReRAM crossbar	Subarray size	128×128
	Cell percision	2 bit
	R_{off}/R_{on}	$1M\Omega$ /$100k\Omega$
ADC	Percision	5 bit
Shift and add	Percision	14 bit

In order to suppress idle periods of the processing subarrays and keep them busy for better system throughput and higher power efficiency, we propose to slice input matrices of *MatMul* operation inside a scaled dot-product attention into small segment vectors for a finer pipeline granularity. When multiplying two matrices inside a multi-head self-attention layer (e.g., $\mathbf{R} = \mathbf{Q} \cdot \mathbf{W_K}^{\mathrm{T}}$ in the first step of Eq. (2)), $\mathbf{Q}$ is reshaped into a long vector by horizontally concatenating its rows as an input of PIM VMM: $\mathbf{Vec\_Q} = [\mathbf{Q}[0,:], \mathbf{Q}[1,:], \mathbf{Q}[n,:]]$. Here $\mathbf{Vec\_Q} \in \mathbb{R}^{1 \times nd_q}$ is the input fed to the ReRAM crossbar storing $\mathbf{W_K}^{\mathrm{T}}$. In the first cycle, we grab the first segment of $\mathbf{Vec\_Q}$ and conduct PIM VMM, i.e., $\mathbf{R}_0 = \mathbf{Vec\_Q}[0, 0 : d_k - 1] \cdot \mathbf{W_K}^{\mathrm{T}}$. Note that the length of this segment is determined by the following *MatMul* to align the dimensions. This $\mathbf{R}_0 \in \mathbb{R}^{1 \times d_{model}}$ is the first row in $\mathbf{R}$, or, $\mathbf{R}[0, 0 : d_{model} - 1] = \mathbf{R}_0[:,:]$. Then, in the following cycle i, we can get

$$\begin{aligned} \mathbf{R}[i, 0 : d_{model} - 1] &= \mathbf{R}_i[:,:] \\ &= \mathbf{Vec\_Q}[0, (i-1) * d_k : i * d_k - 1] \cdot \mathbf{W_K}^{\mathrm{T}} \\ &= \mathbf{Q}[i, 0 : d_q - 1] \cdot \mathbf{W_K}^{\mathrm{T}}. \end{aligned} \tag{9}$$

Similarly, this "slice and calculate" scheme for $\mathbf{R} = \mathbf{Q} \cdot \mathbf{W_K}^{\mathrm{T}}$ can be also applied to its successive operation, $\mathbf{Out} = \mathbf{R} \cdot \mathbf{X}^{\mathrm{T}}$. Here $\mathbf{R}$ is first flattened into a vector as: $\mathbf{Vec\_R} = [\mathbf{R}[0,:], \mathbf{R}[1,:], \mathbf{R}[n,:]]$. At a finer granularity, the first computation cycle of $\mathbf{Out}$ only utilizes $\mathbf{Vec\_R}[0, 0 : d_{model} - 1]$.

In summary, the two sequential computations $\mathbf{R} = \mathbf{Q} \cdot \mathbf{K}^{\mathrm{T}}$ and $\mathbf{Out} = \mathbf{R} \cdot \mathbf{X}^{\mathrm{T}}$ introduced in Section 4.2 can construct a pipeline: every cycle the output vector from the previous computation can be consecutively fed into the successive computation unit. Fig. 8(b) shows this sub-matrix pipeline which minimizes the idle time of the processing subarrays. Quantitative analysis of this sub-matrix pipeline can be found in Section 5.4.

5 EVALUATION

5.1 Experiment Setup and Benchmark

In our experiment, we compare ReTransformer with a GPU platform and a ReRAM-based PIM design PipeLayer [27]. Benchmarks running on the GPU platform are based on Pytorch. The GPU platform configurations are summarized in Table 1(a).

We adopt the ReRAM technology from [8] to evaluate ReTransformer. We simulate the ReRAM circuit parameters using NeuroSim [22] based on the ReRAM and peripheral configurations [23] shown in Table 1(b). The precision of the ReRAM cells are set to 2-bit, which is the same as [20]. The read and write pulse times of the ReRAM cells are extracted from [25]. The high and low resistance values of ReRAM crossbars (R_{off} and R_{on}) are adopted from [3]. We utilize the design from [2] to implement the *exponential* and *logrithm* look-up tables in the hybird *softmax* design. A modified design of [7] is used as the CMOS baseline of *softmax* design.

Table 2: Transformer network configurations.

Model	N	d_{model}	d_{ff}	h	d_k	d_q
Transformer (A)	6	512	2048	8	64	64
Transformer (B)	6	1024	4096	16	64	64

The networks that we use to evaluate our proposed methods are Transformer (A) and Transformer (B), the details of which are included in Table 2. We denote these two models as Model A and Model B, respectively. Model A is the base model and its size is relatively small. Model B increases the model dimension and the head number. The activations and weights are quantized to 8 bit like [33]. The training dataset is WMT 2016 Translation Task (English to German) and the testing dataset is newstest2016.

5.2 Impacts of MatMul Optimization

Fig. 9 compares the computation latency breakdowns of *MatMul* in Model A and Model B. When the CWC dependency exists, the $\mathbf{K}^{\mathrm{T}}$ related writes contribute to majority of the overall latency, as shown in the bars marked as "Model A/B with Dependency". Through the elimination of the $\mathbf{K}^{\mathrm{T}}$ related writes, the optimized *MatMul* substantially reduces the overall latency by 1.32× and 1.16× for Model A and B, respectively.

Moreover, *MatMul* optimization decreases the number of writings to crossbar array and improve the power efficiency. We will include this result in Section 5.5.

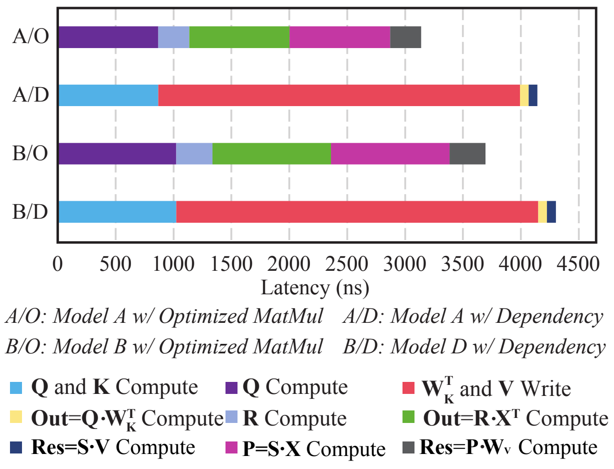
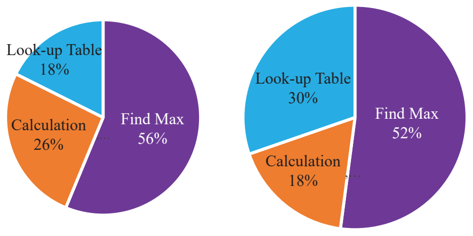


Figure 9: MatMul computation latency comparisons.



(a) Hybrid Softmax
Total Power: 0.691mW

(b) CMOS-based Softmax
Total Power: 1.023mW

Figure 10: Softmax design comparison.

Table 3: Performance comparison of two pipeline designs.

Model	Layer	Finer	Improvement
Model A	69.24$GOPs/s$	81.85$GOPs/s$	1.18×
Model B	67.89$GOPs/s$	80.07$GOPs/s$	1.18×

5.3 Lower Power Hybrid Softmax Design

Fig. 10 depicts the power consumption breakdowns of the proposed hybrid *softmax* design and the CMOS-based design. The total power of the hybrid *softmax* design is 0.6913mW while the one of the CMOS-based *softmax* design is 1.0233mW. In particular, the power consumption of compare and select logics reduces from 0.533mW in the CMOS-based design to 0.3889mW thanks to the efficiency of ReRAM-based implementation.

In CNN and RNN based networks, *softmax* layer is only involved in the last classification step. However, *Transformer* has to calculate *softmax* in the scaled dot-product attention step and the classification step in *Transformer*. For instance, Model A in our experiment has 145 *softmax* layers. Therefore, this low power hybrid *softmax* design brings improvement to the overall power efficiency of *Transformer* acceleration.

5.4 Comparisons of Two Pipeline Designs

Table 3 compares the computation throughput of two ReTransformer pipeline designs when running Model A and B. Here the hardware implementations of the two pipeline designs are exactly the same except for the pipeline method. Table 3 indicates that the finer granularity design outperforms the layer granularity design by 1.18×.

5.5 Comparison with GPU and PipeLayer

Figure 11 compares the computing efficiency and power of GPU, PipeLayer and ReTransformer. Here the computing efficiency measures the number of operations that can be performed by a computing unit every unit time and every watt of power consumed, and power measures the amount of energy transferred or converted every unit time. Our ReTransformer can achieve a computing efficiency of 467.68 $GOPs/s/W$. Compared to GPU (CMOS acceleration) and PipeLayer (conventional ReRAM-based acceleration), ReTransformer improves the computing efficiency by 23.21× and 3.25×, respectively. Besides, ReTransformer reduces the overall power by 1086× and 2.82×, respectively.

The advantage of ReTransformer is mainly because the optimized *MatMul* removes the data dependency and decrease frequent rewrites of intermediate results. Furthermore, ReTransformer utilizes a low power hybrid *softmax* design to handle the *softmax* calculation in scaled dot-product attention step.

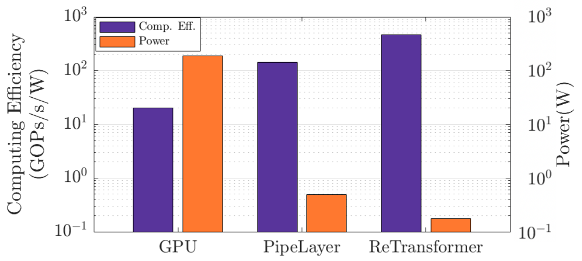


Figure 11: Performance comparison with GPU and PipeLayer.

5.6 Discussion

ReTransformer utilizes the ReRAM-based PIM architecture and focuses on reorganizing the data-dependency computation steps. Therefore, our hardware-based design is orthogonal to existing software-based *Transformer* acceleration designs. For instance, during training process, we use quantization-aware training method to obtain our model parameters. Quantization shrinks the model size by 4× as we quantize the 32 bit (GPU setting) to 8 bit (ReRAM-based setting). The Bilingual Evaluation Understudy (BLEU) score of the testing dataset only drops from 31.17 to 30.45 for Model A, implying the quantization-aware method well retained the accuracy. Moreover, the pruning methods introduced in Section 1 do not affect the implementation of ReTransformer. Random pruning can be helpful to reduce the computation cost without modifying our ReRAM programming and computation process. Structural pruning, e.g., pruning the hidden neurons in feed forward layer, may significantly improve the utilization of ReRAM resources.

ReTransformer is also compatible with the existing software and hardware reliability improving methodologies. Process variation of ReRAM, for example, results in weight inaccuracy and consequently degrades the overall computation accuracy. A variation of 2.5%, 5.0%, 7.5%, 10.0% weights mismatching from the baseline model causes 0.50%, 5.09%, 21.27%, 38.54% accuracy drop in a quantized BERT model in QNLI dataset (the baseline accuracy is 90.79%) under the same setting as [33]. Many methods specifically designed for PIM architectures have been invented to effectively recover the accuracy of a model that suffers from high weight variability to a level comparable to the original one [19]. We refer to the interested readers to [17] for more details.

6 CONCLUSION

In this work, we propose ReTransformer – a ReRAM-based PIM architecture to accelerate the computation of *Transformer* model. Besides accelerating the scaled dot-product attention of *Transformer* using ReRAM-based PIM, ReTransformer also optimizes the *MatMul* operations to remove the data dependency in the computation that hinders the computing efficiency. We also propose a new sub-Matrix pipeline design to shorten the computation latency of multi-head self-attention. Our simulations show that compared to GPU and Pipelayer, ReTransformer improves computing efficiency by 23.21× and 3.25×, respectively. The corresponding power is reduced by 1086× and 2.82×, respectively.

ACKNOWLEDGMENTS

This work is supported in part by NSF 1955246, NSF 1910299, NSF 1725456, and ARO W911NF-19-2-0107.

REFERENCES

[1] Karim Ahmed, Nitish Shirish Keskar, and Richard Socher. 2017. Weighted Transformer Network for Machine Translation. arXiv:1711.02132

[2] Diogo Brito, Taimur Gibran Rabuske, Jorge R. Fernandes, Paulo F. Flores, and José C. Monteiro. 2015. Quaternary Logic Lookup Table in Standard CMOS. *IEEE Trans. Very Large Scale Integr. Syst.* 23, 2 (2015), 306–316. https://doi.org/10.1109/TVLSI.2014.2308302

[3] Meng-Fan Chang, Pi-Feng Chiu, and Shyh-Shyuan Sheu. 2011. Circuit design challenges in embedded memory and resistive RAM (RRAM) for mobile SoC and 3D-IC. In *Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC 2011, Yokohama, Japan, January 25-27, 2011.* IEEE, 197–203. https://doi.org/10.1109/ASPDAC.2011.5722184

[4] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016.* IEEE Computer Society, 27–39. https://doi.org/10.1109/ISCA.2016.13

[5] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv:1412.3555

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805

[7] Gaoming Du, Chao Tian, Zhenmin Li, Duoli Zhang, Yong-Sheng Yin, and Yiming Ouyang. 2019. Efficient Softmax Hardware Architecture for Deep Neural Networks. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI, GLSVLSI 2019, Tysons Corner, VA, USA, May 9-11, 2019.* ACM, 75–80. https://doi.org/10.1145/3299874.3317988

[8] Rich Fackenthal, Makoto Kitagawa, Wataru Otsuka, Kirk Prall, Duane Mills, Keiichi Tsutsui, Johnny Javanifard, Kerry Tedrow, Tomohito Tsushima, Yoshiyuki Shibahara, and Glen Hush. 2014. 19.7 A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *2014 IEEE International Conference on Solid-State Circuits Conference, ISSCC 2014, Digest of Technical Papers, San Francisco, CA, USA, February 9-13, 2014.* IEEE, 338–339. https://doi.org/10.1109/ISSCC.2014.6757460

[9] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O. K. Li, and Richard Socher. 2017. Non-Autoregressive Neural Machine Translation. arXiv:1711.02281

[10] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. FELIX: fast and energy-efficient logic in memory. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018.* ACM, 55. https://doi.org/10.1145/3240765.3240811

[11] Runze Han, Peng Huang, Yudi Zhao, Zhe Chen, Lifeng Liu, Xiaoyan Liu, and Jinfeng Kang. 2017. Demonstration of logic operations in high-performance RRAM crossbar array fabricated by atomic layer deposition technique. *Nanoscale research letters* 12, 1 (2017), 1–6. https://doi.org/10.1186/s11671-016-1807-9

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016.* IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[14] Miao Hu, Hai Li, Qing Wu, Garrett S. Rose, and Yiran Chen. 2012. Memristor crossbar based hardware realization of BSB recall function. In *The 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Australia, June 10-15, 2012.* IEEE, 1–7. https://doi.org/10.1109/IJCNN.2012.6252563

[15] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. 2014. MAGIC - Memristor-Aided Logic. *IEEE Trans. Circuits Syst. II Express Briefs* 61-II, 11 (2014), 895–899. https://doi.org/10.1109/TCSII.2014.2357292

[16] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. arXiv:1909.11942

[17] Bing Li, Bonan Yan, Chenchen Liu, and Hai (Helen) Li. 2019. Build reliable and efficient neuromorphic design with memristor technology. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019.* ACM, 224–229. https://doi.org/10.1145/3287624.3288744

[18] Shuo Li, Nong Xiao, Peng Wang, Guangyu Sun, Xiaoyang Wang, Yiran Chen, Hai Helen Li, Jason Cong, and Tao Zhang. 2019. RC-NVM: Dual-Addressing Non-Volatile Memory Architecture Supporting Both Row and Column Memory Accesses. *IEEE Trans. Computers* 68, 2 (2019), 239–254. https://doi.org/10.1109/TC.2018.2868368

[19] Mengyun Liu, Lixue Xia, Yu Wang, and Krishnendu Chakrabarty. 2019. Fault tolerance in neuromorphic computing systems. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019.* ACM, 216–223. https://doi.org/10.1145/3287624.3288743

[20] Yun Long, Taesik Na, and Saibal Mukhopadhyay. 2018. ReRAM-Based Processing-in-Memory Architecture for Recurrent Neural Network Acceleration. *IEEE Trans. Very Large Scale Integr. Syst.* 26, 12 (2018), 2781–2794. https://doi.org/10.1109/TVLSI.2018.2819190

[21] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010.* ISCA, 1045–1048.

[22] Xiaochen Peng, Shanshi Huang, Yandong Luo, Xiaoyu Sun, and Shimeng Yu. 2019. DNN+NeuroSim: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators with Versatile Device Technologies. In *2019 IEEE International Electron Devices Meeting (IEDM).* IEEE, 32.5.1–32.5.4. https://doi.org/10.1109/IEDM19573.2019.8993491

[23] Xiaochen Peng, Rui Liu, and Shimeng Yu. 2019. Optimizing Weight Mapping and Data Flow for Convolutional Neural Networks on RRAM Based Processing-In-Memory Architecture. In *IEEE International Symposium on Circuits and Systems, ISCAS 2019, Sapporo, Japan, May 26-29, 2019.* IEEE, 1–5. https://doi.org/10.1109/ISCAS.2019.8702715

[24] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016.* IEEE Computer Society, 14–26. https://doi.org/10.1109/ISCA.2016.12

[25] Shyh-Shyuan Sheu, Meng-Fan Chang, Ku-Feng Lin, Che-Wei Wu, Yu-Sheng Chen, Pi-Feng Chiu, Chia-Chen Kuo, Yih-Shan Yang, Pei-Chia Chiang, Wen-Pin Lin, Che-He Lin, Heng-Yuan Lee, Peiyi Gu, Sumin Wang, Frederick T. Chen, Keng-Li Su, Chen-Hsin Lien, Kuo-Hsing Cheng, Hsin-Tun Wu, Tzu-Kun Ku, Ming-Jer Kao, and Ming-Jinn Tsai. 2011. A 4Mb embedded SLC resistive-RAM macro with 7.2ns read-write random-access time and 160ns MLC-access capability. In *2011 IEEE International Solid-State Circuits Conference, ISSCC 2011, Digest of Technical Papers, San Francisco, CA, USA, 20-24 February, 2011.* IEEE, 200–202. https://doi.org/10.1109/ISSCC.2011.5746281

[26] Saeideh Shirinzadeh, Mathias Soeken, Pierre-Emmanuel Gaillardon, Giovanni De Micheli, and Rolf Drechsler. 2017. Endurance management for resistive Logic-In-Memory computing architectures. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017.* IEEE, 1092–1097. https://doi.org/10.23919/DATE.2017.7927152

[27] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017.* IEEE Computer Society, 541–552. https://doi.org/10.1109/HPCA.2017.55

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA.* 5998–6008.

[29] Lixue Xia, Peng Gu, Boxun Li, Tianqi Tang, Xiling Yin, Wenqin Huangfu, Shimeng Yu, Yu Cao, Yu Wang, and Huazhong Yang. 2016. Technological Exploration of RRAM Crossbar Array for Matrix-Vector Multiplication. *J. Comput. Sci. Technol.* 31, 1 (2016), 3–19. https://doi.org/10.1007/s11390-016-1608-8

[30] Tong Xiao, Yinqiao Li, Jingbo Zhu, Zhengtao Yu, and Tongran Liu. 2019. Sharing Attention Weights for Fast Transformer. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019.* ijcai.org, 5292–5298. https://doi.org/10.24963/ijcai.2019/735

[31] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada.* 5754–5764.

[32] Bo Yuan. 2016. Efficient hardware architecture of softmax layer in deep neural network. In *29th IEEE International System-on-Chip Conference, SOCC 2016, Seattle, WA, USA, September 6-9, 2016.* IEEE, 323–326. https://doi.org/10.1109/SOCC.2016.7905501

[33] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8BERT: Quantized 8Bit BERT. arXiv:1910.06188

[34] Biao Zhang, Deyi Xiong, and Jinsong Su. 2018. Accelerating Neural Transformer via an Average Attention Network. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers.* Association for Computational Linguistics, 1789–1798. https://doi.org/10.18653/v1/P18-1166