A Real-time Execution System of Multimodal Transformer through PIM-GPU Collaboration

Shengyi Ji[1], Chubo Liu[1*], Yan Ding[1*], Qing Liao[2], Zhuo Tang[1]
[1]College of Information Science and Engineering, Hunan University, Changsha, China
[2]Department of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
{jishengyi,liuchubo,ding,ztang}@hnu.edu.cn,liaoqing@hit.edu.cn

ABSTRACT

Multimodal transformer excels in various applications, but faces great challenges such as high memory consumption and limited data reuse that hinder real-time performance. To address these issues, we propose a processing-in-memory (PIM)-GPU collaboration oriented compiler to accelerate the multimodal transformers. The PIM-GPU collaboration adapts well to multimodal transformers and significantly accelerates model inference. In addition, we introduce a tailored PIM allocation algorithm for variable-length inputs to further improve computation efficiency. Experimental results show that our scheme can achieve an average 15x end-to-end speedup.

KEYWORDS

Multimodal transformer, processing-in-memory (PIM), real-time

ACM Reference Format:
Shengyi Ji, Chubo Liu, Yan Ding, Qing Liao, Zhuo Tang. 2024. A Real-time Execution System of Multimodal Transformer through PIM-GPU Collaboration. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657336

1 INTRODUCTION

In recent years, there have been significant advancements in natural language processing (NLP) technology across various scenarios. This has enabled the development of a diverse range of intelligent applications in machine translation and automatic abstracting, which have greatly facilitated people's lives and work. To further improve the intelligence capabilities of the applications, it is crucial to integrate multimodal data features (such as linguistic and visual features) for large AI models to enhance their learning and inference abilities. For instance, the BERT [1] model, dedicated to natural language understanding, has employed the pre-training and fine-tuning technique to enhance its performance. This accomplishment has sparked interest in developing visual language pre-training (VLP) models, which have the capability to tackle diverse tasks that involve a blend of visual and linguistic features, such as visual question answering, image text retrieval, and referring expression comprehension.

Multimodal models are increasingly deployed on mobile and embedded devices to provide real-time services while attracting significant investment from industry and academia. Although GPU is widely used to accelerate deep learning networks, the limited resources of embedded devices cannot meet the requirements of running GPU effectively. In particular, the frequent exchange of data between computation units and memory units (known as the memory wall problem), results in significant energy overhead and additional latency. Meanwhile, the matrix-matrix multiplication operations, as mentioned in [2], account for 82.80% of overall inference time. The inference overhead of the models is dominated by the data movement resulting from matrix-matrix multiplication operations [3]. Therefore, the complexity of the attention mechanisms of transformers, with quadratic complexity $O(N^2)$, where N is the length of the token sequence, requires significant computation and memory resources.

PIM is a novel architecture that can effectively reduce the number of data movements. This innovative design places the computation logical units either in or very close to cells of memory. Therefore, we aim to use the compiler to identify PIM offloading opportunities in multimodal models, and leverage collaboration between PIM and GPU to accelerate model inference.

However, since the disparity in resource requirements between visual and language tasks leads to the problem of resource underutilization, it is not recommended to simply combine PIM and GPU to provide computing services. There is an urgent need for an efficient parallel strategy that allows an efficient mapping between computation tasks and hardware resources by a compiler. In addition, multimodal transformers often deal with a large number of variable-length inputs. By considering both the computation tasks and the hardware resources of PIM-GPU, we design an architecture to improve resource utilization efficiency while accelerating the inference of multimodal transformers.

To address the above issues, in this paper we present a solution for real-time embedded systems aimed at optimizing PIM and GPU collaboration for multimodal transformers. The contributions are outlined below:

- We develop a compiler that collaboratively optimizes PIM and GPU for multimodal transformers. As previously discussed, data movement during matrix multiplication predominantly influences the inference efficiency. Despite PIM's limitations in handling complex operations, it is a preferred option for inference acceleration.
- This paper aims to enhance the collaboration between PIM and GPU by incorporating PIM into the compiler design for

* Corresponding author.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657336

the concurrent acceleration of both processing units. We propose a dynamic programming algorithm to strategically offload various operations to PIM and GPU, ensuring the efficient utilization of their computing resources.
- Our study addresses the challenges arising from variable-length inputs in some modalities, such as language tasks, complicating the allocation of PIM. To tackle this issue, we introduce a variable-length-aware PIM optimizer to enhance the efficiency of PIM processing.
- We extend the TVM [4] backend to include PIM command generation, facilitating the parallel initiation of PIM and GPU kernels. Our experimental results affirm that the collaborative PIM-GPU compiler we propose achieves an average 15x end-to-end acceleration.

2 BACKGROUND AND RELATED WORK

2.1 Multimodal Transformers

The goal of VLP is to improve performance on various visual language tasks by acquiring knowledge from multimodal base models. ViLBERT [5] and LXMERT [6] use dual encoders to independently encode visual and language features through different transformers, followed by a third transformer for multimodal fusion. There are also encoder-decoder architectures [7], [8], [9], and unified transformer architectures [10].

A multimodal vision-language model uses a computation architecture based on transformer layers that fuse information from visual and language modalities. As shown in Fig. 1, the architecture consists of some transformer layers, each of which typically consists of multi-head attention blocks (MHAB) and feed-forward blocks (FFB).

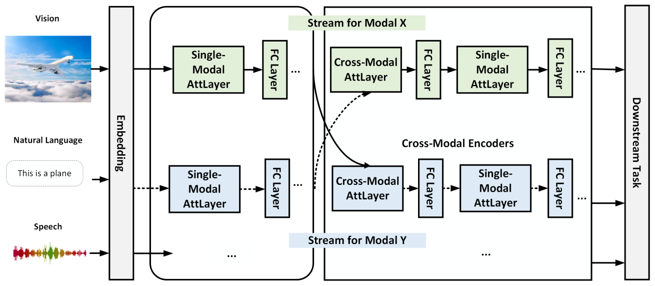

Figure 1: Multimodal transformer.

2.1.1 *Transformer Layers.*

Following the transformer architecture, a layer is defined by inputs X and Y as follows

$$\mathrm{MHAB}(X,Y) = \mathrm{LN}\big(X + \mathrm{MHA}(X,Y,Y)\big), \tag{1}$$

$$\mathrm{FFB}(M) = \mathrm{LN}\big(M + \mathrm{ReLU}(MW_1)W_2\big), \tag{2}$$

where LN is layer normalization, MHA is multi-head attention, M is an input matrix, and W_1, W_2 are learnable matrices.

A standard Transformer layer, performing self-attention, is denoted as

$$\mathrm{M}(X) = \mathrm{FFB}\big(\mathrm{MHAB}(X,X)\big). \tag{3}$$

2.1.2 *Dual-Stream Multimodal Transformer.*

Models such as ViLBERT introduce dual-stream multimodal transformers that include both cross-modal and intra-modal layers. The cross-modal transformer layer captures cross-modal interactions through the cross-modal attention module. Cross-modal multi-head attention is expressed as

$$M_{M\bar{M}} = \mathrm{MHAB}(X_M, X_{\bar{M}}). \tag{4}$$

Here, M represents the language or visual modality, and $\bar{M}$ denotes its complement.

The intra-modal transformer layer independently computes a transformer layer for each modality as

$$M_{MM} = \mathrm{MHAB}(X_M, X_M). \tag{5}$$

This architecture allows the model to jointly process and integrate information from both visual and language modalities through cross-modal attention mechanisms.

2.2 DRAM-PIM Architecture

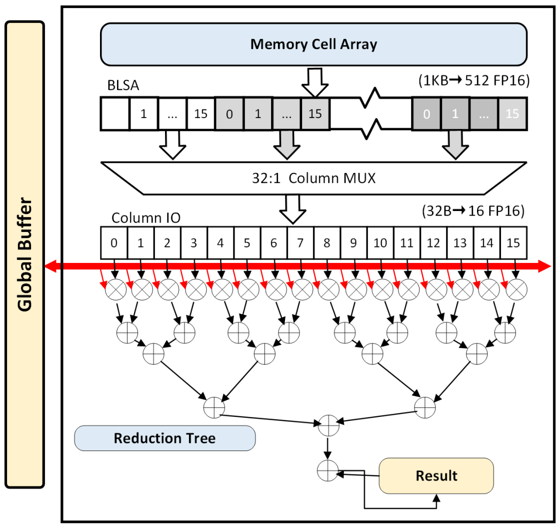

Figure 2: DRAM-PIM architecture.

DRAM-based PIM is an effective solution for accelerating machine learning models due to its significant memory capacity. To maximize the benefits of PIM, Newton [11] focuses specifically on memory-bound deep learning models such as recommendation systems (e.g. Facebook's DLRM [12]) and language models (e.g. Google's BERT [1]). Consequently, Newton introduces an efficient fixed-dataflow accelerator for computing matrix-vector multiplication. Considering area and power constraints, Newton integrates MAC units and buffers directly into commercial DRAM, avoiding the power overhead of previous approaches and making PIM feasible. Fig.2 illustrates the overall architecture of Newton within a single DRAM chip. Each group consists of 16 multipliers, 16 adders in the reduction tree, and a 16-bit accumulator register. The multiplier's two input operands are obtained from the column selector and the memory cell array after the global buffer broadcasts the

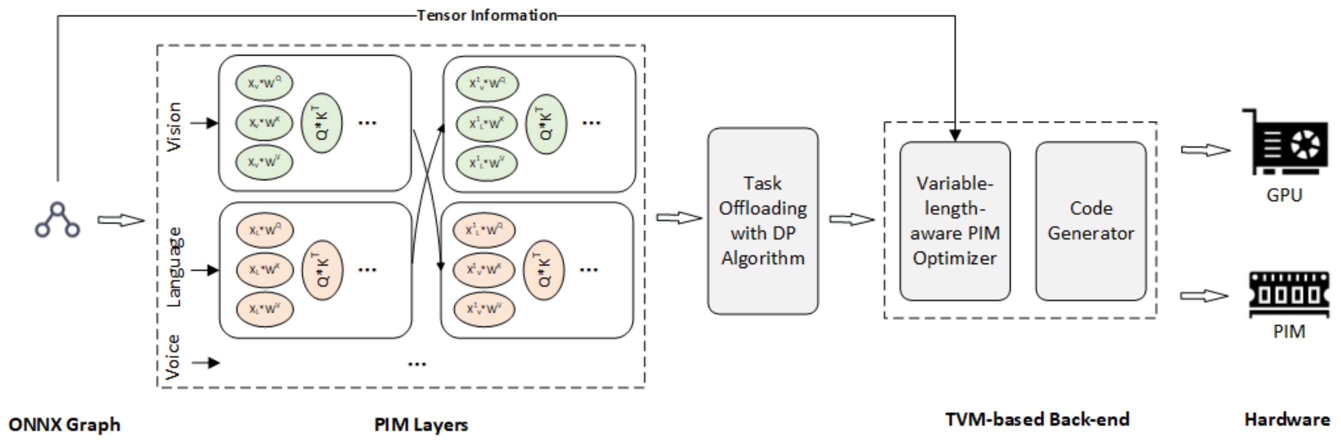

Figure 3: Overview of the compiler that collaboratively optimizes PIM and GPU for multimodal transformer.

input vector to all memory groups. In addition, Newton provides customized PIM commands such as GWRITE, GACT, COMP, and READRES to exploit memory parallelism and reduce latency.

Because of the significant memory usage and the disproportion between computation and memory, data movement emerges as a major contributor to the slow inference of multimodal transformers. The predominant performance bottleneck for multimodal transformer models is the bandwidth of memory [13]. Due to the extensive parameters of the multimodal transformer model, it is critical to decrease data movement without compromising computation efficiency. Therefore, an efficient design for multimodal model inference is essential to address this issue. PIM accelerators have emerged as a category of AI accelerators well suited to mitigate the memory wall problem by significantly reducing data movement between processors and memory.

3 COMPILER DESIGN

We develop a compiler that undergoes collaborative optimization for PIM and GPU, as shown in the Fig.3. The optimization comprises three primary components.

First, suitable nodes for PIM processing are identified through the ONNX graph. While offloading computation to PIM can reduce data movement, performing computation by PIM alone does not make efficient use of the GPU's computing power. Therefore, a dynamic programming algorithm is considered to offload different computation to PIM and GPU, separately. Second, a variable-length-aware PIM optimizer is introduced to address the problem of variable length of transformer inputs. This aims to achieve higher PIM utilization and reduce invalid occupation. Third, our compiler backend extends TVM to implement PIM command generation. We enable parallel initiation of PIM and GPU kernels, thus improving the overall computation efficiency of the collaboration system.

3.1 Task Offloading via Graph Transformation

We start by constructing the ONNX graph for the multimodal transformer. We analyze the computation graph to detect nodes suitable for PIM processing when multi-head attention layer (MHAL) and fully connected layer (FCL) are performed.

First, we derive matrix multiplication operations in MHAL and FCL of multimodal transformers by traversing the graph. Complex operations in transformers, such as "Softmax" and "Norm", are unsuitable for execution in PIM. These operations have excessive complexity and pose challenges to the PIM implementation. Therefore, we exclude them from our acceleration goals as the associated costs outweigh the benefits in terms of acceleration. Consequently, we choose to perform the complex computation on CPU or GPU. To improve the parallel computing efficiency of the PIM-GPU collaboration system, and fully exploit the computation capabilities of both PIM and GPU, we propose the strategy that offloads some matrix multiplication computation to GPU. Additionally, we investigate the strategy that offloads computation to both PIM and GPU in various multimodal transformer models.

By examining the model architecture of the multimodal transformer, we identify the sequence of "MatMul" and "GeMM" nodes that are conducive to PIM processing. When it comes to considering the computation offloading strategy between PIM and GPU, a direct method is to examine all possible allocation schemes to find an optimal solution (i.e., the strategy with minimal inference latency). Unfortunately, this method is impractical because the time complexity increases exponentially with the scale of the computation. Therefore, we present an efficient algorithm based on dynamic programming to solve the problem.

The dynamic programming algorithm is used, as shown in Algorithm 1, to solve the computation offloading problem between PIM and GPU. First, we traverse the ONNX graph of multimodal transformers and identify "MatMul" and "GeMM" nodes that are suitable for offloading to PIM. The computation is hierarchically classified based on its computation parallelism and placed in the same layer [Lines 1-4]. Then, we iterate through each PIM layer, aiming to offload as much computation as possible to PIM and GPU. $CpGPU_{all}$ represents all GPU computation resources, $BwGPU_{all}$ represents all GPU bandwidth resources, $CpGPU_i$ denotes the GPU computation resources required by the i-th task, and $BwGPU_i$ denotes the

GPU bandwidth resources required by the i-th task. Similar definitions apply to $CpPIM_{all}$, $BwPIM_{all}$, $CpPIM_i$ and $BwPIM_i$. Our goal is to maximize the number of tasks that can be offloaded to the existing PIM and GPU. Therefore, we use a dynamic programming algorithm to determine the optimal strategy for offloading the maximum number of tasks to PIM and GPU separately. [Lines 5-13].

Algorithm 1: Task Offloading with DP Algorithm

```
Input  : ONNX Graph $G$
Output : Maximum number of offloaded tasks for parallel_layers
1  for each node in G do
2      if node.op_type = MatMul or GeMM then
3          PIM_mark(G, node)
4  PIM_layers ← DevidePIMNodesToLayers(G)
5  for lay in PIM_layers do
6      $T$ ←lay.tasks.size()
7      for $i = 1$ to $T$ do
8          for $j = CpGPU_i$ to $CpGPU_{all}$ do
9              for $k = BwGPU_i$ to $BwGPU_{all}$ do
10                 for $l = CpPIM_i$ to $CpPIM_{all}$ do
11                     for $m = BwPIM_i$ to $BwPIM_{all}$ do
12                         $Task[i,j,k,l,m] \leftarrow \max(Task[i-1,j,k,l,m], Task[i, j-\mathrm{CpGPU}_i, k-\mathrm{BwGPU}_i, l-\mathrm{CpPIM}_i, m-\mathrm{BwPIM}_i]+1)$
13     parallel_tasks[lay] ← $Task\ [T, \mathrm{CpGPU_{all}}, \mathrm{BwGPU_{all}}, \mathrm{CpPIM_{all}}, \mathrm{BwPI_{all}}]$
14 return parallel_tasks
```

3.2 Optimizer for Variable-Length Inputs

For transformers with fixed-length inputs, the intermediate variables maintain a consistent dimension throughout the computation, allowing for pre-optimization. However, in the case of transformers with variable-length inputs, the variations in intermediate tensors require a customized PIM memory allocation strategy. This is essential to effectively manage the PIM computation associated with variable-length inputs.

To address the variable input scenario inherent in language transformers, we have developed a variable-length-aware PIM optimizer inspired by [2]. Our optimizer exploits information from inputs and the computation graph to achieve higher PIM utilization. We organize PIM memory units into blocks. Through our optimizer, we enable the reuse of allocated units, ensuring efficiency of PIM computing. Intermediate tensors were stored in banks of PIM according to our optimizer. Through the computation graph and the input length, the optimizer anticipates the life cycle of each tensor and calculates the offset in PIM units as new requests arrive at PIM.

Our algorithm, described in Algorithm 2, relies on the *tensor_info* input to represent the tensor information to be allocated. The *start* and *end* denote the initial and final operations used by the tensor, separately. *blocks* denotes a series of PIM blocks, specifying their sizes, PIM addresses, and allocated tensor information. The algorithm starts by arranging the tensors in descending order of size, prioritizing the allocation of larger tensors. Within the existing allocated PIM blocks, we try to find suitable positions for tensor allocation. Initially, we start by traversing the allocated items to identify the allocated tensor with a temporal intersection with t [Lines 7-9]. In front of overlapping items, we look for the smallest available PIM gap for allocation [Lines 9-14]. If no suitable gap is found in the block and the block can contain the tensor behind these items, we attempt to assign it behind these items [Lines 15-16]. If no suitable position is found in existing blocks, a new block consisting of this tensor is appended to the *blocks* collection. This algorithm ensures optimal use of PIM units and avoids wastage.

Algorithm 2: Variable-length-aware PIM Optimizer

```
Input  : tensor_infor : A collection of tuples representing (start, end, size), blocks : a block has size, PIM addr, list of <tensor_id, offset>
Output : The index of the inserted PIM tensor in the blocks pre_block and its position pre_pos within the block
1  Arrange the elements in tensor_infor in descending order of their sizes
2  foreach $t \in$ tensor_infor do
3      foreach block $\in$ blocks do
4          best_pos ← NIL
5          pre ← 0
6          foreach item $x \in$ block do
7              max_start ← $\max(\mathrm{start}_t, \mathrm{start}_x)$
8              min_end ← $\min(\mathrm{end}_t, \mathrm{end}_x)$
9              if max_start ≤ min_end then
10                 inter ← $\mathrm{offset}_x$ − prev
11                 if inter ≥ $size_t$ and inter < min_inter then
12                     min_inter ← inter
13                     best_pos ← prev
14                 prev ← $\max(prev, \mathrm{offset}_x + \mathrm{size}_x)$
15         if best_pos is NIL and block_size ≥ $size_t$ + prev then
16             best_pos ← prev
17         if best_pos not NIL then
18             pre_block ← block_id
19             pre_pos ← best_pos
20             is_set ← true
21             break
22     if is_set then
23         Add a new block to to blocks
24 return pre_block, pre_pos, blocks
```

Fig.4 illustrates an optimization example. The graph shows a language transformer with an input length of 200. The horizontal axis displays operations, including matrix multiplication operations

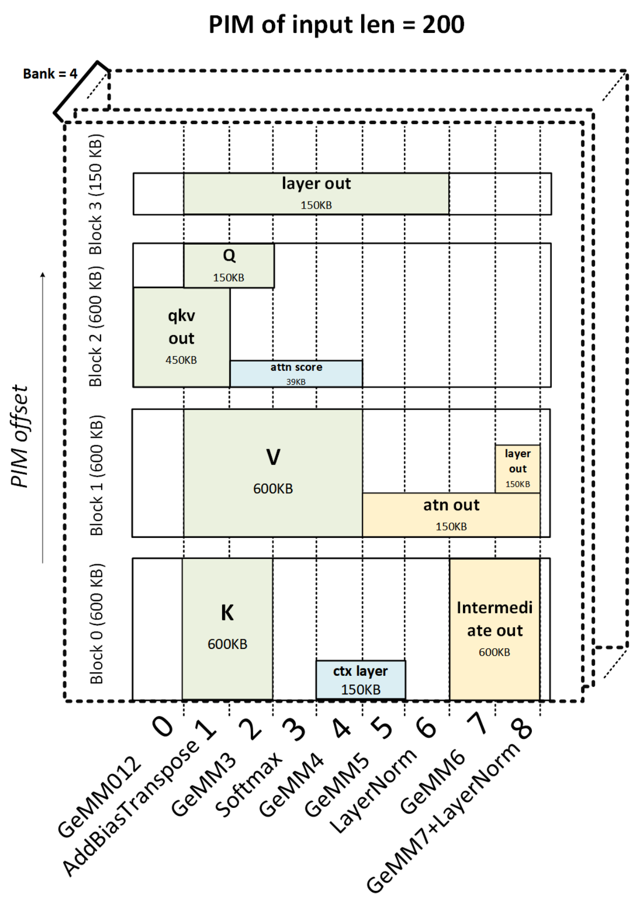

Figure 4: An example of PIM optimization via variable-length input optimizer.

performed in PIM. The vertical axis indicates the offset. As shown in the figure, our algorithm effectively leverages PIM memory units.

3.3 TVM-Based Compilation Backend

The GPU-PIM TVM-based backend is designed to take as input the model graph refined by task offloading strategy and the optimizer for variable-length inputs. It then generates PIM commands corresponding to nodes designated for offloading to PIM, while simultaneously executing PIM and GPU kernels. Prefixes are added to node names to enable the compiler to identify PIM_work nodes.

By extending the TVM compiler, parallel initiation of PIM and GPU kernels is enabled by exploiting the reuse of the mapping. The command generator consists of a scheduling phase that strategically maximizes the utilization of all PIM compute units.

4 EVALUATION

We implement the ONNX graph transformation process in the ONNX opset [14] and extended the TVM compiler version 0.8 [4] to support the TVM DRAM-PIM backend. The TVM DRAM-PIM backend is used to simulate the DRAM-PIM architecture using the extended Ramulator [15], and to generate and simulate GPU traces on the NVIDIA GeForce RTX 2060 GPU using Accel-Sim [16]. The timing parameters of HBM and PIM command latency are the same as [11]. We evaluated the inference time of 4 multimodal transformer models: ViLBERT-Base [5], ViLBERT-Large [5], BLIP-Base [17], and BLIP-Large [17].

This section illustrates the performance improvements achievable through various optimization methods on the PIM-GPU architecture to enable a multimodal transformer model. It also includes an analysis of PIM and GPU resource utilization and the effect of input length to investigate the feasibility and performance impact of our design choices. The specific offloading methods are outlined below:

- GPU-Only: Execution exclusively on the GPU.
- PIM-GPU: Offloading matrix multiplication computation entirely to PIM, with some other computation handled by the GPU.
- PIM-GPU-DP: By task assignment optimization using DP algorithm, collaboratively performing matrix multiplication computation on both PIM and GPU.
- PIM-GPU-DP-VLA: Building upon PIM-GPU-DP, incorporating the variable-length-aware PIM optimizer.

4.1 End-to-End Performance

Fig.5 illustrates the acceleration achieved by our approaches (PIM-GPU, PIM-GPU-DP, and PIM-GPU-DP-VLA) on ViLBERT-Base, ViLBERT-Large, BLIP-Base, and BLIP-Large. The end-to-end model inference times for all evaluated models are normalized to the GPU baseline.

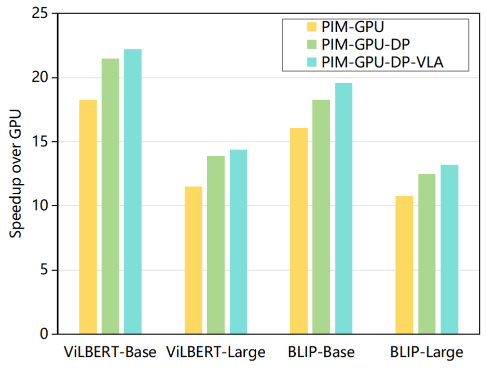

Figure 5: End-to-End speedup.

PIM-GPU represents the platform equipped with embedded DRAM-based PIM and GPU hardware, using the computation offloading approach where matrix multiplication computation is performed by PIM and some other computation is handled by GPU. The performance based on PIM-GPU outperforms the GPU-only method by effectively eliminating the overhead caused by data movement. However, the performance improvement achieved by the offloading approach of placing matrix multiplication computation to PIM alone is limited due to the imbalance in the utilization

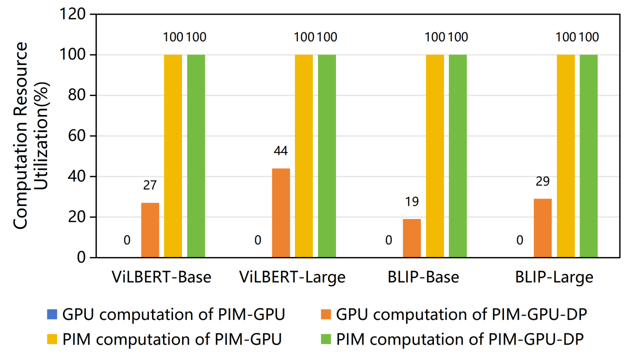

Figure 6: The computation resource usage.

between PIM and GPU computing resources during the matrix multiplication computation.

The PIM-GPU-DP design outperforms the PIM-GPU design, achieving an average inference performance improvement of 13.7% to 20.9%. This improvement is attributed to the dynamic programming task offloading method, which allows full utilization of the GPU and PIM computing resources. Additionally, PIM-GPU-DP-VLA outperforms all designs, achieving a performance improvement of 21.3% to 25.2%. This is because that we introduce the variable-length-aware PIM optimizer, which optimally utilizes PIM units for variable-length inputs. Evaluation results demonstrate that our approach can significantly speed up the inference of multimodal transformer models on PIM-GPU hardware without affecting the hardware itself.

4.2 Resource Utilization

Fig.6 shows the utilization of computation resources of PIM and GPU during matrix multiplication computation. When only the PIM-GPU offloading approach is used for matrix multiplication computation, the GPU resources remain idle at 0, while the PIM is over-utilized. By implementing offloading optimization using the DP algorithm, we achieve a relatively balanced use of PIM and GPU computation resources. Due to the limited GPU bandwidth, offloading as much computation as possible to the GPU core will still leave GPU computation resources underutilized.

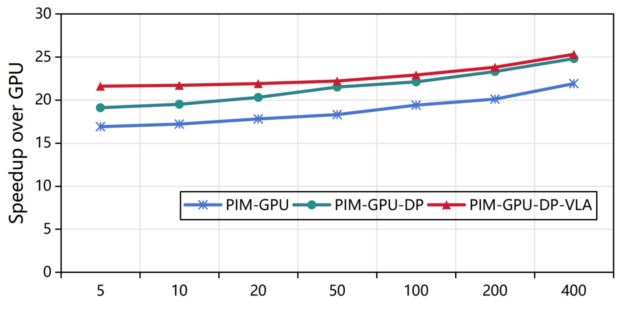

Figure 7: Speedup for different input lengths.

4.3 Comparison on Input of Variable Length

The ability to handle variable-length input is assessed by running with requests of different lengths. In the case of ViLBert-Base, input requests consist of randomly generated text with lengths distributed between 5 and 500, as indicated only by our specified set [10, 20, 50, 100, 200, 400]. The generation process is completely random, although for clarity the graph is presented in order of increasing input length. In Fig7, with ViLBert-Base inference, PIM-GPU-DP-VLA accelerates PIM-GPU by 15.5% to 27.8%. The performance gains are more significant for test cases with shorter input sequences.

5 CONCLUSION

In summary, our work introduces a collaborative optimization strategy for multimodal transformers to accelerate the PIM-GPU collaboration computing architecture. The developed compiler employs a dynamic programming algorithm and a variable-length-aware PIM optimizer to strategically offload operations between PIM and GPU. Through the extension of the TVM backend, we enable the parallel initiation of PIM and GPU kernels. The PIM-GPU collaborative compiler we proposed achieves an average 15x end-to-end speedup, as demonstrated by experimental results. This research provides valuable insights into the seamlessly integrating PIM and GPU for efficient processing of multimodal transformers.

ACKNOWLEDGMENTS

This work was supported in part by the National Key Research and Development Program of China (No. 2021YFB0300300), in part by the Programs of National Natural Science Foundation of China (Nos. 62072165, 62202154, and 62172151.), and in part by the Natural Science Foundation of Hunan Province of China (Nos. 2021JJ10023).

REFERENCES

[1] Devlin et al., "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.
[2] J. Fang et al., "Turbotransformers: an efficient gpu serving system for transformer models," in PPoPP, 2021, pp. 389–402.
[3] Y. Ding et al., "HAIMA: A Hybrid SRAM and DRAM Accelerator-in-Memory Architecture for Transformer," in DAC, 2023, pp. 1–6.
[4] T. Chen et al., "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in OSDI, 2018, pp. 578–594.
[5] J. Lu et al., "Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks," Advances in neural information processing systems, vol. 32, 2019.
[6] H. Tan et al., "Lxmert: Learning cross-modality encoder representations from transformers," arXiv preprint arXiv:1908.07490, 2019.
[7] J. Cho et al., "Unifying vision-and-language tasks via text generation," ICML, 2021, pp. 1931–1942.
[8] Z. Wang et al., "Simvlm: Simple visual language model pretraining with weak supervision," arXiv preprint arXiv:2108.10904, 2021.
[9] X. Chen et al., "Pali: A jointly-scaled multilingual language-image model," arXiv preprint arXiv:2209.06794, 2022.
[10] W. Wang et al., "Image as a foreign language: Beit pretraining for all vision and vision-language tasks," arXiv preprint arXiv:2208.10442, 2022.
[11] M. He et al., "Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning," in MICRO, 2020, pp. 372–385.
[12] M. Naumov et al., "Deep learning recommendation model for personalization and recommendation systems," arXiv preprint arXiv:1906.00091, 2019.
[13] M. Zhou et al., "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in HPCA, 2022, pp. 1071–1085.
[14] (2019) Open neural network exchange. [Online]. Available: https://onnx.ai/
[15] Y. Kim et al., "Ramulator: A fast and extensible dram simulator," IEEE Computer architecture letters, vol. 15, no. 1, pp. 45–49, 2015.
[16] M. Khairy et al., "Accel-sim: An extensible simulation framework for validated gpu modeling," in ISCA, 2020, pp. 473–486.
[17] J. Li et al., "Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation," in ICML, 2022, pp. 12888–12900.