Hermes: Memory-Efficient Pipeline Inference for Large Models on Edge Devices

Xueyuan Han[1], Zinuo Cai[2], Yichu Zhang[3], Chongxin Fan[4], Junhan Liu[2], Ruhui Ma[2], Rajkumar Buyya[5]
[1] ParisTech Elite Institute of Technology, Shanghai Jiao Tong University, Shanghai, China
[2] School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
[3] UM-SJTU Joint Institute, Shanghai Jiao Tong University, Shanghai, China
[4] Shanghai Aerospace System Engineering Institute, Shanghai, China
[5] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, The University of Melbourne, Melbourne, Australia
{hxy771126, kingczn1314, 1654468697}@sjtu.edu.cn, fcs-841220@163.com,
{liujunhan, ruhuima}@sjtu.edu.cn, rbuyya@unimelb.edu.au

***Abstract*—The application of Transformer-based large models has achieved numerous success in recent years. However, the exponential growth in the parameters of large models introduces formidable memory challenge for edge deployment. Prior works to address this challenge mainly focus on optimizing the model structure and adopting memory swapping methods. However, the former reduces the inference accuracy, and the latter raises the inference latency. This paper introduces PIPELOAD, a novel memory-efficient pipeline execution mechanism. It reduces memory usage by incorporating dynamic memory management and minimizes inference latency by employing parallel model loading. Based on PIPELOAD mechanism, we present Hermes, a framework optimized for large model inference on edge devices. We evaluate Hermes on Transformer-based models of different sizes. Our experiments illustrate that Hermes achieves up to $4.24\times$ increase in inference speed and 86.7% lower memory consumption than the state-of-the-art pipeline mechanism for BERT and ViT models, $2.58\times$ increase in inference speed and 90.3% lower memory consumption for GPT-style models.**

***Index Terms*—Edge computing, Memory optimisation, Large model inference, Pipeline execution.**

I. INTRODUCTION

The Transformer architecture has profoundly transformed the landscape of deep learning and brought forward large models with their applications spreading from data centers [1] to edge devices. Large models are generally categorized into Natural Language Processing (NLP), Computer Vision (CV), and Multimodal models. NLP is widely applied on mobile devices [2], [3], from intelligent personal assistants, like Google Assistant and Apple Siri to real-time language translation [4]. CV plays a pivotal role in the field of autonomous driving [5], [6], where it is utilized for tasks such as real-time object detection [7], [8], lane recognition [9], and traffic signal detection [10]. By enriching robots' perception and decision-making capabilities through the integration of diverse data types [11], such as visual, auditory [12], and tactile [13] information, Multimodal large models are revolutionizing the field of robotics [14], [15].

This work was supported by Shanghai Key Laboratory of Scalable Computing and Systems, National Key Laboratory of Ship Structural Safety, and the Eighth Research Institute of China Aerospace Science and Technology Group Company, Ltd., under Grant USCAST2023-17 and Grant USCAST2023-21. (Corresponding author: Ruhui Ma.)

Due to the explosive growth in the size of large models, deploying them at the edge faces critical memory challenges [16]. Specifically, current edge devices offer only a limited amount of memory capacity, ranging from a few tens of megabytes to a few gigabytes. For example, NVIDIA Jetson Nano has 4 GB of memory and Raspberry Pi 4 Model B has up to 8 GB of memory. In contrast, large models' parameters have experienced exponential growth, reaching sizes in the hundreds of billions. For instance, the GPT-3 [17] model has 175 billion trainable parameters, while the recently developed GPT-4 [18] model exceeds the trillion parameter mark. Consequently, the memory usage of these large models can easily reach tens to hundreds of gigabytes, far surpassing the memory capacity of typical edge devices.

Existing works to address the memory challenges of large model inference on edge devices can be classified into two categories. The first attempts to optimize the model structure to reduce the computational load through techniques including model pruning [19]–[21], model compression [22], [23], model quantization [24] and adaptive inference [25], [26]. Although these approaches significantly diminish the number of required computational operations, they often result in reduced model accuracy. Moreover, these approaches are generally tailored for specific models, thus limiting their applicability on a broader scale. The second optimizes the memory usage during model inference by model swapping between different storage media [27]–[29]. This method initially divides the model into separate shards and selectively preloads certain shards from disk into a buffer or the memory as needed for the inference process. Even though memory swapping methods can reduce memory overhead, they can inadvertently prolong inference latency due to the increased frequency of I/O operations between varying storage media.

In this paper, we envisage pipelining the model loading and inference process, hiding the latency of the loading by overlapping it with the inference. Given the ubiquity of CPUs in edge devices, the loading process involves loading model weights from disk to memory, and the inference process refers to performing inference on CPUs. Fig. 1a illustrates the standard pipeline design, which loads model weights from

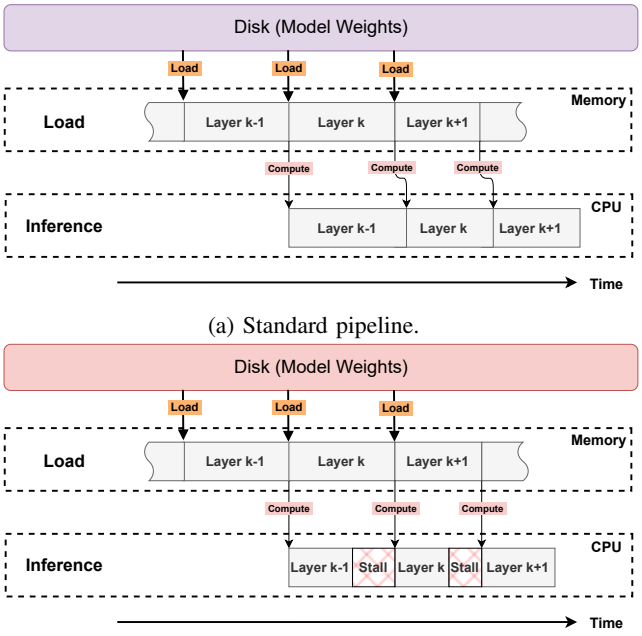
disk to memory at a layer granularity. A transformer layer consists of the multi-head self-attention and the position-wise feed-forward network. Given that the transformer model is comprised of sequential layers, it performs the loading and inference process layer-by-layer. Inference in memory begins immediately following the loading of the initial layer from disk, with the subsequent layer being loaded concurrently. The standard pipeline formed by this process allows inference to commence prior to the complete model being loaded, thus reducing the latency.

Although we are not the first to apply pipeline to large model inference on edge devices, we attempt to solve two challenges that have not been addressed by existing works, such as PipeEdge [30] and PipeSwitch [31]. The first challenge is that pipeline schemes do not reduce the memory requirements of inference. For instance, PipeEdge employs pipeline parallelism, leveraging under-utilised or idle distributed edge resources to enhance inference performance across diverse edge devices. Although this approach enhances inference speed, it lacks memory optimization and does not reduce the memory requirements for inference. The second challenge is that the deployment of pipeline on edge may lead to serious pipeline stalls, because of the huge gap between the model loading and execution latency. Our experiments in §II-B demonstrate that the loading latency is generally an order of magnitude larger than the inference latency, leading to the pipeline stall issue illustrated in Fig. 1b.

To resolve two issues mentioned above, we develop PIPELOAD, a memory-efficient pipeline execution mechanism to streamline the loading and inference process with per-layer granularity. This mechanism incorporates dynamic management of memory with timely destruction of model weights that have been inferred, significantly reducing the memory usage of model inference. And by engaging parallel model loading, PIPELOAD overlaps multiple inference time within a single loading interval to minimise the pipeline stalls, consequently decreasing inference latency.

Building upon this mechanism, we introduce **Hermes**, an innovative framework for large model inference on edge devices. This framework integrates three main components. Firstly, the ***Layer Profiler*** evaluates the performance and memory utilization of each individual model layer for a given transformer model. Next, the ***Pipeline Planner*** utilizes the profiling data from ***Layer Profiler*** to devise a PIPELOAD execution schedule within different memory constraints. Lastly, the ***Execution Engine*** determines the execution strategy from the schedule based on the current memory constraints of the edge device and executes PIPELOAD inference.

We conduct experiments with four transformer-based models, including BERT-Large, GPT-2-Base, ViT-Large and GPT-J, on our CPU cluster server. These four models vary in size from a few hundred megabytes to a dozen gigabytes. Through a comprehensive performance and memory footprint evaluation, **Hermes** achieves up to $4.24\times$ increase in inference speed and 86.7% lower memory footprint than PipeSwitch for BERT and ViT models, $2.58\times$ increase in inference speed and 90.3% lower memory footprint for GPT-style models. We also evaluate **Hermes** under different memory constraints and it works well in all test environments with all results meeting service level objective (SLO) expectations.



(a) Standard pipeline.

(b) Pipeline stalls.

Fig. 1: Standard pipeline design and pipeline stall problem.

Our contributions are highlighted as follows.

- We propose PIPELOAD, a memory-efficient pipeline mechanism designed to reduce the memory footprint and latency during model inference.
- We present **Hermes**, a framework based on the PIPELOAD to mitigate memory usage and pipeline stall for large model inference on edge devices.
- We implement a system prototype of **Hermes** and evaluate it on several transformer models. Experiments show that our method achieves a $4.24\times$ speedup while reducing memory footprints by 90.3%.

II. BACKGROUND AND MOTIVATION

A. Transformer Model Structure

The architectural makeup of transformer models is the basis for developing pipeline inference strategies on edge devices. A typical transformer model comprises embedding, encoder, decoder, pooling, and additional specialized layers. Based on their architectural configurations, they can be classified into three primary categories: encoder-decoder, encoder-only and decoder-only models. Encoder-decoder models, such as BART [32] and T5 [33] integrate both encoder and decoder layers. BART is architected for complex sequence-to-sequence tasks, and T5 generalizes this capability with a comprehensive text-to-text methodology applicable to a wide array of NLP challenges. Encoder-only models like BERT [34] employ a series of encoder layers to interpret input data, ideally suited

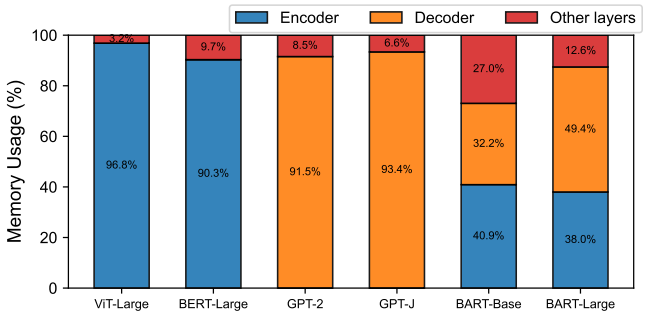


Fig. 2: Decomposition of layers' memory usage.

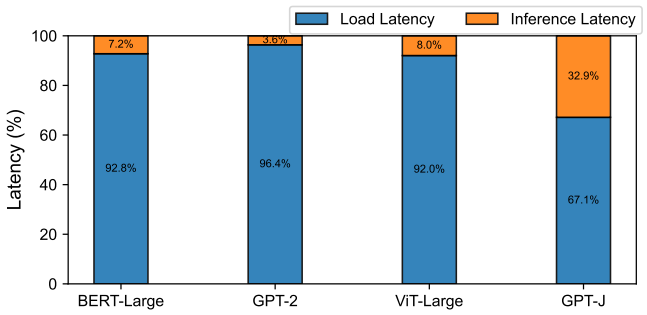


Fig. 3: Decomposition of loading and inference latency.

for tasks that do not involve generating sequences. Meanwhile, Vision Transformer (ViT) presents an innovative adaptation of encoder-only architecture to analyze sequences of image patches, eschewing the conventional decoder layout found in text-centric models. Conversely, decoder-only models such as GPT rely exclusively on decoder layers, focusing on the generation of new content by drawing on recognizable patterns.

B. Characteristics of Transformer-based Models

In order to design an efficient pipeline scheme, we conduct experiments to characterize two key aspects when the model performs forward computation: the allocation of memory among transformer model layers and the latency of model loading and model inference. We first evaluate the memory allocation in five kinds of transformer models, including ViT-Large, BERT-Large, GPT-2, GPT-J and BART (BART-Base and BART-Large), which cover all three categories of Transformer models. Additionally, we evaluate the time requirements of loading and inference for various transformer models, including BERT-Large, GPT-2, ViT-Large and GPT-J, by performing standard model inference. All the experiments are conducted on Intel(R) 193 Xeon(R) Gold 6248R CPU.

TestCase1: Memory distribution. To understand the allocation of memory across layers, we conduct experiments with five kinds of transformer models. Typically, transformer-based models are characterized by their extensive reliance on attention mechanisms, necessitating substantial memory to accommodate attention scores and intermediate representations, particularly within encoder or decoder layers. Fig. 2 delineates the memory usage distribution across different layers for five kinds of prevalent transformer variants, revealing that encoder or decoder layers predominate, consuming between 70% to 95% of the total memory. Notably, the memory consumption attributed to these layers escalates with the model's overall size. For instance, BART-Large necessitates approximately 14.4% more memory relative to BART-Base.

Observation I

> *For general transformer-based models, the encoder or decoder layers occupy the largest memory footprint.*

TestCase2: Latency evaluation. To evaluate the latency of model loading and inference, we run standard model inference processes for four transformer models on CPU. Generally, transformer models exhibit considerably higher latency during layer loading compared to layer inference. Through our experiments, as depicted in Fig. 3, we observe that, for the first three smaller models (each with a memory footprint around 1 GB), the layer loading period substantially exceeds the inference time, by roughly an order of magnitude. Conversely, for the larger GPT-J model (12 GB), the layer loading duration is approximately twice that of the inference time. Subsequently, such disparities contribute to a significant portion of the computational process, between 60% to 80%, being spent idle during typical pipeline execution, underlining a serious pipeline stall issue, as shown in Fig. 1b.

Observation II

> *For general transformer-based models, loading latency is much larger than the inference latency, resulting in the execution process being stalled during most of inference time.*

C. Implications

Our experiments in §II-B analyze the time distribution and memory usage of transformer-based large models during model inference. **Observation I** suggests that a targeted focus on either the encoder or decoder layers is pivotal for optimizing memory management in our pipeline infrastructure. **Observation II** underscores the necessity of adopting a parallel loading strategy by overlapping multiple inference times with a single loading time within our pipeline scheme, to efficiently mitigate pipeline stalls. In summary, we progress our design by addressing the following challenges: (1) memory challenge on edge devices; (2) pipeline stall problem caused by the huge gap between loading and inference latency.

III. PIPELOAD: A MEMORY-EFFICIENT PIPELINE EXECUTION MECHANISM

A. Overview

We present PIPELOAD, a memory-efficient pipeline execution mechanism to reduce memory footprint and latency during model inference on edge devices. There are three core workers in PIPELOAD mechanism: multiple *Loading Agents*, one *Inference Agent* and one *Daemon Agent*. *Loading Agents*

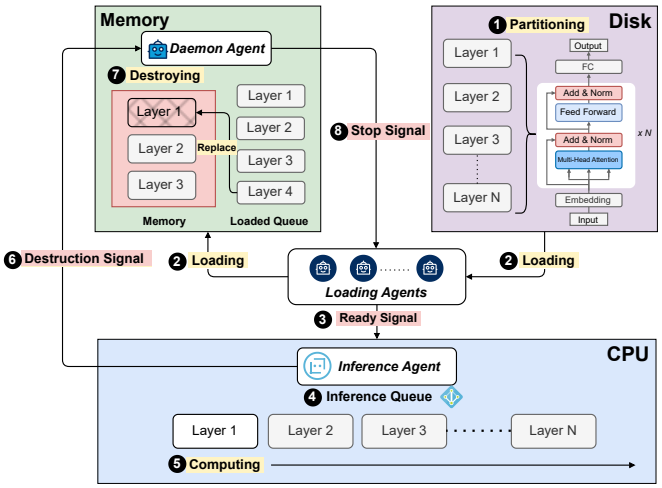


Fig. 4: The overview and workflow of PIPELOAD.

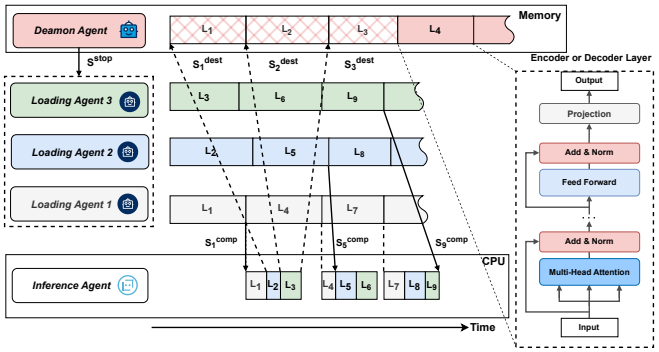


Fig. 5: PIPELOAD with three *Loading Agents*.

work in parallel to load model layers from disk to memory, reducing inference latency. The *Inference Agent* simultaneously executes computations on these loaded layers sequentially in CPU, guaranteeing the model's predictive accuracy and minimizing pipeline stalls. The *Daemon Agent* maintains a queue of loaded layers in memory, detects memory usage and destroys memory space for specific layers at a specific point to reduce memory overhead. Three workers communicate with each other through a signalling mechanism that facilitates the realization of whole memory-efficient pipeline.

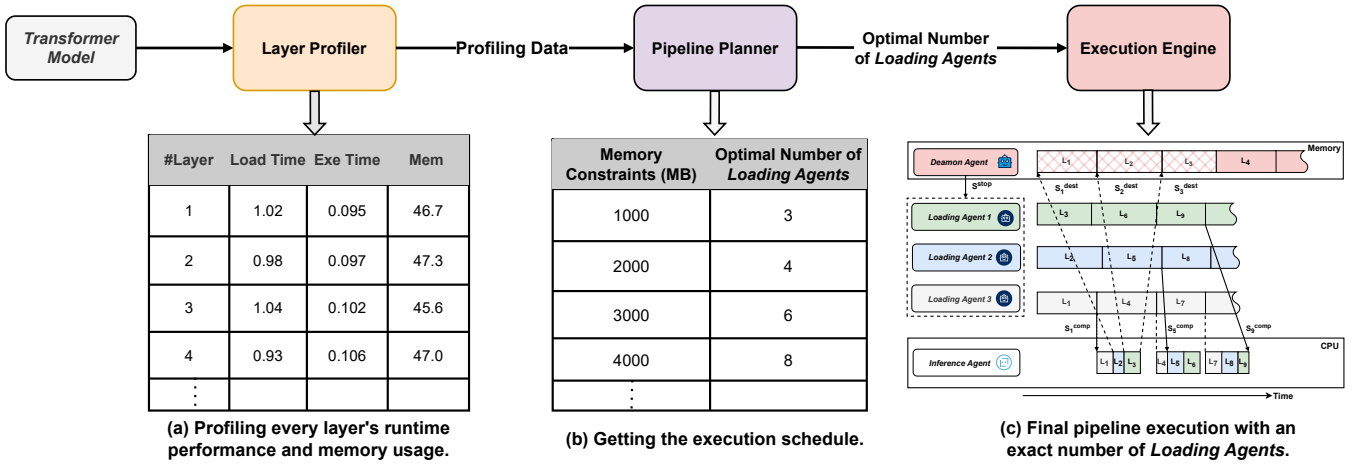
Fig. 4 illustrates the overall workflow of PIPELOAD, including the loading and inference process of model layers as well as the signaling mechanism. Before performing pipeline inference, we ❶ adopt a layer-based model partitioning scheme to pre-process the model weights. Multiple *Loading Agents* constantly ❷ load specific layers from disk into memory in parallel. Once a model layer is successfully loaded, the corresponding *Loading Agent* ❸ transmits a computation ready signal that corresponds to this layer to *Inference Agent*, indicating that this layer is ready for computation. *Inference Agent* ❹ maintains an inference queue in CPU that decides which layer will be processed next, ensuring that model inference respects the original sequence of layers. Upon receiving the computation ready signal, *Inference Agent* ❺ performs forward computation only if all preceding layers have been computed. Following computation of the layer, *Inference Agent* ❻ issues a memory destruction signal to *Daemon Agent*, notifying it to destroy the memory space of the layer. *Daemon Agent* then ❼ destroys the memory space occupied by the layer to reserve enough space for other layers. When memory usage is about to exceed or has exceeded the memory constraints of the edge device, *Daemon Agent* ❽ sends a stop signal to all *Loading Agents*, pausing their loading operations until sufficient memory space is available.

B. Case Study

Fig. 5 presents a simple case of PIPELOAD with three *Loading Agents*. Based on the characteristics of transformer model layers, we adopt a layer-based model partitioning scheme. In our scheme, we methodically segment the general transformer model architecture into its constituent layers: embedding layers, encoder layers, decoder layers and other layers. Among these layers, we focus only on the encoder and decoder layers that occupy most of the model weights in PIPELOAD mechanism design.

For simplicity, we show only three computation ready signals and three memory destruction signals in Fig. 5. And for the sake of clarity, three *Loading Agents* are symbolized as LA_1, LA_2, LA_3. Model layers are signified as L_k where k represents the index within the total number of layers, denoted by N. Symbols S_k^{comp}, S_k^{dest} and S^{stop} respectively represent computation ready signal for layer L_k, memory destruction signal for layer L_k and loading stop signal. During the implementation of PIPELOAD, the i-th *Loading Agent* is assigned a subset of model layers, following the distribution L_{i+jm}, where i ranges from 1 to m, with m representing the total number of *Loading Agents*, and j represents an iterative index, ranging from 0 to $\lfloor (N-i)/m \rfloor$ ($i + jm \leq N$ and $j \in \mathbb{N}$). In this case, LA_1 is responsible for layers $(L_1, L_4, L_7, \ldots)$, LA_2 for layers $(L_2, L_5, L_8, \ldots)$ and LA_3 for layers $(L_3, L_6, L_9, \ldots)$. This layer allocation method is designed to minimize pipeline stalls since we can overlap the inference time of three layers with the loading time of a single layer.

As shown in Fig. 5, the three *Loading Agents* commence the parallel loading process. As the layer L_1 is fully loaded to memory, LA_1 issues the computation ready signal, S_1^{comp} to *Inference Agent*. After receiving S_1^{comp}, *Inference Agent* starts to perform forward computation for L_1. If *Inference Agent* receives S_2^{comp} or S_3^{comp} first, the inference queue in CPU will ensure that the layers are computed in the correct order. Simultaneously, after LA_2 and LA_3 load L_2 and L_3 as well as sending computation ready signals to *Inference Agent*, the three *Loading Agents* are able to continue loading the respective next layers L_4, L_5, L_6 into memory. Following the computation on L_1, *Inference Agent* issues the memory destruction signal, S_1^{dest} to *Daemon Agent*, which initiates the process of de-allocating memory space for L_1. The loading and inference process for the subsequent layers is similar.



(a) Profiling every layer's runtime performance and memory usage.

(b) Getting the execution schedule.

(c) Final pipeline execution with an exact number of *Loading Agents*.

Fig. 6: **Hermes** system architecture.

In addition, when *Daemon Agent* detects the system memory usage has exceeded the memory constraints of the edge device, it sends the loading stop signal, S^{stop} to all *Loading Agents*, pausing their layers' loading operation until sufficient memory space is available.

IV. HERMES: A FRAMEWORK TO OPTIMIZE LARGE MODEL INFERENCE ON EDGE DEVICES

Fig. 6 presents **Hermes** system architecture, a comprehensive framework designed to enhance the performance and reduce memory usage of large model inference in edge computing environments. Specific modules within **Hermes** comprise ***Layer Profiler***, ***Pipeline Planner***, and ***Execution Engine***. This framework encapsulates methodologies for evaluating layer efficiency, deploying an optimal execution schedule, executing the memory-efficient pipeline, PIPELOAD and aims to collaborate diverse elements essential for optimizing model execution in resource-constrained settings, such as memory usage, latency and execution strategy.

1) ***Layer Profiler****:* Fig. 6a presents some possible results of ***Layer Profiler***, which serves as the foundation of our system architecture. The ***Layer Profiler***'s primary function is to profile each layer within a given transformer model to gauge runtime performance and memory usage. Through a pre-run of standard model inference, this profiling enables the accurate measurement of loading time, computation time and memory size for every individual layer of the given model.

2) ***Pipeline Planner****:* Utilizing the data generated by the ***Layer Profiler***, the ***Pipeline Planner*** develops a PIPELOAD execution schedule that includes several optimal execution strategies under different memory constraints, as shown in Fig. 6b. Firstly, drawing from the profiling insights encompassing layer's memory footprint along with layer's load and compute duration for the given model, the planner determines a reasonable range for the number of *Loading Agents* in conjunction with different memory constraints. In general, more *Loading Agents* means fewer pipeline stages, *i.e.*, less latency, but more encoder or decoder layers are reserved in memory, *i.e.*, more memory overhead. Next, the planner pre-runs the PIPELOAD within the range of the number of *Loading Agents* to obtain the exact number of *Loading Agents* under different memory constraints and finally outputs the execution schedule.

3) ***Execution Engine****:* Finally, upon establishing the execution schedule, the inference of PIPELOAD with an exact number of *Loading Agents* will be executed in the ***Execution Engine*** based on the current memory constraints of edge device, as shown in Fig. 6c. This includes actual pipeline inference execution facilitated by the specific number of *Loading Agents*, one *Inference Agent*, one *Daemon Agent* and signalling mechanism.

V. EVALUATION

A. Experimental Setup

1) Workloads: For estimating the memory-efficient pipeline execution mechanism, PIPELOAD, we focus on a quartet of transformer models: *i*) one NLP model: BERT-Large; *ii*) one CV model: ViT-Large; *iii*) and two generative text language models: GPT-2-Base and GPT-J. These four transformer models have different sizes, from a few hundred megabytes to a dozen gigabytes. Each of their configurations is shown in TABLE I, where the number of layers is the number of encoder or decoder layers, excluding other layers such as embedding layers and pooling layers, memory (layers / total) indicates that the memory footprint of encoder or decoder layers accounts for the total memory of the model and memory per layer represents the average memory footprint per encoder or decoder layer.

2) Baselines: In performance and memory usage evaluation, we focus on evaluating four transformer models mentioned above in ***Execution Engine***. And the engine provides three distinct operational modes: Baseline (non-pipeline), PipeSwitch, and our designed PIPELOAD with optional *Loading Agents*. In particular, the workflow of Baseline is the normal process of loading model first and then inferring it, and the workflow of PipeSwitch is basically the same as the standard pipeline.

TABLE I: Model Configurations.

Model	Parameters Size (Millions)	Types of Layers	Number of Layers	Data Type	Memory (Layers/Total) (MB)	Memory per Layer (MB)
ViT-Large	304	encoder	24	FP16	582 / 601	25
GPT-2-Base	355	decoder	24	FP32	1223 / 1433	51
BERT-Large	340	encoder	24	FP32	1317 / 1627	55
GPT-J	6000	decoder	28	FP32	11535 / 12354	412

TABLE II: Performance comparison.

Model	Baseline	PipeSwitch		PIPELOAD with 2 *LAs*		PIPELOAD with 4 *LAs*		PIPELOAD with 6 *LAs*	
	Latency (ms)	Latency (ms)	Speedup	Latency (ms)	Speedup	Latency (ms)	Speedup	Latency (ms)	Speedup
BERT-Large	15891.5	14897.1	1.067	7720.8	2.058	4621.8	3.438	3510.7	**4.527**
GPT-2-Base	1659.5	2457.9	0.675	1704.7	0.974	1396.1	1.189	1121.4	**1.480**
ViT-Large	345.0	157.3	2.193	90.8	3.799	56.8	6.070	43.2	**7.978**
GPT-J	31330.9	76494.6	0.410	51003.3	0.614	33487.2	0.936	29640.9	**1.057**

TABLE III: Memory footprints comparison.

Model	Baseline	PipeSwitch		PIPELOAD with 2 *LAs*		PIPELOAD with 4 *LAs*		PIPELOAD with 6 *LAs*	
	Memory footprint (MB)	Memory footprint (MB)	Ratio	Memory footprint (MB)	Ratio	Memory footprint (MB)	Ratio	Memory footprint (MB)	Ratio
BERT-Large	1627.3	1689.2	1.038	457.1	**0.281**	661.5	0.407	930.8	0.572
GPT-2-Base	1433.8	1436.8	1.002	387.5	**0.270**	518.8	0.362	649.9	0.453
ViT-Large	600.9	626.6	1.043	60.8	**0.101**	110.2	0.183	159.4	0.265
GPT-J	12354.0	12468.6	1.009	1668.6	**0.135**	2455.4	0.199	3242.2	0.262

3) Metrics: We use two performance metrics, latency and memory footprints. In the context of BERT and ViT models, latency is defined as the end-to-end time taken to generate an output with single inference; for GPT-style models, latency defines as the end-to-end output generation time for a given input prompts and a given output tokens length. Memory footprints is quantified as the maximum memory occupation by the model throughout its execution lifecycle.

4) Testbeds: We conduct our experiments on a server consisting of Intel(R) Xeon(R) Gold 6248R CPUs. We deploy a controlled and consistent environment with `docker` that imposes limits on resource usage, including limiting the number of CPU cores to a maximum of 8 and restricting memory size through `docker --memory` command, to simulate resource-constrained scenarios on edge devices.

B. Evaluation of Performance and Memory Footprints

We evaluate the performance and memory footprints of PIPELOAD with 2, 4 and 6 *Loading Agents* and compare them to baseline and to PipeSwitch. We choose these three numbers of *Loading Agents* since they are essentially factors of the number of encoder or decoder layers in four transformer models. The performance and memory footprints test results are shown in TABLE II and TABLE III respectively, where *LAs* is an acronym for *Loading Agents*. In order to show the optimisation results more directly, we add two metrics in tables respectively, the speedup and the ratio, with their expressions are as follows:

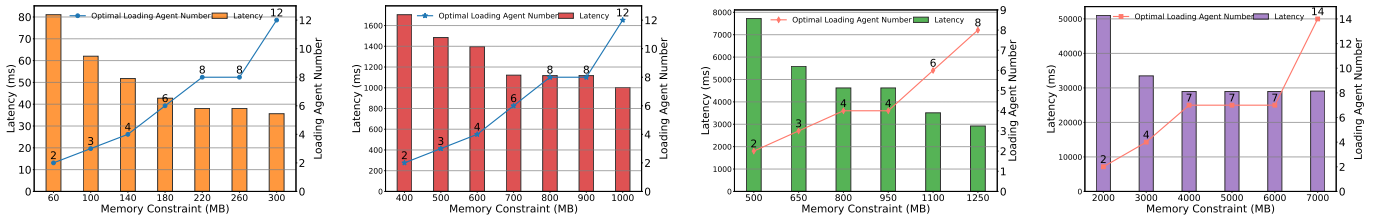
$$\text{Speedup} = \frac{T_{\text{baseline}}}{T_{\text{others}}}$$

$$\text{Ratio} = \frac{M_{\text{others}}}{M_{\text{baseline}}}$$

where T_{baseline} and T_{others} represent the latency of baseline and latency of other methods and M_{others} and M_{baseline} indicate the memory consumption of other methods and memory consumption of baseline.

1) BERT and ViT Models Analysis: For BERT and ViT models, we evaluate them with a single inference since they can generate outputs through loading and inference in a single pass. According to TABLE II and TABLE III, PIPELOAD with multiple *Loading Agents* indicates a promising trend of decreasing memory usage and latency compared to the PipeSwitch implementation. For BERT-Large, the speedup improvement is $1.93 \sim 4.24\times$ and the memory footprint reduction is $44.9\% \sim 73.0\%$. For ViT-Large, the speedup improvement is $1.73 \sim 3.64\times$ and the memory footprint reduction is $74.0\% \sim 86.7\%$. The smaller proportion of memory footprint reduction for BERT model compared to the ViT model is mainly due to the fact that the embedding and pooling layers of BERT-Large have a much larger portion, about 20% while ViT-Large about 1.5%. As we increment the number of *Loading Agents*, the speedup is also significantly increasing while the degree of memory footprint reduction is slowly decreasing. This result is also as expected, since more *Loading Agents* means less pipeline stages and more encoder or decoder layers saved in memory. Specifically, adding one *Loading Agent* implies one additional layer saved in memory.

2) GPT-2 and GPT-J Models Analysis: For the two GPT-style transformer models, we evaluate them for a given input prompts (number of tokens = 4) and for a given output tokens (length of tokens = 8). As shown in TABLE II and



(a) ViT-Large model. (b) GPT-2-Base model. (c) BERT-Large model. (d) GPT-J model.

Fig. 7: Models evaluation under different memory constraints.

TABLE III, PIPELOAD with multiple *Loading Agents* produces excellent results in reducing memory usage and good results in decreasing latency compared to the PipeSwitch. For GPT-2-Base, the speedup improvement is $1.44 \sim 2.20\times$ and the memory footprint reduction is $13.7\% \sim 72.9\%$. For GPT-J, the speedup improvement is $1.50 \sim 2.58\times$ and the memory footprint reduction is $74.6\% \sim 90.3\%$. PIPELOAD works better in GPT-J slightly better because its proportion of decoder layers' size to the total model size is higher. And for the same reasons as the previous two models, with the number of *Loading Agents* growing, the speedup is increasing while the degree of memory footprint reduction is slowly decreasing. However, their improvements in execution speed are less effective in comparison to the baseline when the number of *Loading Agents* is low (≤ 4). This is because the GPT-style transformer model loads memory only once for non-pipeline execution but performs inference multiple times (one inference for each token) while PIPELOAD and other pipeline methods require one loading and inference operation for each token, thus increasing latency when there are a large quantity of tokens.

C. Evaluation under different Memory Constraints

We evaluate the performance of **Hermes** under different memory constraints. In addition, we measure the latency and the corresponding optimal number of *Loading Agents*.

1) ViT and BERT Models Analysis: Fig. 7a and Fig. 7c show the evolution of latency and optimal number of *Loading Agents* with respect to memory constraints for ViT-Large model and BERT-Large model. Across the experiments, a trend is the gradual increase in the optimal number of *Loading Agents*, the decrease in the latency in correlation with the augmentation of memory limits. Specifically, the latency dropped from 81 ms at the 60 MB memory limit to 36 ms at 300 MB memory limit for ViT-Large, a reduction of 55.6% and from 7721 ms at the 500 MB memory limit to 2923 ms at the 1250 MB memory limit for BERT-Large, a reduction of 62.1%. All above results are as expected, since higher memory availability allows for more *Loading Agents*.

2) GPT-2 and GPT-J Models Analysis: Fig. 7b and Fig. 7d show the evolution of latency and optimal number of *Loading Agents* with respect to memory constraints for GPT-2-Base model and GPT-J model. Overall, their trends of latency and optimal number of *Loading Agents* are the same as for the previous two models, from 1705 ms at the 400 MB memory limit to 1004 ms at 1000 MB memory limit for GPT-2-Base, a reduction of 41.1%, and from 51003 ms at the 2000 MB memory limit to 29074 ms at the 7000 MB memory limit for GPT-J, a reduction of 43.0%.

VI. RELATED WORK

Memory Optimization. PQK [35] is a novel model compression method, designed expressly for edge devices with constrained computational resources. This method combines pruning, quantisation, and knowledge distillation processes to fabricate a model that is both lightweight and energy-efficient. Keivan *et al.* address the memory challenges for large model inference under memory constraints by storing model parameters in flash memory and bringing them on demand to DRAM and introduce techniques including windowing and row-column bundling to optimize data transfer and memory usage. STI [36] is a memory optimization architecture through model sharding and elastic pipeline, which employs a preload buffer to optimize resource utilization for large model inference tasks on mobile devices. Our work is complementary, focusing on minimizing inference latency by pipeline scheme while reducing memory overhead.

Pipeline Schemes. Prior works have attempted to apply pipeline schemes to optimize large model inference [37]. DeepPlan [38] is an optimized pipeline system that incorporates two mechanisms, direct-host-access and GPU parallel transmission to reduce the model loading latency on the GPU and improve performance. PipeSwitch is a system designed for fine-grained time-sharing of GPU resources for deep learning applications, aiming to optimize task switching overhead and achieve near 100% GPU utilization. This system leverages the structure and computation pattern of DNN models to enable fast context switching with millisecond-scale overhead, addressing inefficiencies in shared GPU clusters where training and inference tasks are provisioned separately. These works mainly focus on reducing inference latency but do not involve memory optimization and require the use of one or even more GPUs. In this paper, we focus on both memory and latency optimization and do not require GPU usage.

VII. CONCLUSION

In this paper, we present PIPELOAD, a memory-efficient pipeline execution mechanism to mitigate memory overhead and address the pipeline stall issue during large model inference on edge devices. This mechanism incorporates dynamic management of memory and deploys multiple *Loading*

Agents to load model weights in parallel. Based on this mechanism, we introduce **Hermes**, an innovative framework to optimize large model inference performance on edge devices. By our evaluation, **Hermes** reaches $4.24\times$ speedup and 86.7% lower memory consumption than PipeSwitch for BERT and ViT models, $2.58\times$ speedup and 90.3% lower memory consumption for GPT-style models. For future research, we are dedicated to applying the **Hermes** to more Transformer models and exploring its generalization and pervasiveness. For text generation large models like GPT, based on their characteristics, we target to optimize PIPELOAD mechanism to provide better latency reduction.

REFERENCES

[1] Z. Cai, Z. Chen, R. Ma, and H. Guan, "Smss: Stateful model serving in metaverse with serverless computing and gpu sharing," *IEEE Journal on Selected Areas in Communications*, 2023.

[2] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–37, 2020.

[3] M. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine learning at the network edge: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–37, 2021.

[4] Y. Ren, J. Liu, X. Tan, C. Zhang, T. Qin, Z. Zhao, and T.-Y. Liu, "Simulspeech: End-to-end simultaneous speech to text translation," in *Proc. of ACL*, 2020.

[5] K. Muhammad, A. Ullah, J. Lloret, J. Del Ser, and V. H. C. de Albuquerque, "Deep learning for safe autonomous driving: Current challenges and future directions," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4316–4336, 2020.

[6] Y. Deng, T. Zhang, G. Lou, X. Zheng, J. Jin, and Q.-L. Han, "Deep learning-based autonomous driving systems: A survey of attacks and defenses," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 12, pp. 7897–7912, 2021.

[7] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *Proc. of IEEE/CVF conference on computer vision and pattern recognition*, 2023.

[8] Y. Cai, H. Li, G. Yuan, W. Niu, Y. Li, X. Tang, B. Ren, and Y. Wang, "Yolobile: Real-time object detection on mobile devices via compression-compilation co-design," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 2, 2021, pp. 955–963.

[9] J. Tang, S. Li, and P. Liu, "A review of lane detection methods based on deep learning," *Pattern Recognition*, vol. 111, p. 107623, 2021.

[10] R. Ayachi, M. Afif, Y. Said, and M. Atri, "Traffic signs detection for real-world application of an advanced driving assisting system using deep learning," *Neural Processing Letters*, vol. 51, no. 1, pp. 837–851, 2020.

[11] B. Lindqvist, S. Karlsson, A. Koval, I. Tevetzidis, J. Haluška, C. Kanellakis, A.-a. Agha-mohammadi, and G. Nikolakopoulos, "Multimodality robotic systems: Integrated combined legged-aerial mobility for subterranean search-and-rescue," *Robotics and Autonomous Systems*, vol. 154, p. 104134, 2022.

[12] G. Ince, R. Yorganci, A. Ozkul, T. B. Duman, and H. Köse, "An audiovisual interface-based drumming system for multimodal human–robot interaction," *Journal on Multimodal User Interfaces*, vol. 15, pp. 413–428, 2021.

[13] G. Cao, J. Jiang, D. Bollegala, M. Li, and S. Luo, "Multimodal zero-shot learning for tactile texture recognition," *Robotics and Autonomous Systems*, vol. 176, p. 104688, 2024.

[14] J. Wu, W. Gan, Z. Chen, S. Wan, and S. Y. Philip, "Multimodal large language models: A survey," in *Proc. of IEEE BigData*, 2023.

[15] P. Xu, X. Zhu, and D. A. Clifton, "Multimodal learning with transformers: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[16] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," *arXiv preprint arXiv:2312.11514*, 2023.

[17] K. S. Kalyan, "A survey of gpt-3 family large language models including chatgpt and gpt-4," *Natural Language Processing Journal*, p. 100048, 2023.

[18] K. Sanderson, "Gpt-4 is here: what scientists think," *Nature*, vol. 615, no. 7954, p. 773, 2023.

[19] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," *Advances in neural information processing systems*, vol. 36, pp. 21 702–21 720, 2023.

[20] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[21] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," in *Proc. of PMLR ICML*, 2019.

[22] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[23] C. Liu, C. Tao, J. Feng, and D. Zhao, "Multi-granularity structural knowledge distillation for language model compression," in *Proc. of ACL*, 2022.

[24] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. Mahoney *et al.*, "Hawq-v3: Dyadic neural network quantization," in *Proc. of PMLR ICML*, 2021.

[25] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks *et al.*, "Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference," in *Proc. of IEEE/ACM MICRO*, 2021.

[26] Q. Jin, L. Yang, and Z. Liao, "Adabits: Neural network quantization with adaptive bit-widths," in *Proc. of IEEE/CVF CVPR*, 2020.

[27] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *Proc. of ACM ASPLOS*, 2020.

[28] J. Hao, P. Subedi, L. Ramaswamy, and I. K. Kim, "Reaching for the sky: Maximizing deep learning inference throughput on edge devices with ai multi-tenancy," *ACM Transactions on Internet Technology*, vol. 23, no. 1, pp. 1–33, 2023.

[29] P. Jiang, H. Wang, Z. Cai, L. Gao, W. Zhang, R. Ma, and X. Zhou, "Slob: Suboptimal load balancing scheduling in local heterogeneous gpu clusters for large language model inference," *IEEE Transactions on Computational Social Systems*, 2024.

[30] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, "Pipeedge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices," in *Proc. of IEEE DSD*, 2022.

[31] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *Proc. of USENIX OSDI*, 2020.

[32] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proc. of ACL*, 2020.

[33] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL, 2019.

[35] J. Kim, S. Chang, and N. Kwak, "PQK: Model Compression via Pruning, Quantization, and Knowledge Distillation," in *Proc. of Interspeech*, 2021.

[36] L. Guo, W. Choe, and F. X. Lin, "Sti: Turbocharge nlp inference at the edge via elastic pipelining," in *Proc. of ACM ASPLOS*, 2023.

[37] H. Shi, W. Zheng, Z. Liu, R. Ma, and H. Guan, "Automatic pipeline parallelism: A parallel inference framework for deep learning applications in 6g mobile communication systems," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 7, pp. 2041–2056, 2023.

[38] J. Jeong, S. Baek, and J. Ahn, "Fast and efficient model serving using multi-gpus with direct-host-access," in *Proc. of ACM EuroSys*, 2023.