

# Multithreaded Pipeline Parallelism for Distributed LLM Inference Without Source Code Modification

Victor Koloskov(✉) and Anton Pershin

St Petersburg State University, St Petersburg 199034, Russia
st107068@student.spbu.ru, a.pershin@spbu.ru

**Abstract.** Modern large language models (LLMs) reach tens or even hundreds of billions of parameters, making it impossible to fit them into the memory of a single GPU. To perform inference for such models, multiple accelerators are used, across which the model is distributed. However, many current distributed inference frameworks require modifications to the model's source code or adaptation of its architecture. Tools that do not require model changes are typically oriented towards using tensor parallelism within a single node. In the case of inter-node interaction, inference performance with such partitioning can significantly decrease due to frequent data exchange between accelerators and limited network bandwidth. One way to reduce inter-node communication is the use of pipeline parallelism. However, existing solutions rarely provide the ability to use pipeline parallelism for distributed inference without modifying the model code. This paper presents a concept for partitioning a model between accelerators by modifying its layers without affecting the forward pass function itself. A realization of pipeline parallelism with computational overlap between graphics accelerators is also presented. The proposed approaches allow for improved distributed inference performance without changing the LLM source code.

**Keywords:** Large Language Models (LLMs) · Distributed Inference · Pipeline Parallelism

## 1 Introduction

Currently, large language models (LLM) may require hundreds of gigabytes of accelerator memory for deployment and inference. For example, the Llama 3.1-405B model [1] is the flagship model of the Llama family and requires approximately 810 GB in FP16 format just for weight storage. The DeepSeek-V3 model [2] contains 671 billion parameters and requires approximately 1350 GB of memory. At the same time, the NVIDIA H200 [3], currently the accelerator with the largest memory capacity, is equipped with 141 GB HBM, which is insufficient to run Llama 3.1-405B or DeepSeek-V3 on a single accelerator. To solve this problem, the model is split between several interconnected accelerators, adding the

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026 Workshops, LNCS 16763, pp. 464–480, 2027.
https://doi.org/10.1007/978-3-032-30530-5_29

necessary data exchange operations between GPUs. There are two main ways of model partitioning: intra-layer and inter-layer.

**In intra-layer partitioning**, each layer is divided among several GPUs. This allows for the parallel execution of a single layer's computations, which is especially effective for layers with a large number of parameters. The main advantage of this approach is the ability to evenly distribute the computational load between accelerators. However, this method requires frequent exchange of intermediate activations between GPUs, making its efficiency highly dependent on network bandwidth. Therefore, this partitioning method is typically most effective within a single computing node. In LLM distributed computing tasks, this approach is usually implemented as **tensor parallelism**.

**In inter-layer partitioning**, each GPU receives its own group of layers for which it performs computations. Data is passed sequentially between accelerators as it moves through the model's layers. The main advantage of this approach is lower dependence on network bandwidth, allowing it to be used for inter-node interaction. However, the computational load between accelerators may be distributed unevenly and requires correct execution queue management to reduce GPU idle time. In LLM distributed computing tasks, this approach is implemented as **pipeline parallelism**.

Existing frameworks for distributed inference [4–10] typically support both intra-layer and inter-layer model partitioning. However, leveraging these tools typically necessitates modifying the model's source code or adapting its architecture[1], which requires deep knowledge of the model's internal structure. Solutions that do not require such modifications usually rely on tensor parallelism within a single node, a strategy that often proves inefficient for inter-node scaling due to high communication overheads.

Pipeline parallelism is significantly less common as a ready-made solution for distributed inference, and existing implementations do not always provide high efficiency. This is because, in naive implementations, an accelerator cannot start processing a new batch until the previous one is fully processed at all stages of the pipeline, resulting in GPUs being idle for a significant amount of time.

In this work, we present the concept of **placing a model on several accelerators by substituting model layers** which preserves the original forward pass function and eliminates the need for source code modification (Sect. 3). We also present an **implementation of pipeline parallelism for distributed inference with input batch splitting into several microbatches**, which reduces GPU idle time (Sect. 4).

## 2 Related Work

Pipeline parallelism was first examined in detail in the work by Y. Huang et al. [12], where the GPipe library for neural network training was introduced. The

[1] Source code adaptation is defined as the modification of the forward pass implementation and layer computation logic necessary to distribute the model across multiple accelerators for inference.

key innovation was splitting the batch into several microbatches. This made it possible to reduce the "bubble"—the idle period of computing resources occurring when there is no load on the GPU—and thus increase the speed of the forward and backward passes. The next improvement was proposed in the work by A. Harlap et al. [13] with the Pipedream framework. The paper presented a new pipeline schedule that further reduced GPU idle time during model training. Subsequent works on pipeline parallelism [14–17] primarily aimed to improve scheduling to minimize accelerator idle time. Today, pipeline parallelism is implemented in several popular frameworks. Below is their description and comparative analysis.

### 2.1 Megatron-LM [4]

The Megatron-LM framework was first presented in the work by M. Shoeybi et al. [4] as a solution for LLM training using tensor and data parallelism. Today, it also supports pipeline parallelism [18] and has tight integration with NVIDIA accelerators. However, the framework is more often used for model training and requires adapting the model's source code to its architecture.

### 2.2 MindSpore [5]

MindSpore is an open and flexible computing platform for developing and executing AI models, created by Huawei. The platform is distributed with fully open source code and, besides GPUs, can run on Ascend NPU processors. MindSpore supports several types of parallelism and can automatically select optimal model partitioning strategies between accelerators. However, using the platform requires model source code adaptation.

### 2.3 Colossal-AI [6]

Initially, Colossal-AI was developed as a solution for LLM training on multiple GPUs with support for various types of parallelism: data, pipeline, tensor, sequence parallelism, and for working with heterogeneous systems. Subsequently, Colossal-AI also began to be used for distributed inference. For standard models from the Transformers library, additional adaptation is generally not required. In the absence of support for a specific model, it is necessary.

### 2.4 vLLM [7]

The vLLM framework is designed for high-performance LLM inference. The system's primary goal is to increase throughput and reduce latency when serving a large number of requests. To achieve this, optimizations for memory management, batching, and efficient KV-cache management during generation have been implemented.

Furthermore, vLLM is tightly integrated with the HuggingFace Transformers library and provides broad coverage of models used in distributed inference. If a

model is not supported, it can still be used through the "Transformers modeling backend". However, the model must satisfy certain requirements [11].

### 2.5 TensorRT-LLM [8]

TensorRT-LLM is a system for high-performance LLM inference on NVIDIA GPUs. It is based on the NVIDIA TensorRT library and utilizes low-level optimizations such as kernel fusion, graph optimization, and specialized CUDA kernels to accelerate inference. TensorRT-LLM supports several types of parallelism but requires converting the model into a format compatible with TensorRT.

### 2.6 DeepSpeed [9]

The DeepSpeed system provides the DeepSpeed-Inference tool [19], designed to accelerate the inference of large language models. To improve text generation efficiency, the library also implements a specialized engine, DeepSpeed-FastGen [20], designed to accelerate autoregressive token generation. DeepSpeed-Inference fully supports tensor parallelism for any model without additional code adaptation by replacing model layers with parallel ones. It also supports ZeRO-Inference technology [21], which allows for the asynchronous offloading of model layers to RAM or disk and loading them back as needed. For pipeline parallelism, model adaptation is still required.

### 2.7 Accelerate [10]

Accelerate was developed by Hugging Face for distributed LLM training and inference without the need to adapt the model. It can automatically distribute the model between accelerators and work with it as if it were not partitioned. The main disadvantage is that, in a strict sense, the framework does not implement a specific type of parallelism but only shards the model, which can result in lower inference throughput.

### 2.8 Summary Comparison

Table 1 presents a summary comparison of existing solutions for distributed LLM inference. The analysis shows that currently, there is no tool that simultaneously provides efficient use of pipeline parallelism and does not require significant model adaptation for a specific framework.

## 3 Proposed Method for Model Partitioning

The main problem with inter-layer model partitioning is the need to modify the model's forward pass function to ensure its operation on multiple accelerators. This work proposes adapting the approach used in the DeepSpeed-Inference system [19] with tensor parallelism to implement pipeline parallelism. The core

**Table 1.** Comparison of existing distributed inference frameworks

| Framework | Key Characteristics | Source Code Adaptation |
|---|---|---|
| Megatron-LM | tight integration with NVIDIA GPUs | requires model source code adaptation |
| MindSpore | automatically selects optimal model partitioning; supports Ascend NPU | requires model source code adaptation |
| Colossal-AI | support for heterogeneous systems | supports standard Transformers model; requires adaptation if a specific model is not supported |
| vLLM | optimizations for high-performance inference; tight integration with Transformers library | Supports a wide range of models; unsupported Transformers models can be used via the Transformers backend, subject to certain requirements |
| TensorRT-LLM | uses low-level optimizations for NVIDIA GPU acceleration | requires model conversion to TensorRT-compatible format |
| DeepSpeed | multiple inference optimizations; ZeRO-Inference | tensor parallelism is automatic; pipeline parallelism requires adaptation |
| Accelerate | Ease of use; tight integration with Transformers library | requires no code adaptation, though inference throughput is generally lower |

idea is to replace standard model layers with specialized parallel layers, which allows avoiding rewriting the forward pass function and limiting changes only to layer replacement. An additional advantage of this approach is that, in pipeline parallelism, each layer is executed on only one device, so its computational logic does not require modification. However, this property simultaneously creates an additional challenge. Even when partitioning the model between accelerators, it is necessary to execute the forward pass function on each GPU and pass intermediate data between pipeline stages.

To solve this task, three types of layers are introduced: **Compute**, **Fake**, and **Boundary**. The **Compute** layer performs the main computations of the corresponding model layer on the GPU. The **Boundary** layer is responsible for executing computations in the final layer of the current stage and transmitting the output to the subsequent accelerator, as well as receiving input data from the preceding stage. The **Fake** layer contains no trainable parameters and is used to generate dummy zero-value output data necessary for executing the forward pass function on a GPU where the current stage's computations are not being performed. This design ensures that the forward pass function requires no modification, as the substitution operates at the module level rather than altering the underlying computation logic.

Figure 1 shows an example of model partitioning across multiple GPUs. In this case, the model is distributed among several accelerators, yet for the forward pass function, it remains logically complete on each device. Final **Boundary** layer is additionally required to propagate up-to-date output data to all accelerators. This is especially important in iterative inference procedures, such as autoregressive generation, where the first pipeline stage must receive the actual output produced in the previous step.

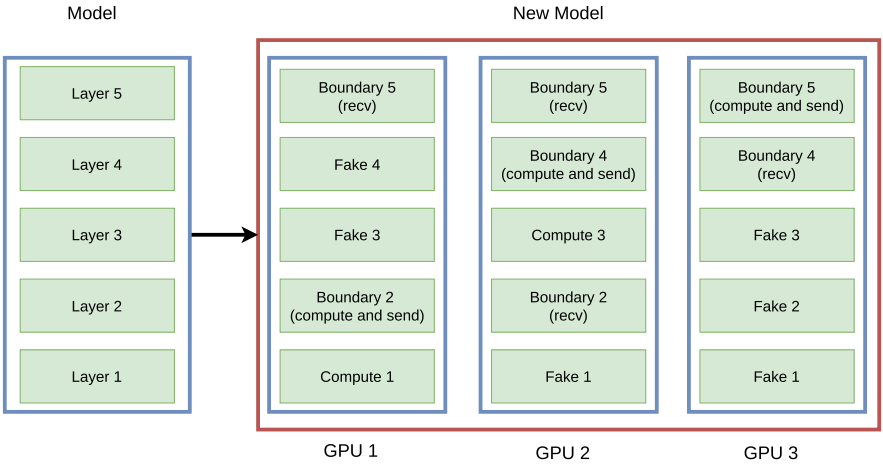


**Fig. 1.** Example of partitioning a five-layer model between several GPUs.

Figure 2 illustrates the forward pass process of a partitioned model. The input batch arrives at all accelerators simultaneously. On GPU 2 and GPU 3, based on the input data, necessary intermediate values are formed in Fake-type layers up to the Boundary layer, where execution is paused until output data are received from the previous stage. At the same time, standard computations are performed on Compute layers on GPU 1. At the Boundary layer, activations of the last layer of the current stage are calculated, and then the data is passed to GPU 2.

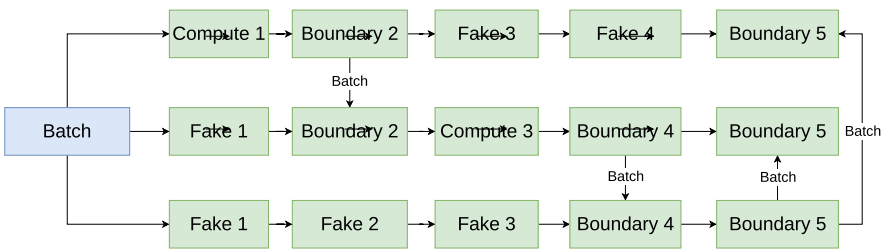


**Fig. 2.** Example of the forward pass of a partitioned model.

After receiving input data on GPU 2, the next stage of the pipeline is executed, and upon its completion, the batch is passed to GPU 3. On GPU 3, the remaining part of the forward pass is executed, after which the final layer's output data is passed to all accelerators.

To simplify the layer replacement process in the model, the **Generator** mechanism is proposed. Initially, the model in each process is placed in RAM. Each

model layer is passed to a corresponding Generator, which, depending on the chosen strategy, replaces the original layer with one of the types: Compute, Fake, or Boundary, after which the modified layer is placed on the GPU. Each process uses its own sequence of generators defining the rules for model layer transformation.

A natural concern with the layer substitution approach is the handling of non-linear architectural patterns such as skip connections and Mixture-of-Experts layers. As a general design principle, the partitioning strategy should aim to avoid splitting such non-linear constructs across pipeline stages: the granularity of partitioning is to be chosen such that these architectural features remain encapsulated within a single indivisible block. This principle is adopted in the present work and ensures that no skip connection crosses a stage boundary.

In cases where such splitting is unavoidable, the framework can be extended to support cross-stage data flow through the introduction of additional Fake-layer variants. Under this scheme, the auxiliary tensors required by the consuming stage are transmitted from the producing stage prior to the start of the next pipeline stage and retained at the receiving stage until the point in the forward pass at which they are consumed. A detailed design and evaluation of this extension is left for future work.

Thus, adapting a new model is reduced to implementing a corresponding Generator that defines the layer replacement policy and establishes a uniform rule across all processes for generating Fake layer outputs based on the input batch. In essence, this reduces model support to a declarative description of the model's structure, without altering its underlying computational logic. If necessary, the architecture can be extended by adding new data transfer strategies between GPUs or additional Fake-category layer types.

## 4 Multithreaded Pipeline Parallelism

The proposed implementation of pipeline parallelism has a significant drawback. GPUs that have finished their stage's computations cannot start processing the next batch because they must wait for output data at the final layer (Fig. 2). As a result, accelerators may stand idle, leading to reduced inference throughput.

To solve this problem, we propose using multithreading within the pipeline. Similarly to GPipe [12], we split the original batch into several microbatches. Each microbatch is processed by a separate execution thread. After completing the current stage's computations, the thread passes the calculated activations to the next pipeline stage and suspends execution. Control is then passed to the next thread, which begins processing another microbatch (Fig. 3). This ensures overlapping of computations across different pipeline stages and reduces GPU idle time.

Figure 4 shows a comparison of pipeline parallelism schedules without and with multithreading.

The multithreading functionality is implemented using Python's standard threading library. A natural concern is Python's Global Interpreter Lock (GIL),

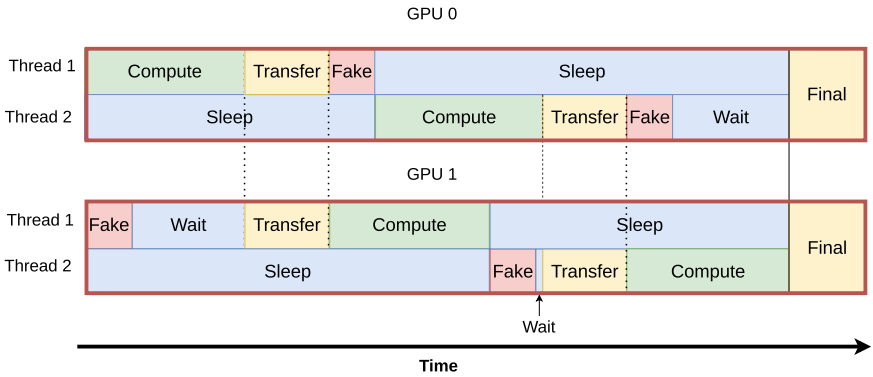


**Fig. 3.** Example of multithreaded pipeline operation with two GPUs and two threads.

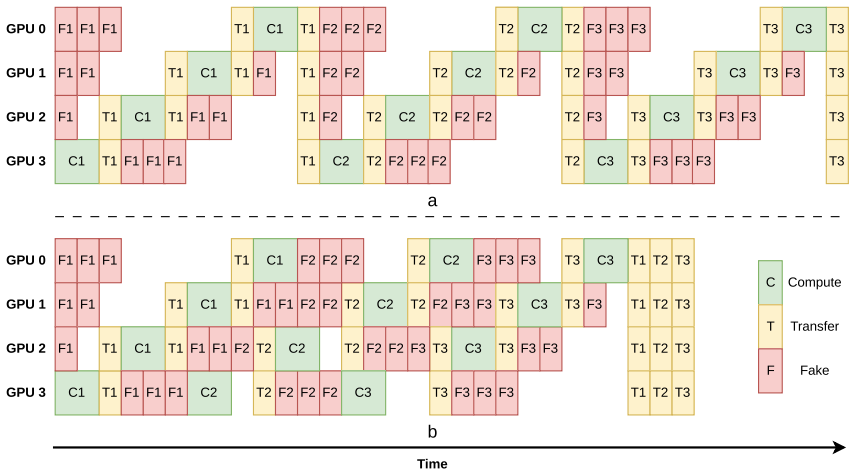


**Fig. 4.** Comparison of pipeline schedules without multithreading (a) and with multithreading (b).

which prevents multiple threads from executing simultaneously within a single process. However, this limitation does not affect performance in our case: each process is responsible for a separate pipeline stage and contains one thread per microbatch. At any given moment, only one thread is active—the others wait for the current stage to complete (Fig. 2). Thus, the scheduler architecture naturally aligns with the constraints imposed by the GIL.

## 5 Evaluation of Multithreaded Pipeline Parallelism

This section demonstrates the results of comparing throughput and peak GPU memory consumption for standard and multithreaded pipeline parallelism. For

the experiments, 2 computing systems were used. Table 2 presents the system characteristics, models used, and the data type in which the experiments were conducted.

It should be noted that pipeline parallelism is typically applied in systems with limited interconnect bandwidth, such as PCIe-based intra-node configurations or inter-node connections, since it requires data transfers only at the boundaries between pipeline stages. In contrast, on systems with high-bandwidth interconnects such as NVLink, tensor parallelism is generally more efficient. For this reason, hardware configurations without NVLink were selected for the experiments.

**Table 2.** Conducted experiments

| System | Models | Data Type |
|---|---|---|
| 2 × NVIDIA P104-100 (8 GB) | OPT-1.3B, OPT-2.7B, DebertaV2-xlarge, DinoV2-Giant | FP32 |
| 4 × NVIDIA Tesla T4 (16 GB) | OPT-6.7B, OPT-13B, BLOOM-7B1, internVIT-6B | FP16 |

To conduct the experiments, a customized parameter grid was established for each model, with boundaries determined by the available accelerator memory. While this constrained the maximum batch size, input sequence length, and number of output tokens, such an approach enables a precise investigation into how pipeline parallelism efficiency depends on key inference parameters.

### 5.1 Experiments with 2 × NVIDIA P104-100 (8 GB)

**Decoder-Only Models.** Figure 5 presents the results for the OPT-1.3B model. The graphs on the left show the ratio of multithreaded pipeline throughput to standard pipeline parallelism throughput under various batch sizes, input sequence lengths, and numbers of generated tokens. The graphs on the right demonstrate the optimal number of microbatches the input batch should be divided into to achieve maximum throughput. The center graphs show the ratio of peak GPU memory consumption when using multithreaded versus standard pipelines. On average, the greatest acceleration is achieved with a microbatch size of 24 and ranges from 1.10× to 1.24×. It can also be noted that as the input sequence length increases, the acceleration grows, whereas the peak memory consumption ratio decreases.

Figure 6 presents the results for the prefill stage of the OPT-1.3B model. It can be seen that the average acceleration is higher in this case, while the peak memory consumption ratio remains virtually unchanged. Based on the results shown in Figs. 5 and 6, it can be concluded that the multithreaded pipeline is most effective during the prefill stage. Furthermore, as the batch size, sequence

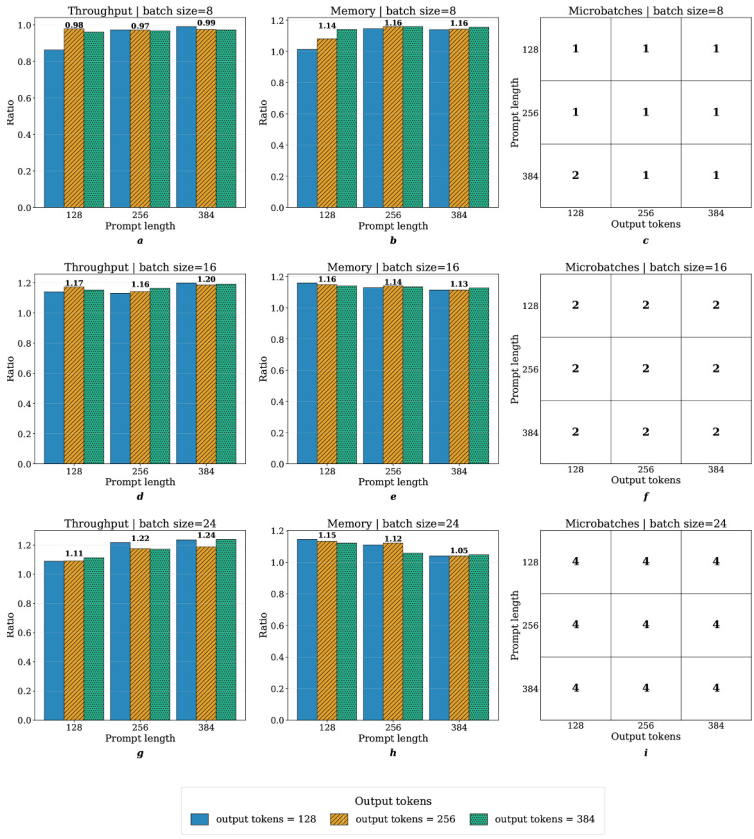


**Fig. 5.** Results for the OPT-1.3B model. Throughput ratio of multithreaded vs. normal pipeline (a, d, g). Peak memory consumption ratio of multithreaded vs. normal pipeline (b, e, h). Optimal number of microbatches for multithreaded pipeline (c, f, i).

length, and number of generated tokens increase, relative peak memory consumption decreases. This is explained by the fact that multithreaded mode requires some additional memory, the impact of which becomes less significant as input data parameters grow.

Figure 7 shows results for the OPT-2.7B model. It is noticeable that as the model size increases, the acceleration grows overall, reaching from 1.22× to 1.39× at a batch size of 24 at the optimal microbatch size. Simultaneously, the optimal number of microbatches increases, while the relative peak memory consumption decreases. This suggests that using a multithreaded pipeline becomes more effective as model size increases. This is due to the fact that in larger models, the computation time of individual layers is longer, increasing the opportunity for overlapping computations between pipeline stages.

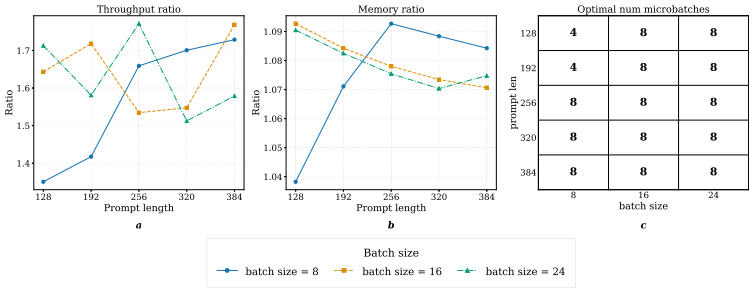


**Fig. 6.** Prefill stage results for the OPT-1.3B model. Throughput ratio of multithreaded vs. normal pipeline (a). Peak memory consumption ratio of multithreaded vs. normal pipeline (b). Optimal number of microbatches for multithreaded pipeline (c).

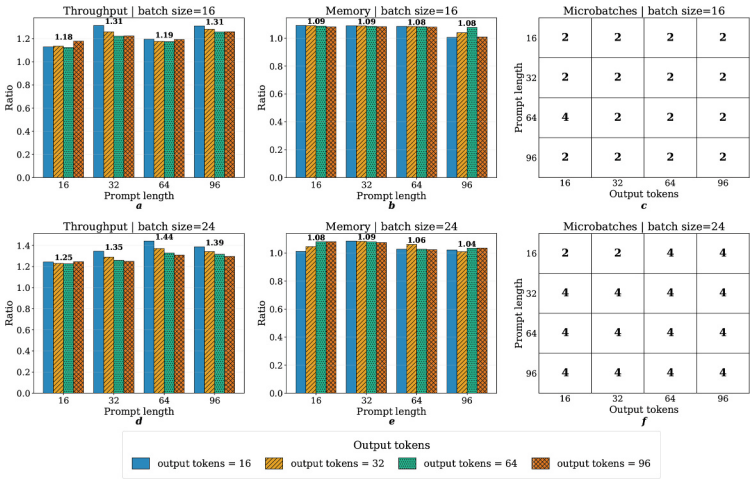


**Fig. 7.** Results for the OPT-2.7B model. Throughput ratio of multithreaded vs. normal pipeline (a, d). Peak memory consumption ratio of multithreaded vs. normal pipeline (b, e). Optimal number of microbatches for multithreaded pipeline (c, f).

**Encoder Models.** Figure 8 presents experimental results for the DebertaV2-xlarge text encoder model. The graphs show that using a multithreaded pipeline allows for an average throughput increase of up to 1.46× compared to standard pipeline parallelism. Acceleration increases with larger input sequence lengths and batch sizes. It is also observed that as these parameters increase, the peak memory consumption ratio decreases and, in some cases, can drop by nearly half. This is because the input batch is divided into smaller microbatches, reducing the memory volume required to store intermediate activations.

Figure 9 shows experimental results for the DinoV2-Giant model. The graphs show that as input batch size increases, the peak memory consumption ratio decreases. Simultaneously, with an increase in the number of microbatches, the

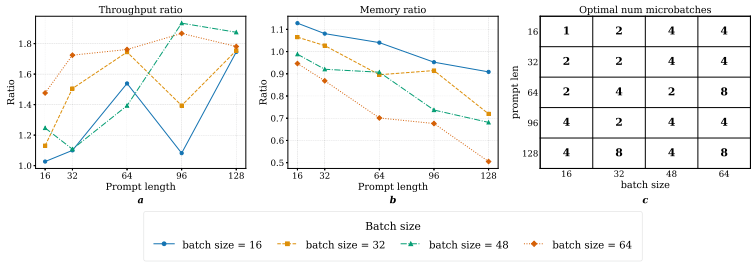


**Fig. 8.** Results for the DebertaV2-xlarge model. Throughput ratio of multithreaded vs. standard pipeline (a). Peak memory consumption ratio (b). Optimal number of microbatches (c).

throughput ratio of the multithreaded pipeline to the standard one increases. Using 8 microbatches results in acceleration between 1.45× and 2×. Peak memory consumption ratio also decreases, reaching as low as 0.64×.

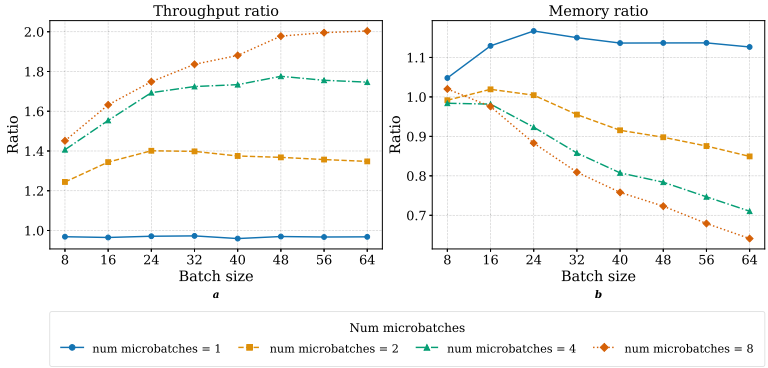


**Fig. 9.** Results for the DinoV2-Giant model. Throughput ratio of multithreaded vs. standard pipeline (a). Peak memory consumption ratio (b).

The results obtained for encoder architectures can be explained by the fact that their layer activation computation time is generally longer. This increases the possibility of overlapping computations between pipeline stages, leading to more efficient utilization of computing resources.

### 5.2 Experiments with 4×NVIDIA Tesla T4 (16 GB)

**Decoder-Only Models.** Figure 10 presents experimental results for the OPT-6.7B model. Acceleration at a batch size of 48 is $1.07 - 1.46\times$, while the relative peak memory consumption increases only by $1.04 - 1.08\times$.

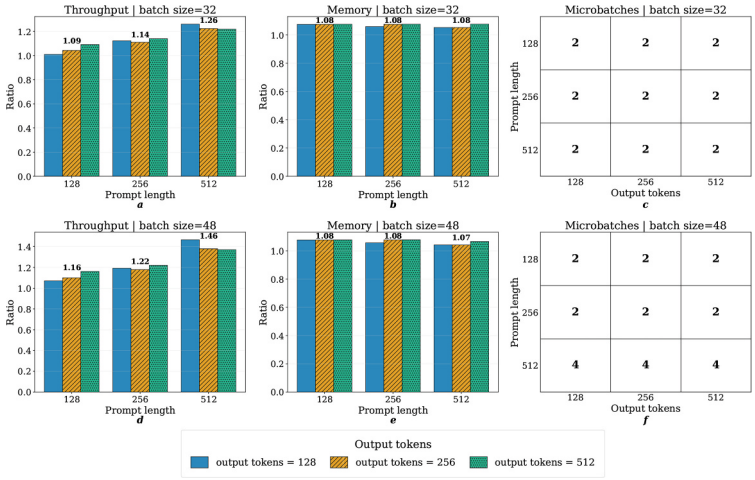


**Fig. 10.** Results for the OPT-6.7B model. Throughput ratio of multithreaded vs. standard pipeline (a, d). Peak memory consumption ratio (b, e). Optimal number of microbatches for multithreaded pipeline (c, f).

Figure 11 presents results for the BLOOM-7B1 model. Compared to OPT-6.7B with comparable batch parameters, sequence length, and token generation count, the average acceleration for BLOOM-7B1 is 3–4% higher. This is explained by architectural features of the model leading to heavier layer computations and a larger number of parameters. It should be noted that if the microbatch size is too small, throughput may decrease due to insufficient overlapping of computations and data transfer to the next pipeline stage.

Figure 12 presents results for the OPT-13B model. Average acceleration compared to OPT-6.7B increased by 5.7%, and the peak memory consumption ratio of the multithreaded pipeline to the standard one decreased by 2.6%. These data confirm that multithreaded pipeline efficiency grows with model size due to longer individual layer computation times and better potential for computational overlap.

A difference from the 2×P104-100 system results should be noted: on the 4×T4 system, relative peak memory consumption increases with the number of generated tokens. This is due to more complex computation distribution across multiple GPUs, resulting in increased load on individual devices, requiring more memory for multithreaded execution.

**Encoder Model.** Figure 13 presents results for the internVIT-6B model. Increasing the number of microbatches leads to higher acceleration, reaching $2.58 - 2.63\times$ with 8 microbatches. Simultaneously, the peak memory consumption ratio drops to $0.34 - 0.55\times$, depending on the batch size.

Comparison of results for DebertaV2-xlarge (Fig. 8), DinoV2-Giant (Fig. 9), and internVIT-6B (Fig. 13) shows that the multithreaded pipeline is especially

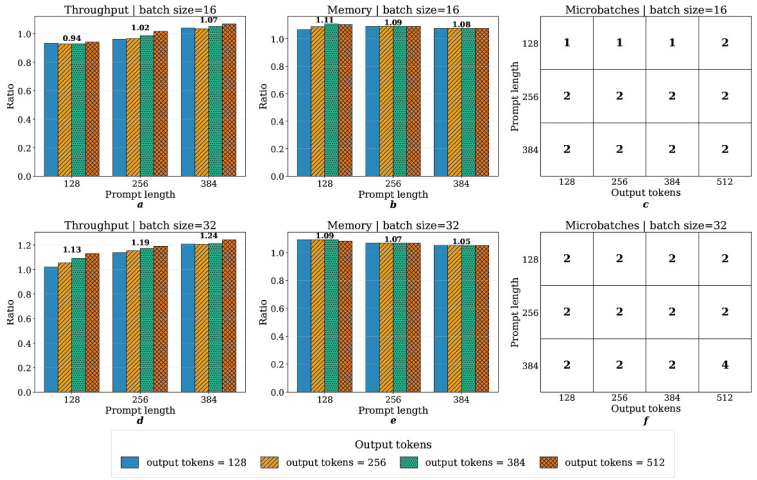


**Fig. 11.** Results for the BLOOM-7B1 model. Throughput ratio of multithreaded vs. standard pipeline (a, d). Peak memory consumption ratio (b, e). Optimal number of microbatches (c, f).

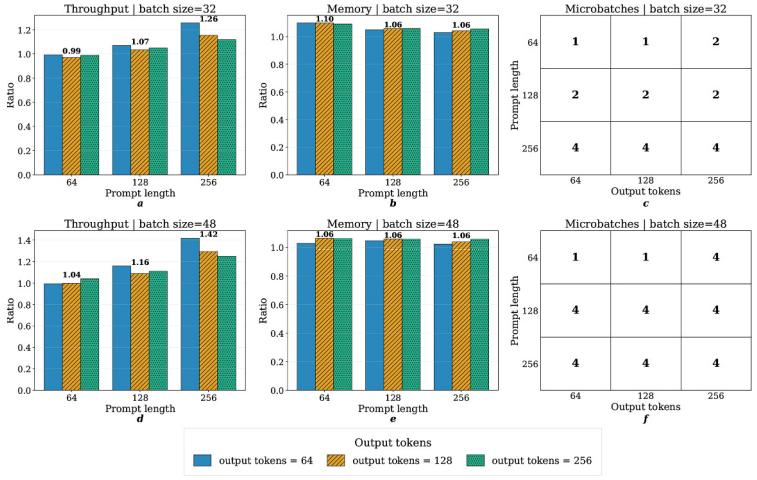


**Fig. 12.** Results for the OPT-13B model. Throughput ratio of multithreaded vs. standard pipeline (a, d). Peak memory consumption ratio (b, e). Optimal number of microbatches (c, f).

effective for encoder architectures. This is linked to the longer computation time for layer activations, allowing for more effective overlapping of computations across pipeline stages, thereby increasing throughput and reducing peak memory consumption.

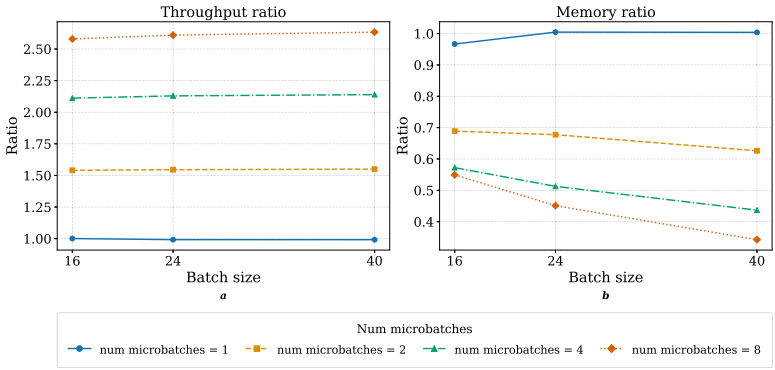


**Fig. 13.** Results for the internVIT-6B model. Throughput ratio of multithreaded vs. standard pipeline (a). Peak memory consumption ratio (b).

## 6 Conclusions and Future Work

In recent years, many powerful tools for distributed inference have been developed. However, as shown in Sect. 2, popular frameworks lack a universal solution that allows for easy adaptation of arbitrary models while ensuring high efficiency in inter-layer model partitioning. In Sect. 3, a model partitioning method based on layer substitution using generators was proposed. This approach avoids changes to the model source code and simplifies its adaptation for multi-GPU inference using pipeline parallelism. Furthermore, Sect. 4 proposed a strategy to accelerate pipeline parallelism using multithreading. Experimental results (Sect. 5) show that for Decoder-Only architectures, this approach increases inference throughput by approximately 15–30%, with an increase in peak memory consumption of about 5–15% given an optimal microbatch count. For Encoder architectures, throughput increases by 40–160% under similar conditions, while peak memory consumption decreased by up to 60%.

**Limitations and Future Work.** Several aspects of this study suggest directions for further investigation. First, while the use of interconnects with limited bandwidth represents a realistic deployment scenario for pipeline parallelism, evaluation on NVLink-enabled systems would help characterize the framework's behavior under high-bandwidth regimes. Second, due to GPU memory constraints of the test systems, batch sizes were limited to 48; in experiments with larger batches, memory management may begin to dominate over computational overlap, and such regimes constitute an important direction for future work. Third, the "normal" pipeline used as a baseline against the "multithreaded" pipeline corresponds to a naive sequential pipeline, which is functionally equivalent to the model sharding employed by Hugging Face Accelerate. A direct comparison with highly optimized inference systems such as vLLM, which integrate pipeline parallelism with continuous batching, paged KV-cache management, and custom CUDA kernels, would provide additional context, although

such comparisons would conflate the contribution of the parallelism mechanism itself with broader system-level optimizations and would require disentangling these factors. We plan to address all three directions in future work, alongside studying the interaction of multithreaded pipeline parallelism with other types of parallelism and optimizing pipeline scheduling to further increase throughput and reduce peak memory consumption.

## References


1. Meta AI: Introducing Meta LLaMA 3.1. https://ai.meta.com/blog/meta-llama-3-1. Accessed 01 Mar 2026
2. DeepSeek-AI, Liu, A., Feng, B., Xue, B., et al.: DeepSeek-V3 Technical report. arXiv:2412.19437 (2024)
3. NVIDIA H200. https://www.nvidia.com/en-us/data-center/h200/. Accessed 25 Feb 2026
4. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-LM: Training Multi-billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 (2019)
5. MindSpore Documentation. https://www.mindspore.cn/docs/en/r2.3.1/index.html. Accessed 21 Feb 2026
6. Li, S., et al.: Colossal-AI: a unified deep learning system for large-scale parallel training. arXiv:2110.14883 (2021)
7. vLLM GitHub. https://github.com/vllm-project/vllm. Accessed 04 Mar 2026
8. TensorRT-LLM Documentation. https://docs.nvidia.com/tensorrt-llm/index.html. Accessed 04 Mar 2026
9. DeepSpeed Official Site. https://www.deepspeed.ai. Accessed 04 Mar 2026
10. Hugging Face Accelerate. https://huggingface.co/docs/accelerate/index. Accessed 02 Mar 2026
11. vLLM Supported Models. https://docs.vllm.ai/en/latest/models/supported_models. Accessed 19 Mar 2026
12. Huang, Y., et al.: GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. arXiv:1811.06965 (2018)
13. Harlap, A., et a.: PipeDream: Fast and Efficient Pipeline Parallel DNN Training. arXiv:1806.03377 (2018)
14. Li, S., Hoefler, T.: Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2021)
15. Li, Z., et al.: TeraPipe: token-level pipeline parallelism for training large-scale language models. In: International Conference on Machine Learning, pp. 6543–6552. PMLR (2021)
16. Qi, P., Wan, X., Huang, G., Lin, M.: Zero Bubble Pipeline Parallelism. arXiv:2401.10241 (2023)
17. Sun, A., et al.: Seq1F1B: Efficient Sequence-Level Pipeline Parallelism for Large Language Model Training. arXiv:2406.03488 (2024)
18. Narayanan, D., et al.: Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 (2021)
19. Aminabadi, R.Y., et al.: DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. arXiv:2207.00032 (2022)

20. Holmes, C., et al.: DeepSpeed-FastGen: High-Throughput Text Generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671 (2024)
21. ZeRO-Inference: Democratizing Massive Model Inference. https://www.deepspeed.ai/2022/09/09/zero-inference.html. Accessed 05 Mar 2026