Empowering Large Language Models with Efficient and Automated Systems

Zhuohan Li

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-150

http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-150.html

July 28, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Empowering Large Language Models with Efficient and Automated Systems

by

Zhuohan Li

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Associate Professor Joseph E. Gonzalez
Associate Professor Matei Zaharia
Professor Eric P. Xing

Summer 2024

Empowering Large Language Models with Efficient and Automated Systems

Copyright 2024

by

Zhuohan Li

Abstract

Empowering Large Language Models with Efficient and Automated Systems

by

Zhuohan Li

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Large Language Models (LLMs) have shown remarkable capabilities in a variety of tasks, including chatting, programming, and searching. However, the high costs of LLMs are preventing these models from being deployed for the vast majority of applications. In this dissertation, we focus on building efficient and automated systems to reduce costs and democratize access to large language models.

We first introduce systems to optimize computational efficiency and reduce the engineering overhead for distributed LLM training. We develop *TeraPipe*, which proposes a new dimension to perform pipeline parallel training for LLMs, and also *Alpa*, the world's first compiler capable of automatically distributing arbitrary neural networks with all existing parallelization methods.

While training is typically a one-time cost, deploying and serving an LLM requires running LLM inference continuously, which is the top blocker for the real-world deployment of LLMs. We improve the serving scalability with *AlpaServe* through model parallelism, and increase the memory utilization and the LLM inference throughput with a new attention algorithm, *PagedAttention*, and an end-to-end serving system, *vLLM*.

Overall, these systems provide comprehensive solutions that significantly improve both training and inference efficiency for large language models. Together, these systems lower the high costs associated with large language models, democratizing their deployment across various real-world applications.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I feel incredibly fortunate to have received support from so many individuals during my PhD journey at Berkeley. Their guidance has profoundly impacted my professional career and personal growth, making these five years a truly unique experience.

First and foremost, I would like to express my gratitude to my advisor, Ion Stoica. Over time, I have come to realize how fortunate I am to have had Ion guiding me throughout my five-year PhD journey. He gave me the freedom to pursue my interests while providing unwavering support for all of my projects. As a highly successful and experienced academic and entrepreneur, Ion offered invaluable guidance on research direction, execution plans, open-source project management, career paths, and life in general. Contrary to what many might expect, we meet almost weekly throughout my entire PhD journey. His unwavering support and mentorship have played a pivotal role in shaping both my academic and personal growth, for which I will always remain deeply grateful.

A special note of gratitude goes to Joey Gonzalez. Joey has been a great mentor and collaborator, contributing to nearly all of my papers throughout my Ph.D. He consistently offers unique insights into research problems, often providing a perspective distinct from Ion's. Their discussions establish a democratic foundation for all research dialogues in my projects, encouraging me to critically evaluate and determine the best path forward. I would also like to thank Matei Zaharia and Eric Xing for being part of my committee and for providing valuable feedback on this dissertation, my research, and my career.

At Berkeley, we often take for granted just how talented our peers are. I am fortunate to have worked with some of the brightest minds in computer science throughout my PhD:

For Hoplite and Terapipe, I collaborated with Siyuan Zhuang, Danyang Zhuo, and Stephanie Wang. Before my Ph.D., I had no experience in systems research. Siyuan, Danyang, and Stephanie showed me how to build a solid system and conduct innovative systems research, laying a strong foundation for my PhD journey.

For Alpa and AlpaServe, I spent two wonderful years with Lianmin Zheng and Hao Zhang. In retrospect, the Alpa project is an engineering marvel that achieved something unimaginable without collaboration. I learned immensely from them throughout our partnership. I was also fortunate to work with Yonghao Zhuang and Yinmin Zhong, two brilliant interns who are now successful PhD students in the field.

For vLLM and PagedAttention, I was privileged to work with Woosuk Kwon to build what I had dreamed of since the beginning of my PhD study: an open-source system widely adopted around the world. We spent countless days and nights refining every aspect of the project together. I am also grateful to have worked with Simon Mo, who took the initiative to become the project's product manager and elevate vLLM to the next level. Thank you to the other vLLM team members, Xiaoxuan (Lily) Liu and Kaichao You, for the fantastic collaboration. What makes vLLM special is the strong support from its vibrant open-source community. I want to thank Antoni Baum, SangBin Cho, Cade Daniel, Nick Hill, Richard Liaw, Philipp Moritz, Robert Shaw, Chen Shen, Roger Wang, and many others for their contributions. The dedication of these contributors keeps the vLLM project thriving.

Beyond the projects mentioned above, I would like to especially thank Eric Wallace, Sheng Shen, and Kevin Lin, with whom I collaborated on scaling law research even before large language models took the world by storm; Zhanghao Wu, for being an excellent roommate and friend over the past three years; and Wei-Lin Chiang and Frank Sifei Luan, my snowboarding companions and wonderful friends.

In the last three years of my PhD, I am fortunate to be a part-time research intern at Google DeepMind. I thank my intern hosts Yanping Huang and Zhifeng Chen for their invaluable insights into my research. I was also fortunate to collaborate with Yuanzhong Xu and Yu (Emma) Wang. Their mentorship and collaboration significantly enriched my experience, broadening my perspective and deepening my understanding of the field.

During my PhD study, I have been fortunate to meet and learn from the following individuals: Shiyi Cao, Xinyun Chen, Audrey Cheng, Xinyang Geng, Linyuan Gong, Shiyuan Guo, Zhiyang He, Xin Jin, Daniel Kang, Kurt Keutzer, Dacheng Li, Eric Liang, Hao Liu, Vincent Liu, Michael Luo, Wenlong Mou, Robert Nishihara, Hongyu Ren, Ying Sheng, Dawn Song, Shangyin Tan, Sijun Tan, Cody Hao Yu, Hexu Zhao, Ruiqi Zhong, and many other bright individuals.

Before joining Berkeley, I was fortunate to begin my research career alongside a wonderful group of advisors and friends. I would like to thank my undergraduate advisors, Liwei Wang and Di He at Peking University, for introducing me to research. My gratitude also extends to Tao Qin, Fei Tian, and Li Yi for their mentorship during my undergraduate years. Additionally I am grateful to my lifelong friends, now spread across various corners of the world: Xinna Chen, Chongming Ni, Luyao Ren, Yuhao Zhang, and many more. Their unwavering support have been instrumental in shaping my academic journey, providing me with a solid foundation and a network of lifelong connections.

Most importantly, I would like to express my deepest gratitude to my family. I am profoundly thankful to my mother, Zhangdong Yu, and my father, Zicun Li, for raising me in the best possible environment and providing me with unconditional love. In the fourth year of my PhD, my family faced the greatest challenge we had ever encountered: my mother's medical emergency, which eventually led to her passing. Thank my extended family—grandparents, uncles, aunts, cousins, and family friends—for their unwavering support during that incredibly difficult time. Lastly, my heartfelt appreciation goes to my partner and best friend, Zi Lin, for the joy, encouragement, and inspiration she has brought to my life and for the support in my hardest time. This dissertation is dedicated to my family, whose love and support made it all possible.

Chapter 1

Introduction

Large language models (LLMs) have brought remarkable advancements to the computing industry. Applications like conversational chatbots like ChatGPT [101] and programming assistants like GitHub Copilot [46] have been heavily involved in people's daily lives.

However, a high barrier exists between the LLMs and the vast majority of researchers and practitioners, brought by the engineering challenges with the enormous model sizes and the substantial compute requirements. LLMs are often large neural networks with hundreds of billions of parameters. For example, the GPT-3 model [18] has 175 billion parameters, meaning that it takes more than 350 GB memory if stored in half floating point precision. The most advanced GPUs on the market still only have 40-80 GB of memory, which is significantly less than the memory required for the LLMs. To execute these LLMs efficiently, distributed execution of these models, sometimes to a scale of thousands of GPUs, becomes a necessity. However, distributed execution requires careful partitioning and scheduling of the model execution. This additional complexity greatly raises the bar of deploying these models in real-world applications.

In this dissertation, we design algorithms and build systems to improve computational efficiency and reduce the engineering overhead when training and deploying LLMs. In the following sections, we will introduce the contributions of this dissertation in LLM training and LLM inference, respectively. In the end, we will discuss the real-world impact of the systems introduced in this thesis.

1.1 Towards Fully-Automated Parallelization for LLM Training

Popular deep learning frameworks such as TensorFlow [1], PyTorch [104], and JAX [17] have greatly simplified the process of training models on a single GPU. However, the complexity will increase considerably with distributed training, i.e., training a model across multiple

GPUs. Especially when a model exceeds the memory capacity of a single GPU, *model parallelism*, the technique that partitions a single model across multiple GPUs, introduces further overheads and makes achieving optimal performance even more challenging.

We start on this problem by **designing new algorithms to improve distributed training efficiency**. One significant inefficiency comes from the computation overhead of under-utilized GPUs due to the dependency within the parallel algorithm. To reduce computation overhead, we focus on *pipeline parallelism*, where a model is partitioned into multiple pipeline stages. In this case, the inefficiency comes from the so-called *pipeline bubbles*, which is caused by not having enough data slices to fulfill all the pipeline stages. In **TeraPipe** (chapter 2), we discover a new dimension to perform pipeline parallelism for LLMs: we show that one can fine-grainedly pipeline different tokens within a single sequence with high GPU utilization, leading to a 5x speedup when training GPT-3 scale models.

Although algorithmic innovations like TeraPipe improve the efficiency of different model parallel algorithms, there is no universally optimal algorithm for every scenario. Picking the optimal algorithm tightly depends on the LLM's architecture and the computing cluster's configuration. This is challenging even for system experts, let alone other LLM researchers and engineers.

To this end, we develop **Alpa** (chapter 3), **the world's first compiler capable of automatically distributing arbitrary neural networks with all existing parallelization methods**, including data, tensor, and pipeline parallelism. Alpa organizes these existing parallelization methods into a *hierarchical space*, corresponding to the *hierarchical structure* of the compute cluster, allowing it to derive efficient parallel execution plans at each parallelism level automatically. As a result, with no modification to the model code, Alpa can transform any neural network into a distributed version with a near-optimal model-parallel strategy. This not only simplifies the engineering effort for model parallelism, but also brings new capacities—Alpa can generate parallelization plans that match or outperform hand-tuned model-parallel training systems and generalize to new models that cannot be parallelized by existing systems.

1.2 Scalable and High-Throughput LLM Inference and Serving

While training has been a major focus in deep learning, a model actually spends most of its time on inference instead of training: A model is typically trained once but can be used for inference over a long period to serve numerous requests. This has motivated me to extend my research towards addressing practical challenges in LLM inference and serving, optimizing the *scalability* and *speed*.

Scalable serving with model parallelism. Scaling an inference system to serve multiple models on a distributed cluster faces unique challenges. A major one is the *bursty requests*: a model might experience a surge of requests within a certain timeframe while others receive none. Existing systems often implement complex request scheduling and swapping policies to handle these bursts.

In **AlpaServe** (chapter 4), we leverage *model parallelism*, the technique originally designed for partitioning models larger than a single GPU, to increase cluster utilization. With model parallelism, the models can be partitioned into smaller parts, and each GPU can hold multiple model parts. In this case, a burst to one model can be jointly handled by multiple GPUs, which effectively *statistically multiplexes* the GPUs across the models. AlpaServe navigates the tradeoff space among different parallelization and placement strategies with an automated search algorithm. It can serve 6x burstier traffic compared to previous systems and is the first work to show that **model parallelism can also benefit small models that can fit into a single GPU**.

High-throughput inference with paged memory. LLM's structure has inherently limited its inference to one token at a time, which makes parallel processors like GPU underutilized. To improve the inference throughput, more requests need to be processed in a batch. However, the batch size is bottlenecked by *memory*. Specifically, to generate a new token, LLMs require the cached states of all previous tokens, known as the *KV cache*. The KV cache is large and dynamic: a single request's KV cache can take GBs of memory, and its exact size depends on input requests' diverse sequence lengths. As a result, efficiently managing the KV cache presents a significant challenge: the static allocation scheme in existing systems causes up to 80% of memory to be wasted.

To address this issue, we developed **PagedAttention** (chapter 5), an attention algorithm inspired by *virtual memory and paging* in operating systems. PagedAttention allows storing the KV cache in non-contiguous memory blocks. Thus, the KV cache can be allocated on-demand and also shared across requests. PagedAttention reduces the memory waste from above 80% to under 4% and eventually improves the throughput by up to 4x, reaching state-of-the-art. Based on PagedAttention, we build **vLLM**, an end-to-end LLM inference and serving engine. Compared to previous state-of-the-art, vLLM achieved up to **4x higher throughput**.

1.3 Real-World Impact and Future Work

The systems we build in this dissertation go beyond research artifacts and are ready for real-world deployment. The open-source projects built on top of the techniques introduced in this dissertation have gained considerable attention and are actively used across the industry:

- **Alpa** has been productionized by Google and has been integrated into the popular deep learning framework JAX and its compiler XLA.
- **vLLM** has become the most popular open-source LLM serving system in the world, with 23K+ GitHub stars and 460+ contributors worldwide. It has become the default LLM server on Microsoft Azure and has been used and deployed in companies including Anyscale, Apple, AWS, Baidu, ByteDance, Databricks, Google Cloud, IBM, Meta, Microsoft, Scale, Spotify, Uber, and more. Thanks to its industrial impact, vLLM was selected as the first cohort of a16z open-source AI Grants.

The wide user community of these projects not only validates the ideas, but also opens up exciting, continuous pathways of first-hand research challenges, presenting numerous opportunities for exploration in the future. In chapter 6, we conclude this dissertation by summarizing the contributions and discussing future directions for research to realize a future where the most advanced LLMs are accessible to everyone in the world.

Chapter 2

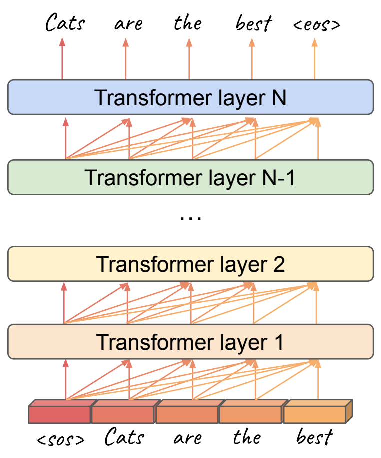
TeraPipe: Token-Level Pipeline Parallelism for Training Large Language Models

Model parallelism has become a necessity for training modern large-scale deep language models. We start the dissertation by designing new algorithms to improve the efficiency of model parallel training for large language models.

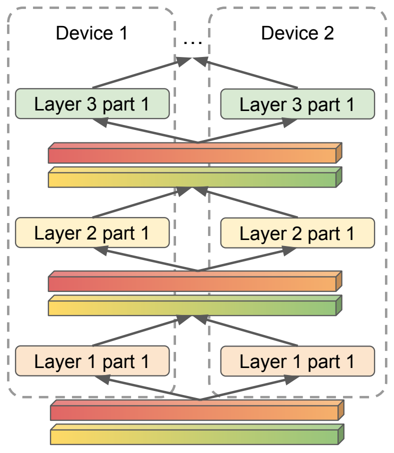
In this chapter, we identify a new and orthogonal dimension from existing model parallel approaches: it is possible to perform pipeline parallelism within a single training sequence for Transformer-based language models thanks to its autoregressive property. This enables a more fine-grained pipeline compared with previous work. With this key idea, we design TeraPipe [83], a high-performance token-level pipeline parallel algorithm for synchronous model-parallel training of Transformer-based language models. We develop a novel dynamic programming-based algorithm to calculate the optimal pipelining execution scheme given a specific model and cluster configuration. We show that TeraPipe can speed up the training by 5.0x for the largest GPT-3 model with 175 billion parameters on an AWS cluster with 48 p3.16xlarge instances compared with state-of-the-art model-parallel methods.

2.1 Introduction

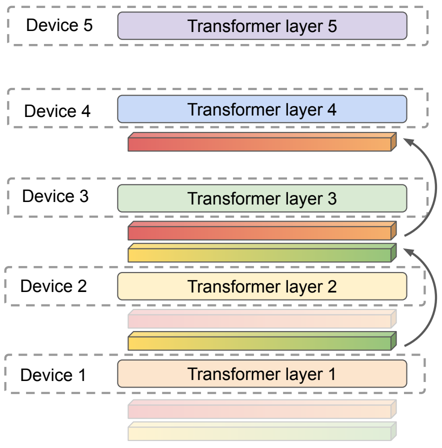
Transformer-based language models (LMs) have revolutionized the area of natural language processing (NLP) by achieving state-of-the-art results for many NLP tasks, including text classification, question answering, and text generation [18, 109]. The accuracy of a Transformer-based LM grows substantially with its model size, attributing to the fact that they can be *unsupervisedly* trained on almost *unlimited* text data. Today, a large LM, such as GPT-3 [18], can have more than 175B parameters, which amounts to 350 GB, assuming 16-bit floating-point numbers. This significantly exceeds the memory capacity of existing



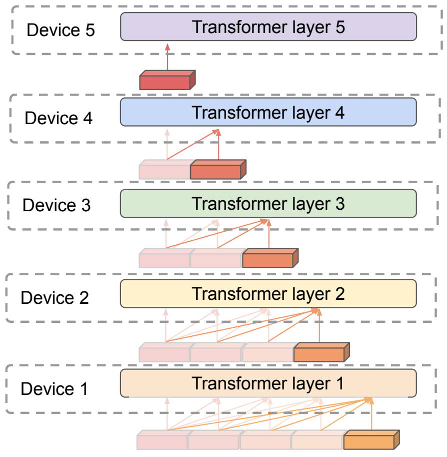
(a) Transformer-based LM



(b) Operation partitioning (Megatron-LM)



(c) Microbatch-based pipeline parallelism (GPipe)



(d) Token-based pipeline parallelism (TeraPipe)

Figure 2.1: Different approaches of model parallel training of Transformer-based LMs. (a) shows a standard multi-layer Transformer LM. In each layer, each position only takes only its previous positions as input. (b) shows operation partitioning [125]. An allreduce operation is required to synchronize the results of each layer. (c) shows microbatch-based pipeline parallelism [57], which allows different microbatches (red and green bars) to be executed on different layers of the DNN in parallel. (d) show TeraPipe (our work), which pipelines along the token dimension.

hardware accelerators, such as GPUs and TPUs, which makes model-parallel training a necessity, i.e., partitioning the model on multiple devices during the training process.

Because of the demands for efficient LM training, many researchers and industry practitioners have proposed different ways for model parallel training. One approach is to partition the weight matrices and dispatch smaller matrix operations to parallel devices [Figure 2.1b; 125, 122]. Another approach is to split a batch of training data into many microbatches and then evenly pipeline the layer computations across different microbatches and devices [Figure 2.1c; 57]. Unfortunately, these approaches either introduce excessive communication overheads between compute devices, or lead to reduced efficiency due to pipeline "bubbles" (i.e. device idle time, see Section 2.2 and 2.3.2 for details).

Our key observation in this chapter is that Transformer-based language models have a key property: the computation of a given input token only depends on previous tokens, but not on future tokens. This lack of dependency on future tokens provides new opportunities for pipeline parallel training.[1] In particular, it allows us to create a fine-grained pipeline within a single training sequence for Transformer-based LMs, by parallelizing the computation of the current token on the current layer with the computation of the previous token on the next layer of the model. For example, in Figure 2.1d, we can pipeline the execution across all 5 devices within a single input sequence. Similar to other synchronous model parallel training methods, e.g., Gpipe [57], Megatron-LM [125], we do not change the underlying optimization algorithm, so the resulting model has exactly the same accuracy.

However, leveraging the token dimension for efficient model parallel training raises several challenges. First, if the partitioning along the token dimension is too fine-grained, it leads to under-utilization on devices that require large blocks of data for efficient processing (e.g., GPU). Second, since each token position in the sequence depends on all previous tokens, different positions in a transformer layer exhibit uneven computation loads. This means that uniformly partitioning along the token dimension might cause uneven load across devices, and degenerate the training efficiency.

To this end, we design and implement *TeraPipe*, a high-performance synchronous model parallel training approach for large-scale Transformer-based language models, which exploits the token dimension to pipeline the computation across devices. TeraPipe uses a small number of simple workloads to derive a performance model and then uses a novel dynamic programming algorithm to compute the optimal partitioning of the token dimension for the pipeline. TeraPipe is orthogonal to previous model-parallel training methods, so it can be used together with these methods to further improve the training performance. Our evaluation shows that for the largest GPT-3 model with 175 billion parameters, TeraPipe achieves a 5.0x speedup improvement over the state-of-the-art synchronous model-parallel training methods on an AWS cluster consisting of 48 p3.16xlarge instances.

[1] In this chapter, we focus on unidirectional autoregressive language models (e.g., GPT [109, 18]) but not bidirectional models like masked language models (e.g., BERT [34]).

Our chapter makes the following contributions:

- We propose a new dimension, token dimension, for pipeline-parallel training of Transformer-based LMs.
- We develop a dynamic programming algorithm to compute a partition along the token dimension to maximize pipeline parallelism.
- We implement TeraPipe and show that we can increase the synchronous training throughput of the largest GPT-3 model (with 175 billion parameters) by 5.0x over the previous state-of-the-art model-parallel methods.

2.2 Related Work

Data parallelism scales ML training by partitioning training data onto distributed devices [169, 72, 49, 110]. Each device holds a model replica, works on an independent data partition, and synchronizes the updates via *allreduce* [72] or a parameter server [78]. Data parallelism alone is not enough to train large-scale DNNs due to two main reasons: (1) every device has to have enough memory to store the model and the gradients generated during the training process; (2) communication can be a performance bottleneck to synchronize model parameters.

Model parallelism allows for training models larger than the memory capacity of a single device, by partitioning the model (e.g., layers) into disjoint parts and executing each on a dedicated device. Existing model parallel training approaches can be roughly categorized as: *operation partitioning* and *pipeline parallelism*.

Operation partitioning. One way to split the model is to partition and parallelize computational operations across multiple devices. For example, the computation of matrix multiplications (matmul) XAB can be spitted across multiple devices by partitioning A and B along its rows and columns, respectively.

$$XAB = X \cdot \begin{bmatrix} A_1 & A_2 \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = XA_1B_1 + XA_2B_2.$$

This means we can have one device calculate XA_1B_1 and another device calculate XA_2B_2 in parallel. After that, cross-device communication is needed to compute the sum of these two parts.

Many existing works [63, 66, 143, 122] study how to optimize the partitioning schemes for different operations to maximize throughput and minimize communication overheads, among which, Megatron-LM [Figure 2.1b; 125] designs partitioning schemes specifically for large-scale Transformers. However, due to the excessive communication required to collect partial results after each layer, it is not efficient when the bandwidth between devices is limited [125]. Flexflow [63] proposes a framework to find the optimal operation partitioning, but it cannot model the new dimension proposed in our work.

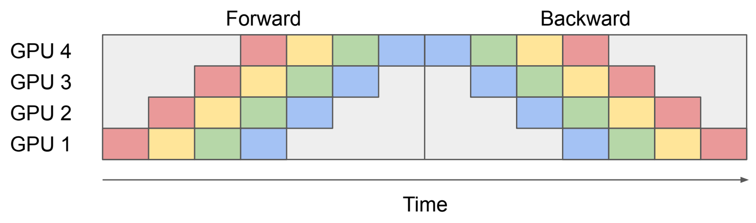
Pipeline parallelism partitions a DNN into layers and put different layers onto different devices [Figure 2.1c; 107]. Each device computes the input on a given layer and sends the result to the next device. Pipeline parallelism significantly reduces communication between devices, because only devices holding neighboring layers need to communicate and they only need to communicate the activations on a particular layer.

Previous pipeline parallel training methods are based on *microbatch* pipelining, e.g., GPipe [57]. This means the computation for a given microbatch in a minibatch on a layer can run in parallel with the next microbatch in the same minibatch on the previous layer. However, microbatch-based pipeline parallelism still cannot achieve high efficiency due to its pipeline bubbles. This is because the start of the forward propagation on a minibatch requires the backward propagation of the previous minibatch to complete (Figure 2.2a). This problem becomes more severe when model sizes increase (see Section 2.3.2). [53] propose using an asynchronous training algorithm to mitigate the effect of pipeline bubbles in microbach-based pipeline parallel training, but asynchronous training introduces uncertainty in model accuracy and is thus not widely adopted for training DNNs.

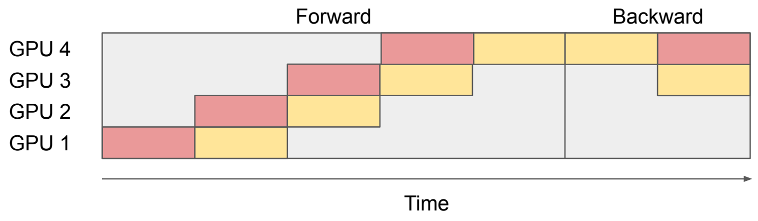
Wavefront parallelism is a variant of pipeline parallelism, broadly applied in shared-memory multiprocessors [128, 86]. In deep learning, it has been used to accelerate the computation of multi-layer RNNs on a single GPU [3], where different input positions of different layers can execute in parallel in a wavefront fashion to maximize the utilization of the GPU. However, wavefront parallelism cannot accelerate the execution of Transformers because there is no dependency between different input positions within a single Transformer layer to begin with. In addition, wavefront parallelism uses fine-grained per-word pipelining due to the temporal data dependency in RNNs, while too fine-grained pipelining in TeraPipe would lead to inferior pipeline efficiency (see Section 2.3.2 and 2.3.3).

2.3 Method

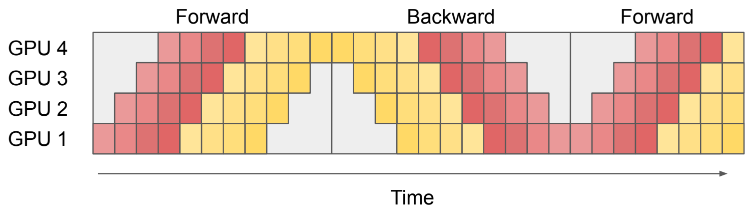
In this section, we briefly introduce language modeling and Transformers. Based on their structures, we identify new opportunities for performing pipelining along the input sequence (which we will notate as the *token dimension* in the rest of the chapter). With that, we derive the optimal slicing scheme over the token dimension to maximize pipeline efficiency using a dynamic programming algorithm. Finally, we show how to combine our new method with existing parallel training techniques.



(a) Microbatch-based pipeline parallelism



(b) Microbatch-based pipeline parallelism with small batch size



(c) TeraPipe

Figure 2.2: Execution timeline for different pipelining methods. Grey blocks indicate GPUs idle time (a.k.a. pipeline bubbles). (a) Microbatch-based pipeline parallelism (e.g. GPipe). Each color corresponds to a microbatch. (b) Microbatch-based pipeline parallelism with longer sequence (hence smaller minibatch size due to fixed GPU memory). Pipeline bubbles significantly increase. (c) TeraPipe. Pipeline bubbles are substantially reduced because of the improved pipelining granularity.

2.3.1 Language Modeling and Transformers

The task of language modeling is usually framed as unsupervised distribution estimation of a text corpus $\mathcal{X}$, where each example $x \sim \mathcal{X}$ is a variable length sequence of tokens $(x_1, x_2, \ldots, x_L)$. Since language has a natural sequential ordering, it is common to factorize the joint probability over the tokens as the product of conditional probabilities [a.k.a. autoregressive decomposition; 13]:

$$P(x) = \prod_{t=1}^{L} P(x_t | x_1, \ldots, x_{t-1}). \tag{2.1}$$

Transformer [138] is the state-of-the-art architecture for modeling these conditional probabilities. As visualized in Figure 2.1a, a Transformer-based LM F takes the sequence

$(\langle sos \rangle, x_1, \ldots, x_{L-1})$ as input, where $\langle sos \rangle$ represents the start of a sentence, and outputs a probability distributions p_t at each position t that models the conditional probability $P(x_t|x_1, \ldots, x_{t-1})$ as in Eq. 2.1. In practice, F is stacked with many Transformer layers $F = f_N \circ f_{N-1} \circ \cdots \circ f_1$ [138, 109]: f_1 takes the embedding of the original sequence as input, while f_i $(i > 1)$ takes the output of f_{i-1} as input. The main components of a Transformer layer f contain a *self-attention layer* and a *position-wise feed-forward network layer*:

$$\begin{aligned} &\text{SelfAtt}(h_t; h_1, \ldots, h_{t-1}) = \sum_{s=1}^{t} \alpha_{ts} \cdot (W_V h_s), \\ &\text{where } \alpha_{ts} = \text{softmax}\left(\frac{(W_Q h_t)^\top (W_K h_s)}{\sqrt{H}}\right); \qquad (2.2) \\ &\text{FFN}(h_t) = W_2 \sigma(W_1 h_t + b_1) + b_2. \qquad (2.3) \end{aligned}$$

$h_1, \ldots, h_L \in \mathbb{R}^H$ are hidden states correspond to each position of the input sequence, W and b are learnable parameters, and σ is the nonlinear activation function. An important note here: for each h_t, Eq. 2.2 takes only the hidden states before position t as inputs and Eq. 2.3 only takes h_t as input.

The operation and data dependency in Transformers make it more amenable to parallelization on GPUs/TPUs compared to RNNs [138]. Therefore, Transformers have been scaled to enormous datasets and achieved state-of-the-art performance on a wide range of NLP tasks [138, 34, 109, 152, 18, 84]. Recently, people show that the accuracy of LMs can consistently improve with increasing model sizes [109, 152]. While the growing model size greatly exceeds the memory capacity of a single GPU [18], model parallelism becomes a necessity for training large-scale LMs [125].

2.3.2 Pipeline Parallelism Within a Sequence

In this subsection, we expose the limitations of existing pipelining parallelism approaches, and develop the proposed new pipelining method for Transformer-based LMs.

Typically, to perform pipeline parallelism, a Transformer model F is partitioned into multiple cells $c_1, \ldots, c_K$. Each cell c_k consists of a set of consecutive Transformer layers $f_j \circ \cdots \circ f_{i+1} \circ f_i$ so that $F = c_K \circ \cdots \circ c_2 \circ c_1$. Each c_k is placed and executed on the k-th device (e.g. GPU). The output of cell c_k is sent to cell c_{k+1} during forward propagation, and the backward states computed on cell c_{k+1} is sent to cell c_k during backward propagation. Since each layer f exhibits the same structure, the entire LM can be uniformly partitioned: each cell possesses the same number of layers hence the same amount of computation workload, to reach optimal pipeline efficiency (see Figure 2.2).

However, previous pipelining methods [57, 53] do not perform well on large Transformer-based LMs due to the growing model size. Consider a minibatch of size B. The input to a Transformer layer f is a 3-dimensional tensor $(h^{(1)}, h^{(2)}, \ldots, h^{(B)})$ of size (B, L, H), where L

is the sequence length and H is the hidden state size. To improve accuracy, large LMs are often configured to have a large L to capture longer-term dependency in language sequences [133, 156]. To fit the model into a GPU, the minibatch size B has to decrease accordingly. The pipeline bubbles become larger (Figure 2.2b) because fewer input sequences can be processed in parallel.

In this chapter, we make a key observation: for Transformer-based LMs, with appropriate scheduling, the *token dimension* L can be pipelined for parallel training; and this pipelining dimension is complementary to other model parallelism approaches. Precisely, for an input hidden state sequence $(h_1, h_2, \ldots, h_L)$, the computation of a self-attention layer $\text{SelfAtt}(h_t)$ only depends on the hidden states of previous positions $(h_1, \ldots, h_{t-1})$, and the computation of a feed-forward layer $\text{FFN}(h_t)$ only depends on h_t itself. These offer a new opportunity for pipelining: the computation of layer f_i at step t can commence once the hidden states of previous steps $(< t)$ at f_{i-1} are ready, which, also, can be parallelized with the computation of latter steps at f_{i-1}, illustrated in Figure 2.1d. This property enables us to perform pipeline parallelism within a single input sequence. Specifically, we can split an input sequence $x_1, \ldots, x_L$ into $s_1, \ldots, s_M$, where each subsequence s_i consists of tokens $(x_l, x_{l+1}, \ldots, x_r)$. The computation of $c_1, \ldots, c_K$ over $s_1, \ldots, s_M$ can be pipelined, for example: when c_k computes over s_i, c_{k+1} can process s_{i-1} and c_{k-1} can process s_{i+1} in parallel.

Considering that nowadays LMs operate on sequences with thousands of tokens [109, 18] (e.g. 2048 for GPT-3), the token dimension opens substantial space to improve the pipelining efficiency. However, applying it in practice is still challenging, especially on GPUs, for the following reasons.

First, finer-grained pipelining (i.e. picking a small $|s_i|$) is prone to underutilizing the computational power of GPUs, and thus lowering the training throughput. As shown on the top part of Figure 2.3, for a single layer of the GPT3-1B model (see Table 2.1 for specs), the forward propagation time for an input sequence with a single token is the same as an input sequence with 256 tokens. In this case, the GPU is not being fully utilized for input sequence lengths less than 256. This means a large subsequence length is needed to achieve high throughput for a single layer (see the bottom part of Figure 2.3). On the other hand, although GPUs have better training throughput per layer for longer sequences due to the SIMD architecture and better locality, longer input slices lead to fewer pipeline stages within a sequence, which will increase the pipeline bubble, and thus reduce the pipeline efficiency and hurt the overall training speed.

Second, splitting inputs into multiple same-size chunks for pipelining, as normally done in existing work [57, 53], is not the ideal way for pipelining on the token dimension. For the self-attention layer, the computation of $\text{SelfAtt}(h_1)$ only requires the hidden state h_1 from its previous layer, while the computation of $\text{SelfAtt}(h_L)$ takes all $h_1, \ldots, h_L$ as inputs, as shown in Figure 2.1a. Therefore, the computation load on a later token position in a sequence is heavier than that of previous tokens. Since the total latency of a pipeline is determined

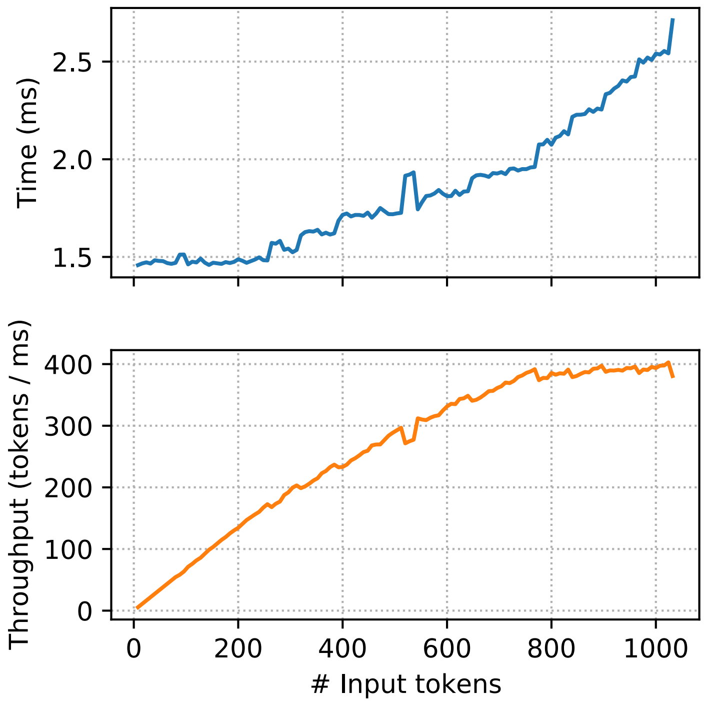


Figure 2.3: Forward propagation time and throughput for a single layer of GPT3-1B model with a single input sequence with different number of input tokens on a single NVIDIA V100 GPU, averaged by 30 independent runs. **Top:** Time per forward propagation. **Bottom:** Throughput measured by number of tokens per millisecond.

by its slowest stage (Figure 2.4), an optimal slicing scheme should have a long slice in the beginning and a shorter slice in the end. We next develop methods to select the optimal slicing scheme over the token dimension.

2.3.3 Selecting Optimal Slicing Scheme

We propose a dynamic programming (DP) algorithm to partition the input sequence to achieve the optimal pipeline efficiency. Specifically, given a partitioned Transformer-based LM $F = c_K \circ \cdots \circ c_1$ and a training input sequence of length L, the goal of the algorithm is to find the *slicing scheme* $l_1, \ldots, l_M$ to minimize the total forward and backward propagation latency, where $l_i = |s_i|$ is the length each sub-sequence slice s_i ($l_1 + \cdots + l_M = L$).

Let's first consider the latency of forward propagation. As shown in Section 2.3.2, all cells c_k have exact same amount of computation.

The forward propagation time t_i for the slice s_i on the cell c_k is determined by the length of

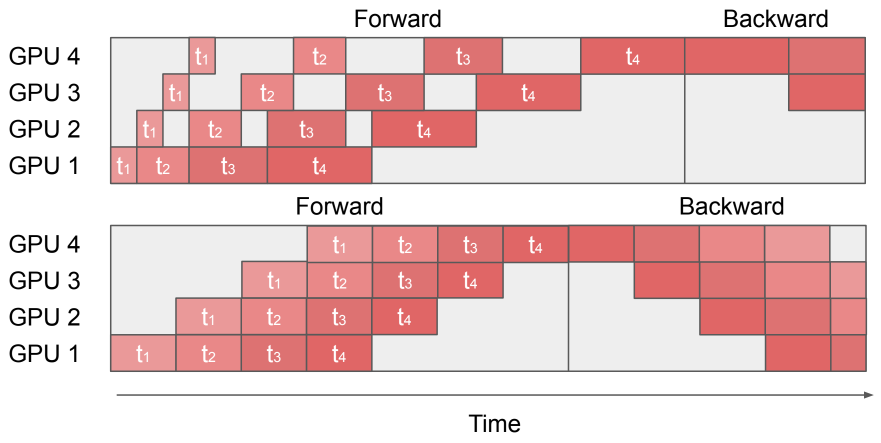


Figure 2.4: Execution timeline for inputs for uniform sequence split with non-uniform running time (top) and non-uniform sequence split with uniform running time (bottom). The total latency of a pipeline is determined by its slowest stage, and thus splits with non-uniform running time result in larger pipeline bubbles and inferior pipeline efficiency.

the ith slice (l_i), the lengths of all the previous subsequences ($l_1, \ldots, l_{i-1}$), and the cluster specifications (e.g., GPU, bandwidth and latency of the underlying computer networks). We use t_{fwd} to denote the sum of the computation latency plus data transmission latency for a given l_i and the previous subsequences $l_1, \ldots, l_{i-1}$. We have:

$$t_i = t_{fwd}\left(l_i, \sum_{j=1}^{i-1} l_j\right). \tag{2.4}$$

Note the second term $\sum_{j=1}^{i-1} l_j$ is the total length of previous subsequences $s_1, \ldots, s_{i-1}$ to compute $\mathrm{SelfAtt}(s_t)$. As visualized in Figure 2.4, The optimal overall pipeline forward propagation latency is:

$$T^* = \min_{l_1,\ldots,l_M}\left\{\sum_{i=1}^{M} t_i + (K-1)\cdot \max_{1\le j\le M}\{t_j\}\right\}. \tag{2.5}$$

The overall latency consists of two terms: The first term here is the total forward propagation time on a device (i.e. on a cell c_k); The second term is the overhead brought by the pipeline execution, which is determined by the slowest component in the whole pipeline multiplied by the number of pipeline stages K minus 1. For example, on the top of Figure 2.4, the total execution time will be $T = (t_1 + \ldots + t_4) + 3t_4$.

Our goal is to find the optimal slicing scheme $l_1, \ldots, l_M$ that achieves the optimal latency T^*. We choose to first enumerate the second term $t_{max} = \max_{1\le j\le M}\{t_j\}$ and minimize the

Algorithm 1 Selecting optimal slicing scheme given t_{max}.

```
Input: Forward propagation time function $t_{fwd}$ and maximum per-slice time $t_{max}$.
Output: Minimal total forward propagation time $S^*(L; t_{max})$ and the corresponding slicing scheme $l_1, \dots, l_M$.
// Dynamic programming for the total forward propagation time.
$S^*(0; t_{max}) \leftarrow 0$
for $i$ from $1$ to $L$ do
    $S^*(i; t_{max}) \leftarrow \min_{1 \le k \le i}\{S^*(i-k; t_{max}) + t_{fwd}(k, i-k) \mid t_{fwd}(k, i-k) \le t_{max}\}$.
    $q_i \leftarrow \arg\min_{1 \le k \le i}\{r_{i-k} + t_{fwd}(k, i-k) \mid t_{fwd}(k, i-k) \le t_{max}\}$.
end for
// Derive the optimal slicing scheme.
$i \leftarrow L, l \leftarrow \{\}$
while $i > 0$ do
    $l.prepend(q_i)$
    $i \leftarrow i - q_i$
end while
```

first term for each different t_{max}. In other words, we reformulate T^* as:

$$T^* = \min_{t_{max}} \left\{S^*(L; t_{max}) + (K-1) \cdot t_{max}\right\}, \tag{2.6}$$

$$S^*(L; t_{max}) = \min_{l_1 + \cdots + l_M = L} \left\{\sum_{i=1}^{M} t_i \mid t_i \le t_{max}\right\}. \tag{2.7}$$

Note that $S^*(\cdot; t_{max})$ has the following optimal substructure:

$$\begin{aligned} S^*(i; t_{max}) = \min_{1 \le k \le i} \{&S^*(i-k; t_{max}) + t_{fwd}(k, i-k) \\ &\mid t_{fwd}(k, i-k) \le t_{max}\}. \end{aligned} \tag{2.8}$$

Therefore, we can get the slicing scheme $l_1, \dots, l_M$ that achieves the total total forward propagation time $S^*(L; t_{max})$ with Algorithm 1. By enumerating all different t_{max}, we can get the optimal slicing scheme that reaches the optimal overall pipeline latency T^*.

Complexity. With our DP algorithm, we can compute the best partition in $O(L^2)$ time for a fixed t_{max}. Note that in total there are at most $O(L^2)$ different choices ($t_{fwd}(i, j)$ for $i, j = 1, \dots, L$) of t_{max}. We therefore can derive the optimal slicing scheme in $O(L^4)$ time.

Optimization. To further accelerate the above DP algorithm, we enumerate different t_{max} from small to large; when $K \cdot t_{max}$ is greater than the current best T, we stop the enumeration since larger t_{max} cannot provide a better slicing scheme. In addition, during enumeration of t_{max}, we only evaluate with t_{max} larger than the last t_{max} by at least ε. In this case, the

Table 2.1: Model settings and parallel training setups used in the evaluation. N: Number of Transformer layers. H: Hidden state size. #Params: Number of total parameters. L: Input sequence length. #GPUs: Total number of GPUs. B: Batch size. #Data: Number of data parallel shards. #Pipe: Number of pipeline stages. #Op: Number of GPUs used for operational partitioning by each Transformer layer.

	Model	N	H	#Params	L	#GPUs	B	#Data	#Pipe	#Op
(1)	GPT3-1B	24	2048	1B	2048	192	128	8	24	1
(2)							72	2	12	8
(3)							72	1	24	8
(4)	GPT3-13B	40	5120	13B	2048	320	32	2	20	8
(5)							32	1	40	8
(6)	GPT3-44B	96	6144	44B	2048	384	8	4	96	1
(7)							8	2	24	8
(8)							8	1	48	8
(9)	GPT3-175B	96	12288	175B	2048	384	2	1	96	4
(10)							2	1	48	8

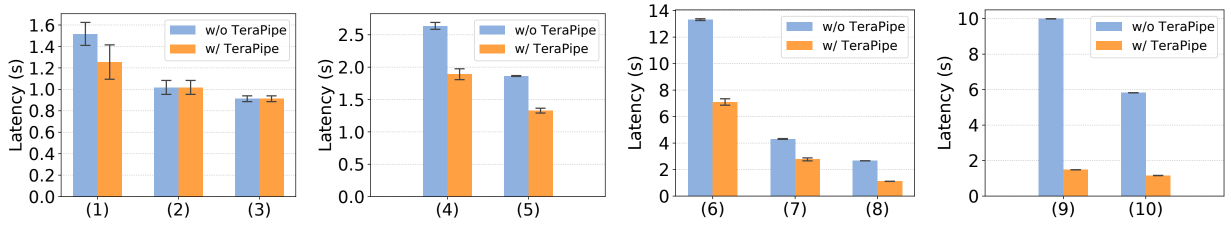
gap between the solution found by the DP algorithm and the global optima is at most $K \cdot \varepsilon$. We choose $\varepsilon = 0.1\,\text{ms}$ in our evaluation and observe that the solution given by Algorithm 1 and the real optimal solution ($\varepsilon = 0$) are always the same in all our evaluated settings. With these two optimizations, the dynamic programming can finish within a minute in our evaluations.

Estimating t_{fwd}. To avoid the cost of evaluating $t_{fwd}(i, j)$ for all $O(L^2)$ combinations of i, j on real clusters, we use a simple performance model to estimate t_{fwd}. Specifically, we split $t_{fwd}(i, j)$ into two terms:

$$t_{fwd}(i, j) = t_{fwd}(i, 0) + t_{ctx}(i, j), \tag{2.9}$$

where $t_{fwd}(i, 0)$ is the forward propagation time without any extra context input and $t_{ctx}(i, j)$ is the latency overhead brought by the extra context input. We measure the first term with all L choices of i and we fit a simple linear model $t_{ctx}(i, j) = a_0 + a_1 i + a_2 j + a_3 ij$ for the second term with a subset of all (i, j) combinations. In our experiments, the linear model can achieve a $< 2\%$ relative prediction error compared to the actual overhead.

The development above can be applied to backward propagation time t_{bwd}, since the backward propagation computation in transformers is symmetric with its forward counterpart. One step further, we can replace all the t_{fwd} above with $t_{fwd} + t_{bwd}$ to derive the optimal slicing scheme that minimizes the total training time.



(a) GPT3-1B (b) GPT3-13B (c) GPT3-44B (d) GPT3-175B

Figure 2.5: Training iteration latency for all configurations with and without TeraPipe. Details for each configuration are listed in Table 2.1.

2.3.4 Combining with Other Parallel Training methods

The new dimension to perform pipeline parallelism proposed by TeraPipe is orthogonal to all previous model parallel techniques, hence can be naturally combined with them. We explain next how TeraPipe can be combined with other parallelization methods and show, when combined, it significantly boosts parallelization performance in Section 2.5.

Combine with microbatch-based pipeline parallelism. To combine with microbatch-based pipeline parallelism [57], we slice the batch dimension and the token dimension jointly to form the pipeline. Specifically, consider a training input batch $(x^{(1)}, x^{(2)}, \ldots, x^{(B)})$, where each $x^{(i)}$ is an input sequence $(x_1^{(i)}, \ldots, x_L^{(i)})$ of length L, we partition the input batch into $(s^{(1)}, s^{(2)}, \ldots, s^{(D)})$, such that each $s_i^{(d)}$ includes $(x_l^{(a)}, x_{l+1}^{(a)}, \ldots, x_r^{(a)})$, $(x_l^{(a+1)}, x_{l+1}^{(a+1)}, \ldots, x_r^{(a+1)})$, $\ldots$, $(x_l^{(b)}, x_{l+1}^{(b)}, \ldots, x_r^{(b)})$, which is the subsequence from position l to r of input data a to b. During training, all slices $s_1^{(1)}, \ldots, s_M^{(1)}, s_1^{(2)}, \ldots, s_M^{(2)}, \ldots, s_1^{(D)}, \ldots, s_M^{(D)}$ can execute on cells $c_1, \ldots, c_K$ in a pipelined fashion. To jointly optimize the sequence slicing and batch splitting, the DP algorithm in Section 2.3.3 can be extended to include the batch dimension: we can first run the whole DP algorithm in Section 2.3.3 for all different batch sizes b from 1 to B. For each b, we derive the optimal T_b and the corresponding slicing scheme s_b. With all T_b and s_b, we only need to determine the size of each slice in the batch dimension $b_1, \ldots, b_D$ such that $b_1 + \cdots + b_D = B$ and $T_{b_1} + \cdots + T_{b_D}$ is minimized. This reduces to a 1D knapsack problem and can be solved using off-the-shelf solvers.

Combine with operation partitioning. TeraPipe is orthogonal from operation partitioning in the sense that: operation partitioning is *intra-operation* parallelism that parallelizes the execution of a single operation, whereas TeraPipe pipelines the execution of different operations. To combine with operation partitioning, we distribute each pipeline parallel cell c_K to a set of target devices and then perform operation partitioning across target devices.

Combine with data parallelism. Similarly, because data parallelism maintains multiple identical copies of the model, we can perform model parallelism for each data parallel model replica and synchronize the gradient updates between the replicas after each forward and backward propagation.

Combine with memory optimization. Same as previous pipeline parallel methods [57], TeraPipe stores the activations of a whole mini-batch in our implementation. TeraPipe can also be combined with various memory optimization techniques, e.g., gradient accumulation [37], rematerialization [23, 62], or memory swapping [114]. See supplementary material for more discussions on combining TeraPipe with gradient accumulation.

2.4 Implementation

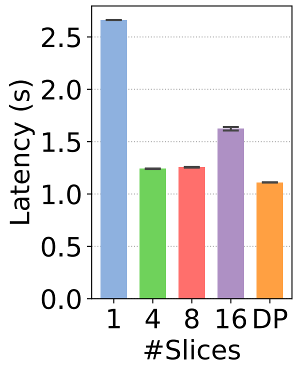
We implement TeraPipe with PyTorch [104] and NCCL [95]. We use Megatron-LM [125] as the library for operation partitioning and implement microbatch-based pipeline parallelism and data parallelism by ourselves. The core of TeraPipe is implemented using 1714 lines of Python. We include the code in the supplementary material and the code will be open-sourced.

2.5 Evaluation

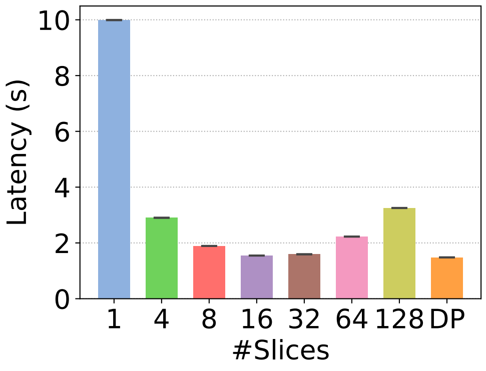
TeraPipe is a synchronous model parallel training method that performs exactly the same underlying optimization algorithm as training the model on a single device. The optimization performance of TeraPipe (i.e. training loss versus training iterations) is hence the same compared to training on a single device. Therefore, in this chapter, we focus on the per-iteration latency (i.e. wall-clock time used per training iteration) as our evaluation metric.

We evaluate TeraPipe following the setup in [18]. Specifically, we test 3 settings in [18]: GPT3-1B, GPT3-13B, and GPT3-175B, which have 1 billion, 13 billion, and 175 billion parameters in total, respectively. Note that GPT3-175B is the largest setting in [18]. In addition, we also test on a GPT3-44B model with half the hidden state size H of the GPT3-175B model, which includes 44 billion parameters in total.

For each model, we select multiple data parallelism, operation partitioning, and pipeline parallelism setup combinations. The configuration details are shown in Table 2.1. For all configurations, we set the input sequence length $L = 2048$ following [18]. We evaluate the configurations on an AWS cluster with p3.16xlarge nodes (each with 8 NVIDIA V100 GPUs). For each model, we select a cluster size based on its model size and number of layers so that each pipeline stage (each cell c_k) has the same number of layers. Since operation partitioning requires higher inter-connection speed compared to pipeline parallelism, we perform operation partitioning only inside a node, where all GPUs have high-speed inter-



(a) GPT3-44B (8)



(b) GPT3-175B (9)

Figure 2.6: Training iteration latency of TeraPipe with uniform slicing scheme with different number of slices and the optimal slicing scheme find by the dynamic programming algorithm.

connection thanks to NVLink. For each configuration, we select the maximal batch size that can fit the memory of the GPUs.

We compare the per-iteration latency achieved by previous model parallel methods without TeraPipe and the latency achieved by TeraPipe for each configuration. Specifically, for the setup without TeraPipe, we measure the training latency with GPipe [57] as the pipeline parallel training method. For TeraPipe, we perform a joint dynamic programming on both batch and token dimension as shown in Section 2.3.4 and measure the training latency with the optimal slicing scheme found by the dynamic programming algorithm. All the latency results in the chapter are averaged over 10 runs. The detailed numbers of the latency results and the solution find by the dynamic programming algorithm can be found in the supplementary material.

2.5.1 Main Results

We show the latency results for all configurations in Figure 2.5. TeraPipe accelerates the training for all models: For GPT3-1B, TeraPipe accelerates training for setting (1) by 1.21x. For setting (2) and (3), because of the large batch size, the optimal slicing scheme found by our dynamic programming algorithm only slices the batch dimension and thus TeraPipe does not provide speedup. For GPT3-13B, TeraPipe speeds up the training by 1.40x for both setting (4) and (5). For GPT3-44B, TeraPipe accelerates the training by 1.88x, 1.56x, and 2.40x for setting (6), (7), and (8), respectively. For GPT3-175B, TeraPipe accelerates the training by 6.75x and 5.02x for setting (9) and (10), respectively.

TeraPipe provides higher speedup for larger models: Larger models have a larger hidden state size H, and a larger portion of GPU memory is devoted to storing the model weights and hidden states. Therefore, the batch size B has to be decreased to fit the model into the GPU memory, as shown in the setup in Table 2.1. Smaller batch size B limits the previous microbatch-based pipeline parallel methods' ability to saturate the pipeline bubbles, while the token dimension used by TeraPipe still provides abundant opportunity to improve pipeline efficiency. In addition, larger models have more pipeline stages compared to smaller models, because larger models have more layers and each layer takes more memory than the smaller models. More pipeline stages require more input slices to saturate the pipeline.

2.5.2 Dynamic Programming

In this subsection, we provide an ablation study on the effectiveness of the dynamic programming algorithm proposed in Section 2.3.3. We compare the training latency with the slicing scheme found by the dynamic programming algorithm, to a simple heuristic that slices the input sequence uniformly. Specifically, we evaluate GPT3-44B with setting (8) and GPT3-175B with setting (9). For the uniform slicing baseline, we slice the whole input on the batch dimension and range the number of slices on the token dimension from 1 to 16 and 1 to 128 for two settings, respectively, and evaluate the iteration latency for each uniform slicing scheme.

The result is shown in Figure 2.6. As in Section 2.3.2, too fine-grained pipeline (e.g. #slices=128 in Figure 2.6b) performs badly because of the underutilization of the GPUs. Also, too coarse-grained pipeline (e.g. #slices=4 in Figure 2.6b) has large pipeline bubbles, which leads to high iteration latency. In addition, because of the non-uniform running time brought by the Transformer structure, the slicing scheme derived by the dynamic programming program achieves better performance compared to the best uniform sliced pipeline: the optimal solutions found by dynamic programming are 1.12x and 1.04x faster compared to the best uniform slicing scheme for GPT3-44B and GPT3-175B model, respectively.

2.5.3 Longer Sequence Length

A growing set of works start to focus on increasing the input sequence length of the Transformers [133, 156, 71]. Long sequence length enables Transformers to reason about long-term dependencies and thus extends its applicability to more complex applications such as modeling documents. However, longer sequences increases the memory usage of a single input sequence, and decreases the maximum batch size allowed, which limits the pipeline efficiency of previous microbatch-based pipeline parallelism methods.

In this subsection, we vary the sequence length from 2048 to 8192 for the GPT3-13B model (setting (5)) and evaluate the training iteration latency. Because of the growth in memory usage, the batch sizes for sequence length 4096, 6144, 8196 are reduced to 8, 4, 2, respectively. We show the results in Figure 2.7. TeraPipe achieves 2.76x, 4.97x, 7.83x speedup for sequence length 4096, 6144, and 8196, respectively. As the sequence length grows, the gap between

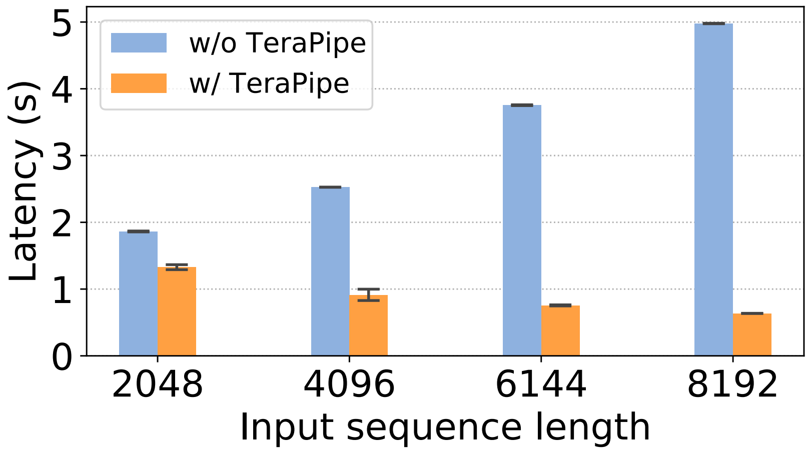


Figure 2.7: Training iteration latency of TeraPipe with different input sequence length for the GPT3-13B model.

the performance with and without TeraPipe significantly increases, as expected. Meanwhile, longer sequence length provides more space on the token dimension and thus TeraPipe can perform even better – TeraPipe enables efficient training of future-emerging LMs with growing sequence lengths.

2.6 Conclusion

We present TeraPipe, a high-performance token-level pipeline parallel algorithm for training large-scale Transformer-based language model. We develop a novel dynamic programming-based algorithm to calculate the optimal pipelining execution scheme, given a specific LM and a cluster configuration. TeraPipe is orthogonal to other model parallel training methods and can be complemented by them. Our evaluations show that TeraPipe accelerates the synchronous training of the largest GPT-3 models with 175 billion parameters by 5.0x on an AWS cluster with 48 p3.16xlarge instances compared to previous methods.

2.7 Appendix: Detailed Experiment Results

Here, we include the detailed numbers (mean and standard deviation of the latency) and the slicing schemes found by the DP algorithms for all experiments in Section 2.5. Specifically, we list the details of Figure 2.5, 2.6, and 2.7 in Table 2.2, 2.3, and 2.4.

Table 2.2: Detailed numbers and slicing schemes in main experiments (Figure 2.5 in the main chapter).

Model	Setting	Algorithm	Slicing Scheme	Latency (s)	TFlops (per GPU)
GPT3-1B	2.5, (1)	w/o TeraPipe	[(1, [2048])] * 16	1.517 ± 0.107	0.8841
		w/ TeraPipe	[(1, [776, 640 ,632])] * 16	1.254 ± 0.160	1.0695
	2.5, (2)	w/o TeraPipe	[(1, [2048])] * 36	1.018 ± 0.065	2.9643
		w/ TeraPipe	[(1, [2048])] * 36	1.018 ± 0.065	2.9643
	2.5, (3)	w/o TeraPipe	[(1, [2048])] * 72	0.913 ± 0.027	6.6105
		w/ TeraPipe	[(1, [2048])] * 72	0.913 ± 0.027	6.6105
GPT3-13B	2.5, (4)	w/o TeraPipe	[(1, [2048])] * 16	2.637 ± 0.055	3.0305
		w/ TeraPipe	[(1, [1024, 1024])] * 16	1.891 ± 0.084	4.2261
	2.5, (5)	w/o TeraPipe	[(1, [2048])] * 32	1.863 ± 0.007	8.5792
		w/ TeraPipe	[(1, [704, 688, 656])] * 32	1.328 ± 0.037	12.0354
GPT3-44B	2.5, (6)	w/o TeraPipe	[(1, [2048])] * 2	13.319 ± 0.067	0.2148
		w/ TeraPipe	[(1, [64] * 26 + [56] * 6 + [48])] * 2	7.103 ± 0.243	0.4028
	2.5, (7)	w/o TeraPipe	[(1, [2048])] * 4	4.311 ± 0.032	1.3274
		w/ TeraPipe	[(1, [368, 384, 384, 368, 256, 288])] * 4	2.771 ± 0.112	2.0652
	2.5, (8)	w/o TeraPipe	[(1, [2048])] * 8	2.662 ± 0.001	4.2995
		w/ TeraPipe	[(1, [384, 384, 368, 320, 296, 296])] * 8	1.111 ± 0.002	10.3018
GPT3-175B	2.5, (9)	w/o TeraPipe	[(1, [2048])] * 2	9.990 ± 0.005	1.1300
		w/ TeraPipe	[(1, [120] * 4 + [112] * 6 + [104] * 8 + [64])] * 2	1.481 ± 0.002	7.6225
	2.5, (10)	w/o TeraPipe	[(1, [2048])] * 2	5.822 ± 0.003	1.9390
		w/ TeraPipe	[(1, [128] * 16)] * 2	1.160 ± 0.001	9.7318

Table 2.3: Detailed numbers and slicing schemes in ablation studies on the effectiveness of the dynamic programming algorithm (Figure 2.6 in the main chapter).

Model	Setting	Algorithm	Slicing Scheme	Latency (s)	TFlops (per GPU)
GPT3-44B	2.6, (a)	#Slices=1	[(1, [2048])] * 8	2.662 ± 0.001	4.2995
		#Slices=4	[(1, [512] * 4)] * 8	1.241 ± 0.003	9.2226
		#Slices=8	[(1, [256] * 8)] * 8	1.255 ± 0.004	9.1197
		#Slices=16	[(1, [128] * 16)] * 8	1.241 ± 0.003	9.2226
		DP	[(1, [384, 384, 368, 320, 296, 296])] * 8	1.111 ± 0.002	10.3018
GPT3-175B	2.6, (b)	#Slices=1	[(1, [2048])] * 2	9.990 ± 0.005	1.1300
		#Slices=4	[(1, [512] * 4)] * 2	2.902 ± 0.003	3.8900
		#Slices=8	[(1, [256] * 8)] * 2	1.892 ± 0.002	5.9667
		#Slices=16	[(1, [128] * 16)] * 2	1.547 ± 0.01	7.2973
		#Slices=32	[(1, [64] * 32)] * 2	1.593 ± 0.002	7.0866
		#Slices=64	[(1, [32] * 64)] * 2	2.227 ± 0.002	5.0691
		#Slices=128	[(1, [16] * 128)] * 2	3.252 ± 0.004	3.4714
		DP	[(1, [120] * 4 + [112] * 6 + [104] * 8 + [64])] * 2	1.481 ± 0.002	7.6225

Table 2.4: Detailed numbers and slicing schemes in experiments with longer sequence lengths (Figure 2.7 in the main chapter).

Model	Input Sequence Length	Algorithm	Slicing Scheme	Latency (s)	TFlops (per GPU)
GPT3-13B	2048	w/o TeraPipe	[(1, [2048])] * 32	1.863 ± 0.007	8.5792
		w/ TeraPipe	[(1, [704, 688, 656])] * 32	1.328 ± 0.037	12.0354
	4096	w/o TeraPipe	[(1, [4096])] * 8	2.526 ± 0.001	1.5819
		w/ TeraPipe	[(1, [552, 536, 528, 512, 504, 496, 488, 480])] * 8	0.913 ± 0.085	4.3765
	6144	w/o TeraPipe	[(1, [6144])] * 4	3.754 ± 0.006	0.5322
		w/ TeraPipe	[(1, [584, 568] + [512] * 6 + [496, 488, 472, 464])] * 4	0.756 ± 0.008	2.6427
	8192	w/o TeraPipe	[(1, [8192])] * 2	4.978 ± 0.004	0.2007
		w/ TeraPipe	[(1, [512] * 6 + [480] * 2 + [416] * 10)] * 2	0.636 ± 0.001	1.5707

Chapter 3

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning

Although algorithmic innovations like TeraPipe (chapter 2) improve the efficiency of model parallel algorithms, there is no universally optimal algorithm for every scenario. Picking the optimal algorithm tightly depends on the model's architecture and the computing cluster's configuration. This is challenging even for system experts, let alone other machine learning researchers and engineers.

Alpa [165] is the world's first system that can automate model-parallel training of large deep learning (DL) models with execution plans that unify data, operator, and pipeline parallelism. Existing model-parallel training systems either require users to manually create a parallelization plan or automatically generate one from a limited space of model parallelism configurations. They do not suffice to scale out complex DL models on distributed compute devices. Alpa distributes the training of large DL models by viewing parallelisms as two hierarchical levels: inter-operator and intra-operator parallelisms. Based on it, Alpa constructs a new hierarchical space for massive model-parallel execution plans. Alpa designs a number of compilation passes to automatically derive efficient parallel execution plans at each parallelism level. Alpa implements an efficient runtime to orchestrate the two-level parallel execution on distributed compute devices. Our evaluation shows Alpa generates parallelization plans that match or outperform hand-tuned model-parallel training systems even on models they are designed for. Unlike specialized systems, Alpa also generalizes to models with heterogeneous architectures and models without manually-designed plans.

3.1 Introduction

Several of the recent advances [57, 18, 125] in deep learning (DL) have been a direct result of significant increases in model size. For example, scaling language models, such as GPT-3, to hundreds of billions of parameters [18] and training on much larger datasets enabled fundamentally new capabilities.

However, training these extremely large models on distributed clusters currently requires a significant amount of engineering effort that is specific to both the model definition and the cluster environment. For example, training a large transformer-based language model requires heavy tuning and careful selection of multiple parallelism dimensions [93]. Training the large Mixture-of-Expert (MoE) transformers model [76, 36] on TPU clusters requires manually tuning the partitioning axis for each layer, whereas training the same model on an AWS GPU cluster calls for new pipeline schemes that can depend on the choices of partitioning (§3.8.1).

More generally, efficient large-scale model training requires tuning a complex combination of data, operator, and pipeline parallelization approaches at the granularity of the individual tensor operators. Correctly tuning the parallelization strategy has been shown [83, 75] to deliver an order of magnitude improvements in training performance, but depends on strong machine learning (ML) and system expertise.

Automating the parallelization of large-scale models would significantly accelerate ML research and production by enabling model developers to quickly explore new model designs without regard for the underlying system challenges. Unfortunately, it requires navigating a complex space of plans that grows exponentially with the dimensions of parallelism and the size of the model and cluster. For example, when all parallelism techniques are enabled, figuring out the execution plan involves answering a web of interdependent questions, such as how many data-parallel replicas to create, which axis to partition each operator along, how to split the model into pipeline stages, and how to map devices to the resulting parallel executables. The interplay of different parallelization methods and their strong dependence on model and cluster setups form a combinatorial space of plans to optimize. Recent efforts [91, 38, 143] to automatically parallelize model training are constrained to the space of a single model-parallelism approach, or rely on strong assumptions on the model and cluster specifications (§3.2.1).

Our key observation is that we can organize different parallelization techniques into a *hierarchical space* and map these parallelization techniques to the *hierarchical structure* of the compute cluster. Different parallelization techniques have different bandwidth requirements for communication, while a typical compute cluster has a corresponding structure: closely located devices can communicate with high bandwidth while distant devices have limited communication bandwidth.

With this observation in mind, in this chapter, we take a different view from conventional data and model parallelisms, and re-categorize ML parallelization approaches as *intra-operator* and *inter-operator* parallelisms. Intra-operator parallelism partitions ML operators along one or more tensor axes (batch or non-batch) and dispatches the partitions to distributed devices (Fig. 3.1c); inter-operator parallelism, on the other hand, slices the model into disjoint stages and pipelines the execution of stages on different sets of devices (Fig. 3.1d). They take place at two different granularities of the model computation, differentiated by whether to partition operators.

Given that, a parallel execution plan can be expressed *hierarchically* by specifying the plan in each parallelism category, leading to a number of advantages. First, intra- and inter-operator parallelisms feature distinct characteristics: intra-operator parallelism has better device utilization, but results in communicating at every split and merge of partitioned operators, per training iteration; whereas inter-operator parallelism only communicates between adjacent stages, which can be light if sliced properly, but incurs device idle time due to scheduling constraints. We can harness the asymmetric nature of communication bandwidth in a compute cluster, and map intra-operator parallelism to devices connected with high communication bandwidth, while orchestrating the inter-operator parallelism between distant devices with relatively lower bandwidth in between. Second, this hierarchical design allows us to solve each level near-optimally as an individual tractable sub-problem. While the joint execution plan is not guaranteed globally optimal, they demonstrate strong performance empirically for training various large models.

Guided by this new problem formulation, we design and implement Alpa, the first compiler that automatically generates parallel execution plans covering all data, operator, and pipeline parallelisms. Given the model description and a cluster configuration, Alpa achieves this by partitioning the cluster into a number of *device meshes*, each of which contains devices with preferably high-bandwidth connections, and partitioning the computation graph of the model into *stages*. It assigns stages to device meshes, and automatically orchestrates intra-operator parallelisms on a device mesh and inter-operator parallelisms between device meshes.

In summary, we make the following contributions:

- We construct a two-level parallel execution plan space (Fig. 3.1e) where plans are specified hierarchically using inter- and intra-operator parallelisms.
- We design tractable optimization algorithms to derive near-optimal execution plans at each level.
- We implement Alpa, a compiler system for distributed DL on GPU clusters. Alpa features: (1) a set of compilation passes that generate execution plans using the hierarchical optimization algorithms, (2) a new runtime architecture that orchestrates the

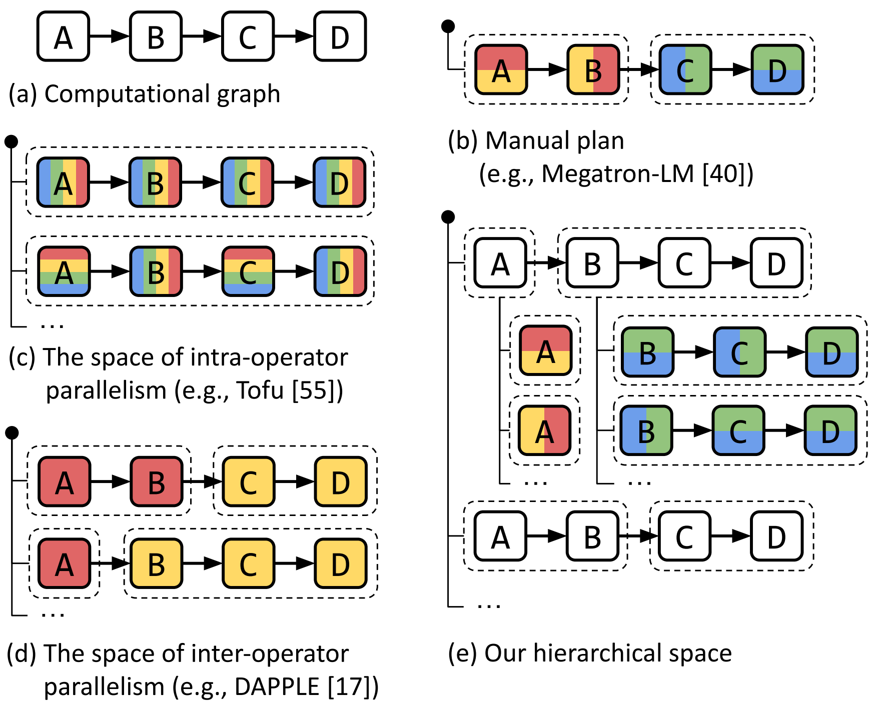

A
B
C
D
(a) Computational graph
(b) Manual plan (e.g., Megatron-LM [40])
(c) The space of intra-operator parallelism (e.g., Tofu [55])
(d) The space of inter-operator parallelism (e.g., DAPPLE [17])
(e) Our hierarchical space
...

Figure 3.1: Generation of parallelization plans for a computational graph shown in (a). Different colors represent different devices, dashed boxes represent pipeline stages. (b) creates the plan manually. (c) and (d) automatically generate plans using only one of intra- and inter-operator parallelisms. (e) shows our approach that creates a hierarchical space to combine intra- and inter-operator parallelisms.

inter-op parallelism between device meshes, and (3) a number of system optimizations that improve compilation and address cross-mesh communication.

- We evaluate Alpa on training large models with billions of parameters. We compare Alpa with state-of-the-art distributed training systems on an Amazon EC2 cluster of 8 p3.16xlarge instances with 64 GPUs. On GPT [18] models, Alpa can match the specialized system Megatron-LM [125, 93]. On GShard MoE models [76], compared to a hand-tuned system Deepspeed[113], Alpa achieves a 3.5× speedup on 2 nodes and a 9.7× speedup on 4 nodes. Unlike specialized systems, Alpa also generalizes to models without manual strategies and achieves an 80% linear scaling efficiency on Wide-ResNet [155] with 4 nodes. This means developers can get efficient model-parallel execution of large DL models out-of-the-box using Alpa.

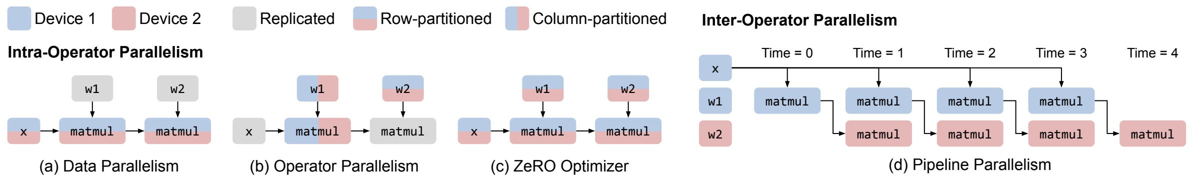


Figure 3.2: Common parallelization techniques for training a 2-layer Multi-layer Perceptron (MLP). Only the forward pass is shown. “x” is the input data. “w1” and “w2” are two weight matrices.

3.2 Background: Distributed Deep Learning

DL computation is commonly represented by popular ML frameworks [1, 104, 17] as a dataflow graph. Edges in the graph represent multi-dimensional tensors; nodes are computational operators, such as matrix multiplication (matmul), that transform input tensors into output tensors. Training a DL model for one iteration consists of computing a loss by *forwarding* a batch of data through the graph, deriving the updates via a reverse *backward* pass, and applying the updates to the parameters via *weight update* operations. In practice, model developers define the dataflow graph. An execution engine then optimizes and executes it on a compute device.

When either the model or data is large that a single device cannot complete the training in a reasonable amount of time, we resort to ML parallelization approaches that parallelize the computation on distributed devices.

3.2.1 Conventional View of ML Parallelism

Existing ML parallelization approaches are typically categorized as data, operator, and pipeline parallelisms.

Data parallelism. In data parallelism, the training data is partitioned across distributed workers, but the model is replicated. Each worker computes the parameter updates on its independent data split, and synchronizes the updates with other workers before the weight update, so that all workers observe consistent model parameters throughout training.

Operator parallelism. When the model is too large to fit in one device, operator parallelism is an effective model parallelism option. Operator parallelism refers to approaches that partition the computation of a specific operator (abbreviated as *op* in the following text), such as matmul shown in Fig. 3.2b, along *non-batch* axes, and compute each part of the operator in parallel across multiple devices.

Because input tensors are jointly partitioned, when a device computes its op partition, the required portions of input tensors may not reside in its local memory. Communication is thus required to fetch the input data from other devices. When the tensors are partitioned evenly, i.e., SPMD[151], all devices will follow the same collective communication patterns such as all-reduce, all-gather, and all-to-all.

Pipeline parallelism. Instead of partitioning ops, pipeline parallelism places different groups of ops from the model graph, referred as *stages*, on different workers; meanwhile, it splits the training batch as a number of microbatches, and pipelines the forward and backward passes across microbatches on distributed workers, as Fig. 3.2d shows. Unlike operator parallelism, pipeline parallelism transfers intermediate activations at the forward and backward passes between different workers using point-to-point communication.

Manual combination of parallelisms. Recent development shows the approaches mentioned above need to be combined to scale out today's large DL models[93, 151]. The state-of-the-art training systems, such as Megatron-LM [125, 93], manually design a specialized execution plan that combines these parallelisms for transformer language models, which is also known as *3D Parallelism*. By assuming the model has the same transformer layer repeated, it assigns an equal number of layers to each pipeline stage and applies a hand-designed operator and data parallelism configuration uniformly for all layers. Despite the requirement of strong expertise, the manual plan cannot generalize to different models or different cluster setups (§3.8.1).

Automatic combination of parallelisms. The configurations of each individual parallelism, their interdependence, and their dependence on model and cluster setups form an intractable space, which prevents the trivial realization of automatically combining these parallelisms. For examples, when coupled with operator parallelism, each time adding a data-parallel replica would require allocating a new set of devices (instead of one single device) as the worker, and figuring out the optimal operator parallelism configurations within those devices. When including pipeline parallelism, the optimal pipelining scheme depends on the data and operator parallelism choices of each pipeline stage and how devices are allocated for each stage. With this conventional view, prior explorations [65, 143, 38, 157] of auto-parallelization are limited to combining data parallelism with at most one model parallelism approach, which misses substantial performance opportunities. We next develop our view of ML parallelisms.

3.2.2 Intra- and Inter-Operator Parallelisms

Different from the conventional view, in this chapter, we re-catalog existing parallelization approaches into two orthogonal categories: intra-operator and inter-operator parallelisms.

They are distinguished by if they involve partitioning operators along any tensor axis. We next use the examples in Fig. 3.2 to introduce the two types of parallelisms.

Intra-operator parallelism. An operator works on multi-dimensional tensors. We can partition the tensor along some dimensions, assign the resulting partitioned computations to multiple devices, and let them execute different portions of the operator at the same time. We define all parallelization approaches using this workflow as intra-operator parallelism.

Fig. 3.2a-c illustrates the application of several typical instantiations of intra-op parallelism on an MLP. Data parallelism [73], by definition, belongs to intra-op parallelism – the input tensors and matmuls are partitioned along the batch dimension, and weight tensors are replicated. Alternatively, when the weights are very large, partitioning the weights (Fig. 3.2b) leads to the aforementioned operator parallelism adopted in Megatron-LM. Besides operators in the forward or backward passes, one can also partition the operators from the weight update phase, yielding the *weight update sharding* or equivalently the ZeRO [111, 150] technique, commonly comprehended as an optimization of data parallelism.

Due to the partitioning, collective communication is required at the split and merge of the operator. Hence, a key characteristic of intra-operator parallelism is that it results in substantial communication among distributed devices.

Inter-operator parallelism. We define inter-operator parallelism as the orthogonal class of approaches that *do not* perform operator partitioning, but instead, assign different operators of the graph to execute on distributed devices.

Fig. 3.2d illustrates the batch-splitting pipeline parallelism as a case of inter-operator parallelism.[1] The pipeline execution can follow different schedules, such as Gpipe [57], PipeDream [91], and synchronous 1F1B [92, 38]. We adopt the synchronous 1F1B schedule throughout this chapter as it respects synchronous consistency, and has the same pipeline latency but lower peak memory usage compared to Gpipe.

In inter-operator parallelism, devices communicate only between pipeline stages, typically using point-to-point communication between device pairs. The required communication volume can be much less than the collective communication in intra-operator parallelism. Regardless of the schedule used, due to the data dependency between stages, inter-operator parallelism results in some devices being idle during the forward and backward computation.

By this categorization, the two parallelisms take place at different granularities of the DL computation and have distinct communication requirements, which happen to match the structure of today's compute clusters. We will leverage these properties to design hierarchical

[1]Device placement [89] is another case of inter-op parallelism, which partitions the model graph and executes them on different devices but does not saturate pipelines using multiple microbatches. Hence pipeline parallelism is often seen as a better alternative to it because of less device idle time.

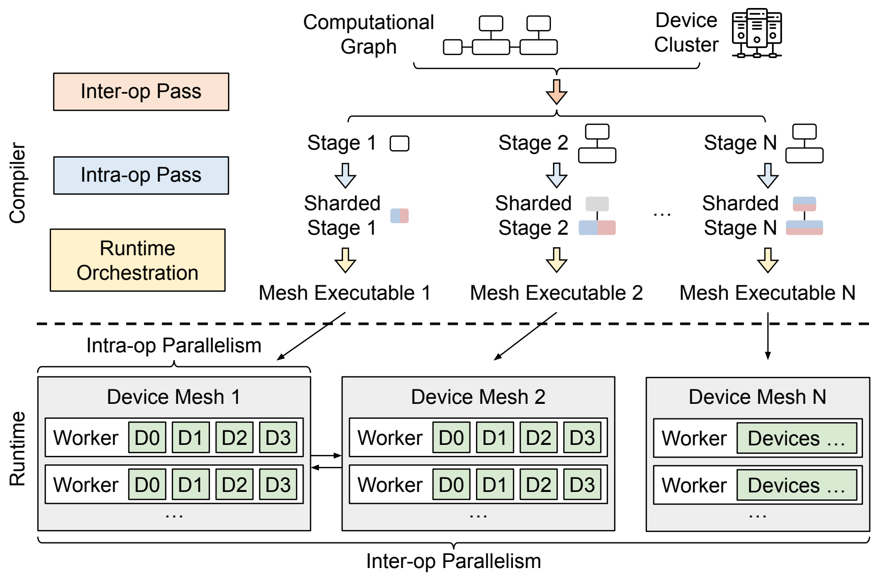


Figure 3.3: Compiler passes and runtime architecture. A sharded stage is a stage annotated with the sharding specs generated by intra-op pass.

algorithms and compilation passes to auto-generate execution plans. Several concurrent work [83, 92, 4, 132] have proposed similar categorization, but Alpa is the first end-to-end system that uses this categorization to automatically generate parallel plans from the full space.

3.3 Overview

Alpa is a compiler that generates model-parallel execution plans by hierarchically optimizing the plan at two different levels: intra-op and inter-op parallelism. At the intra-op level, Alpa minimizes the cost of executing a stage (i.e., subgraph) of the computational graph, with respect to its intra-operator parallelism plan, on a given device mesh, which is a set of devices that may have high bandwidth between each other (e.g., GPUs within a single server). Different meshes might have different numbers of computing devices according to the workload assigned. At the inter-op level, Alpa minimizes the inter-op parallelization latency, with respect to how to slice the model and device cluster into stages and device meshes and how to map them as stage-mesh pairs. The inter-op optimization depends on knowing the execution cost of each stage-mesh pair reported by the intra-op optimizer. Through this hierarchical optimization process, Alpa generates the execution plan consisting of intra-op and inter-op plans which are locally near-optimal at their respective level of the hierarchy.

To achieve this, Alpa implements three novel compilation passes as Fig. 3.3 shows. Given a model description, in the form of a Jax [17] intermediate representation (IR), and a cluster configuration, the inter-op compilation pass slices the IR into a number of stages, and slices

```
# Put @parallelize decorator on top of the Jax functions
@parallelize
def train_step(state, batch):
    def loss_func(params):
        out = state.forward(params, batch["x"])
        return jax.numpy.mean((out - batch["y"]) ** 2)

    grads = grad(loss_func)(state.params)
    new_state = state.apply_gradient(grads)
    return new_state

# A typical training loop
state = create_train_state()
for batch in data_loader:
    state = train_step(state, batch)
```

Figure 3.4: An example to demonstrate Alpa's API for Jax. The developers uses a Python decorator `@parallelize` to annotate functions that need to be parallelized. The rest of the program is kept intact.

the device cluster into a number of device meshes. The inter-op pass uses a Dynamic Programming (DP) algorithm to assign stages to meshes and invokes the intra-op compilation pass on each stage-mesh pair, to query the execution cost of this assignment. Once invoked, the intra-op pass optimizes the intra-op parallel execution plan of the stage running on its assigned mesh, by minimizing its execution cost using an Integer Linear Programming (ILP) formulation, and reports the cost back to the inter-op pass. By repeatedly querying the intra-op pass for each allocation of a stage-mesh pair, the inter-op pass uses the DP to minimize the inter-op parallel execution latency and obtains the best slicing scheme of stages and meshes.

Given the output hierarchical plan and a designated pipeline-parallel schedule, each stage is first compiled as a parallel executable on its located mesh. A runtime orchestration pass is invoked to fulfill the communication requirement between two adjacent stages that require communication between the two meshes they locate on. The runtime orchestration pass then generates static instructions specific to each mesh according to the pipeline-parallel schedule and invokes the execution on all meshes.

API. Alpa has a simple API shown in Fig. 3.4. Alpa requires developers to annotate functions to be parallelized, such as the `train_step()`, using a Python decorator `@parallelize`. Upon the first call to `train_step()`, Alpa traces the whole function to get the model IR, invokes the compilation, and converts the function to a parallel version.

Since the inter-op pass depends on the intra-op pass, in the following text, we first describe

Table 3.1: Sharding specs of a 2-dimentional tensor on a 2×2 device mesh. A is a (N, M) tensor. The device mesh is [[Device 0, Device 1], [Device 2, Device 3]]. Each device stores a partition of A. The first column is the name of the sharding spec. The latter columns use Numpy syntax to describe the partitions stored on each device.

Spec	Device 0	Device 1	Device 2	Device 3
RR	$A[0:N, 0:M]$	$A[0:N, 0:M]$	$A[0:N, 0:M]$	$A[0:N, 0:M]$
S^0S^1	$A[0:\frac{N}{2}, 0:\frac{M}{2}]$	$A[0:\frac{N}{2}, \frac{M}{2}:M]$	$A[\frac{N}{2}:N, 0:\frac{M}{2}]$	$A[\frac{N}{2}:N, \frac{M}{2}:M]$
S^1S^0	$A[0:\frac{N}{2}, 0:\frac{M}{2}]$	$A[\frac{N}{2}:N, 0:\frac{M}{2}]$	$A[0:\frac{N}{2}, \frac{M}{2}:M]$	$A[\frac{N}{2}:N, \frac{M}{2}:M]$
S^0R	$A[0:\frac{N}{2}, 0:M]$	$A[0:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:N, 0:M]$	$A[\frac{N}{2}:N, 0:M]$
S^1R	$A[0:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:N, 0:M]$	$A[0:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:N, 0:M]$
RS^0	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:M]$	$A[0:N, \frac{M}{2}:M]$
RS^1	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:M]$	$A[0:N, 0:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:M]$
$S^{01}R$	$A[0:\frac{N}{4}, 0:M]$	$A[\frac{N}{4}:\frac{N}{2}, 0:M]$	$A[\frac{N}{2}:\frac{3N}{4}, 0:M]$	$A[\frac{3N}{4}:N, 0:M]$
RS^{01}	$A[0:N, 0:\frac{M}{4}]$	$A[0:N, \frac{M}{4}:\frac{M}{2}]$	$A[0:N, \frac{M}{2}:\frac{3M}{4}]$	$A[0:N, \frac{3M}{4}:M]$

the intra-op pass, followed by the inter-op pass, and finally the runtime orchestration pass.

3.4 Intra-Operator Parallelism

Alpa optimizes the intra-operator parallelism plan within a device mesh. Alpa adopts the SPMD-style intra-op parallelism [76, 151] which partitions operators evenly across devices and executes the same instructions on all devices, as per the fact that devices within a single mesh have equivalent compute capability. This SPMD style significantly reduces the space of intra-op parallelism plans; meanwhile, it conveniently expresses and unifies many important approaches such as data parallelism, ZeRO, Megatron-LM's operator parallelism, and their combinations, which are not fully covered by existing automatic operators parallelism systems, such as Tofu [143] and FlexFlow [65]. Unlike systems that perform randomized search [65] or assume linear graphs [143], Alpa formalizes the problem as an integer linear programming (ILP) and shows it can be solved efficiently for computational graphs with tens of thousands of operators. Next, we describe the space of intra-op parallelism and our solution.

3.4.1 The Space of Intra-Operator Parallelism

Given an operator in the computational graph, there are multiple possible parallel algorithms to run it on a device mesh. For example, a matrix multiplication $C_{ij} = \sum_k A_{ik}B_{kj}$ corresponds to a three-level for-loop. To parallelize it, we can parallelize the loop i, loop j,

Table 3.2: Several cases of resharding. $\textit{all-gather}(x, i)$ means an all-gather of x bytes along the i-th mesh axis. M is the size of the tensor. (n_0, n_1) is the mesh shape.

#	Src Spec	Dst Spec	Communication Cost
1	RR	S^0S^1	0
2	S^0R	RR	$\textit{all-gather}(M, 0)$
3	S^0S^1	S^0R	$\textit{all-gather}(\frac{M}{n_0}, 1)$
4	S^0R	RS^0	$\textit{all-to-all}(\frac{M}{n_0}, 0)$
5	S^0S^1	$S^{01}R$	$\textit{all-to-all}(\frac{M}{n_0 \cdot n_1}, 1)$

loop k, or combinations of them across devices, which would have different computation and communication costs, require different layouts for the input tensors, and result in output tensors with different layouts. If an input tensor does not satisfy the layout requirement, a layout conversion is required, which introduces extra communication costs. The goal of the intra-op pass is to pick one parallel algorithm for every operator to minimize the execution time of the entire graph. Next, we formally define the device mesh and the layout of a tensor and discuss the cost of layout conversion.

Device mesh. A device mesh is a 2-dimensional logical view of a set of physical devices. Each device in the mesh has the same compute capability. Devices can communicate along the first mesh dimension and the second mesh dimension with different bandwidths. We assume different groups of devices along the same mesh dimension have the same communication performance. For a set of physical devices, there can be multiple logical views. For example, given 2 nodes and 8 GPUs per node (i.e., 16 devices in total), we can view them as a $2 \times 8, 1 \times 16, 4 \times 4, 8 \times 2$, or 16×1 device mesh. The mapping between physical devices and the logical device mesh view is optimized by the inter-op pass (§3.5). In the rest of this section, we consider one fixed device mesh view.

Sharding Spec. We use *sharding spec* to define the layout of a tensor. For an N-dimensional tensor, its sharding spec is defined as $X_0X_1 \cdots X_{n-1}$, where $X_i \in \{S, R\}$. If $X_i = S$, it means the i-th axis of the tensor is partitioned. Otherwise, the i-th axis is replicated. For example, for a 2-dimensional tensor (i.e., a matrix), SR means it is row-partitioned, RS means it is column-partitioned, SS means it is both row- and column-partitioned. RR means it is replicated without any partitioning. After we define which tensor axes are partitioned, we then have to map the partitioned tensor axes to mesh axes. We only consider 2-dimensional device meshes, so a partitioned tensor axis can be mapped to either the first or the second axis of the device mesh, or both. We added a superscript to S to denote the device assignment. For example, S^0 means the partitions are along the 0-th axis of the mesh, S^{01} means the partitions take place along both mesh axes. S^0R means the tensor is row-partitioned into two parts – The first part is replicated on device 0 and device

Table 3.3: Several parallel algorithms for a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$. The notation *all-reduce*(x, i) means an all-reduce of x bytes along the i-th mesh axis. M is the size of the output tensor. (n_0, n_1) is the mesh shape.

#	Parallel Mapping	Output Spec	Input Specs	Communication Cost
1	$i \to 0, j \to 1$	RS^0S^1	RS^0R, RRS^1	0
2	$i \to 0, k \to 1$	RS^0R	RS^0S^1, RS^1R	*all-reduce*$(\frac{M}{n_0}, 1)$
3	$j \to 0, k \to 1$	RRS^0	RRS^1, RS^1S^0	*all-reduce*$(\frac{M}{n_0}, 1)$
4	$b \to 0, i \to 1$	S^0S^1R	S^0S^1R, S^0RR	0
5	$b \to 0, k \to 1$	S^0RR	S^0RS^1, S^0S^1R	*all-reduce*$(\frac{M}{n_0}, 1)$
6	$i \to \{0, 1\}$	$RS^{01}R$	$RS^{01}R, RRR$	0
7	$k \to \{0, 1\}$	RRR	$RRS^{01}, RS^{01}R$	*all-reduce*$(M, \{0, 1\})$

1, and the second part is replicated on device 2 and device 3. Table 3.1 shows all possible sharding specs of a 2-dimensional tensor on a 2×2 mesh with 4 devices.

Resharding. When an input tensor of an operator does not satisfy the sharding spec of the chosen parallel algorithm for the operator, a layout conversion, namely *resharding*, is required, which might require cross-device communication. Table 3.2 lists several cases of resharding. For instance, to convert a fully replicated tensor to any other sharding specs (case #1), we can slice the tensor locally without communication; to swap the partitioned axis (case #4), we perform an all-to-all.

Parallel algorithms of an operator. With the definitions above, consider parallelizing a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$ on a 2D mesh – Table 3.3 lists several intra-op parallel algorithms for a batched matmul. Algorithm#1 maps loop i to the 0-th mesh axis and loop j to the 1-th mesh axis, resulting in the output tensor C with a sharding spec RS^0S^1. As the LHS operand $A_{b,i,k}$ and RHS operand $B_{b,k,j}$ both have only one parallelized index, their sharding specs are written as RS^0R and RRS^1, respectively. In this algorithm, each device has all its required input tiles (i.e., a partition of the tensor) stored locally to compute its output tile, so there is no communication cost. In Algorithm #2 in Table 3.3, when the reduction loop k is parallelized, all-reduce communication is needed to aggregate the partial sum. Similarly, we can derive the sharding specs and communication costs of other parallel algorithms for a batched matmul.

For other primitive operators such as convolution and reduction, we can get a list of possible parallel algorithms following a similar analysis of their math expressions. In the intra-op pass, the model graph is represented in XLA's HLO format[134], which summarizes common DL operators into less than 80 primitive operators, so we can manually enumerate the possible parallel algorithms for every primitive operator.

3.4.2 ILP Formulation

The total execution cost of a computational graph $G = (V, E)$ is the sum of the compute and communication costs on all nodes $v \in V$ and the resharding costs on all edges $e \in E$. We formulate the cost minimization as an ILP and solve it optimally with an off-the-shelf solver [42].

For node v, the number of possible parallel algorithms is k_v. It then has a communication cost vector c_v of length k_v, or $c_v \in \mathbb{R}^{k_v}$, where c_{vi} is the communication cost of the i-th algorithm. Similarly, node v has a compute cost vector $d_v \in \mathbb{R}^{k_v}$. For each node v, we define an one-hot decision vector $s_v \in \{0, 1\}^{k_v}$ to represent the algorithm it uses. $s_{vi} = 1$ means we pick the i-th algorithm for node v. For the resharding cost between node v and node u, we define a resharding cost matrix $R_{vu} \in \mathbb{R}^{k_v \times k_u}$, where R_{vuij} is the resharding cost from the output of i-th strategy of node v to the input of j-th strategy of node u. The objective of the problem is

$$\min_s \sum_{v \in V} s_v^\intercal (c_v + d_v) + \sum_{(v,u) \in E} s_v^\intercal R_{vu} s_u, \tag{3.1}$$

where the first term is the compute and communication cost of node v, and the second is the resharding cost of the edge (v, u). In this formulation, s is the variable, and the rest are constant values. The term $s_v^\intercal R_{vu} s_u$ in Eq. 3.1 is quadratic, and cannot be fed into an ILP solver. We linearize [43] the quadratic term by introducing a new decision vector $e_{vu} \in \{0, 1\}^{k_v \cdot k_u}$ which represents the resharding decision between node v and u.

Although we can use profiling to get the accurate costs for c_v, d_v, and R_{vu}, we use the following methods to estimate them for simplicity. For communication costs c_v and R_{vu}, we compute the numbers of communicated bytes and divide them by the mesh dimension bandwidth to get the costs. For compute costs d_v, we set all of them as zero following the same motivation in [143]. This is reasonable because: (1) For heavy operators such as matmul, we do not allow replicated computation. All parallel algorithms always evenly divide the work to all devices, so all parallel algorithms of one operator have the same arithmetic complexity; (2) For lightweight operators such as element-wise operators, we allow replicated computation of them, but their computation costs are negligible.

To simplify the graph, we merge computationally-trivial operators, such as element-wise operators, transpose, and reduction, into one of their operands and propagate the sharding spec from the operand. This greatly reduces the number of nodes in the graph, thus the ILP problem size. We do a breath-first-search and compute the depth of each node. The node is merged to the deepest operand.

Once the parallel plan is decided by ILP, we also apply a set of post-ILP communication optimizations, such as replacing all-reduce with reduce-scatter and all-gather, whenever applicable, because the latter reduces the number of replicated tensors and corresponding computations, while keeping the communication volume the same. This achieves the effect of weight update sharding [150] or ZeRO optimizer [111].

3.5 Inter-Operator Parallelism

In this section, we develop methods to slice the model and device cluster into stage-mesh pairs. Our optimization goal is to minimize the *end-to-end* pipeline execution latency for the entire computational graph. Previous works [83, 38] have considered simplified problems, such as assuming the device for each stage is pre-assigned, and all stages have fixed data or operator parallelism plan. Alpa rids these assumptions by jointly considering device mesh assignment and the existence of varying intra-op parallelism plans on each stage.

3.5.1 The Space for Inter-Operator Parallelism

Assume the computational graph contains a sequence of operators following the topology order of the graph[2], notated as $o_1, \ldots, o_K$, where the inputs of an operator o_k are from operators $o_1, \ldots, o_{k-1}$. We slice the operators into S stages $s_1, \ldots, s_S$, where each stage s_i consists of operators $(o_{l_i}, \ldots, o_{r_i})$, and we assign each stage s_i to a submesh of size $n_i \times m_i$, sliced from a computer cluster that contains devices, notated as the *cluster mesh* with shape $N \times M$. Let $t_i = t_{intra}(s_i, Mesh(n_i, m_i))$ be the latency of executing stage s_i on a submesh of $n_i \times m_i$, minimized by the ILP and reported back by the intra-op pass (§3.4). As visualized in Fig. 3.5, assuming we have B different input microbatches for the pipeline, the total minimum latency[3] for the entire computation graph is written as:

$$T^* = \min_{\substack{s_1,\ldots,s_S;\\(n_1,m_1),\ldots,(n_S,m_S)}} \left\{ \sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\} \right\}. \tag{3.2}$$

The overall latency contains two terms: the first term is the total latency of all stages, interpreted as the latency of the first microbatch going through the pipeline; the second term is the pipelined execution time for the rest of $B-1$ microbatches, which is bounded by the slowest stage (stage 3 in Fig. 3.5).

We aim to solve Eq. 3.2 with two additional constraints: (1) For an operator in the forward pass of the graph, we want to colocate it with its corresponded backward operator on the same submesh. Since backward propagation usually uses the similar set of tensors during forward propagation, this effectively reduces the amount of communication to fetch the required tensors generated at the forward pass to the backward pass. We use the sum of forward and backward latency for t_{intra}, so Eq. 3.2 reflects the total latency, including both

[2]We simply use the order of how users define each operator, reflected in the model IR, with the input operator as the origin. This allows us to leverage the inherent locality present in the user's program – closely related nodes in the graph will be more likely to be partitioned into the same stage.

[3]This formulation holds for GPipe and synchronous 1F1B schedules. Other pipeline schedules may require a different formulation.

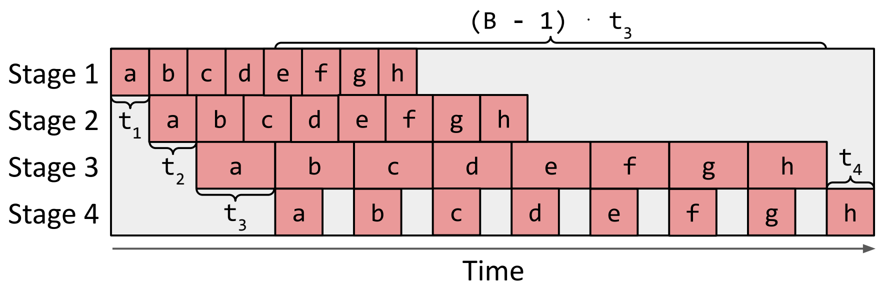


Figure 3.5: Illustration of the total latency of a pipeline, which is determined by two parts: the total latency of all stages ($t_1+t_2+t_3+t_4$) and the latency of the slowest stage ($(B-1)\cdot t_3$).

forward and backward propagation. (2) We need the sliced submeshes $(n_1, m_1), \ldots, (n_S, m_S)$ to fully cover the $N \times M$ cluster mesh – we do not waste any compute device resources. We next elaborate on our DP formulation.

3.5.2 DP Formulation

To ensure all submeshes $(n_1, m_1), \ldots, (n_S, m_S)$ fully cover the $N \times M$ cluster mesh, we reduce the available submesh shapes into two options: (1) one-dimensional submeshes of sizes $(1, 1), (1, 2), (1, 4) \ldots (1, 2^m)$ and (2) two-dimensional submeshes of size $(2, M), (3, M), \ldots, (N, M)$ that fully use the second dimension of the cluster mesh (i.e., on a GPU cluster, this means using all compute devices in each physical machine). We include a theorem in Appendix 3.11 that proves these submesh shapes can always fully cover the cluster mesh. To assign physical devices in the cluster to the resulting submeshes find by the DP algorithm, we enumerate by assigning devices to larger submeshes first and then to smaller ones. When there are multiple pipeline stages with the same submesh shape, we tend to put neighboring pipeline stages closer on the device mesh to reduce communication latency.

The simplification on submesh shapes works well for most available cloud deep learning setups: On AWS [5], the GPU instances have 1, 2, 4, or 8 GPUs; on GCP [48], the TPU instances have 8, 32, 128, 256 or 512 TPUs. The set of submesh shapes (n, m) excluded by the assumption is with $n > 1$ and $m < M$, which we observe lead to inferior results, since an alternative submesh with shape (n', M) where $n' \cdot M = n \cdot m$ has more devices that can communicate with high bandwidth. With this reduction, we only need to ensure that $\sum_{i=1}^{S} n_i \cdot m_i = N \cdot M$.

To find T^* in Eq. 3.2, we develop a DP algorithm. The DP first enumerates the second term $t_{max} = \max_{1\leq j\leq S} t_j$ and minimizes the first term $t_{total}(t_{max}) = \sum_{1\leq i\leq S} t_i$ for each different t_{max}. Specifically, we use the function $F(s, k, d; t_{max})$ to represent the minimal total latency when slicing operators o_k to o_K into s stages and putting them onto d devices so that the latency of each stage is less than t_{max}. We start with $F(0, K+1, 0; t_{max}) = 0$, and derive the optimal substructure of F as

$$F(s,k,d;t_{max}) = \min_{\substack{k\le i\le K \\ n_s\cdot m_s\le d}} \left\{ \begin{array}{l} t_{intra}((o_k,\ldots,o_i), Mesh(n_s,m_s), s) \\ \quad + F(s-1, i+1, d-n_s\cdot m_s; t_{max}) \\ \mid t_{intra}((o_k,\ldots,o_i), Mesh(n_s,m_s), s) \le t_{max} \end{array} \right\}, \tag{3.3}$$

and derive the optimal total latency as

$$T^*(t_{max}) = \min_s\{F(s,0,N\cdot M;t_{max})\} + (B-1)\cdot t_{max}. \tag{3.4}$$

The value of $t_{intra}((o_k,\ldots,o_i), Mesh(n_s,m_s), s)$ is determined by the intra-op pass. It is the lowest latency of executing the subgraph $(o_k,\ldots,o_i)$ on mesh $Mesh(n_s,m_s)$ with s subsequent stages. Note that $Mesh(n_s,m_s)$ is a set of physical devices – hence, we enumerate all the potential choices of logical device mesh shapes (n_l,m_l) satisfying $n_l\cdot m_l = n_s\cdot m_s$. For each choice, we query the intra-op pass with subgraph $(o_k,\ldots,o_i)$, logical mesh (n_l,m_l), and other intra-op options as inputs and get an intra-op plan. We then compile the subgraph with this plan and all other low-level compiler optimizations (e.g., fusion, memory planning) to get an executable for precise profiling. The executable is profiled in order to get the stage latency (t_l) and the memory required on each device to run the stage (mem_{stage}) and to store the intermediate activations (mem_{act}). We check whether the required memory fits the device memory (mem_{device}) according to the chosen pipeline execution schedule. For example, for 1F1B schedule [38, 92], we check

$$mem_{stage} + s\cdot mem_{act} \le mem_{device}. \tag{3.5}$$

We pick the logical mesh shape that minimizes t_l and fits into the device memory. If none of them fits, we set $t_{intra} = \infty$.

Our algorithm builds on top of that in TeraPipe [83]. However, TeraPipe assumes all pipeline stages are the same, and the goal is to find the optimal way to batch input tokens into micro-batches of different sizes. Instead, Alpa aims to group the operators of a computational graph into different pipeline stages, while assuming the input micro-batches are of the same size. In addition, Alpa optimizes the mesh shape in the DP algorithm for each pipeline stage in inter-op parallelism.

Complexity. Our DP algorithm computes the slicing in $O(K^3NM(N+\log(M)))$ time for a fixed t_{max}. t_{max} has at most $O(K^2(N+\log(M)))$ choices: $t_{intra}((o_i,\ldots,o_j), Mesh(n_s,m_s))$ for $i,j=1,\ldots,K$ and all the submesh choices. The complexity of this DP algorithm is thus $O(K^5NM(N+\log(M))^2)$.

This complexity is not feasible for a large computational graph of more than ten thousand operators. To speed up this DP, we introduce a few practical optimizations.

Algorithm 2 Inter-op pass summary.

```
1: Input: Model graph $G$ and cluster $C$ with shape $(N, M)$.
2: Output: The minimal pipeline execution latency $T^*$.
3: // Preprocess graph.
4: $(o_1, \ldots, o_K) \leftarrow \text{Flatten}(G)$
5: $(l_1, \ldots, l_L) \leftarrow \text{OperatorClustering}(o_1, \ldots, o_K)$
6: // Run the intra-op pass to get costs of different stage-mesh pairs.
7: $submesh\_shapes \leftarrow \{(1,1),(1,2),(1,4),\ldots,(1,M)\} \cup \{(2,M),(3,M),\ldots,(N,M)\}$
8: for $1 \le i \le j \le L$ do
9:     $stage \leftarrow (l_i, \ldots, l_j)$
10:    for $(n, m) \in submesh\_shapes$ do
11:        for $s$ from 1 to $L$ do
12:            $t\_intra(stage, Mesh(n,m), s) \leftarrow \infty$
13:        end for
14:        for $(n_l, m_l), opt \in \text{LogicalMeshShapeAndIntraOp Options}(n, m)$ do
15:            $plan \leftarrow \text{IntraOpPass}(stage, Mesh(n_l, m_l), opt)$
16:            $t_l, mem_{stage}, mem_{act} \leftarrow \text{Profile}(plan)$
17:            for $s$ satisfies Eq. 3.5 do
18:                if $t_l < t\_intra(stage, Mesh(n,m), s)$ then
19:                    $t\_intra(stage, Mesh(n,m), s) \leftarrow t_l$
20:                end if
21:            end for
22:        end for
23:    end for
24: end for
25: // Run the inter-op dynamic programming
26: $T^* \leftarrow \infty$
27: for $t_{max} \in \text{SortedAndFilter}(t\_intra, \varepsilon)$ do
28:    if $B \cdot t_{max} \ge T^*$ then
29:        break
30:    end if
31:    $F(0, L+1, 0; t_{max}) \leftarrow 0$
32:    for $s$ from 1 to $L$ do
33:        for $l$ from $L$ down to 1 do
34:            for $d$ from 1 to $N \cdot M$ do
35:                Compute $F(s, l, d; t_{max})$ according to Eq. 3.3
36:            end for
37:        end for
38:    end for
39:    $T^*(t_{max}) \leftarrow \min_s\{F(s, 0, N \cdot M; t_{max})\} + (B-1) \cdot t_{max}$
40:    if $T^*(t_{max}) < T^*$ then
41:        $T^* \leftarrow T^*(t_{max})$
42:    end if
43: end for
```

Performance optimization #1: early pruning. We use one optimization that is similar to that in TeraPipe [83]. We enumerate t_{max} from small to large. When $B \cdot t_{max}$ is larger than the current best T^*, we immediately stop the enumeration. This is because larger t_{max} can no longer provide a better solution. Also, during enumeration of t_{max}, we only evaluate a choice of t_{max} if it is sufficiently larger than the last t_{max} (by at least ϵ). This allows the gap between the solution found by the DP algorithm and the global optima to be at most $B \cdot \epsilon$. We empirically choose $\epsilon = 10^{-6}$ s, and we find that the solution output by our algorithm is the same as the real optimal solution ($\epsilon = 0$) for all our evaluated settings.

Performance optimization #2: operator clustering. Many operators in a computational graph are not computationally intensive (e.g., ReLU), and the exact placement of these operators has little impact on the total execution time. We develop another DP algorithm [6] to cluster neighboring operators to reduce the total size of the graph Eq. 3.2 works on. We cluster the operators $(o_1, \dots, o_K)$ into a series of layers[4] $(l_1, \dots, l_L)$, where $L \ll K$. The goal of the algorithm is to merge two types of operators: (1) those that do not call for much computation but lengthen the computational graph and (2) neighboring operators that may cause substantial communication if put on different device meshes. We define function $G(k, r)$ as the minimum of maximal amount of data received by a single layer when clustering operators $(o_1, \dots, o_k)$ into r layers. Note that G has the following optimal substructure:

$$\begin{aligned} &G(k, r) \\ &= \min_{1 \le i \le k} \left\{ \begin{array}{l} \max\{G(i-1, r-1), C(i, k)\} \\ \middle| \; FLOP(o_i, \dots, o_k) \le \dfrac{(1+\delta) FLOP_{total}}{L} \end{array} \right\}, \end{aligned} \tag{3.6}$$

where $C(i, k)$ denotes the total size of inputs of $(o_i, \dots, o_k)$ received from $(o_1, \dots, o_{i-1})$ and $FLOP_{total} = FLOP(o_1, \dots, o_K)$ is the total FLOP of the whole computational graph. We make sure that each clustered layer's FLOP is within $1 + \delta$ times of the average FLOP per layer while minimizing the communication. For the solutions with the same communication cost, we choose the one with the most uniform structure by also minimizing the variance of per-layer FLOP. With our DP algorithm, we can compute the best layer clustering in $O(K^2 L)$ time. Note that L here is a hyperparameter to the algorithm. In practice, we choose a small L based on the number of devices and the number of heavy operators in the graph. We find different choices of L do not affect the final performance significantly.

Alg. 2 summarizes the workflow of the inter-op pass and illustrates its interactions with the intra-op pass in §3.4.

[4]Note that the clustering does not exactly reproduce the layers with original machine learning semantics in the model definition.

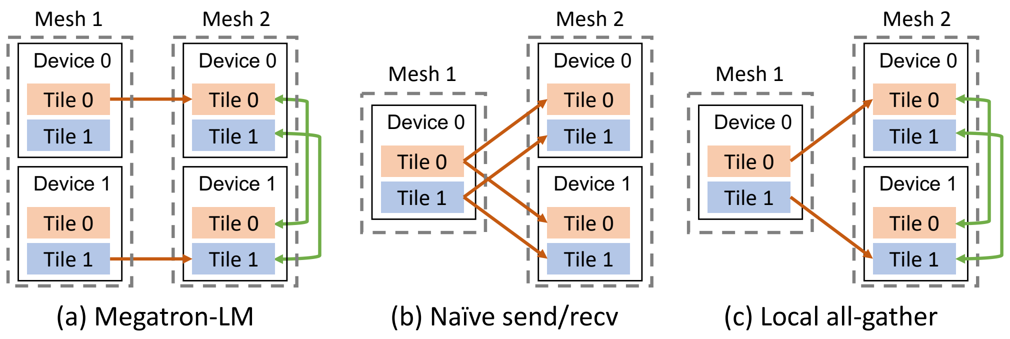


Figure 3.6: Cross-mesh resharding. Red arrows denote send/recv on slow connections. Green arrows denote all-gather on fast connections. (a) The scatter-gather optimization for equal mesh shapes in Megatron-LM. (b) The naive send/recv for unequal mesh shapes. (c) The generalized local all-gather optimization for unequal mesh shapes.

3.6 Parallelism Orchestration

After stages, device meshes, and their assignments are decided, at the intra-op level, Alpa compiles each stage against its assigned device mesh, respecting the intra-op parallelism plan output by the ILP solver. The compilation depends on XLA [134] and GSPMD [151], and generates parallel executables for each stage-mesh pair. When needed, the compilation automatically inserts collective communication primitives (see §3.4) to address the *within-mesh* communication caused by intra-op parallelism.

At the inter-op level, Alpa implements an additional parallelism orchestration pass to address the *cross-mesh* communication between stages, and generate static instructions for inter-op parallel execution.

Cross-mesh resharding. Existing manual systems, such as Megatron-LM [113, 125], constrain all pipeline stages to have the same degrees of data and tensor model parallelism, so the communication between pipeline stages is trivially realized by P2P send/recv between corresponded devices of two equivalent device meshes (Fig. 3.6a). In Alpa, the device meshes holding two adjacent stages might have different mesh shapes, and the tensor to communicate between two stages might have different sharding specs (Fig. 3.6b and Fig. 3.6c). We call this communication pattern as *cross-mesh resharding*, which is a many-to-many multicast problem.

Given the sharding specs of the tensor on the sender and receiver mesh, Alpa generates a communication plan to address cross-mesh sharding in two iterations. In the first iteration, Alpa calculates the correspondences between tensor partitions (a.k.a. tiles) on the source and destination mesh. Based on that, it generates P2P send/recv primitives between the source devices and destination devices to fulfill the communication. It then takes a second iteration to identify opportunities where the destination tensor has a replication in its sharding spec. In this case, the tensor only needs to be transferred once between two meshes, then exchanged via all-gather across the devices on the destination mesh using its higher bandwidth (Fig. 3.6) – it rewrites send/recv generated at the first iteration into all-gather to avoid repeated communication.

We call this approach as *local all-gather* cross-mesh resharding. Since the communication between stages is normally small by our design, our experiments show that it performs satisfactorily well (§3.8.5). We defer the development of the optimal cross-mesh resharding plan to future work.

Generating pipeline execution instructions. As the final step, Alpa generates static execution instructions to launch the training on clusters. Since each stage has different sets of operators and may locate on meshes with different shapes, in contrast to many SPMD pipeline-parallel training systems [151, 93], Alpa adopts an MPMD-style runtime to orchestrate the inter-op parallel execution – Alpa generates distinct static execution instructions for each device mesh.

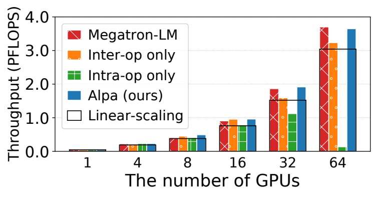
Alpa develops a set of instructions for inter-op parallel execution, including instructions for allocating and deallocating memory for tensors in a stage, communicating tensors between stages following the cross-mesh resharding plan, synchronization, and computation, etc. According to a user-selected pipeline schedule, Alpa uses a driver process to generate the instructions in advance and dispatches the whole instruction lists to each worker before execution, avoiding driver-worker coordination overheads during runtime.

3.7 Limitations and Discussion

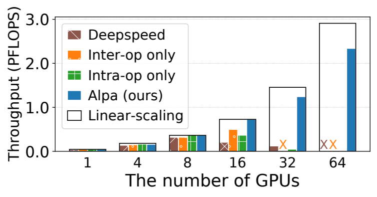
In this section, we discuss advantages of our view of parallelisms and several limitations of our algorithms.

Compared to existing work that manually combines data, operator, and pipeline parallelism, such as 3D parallelism [113] and PTD-P [93], Alpa's hierarchical view of inter- and intra-op parallelisms significantly advances them with three major flexibility: (1) pipeline stages can contain an uneven number of operators or layers; (2) pipeline stages in Alpa might be mapped to device meshes with different shapes; (3) within each stage, the data and operator parallelism configuration is customized non-uniformly on an operator-by-operator basis. Together, they allow Alpa to unify all existing model parallelism approaches and generalize to model architectures and cluster setups with more heterogeneity.

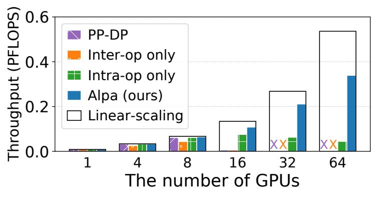
Despite these advantages, Alpa's optimization algorithms currently have a few limitations:



(a) GPT



(b) MoE



(c) Wide-ResNet

Figure 3.7: End-to-end evaluation results. "$\times$" denotes out-of-memory. Black boxes represent linear scaling.

- Alpa does not model the communication cost between different stages because the cross-stage communication cost is *by nature small*. In fact, modeling the cost in either the DP or ILP is possible, but would require enumerating exponentially more intra-op passes and DP states.
- The inter-op pass currently has a hyperparameter: the number of micro-batches B, which is not optimized by our current formulation but can be searched by enumeration.
- The inter-op pass models pipeline parallelism with a static linear schedule, without considering more dynamic schedules that, for example, parallelize different branches in a computational graph on different devices.
- Alpa does not optimize for the best scheme of overlapping computation and communication; Alpa can only handle static computational graphs with all tensor shapes known at compilation time.

Nevertheless, our results on weak scaling (§3.8) suggest that Alpa is able to generate near-optimal execution plans for many notable models.

3.8 Evaluation

Alpa is implemented using about 16K LoC in Python and 6K LoC in C++. Alpa uses Jax as the frontend and XLA as the backend. The compiler passes are implemented on Jax's and XLA's intermediate representation (i.e., Jaxpr and HLO). For the distributed runtime, we use Ray [90] actor to implement the device mesh worker, XLA runtime for executing computation, and NCCL [95] for communication.

We evaluate Alpa on training large-scale models with billions of parameters, including GPT-3 [18], GShard Mixture-of-Experts (MoE) [76], and Wide-ResNet [155]. The testbed is a typical cluster consisting of 8 nodes and 64 GPUs. Each node is an Amazon EC2 p3.16xlarge instance with 8 NVIDIA V100 16 GB GPUs, 64 vCPUs, and 488 GB memory. The 8 GPUs

Table 3.4: Models used in the end-to-end evaluation. LM = language model. IC = image classification.

Model	Task	Batch size	#params (billion)	Precision
GPT-3 [18]	LM	1024	0.35, 1.3, 2.6, 6.7, 15, 39	FP16
GShard MoE [76]	LM	1024	0.38, 1.3, 2.4, 10, 27, 70	FP16
Wide-ResNet [155]	IC	1536	0.25, 1.0, 2.0, 4.0, 6.7, 13	FP32

in a node are connected via NVLink. The 8 nodes are launched within one placement group with 25Gbps cross-node bandwidth.

We compare Alpa against two state-of-the-art distributed systems for training large-scale models on GPUs. We then isolate different compilation passes and perform ablation studies of our optimization algorithms. We also include a case study of the execution plans found by Alpa.

3.8.1 End-to-End Performance

Models and training workloads. We target three types of models listed in Table 3.4, covering models with both homogeneous and heterogeneous architectures. GPT-3 is a homogeneous transformer-based LM by stacking many transformer layers whose model parallelization plan has been extensively studied [125, 93]. GShard MoE is a mixed dense and sparse LM, where mixture-of-experts layers are used to replace the MLP at the end of a transformer, every two layers. Wide-ResNet is a variant of ResNet with larger channel sizes. It is vastly different from the transformer models and there are no existing manually designed strategies.

To study the ability to train large models, we follow common ML practice to scale the model size along with the number of GPUs, with the parameter range reported in Table 3.4. More precisely, for GPT-3, we increase the hidden size and the number of layers together with the number of GPUs following [93], whereas for MoE we mainly increase the number of experts suggested by [76, 151]. For Wide-ResNet, we increase the channel size and width factor in convolution layers. For each model, we adopt the suggested global batch size per ML practice [93, 18, 76, 155] to keep the same statistical behavior. We then tune the best micro-batch size for each model and system configuration that maximizes the system performance. The gradients are accumulated across microbatches. The detailed model specifications are provided in Appendix 3.12.

Baselines. For each model, we compare Alpa against a strong baseline. We use Megatron-LM v2 [93] as the baseline system for GPT-3. Megatron-LM is the state-of-the-art system for training homogeneous transformer-based LMs on GPUs. It combines data parallelism, pipeline parallelism, and manually-designed operator parallelism (denoted as TMP later). The combination of these techniques is controlled by three integer parameters that specify the parallelism degrees assigned to each technique. We grid-search the three parameters following the guidance of their paper and report the results of the best configuration. Megatron-LM is specialized for GPT-like models, so it does not support other models in Table 3.4.

We use DeepSpeed [113] as the baseline for MoE. DeepSpeed provides a state-of-the-art implementation for training MoE on GPUs. It combines handcrafted operator parallelism for MoE layers and ZeRO-based [111] data parallelism. The combination of these techniques is controlled by several integer parameters that specify the parallelism degree assigned to each technique. We also grid-search them and report the best results. The performance of DeepSpeed on GPT-3 is similar to or worse than Megatron-LM, so we skip it on GPT-3. Note that original GShard-MoE [76] implementation is only available on TPUs, thus we do not include its results, though their strategies [76] are covered by Alpa's strategy space.

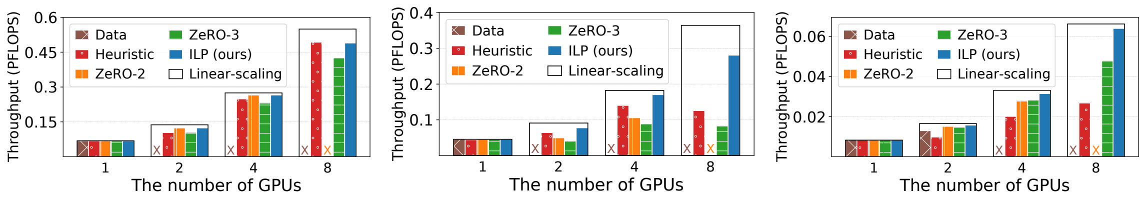
For large Wide-ResNet, there is no specialized system or manually designed plan for it. We use Alpa to build a baseline "PP-DP" whose space only consists of data parallelism and pipeline parallelism, which mimics the parallelism space of PipeDream [91] and Dapple [38].

For all models, we also include the results of using Alpa with only one of intra- and inter-operator parallelism, which mimics the performance of some other auto-parallel systems. The open-source Flexflow [65] does not support the models we evaluate, as it lacks support for many necessary operators (e.g., layer normalization [7], mixed-precision operators). Tofu [143] only supports single node execution and is not open-sourced. Due to both theoretical and practical limitations, we do not include their results and we do not expect Flexflow or Tofu to outperform the state-of-the-art manual baselines in our evaluation.

Evaluation metrics. Alpa does not modify the semantics of the synchronous gradient descent algorithm, thus does not affect the model convergence. Therefore, we measure training throughput in our evaluation. We evaluate weak scaling of the system when increasing the model size along with the number of GPUs. Following [93], we use the aggregated peta floating-point operations per second (PFLOPS) of the whole cluster as the metric[5]. We measure it by running a few batches with dummy data after proper warmup. All our results (including those in later sections) have a standard deviation within 0.5%, so we skip the error bars in our figures.

GPT-3 results. The parallelization plan for GPT-3 has been extensively studied [18, 93, 83]. We observe in Fig. 3.7a that this manual plan with the best grid-searched parameters enables Megatron-LM to achieve super-linear weak scaling on GPT-3. Nevertheless, compared to Megatron-LM, Alpa automatically generates execution plans and even achieves slightly

[5]As the models are different for different numbers of GPUs, we cannot measure scaling on the system throughput such as tokens per second or images per second.



(a) GPT (b) MoE (c) Wide-ResNet

Figure 3.8: Intra-operator parallelism ablation study. "×" denotes out-of-memory. Black boxes represent linear scaling.

better scaling on several settings. If compared to methods that only use intra-operator parallelism, our results are consistent with recent studies – "Intra-op only" performs poorly on >16 GPUs because even the best plan has to communicate tensors heavily on cross-node connections, making communication a bottleneck. Surprisingly, "Inter-op only" performs well and maintains linear scaling on up to 64 GPUs.

We investigate the grid-searched parameters of the manual plan on Megatron-LM, and compare it to the plan generated by Alpa. It reveals two major findings. First, in Megatron-LM, the best manual plan has TMP as 1, except in rare settings, such as fitting the 39B model on 64 GPUs, where pipeline parallelism alone is unable to fit the model (stage) in GPU memory; meanwhile, data parallelism is maximized whenever memory allows. In practice, gradient accumulation (GA) is turned on to achieve a desired global batch size (e.g., 1024 in our setting). GA amortizes the communication of data parallelism and reduces the bubbles of pipeline parallelism, but the communication of TMP grows linearly with GA steps, which puts TMP disadvantaged. Second, Alpa-generated plan closely resembles the best-performed ones in Megatron-LM, featuring (1) evenly-sized stages, (2) partitioning along the batch dimension in stages when memory is not stressed, but along non-batch dimensions when memory is stressed. One key difference between our plan and the manual plan is that Alpa also partitions the weight update operations when data parallelism exists, which contributes to the slight performance improvement over Megatron-LM. This attributes to the fact that Alpa, as a generic compiler system, can compose a wide range of parallelism approaches, while Megatron-LM, for now, misses weight update sharding support.

MoE results. DeepSpeed adopts a manual operator parallelism plan for MoE models, developed by GShard [76], called *expert parallelism*, which uses a simple rule: it partitions the expert axis for the operators in MoE layers, but switches back to data parallelism for non-expert layers. This expert parallelism is then combined with ZeRO data parallelism and TMP. All of these techniques belong to intra-operator parallelism. Unfortunately, DeepSpeed's specialized implementation does not include any inter-operator parallelism approach, which is required for scaling across multiple nodes with low inter-node bandwidth. Therefore, Deepspeed only maintains a good performance within a node ($\leq$ 8 GPUs) on this cluster. “Intra-op only” fails to scale across multiple nodes due to the same reason. “Inter-op only” runs out of memory on 32 GPUs and 64 GPUs because it is not easy to equally slice the model when the number of GPUs is larger than the number of layers of the model. The imbalanced slicing makes some memory-intensive stages run out of memory.

By contrast, Alpa automatically discovers the best execution plans that combine intra- and inter-operator parallelism. For intra-operator parallelism, Alpa finds a strategy similar to expert parallelism and combines it with ZeRO data parallelism, thanks to its ILP-based intra-op pass. Alpa then constructs stages and uses inter-operator parallelism to favor small communication volume on slow connections. Alpa maintains linear scaling on 16 GPUs and scales well to 64 GPUs. Compared to DeepSpeed, Alpa achieves 3.5× speedup on 2 nodes and a 9.7× speedup on 4 nodes.

Wide-ResNet results. Unlike the previous two models that stack the same layer, Wide-ResNet has a more heterogeneous architecture. As the data batch is forwarded through layers, the size of the activation tensor shrinks while the size of the weight tensor inflates. This leads to an imbalanced distribution of memory usage and compute intensity across layers. For this kind of model, it is difficult, if not impossible, to manually design a plan. However, Alpa still achieves a scalable performance on 32 GPUs with 80% scaling. The baselines “PP-DP” and “Inter-op only” run out of memory when training large models, because they cannot partition weights to reduce the memory usage, and it is difficult to construct memory-balanced stages for them. “Intra-only” requires a lot of communication on slow connections, so it cannot scale across multiple nodes. A case study on the generated plan for Wide-ResNet is in §3.8.6.

3.8.2 Intra-Op Parallelism Ablation Study

We study the effectiveness of our intra-operator parallelism optimization algorithm. We compare our ILP-based solution against alternatives such as ZeRO optimizer and rule-based partitioning strategies.

Experimental setup. We run a weak scaling benchmark in terms of model size similar to §3.8.1, but disable pipeline parallelism and gradient accumulation to control variables. The benchmark is done on one AWS p3.16xlarge instance with 8 GPUs. In order to simulate an execution environment of large-scale training in one node, we use larger hidden sizes, smaller batch sizes, and smaller numbers of layers, compared to the model configurations in §3.8.1.

Baselines. We compare automatic solutions for intra-operator parallelism. “Data” is vanilla data parallelism. “ZeRO-2” [111] is a memory-efficient version of data parallelism which partitions gradients and optimizer states. “ZeRO-3” [111] additionally partitions parameters on top of “ZeRO-2”. “Heuristic” uses a rule combined with the sharding propagation

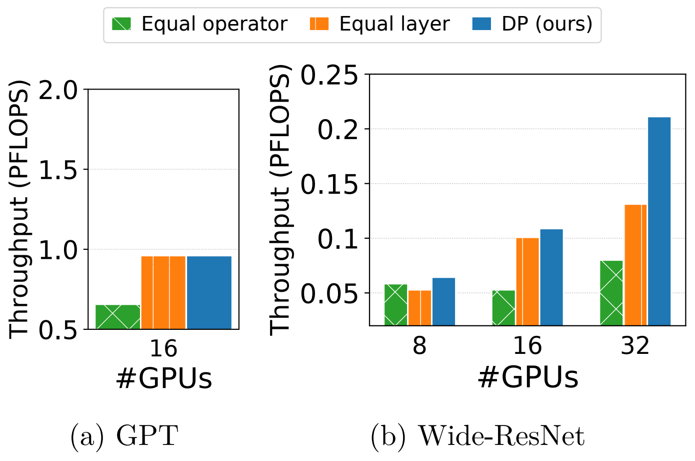


Figure 3.9: Inter-operator parallelism ablation study.

in GSPMD. It marks the largest dimension of every input tensor as partitioned and runs sharding propagation to get the sharding specs for all nodes in the graph. “ILP” is our solution based on the ILP solver.

Results. As shown in Fig. 3.8, “Data” runs out of memory quickly and cannot train large models. “ZeRO-2” and “ZeRO-3” resolve the memory problem of data parallelism, but they do not optimize for communication as they always communicate the gradients. When the gradients are much larger than activations, their performance degenerates. “Heuristic” solves the memory issue by partitioning all tensors, but can be slowed down by larger communication. “Auto-sharding” performs best in all cases and maintains a near-linear scaling, because it figures out the correct partition plan that always minimizes the communication overhead.

3.8.3 Inter-Op Parallelism Ablation Study

We study the effectiveness of our inter-operator parallelism optimization algorithm. We use “DP” to denote our algorithm.

Experimental setup. We report the performance of three variants of our DP algorithm on GPT and Wide-ResNet. The benchmark settings are the same as the settings in §3.8.1.

Baselines. We compare our DP algorithm with two rule-based algorithms. “Equal operator” disables our DP-based operator clustering but assigns the same number of operators to each cluster. “Equal layer” restricts our DP algorithm to use the same number of layers for all stages.

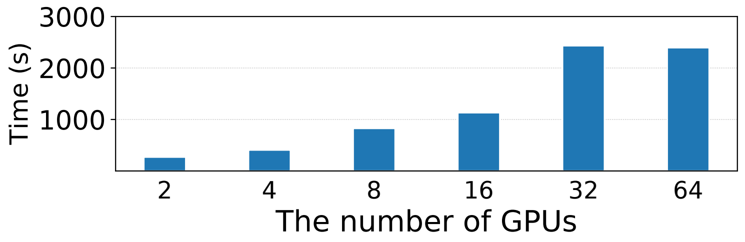


Figure 3.10: Alpa's compilation time on all GPT models. The model size and #GPUs are simultaneously scaled.

Results. Fig. 3.9 shows the result. "DP" always outperforms "Equal operator". This is because "Equal operator" merges operator that should be put onto different device meshes. Alpa's algorithm can cluster operators based on the communication cost and computation balance. Whether "DP" can outperform "Equal layer" depends on the model architecture. On homogeneous models like GPT, the solution of our DP algorithm uses the same number of layers for all stages, so "Equal layer" performs the same as "DP". On Wide-ResNet, the optimal solution can assign different layers to different stages, so "Equal layer" is worse than the full flexible DP algorithm. For Wide-ResNet on 32 GPUs, our algorithm outperforms "Equal operator" and "Equal layer" by 2.6× and 1.6×, respectively.

3.8.4 Compilation Time

Fig. 3.10 shows Alpa's compilation time for all the GPT settings in §3.8.1. The compilation time is a single run of the full Alg. 2 with a provided number of microbatches B. According to the result, Alpa scales to large models or large clusters well, because compilation time grows linearly with the size of the model and the number of GPUs in the cluster. Table 3.5 reports the compilation time breakdown for the largest GPT model in our evaluation (39B, 64 GPUs). Most of the time is spent on enumerating stage-mesh pairs and profiling them. For the compilation part, we accelerate it by compiling different stages in parallel with distributed workers. For profiling, we accelerate it using a simple cost model built at the XLA instruction level, which estimates the cost of matrix multiplication and communication primitives with a piece-wise linear model. With these optimizations, the compilation and search for a model take at most several hours, which is acceptable as it is much shorter than the actual training time, which can take several weeks.

3.8.5 Cross-Mesh Resharding

We evaluate our generalized local all-gather optimization for cross-mesh resharding between meshes with different shapes on Wide-ResNet, as shown in Fig. 3.11. "signal send/recv" is a synthetic case where we only send 1 signal byte between stages, which can be seen as

Table 3.5: Compilation time breakdown of GPT-39B.

Steps	Ours	w/o optimization
Compilation	1582.66 s	> 16hr
Profiling	804.48 s	> 24hr
Stage Construction DP	1.65 s	N/A
Other	4.47 s	N/A
Total	2393.26 s	> 40hr

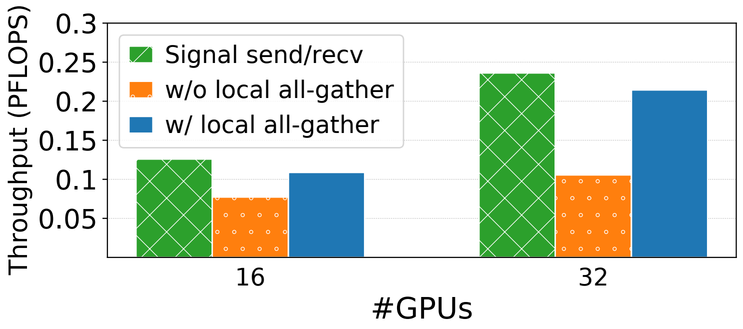


Figure 3.11: Cross-mesh resharding on Wide-ResNet.

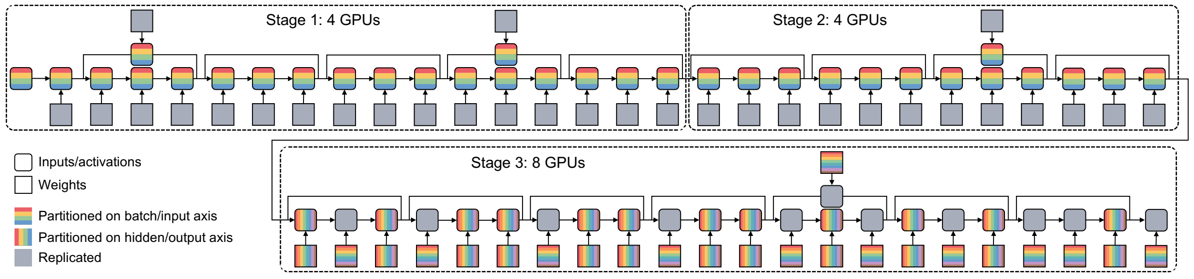


Figure 3.12: Visualization of the parallel strategy of Wide-ResNet on 16 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across the devices. The input data and resulting activation of each convolution and dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

the upper bound of the performance. "w/o local all-gather" disables our local all-gather optimization and uses only send/recv. "w/ local all-gather" enables our local all-gather optimization to move more communication from slow connections to fast local connections, which brings 2.0× speedup on 32 GPUs.

3.8.6 Case Study: Wide-ResNet

We visualize the parallelization strategies Alpa finds for Wide-ResNet on 16 GPUs in Fig. 3.12. We also include the visualization of results on 4 and 8 GPUs in Appendix 3.13. On 4 GPUs, Alpa uses only intra-operator parallelism. The intra-operator solution partitions along the batch axis for the first dozens of layers and then switches to partitioning the channel axis for the last few layers. On 16 GPUs, Alpa slices the model into 3 stages and assigns 4, 4, 8 GPUs to stage 1, 2, 3, respectively. Data parallelism is preferred in the first two stages because the activation tensors are larger than weight tensors. In the third stage, the ILP solver finds a non-trivial way of partitioning the convolution operators. The result shows that it can be opaque to manually create such a strategy for a heterogeneous model like Wide-ResNet, even for domain experts.

3.9 Related Work

Systems for data-parallel training. Horovod [119] and PyTorchDDP [79] are two commonly adopted data-parallel training systems that synchronize gradients using all-reduce. BytePS [106, 67] unifies all-reduce and parameter servers and utilizes heterogeneous resources in data center clusters. AutoDist [157] uses learning-based approaches to compose a data-parallel training strategy. ZeRO [111, 150] improves the memory usage of data parallelism by reducing replicated tensors. MiCS [161] minimizes the communication scale on top of ZeRO for better scalability on the public cloud. In Alpa, data parallelism [69] reduces to a special case of intra-operator parallelism – partitioned along the batch axis.

Systems for model-parallel training. The two major classes of model parallelisms have been discussed in §3.2. Mesh-TensorFlow [122], GSPMD [151, 76] and OneFlow [154] provide annotation APIs for users to manually specify the intra-op parallel plan. ColocRL [89] puts disjoint model partitions on different devices *without pipelining*, thereby the concurrency happens only when there exist parallel branches in the model. In contrast, Gpipe [57] splits the input data into micro-batches and forms pipeline parallelisms. PipeDream [91, 92] improves GPipe by using asynchronous training algorithms, reducing memory usage, and integrating it with data parallelism. However, PipeDream is asynchronous while Alpa is a synchronous training system. TeraPipe [83] discovers a new pipeline parallelism dimension for transformer-based LMs. Google's Pathway system [10] is a concurrent work of Alpa. Pathway advocates a single controller runtime architecture combining "single program multiple data" (SPMD) and "multiple program multiple data" (MPMD) model. This is similar to Alpa's runtime part, where SPMD is used for intra-op parallelisms and MPMD is used for inter-op parallelism.

Automatic search for model-parallel plans. Another line of work focuses on the automatic discovery of model-parallel training plans. Tofu [143] develops a dynamic programming algorithm to generate the optimal intra-op strategy for *linear* graphs on *a single node*. FlexFlow [65] proposes a “SOAP” formulation and develops an MCMC-based randomized search algorithm. However, it only supports device placement without pipeline parallelism. Its search algorithm cannot scale to large graphs or clusters and does not have optimality guarantees. TensorOpt [19] develops a dynamic programming algorithm to automatically search for intra-op strategies that consider both memory and computation cost. Varuna [4] targets low-bandwidth clusters and focuses on automating pipeline and data parallelism. Piper [132] also finds a parallel strategy with both inter- and intra-op parallelism, but it relies on manually designed intra-op parallelism strategies and analyzes on a uniform network topology and asynchronous pipeline parallel schedules.

Techniques for training large-scale models. In addition to parallelization, there are other complementary techniques for training large-scale models, such as memory optimization [23, 62, 20, 56, 70, 114], communication compression [8, 139], and low-precision training[88]. Alpa can incorporate many of these techniques. For example, Alpa uses rematerialization to reduce memory usage and uses mixed-precision training to accelerate computation.

Compilers for deep learning. Compiler techniques have been introduced to optimize the execution of DL models [136, 134, 22, 164, 64, 85, 140]. Most of them focus on optimizing the computation for a single device. In contrast, Alpa is a compiler that supports a comprehensive space of execution plans for distributed training.

Distributed tensor computation in other domains. Besides deep learning, libraries and compilers for distributed tensor computation have been developed for linear algebra [15] and stencil computations [32]. Unlike Alpa, they do not consider necessary parallelization techniques for DL.

3.10 Conclusion

We present Alpa, a new architecture for automated model-parallel distributed training, built on top of a new view of machine learning parallelization approaches: intra- and inter-operator parallelisms. Alpa constructs a hierarchical space and uses a set of compilation passes to derive efficient parallel execution plans at each parallelism level. Alpa orchestrates the parallel execution on distributed compute devices on two different granularities. Coming up with an efficient parallelization plan for distributed model-parallel deep learning is historically a labor-intensive task, and we believe Alpa will democratize distributed model-parallel learning and accelerate the adoption of emerging large deep learning models.

3.11 Appendix: Proof of Submesh Shape Covering

We prove the following theorem which shows we can always find a solution that fully covers the cluster mesh (N, M) with our selected submesh shapes in §5.2: (1) one-dimensional submeshes of shape $(1,1),(1,2),(1,4)\ldots(1,2^m)$ where $2^m = M$ and (2) two-dimensional submeshes of shape $(2, M),(3, M),\ldots,(N, M)$.

Jibberish 1. *For a list of submesh shapes $(n_1, m_1),\ldots(n_S, m_S)$, if $\sum_i n_i \cdot m_i = N \cdot M$ and each (n_i, m_i) satisfies either (1) $n_i = 1$ and $m_i = 2^{p_i}$ is a power of 2 or (2) $m_i = M$, then we can always cover the full (N, M) mesh where $M = 2^m$ with these submesh shapes.*

Proof. We start with putting the second type submesh into the full mesh. In this case, because $m_i = M$, these submeshes can cover the full second dimension of the full mesh. After putting all the second kind of submeshes into the mesh, we reduce the problem to fit a cluster mesh of shape (N, M) with submeshes with shape $(1, 2^{p_1}),\ldots,(1, 2^{p_S})$ where all $p_i \in \{0, 1, \ldots, m-1\}$. Note that now we have

$$2^{p_1} + \cdots + 2^{p_S} = N \cdot 2^m. \tag{3.7}$$

We start an induction on m. When $m = 1$, we have all $p_i = 0$ and thus all the submeshes are of shape $(1, 1)$, which means that all the submeshes can definitely cover the full mesh. Assume the above hold for all $m = 1, 2, \ldots, k-1$. When $m = k$, note that in this case the number of submeshes with $p_i = 0$ should be an even number, because otherwise the left hand side of Eq. 3.7 will be an odd number while the right hand side is always an even number. Then we can split all submeshes with shape $p_i = 0$ into pairs, and we co-locate each pair to form a $(1, 2)$ mesh. After this transformation, we have all $p_i > 0$, so we can subtract all p_i and m by 1 and reduce to $m = k - 1$ case. Therefore, the theorem holds by induction. □

3.12 Appendix: Model Specifications

For GPT-3 models, we use sequence length = 1024 and vocabulary size = 51200 for all models. Other parameters of the models are listed in Table. 3.6. The last column is the number of GPUs used to train the corresponding model.

For GShard MoE models, we use sequence length = 1024 and vocabulary size = 32000 for all models. Other parameters of the models are listed in Table. 3.7. The last column is the number of GPUs used to train the corresponding model.

For Wide-ResNet models, we use input image size = (224, 224, 3) and #class = 1024 for all models. Other parameters of the models are listed in Table. 3.8. The last column is the number of GPUs used to train the corresponding model.

Table 3.6: GPT-3 Model Specification

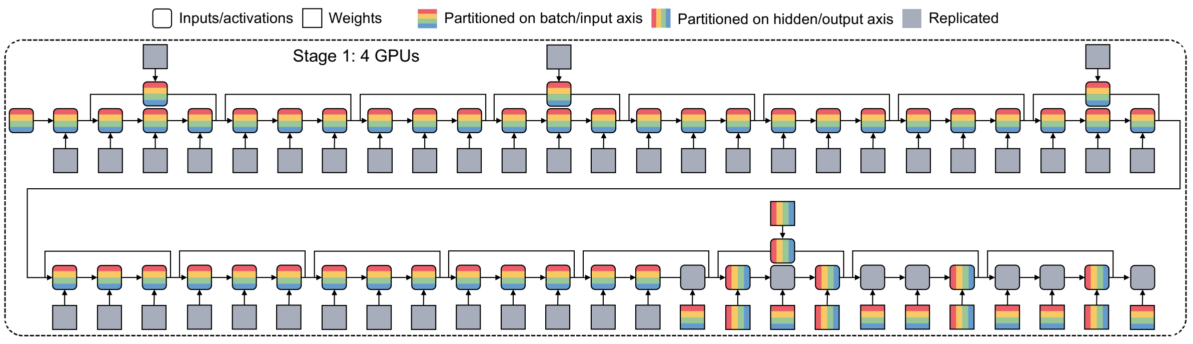
#params	Hidden size	#layers	#heads	#gpus
350M	1024	24	16	1
1.3B	2048	24	32	4
2.6B	2560	32	32	8
6.7B	4096	32	32	16
15B	5120	48	32	32
39B	8192	48	64	64

Table 3.7: GShard MoE Model Specification

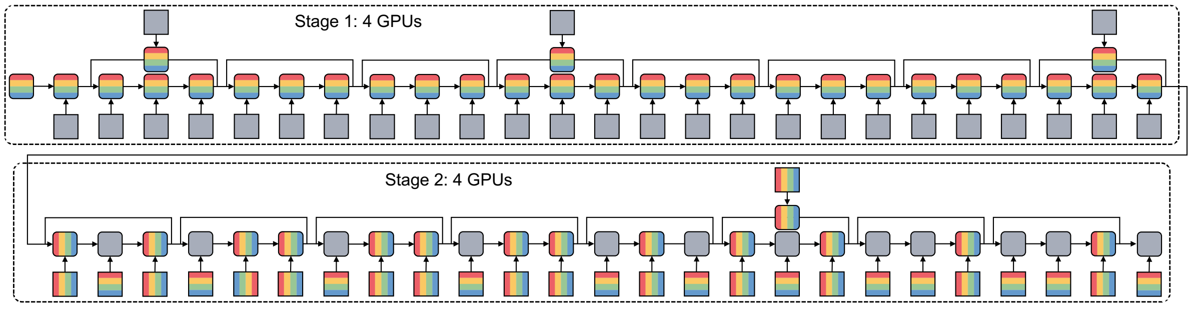
#params	Hidden size	#layers	#heads	#experts	#gpus
380M	768	8	16	8	1
1.3B	768	16	16	16	4
2.4B	1024	16	16	16	8
10B	1536	16	16	32	16
27B	2048	16	32	48	32
70B	2048	32	32	64	64

Table 3.8: Wide-ResNet Model Specification

#params	#layers	Base channel	Width factor	#gpus
250M	50	160	2	1
1B	50	320	2	4
2B	50	448	2	8
4B	50	640	2	16
6.8B	50	320	16	32
13B	101	320	16	64


Inputs/activations
Weights
Partitioned on batch/input axis
Partitioned on hidden/output axis
Replicated
Stage 1: 4 GPUs

(a) Parallel strategy of Wide-ResNet on 4 GPUs.



(b) Parallel strategy of Wide-ResNet on 8 GPUs.

Figure 3.13: Visualization of the parallel strategy of Wide-ResNet on 4 and 8 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across all devices. The input data and resulting activation of each convolution or dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

3.13 Appendix: Extra Case Study

We visualize the parallelization strategies Alpa finds for Wide-ResNet on 4 and 8 GPUs in Fig. 3.13.

Chapter 4

AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving

Model parallelism is conventionally viewed as a method to scale a single large deep learning training model beyond the memory limits of a single device, as illustrated in chapters 2 and 3. Additionally, while training has been a major focus in deep learning, a model actually spends most of its time on inference instead of training: A model is typically trained once but can be used for inference over a long period to serve numerous requests.

In this chapter, we demonstrate that model parallelism can be additionally used for the statistical multiplexing of multiple devices when serving multiple models, even when a single model can fit into a single device. Our work reveals a fundamental trade-off between the overhead introduced by model parallelism and the opportunity to exploit statistical multiplexing to reduce serving latency in the presence of bursty workloads. We explore the new trade-off space and present a novel serving system, AlpaServe, that determines an efficient strategy for placing and parallelizing collections of large deep learning models across a distributed cluster. Evaluation results on production workloads show that AlpaServe can process requests at up to 10× higher rates or 6× more burstiness while staying within latency constraints for more than 99% of requests.

4.1 Introduction

Advances in self-supervised learning have enabled exponential scaling in model sizes. For example, large pretrained models like BERT [34] and GPT-3 [18] have unlocked a plethora of new machine learning (ML) applications from Copilot [46] to copy.ai [26] and ChatGPT [101].

Serving these very large models is challenging because of their high computational and mem-

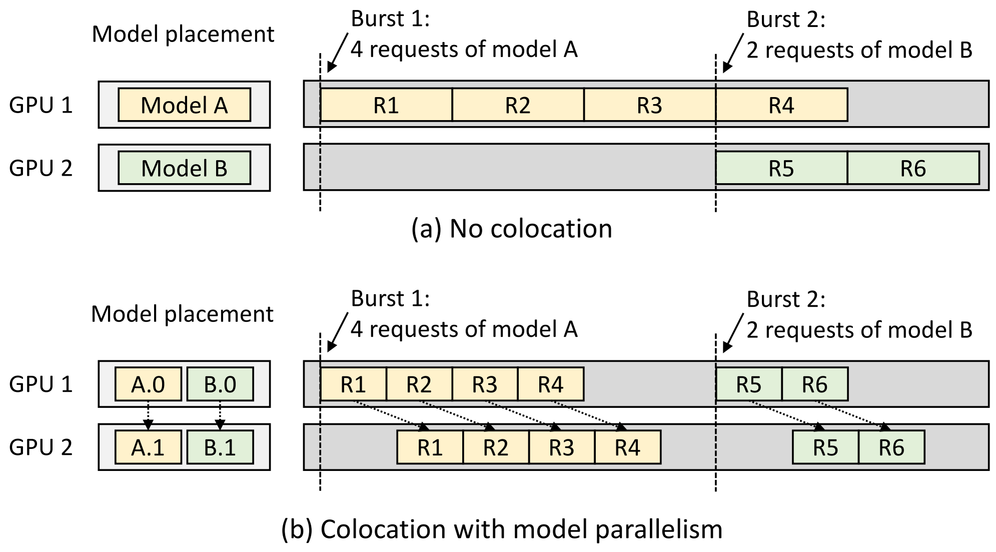


Figure 4.1: Two placement strategies for serving two models on two GPUs. In each subfigure, the left part shows the model placements and the right part shows the timeline for handling bursty requests. At the time of "Burst 1", 4 requests of model A come at the same time. Colocation with model parallelism can reduce the average completion time of bursty requests.

ory requirements. For example, GPT-3 requires 325 GB of memory to store its parameters as well as a requisite amount of computation to run inference. To serve this model, one would need at least 5 of Nvidia's newest Hopper 80 GB GPUs just to hold the weights and potentially many more to run in real-time. Worse yet, the explosive growth of model sizes continues unabated [25, 41]. Techniques like model compression and pruning are not sufficient in face of the exponential growth in model sizes and often come at the expense of reduced model quality [35].

Provisioning sufficient resources to serve these models can be arduous as request rates are bursty. For example, using common workload traces, we observe frequent spikes in demand of up to 50× the average [160]. Meeting the service level objective (SLO) of latency usually means provisioning for these peak loads, which can be very expensive; additional devices allocated for this purpose would remain underutilized most of the time. Making matters worse, it is increasingly common to serve multiple models and multiple variations of the same large model in situations like A/B testing or serving fine-tuned models for specific domains (§4.2).

This chapter studies how to efficiently serve multiple large models concurrently. Specifically, we explore the underappreciated benefits of model parallelism in online model serving, even for smaller models that can fit on a single device. Model parallelism refers to partitioning and executing a model on distributed devices (§4.2.1). The benefits of model parallelism have been well studied [165, 57, 76, 94] in the *throughput-oriented* training setting. However, its effects for model serving under *latency-sensitive* settings remains largely untapped.

We observe that there are fundamental transition points in the model serving design space that challenge prior assumptions about serving, even for models that fit on a single device. For example, consider the scenario with two models and two GPUs, each of which has sufficient memory to hold one complete model. As shown in Fig. 4.1(a), the natural approach assumed by almost all existing serving systems [97, 28, 98] is to allocate one dedicated GPU for one model. This approach appears rational because partitioning the model across GPUs would incur communication overheads that would likely increase the prediction latency. However, we find that inducing additional model parallelism (to the point where per-example execution time actually *increases*) enables a wider range of placement strategies, e.g., model co-location, which can improve the statistical multiplexing of the system under bursty workloads. In Fig. 4.1(a), assuming the execution time of a model is y, the average end-to-end latency of request 1 through 4 is $(1y + 2y + 3y + 4y)/4 = 2.5y$. In Fig. 4.1(b), assuming a 10% model-parallel overhead, the average latency of request 1 through 4 is reduced to $(1.1y + 1.6y + 2.1y + 2.6y)/4 = 1.85y$. Co-location with model parallelism can utilize more devices to handle bursty requests and reduces the average completion time, despite its overheads (§4.3.1). Even if we batch the requests, the case still holds (§4.6.5).

Unfortunately, the decision of how to optimally split and place a collection of models is complex. Although leveraging model parallelism as above has its benefits, it still adds overheads that may negate those benefits for less bursty workloads. For example, we find that a particularly influential axis on the efficacy of model parallelism is per-GPU memory capacity (§4.3.2), although other factors (e.g., the arrival pattern, SLO) can also have a significant effect. Further, besides the inter-op model parallelism presented in fig. 4.1, another kind of model parallelism, intra-op parallelism, presents its own distinct tradeoffs (§4.3.3). Ultimately, different styles of parallelism and their tradeoffs create a complex, multi-dimensional, and multi-objective design space that existing systems largely ignore and/or fail to navigate. However, not leveraging model parallelism in the serving setting is typically not an option for large models, and not addressing this trade-off space directly results in significant increases in cost and serving latency.

To that end, we present AlpaServe, a system that automatically and efficiently explores the tradeoffs among different parallelization and placement strategies for model serving. AlpaServe takes a cluster resource specification, a set of models, and a periodic workload profile; it then partitions and places the models and schedules the requests to optimize SLO attainment (i.e., the percentage of requests served within SLO). To assist the design of AlpaServe, we first introduce a taxonomy and quantify the tradeoffs between different parallelization strategies in model serving (§4.3). We then present key algorithms to navigate the tradeoff space (§4.4). We design an iterative simulator-guided model placement algorithm to optimize the colocation of models and a group partition algorithm to search for the best way to partition the cluster into disjoint model-parallel groups. In addition, we extend the existing auto-parallelization algorithms for training to make them more suitable for inference.

We evaluate AlpaServe with production workloads on a 64-GPU cluster (§4.6). Evaluation results show that, when optimizing one metric at a time, AlpaServe can choose to increase the request processing rate by 10×, achieve 2.5× lower latency deadlines, or tolerate 6× burstier traffic compared to previous state-of-the-art serving systems.

In summary, we make the following contributions:

- A detailed analysis of the tradeoff space of different model parallel strategies for efficient model serving.
- Novel model placement algorithms to incorporate model parallelism in a serving system.
- A comprehensive evaluation of AlpaServe with both synthetic and production workloads.

4.2 Background

Over the past few years, increasingly capable models have been developed for everything from recommendations to text generation. As a result, serving predictions from these models has become an essential workload in modern cloud systems. The structure of these workloads often follows a simple request-response paradigm. Developers upload a pre-trained model and its weights ahead of time; at runtime, clients (either users or other applications) submit requests for that model to a serving system, which will queue the requests, dispatch them to available GPUs/TPUs, and return the results.

The requirements of these model-serving systems can be stringent. To satisfy user demand, systems often must adhere to aggressive SLO on latency. At the same time, serving systems that must run continuously need to minimize their operational costs associated with expensive accelerators. Minimizing serving costs can be challenging because dynamically scaling compute resources would be too slow on the critical path of each prediction request: it can take multiple seconds just to swap a large model into accelerator memory [111]. Furthermore, there is significant and unpredictable burstiness in the arrival process of user requests. To meet tight SLO, contemporary serving systems are forced to over-provision compute resources, resulting in low cluster utilization [146].

Another pattern that emerges in serving large models is the use of multiple instances of the same or similar model architectures. This is commonly seen in the practice of pretraining on large unlabeled data and fine-tuning for various downstream tasks [34], which can significantly boost accuracy but results in multiple instances of the same model architecture. For example, Hugging Face serves more than 9,000 versions of fine-tuned BERT [58]. They either share a portion of the parameters or do not share any parameters at all for better

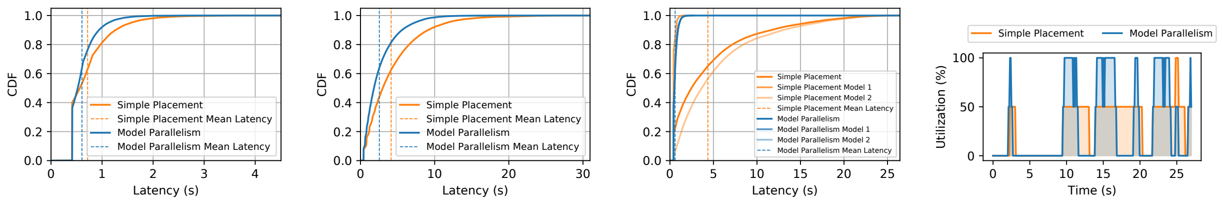


Figure 4.2: Latency CDF and cluster utilization in the 2-model example.

accuracy. Prior works have [166, 123] exploited the property of shared parameters, but we do not consider the shared parameters in this chapter because AlpaServe targets general settings and full-weight tuning is still a major use case.

4.2.1 Model Parallelism in Model Serving

Distributed parallel model execution is necessary when attempting to satisfy the serving performance requirements or support large models that do not fit in the memory of a single device. At a high level, distributed execution of deep learning models can be classified into two categories: intra-operator parallelism and inter-operator parallelism [165].

Intra-operator parallelism. DL models are composed of a series of operators over multi-dimensional tensors, e.g., matrix multiplication over input and weight tensors. Intra-operator parallelism is when a single operator is partitioned across multiple devices, with each device executing a portion of the computation in parallel [151, 122, 126]. Depending on the specific partitioning strategy and its relationship to prior and subsequent operators in the model, partitioning can require communication among participating GPUs to split the input and then merge the output.

The benefit of intra-operator parallelism for single-request execution is twofold. First, it can expand the total amount of computation available to the target model, reducing its end-to-end latency. In a similar fashion, it can expand the total memory available to the model for storing its inputs, weights, and intermediate values. The cost is the aforementioned communication overhead.

Inter-operator parallelism. The other type of parallelism available to DL models is inter-operator parallelism, which assigns different operators of the model's execution graph to execute on distributed devices in a pipeline fashion (a.k.a. pipeline parallelism) [57, 91, 83]. Here, devices communicate only between pipeline stages, typically using point-to-point communication between device pairs.

Unlike intra-operator parallelism, pipeline parallelism does not reduce the execution time of a single request. In fact, it typically increases the execution time due to modest amounts of communication latency between pipeline stages, although the total amount of transferred data is often lower than it is in intra-operator parallelism. Instead, the primary use of inter-operator parallelism in traditional serving systems is to allow the model to exceed the memory limitation of a single GPU.

4.3 Motivation and Tradeoff Analysis

As mentioned, both types of model parallelism reduce per-device memory usage by partitioning a model on multiple devices. A key motivation for this work is that we can use this property to fit more models on one device, enabling better statistical multiplexing of the devices when handling bursty requests. We explore this idea through a series of empirical examinations and theoretical analysis, starting with an illustrative example (§4.3.1), followed by an empirical analysis of when model parallelism is beneficial (§4.3.2), the overhead of model parallelism (§4.3.3), and a queueing theory-based analysis (§4.3.4). All the experiments in this section are performed on an AWS EC2 p3.16xlarge instance with 8 NVIDIA 16GB V100 GPUs.

4.3.1 Case Study: A Two-model Example

We start with an illustrative experiment to show how model parallelism can benefit the serving of multiple models. We use two GPUs to serve two Transformer models with 6.7 billion parameters each (13.4 GB to store its FP16 weights). Because each GPU has 16 GB of memory, it can fit one and only one model. A single request takes around 0.4 s to process on one GPU.

We compare the following model placements, corresponding to the strategies in fig. 4.1. The first is *simple placement*, where we place one model on each GPU due to the memory constraint. The second is *model-parallel placement*, where we use inter-op parallelism to partition each model to a 2-stage pipeline and let each GPU execute half of each model.

We evaluate the two placements when the requests to each model follow an independent Poisson process with an arrival rate of 1.5 request/s. section 4.2 shows the cumulative distribution function (CDF) and average of request latency (which includes the GPU execution time and queuing delay). Model-parallel placement reduces the average latency of the simple placement from 0.70s to 0.55s, a 1.3× speedup. The speedup comes from the better burst tolerance: when a burst arrives that exceeds the capability of a single GPU, simple placement must begin queuing requests. However, as long as the other model does not receive many requests, the model parallel placement can use both GPUs to serve the requests for the popular model via statistical multiplexing of the GPUs.

This effect becomes more pronounced with higher burstiness, which we can demonstrate using a Gamma request arrival process with the same average request rate as above but a

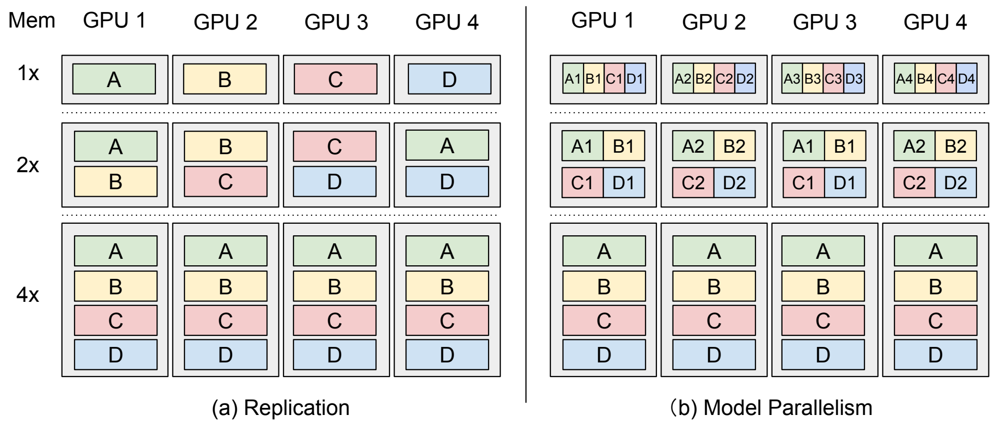


Figure 4.3: Replication and model parallel placement illustration with different memory budgets, where the memory budgets are set to be multiples of a single model's size.

higher coefficient of variance (CV) of 3. As shown in section 4.2, the speedup on mean latency is now increased to 1.9×. section 4.2 shows a representative trace of the corresponding total cluster utilization over time. Note that for each request burst, model-parallel placement can use the whole cluster and only take half of the time to process, while simple placement can only use half of the cluster.

In addition, we also evaluate the case where one model receives more requests than another. In section 4.2, we use Poisson arrival but let 20% of the requests ask for model 1 and 80% ask for model 2. Although replication performs slightly better for model 1 requests, it is drastically worse on model 2 requests compared to the model-parallel placement. For model-parallel placement, because both GPUs are shared across two models, the requests to both models follow the same latency distribution. Overall, model-parallel placement reduces the mean latency by 6.6×.

4.3.2 When is Model Parallelism Beneficial

To further explore the nuances of model parallelism in serving, we increase the size of the deployment to 8 GPUs and 8 Transformer models with 2.6B parameters each. As a base setting, we set the requests to each model as a Gamma process with an average rate of 20 request/s and CV of 3; we then vary a range of factors to see their effects. Note that some of the settings we evaluate are impossible on real hardware (e.g., exceeding the memory capacity of a single device) so we leverage the simulator introduced in §4.5. The fidelity of the simulator is very high as verified in §4.6.1.

The model in this case is smaller (5.2GB), so one GPU can also store multiple models without model parallelism. We compare two placement methods: (1) *Replication.* In this setting,

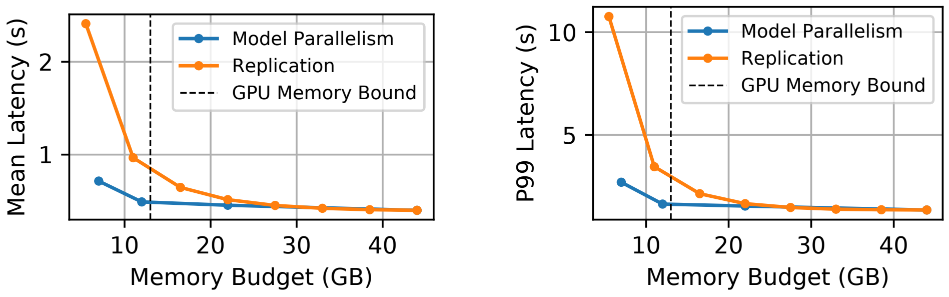


Figure 4.4: Serving performance with changing per-GPU memory budgets. Model parallelism is beneficial for limited memory budget. The dashed vertical line is the real per-GPU memory bound of a 16GB V100. The value is around 13GB due to the need to store activations and other runtime context.

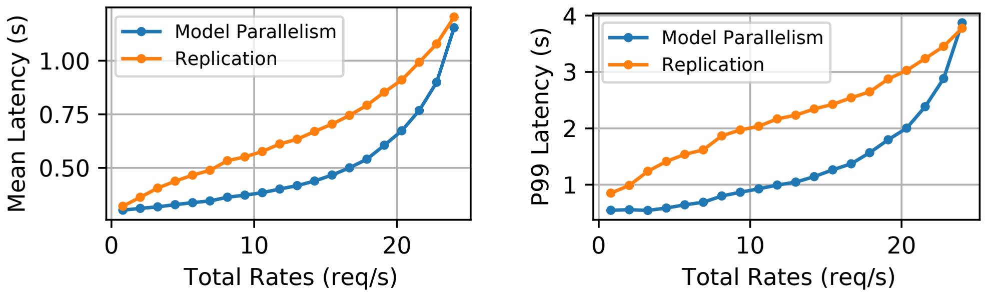


Figure 4.5: Serving performance with changing arrival rates. Model parallelism is beneficial for smaller rates.

we replicate the models to different devices until each device cannot hold any extra models. Because all the models receive equal amounts of loads on average, we replicate each model the same number of times (fig. 4.3a). (2) *Model Parallelism.* Here we use inter-operator parallelism and uniformly assign the Transformer layers to different GPUs.

Device memory. We evaluate the mean and the tail latency of the two placement methods under different device memory capacities. For replication, more GPU memory can fit more models onto a single GPU. For model parallelism, more GPU memory can also reduce the number of pipeline stages and reduce the overhead as in fig. 4.3b. The resulting mean and P99 latency is shown in fig. 4.4. With more memory, more models can fit into a single GPU, so the benefit of statistical multiplexing diminishes because replication can also effectively use multiple devices to serve the bursty requests to a single model. When the GPU memory capacity is large enough to hold all models, there is no gain from model parallelism.

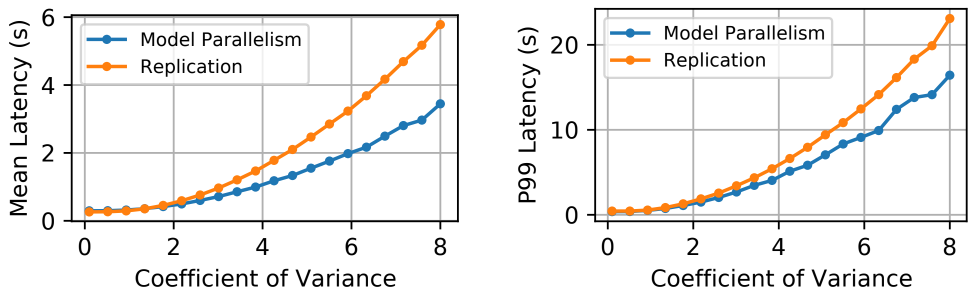


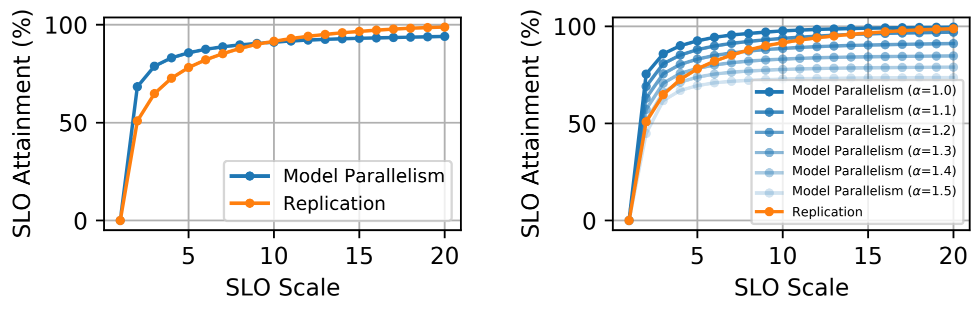
Figure 4.6: Serving performance with changing CVs. Model parallelism is beneficial for larger CVs.

Request arrival. We vary the parameters of the arrival process and compare the replication placement with the model-parallel placement with 8-stage pipeline parallelism. The mean and P99 latency results of changing arrival rate are shown in fig. 4.5. When the arrival rate is low, model parallelism can greatly reduce the serving latency. However, when the arrival rate approaches the peak serving rate of the cluster, the benefit of model-parallel placement starts to diminish. Eventually, it starts to perform worse than replication. This is because when all models are equally saturated, the replication placement is able to achieve efficient cluster utilization and there is no benefit to the statistical multiplexing afforded by model parallelism. Instead, the overhead of model parallelism (§4.3.3) starts to become a significant factor.

The mean and P99 latency results of changing CV are in fig. 4.6. With a higher CV, the requests become more bursty, and the benefit of model parallelism becomes more significant. As shown in the results, with a higher CV, model parallelism can greatly outperform the performance of replication.

Service level objectives. In prediction serving settings, it is common to have tight latency SLO and predictions made after these deadlines are often discarded [50]. For example, advertising systems may choose not to show an ad rather than delay rendering user content. In this case, the goal of the serving system is to optimize the percentage of requests that can be finished within the deadline, i.e., *SLO attainment*.

In this experiment, we measure how SLOs affect the performance of the placement methods. We compare the replication and the model-parallel placement with 8-stage pipeline parallelism. During execution, we drop the requests that will exceed the deadline even if we schedule it immediately. We scale the SLO to different multiplies of the single device execution latency (*SLO Scale* in fig. 4.7a) and compare the SLO attainment of the two methods.



(a) Real model latency.

(b) Changing overhead.

Figure 4.7: SLO attainment with changing SLOs. Model parallelism is beneficial for smaller SLOs.

As in fig. 4.7a, when SLO is tight ($< 10\times$ model latency), model parallelism can greatly improve SLO attainment. However, when the SLO becomes looser, its SLO attainment plateaus but that of the replication placement keeps growing. This result shares the same core logic as previous experiments: When SLO becomes looser, more requests can stay in the waiting queue, and thus the effective burstiness of the requests decreases. When many requests are queued, the system is bounded by its total processing capability, which might be affected by the model parallelism overhead. In the real world, the SLO requirement is often less than $5\times$ of the model execution latency [50], where model parallelism can improve SLO attainment.

Summary: Model parallelism benefits model serving through statistical multiplexing when the device memory is limited, the request rate is low, the request CV is high, or the SLO is tight.

4.3.3 Overhead of Model Parallelism

In this section, we further investigate the overheads of different model parallel strategies and how they affect serving performance. Similar to the setup in fig. 4.7a, we manually modify the overhead of model parallelism. Specifically, let the latency of a single model executing on the GPU be L and the number of pipeline stages be n. We set the total latency of pipeline execution to be αL and the latency of each pipeline stage to be $\alpha L/n$, where α is a parameter that controls the overhead. When $\alpha = 1$, model parallelism does not have any overhead and larger α means higher overhead.

We show the results in fig. 4.7b. If model parallelism does not have any overhead ($\alpha = 1$), it can always outperform replication due to its ability to multiplex the devices. When the

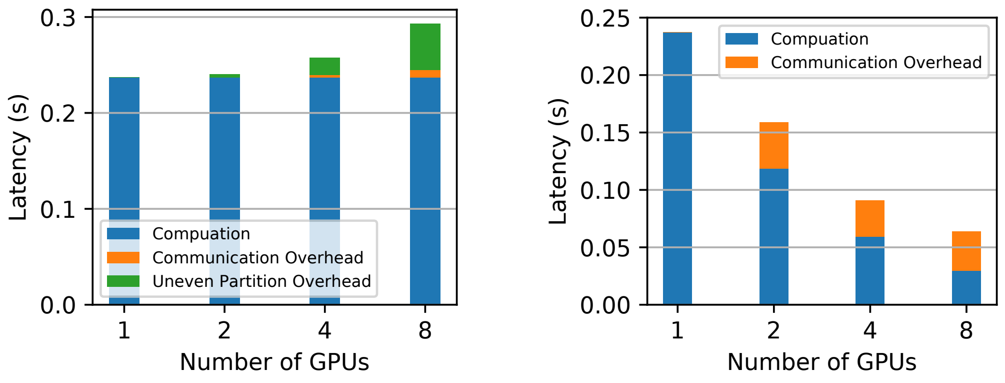


Figure 4.8: The overhead decomposition. The overhead of inter-op parallelism mainly comes from uneven partition while the overhead of intra-op parallelism comes from communication.

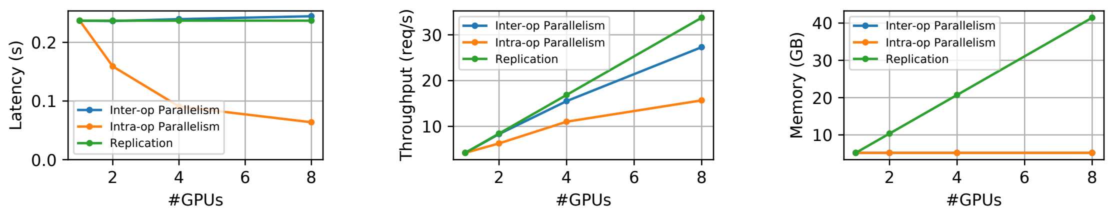


Figure 4.9: The latency, throughput and memory usage vs. #GPUs for inter-op parallelism, intra-op parallelism, and replication. In subfigure (c), the lines for inter-op and intra-op parallelism overlap.

overhead becomes larger and the SLO is low, model parallelism still outperforms replication. However, with a larger SLO, the effective burstiness is reduced and the performance is dominated by the overhead.

Given that the overhead can greatly affect serving performance, we perform a detailed study of the multiple sources of model-parallel overhead in fig. 4.8. For inter-op parallelism, when partitioning a single model into multiple stages, different stages need to communicate the intermediate tensors, and we denote this overhead as the *communication overhead*. In addition, the pipeline execution will be bottlenecked by the execution time of the slowest stage, making the effective latency to be the number of pipeline stages times the latency of the slowest stage [57]. We denote this as the *uneven partition overhead*. As in section 4.3.3, for inter-op parallelism, most overhead comes from the latency imbalance among different pipeline stages, instead of the communication between stages. While our previous discussion mainly focuses on inter-op parallelism, the other type of model parallelism, intra-op parallelism, has very different performance characteristics. Its overhead is merely brought by the collective communication across multiple devices [94], which cannot be overlapped with the neural network computation due to data dependency. From section 4.3.3, we can see that the communication overhead of intra-op parallelism is much higher than inter-op parallelism.

Finally, we compare the latency, throughput, and memory consumption of different model-parallel placements and the replication placement in fig. 4.9. Because of the sequential dependence between the different stages, inter-op parallelism cannot reduce the execution latency of a single input data. Instead, the latency is slightly higher due to the communication between the stages. On the other hand, intra-op parallelism can largely reduce the latency via the parallel execution of different GPUs (section 4.3.3). However, because inter-op parallelism can pipeline the execution of different stages and only communicate a relatively small amount of data, it attains higher throughput compared to intra-op parallelism (section 4.3.3). Because both parallel methods split the model weight tensors across different GPUs, the total memory usage stays constant with increasing numbers of GPUs (section 4.3.3). This makes the statistical multiplexing of different GPUs across multiple models possible.

In the end, the tradeoff between parallelization strategies and their interplay with cluster resources, arrival patterns, and serving objectives forms an intricate design space.

4.3.4 Queueing Theory Analysis

In this section, we use queuing theory to mathematically verify the conclusions in §4.3.2 and §4.3.3. Specifically, we analyze the case where the inputs follow the Poisson arrival process. Since the execution time of a deep learning inference task is highly predictable [50], we assume the request serving time is deterministic. For the single device case, suppose the request rate to a model is λ_0 and the single device latency is D with the utilization $\lambda_0 D < 1$, then the average number of requests L_Q and the average latency W in this M/D/1 queue [127] are:

$$L_Q = \frac{\lambda_0 D}{2(1-\lambda_0 D)}, \quad W = D + L_Q D = D + \frac{\lambda_0 D^2}{2(1-\lambda_0 D)}.$$

Now consider the example in §4.3.1. Let $p\lambda, (1-p)\lambda$ be the average request rates for the two models respectively, where $p \in [0,1]$ controls the percentage of requests for both models. Then for the simple placement, the average latency can be derived as the average latency of two independent queues:

$$W_{simple} = D + \frac{p^2 \lambda D^2}{2(1-p\lambda D)} + \frac{(1-p)^2 \lambda D^2}{2(1-(1-p)\lambda D)}.$$

Note that W_{simple} reaches minimum when $p = 1/2$. Intuitively, when p is not exactly half, one model receives more requests than the other. This larger portion of requests have a longer queueing delay, which leads to the higher overall mean latency.

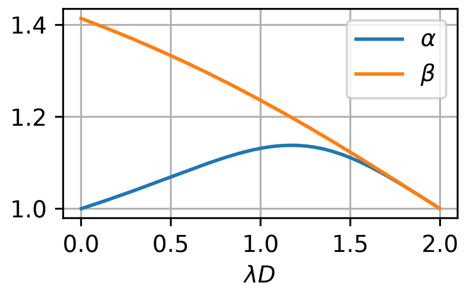


Figure 4.10: Maximal communication overhead α and uneven partition overhead β satisfy $W_{pipeline} \le W_{simple}$ as a function of total utilization λD.

For the model-parallel case, the requests to both models merged to a single Poisson Process with rate λ. For pipeline parallelism, suppose the latency for a single input to be D_s and the maximum stage latency to be D_m, then the average latency would be

$$W_{pipeline} = D_s + \frac{\lambda D_m^2}{2(1-\lambda D_m)}.$$

Suppose there is no model-parallel overhead, then $D_s = 2D_m = D$. Let's first consider the case where $p = 1/2$ (section 4.2). We have

$$W_{simple} = D + \frac{\lambda D^2}{4 - 2\lambda D}, \quad W_{pipeline} = D + \frac{\lambda D^2}{8 - 4\lambda D}.$$

In this case, the waiting time for model-parallel execution is half of the simple placement waiting time, as shown in the vertical lines in section 4.2. When the p is not 1/2, W_{simple} will increase while $W_{pipeline}$ will stay the same, so the gap between W_{simple} and $W_{pipeline}$ will be even larger, as in section 4.2.

Next, we consider the case where model parallelism incurs overhead. We measure the two types of overheads in §4.3.3 separately: With the overhead from communication, $D_s = 2D_m = \alpha D$, where $\alpha \ge 1$ is the overhead factor. With the overhead from uneven stages, we suppose $D_s = D$ still holds, but $D_m = \beta D/2$ where $\beta \ge 1$ is the overhead factor. To keep $W_{pipeline} \le W_{simple}$, we can get the maximal α and β as a function of the total utilization λD separately and we visualize the function in fig. 4.10. When the utilization is high, the benefit of statistical multiplexing diminishes, and thus the overhead needs to be low, as in §4.3.2. On the other hand, when the utilization is very low, most requests will not be queued, and thus the communication overhead α needs to be low to keep the processing latency to be low. Note that the maximal overhead here is based on a uniform Poisson arrival distribution. A more bursty or more non-uniform arrival distribution will make the simple placement performs worse and make the model-parallelism placement outperforms the simple replication placement with even higher overhead.

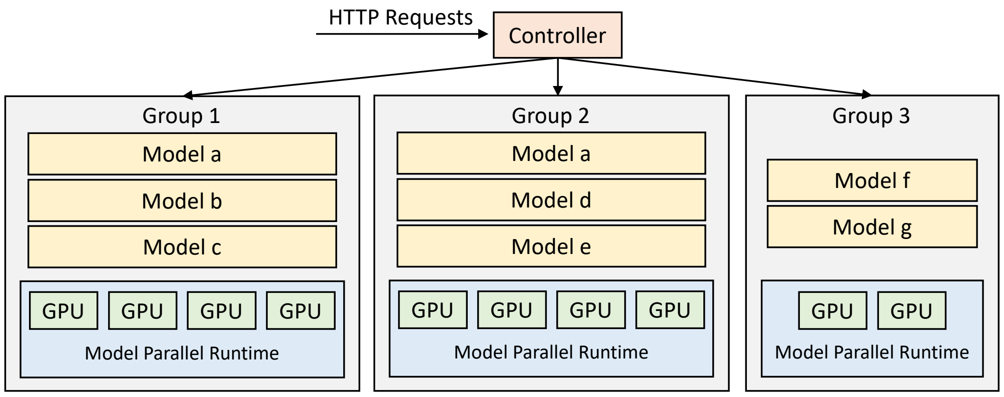


Figure 4.11: AlpaServe Runtime System Architecture

4.4 Method

From §4.3, we can see that there are several key challenges to effectively utilize model parallelism for deep learning serving:

- Derive efficient model parallel strategies for inference to reduce the overhead of model parallelism. Specifically, find a partitioning strategy that minimizes the stage imbalance for inter-operator parallelism.
- Determine model-parallel placements according to the arrival pattern to maximize SLO attainment.

We built *AlpaServe* to specifically tackle these challenges. The runtime architecture of AlpaServe is shown in Fig. 4.11. AlpaServe utilizes a centralized controller to dispatch requests to different groups.[1] Each group hosts several model replicas on a shared model-parallel runtime. This section describes the architecture of AlpaServe and the key algorithms for efficiently leveraging model parallelism in a model serving system.

4.4.1 Automatic Parallelization for Inference

Since different parallelization configurations have different latency and throughput trade-offs, we need to enumerate multiple possible configurations for every single model and let the placement algorithm choose the best combination for all models in the whole cluster. Therefore, given a model, AlpaServe first runs an auto-parallelization compiler with various constraints to generate a list of possible configurations. We build several extensions on

[1]For a larger service, AlpaServe can be extended as a hierarchical deployment with each controller only managing a subset of devices as in [158].

top of an existing auto parallelization training system, Alpa [165], to make it suitable for generating serving parallelization strategies. Alpa includes two passes for generating efficient model parallel partitions: inter-op pass and intra-op pass. The inter-op pass uses a dynamic programming (DP) algorithm to figure out the optimal inter-op parallel plan, and it calls the intra-op pass for each potential pipeline stage, which is formulated as an integer linear programming (ILP) problem, to profile its latency with the optimal intra-op parallel plan. In AlpaServe, we keep the two compilation passes, but extends both passes for serving.

The inter-op pass in Alpa optimizes the overall pipeline execution latency, which includes the time of forward and backward propagation and weight synchronization. However, in serving workloads, only forward propagation is being executed and there is no need for weight synchronization. Therefore, we reformulate the dynamic programming in AlpaServe to merely focus on minimizing the maximal stage latency. Specifically, denote $F(s,k)$ to be the maximum latency when slicing layers 1 to k into s stages. We can derive F as

$$F(s,k) = \min_{1\le i\le k} \left\{\max\{F(s-1,i-1), \mathit{latency}(i,k)\}\right\},$$

where $\mathit{latency}(i,k)$ denotes the latency of a stage composes of layer i to k. In Alpa, the *latency* function of all possible $O(K^2)$ combinations is being profiled by the intra-op pass because of the complicated dependency between forward and backward passes. In AlpaServe, because the pipeline stages only perform forward propagation and only communicate intermediate results once between layer boundaries, we can accelerate the profiling by only profiling K layers and letting $\mathit{latency}(i,k)$ to be the sum of the latencies for layer i to k. This acceleration enables us to efficiently enumerate different inter- and intra-op device partition setups and generate a list of parallel strategies for the placement algorithm in §4.4.2.

For the intra-op pass, we extend the ILP in Alpa to drop all configurations that use data parallelism. For serving workloads, because there is no need for weight synchronization, data parallelism can be achieved by the replication placement. We leave the decision of whether to replicate a model to the placement algorithm in §4.4.2.

4.4.2 Placement Algorithm

Given a set of models and a fixed cluster, AlpaServe partitions the cluster into several groups of devices. Each group of devices selects a subset of models to serve using a shared model-parallel configuration. Different groups can hold the same model as replicas. The requests for a model are dispatched to the groups with the requested model replica. We call a specific cluster group partition, model selection, and parallel configuration as a *placement*. Our goal is to find a placement that maximizes the SLO attainment.

However, finding the optimal placement is a difficult combinatorial optimization problem. The overall placement configuration space grows exponentially with the number of devices and the number of models. To make things worse, the objective "SLO attainment" has no

Algorithm 3 Simulator-Guided Greedy Model Selection.

```
Input: Model list $M$, device group list $G$, group parallel configurations $P$, workload $W$,
  beam size $k$ (default = 1).
Output: The model selection $best\_sel$.
  $best\_sel \leftarrow \emptyset$
  $beam\_sels \leftarrow \{\emptyset\}$
  while true do
      $new\_sels \leftarrow \emptyset$
      for $sel \in beam\_sels$ do
          for $(m, (g, p)) \in M \times (G, P)$ do
              // Parallelize the model as in §4.4.1.
              $m_{parallelized} \leftarrow \text{parallelize}(m, g, p)$
              $sel' \leftarrow sel.\text{add\_model\_to\_group}(m_{parallelized}, g)$
              if $sel'$ is in memory constraint then
                  $sel'.slo\_attainment \leftarrow \text{simulate}(sel', W)$
                  $new\_sels.\text{append}(sel')$
              end if
          end for
      end for
      if $new\_sels = \emptyset$ then
          break
      end if
      $beam\_sels \leftarrow \text{top-}k\text{\_SLO\_attainment}(new\_sels)$
      $sel^* \leftarrow \text{pick\_highest\_SLO\_attainment}(beam\_sels)$
      if $sel^*.slo\_att > best\_sel.slo\_att$ then
          $best\_sel \leftarrow sel^*$
      end if
  end while
  return $best\_sel$
```

simple analytical formula for an arbitrary arrival distribution. Existing tools and approximations from queueing theory can only analyze simple cases in §4.3.4 and cannot model more complex situations [127]. Therefore, we resort to a simulator-guided greedy algorithm that calls a simulator to compute SLO attainment.

To compute the SLO attainment with a given set of requests and placement, in AlpaServe, we assume we know the arrival process in advance. Although short-term burstiness is impossible to predict, the arrival pattern over longer timescales (e.g., hours or days) is often predictable [146]. Given this predictability, AlpaServe either directly uses the history request traces or fits a distribution from the trace and resamples new traces from the distribution as the input workload to the simulator to compute the SLO attainment.

We design a two-level placement algorithm: Given a cluster group partition and a shared model-parallel configuration for each group, algorithm 3 uses a simulator-guided greedy algorithm to decide which models to select for each group. Then, algorithm 4 enumerates various potential cluster partitions and parallel configurations and compares the SLO attainment from algorithm 3 to determine the optimal placement.

Given a cluster group partition with a fixed model-parallel configuration for each group, algorithm 3 selects model replicas iteratively as a beam search algorithm: At each iteration, it enumerates all possible (model, group) pairs, parallelizes the model on the device group with the algorithms in §4.4.1, and checks whether the model can be put on the group under the memory constraint. For all valid selections, it runs the simulator and computes SLO attainment. It then picks the top-k solutions and enters the next iteration. The algorithm terminates when no more replicas can be put into any groups.

The complexity of algorithm 3 is $O(MGRSB)$, where M is the number of models, G is the number of groups, R is the number of replicas we can put according to the memory constraint, S is the number of requests in the workload (the simulation time is proportional to the number of the requests) and B is the beam size. It runs reasonably fast for our medium-scale cluster when the number of requests is small. When the number of requests is very large, we propose another heuristic to accelerate: Instead of using the simulator to evaluate all (model, group) pairs at each iteration, we can run the simulator only once and place a model with the most unserved requests in an available group with the lowest utilization. This reduces the time complexity to $O((M+G)RS)$. We find this heuristic gives solutions with SLO attainment higher than 98% of the SLO attainment get by the original algorithm in our benchmarks.

algorithm 4 enumerates different group partitions and model-parallel configurations and picks the best one via multiple calls to algorithm 3. When designing algorithm 4, the first phenomenon we notice is that putting small and large models in the same group causes convoy effects, where the requests of small models have to wait for the requests of large models and miss the SLO. Therefore, in algorithm 4, we first cluster models into model buckets. Each bucket contains a set of models with relatively similar sizes and every model is assigned to one and only one bucket. Specifically, the function `get_potential_model_buckets` returns all the possible model bucket partitions that separate models whose latency difference is larger than a threshold into different disjoint buckets. We then enumerate all the potential ways to assign the devices to each bucket in `get_potential_device_buckets`.

Because different buckets include a disjoint set of models, we can then figure out the optimal placement for each bucket individually. For each bucket, we enumerate possible ways to partition the devices in the bucket into several groups in `get_potential_group_partitions` and enumerate the potential parallel configurations for each group with the method in `get_potential_parallel_configs`. We then call algorithm 3 with `greedy_placement` to place models in the model bucket to the groups in the device bucket. We send the whole

Algorithm 4 Enumeration-Based Group Partition and Model-Parallel Configuration Selection.

```
Input: Model list $M$, cluster $C$, workload $W$.
Output: The placement $best\_plm$.
  $best\_plm \leftarrow \emptyset$
  $\mathcal{B} \leftarrow$ get_potential_model_buckets($M$)
  for $(B_1, B_2, \dots, B_k) \in \mathcal{B}$ do
      $\mathcal{H} \leftarrow$ get_potential_device_buckets($C, B, k$)
      for $(H_1, H_2, \dots, H_k) \in \mathcal{H}$ do
          // Get the placement for each bucket individually.
          for $i$ from 1 to $k$ do
              $plm_i^* \leftarrow \emptyset$
              $\mathcal{G} \leftarrow$ get_potential_group_partitions($H_i$)
              for $G \in \mathcal{G}$ do
                  $\mathcal{P} \leftarrow$ get_potential_parallel_configs($G$)
                  for $P \in \mathcal{P}$ do
                      $plm \leftarrow$ greedy_selection($B_i, G, P, W$)
                      if $plm.slo\_att > plm_i^*.slo\_att$ then
                          $plm_i^* \leftarrow plm$
                      end if
                  end for
              end for
          end for
          $plm^* \leftarrow$ concat($plm_1^*, ..., plm_k^*$)
          if $plm^*.slo\_att > best\_plm.slo\_att$ then
              $best\_plm \leftarrow plm^*$
          end if
      end for
  end for
  return $best\_plm$
```

workload W to algorithm 3, which ignores the requests that hit the models outside of the current bucket. Finally, a complete solution is got by concatenating the solutions for all buckets. The algorithm returns the best solution it finds during the enumerative search process.

Enumerating all possible choices can be slow, so we use the following heuristics to prune the search space. Intuitively, we want the different buckets to serve a similar number of requests per second. Therefore, we eliminate the bucket configurations with high discrepancies in the estimated number of requests it can serve per second for each bucket. Additionally, in `get_potential_group_partitions` and `get_potential_parallel_configs`, we assume all groups have the same size and the same parallel configurations except for the last group which is used when the number of devices is not divisible by the group size.

4.4.3 Runtime Scheduling

We use a simple policy to dispatch and schedule the requests at runtime. All requests are sent to a centralized controller. The controller dispatches each request to the group with the shortest queue length. Each group manages a first-come-first-serve queue. When a group receives a request, it checks whether it can serve the request under SLO and rejects the request if it cannot. This is possible because the execution time of a DNN model is very predictable and can be got in advance by profiling [50]. In most of our experiments, we do not include advanced runtime policies such as batching [50], swapping, and preemption [51]. These techniques are complementary to model parallelism. Nevertheless, we discuss how they fit into our system.

Batching. Batching multiple requests of the same model together can increase the GPU utilization and thus increase the throughput of a serving system. In our system, we do find batching is helpful, but the gain is limited. This is because we mainly target large models and a small batch size can already fully saturate the GPU, which is verified in §4.6.5. To isolate the benefits of model parallelism and make the results more explainable, we decide to disable any batching in this chapter except for the experiments in §4.6.5.

Preemption. The optimal scheduling decision often depends on future arrivals, and leveraging preemption can help correct previous suboptimal decisions. The first-come-first-serve policy may result in convoy effects when models with significantly different execution times are placed in the same group. We anticipate a least-slack-time-first policy with preemption can alleviate the problems [31].

Swapping. The loading overheads from the CPU or Disk to GPU memory are significant for large models, which is the target of this chapter, so we do not implement swapping in AlpaServe. We assume all models are placed on the GPUs. This is often required due to tight SLOs and high rates, especially for large models. The placement of models in AlpaServe can be updated in the periodic re-placement (e.g., every 24 hours).

Fault tolerance. While the current design of AlpaServe does not have fault tolerance as a focus, we acknowledge several potential new challenges for fault tolerance: With model parallelism, the failure of a single GPU could cause the entire group to malfunction. Additionally, the use of a centralized controller presents a single point of failure.

4.5 Implementation

We implement a real system and a simulator for AlpaServe with about 4k lines of code in Python. The real system is implemented on top of an existing model-parallel training system, Alpa [165]. We extend its auto-parallelization algorithms for inference settings to get the model-parallel strategies. We then launch an Alpa runtime for each group and dispatch requests to these groups via a centralized controller.

The simulator is a continuous-time, discrete-event simulator [116]. The simulator maintains a global clock and simulates all requests and model executions on the cluster. Because the simulator only models discrete events, it is orders of magnitude faster than the real experiments. In our experiment, it takes less than 1 hour for a 24-hour trace. The fidelity of the simulator is very high because of the predictability of DNN model execution, which is verified in §4.6.1.

4.6 Evaluation

In this section, we evaluate AlpaServe's serving ability under a variety of model and workload conditions. The evaluation is conducted on a range of model sizes, including those that do and do not fit into a single GPU, and we show that AlpaServe consistently outperforms strong baselines across all model sizes. In addition, we evaluate the robustness of AlpaServe against changing arrival patterns and do ablation studies of our proposed techniques. Evaluation results show that AlpaServe can greatly improve various performance metrics. Specifically, AlpaServe can choose to save up to 2.3× devices, handle 10× higher rates, 6× more burstiness, or 2.5× more stringent SLO, while meeting the latency SLOs for over 99% requests.

4.6.1 Experiment Setup

Cluster testbed. We deploy AlpaServe on a cluster with 8 nodes and 64 GPUs. Each node is an AWS EC2 p3.16xlarge instance with 8 NVIDIA Tesla V100 (16GB) GPUs.

Model setup. Since Transformer [138] is the default backbone for large models, we choose two representative large Transformer model families: BERT [34] and GShard MoE [76] for evaluation.[2] In ML practice, the large model weights are usually pretrained and then finetuned into different versions for different tasks. Hence, for each model family, we select several most commonly used model sizes [18], and then create multiple model instances at each size for experimentation. Also, we design some model sets to test the serving systems

[2]In this chapter, we focus on non-autoregressive large models which perform inference with one forward pass, but note that the techniques proposed in this chapter can be extended to auto-regressive models like GPT-3.

Name	Size	Latency (ms)	S1	S2	S3	S4
BERT-1.3B	2.4 GB	151	32	0	10	0
BERT-2.7B	5.4 GB	238	0	0	10	0
BERT-6.7B	13.4 GB	395	0	32	10	0
BERT-104B	208 GB	4600	0	0	0	4
MoE-1.3B	2.6 GB	150	0	0	10	0
MoE-2.4B	4.8 GB	171	0	0	10	0
MoE-5.3B	10.6 GB	234	0	0	10	0

Table 4.1: The first three columns list the sizes and inference latency of the models. The latency is measured for a single query with a sequence length of 2048 on a single GPU. BERT-104B's latency is reported using a minimal degree of inter-op parallelism. The latter columns list the number of instances for each model in different model sets named as S1-S4.

under different model conditions; details about model sizes, their inference latency on testbed GPUs, and the number of model instances in each model set are provided in Tab. 4.1.

Metrics. We use *SLO attainment* as the major evaluation metric. Under a specific SLO attainment goal (say, 99%), we concern with another four measures: (1) the minimal number of devices the system needs, (2) the maximum average request rate, (3) the maximum traffic burstiness the system can support, and (4) the minimal SLO the system can handle. We are particularly interested in a SLO attainment of 99% (indicated by vertical lines in all curve plots), but will also vary each variable in (1) - (4) and observe how the SLO attainment changes.

Simulator fidelity. We want to study the system behavior under extensive models, workload, and resource settings, but some settings are just beyond the capacity of our testbed. Also, it is cost- and time-prohibitive to perform all experiments on the testbed for the days-long real traces. To mitigate the problem, we use the simulator introduced in §4.5 for the majority of our experiments, noticing that DNN model execution [50] has high predictability, even under parallel settings [165, 76]. We study the fidelity of the simulator in Tab. 4.2. Given two model placement algorithms, we compare the SLO attainment reported by the simulator and by real runs on our testbed under different *SLO Scales*. The error is less than 2% in all cases, verifying the accuracy of our simulator. Additionally, we conduct experiments on cluster testbed for results in §4.6.3.

4.6.2 End-to-end Results with Real Workloads

In this section, we compare AlpaServe against baseline methods on publicly available real traces.

SLO	Selective Replication		AlpaServe	
Scale	Real System	Simulator	Real System	Simulator
0.5x	00.0%	00.0%	33.3%	33.3%
1x	00.0%	00.0%	53.5%	53.2%
1.5x	29.7%	30.2%	64.1%	64.7%
2x	36.9%	36.8%	79.0%	80.6%
3x	49.5%	48.5%	91.4%	92.1%
4x	58.6%	57.8%	96.4%	96.5%
5x	64.9%	64.0%	97.6%	97.9%
10x	83.1%	82.6%	100.0%	99.7%

Table 4.2: Comparison of the SLO attainment reported by the simulator and the real system under different SLO scales.

Workloads. There does not exist an open-source production ML inference trace to the best of our knowledge. Therefore, we use the following two production traces as a proxy: Microsoft Azure function trace 2019 (MAF1) [121] and 2021 (MAF2) [160]. They were originally collected from Azure serverless function invocations in two weeks, and have been repurposed for ML serving research [14, 59]. The two traces exhibit distinct traffic patterns. In MAF1, each function receives steady and dense incoming requests with gradually changing rates; in MAF2, the traffic is very *bursty* and is distributed across functions in a highly *skewed* way – some function receives orders of magnitude more requests than others. Note that most previous works [50] are evaluated on MAF1 only. Since there are more functions than models, following previous work [14, 59], given a model set from Tab. 4.1, we round-robin functions to models to generate traffic for each model.

Setup. SLO attainment depends on many factors. For each metric (1) - (4) mentioned in §4.6.1, we set a default value, e.g., the default SLO is set as tight as 5× inference latency (SLO Scale=5). This forms a *default setting*, given which, we then vary one variable (while fixing others) at a time and observe how it affects the resulting SLO attainment. To change the two variables (3) and (4), which characterize traffic patterns, we follow Clockwork [50] and Inferline [27] and slice the original traces into time windows, and fit the arrivals in each time window with a Gamma Process parameterized by rate and coefficient of variance (CV). By scaling the rate and CV and resampling from the processes, we can control the rate and burstiness, respectively.

Baselines. We compare AlpaServe to two baseline methods: (1) *Selective Replication (SR)*: use AlpaServe's placement algorithm without model parallelism, which mimics the policy of a wide range of existing serving systems [28, 123]; (2) *Clockwork++*: an improved version of the state-of-the-art model serving system Clockwork [50]. The original Clockwork continuously swaps models into and out of GPUs. This helps for very small models (e.g., w/ several

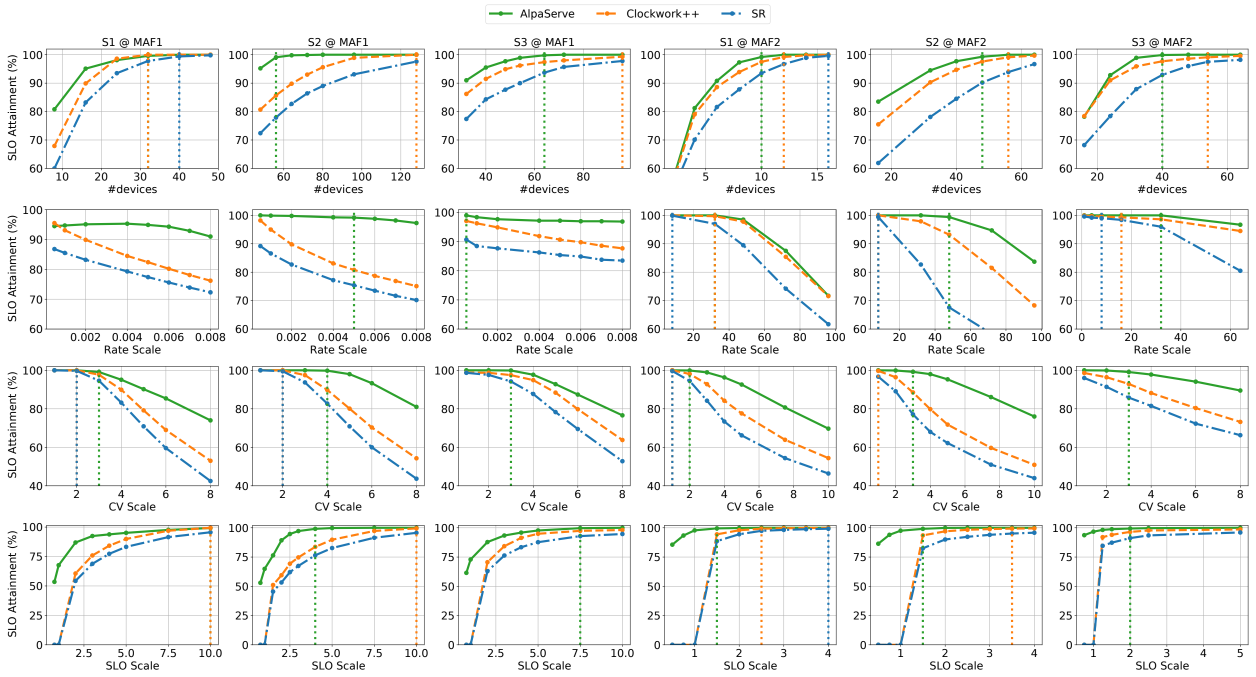


Figure 4.12: SLO attainment under various settings. In column S1@MAF1, we replay the MAF1 trace on the model set S1, and so on. In each row, we focus on one specific metric mentioned in §4.6.2 to see how its variation affects the performance of each serving system. If any, the dotted vertical line shows when the system can achieve 99% SLO attainment.

million parameters) but incurs significant swapping overheads on larger models. For fair comparisons, we implement Clockwork++ in our simulator, which swaps models following Clockwork's replacement strategy at the boundary of every two windows[3] of the trace using SR's algorithm, but *assuming zero swapping overheads*. We believe it represents a hypothetical upper bound of Clockwork's performance. Since all the baselines can only support models that can fit into one GPU memory,[4] we use model set S1, S2 and S3 from Tab. 4.1 in this experiment.

SLO attainment vs. cluster size. fig. 4.12's first row shows the SLO attainment with varying cluster sizes when serving a specific (model set, trace) pair. AlpaServe outperforms the two baselines at all times and uses far fewer devices to achieve 99% SLO attainment thanks to model parallelism. By splitting one model replica onto N devices, AlpaServe can achieve similar throughput as if N replica were created for replication-only methods; but note

[3]For MAF1, we follow Clockwork to set the window size as 60 seconds. For MAF2, we set it as 5.4K seconds.

[4]In our cluster testbed, the per-GPU memory is 16GB, but the actual available space for model weights is around 13GB due to the need to store activations and other runtime context.

AlpaServe uses only one replica of memory. Surprisingly, although we let Clockwork++ adjust to the traffic dynamically with zero overhead, AlpaServe still wins with a static placement; this is because model-parallel placement is by nature more robust to bursty traffic.

It is worth noting that replication-only methods can at most place 2 replicas of BERT-2.6B on a V100 (13GB memory budget), resulting in a substantial memory fraction, while model parallelism can avoid such memory fractions and enable more flexible placement of models.

SLO attainment vs. rate. fig. 4.12's 2nd row varies the rate of the workloads. For a stable trace like MAF1, AlpaServe can handle a much higher rate than baselines. While for a skewed and highly dynamic trace MAF2, whose traffic is dominated by a few models and changes rapidly, the replication-based methods have to allocate the majority of the GPUs to create many replicas for "hot" models to combat their bursty traffic; those GPUs, however, may go idle between bursts, even with frequent re-placement as in Clockwork++. In AlpaServe, each model needs fewer replicas to handle its peak traffic.

SLO attainment vs. CV. fig. 4.12's 3rd row varies the CV of the workloads. The traffic becomes more bursty with a higher CV, which aggravates the queuing effect of the system and increases the possibility of SLO violation. The traditional solution to handle burstiness is by over-provision, wasting a lot of resources. AlpaServe reveals a hidden opportunity to handle this by model parallelism.

SLO attainment vs. SLO. fig. 4.12's 4th row shows the effect of different SLO. Previous work [50] which targets serving small models usually sets SLO to hundreds of milliseconds, even though the actual inference latency is less than 10 ms. Thanks to the intra-op parallelism, AlpaServe can maintain good performance under similar SLO when serving large models, whose inference latency can be over 100 ms. When SLO is tight, even less than the model inference time, AlpaServe favors intra-op parallelism to reduce the inference latency, which also reduces AlpaServe's peak throughput due to the communication overhead but can make more requests to meet their SLO. When SLO becomes looser, AlpaServe will automatically switch to use more inter-op parallelism to get higher throughput.

4.6.3 Serving Very Large Models

Today's large models may possess hundreds of billions of parameters [18, 94, 159]. To serve large models at this scale, the common practice in production is to choose the model parallelism strategy manually and use dedicated GPUs for each model [153]. To show AlpaServe has improved capability in serving very large models, we deploy model set S4 on our testbed, each requiring at least 16 GPUs to serve in terms of memory usage. As baselines, for each model, we enumerate all combinations of inter- and intra-op parallelisms on 16 GPUs. In

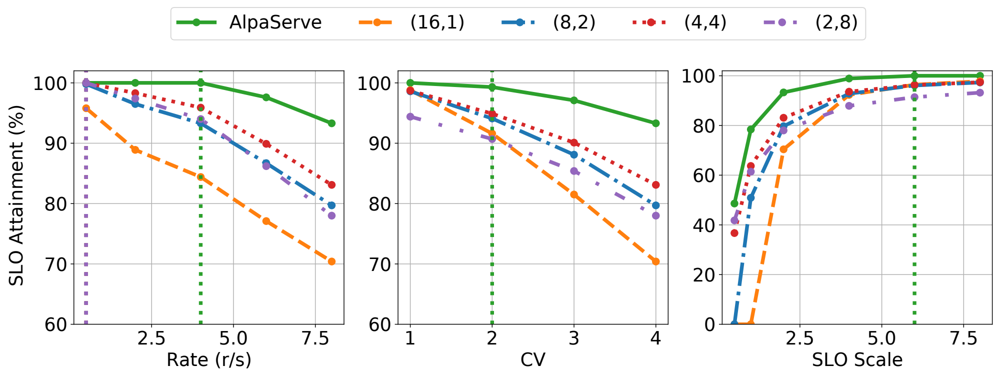


Figure 4.13: SLO attainment as we vary the rate, CV, and SLO scale. (8,2) means 8-way inter-op parallelism and in each pipeline stage using 2-way intra-op parallelism.

contrast, AlpaServe searches for the optimal GPU group allocation and model placement according to the arrival traffic and tries to achieve statistical multiplexing.

Offered load. In the default setting, the traffic is generated via a Gamma Process with an average rate of 8 requests/s and CV of 4. We then split the requests to each model following a power law distribution with an exponent of 0.5 to simulate the real-world skewness.[5] Similar to §4.6.2, we vary one of the rate, CV, or SLO in the default setting to see how each factor contributes to the resulting performance. It is worth noting that all results presented in this section are obtained via real execution on the testbed cluster.

SLO attainment. fig. 4.13 shows the SLO attainment of each system under various settings. Although enumerating parallelism strategies and selecting the best can improve performance, it still remains a substantial gap compared to AlpaServe. This means that the traditional way of using dedicated GPUs to serve large models is not ideal. We check the solution of AlpaServe and find it slices the cluster evenly into two groups, each with the (4, 8) inter-/intra-op parallel configuration, and groups the models in a way that balances the requests between two groups. This further proves that our motivation in §4.3.1 still holds for extremely large models. By space-sharing the devices, AlpaServe can exploit new opportunities for statistical multiplexing, which is advantageous for bursty workloads but largely under-explored by prior work.

4.6.4 Robustness to Changing Traffic Patterns

Until now, AlpaServe's good performance is based on the assumption we make in its placement algorithm that we know the arrival process in advance. In practice, the arrival process

[5]Uniform split yielded similar results.

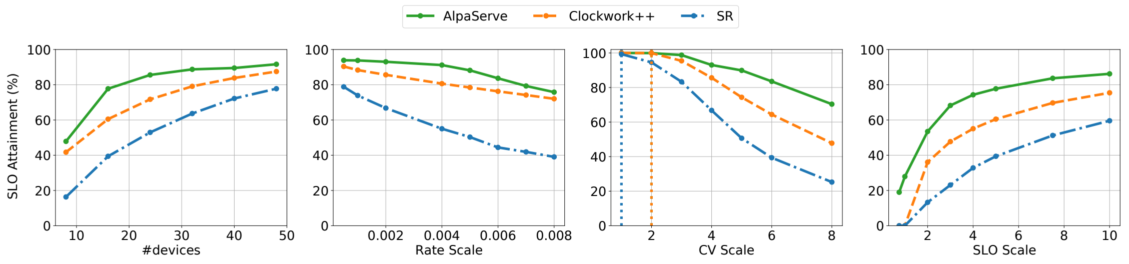


Figure 4.14: The actual arrival traffic for AlpaServe and SR is different from what their algorithms are assumed, while Clockwork++ runs directly on the actual traffic.

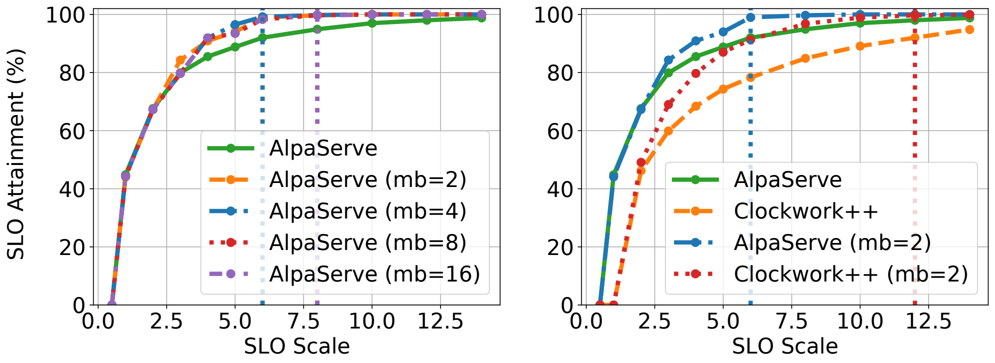


Figure 4.15: SLO Attainment when batching is enabled. mb=2 means the maximum batch size is 2.

can be approximated using historical traces but the unavoidable real-world variance may make the prediction inaccurate. In this experiment, we study how AlpaServe performs if the traffic patterns change.

We reuse the same setting for S2@MAF1 in §4.6.2, but this time for AlpaServe and SR, we randomly slice two one-hour traces from MAF1, one is what their algorithms are assumed, while the other one is used as the actual arrival process. While for Clockwork++, we still run its algorithm directly on the actual arrival process to respect its online nature. Similarly, we vary different factors and compute the SLO attainment for each system. We repeat the experiments three times and show the average results in Fig. 4.14.

Unsurprisingly, SR's performance drops significantly when traffic changes. By contrast, AlpaServe maintains good performance and still outperforms Clockwork++, an online adjustment algorithm, using a static placement generated from substantially different traffic patterns. This confirms that, in face of highly-dynamic traffic patterns, statistical multiplexing with model parallelism is a simple and better alternative than existing replication- or replacement-based algorithms.

4.6.5 Benefits of Dynamic Batching

Batching is a common optimization in other serving systems [50, 97, 98] and the choice of batch size is critical to the performance because it can increase GPU utilization and thus increase the system throughput. However, in large model scenarios, the benefit of batching is limited mainly due to two reasons. First, for large models, a small batch size will saturate the GPU, which means there is little gain to batching more requests. Second, the execution latency grows linearly with the batch size [123], so when the SLO is tight (say SLO Scale is less than 2), batching is simply not a choice.

To isolate the benefits of model parallelism and make the results more explainable, we decide to disable any batching in other experiments but prove that the batching strategy is purely orthogonal to the scope of this chapter in this subsection. To prove this, we implement a standard batching algorithm in AlpaServe and evaluate its performance.

Batching strategy. When a request arrives, it will get executed immediately if any device group is available. Otherwise, it will be put into a per-model requests queue for batching. When a device group becomes idle, it will choose a model which has a replica on it and batch as many requests as possible from the requests queue of the model while satisfying the SLO requirements.

Setup. As the model size increases, the potential benefit of batching decreases. Therefore, we choose to evaluate model set S1. We generate a synthetic Gamma Process traffic with an average rate of 4 requests/s and a CV of 4 for each model.

SLO attainment. fig. 4.15 (left) shows the SLO attainment achieved by AlpaServe with different maximum batch size settings under various SLO scales. When the SLO requirement is tight, any batching will violate the SLO so there is no gain with batching enabled. Also, although we choose to serve the smallest model in Tab. 4.1, a small batch size like 2 combined with a long sequence length of 2048 already saturates the GPU, so a larger maximum batch size brings no performance improvement. fig. 4.15 (right) compares the improvement for AlpaServe and Clockwork++ with our batching algorithm enabled.[6] When the SLO requirement becomes loose, both AlpaServe and Clockwork++ have better SLO attainment to some extent. Since AlpaServe's performance is good even without batching and batched requests with different batch sizes will incur stage imbalance and pipeline bubble in inter-op parallel, the absolute improvement of Clockwork++ is slightly better.

4.6.6 Ablation Study

In this section, we study the effectiveness of our proposed auto-parallelization (§4.4.1) and placement algorithms (§4.4.2).

[6]SR is left out to make the figure clearer as it is worse than Clockwork++.

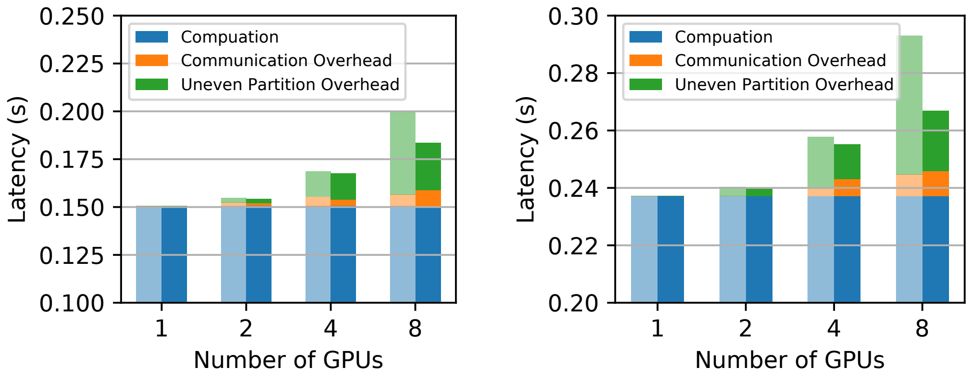


Figure 4.16: Comparison of the model parallel overhead between manual partition (lighter color) and the partition found by the automatic algorithm (darker color).

Benefits of auto-parallelization. We show that the auto-parallelization ability allows AlpaServe to not only generalize to arbitrary model architectures but even also reduce parallelism overheads – hence improved serving performance (see §4.3.3 for more discussion). To see that, typical manual model-parallel parallelization strategies offered in *de facto* systems [2, 94, 96] is to assign an equal number of (transformer) layers to each pipeline stage. These strategies often fail to create balanced workloads across distributed GPUs because contemporary large models have heterogeneous layers, such as embedding operations. The extensions introduced in §4.4.1 automatically partition the models at the computational graph level and generate nearly-balanced stages. Empirically, as shown in Fig. 4.16, for 8 pipeline stages, auto-parallelization reduces the total overhead by 32.9% and 46.7% for Transformer 1.3B and 2.6B respectively, which is necessary for achieving good serving performance when model parallelism is used for serving.

Effectiveness of the placement algorithm. We now test the effectiveness of our placement algorithm on a synthetic workload. We serve the most challenging model set S3 (Tab. 4.1) on our testbed. The rate distribution of the models follows a power law distribution. The arrival pattern of each model is a Gamma process. Three variants of the placement algorithms are evaluated. *Round robin* means placing models in a round-robin fashion and using 4-stage pipelines for all groups. *Greedy placement* uses our greedy placement and 4-stage pipeline for all groups. *Greedy placement + Group partitioning* performs greedy placement plus group partitioning search. As shown in Fig. 4.17, both placement and group partitioning are necessary to achieve good SLO attainment. In the left subfigure, the group partitioning increases the rate by 1.5× compared to greedy placement without group partitioning over 99% SLO attainment, while round robin can never reach 99% SLO attainment. In the right subfigure, the group partitioning increases the traffic burstiness that can be handled to meet 99% SLO attainment by 1.3×.

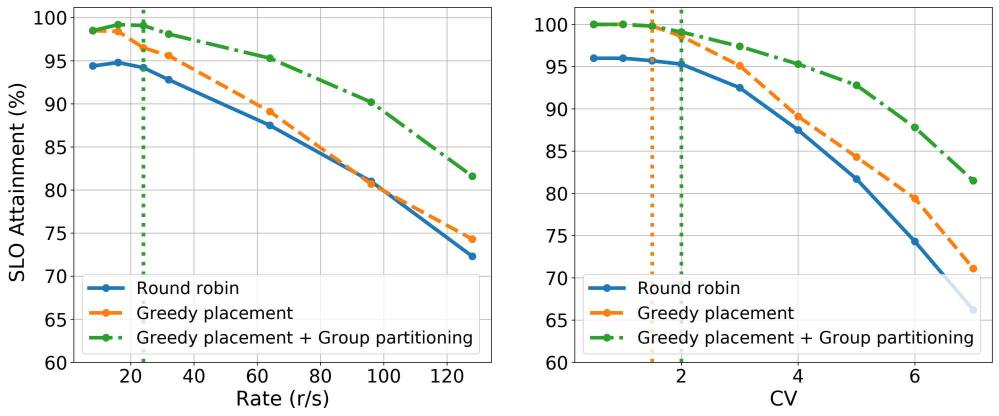


Figure 4.17: Ablation study of placement algorithms.

4.7 Related Work

Model serving systems. There has been a proliferation of model serving systems recently. These range from general-purpose production-grade systems like TensorFlow Serving [98] and NVIDIA Triton [97], which are widely used but do not provide any support for automatic placement or latency constraints. They also include systems that are optimized for single-model serving [153] or serving of specific classes of models (e.g., transformers) [153, 39, 166]. AlpaServe targets a broader set of models and features than these systems.

For SLO-aware, distributed serving, most serving systems ignore placement-level interactions between models. Clockwork [50], for instance, primarily focuses on predictability; when scheduling, it greedily loads and executes models on available GPUs. Shepherd [158] utilizes preemption to correct sub-optimal scheduling decisions. For large models, loading model weights and preemption can easily overwhelm practical SLOs. Other systems like Clipper [28], Infaas [117], and DVABatch [29] also do not reason about the latencies of co-located models.

Nexus [123] is very related to our work in that it examines the placement of models; however, Nexus is an example of a system that takes the traditional replication approach described in §4.3 and, thus, misses a broad class of potential parallelization strategies that we explore in this chapter.

Inference optimizations for large models. AlpaServe is complementary to another large body of work on optimizations for inference over large models. These include techniques like quantization [33], distillation [118], offloading [2], better operator parallelism [108], and CUDA kernel optimization [30, 60]. Some of these optimizations are intended to stem the tide of increasing model sizes; however, all of these gains are partial— the challenge of serving large models has continued to escalate rapidly despite these efforts.

Model parallelism for training. AlpaServe is largely orthogonal to the large body of work on model parallelism in training [165, 94, 57, 111, 83]. As described in §4.3, serving presents a unique set of constraints and opportunities not found in training workloads. Where these systems do intersect with AlpaServe, however, is in their methods for implementing model parallelism along various dimensions. In particular, AlpaServe builds on some of the parallelization techniques introduced in [165].

Resource allocation and multiplexing. The problem of how to multiplex limited resources to the incoming requests is one of the oldest topics in computer science and has been studied in different application domains [112, 87, 11]. Recent work on DL scheduling uses swapping [9], preemption [52], interleaving [162], and space-sharing [148] to realize fine-grained resource sharing. Rather, the contribution of this chapter is a deep empirical analysis of the applications of these ideas to an emerging space: the serving of multiple large models.

4.8 Conclusion and Future Work

In this chapter, we presented AlpaServe, a system for prediction servings of multiple large deep-learning models. The key innovation of AlpaServe is integrating model parallelism into multi-model serving. Because of the inherent overheads of model parallelism, such parallelism is traditionally applied conservatively—reserved for cases where models simply do not fit within a single GPU or execute within the required SLO. AlpaServe demonstrates that model parallelism is useful for many other scenarios, quantifies the tradeoffs, and presents techniques to automatically navigate that tradeoff space.

In the future, we will extend AlpaServe to more complicated scenarios, including serving multiple parameter-efficient adapted models (e.g., LoRA [55]), models with dependencies, and autoregressive models [18].

Chapter 5

vLLM: Efficient Memory Management for Large Language Model Serving with PagedAttention

Chapter 4 discusses how to utilize model parallelism to optimize the serving performance for general large deep learning models. However, serving a large language model (LLM) involves its unique challenges, especially due to its *autoregressive dependency* that requires the output tokens to be generated one by one, which heavily under-utilizes the parallel accelerators like GPUs. Therefore, high throughput serving of large language models typically requires batching sufficiently many requests at a time.

However, existing systems struggle to batch more requests because the key-value cache (KV cache) memory for each request is huge and grows and shrinks dynamically. When managed inefficiently, this memory can be significantly wasted by fragmentation and redundant duplication, limiting the batch size. To address this problem, we propose PagedAttention [74], an attention algorithm inspired by the classical virtual memory and paging techniques in operating systems. On top of it, we build vLLM, an LLM serving system that achieves (1) near-zero waste in KV cache memory and (2) flexible sharing of KV cache within and across requests to further reduce memory usage. Our evaluations show that vLLM improves the throughput of popular LLMs by 2-4× with the same level of latency compared to the state-of-the-art systems, such as FasterTransformer and Orca. The improvement is more pronounced with longer sequences, larger models, and more complex decoding algorithms.

5.1 Introduction

The emergence of large language models like GPT [102, 18] and PaLM [25] have enabled new applications such as programming assistants [46, 21] and universal chatbots [101, 47] that are starting to profoundly impact our work and daily routines. Many cloud companies [99, 120]

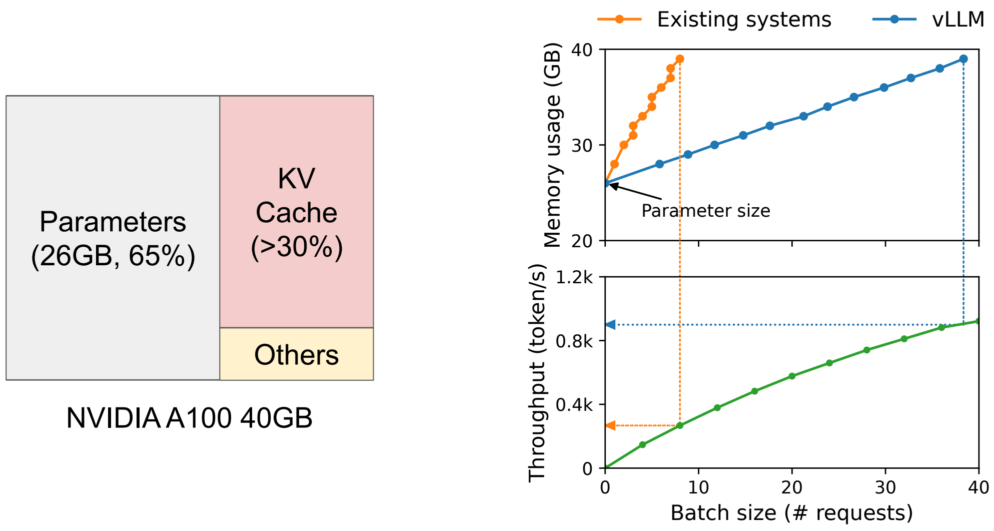


Figure 5.1: *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. *Right:* vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [96, 153], leading to a notable boost in serving throughput.

are racing to provide these applications as hosted services. However, running these applications is very expensive, requiring a large number of hardware accelerators such as GPUs. According to recent estimates, processing an LLM request can be 10× more expensive than a traditional keyword query [115]. Given these high costs, increasing the throughput—and hence reducing the cost per request—of *LLM serving* systems is becoming more important.

At the core of LLMs lies an autoregressive Transformer model [138]. This model generates words (tokens), *one at a time*, based on the input (prompt) and the previous sequence of the output's tokens it has generated so far. For each request, this expensive process is repeated until the model outputs a termination token. This sequential generation process makes the workload *memory-bound*, underutilizing the computation power of GPUs and limiting the serving throughput.

Improving the throughput is possible by batching multiple requests together. However, to process many requests in a batch, the memory space for each request should be efficiently managed. For example, Fig. 5.1 (left) illustrates the memory distribution for a 13B-parameter LLM on an NVIDIA A100 GPU with 40GB RAM. Approximately 65% of the memory is allocated for the model weights, which remain static during serving. Close to 30% of the memory is used to store the dynamic states of the requests. For Transformers,

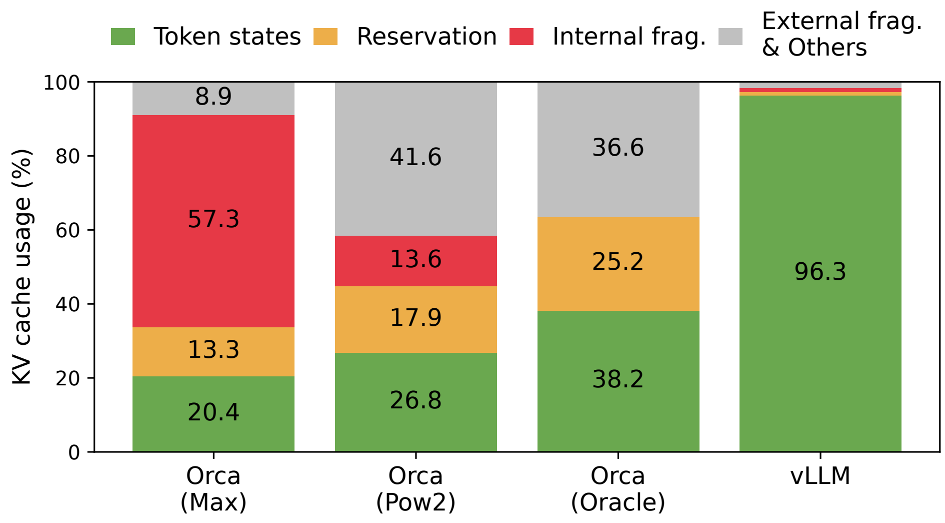


Figure 5.2: Average percentage of memory wastes in different LLM serving systems during the experiment in §5.6.2.

these states consist of the key and value tensors associated with the attention mechanism, commonly referred to as *KV cache* [108], which represent the context from earlier tokens to generate new output tokens in sequence. The remaining small percentage of memory is used for other data, including activations – the ephemeral tensors created when evaluating the LLM. Since the model weights are constant and the activations only occupy a small fraction of the GPU memory, the way the KV cache is managed is critical in determining the maximum batch size. When managed inefficiently, the KV cache memory can significantly limit the batch size and consequently the throughput of the LLM, as illustrated in Fig. 5.1 (right).

In this chapter, we observe that existing LLM serving systems [153, 96] fall short of managing the KV cache memory efficiently. This is mainly because they store the KV cache of a request in contiguous memory space, as most deep learning frameworks [104, 98] require tensors to be stored in contiguous memory. However, unlike the tensors in the traditional deep learning workloads, the KV cache has unique characteristics: it dynamically grows and shrinks over time as the model generates new tokens, and its lifetime and length are not known a priori. These characteristics make the existing systems' approach significantly inefficient in two ways:

First, the existing systems [153, 96] suffer from internal and external memory fragmentation. To store the KV cache of a request in contiguous space, they *pre-allocate* a contiguous chunk of memory with the request's maximum length (e.g., 2048 tokens). This can result in severe internal fragmentation, since the request's actual length can be much shorter than its maximum length (e.g., Fig. 5.11). Moreover, even if the actual length is known a priori, the pre-allocation is still inefficient: As the entire chunk is reserved during the request's lifetime, other shorter requests cannot utilize any part of the chunk that is currently unused. Besides, external memory fragmentation can also be significant, since the pre-allocated size can be different for each request. Indeed, our profiling results in Fig. 5.2 show that only 20.4% - 38.2% of the KV cache memory is used to store the actual token states in the existing systems.

Second, the existing systems cannot exploit the opportunities for memory sharing. LLM services often use advanced decoding algorithms, such as parallel sampling and beam search, that generate multiple outputs per request. In these scenarios, the request consists of multiple sequences that can partially share their KV cache. However, memory sharing is not possible in the existing systems because the KV cache of the sequences is stored in separate contiguous spaces.

To address the above limitations, we propose *PagedAttention*, an attention algorithm inspired by the operating system's (OS) solution to memory fragmentation and sharing: *virtual memory with paging*. PagedAttention divides the request's KV cache into blocks, each of which can contain the attention keys and values of a fixed number of tokens. In PagedAttention, the blocks for the KV cache are not necessarily stored in contiguous space. Therefore, we can manage the KV cache in a more flexible way as in OS's virtual memory: one can think of blocks as pages, tokens as bytes, and requests as processes. This design alleviates internal fragmentation by using relatively small blocks and allocating them on demand. Moreover, it eliminates external fragmentation as all blocks have the same size. Finally, it enables memory sharing at the granularity of a block, across the different sequences associated with the same request or even across the different requests.

In this work, we build *vLLM*, a high-throughput distributed LLM serving engine on top of PagedAttention that achieves near-zero waste in KV cache memory. vLLM uses block-level memory management and preemptive request scheduling that are co-designed with PagedAttention. vLLM supports popular LLMs such as GPT [18], OPT [159], and LLaMA [137] with varying sizes, including the ones exceeding the memory capacity of a single GPU. Our evaluations on various models and workloads show that vLLM improves the LLM serving throughput by 2-4× compared to the state-of-the-art systems [153, 96], without affecting the model accuracy at all. The improvements are more pronounced with longer sequences, larger models, and more complex decoding algorithms (§5.4.3). In summary, we make the following contributions:

- We identify the challenges in memory allocation in serving LLMs and quantify their impact on serving performance.
- We propose PagedAttention, an attention algorithm that operates on KV cache stored in non-contiguous paged memory, which is inspired by the virtual memory and paging

in OS.

- We design and implement vLLM, a distributed LLM serving engine built on top of PagedAttention.
- We evaluate vLLM on various scenarios and demonstrate that it substantially outperforms the previous state-of-the-art solutions such as FasterTransformer [96] and Orca [153].

5.2 Background

In this section, we describe the generation and serving procedures of typical LLMs and the iteration-level scheduling used in LLM serving.

5.2.1 Transformer-Based Large Language Models

The task of language modeling is to model the probability of a list of tokens $(x_1, \ldots, x_n)$. Since language has a natural sequential ordering, it is common to factorize the joint probability over the whole sequence as the product of conditional probabilities (a.k.a. *autoregressive decomposition* [12]):

$$P(x) = P(x_1) \cdot P(x_2 \mid x_1) \cdots P(x_n \mid x_1, \ldots, x_{n-1}). \tag{5.1}$$

Transformers [138] have become the de facto standard architecture for modeling the probability above at a large scale. The most important component of a Transformer-based language model is its *self-attention* layers. For an input hidden state sequence $(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$, a self-attention layer first applies linear transformations on each position i to get the query, key, and value vectors:

$$q_i = W_q x_i, \; k_i = W_k x_i, \; v_i = W_v x_i. \tag{5.2}$$

Then, the self-attention layer computes the attention score a_{ij} by multiplying the query vector at one position with all the key vectors before it and compute the output o_i as the weighted average over the value vectors:

$$a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^{i} \exp(q_i^\top k_t / \sqrt{d})}, \; o_i = \sum_{j=1}^{i} a_{ij} v_j. \tag{5.3}$$

Besides the computation in Eq. 5.3, all other components in the Transformer model, including the embedding layer, feed-forward layer, layer normalization [7], residual connection [54], output logit computation, and the query, key, and value transformation in Eq. 5.2, are all applied independently position-wise in a form of $y_i = f(x_i)$.

5.2.2 LLM Service & Autoregressive Generation

Once trained, LLMs are often deployed as a conditional generation service (e.g., completion API [99] or chatbot [101, 47]). A request to an LLM service provides a list of *input prompt* tokens $(x_1, \ldots, x_n)$, and the LLM service generates a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$ according to Eq. 5.1. We refer to the concatenation of the prompt and output lists as *sequence*.

Due to the decomposition in Eq. 5.1, the LLM can only sample and generate new tokens one by one, and the generation process of each new token depends on all the *previous tokens* in that sequence, specifically their key and value vectors. In this sequential generation process, the key and value vectors of existing tokens are often cached for generating future tokens, known as *KV cache*. Note that the KV cache of one token depends on all its previous tokens. This means that the KV cache of the same token appearing at different positions in a sequence will be different.

Given a request prompt, the generation computation in the LLM service can be decomposed into two phases:

The prompt phase takes the whole user prompt $(x_1, \ldots, x_n)$ as input and computes the probability of the first new token $P(x_{n+1} \mid x_1, \ldots, x_n)$. During this process, also generates the key vectors $k_1, \ldots, k_n$ and value vectors $v_1, \ldots, v_n$. Since prompt tokens $x_1, \ldots, x_n$ are all known, the computation of the prompt phase can be parallelized using matrix-matrix multiplication operations. Therefore, this phase can efficiently use the parallelism inherent in GPUs.

The autoregressive generation phase generates the remaining new tokens sequentially. At iteration t, the model takes one token x_{n+t} as input and computes the probability $P(x_{n+t+1} \mid x_1, \ldots, x_{n+t})$ with the key vectors $k_1, \ldots, k_{n+t}$ and value vectors $v_1, \ldots, v_{n+t}$. Note that the key and value vectors at positions 1 to $n + t - 1$ are cached at previous iterations, only the new key and value vector k_{n+t} and v_{n+t} are computed at this iteration. This phase completes either when the sequence reaches a maximum length (specified by users or limited by LLMs) or when an end-of-sequence (*<eos>*) token is emitted. The computation at different iterations cannot be parallelized due to the data dependency and often uses matrix-vector multiplication, which is less efficient. As a result, this phase severely underutilizes GPU computation and becomes memory-bound, being responsible for most portion of the latency of a single request.

5.2.3 Batching Techniques for LLMs

The compute utilization in serving LLMs can be improved by batching multiple requests. Because the requests share the same model weights, the overhead of moving weights is amortized across the requests in a batch, and can be overwhelmed by the computational

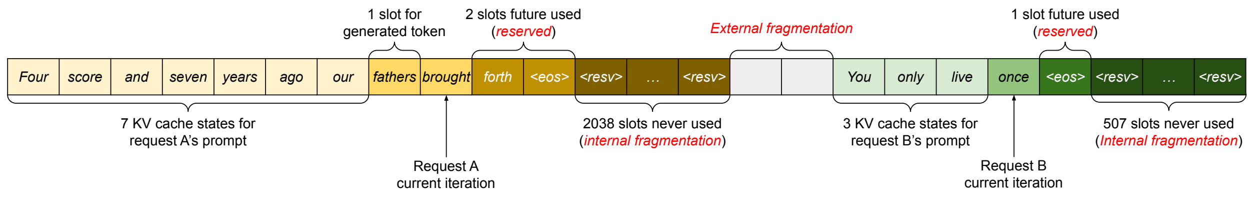

1 slot for generated token
2 slots future used (reserved)
External fragmentation
1 slot future used (reserved)
Four
score
and
seven
years
ago
our
fathers
brought
forth
<eos>
<resv>
…
<resv>
You
only
live
once
<eos>
<resv>
…
<resv>
7 KV cache states for request A's prompt
Request A current iteration
2038 slots never used (internal fragmentation)
3 KV cache states for request B's prompt
Request B current iteration
507 slots never used (Internal fragmentation)

Figure 5.3: KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

overhead when the batch size is sufficiently large. However, batching the requests to an LLM service is non-trivial for two reasons. First, the requests may arrive at different times. A naive batching strategy would either make earlier requests wait for later ones or delay the incoming requests until earlier ones finish, leading to significant queueing delays. Second, the requests may have vastly different input and output lengths (Fig. 5.11). A straightforward batching technique would pad the inputs and outputs of the requests to equalize their lengths, wasting GPU computation and memory.

To address this problem, fine-grained batching mechanisms, such as cellular batching [44] and iteration-level scheduling [153], have been proposed. Unlike traditional methods that work at the request level, these techniques operate at the iteration level. After each iteration, completed requests are removed from the batch, and new ones are added. Therefore, a new request can be processed after waiting for a single iteration, not waiting for the entire batch to complete. Moreover, with special GPU kernels, these techniques eliminate the need to pad the inputs and outputs. By reducing the queueing delay and the inefficiencies from padding, the fine-grained batching mechanisms significantly increase the throughput of LLM serving.

5.3 Memory Challenges in LLM Serving

Although fine-grained batching reduces the waste of computing and enables requests to be batched in a more flexible way, the number of requests that can be batched together is still constrained by GPU memory capacity, particularly the space allocated to store the KV cache. In other words, the serving system's throughput is *memory-bound*. Overcoming this memory-bound requires addressing the following challenges in the memory management:

Large KV cache. The KV Cache size grows quickly with the number of requests. As an example, for the 13B parameter OPT model [159], the KV cache of a single token demands 800 KB of space, calculated as 2 (key and value vectors) $\times$ 5120 (hidden state size) $\times$ 40 (number of layers) $\times$ 2 (bytes per FP16). Since OPT can generate sequences up to 2048 tokens, the memory required to store the KV cache of one request can be as much as 1.6 GB. Concurrent GPUs have memory capacities in the tens of GBs. Even if all available memory was allocated to KV cache, only a few tens of requests could be accommodated. Moreover, inefficient memory management can further decrease the batch size, as shown in Fig. 5.2. Additionally, given the current trends, the GPU's computation speed grows faster than the memory capacity [45]. For example, from NVIDIA A100 to H100, The FLOPS increases by more than 2x, but the GPU memory stays at 80GB maximum. Therefore, we believe the memory will become an increasingly significant bottleneck.

Complex decoding algorithms. LLM services offer a range of decoding algorithms for users to select from, each with varying implications for memory management complexity. For example, when users request multiple random samples from a single input prompt, a typical use case in program suggestion [46], the KV cache of the prompt part, which accounts for 12% of the total KV cache memory in our experiment (§5.6.3), can be shared to minimize memory usage. On the other hand, the KV cache during the autoregressive generation phase should remain unshared due to the different sample results and their dependence on context and position. The extent of KV cache sharing depends on the specific decoding algorithm employed. In more sophisticated algorithms like beam search [130], different request beams can share larger portions (up to 55% memory saving, see §5.6.3) of their KV cache, and the sharing pattern evolves as the decoding process advances.

Scheduling for unknown input & output lengths. The requests to an LLM service exhibit variability in their input and output lengths. This requires the memory management system to accommodate a wide range of prompt lengths. In addition, as the output length of a request grows at decoding, the memory required for its KV cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing prompts. The system needs to make scheduling decisions, such as deleting or swapping out the KV cache of some requests from GPU memory.

5.3.1 Memory Management in Existing Systems

Since most operators in current deep learning frameworks [104, 98] require tensors to be stored in contiguous memory, previous LLM serving systems [153, 96] also store the KV cache of one request as a contiguous tensor across the different positions. Due to the unpredictable output lengths from the LLM, they statically allocate a chunk of memory for a request based on the request's maximum possible sequence length, irrespective of the actual input or eventual output length of the request.

Fig. 5.3 illustrates two requests: request A with 2048 maximum possible sequence length and request B with a maximum of 512. The chunk pre-allocation scheme in existing systems has three primary sources of memory wastes: *reserved* slots for future tokens, *internal fragmentation* due to over-provisioning for potential maximum sequence lengths, and *external*

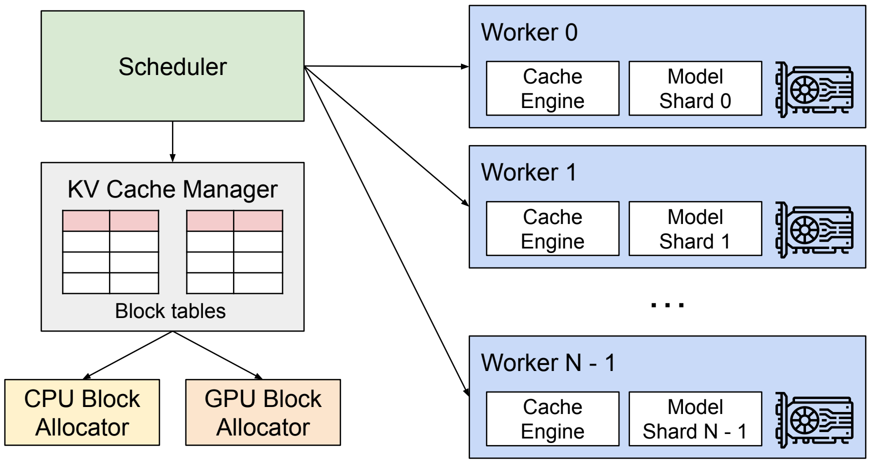


Figure 5.4: vLLM system overview.

fragmentation from the memory allocator like the buddy allocator. The external fragmentation will never be used for generated tokens, which is known before serving a request. Internal fragmentation also remains unused, but this is only realized after a request has finished sampling. They are both pure memory waste. Although the reserved memory is eventually used, reserving this space for the entire request's duration, especially when the reserved space is large, occupies the space that could otherwise be used to process other requests. We visualize the average percentage of memory wastes in our experiments in Fig. 5.2, revealing that the actual effective memory in previous systems can be as low as 20.4%.

Although compaction [141] has been proposed as a potential solution to fragmentation, performing compaction in a performance-sensitive LLM serving system is impractical due to the massive KV cache. Even with compaction, the pre-allocated chunk space for each request prevents memory sharing specific to decoding algorithms in existing memory management systems.

5.4 Method

In this work, we develop a new attention algorithm, *PagedAttention*, and build an LLM serving engine, *vLLM*, to tackle the challenges outlined in §5.3. The architecture of vLLM is shown in Fig. 5.4. vLLM adopts a centralized scheduler to coordinate the execution of distributed GPU workers. The *KV cache manager* effectively manages the KV cache in a paged fashion, enabled by PagedAttention. Specifically, the KV cache manager manages the physical KV cache memory on the GPU workers through the instructions sent by the centralized scheduler.

Next, We describe the PagedAttention algorithm in §5.4.1. With that, we show the design of the KV cache manager in §5.4.2 and how it facilitates PagedAttention in §5.4.3, respectively. Then, we show how this design facilitates effective memory management for various decoding methods (§5.4.4) and handles the variable length input and output sequences (§5.4.5). Finally, we show how the system design of vLLM works in a distributed setting (§5.4.6).

5.4.1 PagedAttention

To address the memory challenges in §5.3, we introduce *PagedAttention*, an attention algorithm inspired by the classic idea of *paging* [68] in operating systems. Unlike the traditional attention algorithms, PagedAttention allows storing continuous keys and values in non-contiguous memory space. Specifically, PagedAttention partitions the KV cache of each sequence into *KV blocks*. Each block contains the key and value vectors for a fixed number of tokens,[1] which we denote as *KV block size* (B). Denote the key block $K_j = (k_{(j-1)B+1}, \ldots, k_{jB})$ and value block $V_j = (v_{(j-1)B+1}, \ldots, v_{jB})$. The attention computation in Eq. 5.3 can be transformed into the following block-wise computation:

$$A_{ij} = \frac{\exp(q_i^\top K_j/\sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t/\sqrt{d}) \cdot \mathbf{1}}, \ o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \tag{5.4}$$

where $A_{ij} = (a_{i,(j-1)B+1}, \ldots, a_{i,jB})$ is the row vector of attention score on j-th KV block.

During the attention computation, the PagedAttention kernel identifies and fetches different KV blocks separately. We show an example of PagedAttention in Fig. 5.5: The key and value vectors are spread across three blocks, and the three blocks are not contiguous on the physical memory. At each time, the kernel multiplies the query vector q_i of the query token ("*forth*") and the key vectors K_j in a block (e.g., key vectors of "*Four score and seven*" for block 0) to compute the attention score A_{ij}, and later multiplies A_{ij} with the value vectors V_j in a block to derive the final attention output o_i.

In summary, the PagedAttention algorithm allows the KV blocks to be stored in non-contiguous physical memory, which enables more flexible paged memory management in vLLM.

5.4.2 KV Cache Manager

The key idea behind vLLM's memory manager is analogous to the *virtual memory* [68] in operating systems. OS partitions memory into fixed-sized *pages* and maps user programs' logical pages to physical pages. Contiguous logical pages can correspond to non-contiguous

[1]In Transformer, each token has a set of key and value vectors across layers and attention heads within a layer. All the key and value vectors can be managed together within a single KV block, or the key and value vectors at different heads and layers can each have a separate block and be managed in separate block tables. The two designs have no performance difference and we choose the second one for easy implementation.

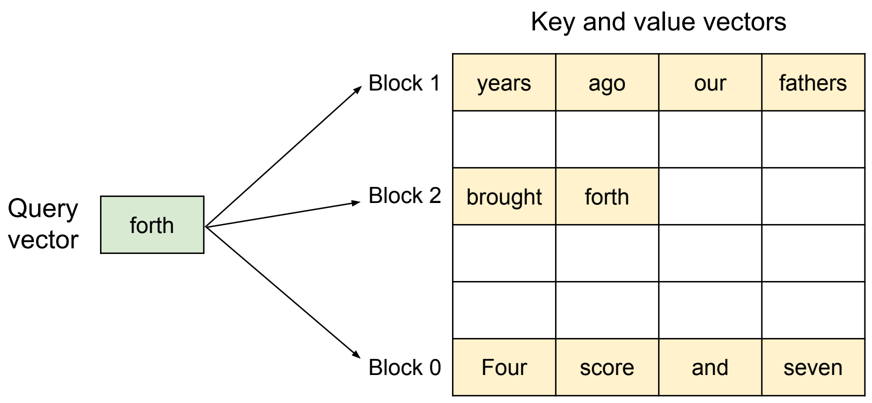


Figure 5.5: Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

physical memory pages, allowing user programs to access memory as though it were contiguous. Moreover, physical memory space needs not to be fully reserved in advance, enabling the OS to dynamically allocate physical pages as needed. vLLM uses the ideas behind virtual memory to manage the KV cache in an LLM service. Enabled by PagedAttention, we organize the KV cache as fixed-size KV blocks, like pages in virtual memory.

A request's KV cache is represented as a series of *logical KV blocks*, filled from left to right as new tokens and their KV cache are generated. The last KV block's unfilled positions are reserved for future generations. On GPU workers, a *block engine* allocates a contiguous chunk of GPU DRAM and divides it into *physical KV blocks* (this is also done on CPU RAM for swapping; see §5.4.5). The *KV block manager* also maintains *block tables*—the mapping between logical and physical KV blocks of each request. Each block table entry records the corresponding physical blocks of a logical block and the number of filled positions. Separating logical and physical KV blocks allows vLLM to dynamically grow the KV cache memory without reserving it for all positions in advance, which eliminates most memory waste in existing systems, as in Fig. 5.2.

5.4.3 Decoding with PagedAttention and vLLM

Next, we walk through an example, as in Fig. 5.6, to demonstrate how vLLM executes PagedAttention and manages the memory during the decoding process of a single input sequence: ① As in OS's virtual memory, vLLM does not require reserving the memory for the maximum possible generated sequence length initially. Instead, it reserves only the necessary KV blocks to accommodate the KV cache generated during prompt computation. In this case, The prompt has 7 tokens, so vLLM maps the first 2 logical KV blocks (0 and 1) to 2 physical KV blocks (7 and 1, respectively). In the prefill step, vLLM generates the KV cache of the prompts and the first output token with a conventional self-attention

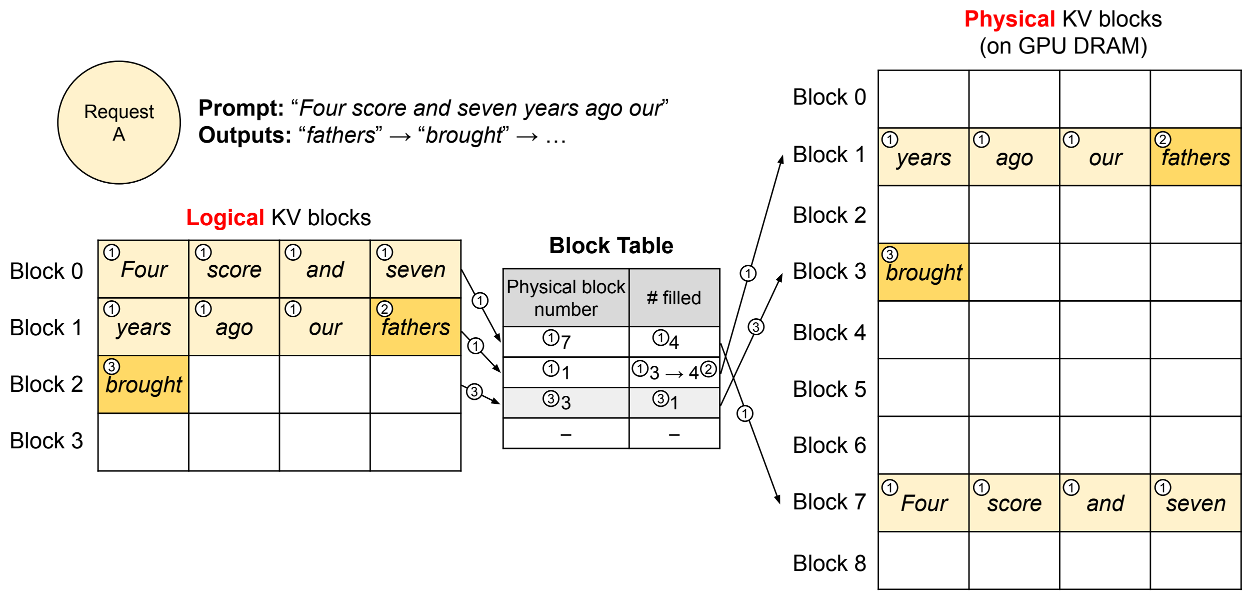


Figure 5.6: Block table translation in vLLM.

algorithm (e.g., [30]). vLLM then stores the KV cache of the first 4 tokens in logical block 0 and the following 3 tokens in logical block 1. The remaining slot is reserved for the subsequent autoregressive generation phase. ② In the first autoregressive decoding step, vLLM generates the new token with the PagedAttention algorithm on physical blocks 7 and 1. Since one slot remains available in the last logical block, the newly generated KV cache is stored there, and the block table's #filled record is updated. ③ At the second decoding step, as the last logical block is full, vLLM stores the newly generated KV cache in a new logical block; vLLM allocates a new physical block (physical block 3) for it and stores this mapping in the block table.

Globally, for each decoding iteration, vLLM first selects a set of candidate sequences for batching (more in §5.4.5), and allocates the physical blocks for the newly required logical blocks. Then, vLLM concatenates all the input tokens of the current iteration (i.e., all tokens for prompt phase requests and the latest tokens for generation phase requests) as one sequence and feeds it into the LLM. During LLM's computation, vLLM uses the PagedAttention kernel to access the previous KV cache stored in the form of logical KV blocks and saves the newly generated KV cache into the physical KV blocks. Storing multiple tokens within a KV block (block size > 1) enables the PagedAttention kernel to process the KV cache across more positions in parallel, thus increasing the hardware utilization and reducing latency. However, a larger block size also increases memory fragmentation. We study the effect of block size in §5.7.2.

Again, vLLM dynamically assigns new physical blocks to logical blocks as more tokens and

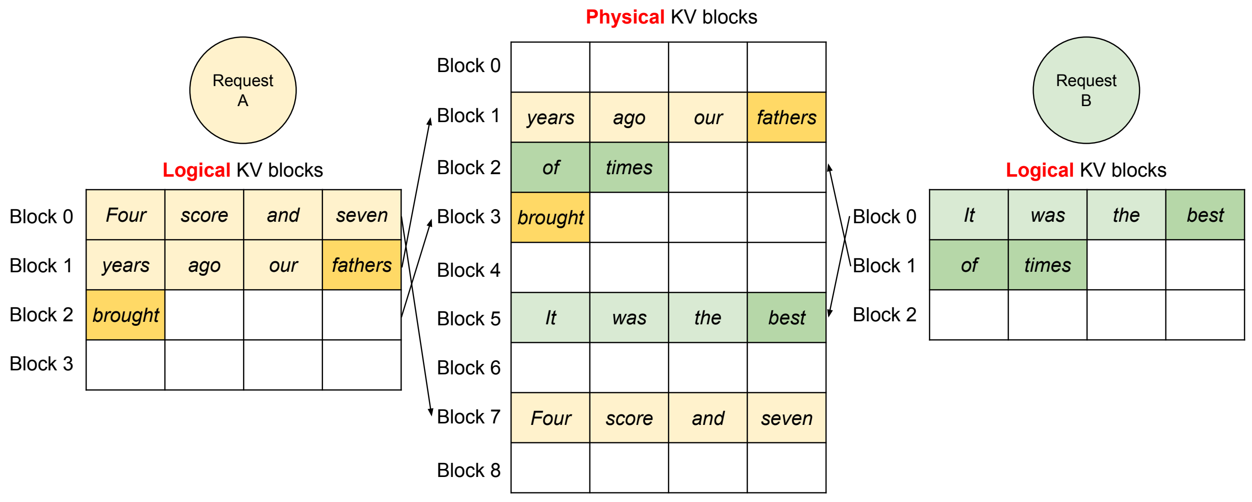


Figure 5.7: Storing the KV cache of two requests at the same time in vLLM.

their KV cache are generated. As all the blocks are filled from left to right and a new physical block is only allocated when all previous blocks are full, vLLM limits all the memory wastes for a request within one block, so it can effectively utilize all the memory, as shown in Fig. 5.2. This allows more requests to fit into memory for batching—hence improving the throughput. Once a request finishes its generation, its KV blocks can be freed to store the KV cache of other requests. In Fig. 5.7, we show an example of vLLM managing the memory for two sequences. The logical blocks of the two sequences are mapped to different physical blocks within the space reserved by the block engine in GPU workers. The neighboring logical blocks of both sequences do not need to be contiguous in physical GPU memory and the space of physical blocks can be effectively utilized by both sequences.

5.4.4 Application to Other Decoding Scenarios

§5.4.3 shows how PagedAttention and vLLM handle basic decoding algorithms, such as greedy decoding and sampling, that take one user prompt as input and generate a single output sequence. In many successful LLM applications [46, 99], an LLM service must offer more complex decoding scenarios that exhibit complex accessing patterns and more opportunities for memory sharing. We show the general applicability of vLLM on them in this section.

Parallel sampling. In LLM-based program assistants [21, 46], an LLM generates multiple sampled outputs for a single input prompt; users can choose a favorite output from various candidates. So far we have implicitly assumed that a request generates a single sequence. In the remainder of this chapter, we assume the more general case in which a request generates multiple sequences. In parallel sampling, one request includes multiple samples sharing the same input prompt, allowing the KV cache of the prompt to be shared as well. Via its PagedAttention and paged memory management, vLLM can realize this sharing easily and save memory.

Fig. 5.8 shows an example of parallel decoding for two outputs. Since both outputs share the same prompt, we only reserve space for one copy of the prompt's state at the prompt phase; the logical blocks for the prompts of both sequences are mapped to the same physical blocks: the logical block 0 and 1 of both sequences are mapped to physical blocks 7 and 1, respectively. Since a single physical block can be mapped to multiple logical blocks, we introduce a *reference count* for each physical block. In this case, the reference counts for physical blocks 7 and 1 are both 2. At the generation phase, the two outputs sample different output tokens and need separate storage for KV cache. vLLM implements a *copy-on-write* mechanism at the block granularity for the physical blocks that need modification by multiple sequences, similar to the copy-on-write technique in OS virtual memory (e.g., when forking a process). Specifically, in Fig. 5.8, when sample A1 needs to write to its last logical block (logical block 1), vLLM recognizes that the reference count of the corresponding physical block (physical block 1) is greater than 1; it allocates a new physical block (physical block 3), instructs the block engine to copy the information from physical block 1, and decreases the reference count to 1. Next, when sample A2 writes to physical block 1, the reference count is already reduced to 1; thus A2 directly writes its newly generated KV cache to physical block 1.

In summary, vLLM enables the sharing of most of the space used to store the prompts' KV cache across multiple output samples, with the exception of the final logical block, which is managed by a copy-on-write mechanism. By sharing physical blocks across multiple samples, memory usage can be greatly reduced, especially for *long input prompts*.

Beam search. In LLM tasks like machine translation [149], the users expect the top-k most appropriate translations output by the LLM. Beam search [130] is widely used to decode the most probable output sequence from an LLM, as it mitigates the computational complexity of fully traversing the sample space. The algorithm relies on the *beam width* parameter k, which determines the number of top candidates retained at every step. During decoding, beam search expands each candidate sequence in the beam by considering all possible tokens, computes their respective probabilities using the LLM, and retains the top-k most probable sequences out of $k \cdot |V|$ candidates, where $|V|$ is the vocabulary size.

Unlike parallel decoding, beam search facilities sharing not only the initial prompt blocks but also other blocks across different candidates, and the sharing patterns dynamically change as the decoding process advances, similar to the process tree in the OS created by compound forks. Fig. 5.9 shows how vLLM manages the KV blocks for a beam search example with $k = 4$. Prior to the iteration illustrated as the dotted line, each candidate sequence has used 4 full logical blocks. All beam candidates share the first block 0 (i.e., prompt). Candidate

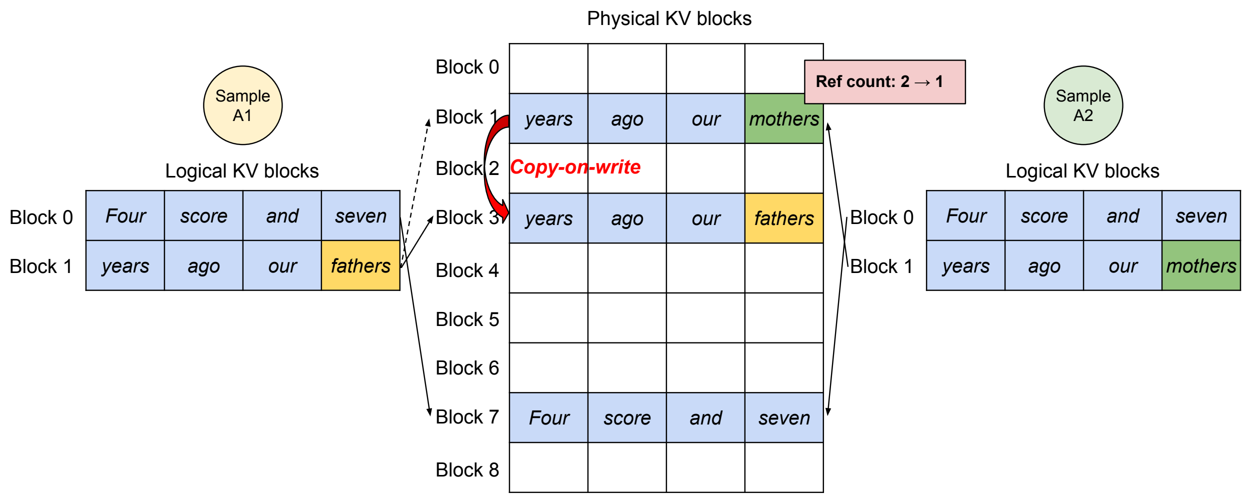


Figure 5.8: Parallel sampling example.

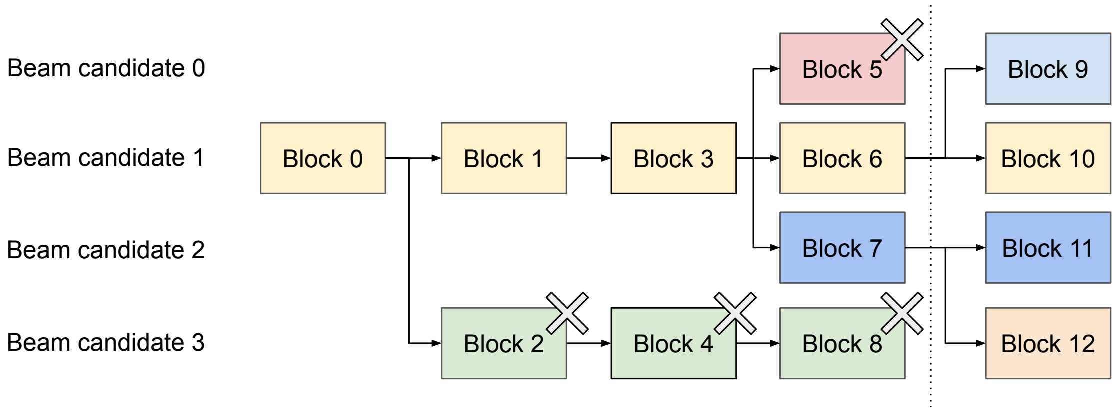


Figure 5.9: Beam search example.

3 digresses from others from the second block. Candidates 0-2 share the first 3 blocks and diverge at the fourth block. At subsequent iterations, the top-4 probable candidates all originate from candidates 1 and 2. As the original candidates 0 and 3 are no longer among the top candidates, their logical blocks are freed, and the reference counts of corresponding physical blocks are reduced. vLLM frees all physical blocks whose reference counts reach 0 (blocks 2, 4, 5, 8). Then, vLLM allocates new physical blocks (blocks 9-12) to store the new KV cache from the new candidates. Now, all candidates share blocks 0, 1, 3; candidates 0 and 1 share block 6, and candidates 2 and 3 further share block 7.

Previous LLM serving systems require frequent memory copies of the KV cache across the beam candidates. For example, in the case shown in Fig. 5.9, after the dotted line, candidate 3 would need to copy a large portion of candidate 2's KV cache to continue generation. This frequent memory copy overhead is significantly reduced by vLLM's physical block sharing. In vLLM, most blocks of different beam candidates can be shared. The copy-on-write mechanism is applied only when the newly generated tokens are within an old shared block, as in parallel decoding. This involves only copying one block of data.

Shared prefix. Commonly, the LLM user provides a (long) description of the task including instructions and example inputs and outputs, also known as *system prompt* [100]. The description is concatenated with the actual task input to form the prompt of the request. The LLM generates outputs based on the full prompt. Fig. 5.10 shows an example. Moreover, the shared prefix can be further tuned, via prompt engineering, to improve the accuracy of the downstream tasks [80, 77].

For this type of application, many user prompts share a prefix, thus the LLM service provider can store the KV cache of the prefix in advance to reduce the redundant computation spent on the prefix. In vLLM, this can be conveniently achieved by reserving a set of physical blocks for a set of predefined shared prefixes by the LLM service provider, as how OS handles shared library across processes. A user input prompt with the shared prefix can simply map its logical blocks to the cached physical blocks (with the last block marked copy-on-write). The prompt phase computation only needs to execute on the user's task input.

Mixed decoding methods. The decoding methods discussed earlier exhibit diverse memory sharing and accessing patterns. Nonetheless, vLLM facilitates the simultaneous processing of requests with different decoding preferences, which existing systems *cannot* efficiently do. This is because vLLM conceals the complex memory sharing between different sequences via a common mapping layer that translates logical blocks to physical blocks. The LLM and its execution kernel only see a list of physical block IDs for each sequence and do not need to handle sharing patterns across sequences. Compared to existing systems, this approach broadens the batching opportunities for requests with different sampling requirements, ultimately increasing the system's overall throughput.

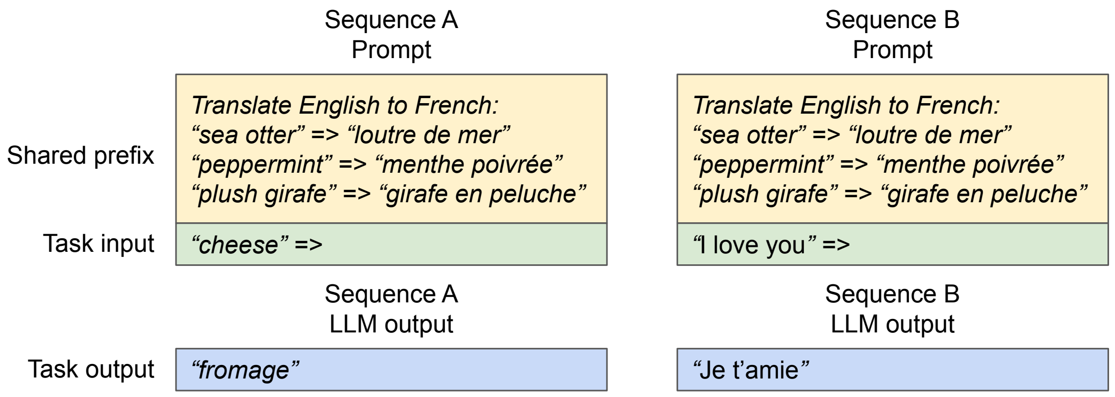


Figure 5.10: Shared prompt example for machine translation. The examples are adopted from [18].

5.4.5 Scheduling and Preemption

When the request traffic surpasses the system's capacity, vLLM must prioritize a subset of requests. In vLLM, we adopt the first-come-first-serve (FCFS) scheduling policy for all requests, ensuring fairness and preventing starvation. When vLLM needs to preempt requests, it ensures that the earliest arrived requests are served first and the latest requests are preempted first.

LLM services face a unique challenge: the input prompts for an LLM can vary significantly in length, and the resulting output lengths are not known a priori, contingent on both the input prompt and the model. As the number of requests and their outputs grow, vLLM can run out of the GPU's physical blocks to store the newly generated KV cache. There are two classic questions that vLLM needs to answer in this context: (1) Which blocks should it evict? (2) How to recover evicted blocks if needed again? Typically, eviction policies use heuristics to predict which block will be accessed furthest in the future and evict that block. Since in our case we know that all blocks of a sequence are accessed together, we implement an all-or-nothing eviction policy, i.e., either evict all or none of the blocks of a sequence. Furthermore, multiple sequences within one request (e.g., beam candidates in one beam search request) are gang-scheduled as a *sequence group*. The sequences within one sequence group are always preempted or rescheduled together due to potential memory sharing across those sequences. To answer the second question of how to recover an evicted block, we consider two techniques:

Swapping. This is the classic technique used by most virtual memory implementations which copy the evicted pages to a swap space on the disk. In our case, we copy evicted blocks to the CPU memory. As shown in Fig. 5.4, besides the GPU block allocator, vLLM includes a CPU block allocator to manage the physical blocks swapped to CPU RAM. When vLLM exhausts free physical blocks for new tokens, it selects a set of sequences to evict and transfer their KV cache to the CPU. Once it preempts a sequence and evicts its blocks, vLLM stops accepting new requests until all preempted sequences are completed. Once a request completes, its blocks are freed from memory, and the blocks of a preempted sequence are brought back in to continue the processing of that sequence. Note that with this design, the number of blocks swapped to the CPU RAM never exceeds the number of total physical blocks in the GPU RAM, so the swap space on the CPU RAM is bounded by the GPU memory allocated for the KV cache.

Recomputation. In this case, we simply recompute the KV cache when the preempted sequences are rescheduled. Note that recomputation latency can be significantly lower than the original latency, as the tokens generated at decoding can be concatenated with the original user prompt as a new prompt—their KV cache at all positions can be generated in one prompt phase iteration.

The performances of swapping and recomputation depend on the bandwidth between CPU RAM and GPU memory and the computation power of the GPU. We examine the speeds of swapping and recomputation in §5.7.3.

5.4.6 Distributed Execution

Many LLMs have parameter sizes exceeding the capacity of a single GPU [18, 25]. Therefore, it is necessary to partition them across distributed GPUs and execute them in a model parallel fashion [165, 82]. This calls for a memory manager capable of handling distributed memory. vLLM is effective in distributed settings by supporting the widely used Megatron-LM style tensor model parallelism strategy on Transformers [125]. This strategy adheres to an SPMD (Single Program Multiple Data) execution schedule, wherein the linear layers are partitioned to perform block-wise matrix multiplication, and the the GPUs constantly synchronize intermediate results via an all-reduce operation. Specifically, the attention operator is split on the attention head dimension, each SPMD process takes care of a subset of attention heads in multi-head attention.

We observe that even with model parallel execution, each model shard still processes the same set of input tokens, thus requiring the KV Cache for the same positions. Therefore, vLLM features a single KV cache manager within the centralized scheduler, as in Fig. 5.4. Different GPU workers share the manager, as well as the mapping from logical blocks to physical blocks. This common mapping allows GPU workers to execute the model with the physical blocks provided by the scheduler for each input request. Although each GPU worker has the same physical block IDs, a worker only stores a portion of the KV cache for its corresponding attention heads.

In each step, the scheduler first prepares the message with input token IDs for each request in the batch, as well as the block table for each request. Next, the scheduler broadcasts

Table 5.1: Model sizes and server configurations.

Model size	**13B**	**66B**	**175B**
GPUs	A100	4×A100	8×A100-80GB
Total GPU memory	40 GB	160 GB	640 GB
Parameter size	26 GB	132 GB	346 GB
Memory for KV cache	12 GB	21 GB	264 GB
Max. # KV cache slots	15.7K	9.7K	60.1K

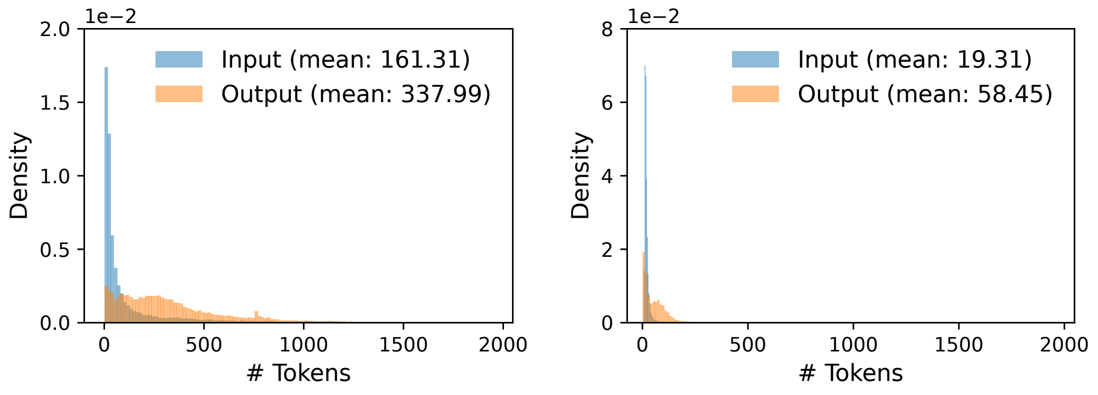
this control message to the GPU workers. Then, the GPU workers start to execute the model with the input token IDs. In the attention layers, the GPU workers read the KV cache according to the block table in the control message. During execution, the GPU workers synchronize the intermediate results with the all-reduce communication primitive without the coordination of the scheduler, as in [125]. In the end, the GPU workers send the sampled tokens of this iteration back to the scheduler. In summary, GPU workers do not need to synchronize on memory management as they only need to receive all the memory management information at the beginning of each decoding iteration along with the step inputs.

5.5 Implementation

vLLM is an end-to-end serving system with a FastAPI [40] frontend and a GPU-based inference engine. The frontend extends the OpenAI API [99] interface, allowing users to customize sampling parameters for each request, such as the maximum sequence length and the beam width k. The vLLM engine is written in 8.5K lines of Python and 2K lines of C++/CUDA code. We develop control-related components including the scheduler and the block manager in Python while developing custom CUDA kernels for key operations such as PagedAttention. For the model executor, we implement popular LLMs such as GPT [18], OPT [159], and LLaMA [137] using PyTorch [104] and Transformers [147]. We use NCCL [95] for tensor communication across the distributed GPU workers.

5.5.1 Kernel-level Optimization

Since PagedAttention introduces memory access patterns that are not efficiently supported by existing systems, we develop several GPU kernels for optimizing it. (1) *Fused reshape and block write.* In every Transformer layer, the new KV cache are split into blocks, reshaped to a memory layout optimized for block read, then saved at positions specified by the block table. To minimize kernel launch overheads, we fuse them into a single kernel. (2) *Fusing block read and attention.* We adapt the attention kernel in FasterTransformer [96] to read KV cache according to the block table and perform attention operations on the fly. To ensure



(a) ShareGPT (b) Alpaca

Figure 5.11: Input and output length distributions of the (a) ShareGPT and (b) Alpaca datasets.

coalesced memory access, we assign a GPU warp to read each block. Moreover, we add support for variable sequence lengths within a request batch. (3) *Fused block copy.* Block copy operations, issued by the copy-on-write mechanism, may operate on discontinuous blocks. This can lead to numerous invocations of small data movements if we use the `cudaMemcpyAsync` API. To mitigate the overhead, we implement a kernel that batches the copy operations for different blocks into a single kernel launch.

5.5.2 Supporting Various Decoding Algorithms

vLLM implements various decoding algorithms using three key methods: `fork`, `append`, and `free`. The `fork` method creates a new sequence from an existing one. The `append` method appends a new token to the sequence. Finally, the `free` method deletes the sequence. For instance, in parallel sampling, vLLM creates multiple output sequences from the single input sequence using the `fork` method. It then adds new tokens to these sequences in every iteration with `append`, and deletes sequences that meet a stopping condition using `free`. The same strategy is also applied in beam search and prefix sharing by vLLM. We believe future decoding algorithms can also be supported by combining these methods.

5.6 Evaluation

In this section, we evaluate the performance of vLLM under a variety of workloads.

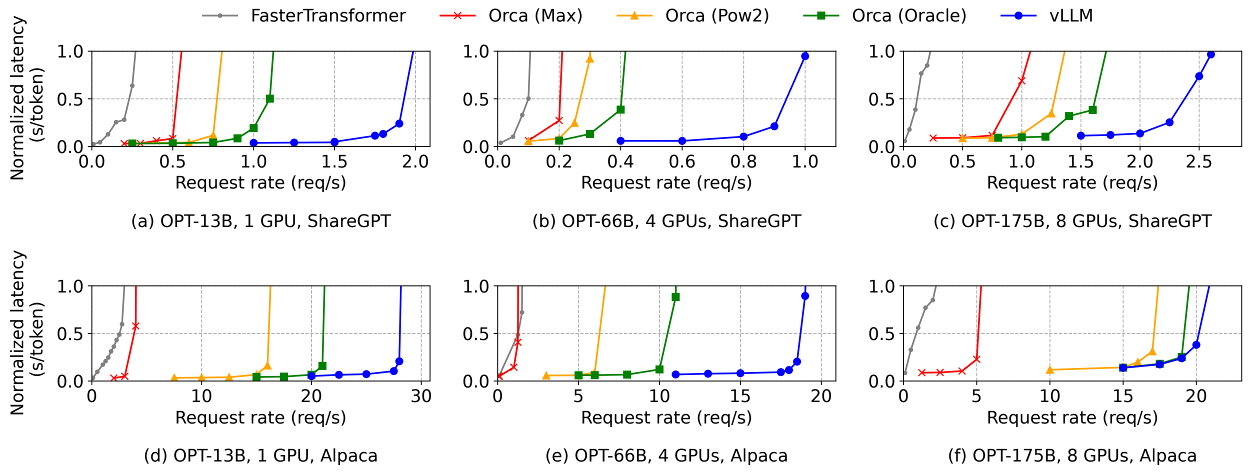


Figure 5.12: Single sequence generation with OPT models on the ShareGPT and Alpaca dataset

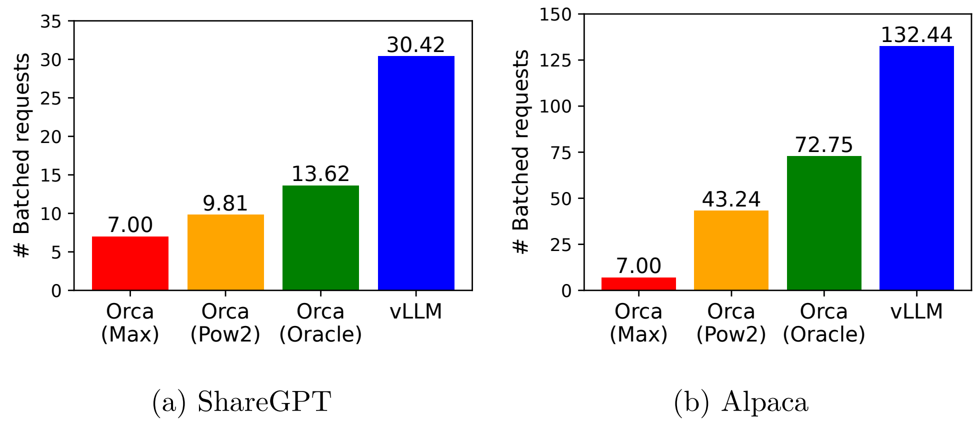


Figure 5.13: Average number of batched requests when serving OPT-13B for the ShareGPT (2 reqs/s) and Alpaca (30 reqs/s) traces.

5.6.1 Experimental Setup

Model and server configurations. We use OPT [159] models with 13B, 66B, and 175B parameters and LLaMA [137] with 13B parameters for our evaluation. 13B and 66B are popular sizes for LLMs as shown in an LLM leaderboard [103], while 175B is the size of the famous GPT-3 [18] model. For all of our experiments, we use A2 instances with NVIDIA A100 GPUs on Google Cloud Platform. The detailed model sizes and server configurations are shown in Table 5.1.

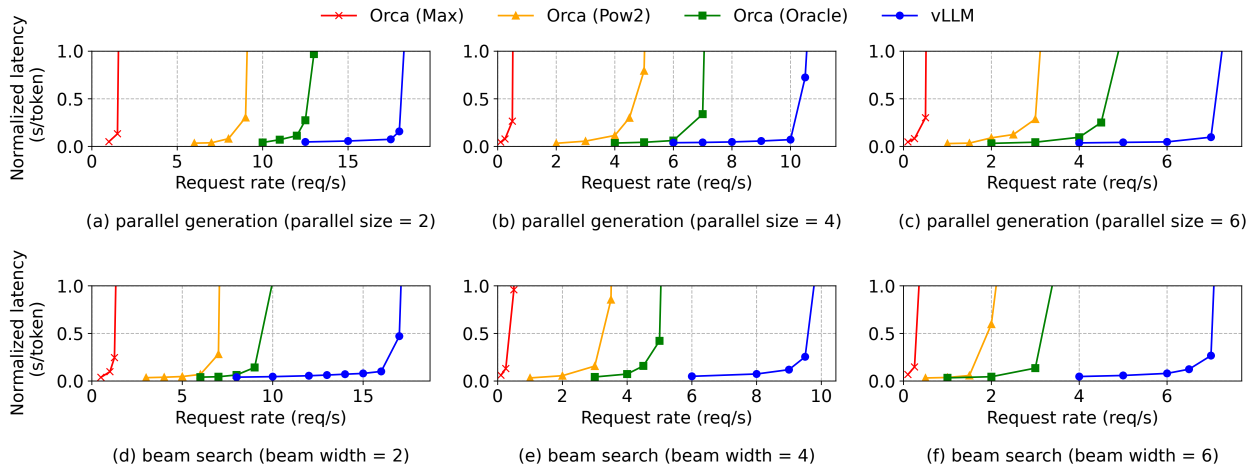


Figure 5.14: Parallel generation and beam search with OPT-13B on the Alpaca dataset.

Workloads. We synthesize workloads based on ShareGPT [135] and Alpaca [131] datasets, which contain input and output texts of real LLM services. The ShareGPT dataset is a collection of user-shared conversations with ChatGPT [101]. The Alpaca dataset is an instruction dataset generated by GPT-3.5 with self-instruct [145]. We tokenize the datasets and use their input and output lengths to synthesize client requests. As shown in Fig. 5.11, the ShareGPT dataset has 8.4× longer input prompts and 5.8× longer outputs on average than the Alpaca dataset, with higher variance. Since these datasets do not include timestamps, we generate request arrival times using Poisson distribution with different request rates.

Baseline 1: FasterTransformer. FasterTransformer [96] is a distributed inference engine highly optimized for latency. As FasterTransformer does not have its own scheduler, we implement a custom scheduler with a dynamic batching mechanism similar to the existing serving systems such as Triton [97]. Specifically, we set a maximum batch size B as large as possible for each experiment, according to the GPU memory capacity. The scheduler takes up to B number of earliest arrived requests and sends the batch to FasterTransformer for processing.

Baseline 2: Orca. Orca [153] is a state-of-the-art LLM serving system optimized for throughput. Since Orca is not publicly available for use, we implement our own version of Orca. We assume Orca uses the buddy allocation algorithm to determine the memory address to store KV cache. We implement three versions of Orca based on how much it over-reserves the space for request outputs:

- **Orca (Oracle).** We assume the system has the knowledge of the lengths of the outputs that will be actually generated for the requests. This shows the upper-bound performance of Orca, which is infeasible to achieve in practice.

- **Orca (Pow2).** We assume the system over-reserves the space for outputs by at most 2×. For example, if the true output length is 25, it reserves 32 positions for outputs.
- **Orca (Max).** We assume the system always reserves the space up to the maximum sequence length of the model, i.e., 2048 tokens.

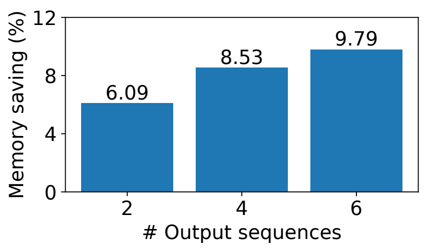
Key metrics. We focus on serving throughput. Specifically, using the workloads with different request rates, we measure *normalized latency* of the systems, the mean of every request's end-to-end latency divided by its output length, as in Orca [153]. A high-throughput serving system should retain low normalized latency against high request rates. For most experiments, we evaluate the systems with 1-hour traces. As an exception, we use 15-minute traces for the OPT-175B model due to the cost limit.

5.6.2 Basic Sampling

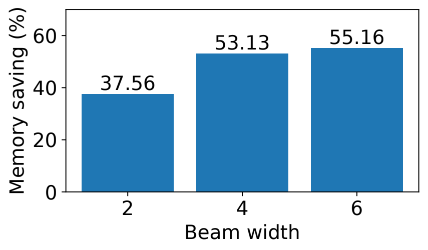
We evaluate the performance of vLLM with basic sampling (one sample per request) on three models and two datasets. The first row of Fig. 5.12 shows the results on the ShareGPT dataset. The curves illustrate that as the request rate increases, the latency initially increases at a gradual pace but then suddenly explodes. This can be attributed to the fact that when the request rate surpasses the capacity of the serving system, the queue length continues to grow infinitely and so does the latency of the requests.

On the ShareGPT dataset, vLLM can sustain 1.7×–2.7× higher request rates compared to Orca (Oracle) and 2.7×–8× compared to Orca (Max), while maintaining similar latencies. This is because vLLM's PagedAttention can efficiently manage the memory usage and thus enable batching more requests than Orca. For example, as shown in Fig. 5.13a, for OPT-13B vLLM processes 2.2× more requests at the same time than Orca (Oracle) and 4.3× more requests than Orca (Max). Compared to FasterTransformer, vLLM can sustain up to 22× higher request rates, as FasterTransformer does not utilize a fine-grained scheduling mechanism and inefficiently manages the memory like Orca (Max).

The second row of Fig. 5.12 and Fig. 5.13b shows the results on the Alpaca dataset, which follows a similar trend to the ShareGPT dataset. One exception is Fig. 5.12 (f), where vLLM's advantage over Orca (Oracle) and Orca (Pow2) is less pronounced. This is because the model and server configuration for OPT-175B (Table 5.1) allows for large GPU memory space available to store KV cache, while the Alpaca dataset has short sequences. In this setup, Orca (Oracle) and Orca (Pow2) can also batch a large number of requests despite the inefficiencies in their memory management. As a result, the performance of the systems becomes compute-bound rather than memory-bound.



(a) Parallel sampling



(b) Beam search

Figure 5.15: Average amount of memory saving from sharing KV blocks, when serving OPT-13B for the Alpaca trace.

5.6.3 Parallel Sampling and Beam Search

We evaluate the effectiveness of memory sharing in PagedAttention with two popular sampling methods: parallel sampling and beam search. In parallel sampling, all parallel sequences in a request can share the KV cache for the prompt. As shown in the first row of Fig. 5.14, with a larger number of sequences to sample, vLLM brings more improvement over the Orca baselines. Similarly, the second row of Fig. 5.14 shows the results for beam search with different beam widths. Since beam search allows for more sharing, vLLM demonstrates even greater performance benefits. The improvement of vLLM over Orca (Oracle) on OPT-13B and the Alpaca dataset goes from 1.3× in basic sampling to 2.3× in beam search with a width of 6.

Fig. 5.15 plots the amount of memory saving, computed by the number of blocks we saved by sharing divided by the number of total blocks without sharing. We show 6.1% - 9.8% memory saving on parallel sampling and 37.6% - 55.2% on beam search. In the same experiments with the ShareGPT dataset, we saw 16.2% - 30.5% memory saving on parallel sampling and 44.3% - 66.3% on beam search.

5.6.4 Shared prefix

We explore the effectiveness of vLLM for the case a prefix is shared among different input prompts, as illustrated in Fig. 5.10. For the model, we use LLaMA-13B [137], which is multilingual. For the workload, we use the WMT16 [16] English-to-German translation dataset and synthesize two prefixes that include an instruction and a few translation examples. The first prefix includes a single example (i.e., one-shot) while the other prefix includes 5 examples (i.e., few-shot). As shown in Fig. 5.16 (a), vLLM achieves 1.67× higher throughput than Orca (Oracle) when the one-shot prefix is shared. Furthermore, when more examples are shared (Fig. 5.16 (b)), vLLM achieves 3.58× higher throughput than Orca (Oracle).

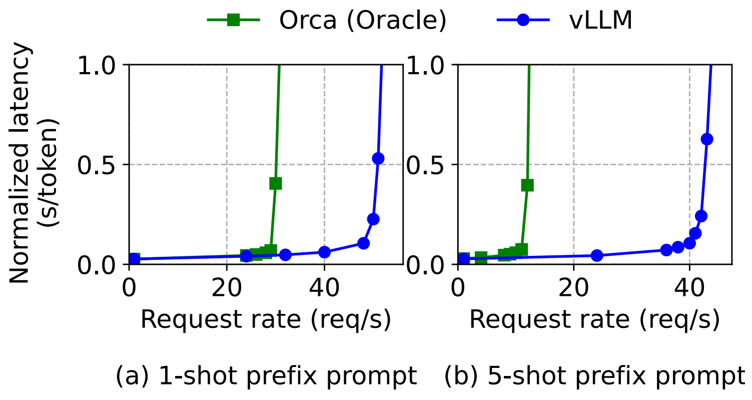


Figure 5.16: Translation workload where the input prompts share a common prefix. The prefix includes (a) 1 example with 80 tokens or (b) 5 examples with 341 tokens.

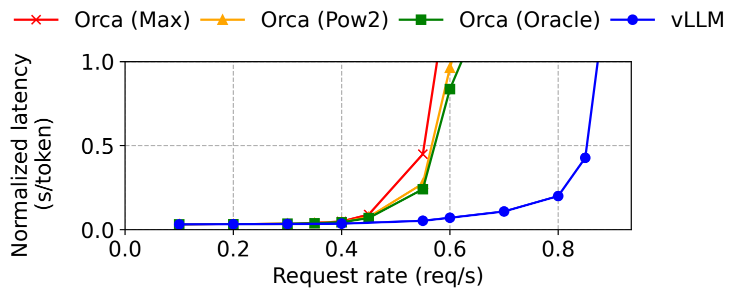
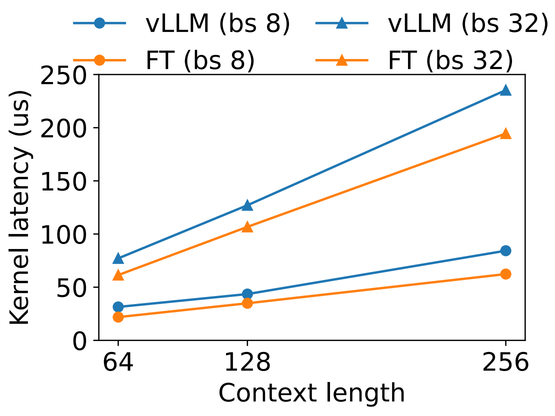


Figure 5.17: Performance on chatbot workload.

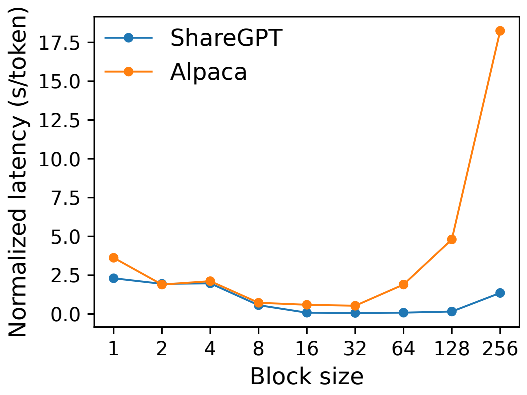
5.6.5 Chatbot

A chatbot [101, 47, 24] is one of the most important applications of LLMs. To implement a chatbot, we let the model generate a response by concatenating the chatting history and the last user query into a prompt. We synthesize the chatting history and user query using the ShareGPT dataset. Due to the limited context length of the OPT-13B model, we cut the prompt to the last 1024 tokens and let the model generate at most 1024 tokens. We do not store the KV cache between different conversation rounds as doing this would occupy the space for other requests between the conversation rounds.

Fig. 5.17 shows that vLLM can sustain 2× higher request rates compared to the three Orca baselines. Since the ShareGPT dataset contains many long conversations, the input prompts for most requests have 1024 tokens. Due to the buddy allocation algorithm, the Orca baselines reserve the space for 1024 tokens for the request outputs, regardless of how they predict the output lengths. For this reason, the three Orca baselines behave similarly.



(a) Latency of attention kernels.



(b) End-to-end latency with different block sizes.

Figure 5.18: Ablation experiments.

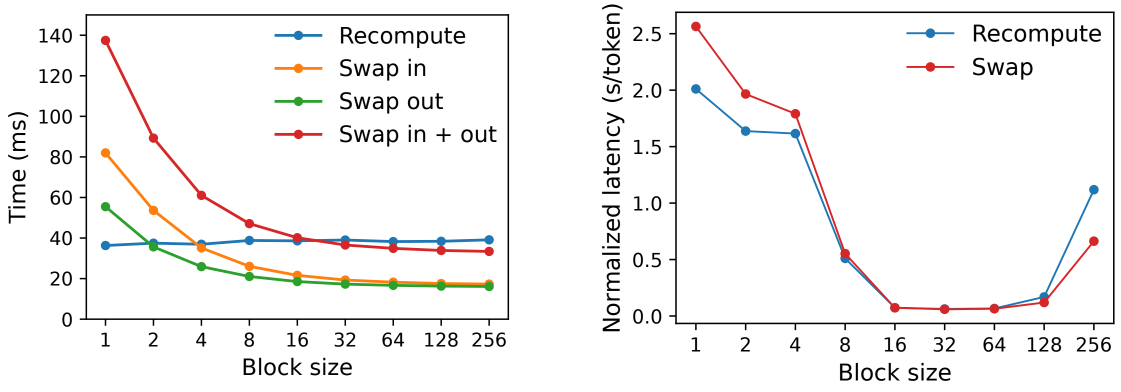
In contrast, vLLM can effectively handle the long prompts, as PagedAttention resolves the problem of memory fragmentation and reservation.

5.7 Ablation Studies

In this section, we study various aspects of vLLM and evaluate the design choices we make with ablation experiments.

5.7.1 Kernel Microbenchmark

The dynamic block mapping in PagedAttention affects the performance of the GPU operations involving the stored KV cache, i.e., block read/writes and attention. Compared to the existing systems, our GPU kernels (§5.5) involve extra overheads of accessing the block table, executing extra branches, and handling variable sequence lengths. As shown in Fig. 5.18a, this leads to 20–26% higher attention kernel latency, compared to the highly-optimized FasterTransformer implementation. We believe the overhead is small as it only affects the attention operator but not the other operators in the model, such as Linear. Despite the overhead, PagedAttention makes vLLM significantly outperform FasterTransformer in end-to-end performance (§5.6).



(a) Microbenchmark (b) End-to-end performance

Figure 5.19: (a) Overhead of recomputation and swapping for different block sizes. (b) Performance when serving OPT-13B with the ShareGPT traces at the same request rate.

5.7.2 Impact of Block Size

The choice of block size can have a substantial impact on the performance of vLLM. If the block size is too small, vLLM may not fully utilize the GPU's parallelism for reading and processing KV cache. If the block size is too large, internal fragmentation increases and the probability of sharing decreases.

In Fig. 5.18b, we evaluate the performance of vLLM with different block sizes, using the ShareGPT and Alpaca traces with basic sampling under fixed request rates. In the ShareGPT trace, block sizes from 16 to 128 lead to the best performance. In the Alpaca trace, while the block size 16 and 32 work well, larger block sizes significantly degrade the performance since the sequences become shorter than the block sizes. In practice, we find that the block size 16 is large enough to efficiently utilize the GPU and small enough to avoid significant internal fragmentation in most workloads. Accordingly, vLLM sets its default block size as 16.

5.7.3 Comparing Recomputation and Swapping

vLLM supports both recomputation and swapping as its recovery mechanisms. To understand the tradeoffs between the two methods, we evaluate their end-to-end performance and microbenchmark their overheads, as presented in Fig. 5.19. Our results reveal that swapping incurs excessive overhead with small block sizes. This is because small block sizes often result in numerous small data transfers between CPU and GPU, which limits the effective PCIe bandwidth. In contrast, the overhead of recomputation remains constant across different block sizes, as recomputation does not utilize the KV blocks. Thus, recomputation is more efficient when the block size is small, while swapping is more efficient when the block size is large, though recomputation overhead is never higher than 20% of swapping's latency. For medium block sizes from 16 to 64, the two methods exhibit comparable end-to-end performance.

5.8 Discussion

Applying the virtual memory and paging technique to other GPU workloads. The idea of virtual memory and paging is effective for managing the KV cache in LLM serving because the workload requires dynamic memory allocation (since the output length is not known a priori) and its performance is bound by the GPU memory capacity. However, this does not generally hold for every GPU workload. For example, in DNN training, the tensor shapes are typically static, and thus memory allocation can be optimized ahead of time. For another example, in serving DNNs that are not LLMs, an increase in memory efficiency may not result in any performance improvement since the performance is primarily compute-bound. In such scenarios, introducing the vLLM's techniques may rather degrade the performance due to the extra overhead of memory indirection and non-contiguous block memory. However, we would be excited to see vLLM's techniques being applied to other workloads with similar properties to LLM serving.

LLM-specific optimizations in applying virtual memory and paging. vLLM re-interprets and augments the idea of virtual memory and paging by leveraging the application-specific semantics. One example is vLLM's all-or-nothing swap-out policy, which exploits the fact that processing a request requires all of its corresponding token states to be stored in GPU memory. Another example is the recomputation method to recover the evicted blocks, which is not feasible in OS. Besides, vLLM mitigates the overhead of memory indirection in paging by fusing the GPU kernels for memory access operations with those for other operations such as attention.

5.9 Related Work

General model serving systems. Model serving has been an active area of research in recent years, with numerous systems proposed to tackle diverse aspects of deep learning model deployment. Clipper [28], TensorFlow Serving [98], Nexus [123], InferLine [27], and Clockwork [50] are some earlier general model serving systems. They study batching, caching, placement, and scheduling for serving single or multiple models. More recently, DVABatch [29] introduces multi-entry multi-exit batching. REEF [51] and Shepherd [158] propose preemption for serving. AlpaServe [82] utilizes model parallelism for statistical multiplexing. However, these general systems fail to take into account the auto-regressive property and token state of LLM inference, resulting in missed opportunities for optimization.

Specialized serving systems for transformers. Due to the significance of the transformer architecture, numerous specialized serving systems for it have been developed. These systems utilize GPU kernel optimizations [144, 2, 96, 85], advanced batching mechanisms [39, 153], model parallelism [108, 153, 2], and parameter sharing [166] for efficient serving. Among them, Orca [153] is most relevant to our approach.

Comparison to Orca. The iteration-level scheduling in Orca [153] and PagedAttention in vLLM are complementary techniques: While both systems aim to increase the GPU utilization and hence the throughput of LLM serving, Orca achieves it by scheduling and interleaving the requests so that more requests can be processed in parallel, while vLLM is doing so by increasing memory utilization so that the working sets of more requests fit into memory. By reducing memory fragmentation and enabling sharing, vLLM runs more requests in a batch in parallel and achieves a 2-4× speedup compared to Orca. Indeed, the fine-grained scheduling and interleaving of the requests like in Orca makes memory management more challenging, making the techniques proposed in vLLM even more crucial.

Memory optimizations. The widening gap between the compute capability and memory capacity of accelerators has caused memory to become a bottleneck for both training and inference. Swapping [56, 142, 114], recomputation [23, 61] and their combination [105] have been utilized to reduce the peak memory of training. Notably, FlexGen [124] studies how to swap weights and token states for LLM inference with limited GPU memory, but it does not target the online serving settings. OLLA [129] optimizes the lifetime and location of tensors to reduce fragmentation, but it does not do fine-grained block-level management or online serving. FlashAttention [30] applies tiling and kernel optimizations to reduce the peak memory of attention computation and reduce I/O costs. This chapter introduces a new idea of block-level memory management in the context of online serving.

5.10 Conclusion

This chapter proposes PagedAttention, a new attention algorithm that allows attention keys and values to be stored in non-contiguous paged memory, and presents vLLM, a high-throughput LLM serving system with efficient memory management enabled by PagedAttention. Inspired by operating systems, we demonstrate how established techniques, such as virtual memory and copy-on-write, can be adapted to efficiently manage KV cache and handle various decoding algorithms in LLM serving. Our experiments show that vLLM achieves 2-4× throughput improvements over the state-of-the-art systems.

Chapter 6

Conclusion and Future Work

In this dissertation, we designed algorithms and built systems to improve computational efficiency and reduce the engineering overhead when training and serving large language models, to democratize access to the most advanced AI models for everyone. In chronological order, we build:

- **TeraPipe** (chapter 2), which identifies sequence dimension as a new dimension for fine-grained pipeline parallel training for large language models.
- **Alpa** (chapter 3), a compiler that can automatically generate and execute parallel strategy for arbitrary neural networks with all existing parallel algorithms, including data, tensor, and pipeline parallelism.
- **AlpaServe** (chapter 4), which utilizes model parallelism to statistically multiplex the devices for deep learning serving. It is the first work to show that model parallelism can also benefit small models that can fit into a single GPU.
- **PagedAttention** (chapter 5), an attention algorithm inspired by virtual memory and paging in operating systems to improve the memory utilization during LLM serving, and **vLLM**, an end-to-end open-source LLM serving system that achieves state-of-the-art inference throughput.

We conclude this dissertation by discussing other related work we developed during the course of constructing this dissertation, reflecting on lessons learned in building open-source projects, and identifying potential areas for future research.

6.1 Other Work During the Construction of This Dissertation

In this section, we list other work developed during the construction of this dissertation.

Communication Optimization for LLMs. Communication has been the major overhead introduced in distributed training and inference. To reduce communication overhead, in [168], we developed a new broadcast-based algorithm for many-to-many communication in large-scale pipeline parallelism, accelerating communication by more than 10x. In **Hoplite** [167], we introduced a new set of collective communication algorithms that accelerate dynamic communication in distributed deep reinforcement learning by 4x.

LLM scaling law. Before GPT-3 [18] came out and showed its remarkable capabilities, people were still questioning whether larger models were more efficient than smaller models given a limited budget, which was later known as the study of *scaling law*. In [81], we showed that **training a larger model can be more efficient than smaller models**. This phenomenon occurs because larger models converge to lower errors faster due to their large number of parameters. Additionally, the large models are more robust to model compression techniques during inference. Thus, one can get the best of both training and serving efficiency by training very large models and then heavily compressing them.

Open-source chatbots. While LLMs have significantly advanced AI, their development has largely been confined to a few tech giants. To disrupt this concentration of power and make LLMs more accessible to a broader audience, we built **Vicuna** [24], a chatbot trained by fine-tuning LLaMA [137] on user-shared conversations on ShareGPT. It is the first open-source chatbot that achieves on-par conversation fluency as ChatGPT.

LLM Evaluation. To evaluate Vicuna's capability, we built **Chatbot Arena** [163], the first crowdsourced LLM benchmark that reflects the actual human preference for LLMs. Notably, Chatbot Arena is the first production deployment of vLLM. The cost reduction from vLLM enables serving Chatbot Arena to millions of users from a research lab. Chatbot Arena has been viewed as the most accurate benchmark for LLM-based chatbots.

6.2 Lessons Learned in Open-Source Development

Many technologies introduced in this dissertation have led to open-source projects that are widely adopted across the industry. In this section, we discuss several lessons we learned in the course of open-source development from an academic research environment.

Exciting new research opportunities come from first-hand experience and deployment. The biggest lesson we learned during the construction of this dissertation is that new research ideas often come from real-world experience, which is typically a unique advantage brought by working on open-source projects. As an example, before we started on the PagedAttention and vLLM project, we deployed a demo service that uses Alpa and AlpaServe to serve the OPT-175B [159] model from Meta to demonstrate the capability of Alpa and AlpaServe. During the deployment of the service, we found that simply deploying LLM without any optimization is extremely expensive and inefficient. We started to build a new system with continuous batching to optimize the serving performance. During our implementation, it became clear that the KV Cache memory management in existing systems was far from optimal, and this led to our research in PagedAttention and vLLM. In summary, having first-hand experience in real-world workloads typically reveals new insights that were previously missing in existing works, which can lead to new exciting research opportunities.

Accessible user interface. In both Alpa and vLLM projects, one of the fundamental philosophies we have at the beginning is to make sure the user interface is simple and easy to start. The definition of the interface here is general: from installation and starting example, all the way to configurations for system adaptation and code structure for advanced users to modify the code to their own use cases. This is crucial for the open-source project to grow: the ease of starting is crucial for bringing in new users to the project and the ease of modification is crucial for turning new users into advanced users who can add new features and eventually contribute to the open-source project. The users and contributors are the key for open-source projects to survive and thrive. Additionally, designing simple interfaces and system architectures can force us to hide the complexity from the user while keeping the system well-structured, which is often the base of a system research problem.

People and community. The most valuable asset of an open-source project is the community of its users and contributors. While lowering the barrier for participation is crucial, there are additional steps that maintainers can take to foster a vibrant open-source community. First, establishing a clear project roadmap helps contributors understand the project's priorities. Second, creating a platform for contributors to communicate with one another facilitates collaboration. Finally, and most importantly, maintainers should build personal connections and trust with contributors, gradually involving them more deeply in the project. In summary, a thriving open-source community is cultivated through a clear vision, effective communication channels, and strong trust between the contributors.

6.3 Future Work

In recent years, LLMs and neural networks for different application domains converge to the Transformer architecture, while the sizes of datasets and models have become the dominant factor for model accuracy. This shift underscores the growing importance of systems optimizations in the field of deep learning. To this end, we would like to propose future directions by exploring the fundamental system properties of LLMs and other deep learning applications, including *scheduling*, *heterogeneity*, and *fault tolerance*:

Application-aware scheduling for LLM serving. LLM applications evolve in a daily manner and the characteristics of these applications bring unique opportunities for scheduling optimizations. For instance, to accomplish a complex task with an LLM, users often specify an LLM to act as multiple specialized agents for the sub-tasks via different prompt instructions, and then these LLM agents interact with each other, as well as external tools, to jointly accomplish the task. The calls to one specific agent often share a large portion of the inputs. The calls to different agents form a structured task graph. Both can be utilized by the caching and execution scheduler. Other examples include retrieval-based generation and memory-augmented LLMs. Developing novel scheduling algorithms that utilize the application properties can further reduce the operational cost of these large models.

Heterogeneous computing for deep learning. Different deep learning models can have drastically different resource requirements, sometimes even for the same model at different application stages: Vision models are typically compute-bound, while LLMs are often memory-bound; Training is compute-bound, while inference is mostly memory-bound; Specifically for LLM inference, the prompt processing stage is often compute-bound, while generating new tokens is memory-bound (which inspired our work on PagedAttention). While GPUs are the king for compute-bound workloads, CPUs can be an efficient and low-cost alternative for memory-bound tasks. At a high level, utilizing heterogeneous hardware is an important approach to handle the different workload requirements. We are exploring developing an abstraction that can capture the requirements of different deep-learning tasks and building systems that can automatically distribute tasks to different hardware based on the requirements.

Fault-tolerance in massive scale GPU cluster. Modern GPU clusters often follow a supercomputer-style design: They assume individual GPUs do not fail often, and when failure happens, the system restarts from a previous global checkpoint. This design worked well back when individual deep learning jobs were using at most tens of GPUs. However, with the rise of LLMs, a single training job can take more than ten thousand GPUs. The failures of GPUs started to become an issue: the training job can fail more than once every day. Some failures are also subtle: a single bit-flip of one neural network weight may lead to a diverged training run hundreds of training steps later. However, GPUs are so expensive that the overhead of traditional fault-tolerant systems becomes intolerable. An exciting future research direction is to find a new computing paradigm that can handle faults in this scenario while minimizing overhead and maximizing performance.

Overall, the philosophy throughout this dissertation is to find fundamental problems in real-world applications, search for highly practical solutions, and meanwhile build deployable algorithms and systems, like Alpa and vLLM. As the scale of LLMs and other deep learning applications continues to grow, we firmly believe the next AI breakthrough will come from system innovations. The combination of rapidly emerging problems and finding real-world impactful solutions allows for rich, intellectually rewarding research that can directly shape the systems powering today's and tomorrow's most exciting applications.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: a system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. "DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale". In: *arXiv preprint arXiv:2207.00032* (2022).

[3] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. "Optimizing performance of recurrent neural networks on gpus". In: *arXiv preprint arXiv:1604.01946* (2016).

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. "Varuna: scalable, low-cost training of massive deep learning models". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 472–487.

[5] AWS Cluster Configuratoins. `https://aws.amazon.com/ec2/instance-types/p3/`.

[6] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. "Distributed Balanced Partitioning via Linear Embedding". In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 2016, pp. 387–396.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[8] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. "Gradient Compression Supercharged High-Performance Data Parallel DNN Training". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 2021, pp. 359–375.

[9] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. "{PipeSwitch}: Fast Pipelined Context Switching for Deep Learning Applications". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 499–514.

[10] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. “Pathways: Asynchronous distributed dataflow for ML”. In: *Proceedings of Machine Learning and Systems* 4 (2022).

[11] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. “Parallelism-Optimizing Data Placement for Faster Data-Parallel Computations”. In: *Proceedings of the VLDB Endowment* 16.4 (2022), pp. 760–771.

[12] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. “A neural probabilistic language model”. In: *Advances in neural information processing systems* 13 (2000).

[13] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. “A neural probabilistic language model”. In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155.

[14] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. “Barista: Efficient and scalable serverless serving system for deep learning prediction services”. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2019, pp. 23–33.

[15] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D’Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users’ guide*. SIAM, 1997.

[16] Ond rej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. “Findings of the 2016 Conference on Machine Translation”. In: *Proceedings of the First Conference on Machine Translation*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 131–198. URL: http://www.aclweb.org/anthology/W/W16/W16-2301.

[17] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[19] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. “TensorOpt: Exploring the tradeoffs in distributed DNN training with auto-parallelism”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.8 (2021), pp. 1967–1981.

[20] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. "ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training". In: *International Conference on Machine Learning*. 2021.

[21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[22] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "TVM: An automated end-to-end optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[23] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. "Training deep nets with sublinear memory cost". In: *arXiv preprint arXiv:1604.06174* (2016).

[24] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: https://lmsys.org/blog/2023-03-30-vicuna/.

[25] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. "Palm: Scaling language modeling with pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[26] Copy.ai. *Copy.ai: Write Better Marketing Copy and Content with AI*. https://www.copy.ai/.

[27] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. "InferLine: latency-aware provisioning and scaling for prediction serving pipelines". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 477–491.

[28] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627.

[29] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. "DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 183–198.

[30] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. "Flashattention: Fast and memory-efficient exact attention with io-awareness". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16344–16359.

[31] Robert I Davis, Ken W Tindell, and Alan Burns. "Scheduling slack time in fixed priority pre-emptive systems". In: *1993 Proceedings Real-Time Systems Symposium*. IEEE. 1993, pp. 222–231.

[32] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. "Distributed halide". In: *ACM SIGPLAN Notices* 51.8 (2016), pp. 1–12.

[33] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. "Llm. int8 (): 8-bit matrix multiplication for transformers at scale". In: *Advances in Neural Information Processing Systems* (2022).

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[35] Mengnan Du, Subhabrata Mukherjee, Yu Cheng, Milad Shokouhi, Xia Hu, and Ahmed Hassan Awadallah. "What do compressed large language models forget? robustness challenges in model compression". In: *arXiv preprint arXiv:2110.08419* (2021).

[36] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. *GLaM: Efficient Scaling of Language Models with Mixture-of-Experts*. 2021. arXiv: 2112.06905 [cs.CL].

[37] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models". In: *arXiv preprint arXiv:2007.01045* (2020).

[38] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. "DAPPLE: A pipelined data parallel approach for training large models". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 431–445.

[39] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. "TurboTransformers: an efficient GPU serving system for transformer models". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 389–402.

[40] FastAPI. *FastAPI*. https://github.com/tiangolo/fastapi. 2023.

[41] William Fedus, Barret Zoph, and Noam Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *Journal of Machine Learning Research* 23.120 (2022), pp. 1–39.

[42] John Forrest and Robin Lougee-Heimer. "CBC user guide". In: *Emerging theory, methods, and applications*. INFORMS, 2005, pp. 257–277.

[43] Richard J Forrester and Noah Hunt-Isaak. "Computational comparison of exact solution methods for 0-1 quadratic programs: Recommendations for practitioners". In: *Journal of Applied Mathematics* 2020 (2020).

[44] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. "Low latency rnn inference with cellular batching". In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–15.

[45] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. "Ai and memory wall". In: *RiseLab Medium Post* 1 (2021), p. 6.

[46] Github. *Github Copilot: Your AI Pair Programmer*. https://github.com/features/copilot. 2022. URL: https://github.com/features/copilot.

[47] Google. 2023. URL: https://bard.google.com/.

[48] Google Clould TPU Cluster Configurations. https://cloud.google.com/tpu.

[49] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. "Accurate, large minibatch sgd: Training imagenet in 1 hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[50] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 443–462.

[51] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. "Microsecond-scale Preemption for Concurrent {GPU-accelerated}{DNN} Inferences". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 539–558.

[52] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. "Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 539–558. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/han.

[53] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. "Pipedream: Fast and efficient pipeline parallel dnn training". In: *arXiv preprint arXiv:1806.03377* (2018).

[54] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[55] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: `2106.09685 [cs.CL]`.

[56] Chien-Chin Huang, Gu Jin, and Jinyang Li. "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1341–1355.

[57] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism". In: *Advances in neural information processing systems* 32 (2019), pp. 103–112.

[58] Huggingface. *Models - Huggingface*. `https://huggingface.co/models`.

[59] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "Serving deep learning models in a serverless platform". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, pp. 257–262.

[60] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. "Data movement is all you need: A case study on optimizing transformers". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 711–732.

[61] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. "Checkmate: Breaking the memory wall with optimal tensor rematerialization". In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 497–511.

[62] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. "Checkmate: Breaking the memory wall with optimal tensor rematerialization". In: *arXiv preprint arXiv:1910.02653* (2019).

[63] Zhihao Jia, Sina Lin, Charles Ruizhongtai Qi, and Alex Aiken. "Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks". In: (Feb. 2018).

[64] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. "TASO: optimizing deep learning computation with automatic generation of graph substitutions". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 47–62.

[65] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond data and model parallelism for deep neural networks". In: *arXiv preprint arXiv:1807.05358* (2018).

[66] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond data and model parallelism for deep neural networks". In: *SysML 2019* (2019).

[67] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 463–479.

[68] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. "One-level storage system". In: *IRE Transactions on Electronic Computers* 2 (1962), pp. 223–235.

[69] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. "Parallax: Sparsity-aware data parallel training of deep neural networks". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–15.

[70] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. "Dynamic tensor rematerialization". In: *arXiv preprint arXiv:2006.09616* (2020).

[71] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. "Reformer: The efficient transformer". In: *arXiv preprint arXiv:2001.04451* (2020).

[72] A. Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *ArXiv* abs/1404.5997 (2014).

[73] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *arXiv preprint arXiv:1404.5997* (2014).

[74] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. "Efficient memory management for large language model serving with pagedattention". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 611–626.

[75] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, Markus Weimer, et al. "Automating system configuration of distributed machine learning". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 2057–2067.

[76] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding". In: *International Conference on Learning Representations*. 2020.

[77] Brian Lester, Rami Al-Rfou, and Noah Constant. "The power of scale for parameter-efficient prompt tuning". In: *arXiv preprint arXiv:2104.08691* (2021).

[78] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. "Scaling distributed machine learning with the parameter server". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[79] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. "Pytorch distributed: Experiences on accelerating data parallel training". In: *arXiv preprint arXiv:2006.15704* (2020).

[80] Xiang Lisa Li and Percy Liang. "Prefix-tuning: Optimizing continuous prompts for generation". In: *arXiv preprint arXiv:2101.00190* (2021).

[81] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. "Train big, then compress: Rethinking model size for efficient training and inference of transformers". In: *International Conference on machine learning*. PMLR. 2020, pp. 5958–5968.

[82] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving". In: *arXiv preprint arXiv:2302.11665* (2023).

[83] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. "Terapipe: Token-level pipeline parallelism for training large-scale language models". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 6543–6552.

[84] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[85] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. "Rammer: Enabling holistic deep learning compiler optimizations with rtasks". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 2020, pp. 881–897.

[86] Naraig Manjikian and Tarek S Abdelrahman. "Scheduling of wavefront parallelism on scalable shared-memory multiprocessors". In: *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*. Vol. 3. IEEE. 1996, pp. 122–131.

[87] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. "Efficient resource provisioning in compute clouds via vm multiplexing". In: *Proceedings of the 7th international conference on Autonomic computing*. 2010, pp. 11–20.

[88] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. "Mixed precision training". In: *arXiv preprint arXiv:1710.03740* (2017).

[89] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. "Device placement optimization with reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2430–2439.

[90] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 561–577.

[91] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. "PipeDream: generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 1–15.

[92] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. "Memory-efficient pipeline-parallel dnn training". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 7937–7947.

[93] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. "Efficient large-scale language model training on GPU clusters using megatron-LM". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.

[94] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421.

[95] NVIDIA. *NCCL: The NVIDIA Collective Communication Library*. https://developer.nvidia.com/nccl. 2023. URL: https://developer.nvidia.com/nccl.

[96] NVIDIA. *FasterTransformer*. https://github.com/NVIDIA/FasterTransformer. 2023.

[97] NVIDIA. *Triton Inference Server*. https://developer.nvidia.com/nvidia-triton-inference-server.

[98] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. "Tensorflow-serving: Flexible, high-performance ml serving". In: *arXiv preprint arXiv:1712.06139* (2017).

[99] OpenAI. 2020. URL: https://openai.com/blog/openai-api.

[100] OpenAI. 2023. URL: https://openai.com/blog/custom-instructions-for-chatgpt.

[101] OpenAI. *ChatGPT*. https://chat.openai.com/chat. 2022. URL: https://openai.com/blog/chatgpt.

[102] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[103] LMSYS ORG. *Chatbot Arena Leaderboard Week 8: Introducing MT-Bench and Vicuna-33B*. https://lmsys.org/blog/2023-06-22-leaderboard/. 2023.

[104] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019), pp. 8024–8035.

[105] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. "POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 17573–17583.

[106] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. "A generic communication scheduler for distributed dnn training acceleration". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 16–29.

[107] A. Petrowski, G. Dreyfus, and C. Girault. "Performance analysis of a pipelined backpropagation parallel algorithm". In: *IEEE Transactions on Neural Networks* 4.6 (1993), pp. 970–981. DOI: 10.1109/72.286892.

[108] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. "Efficiently Scaling Transformer Inference". In: *arXiv preprint arXiv:2211.05102* (2022).

[109] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language models are unsupervised multitask learners". In: ().

[110] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "Zero: Memory optimization towards training a trillion parameter models". In: *arXiv preprint arXiv:1910.02054* (2019).

[111] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "Zero: Memory optimizations toward training trillion parameter models". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–16.

[112] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 401–417.

[113] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 3505–3506.

[114] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. "ZeRO-Offload: Democratizing Billion-Scale Model Training." In: *USENIX Annual Technical Conference*. 2021, pp. 551–564.

[115] Reuters. 2023. URL: https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/.

[116] Stewart Robinson. *Simulation: the practice of model development and use*. Bloomsbury Publishing, 2014.

[117] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/romero.

[118] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[119] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[120] Amazon Web Services. 2023. URL: https://aws.amazon.com/bedrock/.

[121] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 205–218.

[122] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. "Mesh-TensorFlow: Deep Learning for Supercomputers". In: *Neural Information Processing Systems*. 2018, pp. 10414–10423.

[123] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: A GPU cluster engine for accelerating DNN-based video analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[124] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. "High-throughput Generative Inference of Large Language Models with a Single GPU". In: *arXiv preprint arXiv:2303.06865* (2023).

[125] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. "Megatron-lm: Training multi-billion parameter language models using model parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[126] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. "Megatron-lm: Training multi-billion parameter language models using model parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[127] John F Shortle, James M Thompson, Donald Gross, and Carl M Harris. *Fundamentals of queueing theory*. Vol. 399. John Wiley & Sons, 2018.

[128] Balaram Sinharoy and Boleslaw Szymanski. "Finding Optimum Wavefront of Parallel Computation". In: *Parallel Algorithms and Applications* 2 (Aug. 1994). DOI: 10.1080/10637199408915404.

[129] Benoit Steiner, Mostafa Elhoushi, Jacob Kahn, and James Hegarty. "OLLA: Optimizing the Lifetime and Location of Arrays to Reduce the Memory Usage of Neural Networks". In: (2022). DOI: 10.48550/arXiv.2210.12924.

[130] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27 (2014), pp. 3104–3112.

[131] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.

[132] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. "Piper: Multidimensional Planner for DNN Parallelization". In: *Advances in Neural Information Processing Systems* 34 (2021).

[133] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. "Long Range Arena: A Benchmark for Efficient Transformers". In: *arXiv preprint arXiv:2011.04006* (2020).

[134] Google XLA Team. *XLA: Optimizing Compiler for Machine Learning*. 2017. URL: https://www.tensorflow.org/xla.

[135] ShareGPT Team. 2023. URL: https://sharegpt.com/.

[136] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv preprint arXiv:1605.02688* (2016).

[137] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[138] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017), pp. 5998–6008.

[139] Thijs Vogels, Sai Praneeth Karinireddy, and Martin Jaggi. “PowerSGD: Practical low-rank gradient compression for distributed optimization”. In: *Advances In Neural Information Processing Systems 32 (Nips 2019)* 32.CONF (2019).

[140] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. “PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 37–54.

[141] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. “Pacman: An Efficient Compaction Approach for {Log-Structured}{Key-Value} Store on Persistent Memory”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 773–788.

[142] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. “Superneurons: Dynamic GPU memory management for training deep neural networks”. In: *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 2018, pp. 41–53.

[143] Minjie Wang, Chien-chin Huang, and Jinyang Li. “Supporting very large models using automatic dataflow graph partitioning”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–17.

[144] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. “LightSeq: A High Performance Inference Library for Transformers”. In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*. 2021, pp. 113–120.

[145] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. “Self-Instruct: Aligning Language Model with Self Generated Instructions”. In: *arXiv preprint arXiv:2212.10560* (2022).

[146] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. “MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 945–960.

[147] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. “Transformers: State-of-the-art natural language processing”. In: *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 2020, pp. 38–45.

[148] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. "Transparent GPU Sharing in Container Clouds for Deep Learning Workloads". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 69–85. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/wu.

[149] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[150] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. "Automatic cross-replica sharding of weight update in data-parallel training". In: *arXiv preprint arXiv:2004.13336* (2020).

[151] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. "GSPMD: general and scalable parallelization for ML computation graphs". In: *arXiv preprint arXiv:2105.04663* (2021).

[152] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. "Xlnet: Generalized autoregressive pretraining for language understanding". In: *Advances in neural information processing systems*. 2019, pp. 5753–5763.

[153] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. "Orca: A Distributed Serving System for {Transformer-Based} Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 521–538.

[154] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. "OneFlow: Redesign the Distributed Deep Learning Framework from Scratch". In: *arXiv preprint arXiv:2110.15032* (2021).

[155] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[156] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. "Big bird: Transformers for longer sequences". In: *arXiv preprint arXiv:2007.14062* (2020).

[157] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. "AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning". In: *Advances in Neural Information Processing Systems* 33 (2020).

[158] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. "SHEPHERD: Serving DNNs in the Wild". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 787–808. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/zhang-hong.

[159] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. “Opt: Open pre-trained transformer language models”. In: *arXiv preprint arXiv:2205.01068* (2022).

[160] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. “Faster and cheaper serverless computing on harvested resources”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 2021, pp. 724–739.

[161] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. “MiCS: Near-linear Scaling for Training Gigantic Model on Public Cloud”. In: *arXiv preprint arXiv:2205.00119* (2022).

[162] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. “Multi-resource interleaving for deep learning training”. In: *Proceedings of the ACM SIGCOMM 2022 Conference.* 2022, pp. 428–440.

[163] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. “Judging llm-as-a-judge with mt-bench and chatbot arena”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[164] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. “Ansor: Generating high-performance tensor programs for deep learning”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* 2020, pp. 863–879.

[165] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. “Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22).* 2022, pp. 559–578.

[166] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. “PetS: A Unified Framework for Parameter-Efficient Transformers Serving”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22).* 2022, pp. 489–504.

[167] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. “Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems”. In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference.* 2021, pp. 641–656.

[168] Yonghao Zhuang, Lianmin Zheng, Zhuohan Li, Eric Xing, Qirong Ho, Joseph Gonzalez, Ion Stoica, Hao Zhang, and Hexu Zhao. “On optimizing the communication of model parallelism”. In: *Proceedings of Machine Learning and Systems* 5 (2023).

[169] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. "Parallelized Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta. Vol. 23. Curran Associates, Inc., 2010, pp. 2595–2603.