Architecture Design for Efficient LLM Training and Inference

By

Song Bian

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: February 2nd, 2026

The dissertation is approved by the following members of the Final Oral Committee:

Shivaram Venkataraman, Associate Professor, Computer Sciences
Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Frederic Sala, Assistant Professor, Computer Sciences
Junjie Hu, Assistant Professor, Biostatistics and Medical Informatics

To my parents, family, and friends.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Shivaram Venkataraman. He was the primary reason I chose to begin my PhD at the University of Wisconsin-Madison. Before starting my doctoral studies, my training was largely in databases; through his teaching, mentorship, and guidance, I developed the systems and machine learning foundation that shaped me into a machine learning systems researcher.

I am grateful to Andrea C. Arpaci-Dusseau, Frederic Sala, and Junjie Hu for serving on my dissertation committee and for their guidance throughout this work. I also thank Paraschos Koutris for insightful discussions that helped shape my first project at UW-Madison. In addition, I would like to thank Anhai Doan, Ming Liu, Yingyu Liang, and Matthew D. Sinclair for their generous support during my PhD studies. Finally, I thank Zhao Zhang at Rutgers University for providing GPU resources that supported my research projects.

During my time at Madison, I had the privilege of collaborating with Saurabh Agarwal, Tareq Mahmood, Hongyi Wang, and Minghao Yan. I am also grateful to the members of Shivaram's group, Tzu-Tao Chang, Fanchao Chen, Johannes Freischuetz, Rutwik Jain, Konstantinos Kanellis, Jason Mohoney, Seth Ockerman, and Brandon Tran, for their thoughtful feedback and support throughout my PhD. I was fortunate to be part of madSystems, an exceptionally talented community, and I especially thank Vinay Banakar, Tingjia Cao, Xiangpeng Hao, Guanzhou Hu, Suyan Qu, Sujay Yadalam, Chenhao Ye, and Shawn Zhong for helping me build a strong foundation in core systems research.

I also want to thank my friends, Mu Cai, Haotian Liu, Yue Zhang, Ting Cai, Shaleen Deep, Austen Fan, Zhiwei Fan, Yujun He, Wenjie Hu, Xiating Ouyang, Yifei Yang, Ling Zhang, Hangdong Zhao, Zerui Guo, Wentao Hou, Yuyuan Kang, Chendong Wang, Cong Ding, Yunhan Hu, Hongyi Huang, Xuanyu Peng, Leitian Tao, Yadu Babuji, Minu Mathew, and Haotian Xie for their friendship and support throughout this journey.

I would like to acknowledge several collaborators and mentors from my summer intern-

ships. During the summer of 2022, I interned at Alibaba Group, where I worked with Bolin Ding. Bolin taught me invaluable lessons about conducting research with real industrial impact. I also benefited greatly from the guidance of senior researchers and engineers, including Wei Lin, Harry Liu, Jason Wu, and Wotao Yin, whose insights helped shape my research direction.

In the summer of 2025, I interned at Amazon and worked with Tao Yu and Youngsuk Park. I am especially grateful to Tao for his outstanding mentorship and guidance in machine learning research and academic writing. I also thank Youngsuk for generously providing GPU resources and support.

Finally, I am deeply grateful to my parents and grandparents for their unwavering support and constant encouragement throughout this journey.

Contents

List of Tables

List of Figures

Abstract

Text data plays an important role in many real-world applications, including information retrieval and question answering. To address these text-centric tasks, large language models (LLMs) have been trained on large-scale corpora in recent years, achieving strong performance across a wide range of downstream tasks, including text classification, text generation, mathematical reasoning, and code generation. This progress has driven widespread industrial efforts, with many companies developing and training their own commercial language models, such as GPT, Gemini, DeepSeek, and Qwen, for real-world use. These performance gains have been driven primarily by scaling training data, model size, and compute. However, scaling also introduces challenges for efficient training and inference in LLMs.

This dissertation explores approaches to efficient model training and deployment via model architecture design. We first introduce a training-efficient architecture designed to alleviate the communication overhead associated with model parallelism, which is required when large language models surpass the memory limits of a single GPU. Our approach reduces the activations exchanged during communication and has been integrated into Megatron-LM, a widely used industrial distributed training system, improving end-to-end pretraining throughput by 1.26×. Next, we develop architecture-aware scheduling strategies to increase GPU utilization by jointly optimizing the model-architecture design and the scheduler. In both cluster experiments and simulations, our approach improves JCT by up to 1.62× and makespan by up to 1.15×, while reducing the worst finish-time fairness ratio by 3.77× over existing baselines. We further show that architectural choices are critical to inference efficiency and downstream performance. In particular, choosing an appropriate aspect ratio, defined as the ratio of hidden size to the number of layers, improves inference efficiency for large language models by 1.8× and maintains accuracy over downstream tasks. Building on this insight, we finally investigate how to incorporate architectural factors

into existing scaling laws. We propose a two-step conditional approach that extends the Chinchilla scaling laws with additional architectural parameters, including hidden size, grouped-query attention, and the mlp-to-attention ratio, to capture the trade-off between inference efficiency and model accuracy. Guided by these extended scaling laws, we train models with optimized architectures that achieve up to 2.1% higher accuracy and 42% greater inference throughput than LLaMA-3.2, while using the same fixed pretraining datasets and the same parameter count.

Chapter 1

Introduction

1.1 Motivation

Text data is one of the most abundant and valuable modalities in computing, underpinning a wide range of applications, including information retrieval [87, 101, 118, 163], recommendation [88, 196], question answering [83, 153, 219], translation [100, 137], and summarization [149, 227]. To handle natural language processing (NLP) tasks, traditional approaches relied on relatively simple language representations. For example, bag-of-words [162] models represent documents as unordered word-count vectors; TF-IDF [161, 180] refines this by reweighting counts to emphasize terms that distinguish a document from the broader corpus; and word embeddings such as Word2Vec [122] move beyond sparse counts to dense vectors that capture semantic and syntactic similarity, improving robustness to basic lexical variation. Together, these methods enabled many effective NLP pipelines, but they share a key limitation: they only partially capture how meaning emerges from context and interactions among words within a sentence or discourse. Bridging this gap calls for models that learn contextual representations directly from large-scale text and optimize language understanding and generation end-to-end, setting the stage for large language models.

Large language models (LLMs), built on the Transformer architecture [192], leverage large-scale pretraining to acquire general linguistic competence and broad world knowledge [35, 46, 113, 216]. As a result, they perform strongly across tasks spanning classification [165, 195], generation [160, 226], reasoning [90, 182, 203], and mathematics [44, 67]. More recently, LLMs have become key building blocks for tool-using assistants and agentic

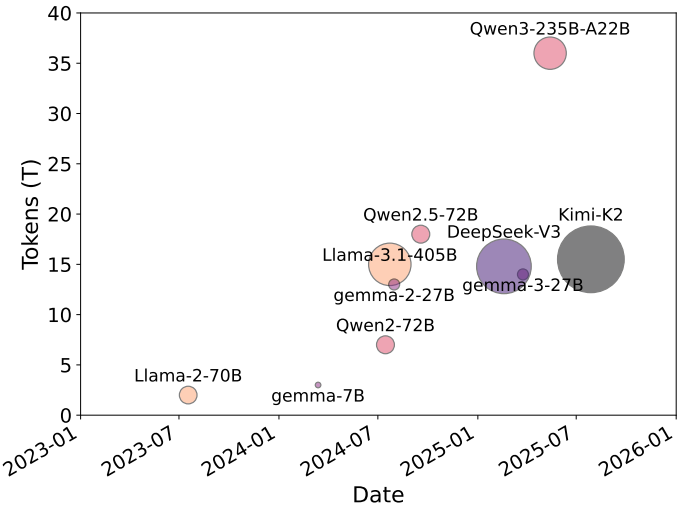


Figure 1.1: Open-Source LLMs on Model Size and Training Tokens: We summarize model size and training-token counts for the evaluated open-source LLMs: LLaMA [56, 190], Qwen [216, 217], Gemma [186–188], DeepSeek [113], and Kimi [189].

systems that can plan, invoke external APIs, generate and execute code, and interact with complex software environments [167, 173, 198, 220]. A defining feature of these models is their generality: improvements to the base model often yield broad gains across many downstream applications without task-specific architectural changes. This leverage has amplified the impact of LLM progress and motivated many organizations to train and deploy their own models, for example, OpenAI's GPT series [147], Anthropic's Claude [7], xAI's Grok [8], and DeepSeek [62, 113].

A major driver of progress in LLMs has been scaling. Over the past several years, a consistent empirical pattern has emerged: increasing model size, training data, and compute budget produces predictable improvements in training loss and downstream capability [113, 189, 216]. For example, as shown in Figure 1.1, Qwen2.5-72B scales pretraining data from 7T to 18T tokens relative to Qwen2-72B [217]; Llama-3.1-405B expands model size from 70B to 405B parameters compared with Llama-2-70B [56]; and Kimi-K2 adopts a Mixture-of-Experts (MoE) design with 32B activated parameters and 1T total parameters [189], achieving strong performance across downstream tasks. These trends underscore scaling as the primary driver of quality gains, with scaling laws helping predict the benefits of extra data and compute [70, 85, 125].

However, scaling also exposes the practical limits of LLMs. Larger models require

disproportionately more resources to train, and bottlenecks increasingly arise from system constraints, memory capacity, bandwidth, and communication across accelerators, rather than compute alone. In response, the community has developed parallelization techniques including ZeRO optimization [151, 152, 156, 197], Fully Sharded Data Parallelism (FSDP) [230], pipeline parallelism [75, 127], and tensor parallelism [174], along with open-source distributed training frameworks such as Megatron-LM [174] and DeepSpeed [154]. In production, overall costs are often driven more by inference than by training, since inference is executed repeatedly at scale; methods such as continuous batching [224], speculative decoding [38, 99], and KV-cache management [97, 222, 235] are widely used to increase throughput and reduce latency.

Despite notable progress, several challenges remain. On the training side, memory-saving strategies, such as Fully Sharded Data Parallelism (FSDP) [230] and model parallelism (e.g., pipeline [75, 127] and tensor parallelism [174]), can substantially reduce the memory footprint of large language models, but they often shift the bottleneck to communication, introducing extra synchronization and data-transfer costs. Moreover, large language models now dominate deep learning workloads, yet their training characteristics differ substantially from those of smaller models [72, 221]. This gap creates opportunities to further optimize existing GPU cluster schedulers by incorporating LLM architectural considerations. On the inference side, most optimizations are still system-level (e.g., batching [224] and KV-cache management [97, 235]). However, architectural choices can also raise throughput; for example, Mixture-of-Experts (MoE) activates only a subset of parameters per token, increasing effective capacity while reducing per-token compute, which can translate into higher serving throughput [150, 171]. More broadly, model-system co-design remains an open area: it is still unclear how these architectural modifications should be scaled to larger settings, and whether they can maintain downstream accuracy as model size, sparsity, and deployment constraints evolve.

1.2 Dissertation Goal

Motivated by the above, this dissertation proposes LLM architectural optimizations to alleviate scaling-related efficiency bottlenecks. We target two coupled challenges, efficient training and efficient inference, and tackle the following technical problems:

Problem 1: Efficient Large Language Model Training. First, we focus on improving training throughput in distributed training of large language models. Training frontier-scale LLMs means spreading the workload across many accelerators, and the biggest bottlenecks are often not peak FLOPs. More often, performance is constrained by memory capacity for parameters, optimizer state, and activations. To mitigate the memory constraints of large language model training, model parallelism, such as pipeline parallelism [127, 129] and tensor parallelism [131, 174], is widely used. However, model parallelism introduces substantial communication overhead. As models scale, the volume of intermediate activations exchanged across devices can grow large enough to dominate step time. Even when computation remains tractable, end-to-end efficiency may deteriorate because communication is costly, difficult to overlap with compute, and increases memory traffic [239].

Numerous prior works in the data-parallel setting have explored gradient compression to reduce the communication costs of training [13, 27, 111, 179, 194, 199]. To be specific, signSGD [27] addresses the gradient-communication bottleneck by transmitting only the sign bits of the mini-batch stochastic gradients. Moreover, PowerSGD [194] presents a power-iteration low-rank gradient compression method that provides fast compression, communication-efficient all-reduce aggregation, and end-to-end test accuracy comparable to SGD. However, compression in model parallelism (MP) differs fundamentally from compression in data parallelism (DP), due to the distinct statistical properties of activations and gradients. Existing compression methods may not adequately reduce activation communication, so new techniques are required to lower the communication cost of model parallelism.

Problem 2: Model Architecture Affects Scheduling Efficiency. Deep learning (DL) models are trained on large GPU clusters, which are shared by several different jobs, and a scheduler is used to assign the resources to each DL training job. Given the prevalence of DL jobs in data centers, several schedulers [60, 73, 79, 117, 130, 145, 212, 213, 236] have been designed to tailor their policies to the unique characteristics of DL jobs.

The initial set of DL schedulers was designed to improve key scheduling metrics such as job completion time (Tiresias [60]) or finish-time fairness (Themis [117]). At a high level, these schedulers considered a set of queued jobs, available cluster resources, and various metrics (*e.g.*, attained service [60]), and selected a subset of jobs to execute on the cluster. While these works successfully optimized their target metrics, cluster operators still

observed low GPU utilization [80, 205]. Prior studies further show that utilization is strongly influenced by how and where jobs are placed in the cluster [117, 212]. In addition, as models grow [56, 113, 189], LLM migration costs rise sharply relative to traditional workloads, increasing the importance of migration decisions. Therefore, beyond scheduling policies, DL resource managers should incorporate placement policies that determine where jobs run, how they are migrated across scheduling rounds, which parallelization strategy to use based on model architecture [233], and how jobs are co-located or packed [73].

In existing works, these placement policies are either implemented as ad-hoc heuristics, or incorporated as a part of a larger joint optimization problem. For example, Tiresias [60] uses a heuristic to classify which DL training jobs require consolidated placement and Pollux [145] uses a heuristic to minimize interference by "ensuring at most one distributed job is allocated to each node". On the other hand, Gavel [130] and POP [128] jointly perform packing with scheduling by formulating the packing constraints as part of their optimization problem.

Unfortunately, both approaches, adding as ad-hoc heuristics or incorporating placement constraints within the optimization problem, have drawbacks. Introducing placement constraints through ad-hoc heuristics often results in suboptimal performance as these heuristics fail to capture the complex inter-dependencies among scheduling decisions (Figure 4.1). Secondly, as the hardware and jobs evolve, these heuristics can become obsolete, and new heuristics might be needed, requiring manual effort. For example, Tiresias uses a model's parameter count to determine placement; however, the evolution in model architecture has led to this heuristic being obsolete [12]. Meanwhile, integrating placement constraints into the core scheduling optimization significantly increases the number of variables while making the optimization problem more complex. This leads to poor scalability as the cluster sizes and number of jobs increase (Figure 4.2). These challenges highlight the need for a principled and scalable approach for integrating placement policies in DL schedulers, particularly with model architecture taken into account.

Problem 3: Scalable LLM Inference via Architectural Optimization. Inference creates a distinct efficiency challenge that is often more economically significant than training. Once trained, LLMs are deployed for high-concurrency serving or integrated into workloads with massive token generation rates [17, 97, 235]. At that scale, inference costs can escalate quickly, and even small inefficiencies may compound into substantial operat-

ing expenses. The main determinants of inference efficiency include latency, throughput, memory footprint, and the overhead of preserving autoregressive state (e.g., key-value caches) [16, 17, 97, 235]. Together, these factors influence both end-user experience and the total cost of operating models at scale.

Inference efficiency depends not only on system-level optimization but also critically on model architecture [171]. Even with similar parameter counts, models can exhibit markedly different inference behavior because architectural choices affect memory-access patterns, layer organization, attention mechanisms, and the balance between computation and bandwidth demands. For example, Gemma-7B [187], Qwen2.5-7B [217], and Mistral-7B-v0.3 [82] have comparable parameter budgets, yet their differing architectures result in different inference throughput. To be specific, on an A100 GPU with fixed input and output lengths, Gemma-7B achieves a throughput of 449.61 tokens/s, whereas Qwen2.5-7B achieves 1535.56 tokens/s. Architecture also impacts accuracy on downstream tasks [85], motivating an open question: Given the same pretraining data and parameter constraints, can we design models that are both inference-efficient and highly accurate? To address this question, it is crucial to study how model architecture affects inference efficiency and accuracy, and to optimize the architectures of existing open-weight models.

Problem 4: Scaling Architecture-Aware Large Language Models. Prior scaling law studies [10, 70, 85, 91, 125] consistently find that scaling model parameters, training data, and compute lowers pre-training loss, improves downstream performance [24, 67], and correlates with emergent capabilities [202]. These principles underpin many modern frontier LLMs [62, 190, 216]. However, an exclusive focus on training neglects the dominant challenges of real-world deployment [40, 126, 210]. In practice, inference often drives the marginal cost of operating large models [139, 164], especially as LLMs are increasingly embedded in multi-step reasoning and agentic pipelines [34, 61, 116, 144, 176]. Although numerous architectural ideas have been proposed to improve inference efficiency, e.g., grouped-query attention [18], linear attention [86], sliding window attention [26], and DeltaNet [218], it is unclear how their accuracy-efficiency tradeoffs evolve with scale. This leaves a central question: can we design scaling laws that capture the trade-off between inference efficiency and accuracy in LLMs, incorporating the impact of architectural decisions?

A recent study proposed scaling laws that incorporate the total FLOPs from both training and inference [164]. However, their formulation requires estimating the total number of

tokens generated over a model's entire lifespan. Because inference is performed repeatedly during deployment, this assumption renders the proposed scaling law impractical for real-world use. Therefore, we need a general approach that incorporates architectural factors into existing scaling laws to jointly optimize inference efficiency and training loss at scale, enabling architecture-aware scaling of large language models.

1.3 Contributions

This dissertation addresses the challenges outlined above through the following technical contributions. These contributions are supported by three papers [28–30], each providing a distinct perspective on efficient LLM design.

Integrating Compression into LLM Architecture. We propose MCBench, a benchmarking tool that builds on Megatron-LM [174] and develops APIs for implementing and evaluating MP compression. As a part of MCBench, we implement quantization-based compression method [21, 27], low-rank and sparsification-based gradient compression techniques such as PowerSGD [194], Top-K [19, 22], and Random-K [183, 214], along with a learning-based compression approach, *i.e.*, auto-encoders (AEs) [68]. We view the AE-based method as a form of architectural design, since the autoencoder can be integrated into the model. We then show examples of how these algorithms can be integrated with several popular datasets, such as CIFAR-10 and CIFAR-100, and models such as $\text{BERT}_{\text{BASE}}$, ViT-Base, OPT, and XLM-RoBERTa-XL from HuggingFace [208]. In addition, this benchmark can also be extended to integrate recent models such as LLaMA [190] and Code LLaMA [158].

Using MCBench, we present the first comprehensive study of achieving communication efficiency using compression methods in MP settings. Our experiments cover both pre-training and fine-tuning tasks, assessing the impact of various compression methods on throughput and accuracy. In total, we evaluate compression methods across over 200 unique configurations, employing various compression algorithms, training stages, models, and more than 10 datasets [92, 195]. We also develop the first cost model for MP with activation compression, validate it with experimental data (§3.3.4), and use it to derive insights into MP compression at scale. Our cost model and experiments reveal the following **key takeaways**:

1. Learning-based compression methods are most suitable for MP. In the pre-training stage, only learning-based methods (AEs) offer speedup (up to 16%) and preserve

the model's accuracy. Sparsification-based methods (Top-K) improve training time but compromise accuracy. Quantization methods maintain the model's accuracy but slow down the training time. Low-rank methods (PowerSGD) slow down training time and degrade accuracy. In the fine-tuning stage, all evaluated compression methods fail to provide end-to-end speedup, because their encoding and decoding overhead outweighs the reduced communication time. We provide further analysis in §3.3.2 and §3.3.3.

2. MP compression can yield around 25% throughput improvements for large models when cluster size is proportionally scaled. Using our cost model for MP with activation compression, we find that when we perform weak scaling [131] (*i.e.*, we increase cluster size proportional to model size), AE-based compression can yield around 25% throughput improvements for models with 100s of billions of parameters [35].

3. MP compression should be avoided for challenging ML tasks. Our findings reveal that employing MP compression can significantly compromise model accuracy, especially in the context of challenging ML tasks, *e.g.*, CIFAR-100 is more challenging than CIFAR-10 (§3.3.2 has an in-depth discussion). Moreover, Foundation models (FMs) like ViT-Base, when exposed to MP compression, can deliver performance worse than smaller models such as ResNet-50. As a result, we recommend that for challenging ML tasks, FMs should be trained without applying MP compression.

Efficient Scheduling of LLM workloads. We introduce a new placement policy for training clusters. Our key insight is that many placement constraints can be formulated as a graph matching problem, which can be efficiently solved using established algorithms such as the Hungarian Algorithm [94]. Precisely, we can build a graph $G = (V_1, V_2, E)$, where V_1 represents the set of jobs that are already placed on the cluster and V_2 represents the set of jobs that need to have a placement constraint applied. For example, to minimize job migration, Tesserae constructs a graph $G^M = (V_1^M, V_2^M, E^M)$, where V_1^M represents current placement plan and V_2^M represents the placement plan following migration, the weight w_{e^M} of edge $e^M = (x, y)$ quantifies the migration cost if the node x from the original placement is reassigned to node y in the new placement. With this formulation, we show that we can formulate a number of placement policies, including packing and migration can be efficiently solved even for large clusters.

We incorporate our new placement policies and design Tesserae, a deep learning (DL) scheduler where placement policies take as input the jobs chosen by existing scheduling

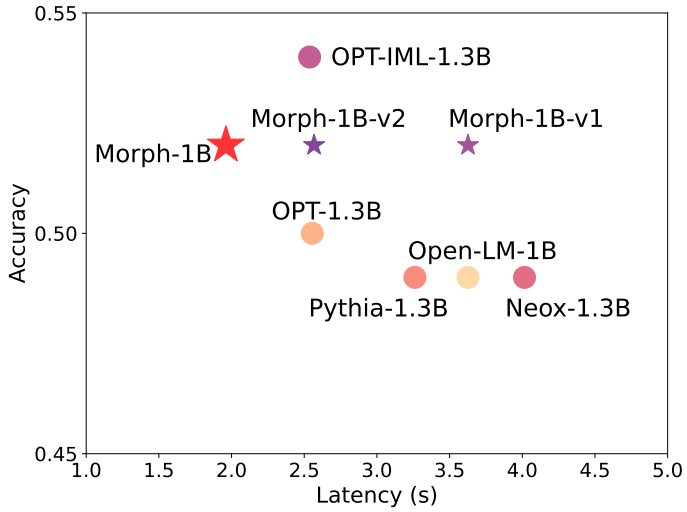


Figure 1.2: We train Morph-1B and its variant models on 30B tokens. The results indicate that Morph-1B maintains high accuracy on downstream tasks and achieves faster inference than open-source models and their variants. OPT-IML-1.3B achieves slightly higher performance on downstream tasks than Morph-1B since it is trained on 180B tokens [77] and is instruction-tuned. We obtain the accuracy by evaluating models on 11 downstream tasks used by Open-LM [63]. The inference latency is collected by using the Hugging Face `generate` function on a single NVIDIA Ampere 40GB A100 GPU with batch size 1, input length 128, and output length 256.

policies such as Tiresias [60], Themis [117], and come up with the final job placement. This design also allows DL schedulers to combine multiple possible placement policies. We evaluate Tesserae using multiple workloads derived from prior schedulers [80, 130, 236]. Our experiments first show that Tesserae's approach leads to significantly higher throughput than the existing heuristic-based placement policies, improving JCT by 1.62× and makespan by 1.15×. Next, we demonstrate that Tesserae can easily adapt to hardware changes, such as varying GPU types, by showing that Tesserae adapts to altered GPU configurations without requiring any additional tuning. Furthermore, we show that Tesserae is compatible with a range of scheduling policies. Finally, we evaluate the scalability of Tesserae and find that it remains efficient as the number of jobs increases, taking less than 1.6 seconds for a cluster with 256 GPUs and 2048 jobs.

Inference Efficiency via Architecture. To answer the following question mentioned in §1.2, we first show that the number of parameters is not the exclusive factor affecting

inference efficiency. As illustrated in Figure 5.1, the model architecture also plays a critical role. To gain deeper insight, Figure 5.2 shows that, with all other architectural factors held constant, increasing the aspect ratio reduces inference latency. Motivated by this observation, we introduce inference-efficient scaling laws that build on the Chinchilla scaling law and incorporate aspect ratio as an explicit variable, as shown in Eq. (5.4). Additionally, due to the disparity between model loss and accuracy in downstream tasks, we develop a novel method shown in Figure 5.6 that utilizes inference-efficient scaling laws to rank various model architectural choices.

To fit the inference-efficient scaling laws, we train more than 60 models ranging from 80 million to 339 million parameters for up to 13 billion tokens and record the loss of models. We also train several models with more than 1 billion parameters and 20 billion tokens to evaluate the predictive power of the fitted inference-efficient scaling laws. We observe that overtraining plays a critical role in obtaining an accurate scaling law and that our inference-efficient scaling law is more accurate and robust than the Chinchilla scaling law. Using only 6 data points and 85 A100 GPU hours for curve fitting, our inference-efficient scaling law can still accurately predict the loss of scaled-up models, as discussed in §5.3 and §5.4.

Lastly, we train the Morph-1B model using the best model configuration predicted by our inference-efficient scaling law and ranking algorithm. Morph-1B is optimized from Open-LM-1B [63] by reducing the depth from 24 to 12 layers and increasing the hidden size from 2048 to 3072. Figure 1.2 summarizes our main results. Compared to other open source models of similar size, Morph-1B improves the inference latency by 1.8$\times$ while maintaining accuracy over downstream tasks.

Architecture-Aware Scaling. While the contribution above extends the Chinchilla scaling laws by incorporating architectural considerations (e.g., aspect ratio), it has several notable limitations. First, the study examines only aspect ratio, defined as the hidden size divided by the number of layers, as the architectural factor. However, as shown in Figure 6.1, aspect ratio alone does not capture the full set of factors that affect inference efficiency in large language models. Second, model depth strongly influences accuracy: reducing the number of layers can impair generalization after fine-tuning [142]. Finally, the study does not provide a general framework for incorporating broader architectural choices, such as hidden size and GQA, into scaling laws.

To address these limitations, we fix the number of layers and study the effect of other architectural factors, including grouped-query attention (GQA), hidden size, and the mlp-to-attention ratio. This design choice is motivated by recent open-weight models such as LLaMA [190], Qwen [216], Gemma [187], and Phi [9], which, despite having a comparable number of parameters, adopt markedly different architectural designs.

Our primary goal is to investigate how model architecture influences both inference efficiency and model accuracy. We begin by comparing the inference efficiency of models with identical parameter counts but varying architectures. Next, we train over 200 models, ranging from 80M to 297M parameters on up to 30B tokens, to systematically characterize the relationship between architectural design and accuracy. Guided by these empirical findings, we introduce a conditional extension of the Chinchilla scaling laws that incorporates architectural parameters, establishing a general framework for identifying model architectures that balance inference efficiency and performance.

Finally, we validate this framework by fitting the proposed scaling law on models between 80M and 297M parameters, and evaluating its predictions when scaling up to pretrain 3B-parameter models. Our results demonstrate that, under identical training setups, the derived optimal 3B-parameter architecture achieves up to 42% higher inference throughput than the LLaMA-3.2-3B architecture, while maintaining better accuracy.

1.4 Organization

The remainder of this thesis is organized as follows.

Chapter 2 provides the background for the rest of the dissertation by covering distributed LLM training, inference systems and optimization, and scaling laws such as the Chinchilla scaling laws.

Chapter 3 examines activation communication in model-parallel training and assesses activation compression as an architectural approach to reducing communication overhead. It also identifies the regimes in which compression translates into improved end-to-end training efficiency.

Chapter 4 introduces a new placement policy that jointly selects parallelism and GPU co-location to improve cluster utilization. Using profiled co-run performance, it formulates packing as maximum-weight matching, improving throughput, JCT, and makespan over prior schedulers.

Chapter 5 characterizes how model architecture determines inference cost. It examines how structural design choices affect per-token efficiency and presents architecture-driven strategies to improve serving efficiency while maintaining model quality.

Chapter 6 proposes an architecture-aware extension of the Chinchilla scaling laws that incorporates inference-efficiency objectives into scaling decisions. Using this framework, it guides the design of models that remain efficient as they scale.

Chapter 7 summarizes the thesis contributions and connects the findings across training, inference, and scaling. It discusses implications for future co-design of model architecture and system infrastructure, emphasizing the role of efficiency considerations in both development and deployment. Finally, it outlines future directions for extending these ideas to new model families, larger scales, and diverse deployment settings.

Chapter 2

Background

2.1 Distributed Training

Large language models (LLMs) are trained by optimizing next-token prediction on large text corpora using gradient-based methods [136, 147]. As model parameter counts and dataset sizes have grown, training has become a systems problem: a single accelerator rarely has sufficient memory or compute to train state-of-the-art models within practical time constraints. As a result, modern LLM training pipelines rely on distributed training across many GPUs connected by high-bandwidth networks [154, 174].

A standard approach is data parallelism (DP), which partitions training examples across workers while replicating the full model on each device [69, 107]. In each iteration, every worker computes gradients on its local mini-batch and synchronizes them across workers [169]. When full model replication exceeds device memory, fully sharded data parallelism (FSDP) reduces the per-device footprint by sharding parameters, gradients, and optimizer states across workers, and gathering full parameters only when needed for computation [151, 230]. For sufficiently large models or long-context training, peak activation memory or per-layer compute may remain too large for a single device even with sharding; in these cases, model parallelism (e.g., tensor or pipeline parallelism), often combined with FSDP, may be required.

Model parallelism (MP) divides the model among multiple workers, allowing large models to be trained by only requiring each worker to maintain a portion of the entire model in memory, as shown in Figure 2.1 and Figure 2.2. There are two main paradigms for MP:

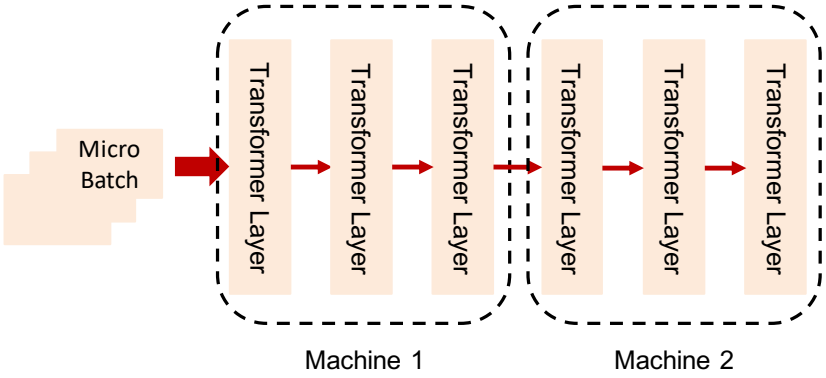
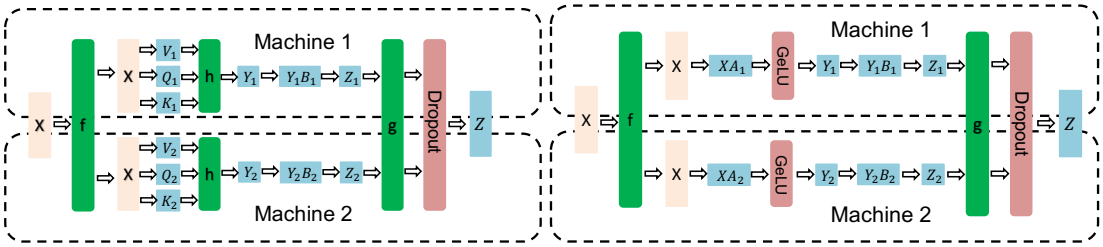


Figure 2.1: Pipeline Parallelism: Split the 6 Transformer layers into 2 pipeline stages and stream micro-batches through them sequentially, overlapping Machine 1's forward/backward passes with Machine 2's computation to increase utilization.



(a) Self-Attention **(b)** MLP

Figure 2.2: Tensor Parallelism: Within each Transformer layer, self-attention and MLP matrix multiplications are sharded across devices, and synchronization merges the partial results into the final output.

inter-layer pipeline parallelism (PP) and intra-layer tensor parallelism (TP). PP divides the layers among workers, with each worker executing the forward and backward computations in a pipelined fashion across different training examples [127, 129]. For example, a mini-batch of training examples can be partitioned into smaller micro-batches [75], with the forward computation of the first micro-batch taking place on one worker while the forward computation of the second micro-batch happens on another worker in parallel. PP involves a communication overhead due to the point-to-point communication between workers. TP divides the tensor computations among workers [89, 115, 170]. In particular, we consider a specialized strategy developed for Transformer models that divides the two GEMM layers in the attention module column-wise and then row-wise, with the same partitioning applied

to the MLP module [131, 174].

2.2 GPU Cluster Schedulers

GPU cluster schedulers have been actively researched in recent years [37, 41, 73, 76, 79, 105, 106, 141, 213]. In detail, Gandiva [212] uses time and space sharing to improve cluster utilization. Tiresias [60] utilizes two-dimensional indexes to reduce queuing delays. Themis [117] and Shockwave [236] design algorithms to trade off fairness for efficiency. Pollox [145] co-optimizes system throughput and statistical efficiency to minimize average JCT. Synergy [123] allocates CPU and memory to sensitive Jobs. Muri [232] considers multi-resource interleaving to improve resource utilization. The closest scheduler to us is Gavel [130]. In contrast, Gavel [130] formalizes each policy as an optimization problem and does not consider optimizing the parallelism strategy for foundation model training with space sharing.

To improve GPU utilization, GPU sharing is one possible way, which has been deployed in GPU cluster schedulers [130, 205, 212, 213]. Here, we introduce several GPU-sharing tools. Multi-Process Service [124] and Multi-Instance GPU [120] enable multiplex jobs on NVIDIA GPUs. PipeSwitch [25] and Salus [225] are designed for fast job switching and memory sharing. Zico [110] efficiently shares memory among packing jobs without exceeding a given memory budget. MuxFlow [231] supports safe and fast space-sharing in the production cluster.

2.3 Serving Systems

Once trained, LLMs are deployed to produce text by autoregressive decoding, where tokens are generated sequentially. This inference workload is quite different from training: it is latency-sensitive, highly available, and subject to variable demand. Consequently, LLM deployment depends on specialized serving systems that manage model execution, resource allocation, and request scheduling.

LLM inference consists of a prefill phase and a decode phase. In prefill, the system processes the user's prompt in parallel to produce initial hidden states and key-value (KV) cache entries. In decode, the model generates one token at a time, reusing the KV cache to

avoid recomputing attention over the full prompt each step [17, 237]. While prefill is compute-heavy and parallelizable, decode is dominated by sequential steps and memory constraints, particularly due to repeated reads and writes of the KV cache [48, 49]. As a result, serving efficiency is closely tied to KV-cache management and batching strategies [97, 224, 235].

A key technique in LLM serving is dynamic batching, where multiple requests are grouped to improve accelerator utilization [47]. However, naive batching can increase tail latency because requests arrive at different times and may have different generation lengths. Serving systems, therefore, often implement continuous batching, which merges requests at token boundaries, allowing new requests to join a batch mid-generation [224]. This improves throughput while limiting the latency penalty compared to static batching.

Another major concern is memory capacity. The KV cache grows with prompt length, batch size, and model depth, and can become the primary memory consumer during inference. Practical serving stacks address this via strategies such as KV-cache paging [97], prompt caching [59], and quantization [71]. In particular, Quantization reduces memory footprint and can increase throughput, but may introduce accuracy degradation depending on the method and precision used.

At scale, serving systems need robust mechanisms for request routing [54, 135], and multi-tenancy [172]. Requests can be routed based on the selected model, context length, and current hardware availability. To maintain service-level objectives under elevated demand, systems commonly enforce rate limits, prioritize traffic classes, and manage queues with bounded waiting times or explicit timeouts. LLM services may also rely on replication to improve throughput and reliability, and sometimes use model-parallel inference for very large models that cannot fit on a single device [23].

Overall, LLM serving is an optimization problem over competing objectives: maximizing throughput and utilization while meeting latency constraints and controlling memory cost. These trade-offs depend on the interaction between model architecture, decoding strategy, hardware characteristics, and workload distributions, which motivates end-to-end system design rather than optimizing components in isolation.

2.4 Scaling Laws

Scaling laws provide a quantitative way to predict how LLM performance changes with scale, helping guide architecture trade-offs for deployment. They have been widely used to describe

systematic changes in LLM behavior with scale: early work showed that training loss exhibits predictable trends as a function of parameter count, dataset size, and compute [85]. More recent studies have extended and sharpened this framework across training settings and constraints, improving how scaling relationships are estimated, interpreted, and applied in practice [10, 58, 70, 91, 96, 125, 159, 164, 184].

A major milestone is the Chinchilla scaling law, which describes compute-optimal training under a fixed budget by balancing model parameters and training tokens to minimize training loss [70]. Data-Constrained scaling extends this view by modeling the effects of repeated data, which become important when effective data diversity is limited [125]. More recent work studies scaling behavior outside the compute-optimal regime and strengthens the connection between training loss and downstream error, helping translate training-time predictions into task-level outcomes [58].

Recent work has further integrated deployment considerations, including inference cost, as exemplified by Beyond Chinchilla-Optimal [164]. However, unlike training compute and token budgets, which are decided during development, inference demand depends on post-deployment usage. In particular, the number of inference tokens is usually unknown a priori, making inference-aware scaling objectives difficult to optimize reliably and motivating approaches that remain robust under uncertain inference workloads.

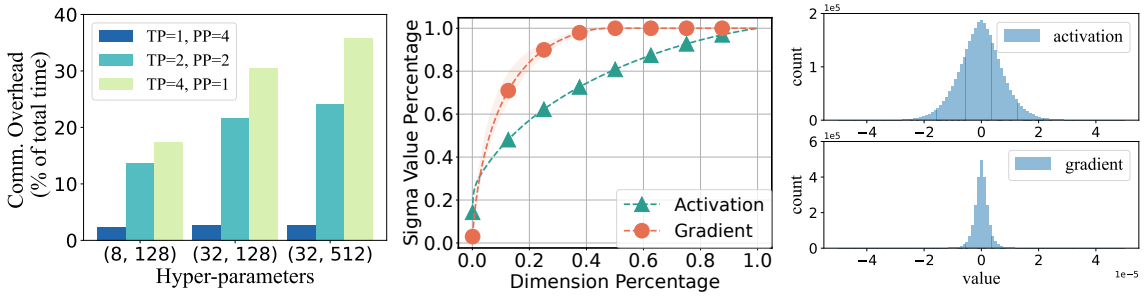
Chapter 3

Architectures for Efficient Training

This chapter proposes a representative benchmarking tool for fairly evaluating model parallelism (MP) compression algorithms in §3.2, with four goals: (1) Allow ML researchers to easily implement new compression algorithms and test it with a wide range of datasets and models; (2) Allow developers to easily compare compression methods in different MP settings; (3) Handle both fine-tuning and pre-training tasks; (4) Enable ML practitioners to ask what-if questions to evaluate if MP compression will be effective at scale. Using this tool as a foundation, we develop novel model architectures to reduce MP communication overhead in §3.2, and evaluate them in §3.3.

3.1 Preliminaries

DP partitions training examples across workers while replicating the full model on each device [69, 107]. In each iteration, every worker computes gradients for all model parameters and synchronizes them across workers, so both the communicated gradient volume and synchronization cost grow with model size. As models scale, gradient exchange can become a communication bottleneck. Numerous prior works in the data-parallel setting have explored gradient compression to reduce the communication costs of training [13, 27, 111, 179, 194, 199]. However, compression in model parallelism (MP) is fundamentally different from that in data parallelism (DP). Firstly, gradients exhibit a low-rank nature, while activations do not, as shown in Figure 3.1b. Consequently, low-rank gradient compression methods, which have been shown to deliver end-to-end speedup in data-parallel training, may not



(a) Communication Analysis **(b)** Low-Rank Analysis **(c)** Distribution

Figure 3.1: Figure (a) illustrates the communication overhead of model parallelism on $\text{BERT}_{\text{LAREG}}$ across 4 GPUs, with varying batch sizes and sequence lengths. The x-axis represents the combination of batch size and sequence length. In Figure (b), curves are plotted based on the ordered singular values from the SVD decomposition, revealing that while the gradient is low-rank, the activation is not. The activation corresponds to the output of the 12^{th} transformer layer in the $\text{BERT}_{\text{LARGE}}$ model. Figure (c) examines the element distribution of activation and gradient for the $\text{BERT}_{\text{LARGE}}$ model.

be directly applicable to MP [194]. Second, gradients and activations have considerably different distributions as shown in Figure 3.1c. Thus, sparsification-based methods may also not be suitable for MP settings [64].

While recent research [200] has shown the promise of quantization-based compression for pipeline parallel training on wide-area networks, it remains unclear which compression algorithms would work well for other parallelism strategies, such as tensor parallelism (TP), or in environments with higher bandwidth. Furthermore, as model sizes [136] and cluster sizes [229] rapidly increase, practitioners need tools that can answer questions like: "Will using quantization lead to better performance compared to Top-K?" or "What will be the throughput benefits if we use a cluster twice as large with NVLink?"

3.2 MCBench Design and Implementation

To address the questions above, we first describe the interface and implementation of compression algorithms in MCBench. Then, we introduce how we build a connection between MCBench and Hugging Face to enable loading models and datasets. Finally, in order to answer what-if questions in terms of scale, we describe an analytical cost model that we develop as a part of MCBench.

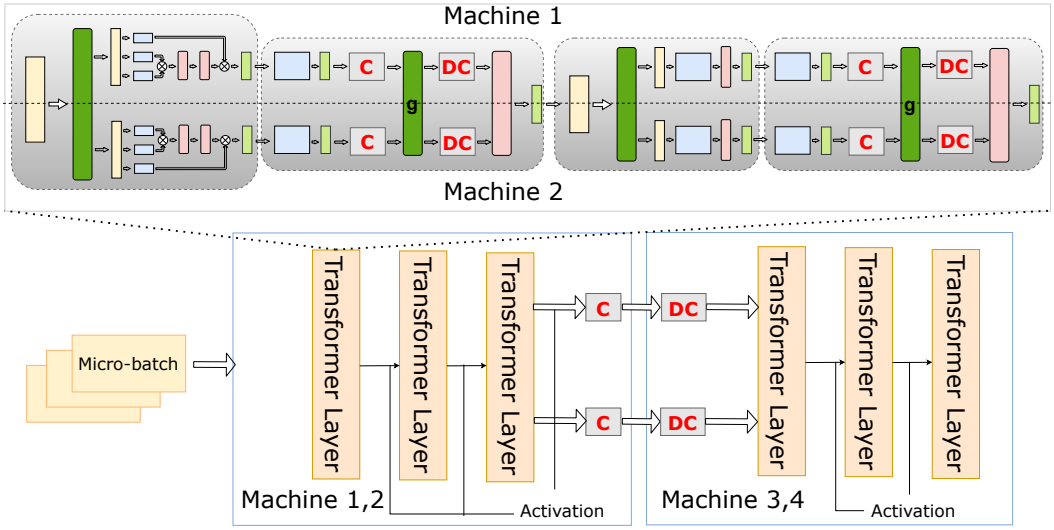


Figure 3.2: Illustration of compression on a 6-Layer Transformer model with 4 machines. Machine 1 and Machine 2 maintain the first three layers according to the TP strategy (pipeline stage 1). g stands for an all-reduce operation in the forward pass. A compression method C is used to reduce the message size for the all-reduce operation to reduce TP communication time. Correspondingly, a de-compression method DC is used after the communication.

3.2.1 Model Parallelism Compression Algorithms

In order to enable ML researchers to design and evaluate new compression algorithms, we develop a new API that integrates with Megatron-LM. Our API design ensures that the algorithms can be used in both tensor parallel and pipeline parallel settings. To ascertain the generality of our API, we implement a range of representative compression methods, including low-rank-based approaches, sparsification-based approaches, learning-based approaches, and quantization-based approaches, as illustrated in Figure 3.2.

Our implementation includes:

- For learning-based approach (AEs), which compress messages using a pair of small neural networks [68], we multiply the activation using a learnable matrix $W_e \in \mathbb{R}^{h \times h_c}$ (the encoder) before the all-reduce step, where $h_c < h$ is the compressed size. After the all-reduce step, another learnable matrix of dimension $W_d \in \mathbb{R}^{h_c \times h}$ (the decoder) is used to decompress the compressed activation. This constitutes architecture-aware compression because the autoencoder module is incorporated directly into the model.

- For the sparsification-based approaches (Top-K and Random-K), we use `torch.topk` function to select the k largest absolute values of the activation and `random.sample` function to randomly select k values from the activation respectively [183].
- For the low-rank-based approach (PowerSGD), we follow the algorithm mentioned in [194] and use `torch.linalg.qr` to do orthogonalize operation.
- Our implementation of the quantization-based approaches is based on the code released by [200]:

$$A^Q = \lceil \frac{A}{\Delta} \rfloor, \quad \Delta = \frac{\max(|A|)}{2^{N_b - 1} - 1}$$

  where A is the activation, A^Q is the quantized activation, Δ is the quantization step size, $\lceil \cdot \rfloor$ is the rounding function, and N_b is the number of bits.

3.2.2 Adding Datasets and Models

Existing code in Megatron-LM (@898a89) only provides three transformer-based models ($\text{BERT}_{\text{LARGE}}$, GPT, and T5). and a few language datasets for fine-tuning tasks. To meet our benckmark's design goal, we build a connection between Megatron-LM and Hugging Face [208] to enable the following features: (1) Load datasets from Hugging Face; (2) Implement the Hugging Face models by using functions provided by Megatron-LM; (3) Convert Hugging Face checkpoints into Megatron-LM format; (4) Split the checkpoints according to MP settings.

We consider image classification with the ViT-Base model as an example to explain the above steps in detail. First, we use the `load_dataset` function from Hugging Face to load CIFAR-10 and CIFAR-100 datasets. Second, we implement the following three parts of transformer-based models: (1) preprocessing; (2) transformer; (3) post-processing. For the preprocessing step, we transform the input into an embedding format that the model can utilize. Subsequently, we employ the `ParallelTransformer` function from Megatron-LM to implement the transformer layers of the model. The `ParallelTransformer` function allows us to train the models under various MP settings. Finally, the post-processing part of ViT-Base is a simple linear layer.

To handle different checkpoint formats, and given that a model's checkpoint can be saved as a dictionary, we transform the Hugging Face checkpoint format to the Megatron-LM format by reordering the keys and values in the dictionary. Then, we split the checkpoints into several files based on MP settings such that they can be loaded into Megatron-LM for fine-tuning tasks.

3.2.3 Analytical Cost Model

While our implementation of MP compression algorithms can be used to measure accuracy and performance with empirical experiments, in MCBench, we also aim to help developers understand how the effects of compression will change as we scale model size and cluster size. In order to minimize the amount of time and resources required to answer scaling questions, we develop a cost model that captures the speedup from the above-named compression methods under various settings.

To develop our cost model, we consider the model parallelism scaling strategy developed in [131]. Concretely, we use tensor model parallelism in the same node, and pipeline model parallelism across nodes. We build a performance model for MP compression for real-world settings similar to [127] in two steps. First, we develop our cost model on a single-node, and analyze how costs change as we scale up the model size on a single node. Second, we increase the cluster size and, according to the model-parallelism strategy we choose, we assign additional GPUs to pipeline parallelism, and use off-the-shelf pipeline parallelism cost models to predict the performance [103, 233].

Denote the vocabulary size as V, hidden size as h, sequence length as s, and batch size as B. From [131], we know that the number of floating points operations (FLOPs) and all-reduce message size in a Transformer layer is $96Bsh^2 + 16Bs^2h + 6BshV$, and Bsh respectively.

Cost Model on Single Node. Without compression, the total time of a Transformer layer can be modeled as a sum of the all-reduce communication step and the computation time step. These two steps can not overlap because the all-reduce communication depends on the computational results:

$$T = T_{comp}(96Bsh^2 + 16Bs^2h + 6BshV) + T_{comm}(Bsh) \tag{3.1}$$

Let X be the compression method and the all-reduce message size after compression be M_c. The total time of a single Transformer layer with model parallelism compression can be written as:

$$T_X = T_{comp}(96Bsh^2 + 16Bs^2h + 6BshV) + T_{comm}(M_c) + T_{overhead} \tag{3.2}$$

where $T_{overhead}$ is the computation time of the compression algorithm. The speedup with L transformer layers on the single node is $\frac{L \times T}{L \times T_X} = T/T_X$.

Scaling Up the Cluster Size. Next, we analyze the speedup when scaling up the cluster size by combining the pipeline parallelism cost model developed in [103, 233]. Formally, the running time is modeled as a sum of per-micro-batch pipeline communication time, per-micro-batch of non-straggler pipeline execution time, and the *per-mini-batch* straggler pipeline execution time. To use the cost model, we denote the number of micro-batches as m, the number of nodes (the cluster size) n, the number of layers L, the pipeline communication time p or p_C.

We use the default pipeline layer assignment strategy in [174], which balances the number of transformer layers. Thus, every pipeline stage requires the same amount of time: $\frac{L}{n}T$ or $\frac{L}{n}T_X$. We use the pipeline communication model in [81, 103], $p = \frac{Bsh}{w}$, $p_X = \frac{M_c}{w}$, where w is the bandwidth. Thus the overall speedup can be written as:

$$\frac{(\frac{m-1}{n} + 1) \times L \times T + (n-1) \times \frac{Bsh}{w}}{(\frac{m-1}{n} + 1) \times L \times T_X + (n-1) \times \frac{M_c}{w}} \tag{3.3}$$

In Section 3.3, we validate the cost model and analyze the implications.

3.3 Performance Analysis With MCBench

We present the first comprehensive study of model parallelism compression by using MCBench to answer the following questions:

- What is the impact of activation compression on system throughput, and which compression method achieves the best throughput?

- What is the impact of compression on model accuracy? How do different downstream tasks affect the performance of compression methods?
- What happens when we scale up the model size and the cluster size?

We answer these questions in the context of two commonly used scenarios: fine-tuning on the GLUE benchmark [195], CIFAR-10, CIFAR-100 [92], and pre-training on the Wikipedia [52] and the BooksCorpus [238] datasets.

3.3.1 Experimental Setup

We first describe the system configuration, evaluated models, and other experimental settings.

System Configuration. In default, we use AWS p3.8xlarge instances where each instance is equipped with 4 V100 GPUs, 10 Gbps interconnect bandwidth, and NVLink in each instance.

Models. We use the $\text{BERT}_{\text{LARGE}}$ model provided by Megatron-LM [174] which has 345M parameters. We configure the model to have 24 layers with each layer having a hidden size of 1024 and 16 attention heads. Moreover, we also integrate $\text{BERT}_{\text{BASE}}$ [53], ViT-Base [55], XLM-RoBERTa-XL [57], and OPT-3B [229] into Megatron-LM [174]. Both $\text{BERT}_{\text{BASE}}$ and ViT-Base have 12 layers with each layer having a hidden size of 768 and 12 attention heads. Additionally, XLM-RoBERTa-XL has 36 layers, each with a hidden size of 2560 and 32 attention heads. Similarly, OPT-3B has 32 layers, each also having a hidden size of 2560 and 32 attention heads. We use fp16 training in all experiments.

Experimental Settings. For fine-tuning, we follow similar settings in previous studies [52, 55, 102, 114]. We use a micro-batch size of 32 and a sequence length of 512 for $\text{BERT}_{\text{LARGE}}$, a micro-batch size of 32 and a sequence length of 128 for $\text{BERT}_{\text{BASE}}$, and a micro-batch size of 512 with a patch size of 32 for ViT-Base. Unless otherwise specified, the model evaluated is $\text{BERT}_{\text{LARGE}}$. We use 4 GPUs (in one machine) for fine-tuning. We vary the tensor model-parallel size and the pipeline model-parallel size across the following three parallelism degrees: $\{(1,4),(2,2),(4,1)\}$, where the first number of the tuple represents the

tensor model-parallel degree and the second number of the tuple stands for the pipeline model-parallel degree. For pre-training, we use 4 machines with 16 GPUs in total. We use the recipe from [78] which uses large batch size with shorter sequence length. We set micro-batch size 128, global batch size 1024, and sequence length 128. To study the impact of the distributed settings, we use the following three different parallelism degrees: $\{(2,8),(4,4),(8,2)\}$.

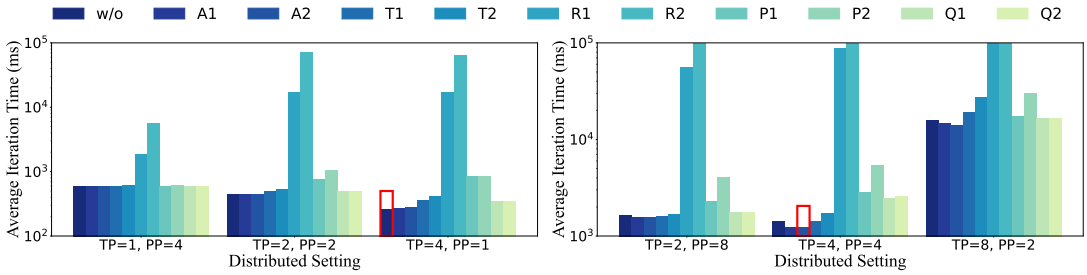
Table 3.1: Notation Table. TP/PP stands for the degree of tensor/pipeline model parallelism. 'comm' and 'comp' are short for 'communication' and 'compression'.

Notation	Description
A1	AE with encoder output dimension 50
A2	AE with encoder output dimension 100
T1	Top-K: same comp. ratio as A1
T2	Top-K: same comp. ratio as A2
R1	Rand-K: same comp. ratio as A1
R2	Rand-K: same comp. ratio as A2
Q1	Quantization: reduce the precision to 2 bits
Q2	Quantization: reduce the precision to 4 bits
P1	PowerSGD: same comp. ratio as A1
P2	PowerSGD: same comp. ratio as A2
TP	Tensor model-parallelism degree
PP	Pipeline model-parallelism degree

Hyperparameters. We also evaluate compression algorithms with different hyper-parameters. For AE, we vary compression dimension between $\{50,100\}$. For Top-K and Random-K algorithms, we keep the same compression *ratio* as AE *(we compress the activation around 10 and 20 times)*. We evaluate quantization with $\{2,4\}$ bits. Due to the instability of PowerSGD under FP16 training, we used FP64 to execute the PowerSGD algorithm.

By default, we perform experiments on $\text{BERT}_{\text{LARGE}}$ model with 24 layers and compress the activation for the last 12 layers. For instance, when the pipeline model-parallel degree is 2 and the tensor model-parallel degree is 2, we compress the activation between two

pipeline stages and the communication cost over tensor parallelism in the last 12 layers (we evaluate the impact of varying the number of compression layers in §3.3.5). Similarly, we compress the activation for the last 6 layers when we perform experiments on $\text{BERT}_{\text{BASE}}$ and ViT-Base. Furthermore, we compress the activations of the final 18 layers of XLM-RoBERTa-XL and the final 16 layers of OPT-3B. Due to the limited space, we only present results from $\text{BERT}_{\text{LARGE}}$, ViT-Base, and XLM-RoBERTa-XL in this section and include other results in the Appendix A.1.



(a) Fine-tuning **(b)** Pre-training

Figure 3.3: Average iteration time (ms) for fine-tuning (left) and pre-training (right) with various compression techniques and distributed setting. For each setting, we repeat experiments for 5 times. Red rectangular boxes highlight the best method. 'w/o' is short for 'with of compression'.

3.3.2 Fine-tuning on Single Node

Takeaway 3.1. *Among all evaluated compression methods, none of the techniques can be used to improve system throughput (by more than 1%) by compressing activations when doing fine-tuning tasks.*

When running fine-tuning experiments on a p3.8xlarge instance on Amazon EC2, we observe that we cannot improve system throughput by using any compression algorithms from Figure 3.3a. We also find that the best configuration for fine-tuning is TP=4, PP=1 without using any evaluated compression methods. This is primarily due to the high bandwidth of NVLink, which means that communication is not a significant bottleneck, and the overhead of compression algorithms, which in turn leads to additional time spent in encoding/decoding.

Table 3.2: Fine-tuning results over GLUE dataset with tensor model-parallel size 2 and pipeline model-parallel size 2. F1 scores are reported for QQP and MRPC, Matthews correlation coefficients are reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	88.07/88.70	92.02	95.07	88.46	62.22	93.39	82.67	89.16	86.64
A1	85.42/85.43	91.07	92.09	86.14	54.18	91.31	70.04	87.61	82.59
A2	85.53/85.65	91.24	93.23	85.86	55.93	91.01	65.34	87.76	82.40
T1	32.05/32.18	74.31	83.60	70.78	0.00	58.37	51.99	0.00	44.81
T2	44.12/45.67	39.68	90.83	78.09	0.00	84.42	49.82	62.70	55.04
P1	83.03/83.43	91.08	90.60	79.07	0.00	89.73	59.21	74.66	72.31
P2	83.46/83.77	91.10	91.86	81.62	51.32	89.71	50.54	85.44	78.76
Q1	87.25/87.81	91.71	93.46	87.01	55.99	61.38	67.51	88.02	80.02
Q2	87.85/88.47	91.93	93.23	87.42	57.67	93.01	78.34	87.43	85.04

Table 3.3: Fine-tuning results over CIFAR 10 and CIFAR 100 on ViT-Base with tensor model-parallel size 2 and pipeline model-parallel size 2.

Compression Algorithm	CIFAR-10	CIFAR-100
w/o	98.81	91.83
A1	97.19	82.76
A2	97.14	83.51
T1	61.77	6.27
T2	65.54	10.74
P1	28.50	47.95
P2	21.32	5.02
Q1	92.96	84.09
Q2	98.85	91.94

Moreover, we see similar results as with other models and our takeaway still holds for the XLM-RoBERTa-XL model. Table 3.4 shows the results from evaluating XLM-RoBERTa-XL over Cloudlab [57] d8545 instances where each instance is equipped with 4 A100 GPUs and NVLink. Note that the hidden dimension for XLM-RoBERTa-XL is 2560, we set the output dimensions of A1 and A2 to 128 and 256, respectively, to maintain the same compression ratio used in prior experiments. Other compression methods maintain the same compression ratio as the autoencoder. In addition, Q1 and Q2 still reduce the precision to 2 bits and 4 bits respectively.

Takeaway 3.2. *For fine-tuning tasks, model parallelism compression can considerably degrade the model's accuracy, especially in the face of complex tasks. Additionally, Transformer-based models employing model parallelism compression can potentially underperform compared to smaller models.*

Given the higher complexity of CIFAR-100 over CIFAR-10 and RTE's status as one of the most challenging tasks in the GLUE benchmarks, an examination of Table 3.2 and Table 3.3 reveals that AE struggles to uphold the fine-tuning accuracy across the RTE and CIFAR-100 tasks. Furthermore, in contrast to the accuracy detailed in [55], ViT-Base featuring AE

compression exhibits inferior performance to Resnet-50 on CIFAR-10 and CIFAR-100 during fine-tuning. Hence, we suggest training smaller models for complex tasks to speed up.

Table 3.4: The average iteration time (ms) for fine-tuning XLM-RoBERTa-XL with various compression techniques by setting TP=2, PP=2. The results are collected from the Cloudlab d8545 machine **with NVLink** by using batch size 16, and sequence length 512. The best setting is **bolded** in the table. And the settings which see benefits compared with the baseline, are underlined.

Distributed Setting	w/o	A1	A2	T1	T2	P1
TP=2, PP=2	549.81	**545.12**	553.34	593.20	613.24	1,077.11
Distributed Setting	w/o	P2	R1	R2	Q1	Q2
TP=2, PP=2	549.81	1,609.01	16,525.53	35,240.28	602.86	609.45

Table 3.5: Fine-tuning results over GLUE dataset by using the checkpoint obtained by pre-training. F1 scores are reported for QQP and MRPC, Matthews correlation coefficient is reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	84.87/84.79	91.25	92.43	86.84	56.36	92.26	70.40	86.83	82.89
A2	83.77/84.32	91.14	91.63	86.55	58.61	91.96	71.48	87.16	82.96
T2	61.06/60.93	80.74	80.16	63.83	10.01	59.55	47.29	0.37	51.55
Q2	84.47/85.32	91.36	93.23	85.10	58.84	91.69	71.84	86.39	83.14

3.3.3 Pre-training on Multiple Nodes

Takeaway 3.3. *Among all evaluated methods, AE is the best compression methods over pre-training. AE not only preserves the model's accuracy but also achieves the highest pre-training throughput.*

For pre-training tasks, from Figure 3.3b, we observe that, by using A2 to compress the activation over the last 12 layers, we can improve throughput for pre-training by 16%.

Table 3.6: We breakdown the average iteration time (ms) for pre-training with various compression techniques when using tensor model-parallel size 4, pipeline model-parallel size 4, micro batch size 128, global batch size 1024, and sequence length 128. The results are collected from 4 AWS p3.8xlarge machines **with NVLink**. The total time (ms) is divided into following parts: forward step, backward step, optimizer, and waiting & pipeline communication. The last three columns further breakdown the tensor encoder/decoder and communication times which are considered part of the forward step.

Compression Algorithm	Forward	Backward	Optimizer	Waiting & Pipeline Comm.	Total Time	Tensor Enc.	Tensor Dec.	Tensor Comm.
w/o	467.73	419.26	7.42	527.99	1,422.40	\	\	91.08
A1	546.95	455.26	7.29	233.47	1,242.97	8.64	16.20	32.76
A2	459.26	467.51	9.64	286.78	1,223.20	12.96	20.52	43.56
T1	813.03	433.42	7.35	156.67	1,410.47	108.00	268.92	115.92
T2	1,068.38	444.26	6.75	202.48	1,721.87	153.36	427.68	151.56
R1	78,906.91	444.88	6.08	3,707.37	83,065.23	73,847.16	279.72	649.44
R2	\	\	\	\	>100,000	\	\	\
P1	2,032.54	605.72	4.01	202.60	2,844.88	169.96	7.53	32.18
P2	4,316.74	712.26	6.81	359.70	5,395.51	405.12	10.20	46.82
Q1	803.63	417.33	8.61	1,205.46	2,435.03	90.72	304.56	193.68
Q2	805.33	417.74	7.55	1,364.32	2,594.94	85.32	271.08	111.60

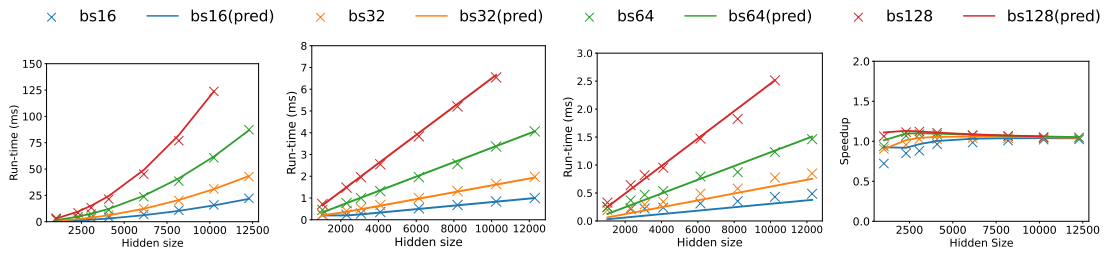
From Table 3.5, compared with the baseline (without compression), we can observe that using AE is able to keep the accuracy when compared to the uncompressed model. In addition, **we observe that we can use the AE at the pre-training phase and remove it during the fine-tuning phase.** In other words, we only need to load the parameter of the $\text{BERT}_{\text{Large}}$ model to do fine-tuning, and the parameters of the AE can be ignored. Furthermore, Table 3.5 shows that pre-trained models suffer significant accuracy loss when using Top-K for compression. Finally, we find that quantization can preserve the model's accuracy, but we cannot achieve end-to-end speedup since the overhead of the quantization method is too large as shown in Table 3.6. In conclusion, it is not a good choice to compress the activation by using quantization or Top-K.

Takeaway 3.4. *Compressing activation for models can improve throughput for multi-node pre-training by 16%. The reason behind this is that we can reduce the communication cost between the two pipeline stages. In addition, the main factor affecting the performance of the compression algorithm is the overhead of the compression methods.*

Table 3.7: The average communication time (ms) per iteration between two pipeline stages. The first column indicates the pipeline stage. And the second column shows the communication time per iteration without compression. Moreover, the third column presents the communication time with A2. We only compress the activation in the last 12 layers and thus the time for the first pipeline stage is unchanged.

Pipeline Stages	Comm. (w/o)	Comm. (A2)
$0 \leftrightarrow 1$	77.82	76.13
$1 \leftrightarrow 2$	88.69	13.19
$2 \leftrightarrow 3$	97.67	14.09

From Table 3.6, we notice that using AE and Top-K can reduce the waiting time and pipeline communication time of pre-training. This is because the inter-node bandwidth (10Gbps) is smaller than the intra-node bandwidth (40GB/s with NVLink), so compression is effective at reducing the communication time between two pipeline stages. From Table 3.7, we can observe that, by using A2 to compress the activation over the last 12 layers, we can reduce the communication cost between the two pipeline stages effectively. Additionally, as seen in Table 3.6, the performance impact of MP compression is primarily due to the overhead of the compression techniques.



(a) T_{comp} **(b)** T_{comm} **(c)** $T_{overhead}$ **(d)** Speedup

Figure 3.4: Cost model validation with different batch size and hidden sizes. From left to right, we show computation time, communication time, overhead by using AE compression, and end-to-end speedup. We use a fixed tensor model-parallel degree 4. 'bs' is short for 'batch size', and 'pred' means the line is predicted by our developed cost model.

3.3.4 Analysis of Scaling Up

Next, we study how the benefits of compression vary as model size and cluster size increase. Previous studies [131] have focused solely on modeling the computational costs of Transformer-based models, overlooking the communication costs and compression overheads associated with model parallelism. To overcome these limitations, we develop the first cost model specifically tailored for hybrid-parallel distributed training that incorporates compression techniques. Our model accounts for both compression overhead and communication costs across nodes, making it more generally applicable for distributed training.

Given the promising results from AE in the previous section, we choose AE as the compression method with our cost model from §3.2.3. We first validate the correctness of our cost model by comparing its prediction to ground truth. The ground truth is real experimental results collected from the AWS platform by using MCBench.

Modeling T_{comp}. We model T_{comp}, the computation time of each Transformer layer, as a linear function of FLOPs with the coefficient α that corresponds to the peak performance of the GPU. In particular, we estimate α using ground truth wall clock time of the largest hidden size we can fit, where the GPU is likely to be of the peak utilization [206]. During experiments, we found that fitting α using time of smaller hidden sizes can result in a *30x* higher prediction time for larger hidden sizes because of low GPU utilization. Our prediction versus the ground truth time is plotted in Figure 3.4a.

Modeling T_{comm}. We model T_{comm}, the communication time of each Transformer layer, as a piece-wise function of the message size [14]. Formally,

$$T_{comm}(Bsh) = \begin{cases} C & \text{if } Bsh < d \\ \beta Bsh & \text{if } Bsh \geqslant d \end{cases}$$

If the message size is smaller than a threshold d, then $T_{comm}(Bsh)$ is a constant C because the worker needs to launch at least one communication round [109]. Otherwise, the number of communication rounds is proportional to the message size. The fitting results are shown in Figure 3.4b.

Modeling $T_{overhead}$. In AE, $T_{overhead}$ is the encoder and decoder computation time. It is a batched matrix multiplication with input dimension $B \times s \times h$ and $h \times h_c$ ($h_c < h$). $T_{overhead} = \gamma Bsh$ since $h \times h_c$ is negligible compared to $B \times s \times h$. The fitting results are shown in Figure 3.4c.

Using the above cost model, we now compute the speedup as we vary the FM size by computing $\frac{T}{T_{AE}}$. Using a fixed encoder dimension h_c for AE (we set h_c to 100), the communication time in Eq. (3.2) can be modeled as $T_{comm}(Bsh_c)$. Compared to the setting without compression, the computation time remains unchanged. In addition, $T_{comm}(Bsh_c)$ is roughly equal to C because Bsh_c is usually smaller than the threshold d.

Since each Transformer layer has identical configurations in popular Transformer models [52, 147], the overall speedup ratio from using compression will remain the same as we vary the number of layers. Thus, we can estimate the speedup of different hidden sizes of any number of Transformer layers using $\frac{T}{T_{AE}}$. We provide the fitting result for this fraction in Figure 3.4d.

To sum up, the fitting results shown in Figure 3.4 indicate that our cost model can predict the performance of MP compression correctly with various batch size and hidden size. Therefore, we use the cost model to estimate speedup for various FM sizes and cluster sizes next.

Understanding the Trend. The asymptotic behavior of large hidden size h based on Eq. (3.1) and (3.2):

$$\frac{T}{T_{AE}} \approx \frac{\alpha \times \text{FLOPs} + \beta Bsh}{\alpha \times \text{FLOPs} + \gamma Bsh + C}$$

where $\text{FLOPs} = 96Bsh^2 + 16Bs^2h + 6BshV$.

For a fixed cluster, as hidden size increases, the benefits from model parallelism compression diminish and has less than 5% benefit when hidden size $\geqslant 12288$. [1]

Scaling Up the Cluster Size. The overall speedup can be obtained by using $M_c = Bsh_c$ and $T_X = T_{AE}$ based on Eq. (3.3). Under the pre-training setup, the cost model predicts an

[1] In our hardware setup, $\alpha \approx 6.33 \times 10^{-13}$, $\beta \approx 3.37 \times 10^{-8}$, $\gamma \approx 1.5 \times 10^{-10}$, $C \approx 0.2$, and we pick $d = 204,800$.

acceleration of 15%, which is in agreement with our experimental results. From Table 3.8, the setting with a hidden size of 6144 is very similar to GPT-2 [148], and we see that we can achieve ~1.09× speedup. When the hidden size is scaled up to 25,600, AE compression can achieve ~1.26× speedup. This shows that if we increase the number of nodes when we increase in the number of layers, AE compression benefits can increase. Moreover, from Table 3.9, model compression with AE attains ~25% per-iteration speedup at scale. And we observe, for a static model, an upward trend in the advantages gained from model parallelism compression when the count of nodes escalates from 8 to 64.

In summary, model parallelism compression has diminishing returns if we only scale up the model on a fixed cluster. To gain benefits, one needs to also **properly manage other parameters in the cost model, *e.g.*, scaling up the number of nodes and using pipeline parallelism**.

Table 3.8: Weak-scaling speedup for the Transformer models. The degree of tensor model parallelism is 4, and the micro-batch size is min{128, batch size/# nodes}. We follow the other hyper-parameters as in Table 1 of [131].

hidden size	# layers	# nodes	batch size	speedup
6144	40	1	1024	1.09×
8192	48	2	1536	1.08×
10240	60	4	1792	1.09×
12288	80	8	2304	1.10×
16384	96	16	2048	1.15×
20480	105	35	2520	1.22×
25600	128	64	3072	1.26×

3.3.5 Varying compression layers and location

Takeaway 3.5. *When the number of compressed layers increases, the model accuracy decreases.*

From Figure 3.5a, we can observe that the accuracy for RTE and the matthews correlation coefficient for CoLA decreases as we increase the number of layers compressed. This is

Table 3.9: Strong-scaling speedup for the Transformer models. The number of tensor model parallelism is 4, and the micro-batch size is min{128, batch size/# nodes}. As for the hidden size, the number of layers, and the batch size, we follow the setting of Table 1 in [131].

hidden size	# layers	# nodes	batch size	speedup
25600	128	8	3072	1.04×
25600	128	16	3072	1.07×
25600	128	32	3072	1.14×
25600	128	64	3072	1.26×

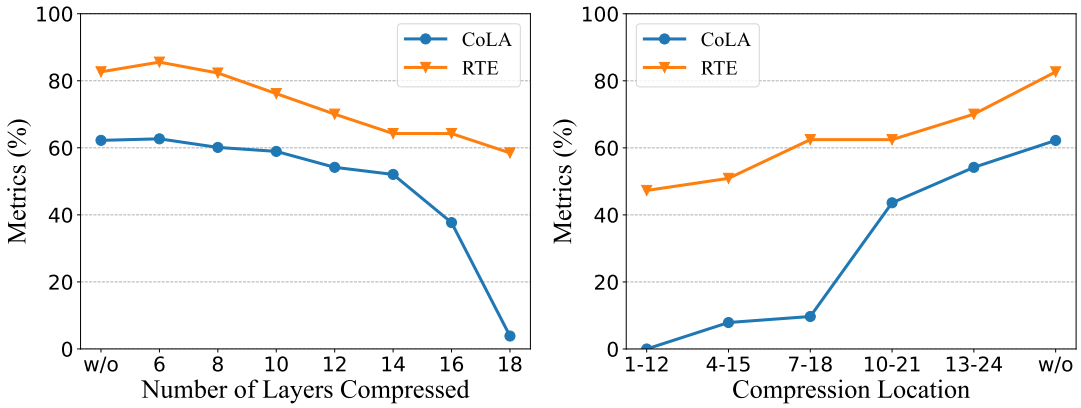
because as we increase number of layers compressed, we lose more information in the activations leading to a loss in accuracy. From Figure 3.5a, we observe that compressing activations of the last 8 layers is the best strategy to keep the accuracy loss within 3% for both datasets.

Takeaway 3.6. *Compressing the activation for the initial layers harms the accuracy of the model.*

We keep the number of layers compressed constant and vary the location where we apply compression (Figure 3.5b). The results indicate that compressing activations of the first few layers of the model significantly harms the model's accuracy. This is because compressing activations generates error and the error in the early layers can be accumulated and propagated to later layers.

3.3.6 Limitations

Our experimental study, due to resource constraints, does not encompass some of the recent advances in FMs pre-training, such as LLaMA [190]. Nevertheless, our cost model can offer accurate guidance on performance at these scales. Additionally, our study does not account for the *error feedback* (EF) schemes, which are frequently incorporated in data-parallel compression [168, 183]. While EF might help improve accuracy, we note that using EF will increase the overhead of compression methods.



(a) Vary Number of Layers Compressed **(b)** Vary Compression Location

Figure 3.5: Fine-tuning results over CoLA and RTE datasets by varying the compression location and number of layers compressed. The above figure shows that model performance vs the number of layers compressed. The below figure shows that model performance versus the compression location. We use tensor model-parallel degree 2, pipeline model-parallel degree 2, batch size 32, and sequence length 512.

3.4 Conclusion

In this work, we studied the impact of compressing activations for models trained using model parallelism. We first developed a general performance model for model parallelism compression. Next, we implemented and integrated several popular compression algorithms into an existing distributed training framework (Megatron-LM) and evaluated their performance in terms of throughput and accuracy under various settings. Our results show that learning-based compression algorithms are the most effective approach for compressing activations in model parallelism. Based on the experimental results, we evaluate the correctness of our performance model and analyze the speedup when scaling up the model. Our experiments provide valuable insights for the development of compression algorithms in the future.

Chapter 4

Architectures for Scheduling Efficiency

This chapter introduces an architecture-aware scheduler that jointly optimizes parallelism strategy and GPU co-location. We first provide background on deep learning cluster scheduling in §4.1. We then present an overview of Tesserae in §4.2 and describe the detailed algorithm in §4.3. Next, we discuss implementation details and explain how Tesserae is realized as a placement-policy plugin in §4.4. Finally, we evaluate Tesserae on a real cluster and in simulation using production workloads in §4.5.

4.1 Preliminaries

We begin by providing an overview of deep learning scheduling in GPU clusters in §4.1.1. Next, we highlight how scheduling policies are augmented by placement constraints to improve cluster utilization in §4.1.2. Finally, we list the challenges with the way existing placement constraints have been defined in §4.1.3.

4.1.1 DL Scheduling

A multitude of studies [60, 73, 117, 212, 232, 236] have developed schedulers for DL workloads, targeting diverse optimization goals. In DL scheduling, schedulers assign priorities to a set of jobs with the goal of optimizing a specific performance metric. For example, Gandiva [212] focuses on maximizing cluster utilization, whereas Tiresias [60] is designed to reduce average job completion time. Similarly, Pollux [145] aims to optimize goodput—a metric combining system throughput and statistical efficiency.

To generalize a broad range of existing scheduling policies, Gavel [130] introduces a linear programming framework that unifies optimization objectives, placement strategies, and packing policies within a single framework. Gavel computes a priority score for each job based on the exact solution to an optimization problem and the number of rounds of GPU allocation the job has received.

4.1.2 Placement Policies.

In addition to just scheduling, several schedulers [60, 117, 212] have highlighted the importance of placement constraints on the throughput achieved by jobs. The placement constraints determine which GPUs in the cluster are used to run the chosen jobs. We observe that placement constraints are handled primarily using two approaches:

Heuristic based. Gandiva [212] depicted that distributed jobs prefer consolidated placements, *i.e.*, jobs requiring more than one GPU prefer that both GPUs exist on the same machine. Subsequently, Tiresias [60] highlighted that certain jobs are more placement-sensitive than others, and cluster utilization can be improved by considering the properties of the individual job for performing placement. Furthermore, to mitigate low GPU utilization observed in production clusters [80, 205], several schedulers [73, 130, 212] adopt GPU sharing techniques to better utilize available resources. When sharing GPUs, multiple jobs are concurrently run on the same GPUs.

Optimization based. Gavel [130] demonstrates that specific placement constraints can be formulated as part of an optimization problem. Specifically, Gavel [130] introduces packing multiple GPU jobs as a throughput aware optimization problem.

However, both approaches— heuristic based algorithms and optimization based methods for handling placement constraints—pose certain challenges. In the following section, we discuss these challenges in detail.

4.1.3 Challenges

We highlight the challenges associated with different type of placement constraints.

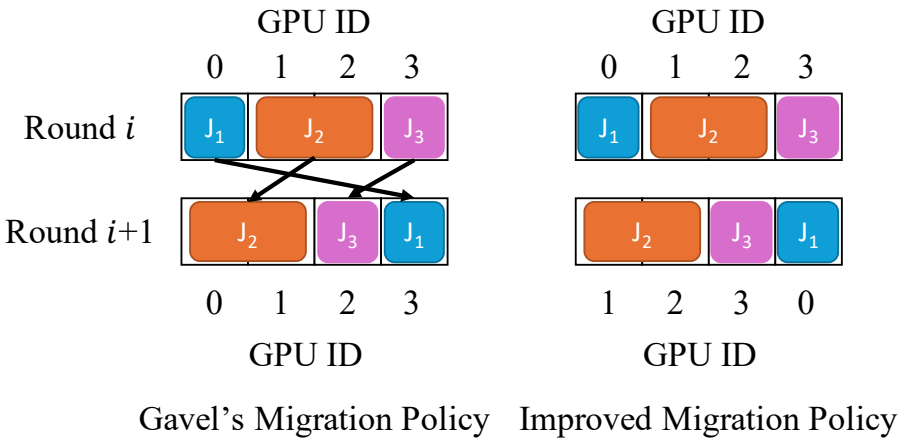


Figure 4.1: Performance Limitations: In the left figure, Gavel's policy migrates three jobs between two nearby plans. However, the right figure shows that we can avoid this overhead and improve throughput by remapping GPU ID.

Performance Limitations We observe that heuristic-based placement approach often fail to maximize the possible throughput. For example, Gavel's migration policy states that job migration is unnecessary if a job uses the same GPU in two consecutive placement rounds; otherwise, migration is required. However, in Figure 4.1, we observe that Gavel's migration policy [130] results in three job migrations, whereas the optimal solution requires none.

Poor Adaptability The deep learning ecosystem is rapidly evolving across the hardware stack, software stack, and workloads. On the hardware side, GPUs have undergone significant advancements, resulting in varying compute characteristics. For instance, newer GPUs support Multi-Instance GPU (MiG) [120], enabling fine-grained resource sharing. Meanwhile the software stack has also evolved, previously distributed jobs used to use a parameter server setup to exchange parameters [108]. Lately, the jobs have been using decentralized collective communication call like all-reduce [169]. These software changes have led older heuristics obsolete and require users to come up with new heuristics, requiring constant manual intervention. Furthermore, as workloads transition between MLPs [15, 132], CNNs [93], transformers [192], and MoEs [171], users need to develop heuristics that consider their distinct compute and communication demands.

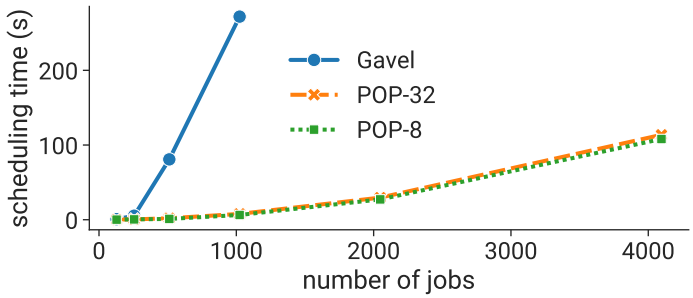


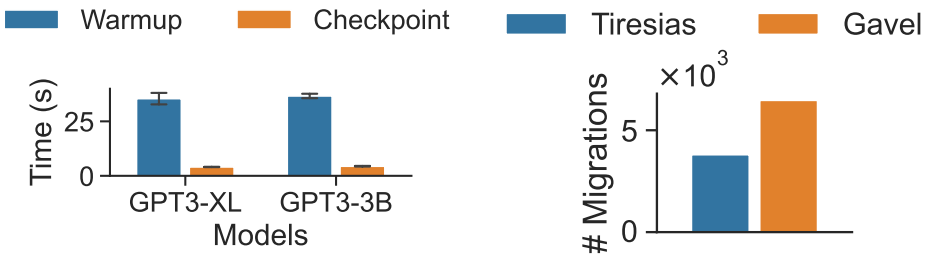
Figure 4.2: Overhead of Schedulers: Decision-making time of each scheduler under varying numbers of active jobs in a 256-GPU cluster. The workload consists of jobs running ResNet-50, VGG-19, DCGAN, and PointNet, each with varying GPU requirements. The results indicate that both Gavel and POP exhibit limited scalability as the number of active jobs increases.

Limited Scalability. Gavel's scalability is limited by the computational overhead of solving linear programs [45]. To overcome this limitation, POP [128] is proposed as a scalable alternative to Gavel [130]. However, as the number of GPUs and active jobs grows [72], scheduler efficiency becomes a significant concern. As shown in Figure 4.2, we fix the cluster size and vary the number of active jobs to evaluate the decision-making time of each scheduler. We observe that POP also faces challenges in scaling efficiently with an increasing number of active jobs—a limitation similarly observed in [95]. As a result, designing a truly scalable policy remains an open research challenge.

4.1.4 Goals

These challenges highlight the need for a framework that enables users to specify placement constraints. In the following paragraphs, we describe the desirable characteristics of such a framework.

Improved Performance. A primary objective of the placement framework should be to enhance hardware utilization. Additionally, it should outperform widely adopted heuristics (*e.g.*, , Tiresias [60]) and match the performance of linear programming-based solvers (*e.g.*, , Gavel [130]).



(a) Warmup and Checkpoint Time **(b)** Number of Job Migrations

Figure 4.3: Migration Overhead: The warmup time is the duration from entering the command to the start of the first iteration. Additionally, the checkpoint overhead represents the total time spent on loading and saving the checkpoint. Further, we evaluate the number of job migrations of Tiresias and Gavel.

Adaptability. The framework should be adaptable to the constant changes in the deep learning stack. It should automatically adjust placement constraints as the underlying hardware [133, 134] and job types evolve [72], without requiring manual intervention.

Compatibility The framework should also be compatible with existing scheduling policies, including Tiresias [60], and Themis [117]. Users should be able to apply their chosen scheduler and use the framework solely for placement decisions, enabling wide adoption.

Scalability As deep learning clusters grow in size and handle an increasing number of jobs [72, 205], the proposed framework demonstrates the ability to efficiently manage large-scale cluster scheduling.

4.2 Tesserae

To tackle the challenges outlined in §4.1, we present Tesserae. We first present an overview of our approach which involves separating scheduling policies from placement policies in §4.2.1. Following that, in §4.2.2, we present an overview of how scheduling in Tesserae works.

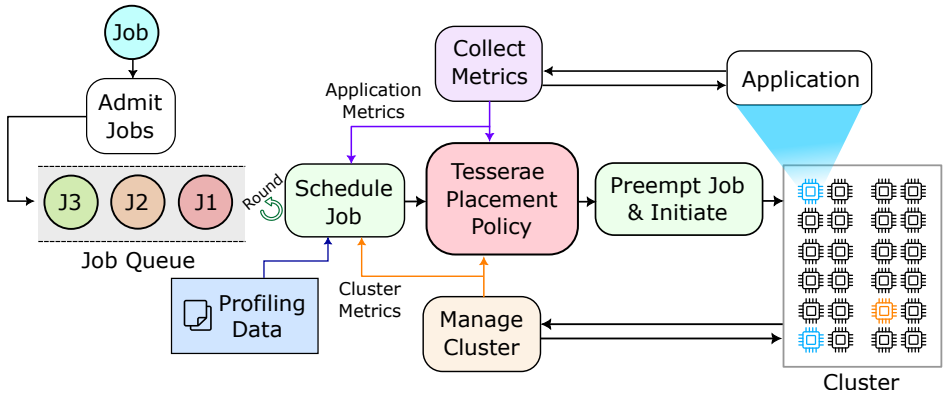


Figure 4.4: Overview of Tesserae system architecture: all components of the system and their interactions. We design Tesserae as the placement policy for the whole system.

4.2.1 Decomposing Scheduling and Placement

Existing DL schedulers typically take one of two approaches. First, the approach is the one taken by linear programming-based frameworks (e.g., Gavel [130]). These frameworks formulate the scheduling and placement policy as a single optimization problem. Formulating such an optimization problem enables capturing multiple objectives, such as minimizing job completion time (JCT) and improving GPU utilization through job packing. This approach is effective in making high-quality decisions, but it limits scalability due to high computational complexity. The second approach is to treat scheduling and placement policies as distinct modules [12, 60, 141, 145, 212].

In this work, we use the latter approach and view DL schedulers as a composition of different policies, such as a scheduling policy, consolidation policy, packing policy, etc. The benefit of this approach is that each policy can be independently designed and then composed together to build a scheduler (Figure 4.4). This approach aligns well with our design goals. First, the independent design of each policy ensures adaptability to changes in the deep learning stack. Second, it supports compatibility, allowing different scheduling objectives to be integrated with various placement policies. Moreover, if each policy is efficient and scalable, the overall scheduler inherits these properties. Our evaluation (§4.5) also shows that our approach of treating scheduling and placement as disjoint policies does not adversely impact the quality of scheduling.

4.2.2 Overview

Figure 4.4 shows a high-level design overview of Tesserae. We next discuss a concrete example of how the Tesserae scheduler can apply policies such as migration-awareness and packing (Listing 5).

Scheduling in DL clusters typically happens in rounds [123, 130] (rounds are typically a few minutes). At the end of each round, the cluster scheduler is invoked to determine the set of jobs that should run for the next round. In every round, the job scheduling policy sorts active jobs by their priority. The priority for jobs is determined based on the underlying scheduling policy, such as arrival time for FIFO or LAS for Tiresias [60], etc. Given the list of sorted jobs, a placement policy first places as many jobs as possible on the GPU cluster without packing, as depicted in Figure 4.5. This is because we would like to first place as many high-priority jobs on the cluster as we can to ensure scheduling priorities are met (lines 5-12 of Listing 5). However, we note that placement (in line 8) can fail if there are fewer available GPUs than the number needed by the jobs or some placement constraint that cannot be satisfied, *e.g.*, a multi-GPU job cannot find a consolidated placement. After following these steps, given the significant migration overheads, we use our novel migration algorithm to minimizes the number of migrations between successive rounds. Additionally, if GPU sharing is enabled, we use a packing policy to determine which jobs from pending_jobs should be packed with already placed_jobs on the GPU cluster, before determining the job migration strategy.

Next, we present the design of an efficient migration and packing policy that can operate alongside any scheduling policy.

4.3 Efficient Migration and Packing Policies

We next describe new placement policies that can be integrated with Tesserae's design described before. Our scalable and performant placement policies are based on the insight that many placement problems can be viewed as instances of graph matching problems. We first present a migration algorithm (§4.3.1) to reduce the number of migrations and then introduce a packing policy (§4.3.2) which can maximize throughput. Finally, we present approaches for minimizing profiling overhead and discuss the properties of Tesserae framework in §4.3.3.

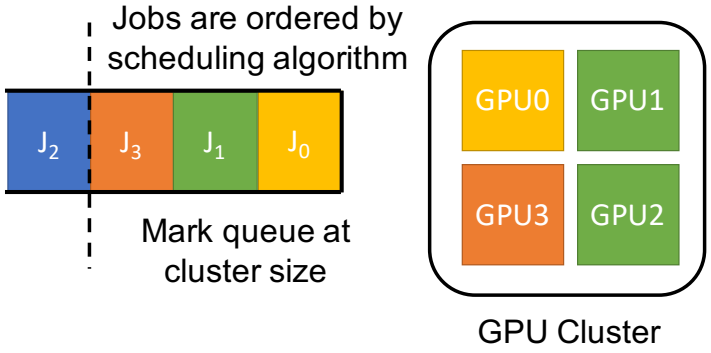


Figure 4.5: Allocation without Packing: This example demonstrates how to allocate as many jobs as possible to a GPU cluster without GPU sharing.

4.3.1 Minimizing Migrations

We first introduce a novel and efficient migration algorithm designed to decrease the number of job migrations between consecutive rounds. To begin with, we define job migration below:

Definition 4.1 (Job Migration). *A job is migrated between consecutive rounds if it is present in both rounds and utilizes different sets of GPUs within the cluster.*

Note that if a job does not appear in both round i and round $i+1$, it is not considered to have been migrated. Next, we present the problem we study in this section:

Definition 4.2 (Job Migration Minimization). *Given two placement plans, P_i from round i and P_{i+1} from round $i+1$, the job migration minimization problem aims to determine a migration strategy that reduces the number of job migrations between these rounds while still meeting the constraints of consolidated placement.*

We design a migration algorithm based on the following observation. Let the placement plans at round i and round $i+1$ be $\{(0,1),(1,2),(2,2),(3,4)\}$ and $\{(0,4),(1,1),(2,2),(3,2)\}$, respectively. In each tuple, the first number represents the GPU ID and the second number represents the job ID. We assume GPUs in the cluster are homogeneous. Although the two placement plans differ, job migrations are unnecessary as we can simply rename the GPU IDs through the following reassignments: $0 \rightarrow 1$, $1 \rightarrow 3$, and $3 \rightarrow 0$. Therefore, it is unnecessary to migrate any jobs. In view of this, we draw inspiration from established

Listing 1: TESSERAE Framework

```
placed_jobs ← [] and pending_jobs ← []
active_jobs ← all submitted and unfinished jobs
Sort active_jobs based on priority
num_gpus_remain ← number of total GPUs
while num_gpus_remain > 0 do
    j ← active jobs with highest priority
    Remove job j from active_jobs
    if we fail to place job j then
        pending_jobs.append(job j)
        continue
    end
    num_gpus_remain -= GPUs required by job j
    placed_jobs.append(job j)
end
if GPU sharing is enabled then
    M ← Packing(placed_jobs, pending_jobs)
    Packing the jobs in pending_jobs with jobs in placed_jobs based on M
end
Determine job migration strategy and place jobs across GPUs in the cluster
```

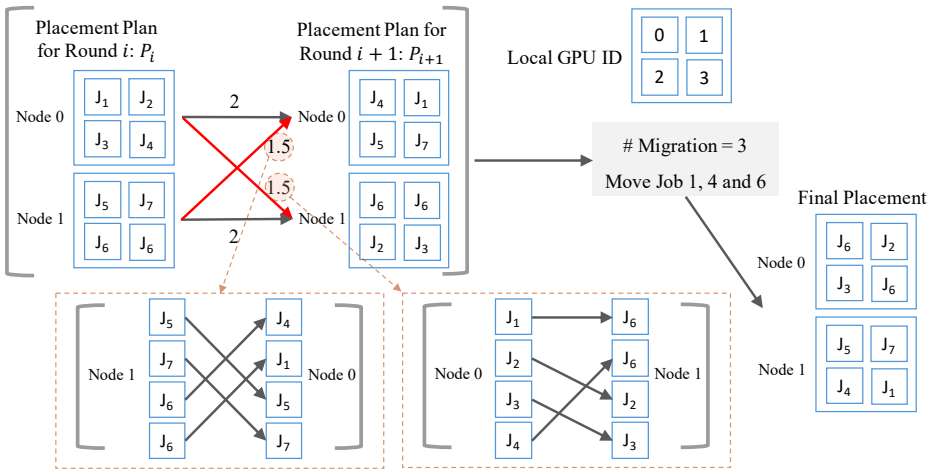


Figure 4.6: An Example of Migration Method: Given two placement plans P_i and P_{i+1} from consecutive round i and $i+1$, we show how to use Algorithm 2 and 3 to compute the migration plan and get the final placement plan in the end.

assignment problems and show our migration algorithm in Algorithm 2. In particular, the algorithm first removes any jobs that are not present in both rounds concurrently (line 2

Algorithm 2: Job Migration

Input: placement_plan for round i: P_i, placement_plan for round $i+1$: P_{i+1}.
Output: Migration Plan M

1 $C \leftarrow []$, $M \leftarrow []$
2 Remove all jobs j in P_i and P_{i+1}, where job j satisfies $j \in ((P_i \cup P_{i+1}) - (P_i \cap P_{i+1}))$
3 **foreach** placement plan of node k, $P_{i,k} \in P_i$ **do**
4 **foreach** placement plan of node ℓ, $P_{i+1,\ell} \in P_{i+1}$ **do**
5 $C_{k,\ell}, M_{k,\ell} \leftarrow$ Node-Level Matching($P_{i,k}$, $P_{i+1,\ell}$)
6 **end**
7 **end**
8 match_solution $\leftarrow$ Hungarian Algorithm(C)
9 **return** M[match_solution]

Algorithm 3: Node-Level Matching

Input: placement_plan for round i of node k: $P_{i,k}$, placement_plan for round $i+1$ of node ℓ: $P_{i+1,\ell}$.
Output: Migration Cost $C_{k,\ell}$, Migration Plan $M_{k,\ell}$

1 $C_{sum} \leftarrow 0$, $C \leftarrow [0]_{k_\ell \times k_\ell}$, where k_ℓ is the number of GPUs in the node.
2 **foreach** GPU $u \in P_{i,k}$ **do**
3 **foreach** GPU $v \in P_{i+1,\ell}$ **do**
4 $JS_u \leftarrow$ Job sets on GPU u and $JS_v \leftarrow$ Job sets on GPU v
5 **foreach** job $j \in JS_u \cup JS_v$ **do**
6 **if** job $j \in ((JS_u \cup JS_v) - (JS_u \cap JS_v))$ **then**
7 $C_{u,v} \leftarrow C_{u,v} + 1/(2 \cdot \text{num\_GPUs}(j))$
8 **end**
9 **end**
10 **end**
11 **end**
12 $C_{sum}, M_{k,\ell} \leftarrow$ Hungarian Algorithm(C)
13 **return** $C_{sum}, M_{k,\ell}$

of Algorithm 2). Next, the algorithm computes the migration cost for every pair of nodes, where each pair consists of one node k from round i and one node ℓ from round $i+1$ (lines 3-5 of Algorithm 2). We use Algorithm 3 to compute the migration cost between node pairs. Specifically, we first calculate the migration cost for each GPU pair across two given nodes (lines 2–7 of Algorithm 3). The cost is determined by the number of GPUs required by each job, as the migration cost is amortized across all processes in a multi-GPU job. Moreover,

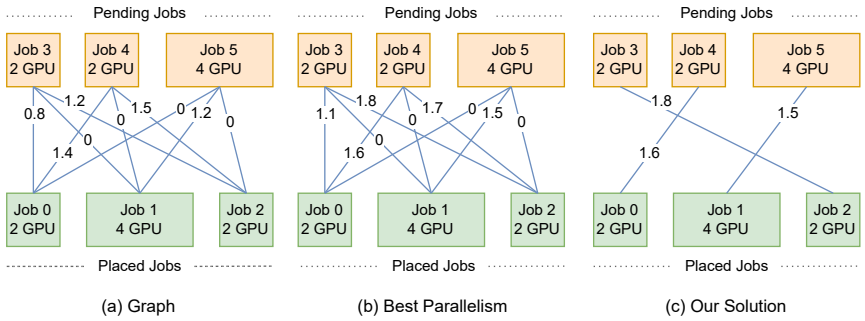


Figure 4.7: An Example of Job Packing: Packing plans are developed by formulating them as weighted bipartite graph matching problems, where the weight of each edge represents the combined throughput of two jobs. We show the matching results obtained from our designed strategy.

each move-in or move-out operation incurs a cost of 0.5 per job, which is why we multiply the cost by $1/2$ in line 7 of Algorithm 3. Finally, we apply the Hungarian algorithm [94] to determine the optimal GPU-level migration between node pairs.

After determining the migration cost for every pair of nodes, we can then apply the Hungarian algorithm [94] again to derive a migration plan that minimizes the total number of migrations between the two placement plans (lines 6-7 of Algorithm 2). We illustrate the complete process using the following example.

Example 4.3. *We illustrate our migration strategy with an example in Figure 4.6. Specifically, lines 3–5 of Algorithm 2 are used to compute the migration cost between each node pair. For instance, the migration cost between node 0 in round* i *and node 1 in round* $i+1$ *is 1.5, as jobs 1 and 4 must be moved out and job 6 must be moved in. Each move-in or move-out operation incurs a cost of 0.5 per job. The corresponding migration plan between node 0 and node 1 is also shown in Figure 4.6. Finally, using line 6 of Algorithm 2, we determine the total number of migrations to be 3, along with the associated migration plan.*

Running Time. We assume that the cluster has k_c nodes and each node contains k_ℓ GPUs. The Hungarian algorithm [94] is widely used to address the assignment problem, and the time complexity of the Hungarian algorithm is $O(n^3)$, where n is the number of tasks in the assignment problems. Therefore, the time complexity of Algorithm 3 is $O(k_\ell^3)$ and the time complexity of Algorithm 2 is $O(k_c^2 k_\ell^3 + k_c^3)$.

Algorithm 4: Packing

Input: placed_jobs, pending_jobs, profile_data
Output: Matching Results M

```
G ← an empty graph
foreach job j ∈ placed_jobs do
    G.addNode(j)
end
foreach job pj ∈ pending_jobs do
    G.addNode(pj)
    foreach job j ∈ placed_jobs do
        if j and pj require the same number of GPUs then
            w ← the sum of throughput of pj and j from profile_data
            G.addEdge(pj, j, w)
        end
    end
end
M ← computing maximum weighted bipartite graph matching with Hungarian
  Algorithm
return M
```

4.3.2 Packing Jobs Efficiently

We next introduce a novel packing algorithm which uses a graph-based formulation to scale to a large number of jobs.

Profiling. Our initial step in achieving optimized packing results is to profile and estimate the throughput of job packing. To normalize the throughput, we divide the packed throughput of jobs by their isolated throughput. For example, without packing, consider the throughput of PointNet to be 50 iterations per second, while GPT3-3B has a throughput of 2 iterations per second. With packing, say the throughput of PointNet drops to 15 iterations per second, while GPT3-3B drops to 1 iteration per second. Consequently, the normalized throughput of PointNet is 0.3 and GPT3-3B is 0.5. The combined or sum throughput in this case is 0.8. Our definition of normalized throughputs is similar to that used in prior work [130] and in the following section, we will use normalized throughputs to formulate our graph-based problem.

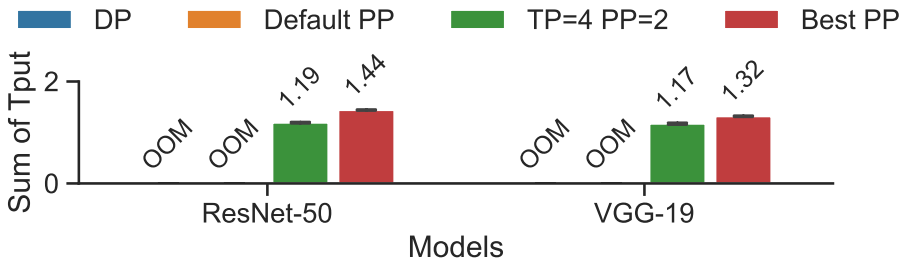


Figure 4.8: Throughput with packing: We evaluate performance of training language models under different parallelization strategies with packing on 8 A100 GPUs. The Default PP is provided by Megatron-LM [174] and the Best PP is picked from the candidate of possible PP strategies. The throughput is normalized by the best performance achieved in isolation. We observe that packing under certain scenarios can improve total throughput from the cluster

Graph-based Problem Formulation. We first reiterate the goal of the packing problem: given a list of placed_jobs and pending_jobs, we wish to pack jobs such that we can maximize the total cluster throughput.

We observe that the job packing problem can be converted to maximum weighted bipartite graph matching problem. To be specific, we build a graph $G = (V_1, V_2, E)$, where $v_1 \in V_1$ represents a job in placed_jobs, $v_2 \in V_2$ stands for a job in pending_jobs, and $e = (u, v) \in E$ indicates that we can pack job u and job v on the same set of GPUs. In other words, job u and job v require the same number of GPUs. The weight w_e of edge $e = (u, v)$ equals the combined throughput of job u and job v, as determined by profiling the combined throughput of job u and job v. Figure 4.7(a) illustrates a graph constructed using profiling data.

Parallelism Strategy. Here, we consider a new dimension introduced by 3D parallel training jobs: that of changing the parallelism strategy and studying how that affects packing. Prior work [81, 201, 233] has studied the problem of choosing the best parallelization strategy to improve the throughput of a given LLM training job. We consider how this degree of freedom in choosing parallelization strategies can affect packing. We conduct a benchmarking study where we use model GPT3-3B and consider an 8 GPU setup. Figure 4.8 shows the results of the above study. We observe that selecting an appropriate parallelism strategy not only prevents out-of-memory (OOM) issues-such as those observed when

packing VGG-19 using the default pipeline-parallel (PP) strategy but also significantly boosts overall throughput. For instance, by adopting the parallelism strategy PP $= (3,3,3,4,4,5,5,5)$, which specifies the number of layers on each GPU, the sum of normalized throughput for ResNet-50 packed with GPT3-3B increases from 1.19 to 1.44.

We can easily integrate the determination of the parallelization strategy into the previously described graph-based problem. If job u is packed with job v, we have to modify the edge weight w_e of edge $e = (u,v)$ in the constructed bipartite graph G, when optimizing the parallelism strategy of job u. In Figure 4.7(b), we demonstrate the enhancement of an edge's weight through the choice of the best packing parallelism strategy. For example, selecting the best parallelism strategy for Job 1 can enhance the weight $w_{e'}$ of the edge between job 1 and job 5 from 1.2 to 1.5.

Solving Graph-based Problems. For each scheduling round, given n active jobs and the number of GPUs in the cluster, the scheduling policy, which determines the priority of each job, creates placed_jobs and pending_jobs. Therefore, V_1 and V_2 in G are fixed with each edge's weight established based on offline profiling data. To solve such a bipartite graph matching problem, we use the classic Hungarian Algorithm [94]. The time complexity of the Hungarian Algorithm is $O(n^3)$, where n is the number of nodes in the graph. Solving the weighted bipartite graph matching problem yields a solution with each placed_job is matched with at most one pending_job and the algorithm ensures that overall sum of the weights of the chosen edges is maximized in the solution, thus yielding the maximum combined throughput from packing. Algorithm 4 presents our packing algorithm, and its resulting solution for the example in Figure 4.7(b) is illustrated in Figure 4.7(c).

Furthermore, in Figure 4.2, we compare the overhead associated with Tesserae against Gavel [130] and POP [45]. We can see that the use of the Hungarian algorithm for packing decisions marginally increases the overhead already present in existing scheduling algorithms. We also see that Tesserae is more efficient than Gavel because Tesserae involves fewer variables. Although POP [128] is designed to speed up the Gavel solver, it remains less efficient compared to Tesserae. Notably, even with 3000 active jobs, Tesserae is capable of making placement decisions within 1 second, which means that Tesserae can be deployed in large-scale cluster management systems.

4.3.3 Discussion

Minimizing Profiling Cost. Although incorporating parallelism strategies into the maximum weighted bipartite graph matching problem is simple, profiling all such strategies offline is impractical. To reduce the profiling cost, we use the following two strategies:

For models trained by using only data parallelism, e.g. ResNet-50, VGG-19, we build the estimation model based on the following assumption: If the model and GPU type are the same, the throughput of the 2-GPU job is double that of the 1-GPU job [79]. Therefore, for Job J, we first profile it on a single GPU to determine its throughput, denoted as tput_J. Then, we use the mathematical model to predict the throughput of Job J over N GPUs: $\text{tput}_J(N) = N \times \text{tput}_J$.

Furthermore, if Job J is packed with Job K, then we first profile them on a single GPU to determine their packed throughput, denoted as $\widehat{\text{tput}}_J$ and $\widehat{\text{tput}}_K$ respectively. Next, we estimate the throughput of Job J and Job K packed over N GPUs as $\widehat{\text{tput}}_J(N) = N \times \widehat{\text{tput}}_J$ and $\widehat{\text{tput}}_K(N) = N \times \widehat{\text{tput}}_K$.

Figure 4.8 shows that when using an identical number of GPUs, selecting an optimal parallelism strategy can significantly increase the throughput of large language models. Consequently, the linear model, which consistently yields the same throughput for a given number of GPUs, is inadequate for models trained using 3D parallelism. To reduce profiling costs, we first profile large language models with randomly generated strategies. We then use Bayesian Optimization [177] to iteratively profile the model with subsequent parallelism strategies until the profiling budget is exhausted. The above strategy is similar to parameter tuning approaches developed in prior works [84, 191]. We evaluate the effectiveness of our profiling optimizations in §4.6.

Extensibility. Tesserae is compatible with numerous established scheduling policies. For policies that do not account for packing, it suffices to modify the sorting priority at line 3 of Algorithm 5. Moreover, Tesserae is equivalent to heuristic policies if we do not execute Packing function in Algorithm 5 since we use the same strategy as heuristic policies to order the active jobs. Additionally, we can also use Gavel to compute priority score to order the active jobs without considering GPU sharing. We show how Tesserae can work with LAS-based schedulers (e.g., Tiresias [60]) and fairness-based schedulers (e.g., Themis [117]) in Section 4.5.

Fairness. Compared with isolated execution, the packing policy increases the total throughput of packed jobs, but it could reduce the throughput of each job. For jobs with high priority or strict deadlines, we can bypass the packing process by not creating edges between such a job and others when formulating the graph in Algorithm 4.

Consolidated Placement. If the jobs from placement plan P_i and P_{i+1} in rounds i and $i+1$ respectively are consolidated, Algorithm 2 will ensure that all jobs remain in a consolidated setting. Because Algorithm 3 performs GPU matching at the node level, it ensures that the processes of a distributed job either remain on the same node or are collectively relocated to other nodes.

4.4 Implementation

Tesserae is built on Blox [12] using Python in approximately 3000 lines of code for both real cluster mode and simulation mode. We have incorporated Tesserae into the Blox framework by developing it as a placement policy. We use Scipy [193] to generate the migration plan for each round outlined in §4.3.1 and solve the weighted bipartite graph matching problem mentioned in §4.3.2. Moreover, we employ gRPC [4] for message communication between schedulers and applications. Lastly, we use CUDA-MPS [124] to run multiple jobs on the same GPU.

Profiling. In this paper, we adopt a simple approach to collect profiling data by running all job combinations offline. Tesserae accumulates profiling data in an offline mode. It operates language models under various parallelism strategies, running each model listed in Table 4.1 and each possible model combination for three minutes to measure their respective throughput. However, collecting all profiling data offline is impractical in real applications. We use strategies mentioned in §4.3.3 to reduce profiling costs and we evaluate these strategies in §4.6.

Schedulers. Tesserae, like prior round-based schedulers [130, 213], make scheduling decisions every six minutes, taking the following steps: For all active jobs, it develops packing and placement plans based on the offline profiling data and the given scheduling method. The scheduler, after formulating the packing and placement plans, will notify all

Table 4.1: Models used in the evaluation. ♣: Image Classification, ♢: Image-to-Image Translation, ♡: 3D Point Cloud Classification, ♠: Language Modeling.

Model	Task	Dataset	Batch Size
ResNet-50 [65]	♣	ImageNet [51]	32-256
VGG-19 [175]	♣	ImageNet [51]	16-128
DGCAN [146]	♢	LSUN [223]	128-1024
PointNet [143]	♡	ShapeNet [36]	32-256
GPT3-Medium [35]	♠	Wikipedia [52]	512
GPT3-XL [35]	♠	Wikipedia [52]	512
GPT3-3B [35]	♠	Wikipedia [52]	512

nodes to stop current jobs and start new ones. Note that Tesserae only preempts the job after the job finishes the current iteration. Then, the scheduler pauses for six minutes to gather updated metrics, such as service attained and training progress. Following that, it makes decisions for the next round.

Applications. All applications are implemented using PyTorch [140]. For all models listed in Table 4.1, Tesserae classifies the model into two groups depending on the presence of Transformer layers [192], leading to models being divided into two groups: (1) ResNet-50, VGG-19, DCGAN, and PointNet; (2) GPT3-Medium, GPT3-XL, and GPT3-3B. For the first group, distributed training is conducted using PyTorch DDP. In contrast, for the second group, 3D parallelism training is implemented using Megatron-LM [174]. The parallelism strategy can be adjusted before launching jobs on the cluster. Furthermore, Tesserae imposes a limit of two models running simultaneously on each GPU because packing more than two jobs typically does not provide additional benefits [73, 130].

4.5 Evaluation

In this section, we evaluate Tesserae and focus on the following aspects - (i) The effectiveness of Tesserae on a real cluster; (ii) The impact of each of our performance optimizations on Tesserae's efficiency; (iii) The adaptability and compatibility of Tesserae; (iv) The scalability of Tesserae.

4.5.1 Experimental Setup

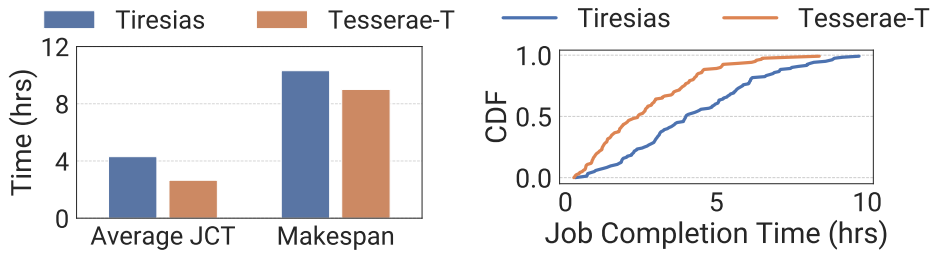
Testbed. We conduct cluster experiments using 32 GPUs across 8 nodes on NERSC Perlmutter [6]. Each node has four 40 GB NVIDIA A100 (Ampere) GPUs, 256 GB DDR4 DRAM, and a single AMD EPYC 7763 (Milan) CPU. We also run simulation experiments on a Cloudlab server [57]. The server contains 188 GB memory and two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz.

Traces. To evaluate the schedulers we use traces introduced by two prior works Shockwave [236] and Gavel [130]. Table 4.1 lists the workload details including models, dataset, and batch size. Our default trace is similar to Shockwave [236], we follow the same setting as Shockwave by setting the probability of generating Small, Medium, Large, and Extra Large jobs to be 0.72, 0.2, 0.05, and 0.03. We also set the probability of generating 1-GPU, 2-GPU, 4-GPU, and 8-GPU jobs to be 0.6, 0.3, 0.09, and 0.01 to align with those used in Shockwave. The job arrival rate is configured at 80 jobs per hour, consistent with the settings used in prior work. We show how Tesserae's benefits change across workloads by using an additional trace similar to the one used in Gavel [130] in §4.6. By default, we use a 120 jobs trace for physical experiments and a 900 jobs trace for simulated experiments, both with a job arrival rate of 80 jobs per hour.

Baselines. We compare Tesserae with *four* different baselines (i) Tiresias [60]; (ii) Tiresias (Single) [60, 73]; (iii) Gavel [130]. Tiresias [60] uses 2D-LAS to perform fair sharing of the cluster. Tiresias (Single) employs the Tiresias [60] scheduling policy and uses Tesserae for job packing; however, similar to Lucid [73] and Pollux [145] it does not pack distributed jobs by default. Gavel [130] formulates scheduling as an optimization framework that supports the LAS policy and incorporates job packing.

In addition, we also compare Tesserae with Gavel-FTF [130] to study how Tesserae affects fairness metrics. Gavel-FTF [130] performs job packing and solves an optimization problem associated with the finish-time fairness (FTF) metric.

Configuration. Tesserae similar to prior works [130, 236] is a round-based scheduler. We set the round duration to six minutes. Tesserae is designed as a modular packing policy that can work with any scheduling policy. We evaluate Tesserae by combining it with Tiresias (Tesserae-T), and FTF (Tesserae-FTF). Tesserae-T denotes using Tiresias scheduling



(a) Scheduling Efficiency **(b)** CDF for Scheduler

Figure 4.9: Physical cluster evaluation: We evaluate Tesserae-T against Tiresias on a 32-GPU physical cluster. Compared to Tiresias, Tesserae-T improves Avg. JCT by 1.62× and Makespan by 1.15×.

with Tesserae as a placement policy. Tesserae-FTF denotes using FTF scheduling with Tesserae.

Performance Metrics. Similar to the prior DL schedulers, we report standard metrics: Average job completion time (Avg. JCT), and the time needed to complete all jobs (Makespan). Additionally, we also evaluate the finish-time fairness (FTF) ratio to capture fairness.

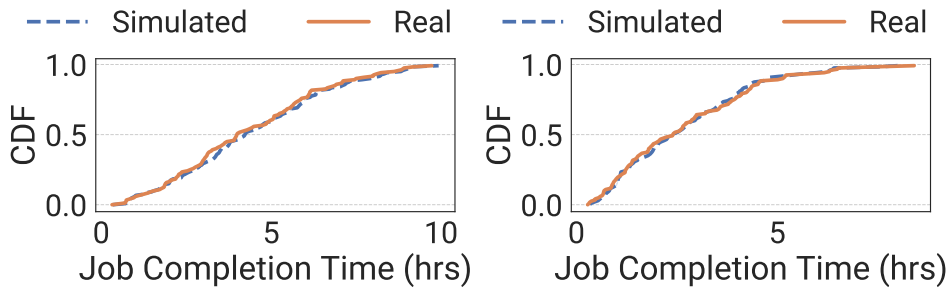
4.5.2 End-to-End Real Cluster Experiments

In this section, we evaluate Tesserae in both cluster and simulation settings.

Tesserae Comparison. To evaluate the benefits of Tesserae, we first run Tesserae-T and compare it against Tiresias on a 32-GPU physical cluster. In Figure 4.9a, we show that Tesserae-T can improve Avg. JCT by 1.62×, and Makespan by 1.15× on physical clusters. Figure 4.9b shows a CDF of JCTs. In Figure 4.9b, we observe that Tesserae-T can significantly reduce the Avg. JCT for jobs with a short duration, which is especially impactful given that Tiresias (and LAS scheduling) is designed to prefer short jobs.

Simulating Tesserae. Due to lack of access to large scale production clusters, prior works [60, 73, 79, 117, 123, 130, 145, 205, 232, 236] have used simulations to perform detailed study of large scale traces and compare different metrics. We follow a similar approach.

First, we verify that our simulation closely approximates runs on a real cluster. Since profiling can often have significant noise when performing packing, we run profiling five



(a) Tiresias **(b)** Tesserae-T

Figure 4.10: Comparison of CDFs between cluster and simulator: We depict the CDF for Tiresias and Tesserae-T obtained from physical experiments compared with simulated results. The results demonstrate our simulator's low fidelity.

Table 4.2: The Fidelity of simulator: We run the simulation five different times and depict the mean deviation and standard deviation. We observe the maximum deviation being 5.42% highlighting that our simulator closely follows the real cluster.

Method	Avg. JCT (s)	Makespan (s)
Tiresias	3.36% $\pm$ 0.46%	2.05% $\pm$ 0.03%
Tesserae-T	0.35% $\pm$ 0.33%	5.42% $\pm$ 0.95%

different times and during simulation we choose one of them at random. To account for noise, we also run the simulation five different times. In Table 4.2, we show that our simulation shows the maximum average deviation for JCT between physical cluster and simulation is 3.36% and the maximum deviation for makespan is 5.42%.

In Figure 4.10, we also randomly sample one simulation run and present the CDF of JCTs to highlight the accuracy of our simulation. The average JCT deviation between the physical cluster and simulation results for Tesserae-T is 0.21%, as shown in Figure 4.10.

4.5.3 End-to-End Results in Simulation

To further evaluate Tesserae, we use simulation on a large 900 job trace with an 80 GPU cluster. We first evaluate the effect of the migration and packing algorithms introduced in §4.3.1 and §4.3.2, respectively, to isolate the performance benefits of Tesserae. Next, we investigate the adaptability and compatibility of Tesserae with various hardware and schedulers. Finally, we evaluate the scalability of Tesserae by varying the number of active

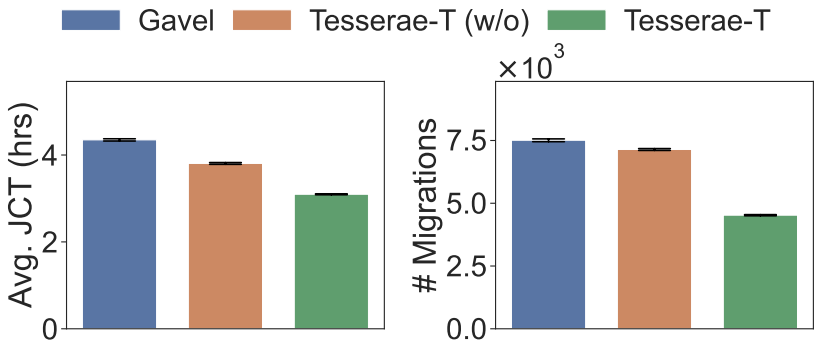
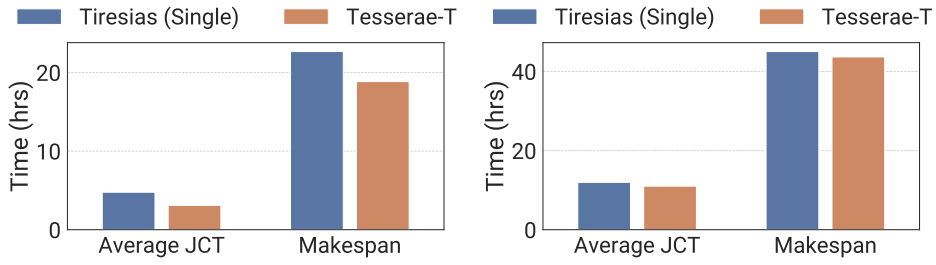


Figure 4.11: Evaluating Tesserae-T against optimization-based solutions: We use w/o to denote the use of the basic migration algorithm described in [130]. First, we notice that our packing policy and migration policy improve the Avg. JCT by 1.41× for Tesserae-T compared with Gavel. Second, we observe that our migration policy reduces the number of migrations by 36% for Tesserae-T.

jobs and analyzing the overhead introduced by each policy.

Performance Comparison against Optimization Solutions. Figure 4.11 shows that our packing policy improves the Avg. JCT by 1.15× compared to the optimization-based scheduler Gavel [130]. Tesserae-T leverages Tiresias as the scheduling policy while aiming to use our graph-based packing policy (§4.3.2) to maximize total throughput. In addition, we also observe that our migration algorithm outlined in §4.3.1 reduces the migrations of Tesserae by 36%, compared to the basic migration algorithm used in [130]. Furthermore, the reduced migration improves Avg. JCT by 1.22×. This highlights that reducing migrations can significantly enhance scheduling efficiency. This also indicates that schedulers should also account for minimizing migrations.

Performance Comparison against Heuristic Methods Figure 4.12 compares Tesserae with Tiresias (Single), which uses Tiresias for scheduling and applies the packing policy from §4.3.2 only to 1-GPU jobs due to network contention. Figure 4.12a shows that Tesserae improves Avg. JCT by 1.54× and reduces makespan by 1.20× compared to Tiresias (Single), by leveraging more packing opportunities to increase per-round throughput. This highlights the effectiveness of our packing policy in enhancing scheduling efficiency.



(a) NERSC A100 Machine **(b)** AWS V100 Machine

Figure 4.12: Evaluating Tesserae against heuristic solution: Tiresias (Single) employs the Tiresias scheduling policy [60] and utilizes Tesserae for job packing; however, following [73], it defaults to packing only 1-GPU jobs. Experimental results demonstrate that Tesserae improves the Avg. JCT and makespan by up to 1.54× and 1.20×, respectively, compared to Tiresias (Single).

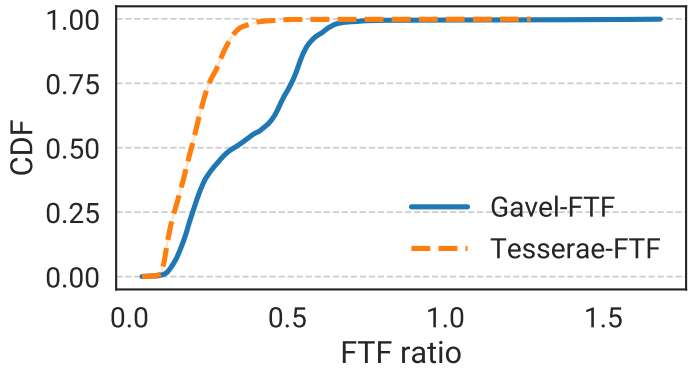
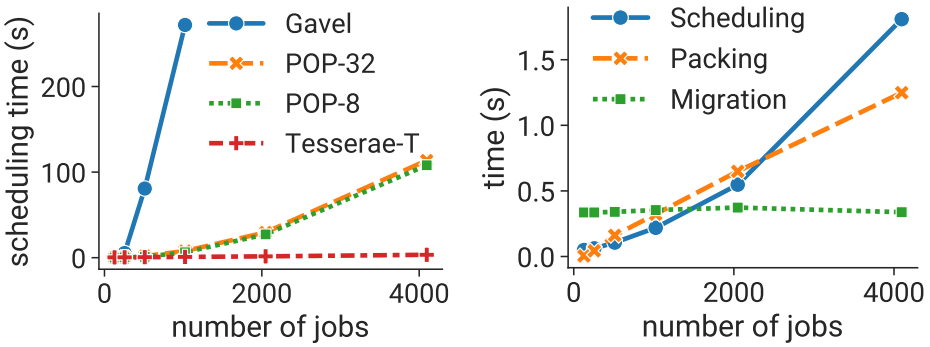


Figure 4.13: Evaluation Tesserae-FTF's fairness: The CDF of Finish-time fairness (FTF) ratio [117]. The results indicate that Tesserae-FTF achieves the lowest worst-case FTF ratio, outperforming Gavel-FTF.

Adaptability of Tesserae. To evaluate Tesserae can adapt to changing hardware we switch our experiment testbed to V100 GPUs using AWS p3.16xlarge instances. We use the same workload used in Figure 4.12a. We observe that Tesserae can easily adapt to changing hardware and as shown in Figure 4.12b improves the Avg. JCT and Makespan by 1.08× and 1.03× respectively. Compared to the results on the A100 GPU, the V100 GPU's lower performance and limited memory capacity reduce packing opportunities, thereby diminishing the overall scheduling gains. Tesserae adapts to this changing hardware and outperforms heuristic methods.



(a) Overhead **(b)** Breakdown

Figure 4.14: Scalability of Schedulers: The left figure shows the overhead of Tesserae-T compared with Gavel [130] and POP [128] with the increased number of active jobs. The right figure presents the overhead breakdown of Tesserae-T.

Compatibility with Other Schedulers Tesserae is implemented as a modular packing and placement plugin on top of existing schedulers. We can use Tesserae over existing schedulers without modifying the underlying scheduler. To show modularity of Tesserae and it's impact on underlying metric, we implement Tesserae over Gavel-FTF. To evaluate fairness, similar to prior work [117] we use FTF ratio as a metric. FTF is defined as $\rho = \frac{T_s}{T_f}$, where T_s is the job completion time in a shared cluster and T_f is the job completion time in an isolated and fairly shared cluster. In Figure 4.13, we show that Tesserae-FTF can enhance the performance of Gavel-FTF. This highlights that Tesserae-FTF also provides higher fairness than existing baselines.

Scalability Analysis We fix the number of GPUs in the cluster and vary the number of active jobs to evaluate the scalability of Tesserae. In Figure 4.14a, we observe that Tesserae scales better than Gavel [130] and POP [128] as the number of active jobs increases significantly. Moreover, Figure 4.14b presents a breakdown of Tesserae-T's overhead. We notice that the overhead of scheduling and packing increases with the number of active jobs, while the migration overhead remains stable. This is because the scheduling and packing algorithms scale with the number of active jobs, whereas the migration overhead depends on the number of GPUs in the cluster.

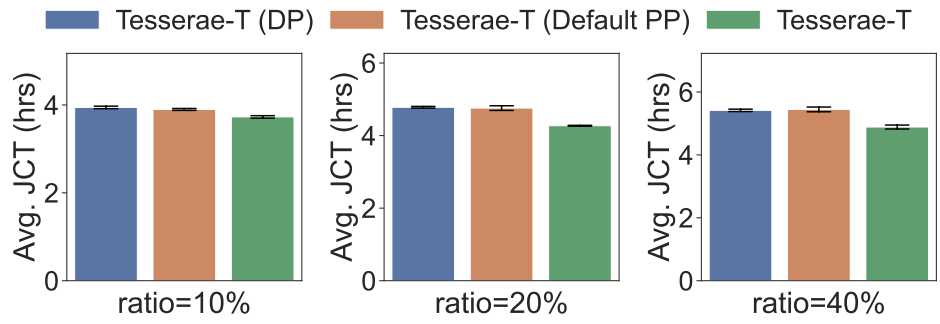


Figure 4.15: Impact of Parallelization strategy: We compare the impact of parallelism strategy on Avg. JCT of Large Language Models (GPT3-Medium, GPT3-XL, and GPT3-3B). The Default PP (Def PP) is provided by Megatron-LM [174]. Tesserae-T selects the best parallelism strategy from DP, TP, and the candidate of possible PP strategies. By varying the ratio of large language models in the workload, we observe that selecting the best parallelism strategy can improve Avg. JCT of large language models by 1.12×.

4.6 Ablation Studies

In this section, we investigate various parameters of Tesserae and evaluate the packing and parallelism strategy with ablation experiments.

4.6.1 Impact of Parallelization Strategy

To highlight the impact parallelization strategies can have on the throughput of packed jobs, we compare Tesserae-T (DP), Tesserae-T (Default PP), and Tesserae-T in this experiment. Tesserae-T (DP) selects data parallelism as the parallelism strategy for packed language model training jobs. The parallelism strategy selected by Tesserae-T (Default PP) is the default pipeline parallelism strategy used in Megatron-LM [174]. In contrast, Tesserae-T picks the best parallelism strategy from the candidate set defined by users. As demonstrated in Figure 4.15, by adjusting the parallelism strategy for packed jobs, there is a 12% improvement in Avg. JCT for large language models.

4.6.2 Parameter Sensitivity

Sensitivity to Profiling Errors. It is possible that the profiling results are not correct due to software or hardware variabilities [119, 166]. In this experiment, the profiling data we used to make packing decisions is multiplied by a random factor sampled from $[1-n_p, 1+n_p]$,

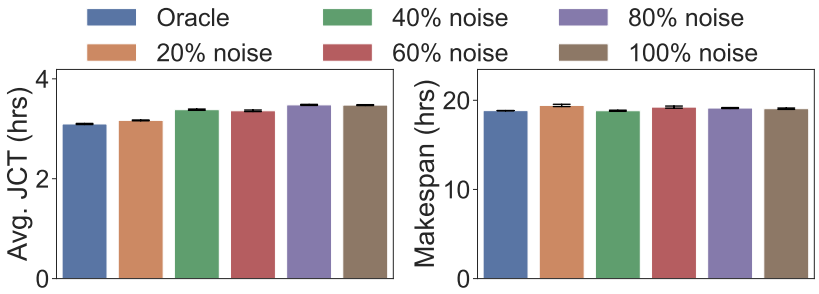


Figure 4.16: Impact of inaccurate profiling on Tesserae-T: Our results indicate that Tesserae-T is robust to noise in profiling data, even when the noise is 100%.

where n_p is a noise parameter from $[0, 1]$. From the results shown in Figure 4.16, we notice that the Avg. JCT is increased by at most 1.12× with 100% noise, while the Makespan is robust with the increased noise parameter n_p. However, in reality, we find the profiling noise is always under 20%, and thus the profiling noise has little impact on Tesserae.

Sensitivity to Workload. We also compare Tesserae-T with other baselines on a 900 jobs trace, which is generated by Gavel's trace generator, with an 80-GPU cluster. For this trace, pursuant to Gavel, the duration of jobs is uniformly sampled between $10^{[1.5,3]}$ minutes with 80% probability, and the remaining 20% jobs have their duration uniformly sampled $10^{[3,4]}$ minutes. Similar to Gavel trace, 70% of the jobs request a single GPU, 10% of the jobs request 2 GPUs, 15% of the jobs request 4 GPUs, and the remaining 5% of the jobs request 8 GPUs. From Figure 4.17, we have a similar observation, Tesserae-T outperforms all evaluated baselines over performance metrics. Specifically, Tesserae-T can reduce Avg. JCT, and Makespan by up to 1.87×, and 1.32× respectively.

Reduce Profiling Cost. It is expensive to profile each model and each possible model combination. To reduce the profiling cost, we utilize a linear model [79] to estimate the throughput for the data parallel applications and bayesian optimization [177] to predict the throughput of LLM workloads with varying parallelism strategies. Figure 4.18 shows that our strategy outperforms the matrix completion method used in Gavel [130] and Quasar [50]. In addition, our throughput estimator can predict missing throughput with only a minor reduction in Avg. JCT compared to Oracle, which involves offline profiling of each model and each model combination.

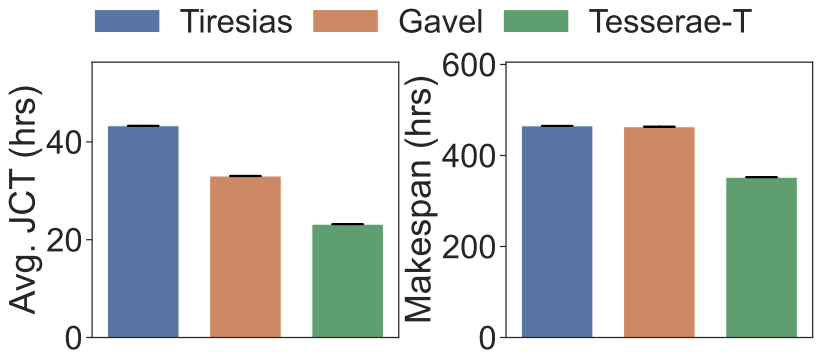


Figure 4.17: Evaluating Tesserae-T's scheduling efficiency by varying the workload: We evaluate Tesserae-T's scheduling efficiency on a large-scale cluster with 80 GPUs over trace generated by Gavel's trace generator. This trace holds 900 jobs but follows a different duration distribution. The results show that Tesserae-T improves Avg. JCT by up to 1.87× compared with existing scheduling algorithms.

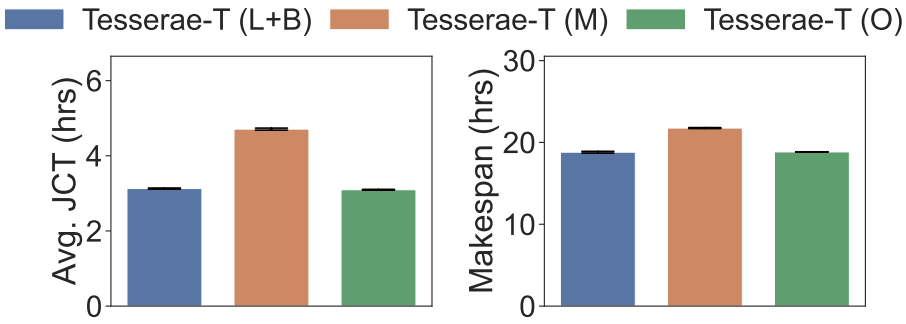


Figure 4.18: Reduce Profiling Cost: We compare our throughput estimator (Linear model and Bayesian optimization) with Matrix Completion and Oracle. The results show that our throughput estimator can be used to reduce profiling costs and maintain scheduling efficiency.

4.7 Conclusion

In this paper, we propose Tesserae, a general GPU cluster scheduler that supports various existing scheduling policies. Rather than formulating the packing problem as an optimization problem, we develop an innovative and efficient packing algorithm inspired by the Hungarian algorithm. Furthermore, Tesserae is the first scheduler to facilitate the packing of 3D Parallelism training jobs. We also explore opportunities for optimizing the parallelism strategy for packing jobs. Lastly, Tesserae develops a novel migration algorithm to reduce

migrations during scheduling. Our physical and simulated experiments show that Tesserae outperforms existing state-of-the-art schedulers.

Chapter 5

Architectures for Efficient Inference

This chapter first shows that the number of parameters is not the exclusive factor affecting inference efficiency. As illustrated in Figure 5.1, the model architecture also plays a critical role. Following this observation, this chapter introduces inference-efficient scaling laws, building upon the Chinchilla scaling law and incorporating model architecture considerations in §5.2. Additionally, due to the disparity between model loss and accuracy in downstream tasks, we develop a novel method, in Figure 5.6, that utilizes inference-efficient scaling laws to rank various model architectural choices in §5.2. In §5.3, we describe the experimental setup, and in §5.4, we summarize the main results. Finally, we train the Morph-1B[1] model using the best model configuration predicted by the inference-efficient scaling law and ranking algorithm. Compared to the default Open-LM-1B model, this configuration reduces inference latency by 1.8× while maintaining downstream-task accuracy.

5.1 Preliminaries

Scaling laws predict a model's loss based on the allocated compute resource C. Following OpenAI [85] and Chinchilla [70], the compute resource C is a function dependent on the model size N and the number of training tokens D. The goal is to minimize model loss

[1]The training code is available at `https://github.com/Waterpine/open-lm-morph`. The Morph-1B model checkpoint is available at `https://huggingface.co/NaiveUser/morph-1b`.

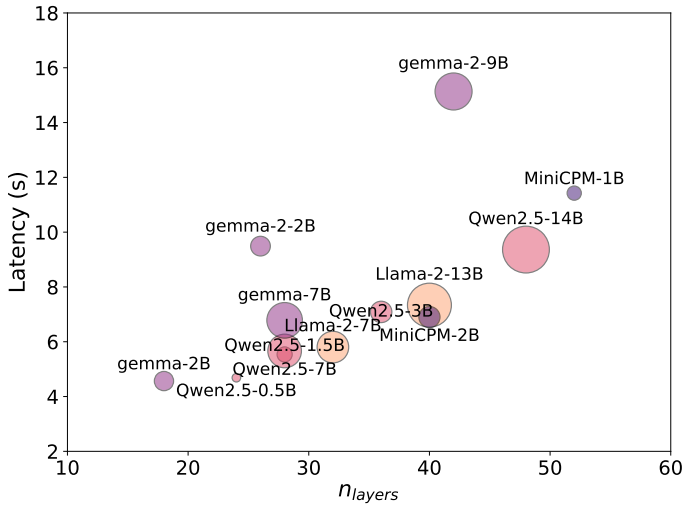


Figure 5.1: Open-Source LLMs' Inference Latency: An overview of inference latency in open-source LLMs. The evaluated models include LLaMA [190], Qwen [217], Gemma [187, 188], and MiniCPM [74]. All evaluations were performed using the Hugging Face `generate` function on a single NVIDIA Ampere 40GB A100 GPU with batch size 1, input length 128, and output length 256.

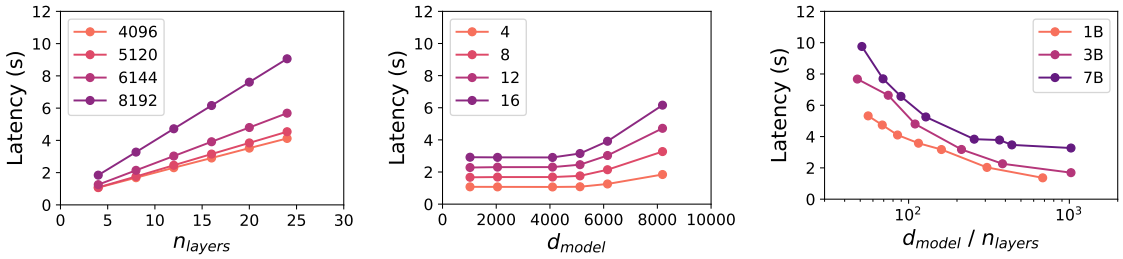
within the constraints of the available compute resources:

$$\arg\min_{N,D} L(N,D) \text{ s.t. } \text{FLOPs}(N,D) = C \tag{5.1}$$

Using the formulation above, several scaling laws have been established [70, 85, 125, 164] to accurately model the performance of large language models from training a series of much smaller ones. The Chinchilla loss function $L(N,D)$ is widely adopted to predict a model's training loss:

$$L(N,D) = E + AN^{-\alpha} + BD^{-\beta} \tag{5.2}$$

where N is the number of parameters, D is the number of tokens used for training and A, B, E, α, β are parameters to be learned. Through training multiple models and curve fitting, Chinchilla [70] identifies $D \approx 20N$ as the compute-optimal solution for large language model pretraining.



(a) Vary layers (n_{layers}), fix hidden size

(b) Vary hidden size (d_{model}), fix layers

(c) Vary ratio (d_{model}/n_{layers}), fix size N

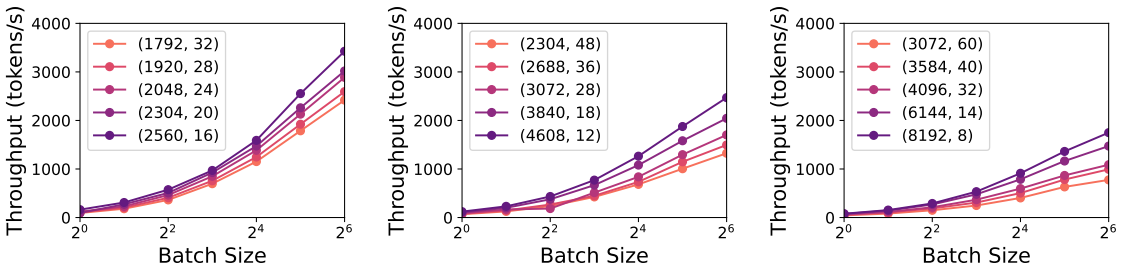
Figure 5.2: Model Shape on End-to-End Inference Latency: (Left) We illustrate the correlation between inference latency and the number of layers, with constant hidden size. Due to the sequential nature of LLM execution, latency increases linearly with the number of layers. (Center) We plot the relationship between inference latency and hidden size with the number of layers fixed. We see that model width does not affect latency for smaller models but only for larger models. (Right) We show the relationship between inference latency and aspect ratio, with the number of model parameters fixed. We see a downward trend in inference latency as we make the model wider and shallower. All evaluations were performed using the Hugging Face generate function on a single NVIDIA Ampere 40GB A100 GPU with batch size 1, input length 128, and output length 256.

5.2 Inference-Efficient Scaling Laws

Despite its popularity, the Chinchilla scaling law fails to resolve the following challenges:

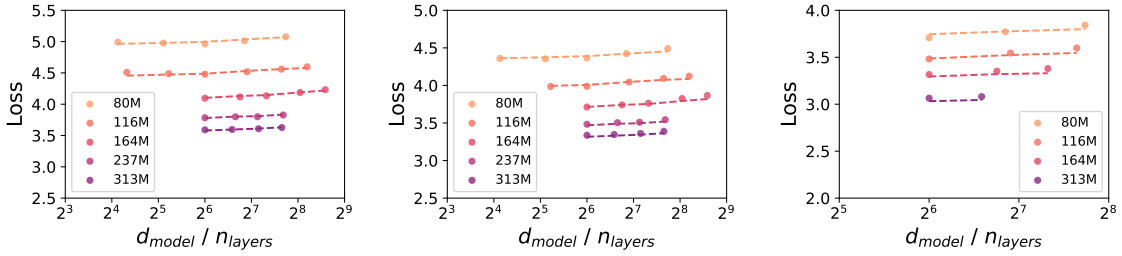
- The FLOPs constraint outlined in Eq. (5.1) does not reflect how model training decisions are made in practice. First, both the model size and the training corpus are determined in advance to accommodate for resource constraints when deploying these models [190]. Therefore, for each model and training corpus pair, training FLOPs is essentially a fixed constant (assuming training epochs are also predetermined). Furthermore, while the Chinchilla scaling law suggests training a 10B parameter model with 200B tokens, overtraining frequently occurs in practice. For example, the LLaMA-3-8B model uses 15 trillion tokens for training [190], while the Gemma-2-9B model utilizes 8 trillion tokens [188]. These numbers are 44-93$\times$ larger than the Chinchilla optimal recommendation.

- Existing scaling laws focus only on how the number of parameters affects inference latency. However, as depicted in Figure 5.1, smaller models can sometimes exhibit



(a) 1B Model Variants **(b)** 3B Model Variants **(c)** 7B Model Variants

Figure 5.3: Model Shape on Throughput: We examine the relationship between inference throughput and model architecture by fixing the total parameter count and varying the hidden size and number of layers. Across different batch sizes, wider and shallower models consistently yield better inference throughput for large language models. Each tuple in the legend represents a model configuration: the first number is the hidden size d_{model}, and the second is the number of layers n_{layers}. All evaluations were performed using the Hugging Face generate function on a single NVIDIA Ampere 40GB A100 GPU with input length 128 and output length 256.



(a) 20N tokens **(b)** 40N tokens **(c)** 160N tokens

Figure 5.4: Inference-Efficient Scaling Laws: In this plot, each data point represents a training run with the given configuration. The dashed lines represent predictions based on the inference-efficient scaling laws outlined in Eq. (5.4). (Left) The number of training tokens is 20N; (Center) The number of training tokens is 40N; (Right) The number of tokens used for training is 160N, where N denotes the number of parameters. Our scaling law accurately captures the training loss across different training durations.

higher inference latencies than larger models. For instance, MiniCPM-1B [74] has a higher latency compared to Qwen2.5-14B [217].

Table 5.1: Model Configurations: We present the configurations of models available on Hugging Face.

Model	d_{model}	n_{layers}	d_{model} / n_{layers}
Llama-3.2-1B [56]	2048	16	128
Llama-3.2-3B [56]	3072	28	109.7
Qwen2.5-0.5B [217]	896	24	37.3
Qwen2.5-1.5B [217]	1536	28	54.9
Qwen2.5-3B [217]	2048	36	56.9
Qwen2.5-7B [217]	3584	28	128
Qwen2.5-14B [217]	5120	48	106.7
gemma-2b [187]	2048	18	113.8
gemma-7b [187]	3072	28	109.7
gemma-2-2b [188]	2304	26	88.6
gemma-2-9b [188]	3584	42	85.3
gemma-2-27b [188]	4608	46	100.2
microsoft-phi-2 [2]	2560	32	80
microsoft-phi-4 [9]	5120	40	128

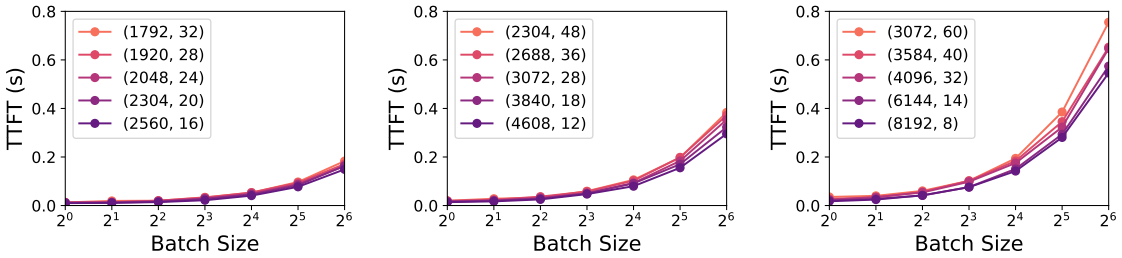
In view of this, we propose rewriting Eq. (5.1) as below to meet practical requirements:

$$\arg\min_{N,D} L(N,D) \text{ s.t. } N \leqslant N_C, D \leqslant D_C, T_{\text{inf}} \leqslant T_C \tag{5.3}$$

where N_C represents the constraint on model size and D_C denotes the constraint on the number of training tokens. To account for the inference latency budget, we introduce a new term T_C to our scaling law formulation to represent the inference latency constraint.

Motivated by Figure 5.1, we closely examine the effect of the aspect ratio ($d_{\text{model}}/n_{\text{layers}}$) on inference latency and throughput by altering the hidden size d_{model} and the number of layers n_{layers} as shown in Figure 5.2 and Figure 5.3. Reasonable aspect ratios are chosen based on open-weight models listed in Table 5.1, which presents model configurations from Hugging Face, highlighting the vast space of architectural design choices.

Figure 5.2a shows that inference latency increases linearly with the number of layers when the hidden size remains constant. This occurs as the inference computation must be performed sequentially, one layer at a time [215]. However, the matrix computations within



(a) 1B Model Variants **(b)** 3B Model Variants **(c)** 7B Model Variants

Figure 5.5: Model Shape on Time To First Token (TTFT): We examine the relationship between TTFT and model architecture by fixing the total parameter count and varying the hidden size and number of layers. Across different batch sizes, wider and shallower models consistently achieve lower TTFT. Each tuple in the legend represents a model configuration: the first number is the hidden size d_{model}, and the second is the number of layers n_{layers}. All evaluations were performed using the Hugging Face generate function on a single NVIDIA Ampere 40GB A100 GPU with input length 128, and output length 1.

a single layer can be performed in parallel. Furthermore, Figure 5.2c indicates that for the same number of parameters, we can achieve different latency targets by changing the ratio of the number of hidden parameters in one layer (d_{model}) vs. the number of layers (n_{layers}). Moreover, in Figure 5.3, We study the relationship between model shape and inference throughput under a fixed parameter budget. We observe that, under a fixed parameter budget, wider and shallower models consistently achieve higher inference throughput. Due to space constraints, results on the relationship between aspect ratio and time to first token (TTFT) are provided in Figure 5.5. From Figure 5.5, we observe that wider and shallower models consistently achieve lower TTFT.

Prior work [85] has shown the impact of the aspect ratio (d_{model}/n_{layers}) on the performance of the model. However, it does not define the connection between model size, number of training tokens, and model shape. To establish this relationship, we trained several small models $N \in \{80, 116, 164, 237, 313\}M$ by varying the aspect ratio and setting $D \in \{20, 40, 160\}N$. Due to resource limitations, we only train a subset of the models at $D = 160N$. We plot the loss values against the aspect ratio in Figure 5.4. From the figure, we can see that the most suitable model shape adjustment is the inclusion of the term $(1 + \varepsilon R^{\gamma})$ to the Chinchilla scaling law [70]. Therefore, we derive the following inference-efficient

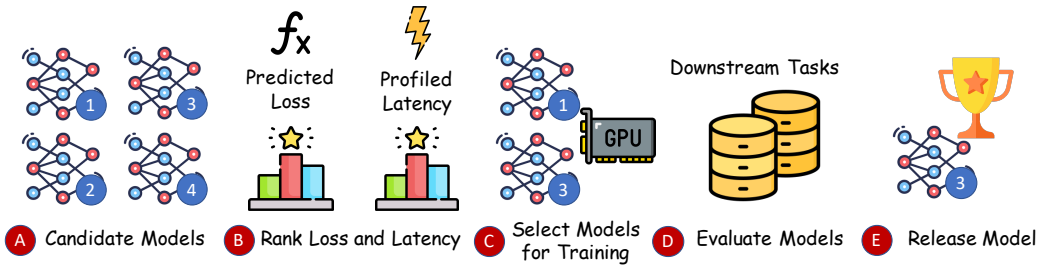
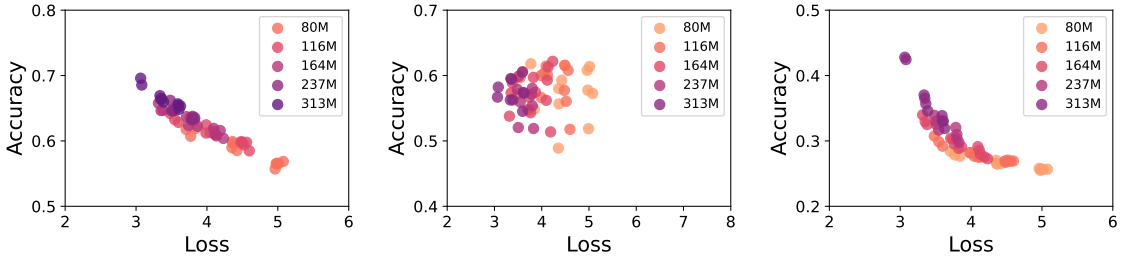


Figure 5.6: An Overview of Methodology: (A) The model training team first selects several candidate models with various model sizes and configurations; **(B)** Measure the inference latency using open-source inference systems and predict model loss with fitted scaling laws; **(C)** Select top-k candidate models for training based on inference latency and loss; **(D)** Evaluate the models over downstream tasks after training; **(E)** Release the best model based on inference efficiency and performance over downstream tasks.



(a) PIQA **(b)** BoolQ **(c)** HellaSwag

Figure 5.7: Accuracy vs. Loss: (Left) We illustrate the correlation between accuracy and model loss on PIQA [32]. (Center) We present the connection between accuracy and model loss on BoolQ [42]. (Right) We show the connection between accuracy and model loss on HellaSwag [226]. These three patterns shown in the plots demonstrate the difficulty in robustly predicting individual downstream task accuracies from scaling laws.

scaling law formulation:

$$L(N, D, R) = (E + AN^{-\alpha} + BD^{-\beta}) \cdot (1 + \varepsilon R^{\gamma}) \tag{5.4}$$

where N is the number of parameters, D is the number of training tokens, and $R = d_{\text{model}}/n_{\text{layers}}$ is the aspect ratio.

Moreover, $A, B, E, \alpha, \beta, \gamma, \varepsilon$ are learned parameters. In Figure 5.4, we plot the predicted values from the scaling law against the observed values from training. More details of the experimental setup and fitting procedure can be found in §5.3.

Scaling laws were first developed to predict the loss of language models. However, LLMs are evaluated on the *performance of downstream tasks*. A recent study [58] attempts to establish scaling laws that link evaluation loss to errors in downstream tasks.

Inherently, predicting the error in downstream tasks becomes challenging when model losses are similar, due to noise and inaccuracies in scaling laws. We observe this in Figure 5.7. To tackle this challenge, we develop a new method for training inference-efficient models, as shown in Figure 5.6. Our key idea is that inference latency measurement has negligible overhead, and scaling laws can help us estimate the loss of scaled-up models. Thus, we propose identifying top-k candidate models using inference latency and loss data, where the user can choose k. After training, we evaluate these models on downstream tasks and release the best-performing model to the public, taking into account both inference latency and performance on downstream tasks. Our method, as shown in Figure 5.6, can also be applied to different architectural optimizations, such as MLA [112], to quantify the accuracy-efficiency tradeoff.

5.3 Experiments

We next discuss the experiment setup we use for model training and evaluation in §5.3.1. Following that, in §5.3.2, we demonstrate how to fit scaling laws using our experimental results.

5.3.1 Experimental Setup

Training Setup. For all experiments, we train transformer-based decoder-only language models [192]. Following [58, 63], the model's architecture is similar to GPT-2 [148] and LLaMA [190], with GPT-NeoX [33] employed as the tokenizer. We train models with a maximum of 1.5 billion parameters for up to 30 billion tokens, following the compute-optimal setup in [70]. The models are trained on uniformly sampled subsets of DCLM-Baseline [104] with one epoch, ensuring no repetition in data (other than possible data repetition in the dataset itself). We follow the hyperparameters mentioned in [58, 104] with the specific details presented in Table 5.2. A cooldown rate of 3e-5 is used in all experiments. All models are trained in bfloat16 precision using the AdamW optimizer. The number of parameters is computed using `sum(p.numel() for p in model.parameters())`. To examine how

Table 5.2: Hyperparameters: We show the hyperparameters used for training in this paper. In addition, the batch size is the global batch size and the default sequence length is 2048.

Model Size	Warmup	Learning rate	Weight decay	z-loss	Batch size
<400M	2000	3e-3	0.033	1e-4	512
1B	5000	3e-3	0.033	1e-4	256

model architecture influences loss metrics and inference performance, we vary the model configurations. Architectural details are provided in Table B.1.

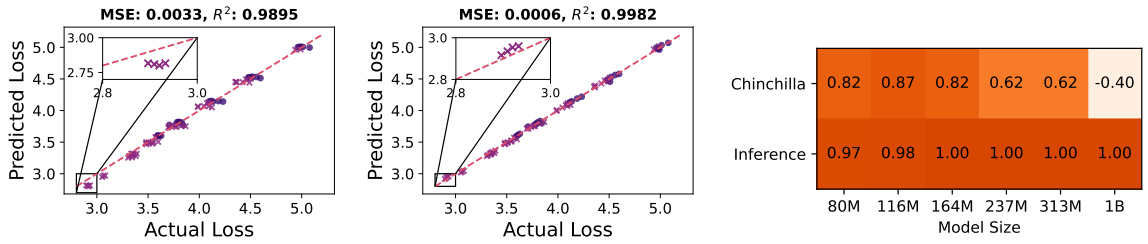
Evaluation Setup. We use HuggingFace [207] to measure the inference efficiency of models over a single NVIDIA Ampere 40GB A100 GPU. By default, we set the number of input and output tokens to be 128 and 256, respectively, aligning with the distribution outlined in ShareGPT [97].

We use LLM-foundry [5] along with a zero-shot evaluation approach to evaluate model performance on downstream tasks. We evaluate the downstream task accuracy of models derived from the methodology outlined in Figure 5.6 using the following datasets: ARC-Easy [43], ARC-Challenge [43], BoolQ [42], COPA [157], HellaSwag [226], LAMBADA [138], PIQA [32], WinoGrande [160], MMLU [66], Jeopardy [1], and Winograd [98].

Furthermore, to compare the predicted loss against the actual loss, we measure relative prediction error: $|\psi - \hat{\psi}|/\psi$, mean squared error (MSE): $\frac{1}{n}\sum_{i=1}^{n}(\psi_i - \hat{\psi}_i)^2$, and $R^2 = 1 - \sum_{i=1}^{n}(\psi_i - \hat{\psi}_i)^2 / \sum_{i=1}^{n}(\psi_i - \bar{\psi})^2$, where ψ represents the actual loss, $\hat{\psi}$ the predicted loss from scaling laws, and $\bar{\psi} = \frac{1}{n}\sum_{i=1}^{n}\psi_i$. We also apply Spearman's rank correlation coefficient [181] to evaluate how well the predicted rankings correspond to the actual rankings.

5.3.2 Fitting Scaling Laws

Following [58], we use the Levenberg-Marquardt algorithm to fit Eq. (5.4). The Levenberg-Marquardt algorithm solves least-squares curve fitting problems, where the goal is to find the parameter vector β of a model $f(x, \beta)$ that minimizes the sum of squared deviations. Formally, the problem can be expressed as $\arg\min_\beta \sum_{i=1}^{m}[y_i - f(x_i, \beta)]^2$, where (x_i, y_i) are



(a) Chinchilla **(b)** Inference-Efficient **(c)** Spearman Correlation

Figure 5.8: Comparison: (Left) We illustrate the predicted versus actual loss using Eq. (5.2). (Center) We display the comparison of predicted to actual loss based on Eq. (5.4). Dots represent data points used for curve-fitting, while cross marks represent test data points. (Right) We demonstrate that our inference-efficient scaling law yields a significantly higher Spearman correlation, resulting in more precise predictions of the optimal model configuration.

data pairs. Following observations from Chinchilla scaling law [70] and another recent work [58], we set α, β, and γ equal to simplify the fitting procedure. To fit and evaluate the scaling law, we train 63 models using a range of model sizes, shapes, and amounts of training tokens. The size of our model ranges from 80M to 339M and the number of tokens used for training ranges from 1.6B to 12.8B. Detailed model configurations can be found in Table B.1 in Appendix B.1.

5.4 Results

In this section, we first study the predictive power of our inference-efficient scaling laws in §5.4.1. Then, in §5.4.2, we release an inference-efficient model that maintains accuracy on downstream tasks compared with open-sourced models by using the methodology outlined in Figure 5.6. We also show that our method significantly outperforms Chinchilla in predicting the best model configurations. Finally, we perform ablation studies on obtaining robust scaling laws and show that our inference-efficient scaling law is more robust than Chinchilla in various scenarios in §5.4.3.

Table 5.3: Data Used to Fit Scaling Laws: In this table, we show the number of parameters and tokens used in model training to fit the scaling laws in Figure 5.8-5.10. ✓ indicates we use all model variants with the given size and ✗ means we do not use any model variants with the given size. ❖ indicates that we randomly sample one model variant from the candidate set. The details of model variants are included in Appendix B.1.

N	D	Figure 5.8	Figure 5.9	Figure 5.10
80M	1.6B	✓	✓	❖
116M	2.3B	✓	✓	❖
164M	3.2B	✓	✓	❖
237M	4.7B	✓	✓	❖
313M	6.2B	✓	✓	❖
80M	12.8B	✓	✗	❖

5.4.1 Prediction acccuracy

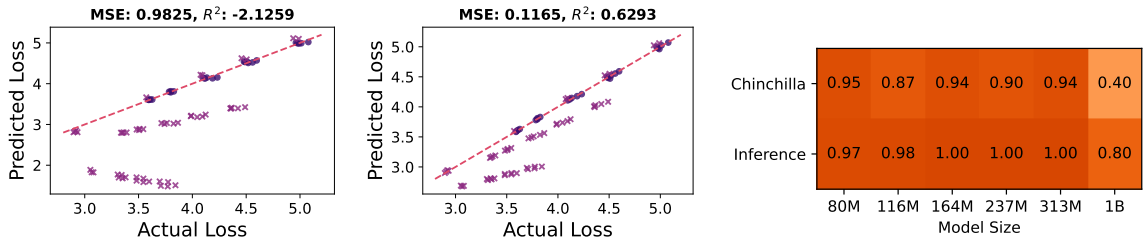
As shown in §5.3.2, we obtain the actual losses of various models by training multiple small models with different model configurations to establish the scaling law. We set $N \in \{80, 116, 164, 237, 313\}$M and $D = 20N$ to train small models and collect the data to fit the learnable parameters in Eq. (5.2) and Eq. (5.4). Furthermore, to enhance the generality of the scaling law, we train 80M models with $D = 160N$ tokens, thereby collecting data from an over-training setting.

Then, we train larger models on more tokens to evaluate the predictive power of our inference-efficient scaling law. We present the results in Figure 5.8. Figure 5.8 demonstrates that our scaling law achieves higher accuracy than the Chinchilla scaling law, as shown by a smaller MSE and a larger R^2 [209] value. We reduce MSE from 0.0033 to 0.0006 while improving R^2 from 0.9895 to 0.9982. In addition, the relative prediction error for the inference-efficient scaling law is less than 1.2%, whereas for the Chinchilla scaling laws, it ranges from 2.7% to 4.1%. This demonstrates that the inference-efficient scaling law predicts more accurately than the Chinchilla scaling law.

Furthermore, as illustrated in Figure 5.6, prioritizing the ranking of predicted loss is more critical than its absolute value when employing the training methodology described in Figure 5.6 for inference-efficient models. We calculate Spearman's rank correlation coefficient [181] for both the Chinchilla scaling law and the inference-efficient scaling law when

Table 5.4: Inference-Efficient Models: In this table, we compare the results of Morph-1B variants against other open pretrained models of similar size. The evaluation of large language models such as Open-LM-1B [63], OPT-1.3B [229], Pythia-1.3B [31], Neox-1.3B [33] and OPT-IML-1.3B [77] is summarized from [63].

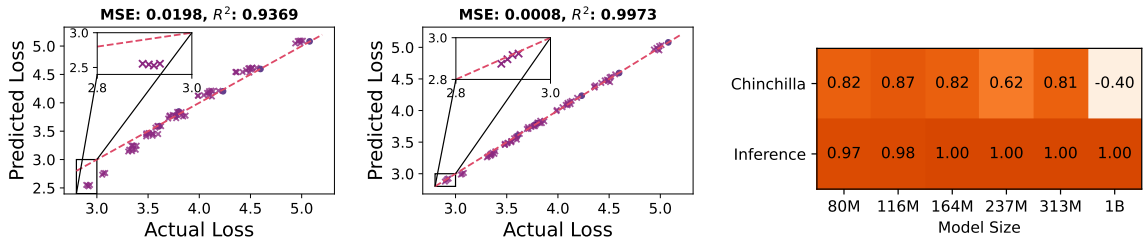
Models	d_{model}	n_{layers}	Avg.	Latency (s)
Open-LM-1B	2048	24	0.49	3.61
OPT-1.3B	2048	24	0.50	2.55
Pythia-1.3B	2048	22	0.49	3.28
Neox-1.3B	2048	24	0.49	3.99
OPT-IML-1.3B	2048	24	0.54	2.54
Morph-1B-v1	2048	24	0.52	3.61
Morph-1B-v2	2560	16	0.52	2.57
Morph-1B	3072	12	0.52	1.96



(a) Chinchilla **(b)** Inference-Efficient **(c)** Spearman Correlation

Figure 5.9: Excluding Over-training Data: We avoid using over-training data to fit the scaling laws. (Left) The figure is plotted by using Eq. (5.2). (Center) the center figure is created with Eq. (5.4). (Right) We plot the Spearman correlation of our scaling law versus the Chinchilla scaling law. The results indicate that additional training data can enhance the precision of scaling laws.

predicting the loss of 1B models. The results are shown in Figure 5.8c. The results indicate that our inference-efficient law is more effective in ranking different model configurations. For example, the inference-efficient scaling law shows a Spearman correlation of 1.00 for the 1B model loss prediction, in contrast to Chinchilla's -0.40. In Appendix B.1, we include more details on model configurations.



(a) Chinchilla **(b)** Chinchilla **(c)** Spearman Correlation

Figure 5.10: Random Choice of Model Shape: We randomly select the model shape to fit the scaling laws. (Left) The figure is plotted by using Eq. (5.2). (Center) The center figure is created with Eq. (5.4). (Right) We plot the Spearman correlation of our scaling law versus the Chinchilla scaling law. The results show that inference-efficient scaling laws are more robust than Chinchilla scaling laws.

5.4.2 Inference-Efficient Models

Guided by the accurate inference-efficient scaling law, we employ the predict, rank, and select method outlined in Figure 5.6 to train inference-efficient models. First, we generate a range of variants from the Open-LM-1B model [63] by adjusting the aspect ratio. Then, we measure the inference latency of model variants on a single A100 GPU. Next, we select 3 models based on the measured inference latency and predicted loss, and train candidate models with the same training dataset. Finally, we evaluate the trained models over 20 downstream tasks and we outline the results in Figure 1.2 and Table 5.4.

As a baseline, the architecture of Morph-1B-v1 is identical to that of Open-LM-1B. The superior performance of Morph-1B-v1 over Open-LM-1B can be attributed to the higher quality DCLM-Baseline dataset [3]. Additionally, OPT-IML-1.3B outperforms Morph-1B-v1 since it undergoes pre-training on 6$\times$ more unique tokens (180B vs 30B) followed by a fine-tuning stage [77]. Next, we train Morph-1B and Morph-1B-v2 which are derived from Morph-1B-v1 by modifying the aspect ratio. We use the same 30B tokens to train Morph-1B, Morph-1B-v1, and Morph-1B-v2. As illustrated in Table 5.4, the inference latency for Morph-1B-v1 is 1.8$\times$ lower compared to Morph-1B, without any loss in accuracy.

5.4.3 Insights from Scaling Laws Fitting

Scaling laws provide a cheap and accurate way to predict language model performance at larger scales. However, a drawback of building scaling laws is the requirement to train

models at various scales. In this section, we study how to make scaling laws robust and data-efficient.

Exclude Over-training Data. In this ablation study, we fit the scaling law based entirely on the Chinchilla-optimal setup, using only data points where training tokens are set to be Chinchilla-optimal. We vary the model size $N \in \{80, 116, 164, 237, 313\}$M and set the number of training tokens $D = 20N$, excluding data from $N = 80$M and $D = 160N$. Table 5.3 shows the configurations we run on and the results are shown in Figure 5.9. Compared to Figure 5.8, we observe that the inference-efficient scaling law is more robust than the Chinchilla scaling law. We achieve a much lower MSE of 0.1165 compared to Chinchilla's 0.9825 and an R^2 score of 0.6293 compared to Chinchilla's -2.1259. However, we note that both scaling laws' performance deteriorates when applied to predicting losses in over-trained models. Therefore, data from over-training is essential to fit our inference-aware scaling law.

Select Model Shape Randomly. In this ablation study, we explore the robustness of our scaling laws via fitting models with random model architecture configurations. In this setting, the model architecture configuration for each size is chosen randomly. We randomly select a configuration from our model configuration pools (The complete list of candidate configurations can be found in Table B.1 in the Appendix). Figure 5.10 shows the experiment results. Compared to Chinchilla scaling laws, our inference-efficient scaling laws exhibit greater robustness with much smaller MSE (0.0008 vs 0.0198) and higher R^2 value (0.9973 vs 0.9369). We then use these two laws to predict the loss of 1B models. The results show that the relative prediction error for the inference-efficient scaling law is less than 0.72%, significantly lower than the Chinchilla scaling law's relative prediction error, which ranges from 11.8% to 13.4%. Finally, by using only six data points to fit the two scaling laws, we significantly reduce the training costs associated with developing these laws. The GPU hours for fitting have been reduced from 450 to 85 A100 GPU hours.

5.5 Conclusion

In this chapter, we perform an extensive empirical study to develop scaling laws that guide us in designing inference-efficient model architecture. We first demonstrate that model architecture impacts inference efficiency and that existing scaling laws do not account for

inference costs. To jointly optimize inference cost and model loss, we propose inference-efficient scaling laws. We conduct count number, each point is a number experiments to fit and evaluate the inference-efficient scaling laws. To tackle the disparity between model loss and downstream task performance, we have developed a novel methodology to train and rank inference-efficient models using our scaling law. Finally, we design and train Morph-1B model by leveraging inference-efficient scaling law, which enhances inference efficiency while maintaining accuracy in downstream tasks, compared to similar-sized open-sourced models.

Chapter 6

Scaling Laws Meet Model Architecture

This chapter presents novel methods for incorporating architectural factors into existing scaling laws. We first show that aspect ratio is not the only architectural factor affecting inference efficiency, as shown in Figure 6.1. Next, we study how various architectural factors influence inference efficiency and downstream-task accuracy in §6.2. Based on these observations, we propose a two-step conditional approach that integrates architectural factors into existing scaling laws in §6.2. Then, we fit the proposed scaling law to models ranging from 80M to 297M parameters and test its predictions when scaling up to pretraining 3B-parameter models. As shown in §6.3 and §6.4, under identical training setups, our optimal 3B-parameter architecture delivers up to 42% higher inference throughput than LLaMA-3.2-3B, while maintaining better accuracy.

6.1 Preliminaries

Accurately predicting the performance of large language models during scaling is essential. This enables us to answer key questions: (i) what is the optimal allocation of available resources between model size and training tokens, and (ii) what performance gains can be expected from additional resources? Fortunately, the model loss has been observed to follow a power-law relationship with respect to the number of parameters N and training tokens D [70, 125] with:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \tag{6.1}$$

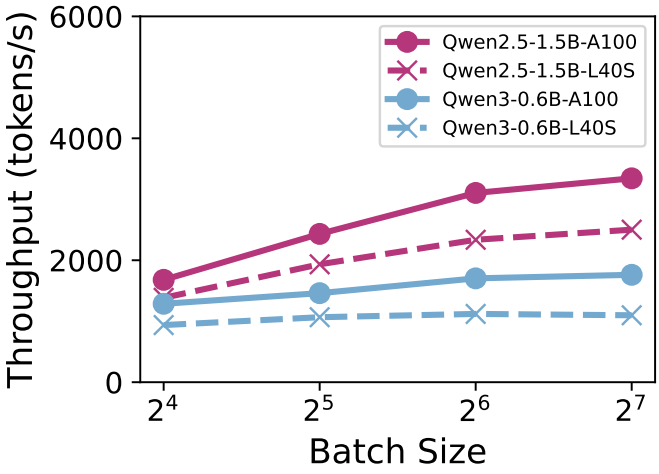


Figure 6.1: Although larger models generally achieve lower inference throughput than smaller ones, Qwen2.5-1.5B outperforms Qwen3-0.6B. Despite having the same number of layers, Qwen2.5-1.5B benefits from a higher hidden size, GQA, and mlp-to-attention ratio.

where L is the model loss, N is the number of total parameters and D is the number of tokens used for training and A, B, E, α, β are parameters to be learned.

To fit the learnable parameters in Eq. (6.1), Chinchilla [70] employs two strategies: (i) training models with a fixed number of parameters while varying the number of training tokens, and (ii) training models under a fixed compute budget[1], varying both parameters and tokens. The resulting data are combined to fit the learned parameters in Eq. (6.1). With the fitted scaling laws, Chinchilla addresses the following question to determine optimal allocation:

$$\arg\min_{N,D} L(N,D) \text{ s.t. } \text{FLOPs}(N,D) = C \tag{6.2}$$

where C denotes the resource constraint, N the total number of parameters, and D the number of training tokens.

In this paper, we do not address how to optimally allocate compute between model size and training data under a fixed compute budget. Instead, our focus is on identifying model

[1]The compute cost is approximated as $\text{FLOPs}(N,D) \approx 6ND$ in [70, 125], where N denotes the number of parameters and D the number of training tokens. In this work, we adopt the same settings as prior studies.

architectures that optimize inference efficiency and accuracy under fixed parameter and token budgets. For example, given a model with 7B parameters trained on 14T tokens, we study how to design an architecture that satisfies both efficiency and accuracy requirements.

6.2 Model Architecture-Aware Scaling Laws

6.2.1 Model Architecture Variations

The architecture of a decoder-only transformer is composed of a sequence of stacked decoder blocks, each sharing the same structure to facilitate model-parallel deployment across devices. Under this design, the overall architecture of dense LLMs is primarily determined by the hidden size and the MLP intermediate size, which together specify the attention and MLP layers structure. This work studies the optimal model architecture given a fixed total number of non-embedding parameters $N_{\text{non-embed}}$ (at different levels). Although the number of layers n_{layer} also plays a critical role (closely related to aspect ratio [142]), varying n_{layer} under a fixed $N_{\text{non-embed}}$ substantially impacts both inference cost and accuracy [20, 185]. Therefore, we fix n_{layer} and focus on the effects of hidden size d_{model} and the mlp-to-attention ratio $r_{\text{mlp/attn}}$ on inference efficiency in §6.2.2 and accuracy in §6.2.3, noting that n_{layer} still varies across different $N_{\text{non-embed}}$ levels. In §6.2.3, we introduce a conditional scaling law to predict the performance of architectural variants, and in §6.2.4, we present a lightweight framework for identifying architectures that optimally balance inference efficiency and accuracy.

Note that the number of attention parameters is primarily determined by the hidden size d_{model} and the attention projection dimension, since most open-weight models adopt non-square q, k, v projection matrices, as seen in Gemma [187] and Qwen3 [216]. For consistency, we fix the per-head dimension d_{head} to 64 for models with $N_{\text{non-embed}} \leqslant$1B and to 128 for models with $N_{\text{non-embed}} \geqslant$3B. Consequently, to maintain a constant $r_{\text{mlp/attn}}$, we adjust the number of attention heads n_{head} rather than altering the projection dimension directly. This design choice also provides flexibility to incorporate architectural variants such as grouped-query attention.

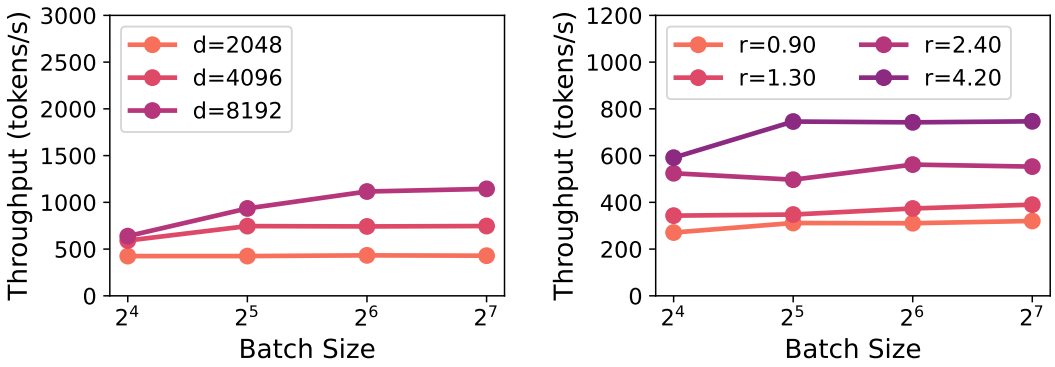


Figure 6.2: Inference throughput. (left) hidden size $d = d_{\text{model}}$ and (right) mlp-to-attention ratio $r = r_{\text{mlp/attn}}$ on the 8B model. Under a fixed parameter budget $N_{\text{non-embed}}$, larger hidden sizes and higher mlp-to-attention ratios improve inference throughput for varying batch sizes.

6.2.2 Inference Efficiency

Inspired by the success and widespread adoption of open-weight dense models such as Qwen3 [216], LLaMA-3.2 [56], and Gemma-2 [188] family, we construct architectural variants by modifying the configurations of the LLaMA-3.2 and Qwen3 dense models. The results are shown in Figure 6.3-6.8. In addition to hidden size and the mlp-to-attention ratio, we find that group-query attention has a critical impact on inference efficiency, even though it only modestly reduces the number of attention parameters (by shrinking the key and value matrices). To disentangle these effects, we conduct controlled ablations of hidden size, MLP-to-attention ratio, and GQA under the following setups:

- *hidden size* d_{model}*:* fix $N_{\text{non-embed}}$, $r_{\text{mlp/attn}}$ and GQA$= 4$, vary d_{model} and number of attention heads n_{head} (Figure 6.2 left, Figure 6.3, and Figure 6.6).
- *mlp-to-attention ratio* $r_{mlp/attn}$*:* fix $N_{\text{non-embed}}$, d_{model} and GQA$= 4$, vary n_{head} and intermediate size (Figure 6.2 right, Figure 6.4, and Figure 6.7).
- *GQA:* fix $N_{\text{non-embed}}$, d_{model} and $r_{\text{mlp/attn}}$, vary n_{head} and number of key-value heads (Figure 6.5, and Figure 6.8).

Figure 6.2 shows the ablation of varying hidden sizes d_{model} and mlp-to-attention $r_{\text{mlp/attn}}$ on the LLaMA-3.1-8B model variants. We observe that larger hidden size (or fewer attention heads) and higher mlp-to-attention ratios improve inference throughput. Similar trends are

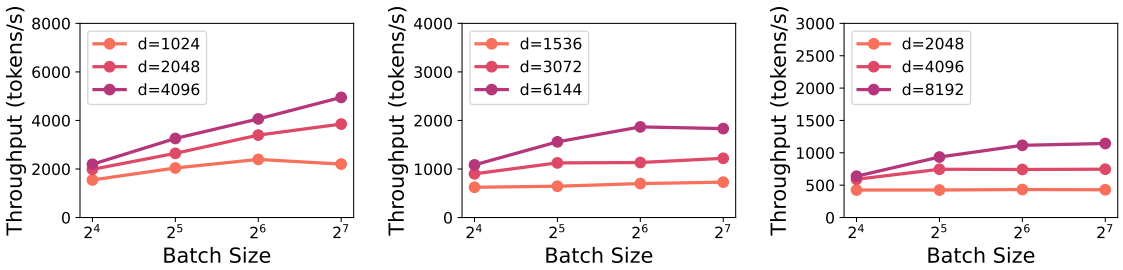


Figure 6.3: Hidden size on Inference Throughput: (left) 1B model variants; (center) 3B model variants; (right) 8B model variants. Across varying batch sizes and model scales, larger hidden sizes yield higher inference throughput under a fixed parameter budget. The legend indicates the hidden size of the models, where $d = d_{\text{model}}$.

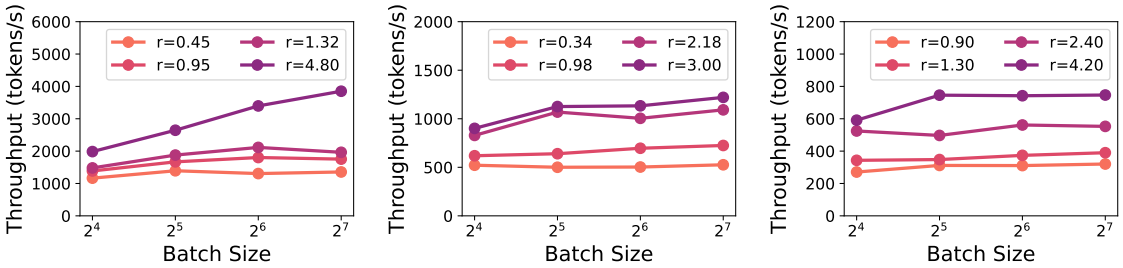


Figure 6.4: MLP-to-Attention ratio on Inference Throughput: (left) 1B model variants; (center) 3B model variants; (right) 8B model variants. Across varying batch sizes and model scales, a larger MLP-to-Attention ratio increases inference throughput under a fixed parameter budget. The legend indicates the MLP-to-Attention ratio of the models, where $r = r_{\text{mlp/attn}}$.

observed in the LLaMA-3.2-1B and 3B model variants (Figure 6.3 and Figure 6.4). These gains arise in part because larger d_{model} and higher $r_{\text{mlp/attn}}$ reduce the total FLOPs, as detailed in the inference FLOPs analysis (Appendix C.2). In addition, these architectural choices shrink the KV cache, lowering I/O cost during inference and further improving throughput [11]. Figure 6.5 presents the GQA ablation, confirming prior observations [18] that increasing GQA consistently improves inference throughput. A comparable set of ablation experiments on Qwen3 models, reported in Figure 6.6-6.8, further corroborates these findings.

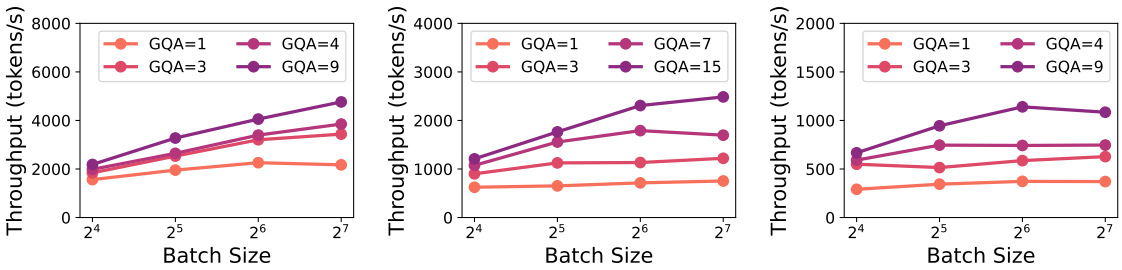


Figure 6.5: GQA on Inference Throughput: (left) 1B model variants; (center) 3B model variants; (right) 8B model variants. This figure shows the impact of GQA on inference throughput. With the total parameter count fixed, hidden size is set to 2048 (1B), 3072 (3B), and 4096 (8B), and the MLP-to-Attention ratio is 4.0, 2.67, and 4.2, respectively. Across varying batch sizes, models with larger GQA achieve higher throughput. All evaluations are performed using the vLLM framework [97] on a single NVIDIA Ampere 40GB A100 GPU with 4096 input and 1024 output tokens.

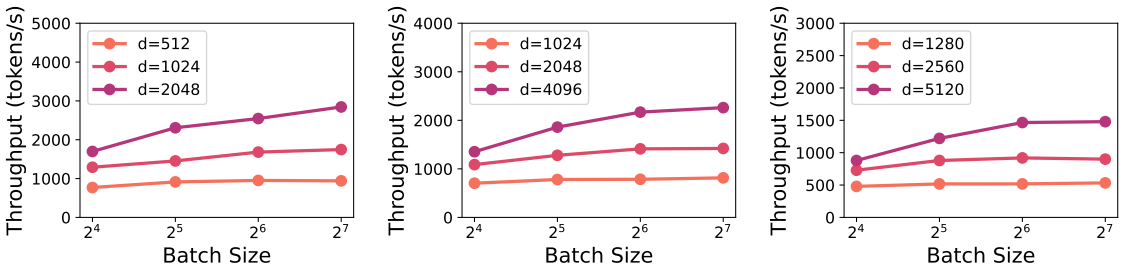


Figure 6.6: Hidden size on Inference Throughput (Qwen3): (left) Qwen3-0.6B model variants; (center) Qwen3-1.7B model variants; (right) Qwen3-4B model variants. Across varying batch sizes and model scales, larger hidden sizes yield higher inference throughput under a fixed parameter budget. The legend indicates the hidden size of the models, where $d = d_{model}$. All evaluations are performed using the vLLM framework [97] on a single NVIDIA Ampere 40GB A100 GPU with 4096 input and 1024 output tokens.

6.2.3 A Conditional Scaling Law

Improving inference efficiency should not come at the expense of significantly reducing model accuracy, making it crucial to understand how architectural choices affect accuracy and training loss. Because training large-scale language models is prohibitively expensive, a common strategy is to study smaller models and use scaling laws to extrapolate insights to larger scales, for example, the Chinchilla scaling laws [70]. However, incorporating multiple

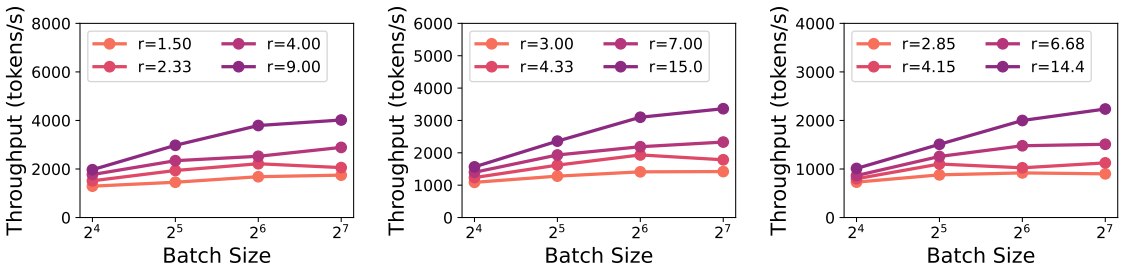


Figure 6.7: MLP-to-Attention ratio on Inference Throughput (Qwen3): (left) Qwen3-0.6B model variants; (center) Qwen3-1.7B model variants; (right) Qwen3-4B model variants. Across varying batch sizes and model scales, a larger MLP-to-Attention ratio increases inference throughput under a fixed parameter budget. The legend indicates the MLP-to-Attention ratio of the models, where $r = r_{mlp/attn}$. All evaluations are performed using the vLLM framework [97] on a single NVIDIA Ampere 40GB A100 GPU with 4096 input and 1024 output tokens.

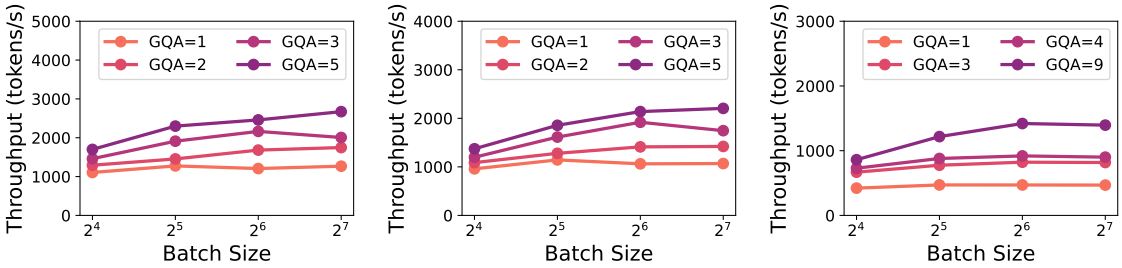


Figure 6.8: GQA on Inference Throughput (Qwen3): (left) Qwen3-0.6B model variants; (center) Qwen3-1.7B model variants; (right) Qwen3-4B model variants. This figure shows the impact of GQA on inference throughput. With the total parameter count fixed, hidden size is set to 1024 (0.6B), 2048 (1.7B), and 2560 (4B), and the MLP-to-Attention ratio is 1.5, 3.0, and 2.85, respectively. Across varying batch sizes, models with larger GQA achieve higher throughput. All evaluations are performed using the vLLM framework [97] on a single NVIDIA Ampere 40GB A100 GPU with 4096 input and 1024 output tokens.

architectural factors into such laws remains challenging. To address this, we examine the effect of architectural choices on training loss L in a conditional manner, varying one factor at a time while keeping the others fixed.

hidden size d_{model}. We note that d_{model} generally scales linearly with $\sqrt{N_{non\text{-}embed}}$. Assuming squared attention weight matrices, the number of attention parameters N_{attn} can

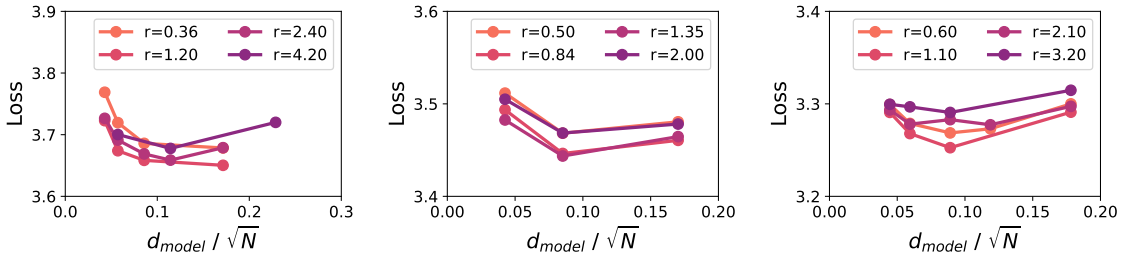


Figure 6.9: Loss vs. hidden size. (Left) 80M model variants; (Center) 145M model variants; (Right) 297M model variants. Across model sizes, the relationship between training loss and $d_{model}/\sqrt{N}$ exhibits a consistent U-shaped curve when architectural factors such as GQA and the MLP-to-attention ratio are held fixed. The legend denotes the MLP-to-attention ratio $r = r_{mlp/attn}$ for each model.

be expressed as

$$4d_{model}^2 \propto N_{attn} = N_{non\text{-}embed} \times \frac{r}{r+1},$$

where $r = r_{mlp/attn}$ is fixed, and the constant factor 4 arises from the query, key, value, and output projection layers in each attention block. To capture this scaling behavior, we normalize d_{model} by $\sqrt{N_{non\text{-}embed}}$ and examine its relation to loss L in Figure 6.9. The resulting U-shaped curves $L(d/\sqrt{N} \mid r, N, D)$ exhibit nearly identical optima across different model sizes. Moreover, Figure 6.9 confirms that excessively large hidden sizes, which reduce the number of attention heads n_{head}, can degrade accuracy, a phenomenon consistently observed in prior analyses of transformer capacity and head allocation [70, 85].

mlp-to-attention ratio $r_{mlp/attn}$. Figure 6.10 illustrates how the loss varies with $r_{mlp/attn}$, conditioned on d_{model} fixed at different levels, where we consistently observe a U-shaped curve $L(r \mid d/\sqrt{N}, N, D)$. While the attention mechanism is central to the success of transformers [192], recent open-weight models have allocated a progressively smaller fraction of parameters to attention as overall model size increases (e.g., LLaMA and Qwen families). Our analysis indicates that this trend is not universally optimal: there exists an interior optimum in the allocation of attention parameters, and deviating from it in either direction degrades model performance. This suggests that careful tuning of the mlp-to-attention ratio is critical for scaling transformers effectively.

As shown in Figures 6.9 and Figure 6.10, both hidden size and the MLP-to-attention ratio

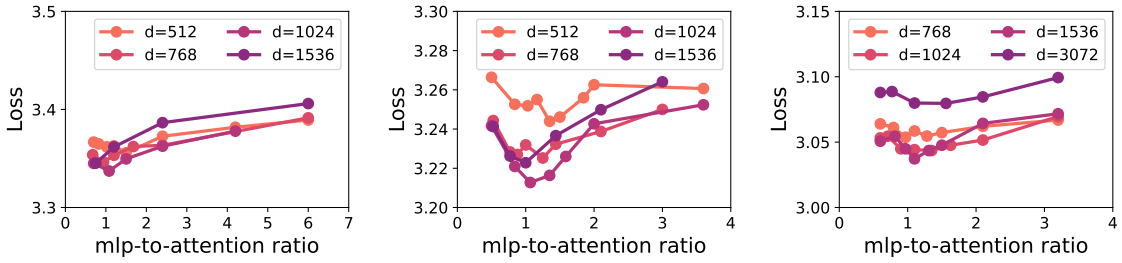


Figure 6.10: Loss vs. MLP-to-attention ratio. (Left) 80M model variants; (Center) 145M model variants; (Right) 297M model variants. Across model sizes, the relationship between training loss and $r_{\text{mlp/attn}}$ exhibits a consistent U-shaped curve when architectural factors such as GQA and hidden size are held fixed. The legend denotes the hidden size $d = d_{\text{model}}$ for each model.

exhibit U-shaped relationships with training loss. To capture these trends, we fit the function $c_0 + c_1 \log x + c_2/x$ separately for $x = r_{\text{mlp/attn}}$ and $d_{\text{model}}/\sqrt{N_{\text{non-embed}}}$. This formulation effectively models the U-shaped behavior while ensuring sublinear growth as x increases. However, incorporating $r_{\text{mlp/attn}}$, d_{model}, N, and D into a unified, architecture-aware scaling law remains challenging. Since fitting a single all-purpose scaling law $L(d/\sqrt{N}, r, N, D)$ is unrealistic across all possible configurations, we instead propose a two-step conditional approach:

1. For given N and D, obtain the optimal loss $L_{\text{opt}}(N, D) = \min L(N, D) = \min\left(E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}\right)$ from the Chinchilla scaling law, which is shown in Eq. (6.1), as a reference point.
2. Calibrate the loss of architectural variants $L(d/\sqrt{N}, r \mid N, D)$ relative to this reference.

We focus on two simple and transparent calibration schemes:

- (multiplicative)

$$L(d/\sqrt{N}, r \mid N, D) = (a_0 + a_1 \log(\frac{d}{\sqrt{N}}) + a_2 \frac{\sqrt{N}}{d}) \cdot (b_0 + b_1 \log r + \frac{b_2}{r}) \cdot L_{\text{opt}} \quad (6.3)$$

- (additive) $L(d/\sqrt{N}, r \mid N, D) = (a_0 + a_1 \log(\frac{d}{\sqrt{N}}) + a_2 \frac{\sqrt{N}}{d}) + (b_1 \log r + \frac{b_2}{r}) + L_{\text{opt}}$

Here, a_i and b_i are learnable parameters that are shared across all N, D. Note that both functional forms assume the effects of $r_{\text{mlp/attn}}$ and d_{model} on loss are separable.

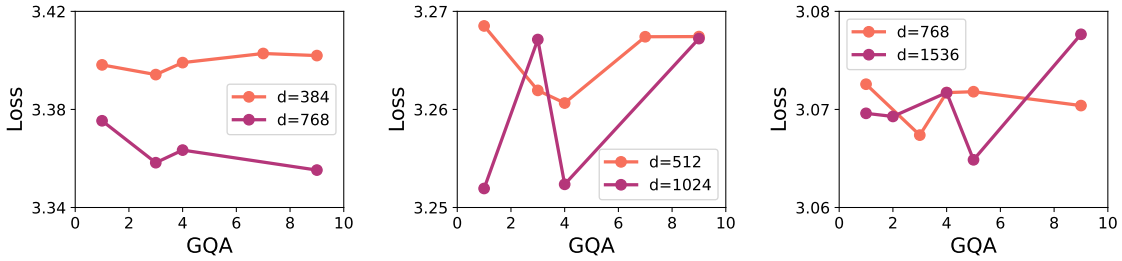


Figure 6.11: Loss vs. GQA: (left) 80M model variants; (center) 145M model variants; (right) 297M model variants. Across different model sizes, the relationship between training loss and GQA varies substantially when hidden size and the mlp-to-attention ratio are fixed. The legend denotes the hidden size of each trained model.

6.2.4 Searching for Inference-Efficient Accurate Models

With the conditional scaling law, we can identify architectures that are both inference-efficient and accurate by solving the following optimization problem: given N, D, and a set of architectural choices P,

$$\text{argmax}_P I_N(P), \qquad \text{s.t.} \quad L(P \mid N, D) \leqslant L_t, \tag{6.4}$$

where $I_N(P)$ denotes the inference efficiency of an architecture P with total $N_{\text{non-embed}}$ parameters, and $L_t, (\geqslant L_{\text{opt}})$ is the maximum allowable training loss.

As shown in Figure 6.5, GQA has a substantial impact on inference efficiency; However, unlike hidden size and the mlp-to-attention ratio, GQA does not exhibit a consistent continuous relationship with loss, which is shown in Figure 6.11, and is highly variable, making it challenging to identify settings that achieve both accuracy and efficiency. Fortunately, the search space for GQA is relatively small once $N_{\text{non-embed}}$, d_{model}, and $r_{\text{mlp/attn}}$ are fixed, since GQA must be a prime factor of the number of attention heads n_{head}. In practice, we perform a local GQA search by enumerating feasible values and applying early stopping once performance falls below that of the GQA$=4$ baseline. Algorithm 5 summarizes our overall framework for identifying inference-efficient and accurate architectures.

Algorithm 5: Searching for Inference-Efficient Accurate Model

Input: Model parameters N, training tokens D, target loss L_t; inference efficiency $I_N(\cdot)$; optional: the optimal loss $L_{opt}(N, D)$

1 Train smaller models to fit the Chinchilla scaling laws, as shown in Eq. (6.1), if $L_{opt}(N, D)$ is unavailable

2 Solve the constrained optimization, as shown in Eq. (6.4), for d_{model}, $r_{mlp/attn}$ and corresponding architecture P

3 Perform a local search over GQA values with early stopping to maximize inference efficiency

4 **return** Final model architecture $\{P, GQA\}$

6.3 Experiment Setup

We first detail the experimental setup of training, inference, and downstream task evaluation, and then describe how we derive the conditional scaling law and scale up to larger sizes.

Training Setup. We sample the training data from Dolma-v1.7 [178], which contains data from 15 different sources. Tokens are sampled with probability proportional to each source's contribution, ensuring the sampled dataset preserves a similar distribution to Dolma-v1.7. We train decoder-only LLaMA-3.2 [56] style transformers with $N_{non\text{-}embed}$ in $\{80M, 145M, 297M, 1B, 3B\}$, for each $N_{non\text{-}embed}$, we obtain model architecture candidates by varying hidden size $d_{model}/\sqrt{N_{non\text{-}embed}}$ and mlp-to-attention ratio $r_{mlp/attn}$. (changing intermediate size and number of attention heads n_{head}) while holding other architectural factors fixed, e.g., GQA$=4$. A full list of over 200 model architectures used can be found in Appendix C.1.

All models are trained on $100N_{non\text{-}emb}$ tokens ($5\times$ Chinchilla optimal) to ensure convergence. We tuned training hyper-parameters (mainly following prior work [39]), with a full list in Table 6.1.

Inference Setup. We evaluate the inference efficiency using the vLLM framework [97]. By default, inputs consist of 4096 tokens and outputs of 1024 tokens. We report the averaged inference throughput (tokens/second) from 5 repeated runs. Unless otherwise specified, all experiments are conducted on NVIDIA Ampere A100 GPUs (40GB) with vLLM.

Table 6.1: Hyper-parameters: We show the hyper-parameters used for training in this paper.

Model Size	80M	145M	297M	1B	3B
Batch Size	256	256	512	512	512
Max LR	1.5e-3	1.0e-3	8.0e-4	6.0e-4	6.0e-4
Min LR	$0.1\times$ Max LR				
Optimizer	AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$)				
Weight Decay	0.1				
Clip Grad Norm	1.0				
LR Schedule	Cosine				
Warmup Steps	500				
Sequence Length	2048				

LLM Evaluation Setup. Following prior works [31, 228], we evaluate pretrained models in the zero-shot setting using `lm-evaluation-harness`[2] on nine benchmarks: ARC-Easy [43], ARC-Challenge [43], LAMBADA [138], HellaSwag [226], OpenBookQA [121], PIQA [32], SciQ [204], WinoGrande [160], and CoQA [155].

Fitting Scaling Laws. Following [29, 58], we use the Levenberg-Marquardt algorithm to fit the conditional scaling laws, as shown in Eq. (6.3). The Levenberg–Marquardt algorithm does least-squares curve fitting by estimating $\hat{\beta}$ as the solution to $\arg\min_\beta \sum_{i=1}^{m} [y_i - f(x_i, \beta)]^2$, where (x_i, y_i) are the observed data pairs. Note that instead of fitting the Chinchilla scaling law, we empirically searched over architecture variants to find the optimal loss $L_{\text{opt}}(N, D)$ for $N_{\text{non-embed}}$ <1B scale.

We scale up the scale law fitting in the following progressive manner:

(Task 1) fit on the 80M results and evaluate on 145M results;

(Task 2) fit on 80, 145M results and evaluate on 297M results;

(Task 3) fit on 80, 145, 297M results and evaluate on 1B results;

This ensures a robust and consistent way of scaling up the model sizes and evaluating our conditional scaling law. Following prior work [96], we evaluate the fitted scaling law with mean squared error (MSE) metric, defined as $\frac{1}{n}\sum_{i=1}^{n}(l_i - \hat{l}_i)^2$ where l_i denotes the

[2] `https://github.com/EleutherAI/lm-evaluation-harness`

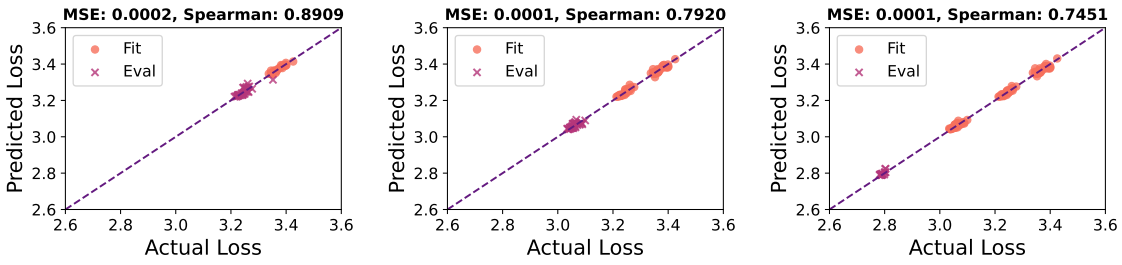


Figure 6.12: Predictive performances of the fitted conditional scaling law on: (left) Task 1: Fit on 80M, evaluate on 145M; (center) Task 2: Fit on 80, 145M, evaluate on 297M; (right) Task 3: Fit on 80, 145, 297M, evaluate on 1B. Orange dots denote fitting data points, and purple crosses indicate the test data points. We compare scaling-law predicted loss with actual pretraining loss of architectures and observed a consistently low MSE and high Spearman correlation across model scales.

actual loss and $\hat{l}_i$ the predicted loss. We additionally report the Spearman's rank correlation coefficient [181] to compare predicted and actual rankings. Both metrics are calculated on the val data points.

6.4 Experiment Results

We begin by evaluating the predictive performances of the conditional scaling laws with multiplicative calibration. We then conduct ablation studies to assess the impact of data selection and to evaluate the performance of the scaling laws under additive calibration. Finally, we apply the fitted scaling laws to guide the training of large-scale models following the search framework in §6.4.1.

Predictive Accuracy. As Task 1-3 described in §6.3, we fit the conditional scaling laws on 80M, (80M, 145M), and (80M, 145M, 297M) loss-architecture data points, and subsequently evaluate on 145M, 297M, and 1B data, respectively. In Figure 6.12, the low MSE and high Spearman correlation in tasks across different model scales validate the effectiveness and strong predictive performance of the proposed conditional scaling laws.

Ablation of Outliers. The mlp-to-attention ratio $r_{\text{mlp/attn}}$ of open-weights models typically fall between 0.5 and 5, for example, the mlp-to-attention ratio for LLaMA-3.2-1B,

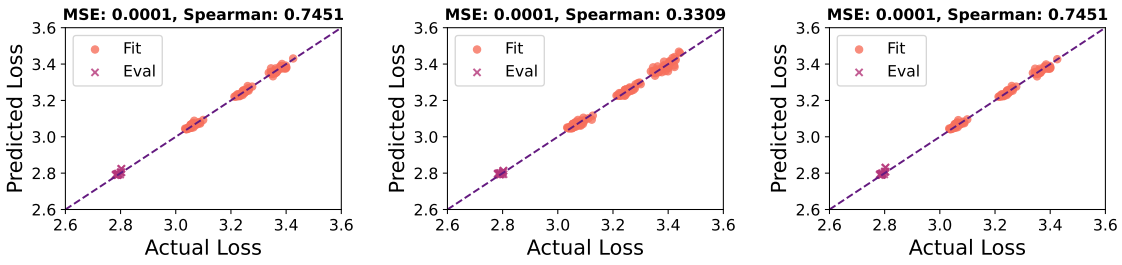


Figure 6.13: Ablation Study: (left) use multiplicative calibrations without outliers; (center) use multiplicative calibrations with outliers; (right) use additive calibrations without outliers. The outlier refers to models trained with an mlp-to-attention ratio below 0.5 or above 5. We observe that outlier data points harm the scaling law fit. Moreover, while multiplicative and additive calibrations differ in formulation, their MSE and Spearman values remain nearly identical. Dots denote the data points used for fitting, while crosses indicate the test data points.

LLaMA-3.2-3B, and Qwen3-8B are 4.81, 1.5, and 4.67, respectively. In Figure 6.12, we fit the conditional scaling law using only model architectures with $r_{\text{mlp/attn}} \in [0.5, 5]$. We ablate this choice by training model architectures with outlier $r_{\text{mlp/attn}}$ below 0.5 and above 5 (such as $0.1, 12.6$) in Appendix C.1. In Figure 6.13 left and Figure 6.13 center, we show on Task 3 a comparison of fitting the conditional scaling law without and with these outliers (with a clear Spearman correlation score degradation), which suggests to exclude extreme outliers for better predicted performances.

Ablation of Calibration. In Figure 6.13 right, we ablate an alternative formulation of the scaling laws with additive calibration, as discussed in §6.2.3. The results on Task 3 show that multiplicative and additive calibrations achieve similar MSE and Spearman correlations. Note that, unlike the conventional unified formulation, both calibrations assume that the effects of $r_{\text{mlp/attn}}$ and d_{model} on loss are separable. We further ablate more complex joint, non-separable formulations in Figure 6.14 and find that they do not provide superior predictive performance. The two-step reference-and-calibration framework appears robust enough that simple calibrations perform well.

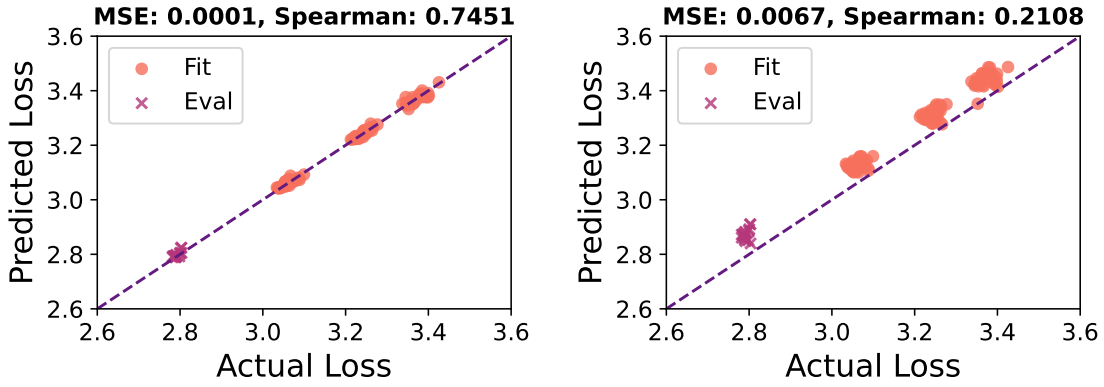


Figure 6.14: Joint and non-separable calibrations: (left) use multiplicative calibrations; (right) use joint and non-separable calibrations. We observe that joint and non-separable calibrations yield higher MSE and lower Spearman scores than multiplicative calibrations, indicating inferior performance. Dots denote the data points used for fitting, while crosses indicate the test data points.

Table 6.2: Large-Scale Model Results. We evaluate the scaling laws at 1B and 3B scales by training Panda-1B, Surefire-1B, and Panda-3B, and compare them with LLaMA-3.2-1B and LLaMA-3.2-3B, respectively. The Avg. column reports the mean accuracy across the nine downstream tasks. Panda-1B and 3B are trained using the optimal architectural configurations predicted by our scaling laws, whereas Surefire-1B and 3B satisfy the loss constraint in Eq. (6.4) and achieve Pareto optimality.

Models	d_{model}	f_{size}	n_{layers}	GQA	$d_{model}/\sqrt{N}$	r	Loss (↓)	Avg. (↑)
LLaMA-3.2-1B	2048	8192	16	4	0.066	4.80	2.803	54.9
Panda-1B	2560	4096	16	4	0.082	1.07	2.782	57.0
Surefire-1B	2560	6144	16	9	0.082	3.6	2.804	55.4
LLaMA-3.2-3B	3072	8192	28	3	0.058	4.80	2.625	61.9
Panda-3B	4096	4096	28	3	0.077	1	2.619	62.5
Surefire-3B	4096	4096	28	7	0.077	1	2.620	62.6

6.4.1 Optimal Model Architecture

Validating the conditional scaling law. We validate the conditional scaling law at the 1B scale by applying multiplicative calibration on Task 3 using data from the (80M, 145M,

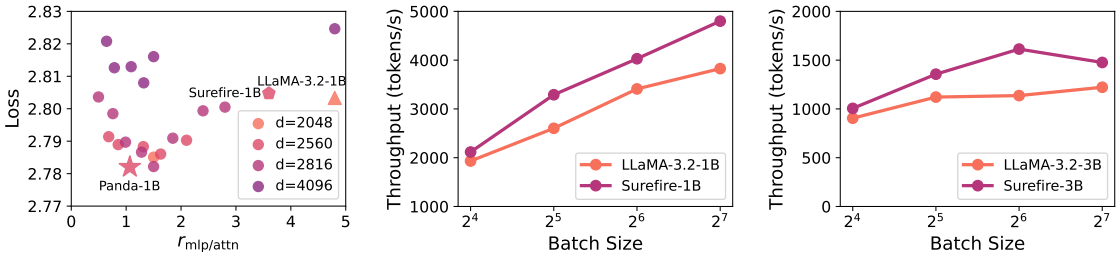


Figure 6.15: Results for 1B and 3B models. (Left) Panda-1B closely follows the scaling law predictions for minimizing training loss. (Center & Right) Inference throughput comparison between LLaMA-3.2 and Surefire models, where Surefire is consistently efficient across all batch sizes.

and 297M) model variants. The learned parameters are

$$a_0 = 2.697, a_1 = 0.0974, a_2 = 0.0078, b_0 = 0.3870, b_1 = 0.0063, \text{and } b_2 = 0.0065.$$

From this, we obtain the optimal architectural configuration of $d_{\text{model}}/\sqrt{N} = 0.08, r = 1.032$ for 1B model by solving $\frac{\partial L}{\partial d_{\text{model}}} = 0$ and $\frac{\partial L}{\partial r} = 0$. Using this configuration, we train a LLaMA-3.2-style 1B dense model on 100B tokens, denoted as Panda-1B. Panda-1B outperforms the open-weight LLaMA-3.2-1B baseline configs by 2.1% on average across downstream tasks, as shown in Table 6.2. Figure 6.15 left further confirms the effectiveness of the conditional scaling law by showing that Panda-1B achieves the lowest training loss among the exhaustively trained 1B variants under the same setup.

We also scale up our methodology to 3B models. Using the same approach but with data from the 80M, 145M, 297M, and 1B variants, we fit the scaling law and obtain $d_{\text{model}}/\sqrt{N} = 0.08$ and $r = 1.055$ for the Panda 3B model. Trained on 100B tokens, Panda-3B outperforms the open weight LLaMA-3.2-3B configuration by 0.6% on average across downstream tasks, as shown in Table 6.2.

With all components in place, we apply the search framework for inference-efficient and accurate models (Alg. 5). For the $N_{\text{non-embed}} = 1\text{B}$ and 3B setting trained on 100B tokens, we set the target loss L_t to match the training loss achieved by the LLaMA-3.2-1B and LLaMA-3.2-3B architectures, respectively.

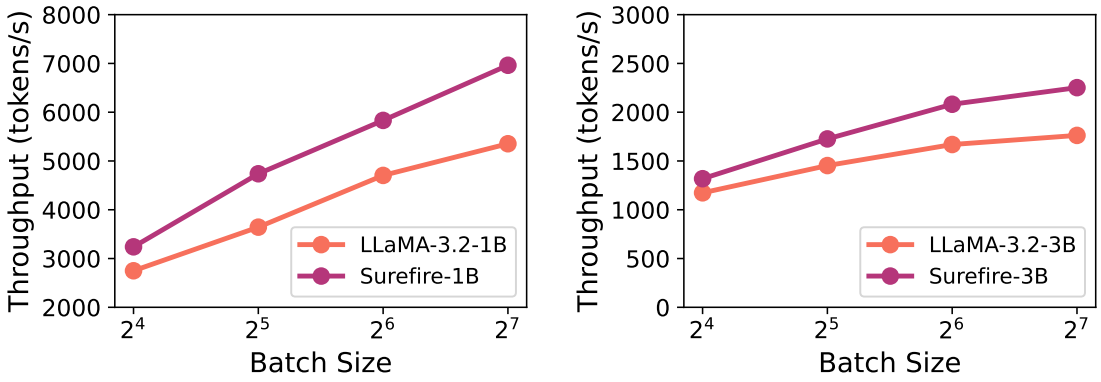


Figure 6.16: Results for 1B and 3B models over A100 GPU: (left) Inference throughput comparison between LLaMA-3.2-1B and Surefire-1B, showing that Surefire-1B consistently achieves higher efficiency across batch sizes. (right) Inference throughput comparison between LLaMA-3.2-3B and Surefire-3B, demonstrating that Surefire-3B consistently delivers higher efficiency across all batch sizes. The results are collected using the SGLang framework [234] on a single A100 GPU with 4096 input and 1024 output tokens.

Ablation of inference efficiency. Although inference efficiency $I_N(P)$ could, in principle, be expressed analytically, it depends heavily on hardware and inference configurations. Therefore, rather than solving for $I_N(P)$ directly, we search over feasible configurations P_i that satisfy the loss constraint on A100 with vLLM and select Pareto-optimal points, which we denote as Surefire-1B and Surefire-3B. Surefire-1B and Surefire-3B outperform LLaMA-3.2-1B and LLaMA-3.2-3B on downstream tasks, as shown in Table 6.2 with details in Appendix C.3, and deliver up to 42% higher inference throughput, as shown in Figure 6.15, center and right. We also ablate inference efficiency using SGLang [234] on A100 and NVIDIA H200 GPUs, as shown in Figure 6.16 and Figure 6.17. The results remain consistent with our vLLM-A100 evaluation: Surefire-1B and 3B outperform LLaMA-3.2-1B and 3B across all settings, achieving up to 47% higher throughput with SGLang on H200. This demonstrates that the efficiency gains transfer across serving stacks and hardware platforms. Detailed throughput statistics are provided in Table 6.3.

We further compare design choices across existing open-source models at the 1B and 3B scales in Table 6.5 and Table 6.6. For the LLaMA-3.2-1B, Panda-1B, and Surefire-1B models we pretrained, we report inference throughput (tokens/s), byte-level WikiText perplexity, and full architectural configurations in the accompanying tables. All throughput measurements

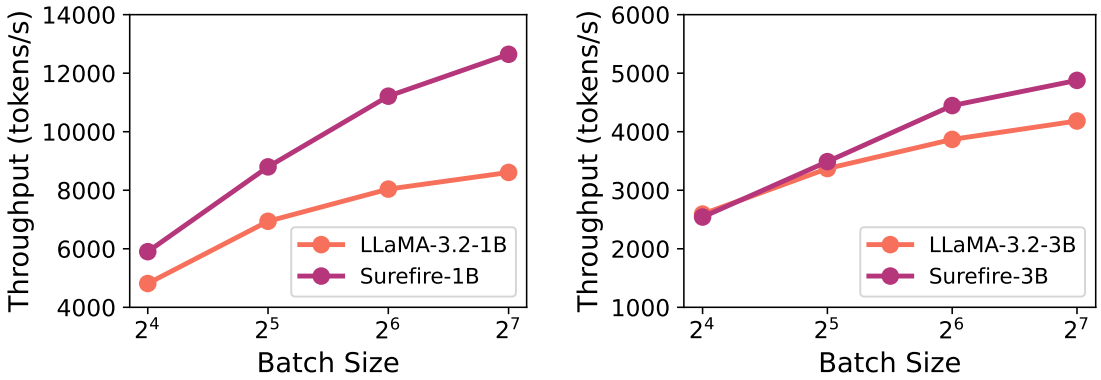


Figure 6.17: Results for 1B and 3B models over H200 GPU: (left) Inference throughput comparison between LLaMA-3.2-1B and Surefire-1B, showing that Surefire-1B consistently achieves higher efficiency across batch sizes. (right) Inference throughput comparison between LLaMA-3.2-3B and Surefire-3B, demonstrating that Surefire-3B consistently delivers higher efficiency across all batch sizes. The results are collected using the SGLang framework [234] on a single NVIDIA H200 GPU with 4096 input and 1024 output tokens.

are performed with vLLM on H200 GPUs using batch size 128. For the 1B scale, we include LLaMA-3.2-1B-HF and OLMo-2-1B-HF. Because OLMo supports only a 4k context window and cannot run our standard 4k/1k setup (4096 input tokens and 1024 output tokens), we additionally report results under a 2k/1k setup (2048 input tokens and 1024 output tokens). For the 3B scale, we add LLaMA-3.2-3B-HF and Qwen2.5-3B-HF, all evaluated under the 4k/1k configuration.

Our observations are as follows:

- OLMo-2-1B-HF is relatively close to our predicted optimal design, with an MLP-to-attention ratio of 3 (near our predicted 3.6), but remains inference-inefficient due to its hidden dimension and GQA choices.
- At the 3B scale, LLaMA-3.2-3B-HF achieves good accuracy but is not inference-efficient, while Qwen2.5-3B-HF is inference-efficient but less accurate.

These comparisons further underscore the necessity and relevance of our inference-efficient, high-accuracy model designs.

Table 6.3: Summary of Results for 1B and 3B Models: We summarize the inference throughput (tokens/s) of LLaMA-3.2-1B, Surefire-1B, LLaMA-3.2-3B, and Surefire-3B across vLLM and SGLang on A100 and H200 GPUs using 4096 input tokens and 1024 output tokens.

Hardware	Framework	Model	Batch Size			
			16	32	64	128
A100	vLLM	LLaMA-3.2-1B	1931.87	2602.72	3409.85	3825.91
		Surefire-1B	2116.49	3290.23	4028.69	4800.05
		LLaMA-3.2-3B	904.83	1121.39	1136.61	1222.03
		Surefire-3B	1005.44	1356.07	1613.32	1476.22
A100	SGLang	LLaMA-3.2-1B	2748.84	3643.27	4703.92	5353.29
		Surefire-1B	3239.55	4737.63	5832.01	6962.24
		LLaMA-3.2-3B	1173.51	1452.97	1668.67	1762.18
		Surefire-3B	1318.23	1726.20	2081.44	2251.74
H200	vLLM	LLaMA-3.2-1B	4311.97	6221.14	8131.65	9306.36
		Surefire-1B	4532.85	6992.71	9493.46	11282.56
		LLaMA-3.2-3B	2269.53	3119.94	3872.14	4311.43
		Surefire-3B	2309.48	3271.63	4242.33	4841.53
H200	SGLang	LLaMA-3.2-1B	4812.67	6939.88	8038.34	8608.57
		Surefire-1B	5900.52	8798.68	11214.40	12645.55
		LLaMA-3.2-3B	2593.04	3370.42	3868.42	4183.09
		Surefire-3B	2542.21	3488.79	4446.66	4877.16

Table 6.4: 3B Model Ablations. We assess the robustness of fitting-data strategy at 3B scale by training Panda-3B (using 80M, 145M, and 297M data) and Panda-3B$^\circ$ (using only on 1B data), and compare both with LLaMA-3.2-3B. Avg. denotes mean accuracy across nine downstream tasks.

Models	d_{model}	f_{size}	n_{layers}	GQA	$d_{model}/\sqrt{N}$	r	Loss (↓)	Avg. (↑)
LLaMA-3.2-3B	3072	8192	28	3	0.058	4.80	2.625	61.9
Panda-3B	4096	4096	28	3	0.077	1	2.619	62.5
Panda-3B$^\circ$	4096	4608	28	3	0.076	1.23	2.606	62.5

Ablation of fitting data strategy. While we adopt a progressive strategy for selecting fitting data across tasks in §6.3, results from small models (e.g., 80M) may not reliably predict behaviors at larger scales such as 3B. To assess this, we fit the conditional scaling

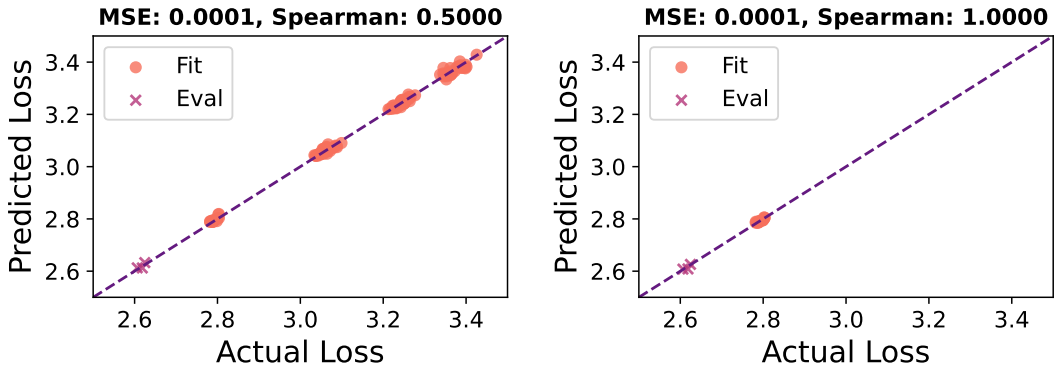


Figure 6.18: Effect of the Fitting Data Strategy on Predictive Performance. (left) Fit on 80M, 145M, 297M, 1B, evaluate on 3B; (right) Fit on 1B, evaluate on 3B. Orange dots denote fitting data, and purple crosses indicate the test data. We compare scaling-law predicted loss with actual pretraining loss of architectures and we observe that fitting the scaling laws with only 1B model data yields lower MSE and higher Spearman correlation for the 3B model loss prediction.

law for the 3B model using only the 1B variants. As shown in Figure 6.18, fitting with 1B data yields lower MSE and higher Spearman correlation when predicting 3B behavior, suggesting that the law's coefficients shift with model size. We therefore refit the law with multiplicative calibration using only the 1B variants, yielding the coefficients $a_0 = 2.319$, $a_1 = 0.238$, $a_2 = 0.0176$, $b_0 = 0.5104$, $b_1 = 0.0051$, and $b_2 = 0.0062$.

This produces an alternative optimal configuration for the 3B model, with $d_{\text{model}}/\sqrt{N} = 0.074$ and $r = 1.229$. We train a 3B model (Panda-3B$^\circ$) under this configuration on 100B tokens and compare it with both LLaMA-3.2-3B and Panda-3B (fitted from 80M, 145M, 297M, and 1B data). As shown in Table 6.4, Panda-3B$^\circ$ achieves a lower training loss and comparable downstream accuracy to Panda-3B, with detailed results given in Appendix C.3. These findings suggest that when scaling up, it is often sufficient, and sometimes preferable, to fit the law using models within a closer size range to the target, such as about one third of its scale.

6.5 Conclusion

This work explores the trade-off between model accuracy and inference cost under a fixed training budget. We begin by demonstrating how architectural choices influence both inference throughput and model accuracy. Building on this, we extend Chinchilla scaling laws to incorporate architectural factors and propose a two-step conditional framework for optimal architecture search: (i) train small models to fit the conditional scaling law, as shown in Eq. (6.3), and (ii) solve Eq. (6.4) for the predicted optimal architecture, followed by a local search over GQA to maximize inference efficiency. Using the fitted scaling laws and our framework, we trained models up to 3B parameters, achieving up to 42% higher inference throughput and 2.1% accuracy gains across nine downstream tasks. In Table 6.5 and Table 6.6, we compare design choices across existing open-source models at the 1B and 3B scales, further underscoring the need for our inference-efficient, accurate model designs.

Table 6.5: Comparison against open-source models at the 1B scale: We compare our pretrained LLaMA-3.2-1B, Panda-1B, and Surefire-1B models with LLaMA-3.2-1B-HF and OLMo-2-1B-HF in terms of inference throughput (on H200 GPUs using vLLM) and byte-level WikiText perplexity.

Model	LLaMA-3.2-1B	Panda-1B	Surefire-1B	LLaMA-3.2-1B-HF	OLMo-2-1B-HF
Wikitext PPL	1.7151	1.7016	1.7142	1.5807	1.5798
Tput (4k/1k)	9306	6218	11283	9306	/
Tput (2k/1k)	11948	8961	13890	11948	7486
Model Architectural Config					
n_{layers}	16	16	16	16	16
d_{model}	2048	2560	2560	2048	2048
$r_{\text{mlp/attn}}$	4.8	1.067	3.6	4.8	3
GQA	4	4	9	4	1
$N_{\text{non-embed}}$	973M	975M	965M	973M	1.074B

Table 6.6: Comparison against open-source models at the 3B scale: We compare our pretrained LLaMA-3.2-3B, Panda-3B, and Surefire-3B models with LLaMA-3.2-3B-HF and Qwen2.5-3B-HF in terms of inference throughput (on H200 GPUs using vLLM) and byte-level WikiText perplexity.

Model	LLaMA-3.2-3B	Panda-3B	Surefire-3B	LLaMA-3.2-3B-HF	Qwen2.5-3B-HF
Wikitext PPL	1.6489	1.6454	1.6462	1.5164	1.6185
Tput (4k/1k)	4311	3335	4842	4311	6470
Model Architectural Config					
n_{layers}	28	28	28	28	36
d_{model}	3072	4096	4096	3072	2048
$r_{\text{mlp/attn}}$	3	1	1	3	7.167
GQA	3	3	7	3	8
$N_{\text{non-embed}}$	2.82B	2.82B	2.82B	2.82B	2.77B

Chapter 7

Conclusion and Future Work

Scaling improves LLMs, but it also shifts the bottleneck. In training, scaling increases activation memory and cross-device transfers, making bandwidth, synchronization, and overlap decisive for throughput. Scaling introduces additional scheduling challenges, such as designing migration policies and identifying opportunities to co-locate jobs on GPU clusters. In deployment, repeated inference pushes per-token latency, throughput, and memory traffic to dominate cost. This thesis argues that architecture links training and deployment by shaping activation volume, parallel communication patterns, and per-token compute, affecting not only accuracy but also end-to-end efficiency.

We first show that modifying the model architecture to compress activations provides a practical mechanism for accelerating model-parallel training. By treating activation compression as part of the forward and backward computation graph rather than a purely systems-level trick, we identify regimes where it reduces communication overhead while maintaining model quality.

Second, as model sizes scale up, migration policies for round-based GPU cluster schedulers become critical to reducing migration costs. We also observe that better parallelism choices and GPU co-location can increase utilization. To address both issues, we formulate them as maximum-weight matching problems and integrate the resulting solutions into existing schedulers.

Furthermore, we argue that inference efficiency is fundamentally shaped by model architecture, not just serving-layer optimizations. Models with similar parameter counts can have very different inference latency because architectural choices govern per-token

compute and memory behavior, making model shape a first-class design variable. Building on this observation, we introduce an inference-aware design approach that jointly optimizes architectural shape and training for deployment constraints. We show that selecting architectures along the accuracy-latency Pareto frontier yields substantially faster inference without sacrificing downstream quality.

Finally, we emphasize that inference efficiency cannot be understood from training-budget trends alone, and model architecture must be part of the story. Traditional scaling laws predict how loss changes with more compute and data, but often omit the architectural degrees of freedom that dictate how expensive inference will be. We demonstrate that architecture strongly influences the quality-cost frontier: even under similar training budgets, models can differ widely in latency and cost because architecture sets per-token FLOPs, memory movement, and cache growth. Incorporating these architectural factors provides a practical basis for selecting model families that deliver strong capability at low deployment cost.

Overall, we conclude that system optimizations help, but their impact is limited when they don't shorten the critical path; architectural choices can shift the frontier by reshaping computation, memory movement, and communication from training through deployment.

7.1 Future Work

Next, we outline several promising directions for architecture research. We believe it's critical to prioritize designs that reduce training and inference cost and memory footprint while preserving quality at scale, including: (i) architecture decisions designed for distributed training that reduce synchronization overhead and improve multi-GPU scalability; (ii) extend our scaling-law results to larger model sizes, generalize these findings to Mixture-of-Experts (MoE) architectures [171], and expand the analysis to post-training stages; (iii) recent attention variants, including linear attention [86], sliding window attention [26], DeltaNet [218], and DuoAttention [211], raise an open question: how should scaling laws be extended to capture attention-design choices, and how can they guide hybrid-attention models that are efficient for both training and inference while preserving performance of downstream tasks?

7.2 Concluding Remarks

Scaling has delivered extraordinary capability gains, but it has also made efficiency the defining constraint of modern LLM development. This thesis maintains that the practical bounds of scaling are most accurately characterized by the dual challenges of training and inference, and that architecture provides the most direct means to extend both frontiers. In training, communication is not simply something to optimize after the fact; It is shaped by architectural partitioning and activation interfaces, and reducing it requires designs that account for system-level critical paths. As scale increases, scheduler migration costs rise, and architectural choices, especially parallelism and partitioning, determine cluster utilization. Jointly optimizing model architecture design, migration cost, and scheduling policy offers a clear path to improving overall GPU cluster utilization. In inference, efficiency must be treated as a primary scaling objective, where architecture and scaling behavior jointly determine quality per dollar and per millisecond. The broader conclusion is optimistic: the next stage of progress can come from turning compute into capability more effectively, through architectures and systems built for the constraints of real deployment.

Appendix A

Appendix: Architectures for Efficient Inference

A.1 More experimental results and takeaways

A.1.1 Experimental setup

System configuration. In order to measure the performance of compression algorithms over different hardware, we conduct our experiments on two different setups. The first setup uses AWS p3.8xlarge machines which have 4 Tesla V100 GPUs with all GPUs connected by NVLink. AWS p3.8xlarge instances have 10 Gbps network bandwidth across instances. Moreover, we also use a local machine which also has 4 Tesla V100 GPUs but does not have NVLink. All the GPUs are connected by a single PCIe bridge. The local server runs Ubuntu 18.04 LTS and the server has 125GB of memory.

Models. The models we use in this section are the same as what we mentioned in §3.3.1.

Experiment parameters. Consistent with the specifications laid out in §3.3.1, our experiments maintain the same settings. We have also expanded the scope of our investigations to study the effect of various hyper-parameters. This includes changing the batch size between $\{8, 32\}$, and adjusting sequence length from $\{128, 512\}$ during fine-tuning. Moreover, we explore the influence of the number of nodes, varying from $\{8, 16, 32, 64\}$, on strong-scaling speedup, while keeping the model size constant.

Roadmap. The ensuing sections are structured as follows: §A.1.2 presents the experimental findings on the $\text{BERT}_{\text{BASE}}$ model. In §A.1.3, we delve into the impact of model hyper-parameters on throughput and accuracy. Finally, §A.1.4 offers a theoretical analysis of slow network conditions using an analytical cost model.

A.1.2 Experimental results over $\text{BERT}_{\text{BASE}}$ model

Takeaway A.1. *When the evaluated model is $BERT_{BASE}$, AE and Quantization can preserve the model's accuracy on fine-tuning tasks over GLUE datasets.*

From Table A.1, we can observe that the accuracy loss is within 5% except for the CoLA dataset when using AE and quantization methods for compression. Since CoLA is the smallest dataset among GLUE datasets, slight perturbations can cause drastic changes in the final results.

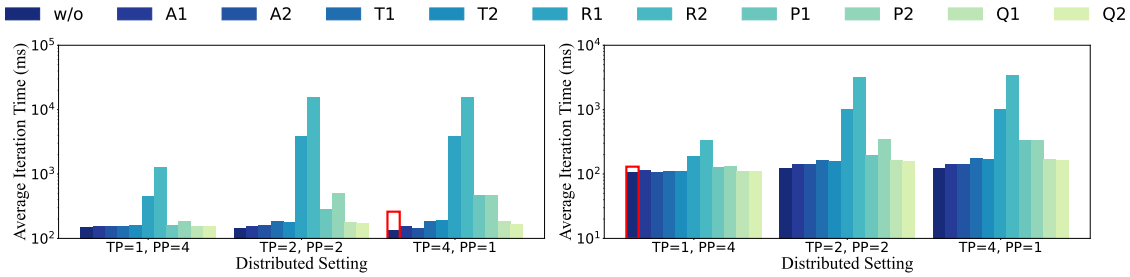
Table A.1: Fine-tuning results over GLUE dataset on $\text{BERT}_{\text{BASE}}$ model under the setting that the tensor model-parallel size is 2 and pipeline model-parallel size is 2. F1 scores are reported for QQP and MRPC, Matthews correlation coefficients are reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	83.45/84.16	90.62	91.40	83.83	57.52	91.31	62.09	84.79	81.02
A1	80.20/81.10	89.75	90.71	80.23	35.74	86.58	61.01	83.03	76.48
A2	80.22/80.83	89.71	90.60	80.64	38.84	86.82	62.82	83.28	77.06
T1	78.83/79.44	88.86	89.91	75.07	23.19	86.66	59.57	80.00	73.50
T2	80.31/80.94	89.20	90.71	74.14	31.60	88.32	62.09	81.61	75.44
P1	74.92/75.39	88.18	87.16	70.20	23.85	84.95	51.99	73.13	69.97
P2	74.04/74.78	87.93	86.93	66.49	0.00	84.88	52.35	71.90	66.59
Q1	82.48/83.04	89.91	91.40	81.33	50.49	89.36	61.01	83.90	79.21
Q2	83.31/84.14	90.50	91.97	83.07	55.22	91.07	61.01	85.15	80.60

A.1.3 Impact of model hyper-parameters

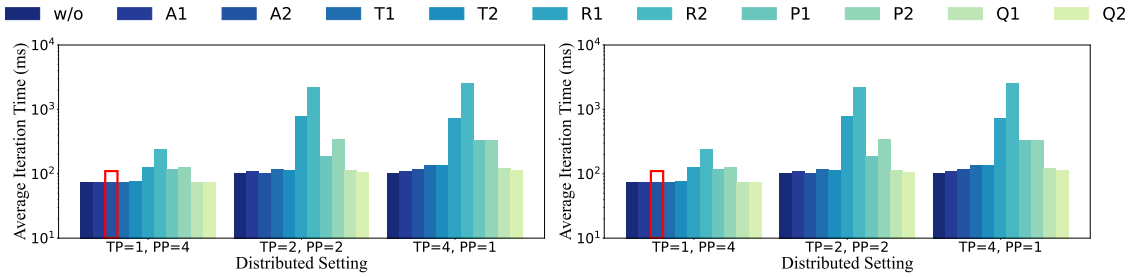
Takeaway A.2. *Using a smaller batch size or sequence length for fine-tuning negates the throughput benefits from compression because of the smaller communication cost.*

We vary the batch size from $\{8, 32\}$ and sequence length from $\{128, 512\}$, and report the results in Figure A.1a-A.2b. We notice that when the communication cost over model parallelism is small, the overhead of the compression methods can become the bottleneck. Therefore, we cannot improve system throughput when using compression algorithms with batch size 8 and sequence length 128.



(a) Batch size=32, Sequence length=128 **(b)** Batch size=8, Sequence length=128

Figure A.1: Average iteration time (ms) for fine-tuning with various batch sizes and sequence lengths. The results are collected from the AWS p3.8xlarge instance **with NVLink**. For each setting, we repeat experiments 5 times. Red rectangular boxes highlight the best method.



(a) Batch size=32, Sequence length=128 **(b)** Batch size=8, Sequence length=128

Figure A.2: Average iteration time (ms) for fine-tuning with various batch sizes and sequence lengths. The results are collected from the local machine **without NVLink**. For each setting, we repeat experiments 5 times. Red rectangular boxes highlight the best method.

Takeaway A.3. *Using a smaller batch size or sequence length for fine-tuning, AE and Quantization can also preserve the model's accuracy.*

From Table A.2 and Table A.3, it can be noted that the decrease in accuracy is kept within a 5% range for all but the CoLA and RTE datasets when employing AE and quantization techniques for compression. Furthermore, despite the utilization of fp64 for running the PowerSGD component, precision overflow remains a concern. This issue arises due to the instability of PowerSGD and the activation is not low-rank. In view of this, PowerSGD does not serve as a viable option for activation compression.

Table A.2: Fintune results over GLUE dataset under the setting using tensor parallelism size 2, pipeline parallelism size 2, batch size 8, and sequence length 128. F1 scores are reported for QQP and MRPC, Matthews correlation coefficient is reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	86.23/86.07	91.22	91.74	88.17	59.02	92.09	78.70	88.40	84.63
A1	82.49/82.41	89.93	91.85	82.43	43.56	89.84	47.29	87.03	77.43
A2	82.18/82.23	90.45	90.52	83.54	0.00	89.02	62.82	87.66	74.27
T1	49.07/47.96	72.02	83.57	69.33	12.04	83.60	55.60	84.96	62.02
T2	83.99/84.37	35.78	68.30	83.54	47.33	60.52	64.62	86.72	68.35
P1	36.66/37.18	68.28	81.19	67.59	0.00	58.23	55.23	7.26	45.74
P2	32.74/32.95	63.18	50.92	66.72	2.76	56.98	47.29	5.66	39.91
Q1	84.91/85.18	90.54	92.43	85.91	53.25	60.68	57.04	87.91	77.54
Q2	85.66/86.09	90.99	91.74	86.84	53.92	91.31	75.81	88.19	83.39

A.1.4 Slow network

Previous research [200] demonstrates that one application of activation compression is to speed up the fine-tuning process in slow network environments. In this section, we demonstrate that our cost model, as outlined in Eq. 3.3, enables a greater overall speedup in slow network environments compared to data center networks. The proof is provided below:

Table A.3: Fintune results over GLUE dataset under the setting using tensor parallelism size 2, pipeline parallelism size 2, batch size 32, and sequence length 128. F1 scores are reported for QQP and MRPC, Matthews correlation coefficient is reported for CoLA, and Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks.

Compression Algorithm	MNLI-(m/mm)	QQP	SST-2	MRPC	CoLA	QNLI	RTE	STS-B	Avg.
w/o	87.87/88.02	91.96	95.18	87.71	59.40	92.99	76.90	88.43	85.38
A1	85.30/85.33	91.28	92.32	84.58	55.18	90.87	59.93	87.92	81.41
A2	85.25/85.19	91.41	93.23	86.72	57.02	90.92	64.26	87.74	82.42
T1	68.76/69.23	64.58	91.40	80.93	0.00	67.34	66.43	69.24	64.21
T2	84.24/85.23	89.17	92.09	81.68	51.54	91.71	63.54	84.80	80.44
P1	32.74/32.95	63.18	49.08	81.68	0.00	50.54	61.73	-7.02	40.54
P2	32.74/32.95	50.27	49.08	78.67	0.00	50.54	44.04	0.00	37.59
Q1	86.85/87.58	91.50	93.58	86.96	59.20	92.24	59.57	86.89	82.71
Q2	87.46/88.02	91.82	94.95	87.48	57.02	93.36	68.95	87.84	84.10

Proof. Referring to the analytical cost model described in Section 3.2.3, we assume that w represents the bandwidth of the data center network and w' represents the bandwidth of a slower network, where $w' < w$. To simplify the notation, we define $A = (\frac{m-1}{n} + 1) \times L \times T$, $B = (\frac{m-1}{n} + 1) \times L \times T_X$, $C = (n-1) \times \frac{Bsh}{w}$, and $D = (n-1) \times \frac{M_c}{w}$.

It is evident that $T_X > T$ and $M_c < Bsh$. Consequently, this leads us to the conclusion that $\frac{A}{B} < 1 < \frac{C}{D}$. Next, we show that

$$\frac{A+C}{B+D} < \frac{A+\alpha C}{B+\alpha D}$$

where $\alpha = \frac{w}{w'} > 1$. The detailed steps are outlined below.

$$\begin{aligned}
\frac{A+C}{B+D} &< \frac{A+\alpha C}{B+\alpha D} \\
\Leftarrow (A+C)(B+\alpha D) &< (B+D)(A+\alpha C) \\
\Leftarrow BC+\alpha AD &< AD+\alpha BC \\
\Leftarrow AD &< BC \\
\Leftarrow \frac{A}{B} &< \frac{C}{D}
\end{aligned}$$

This finishes the proof. □

Appendix B

Architectures for Efficient Inference

B.1 Hyperparameters and Model Architectures

In this section, we provide the model architecture details in Table B.1.

Table B.1: Model Architectures: We list the architectural configurations of all models trained in this paper. d_{model} is the hidden size, f_{size} is the intermediate size, n_{layers} is the number of layers, and n_{heads} is the number of attention heads.

Model Size	Variant	d_{model}	f_{size}	n_{layers}	n_{heads}
80M	v1	512	1536	8	8
80M	v2	576	1536	5	8
80M	v3	640	1792	3	8
80M	v4	448	1280	13	8
80M	v5	384	1024	22	8
86M	v1	576	1536	7	8
86M	v2	640	1792	4	8
116M	v1	640	1792	10	10
116M	v2	720	2048	6	10
116M	v3	800	2304	4	10
116M	v4	880	2560	3	10
116M	v5	560	1536	15	10

Model Size	Variant	d_{model}	f_{size}	n_{layers}	n_{heads}
116M	v6	480	1280	24	10
126M	v1	720	2048	8	10
126M	v2	800	2304	5	10
164M	v1	768	2048	12	12
164M	v2	864	2304	8	12
164M	v3	960	2560	6	12
164M	v4	1056	2816	4	12
164M	v5	1152	3072	3	12
178M	v1	864	2304	10	12
178M	v2	960	2560	7	12
237M	v1	896	2560	14	14
237M	v2	1008	2816	10	14
237M	v3	1120	3072	8	14
237M	v4	1232	3328	6	14
313M	v1	1024	2816	16	16
313M	v2	1152	3072	12	16
313M	v3	1280	3584	9	16
313M	v4	1408	3840	7	16
339M	v1	1152	3072	14	16
Morph-1B	v1	2048	5632	24	16
Morph-1B	v2	2560	6912	16	16
Morph-1B	/	3072	8192	12	16

Appendix C

Scaling Laws Meet Model Architecture

C.1 Model Architectures

Table C.1 provides an overview of the model architectures, all configured with GQA = 4 and employing LLaMA-3.2 as the tokenizer.

Table C.1: Model Architectures: We list the architectural configurations of all models trained in this paper. $N_{\text{non-embed}}$ is the total number of non-embedding parameters, n_{layers} is the number of layers, d_{model} is the hidden size, n_{heads} is the number of attention heads, f_{size} is the intermediate size, and $r_{\text{mlp/attn}}$ is the MLP-to-attention ratio.

$N_{\text{non-embed}}$	Variant	n_{layers}	d_{model}	n_{heads}	f_{size}	$d_{\text{model}}/\sqrt{N}$	$r_{\text{mlp/attn}}$
80M	v1	12	768	16	2048	0.086	2.40
80M	v2	12	768	4	2688	0.086	12.6
80M	v3	12	768	8	2560	0.085	6.00
80M	v4	12	768	24	1536	0.087	1.20
80M	v5	12	768	32	1152	0.086	0.68
80M	v6	12	768	40	768	0.086	0.36
80M	v7	12	768	48	256	0.087	0.10
80M	v8	12	384	32	4096	0.043	2.40
80M	v9	12	384	8	5376	0.043	12.6
80M	v10	12	384	16	5120	0.042	6.00
80M	v11	12	384	48	3072	0.044	1.20

$N_{\text{non-embed}}$	Variant	n_{layers}	d_{model}	n_{heads}	f_{size}	$d_{\text{model}}/\sqrt{N}$	$r_{\text{mlp/attn}}$
80M	v12	12	384	64	2304	0.043	0.68
80M	v13	12	384	80	1536	0.043	0.36
80M	v14	12	384	96	512	0.044	0.10
80M	v15	12	1536	8	1024	0.171	2.40
80M	v16	12	1536	4	1280	0.169	6.00
80M	v17	12	1536	12	768	0.174	1.20
80M	v18	12	1536	16	640	0.169	0.75
80M	v19	12	1536	20	384	0.171	0.36
80M	v20	12	1536	24	128	0.174	0.10
80M	v21	12	512	24	3072	0.057	2.40
80M	v22	12	512	12	3840	0.056	6.00
80M	v23	12	512	16	3584	0.057	4.20
80M	v24	12	512	36	2304	0.058	1.20
80M	v25	12	512	48	1792	0.057	0.70
80M	v26	12	512	60	1152	0.057	0.36
80M	v27	12	512	72	384	0.058	0.10
80M	v28	12	1024	12	1536	0.114	2.40
80M	v29	12	1024	8	1792	0.113	4.20
80M	v30	12	1024	16	1280	0.115	1.50
80M	v31	12	1024	24	896	0.114	0.70
80M	v32	12	1024	36	256	0.114	0.13
80M	v33	12	2048	4	896	0.226	4.20
80M	v34	12	2048	8	640	0.231	1.50
80M	v35	12	2048	16	256	0.226	0.30
80M	v48	12	768	20	1792	0.086	1.68
80M	v49	12	768	28	1408	0.086	0.94
80M	v50	12	384	40	3584	0.043	1.68
80M	v51	12	384	52	3072	0.043	1.11
80M	v52	12	384	56	2816	0.043	0.94
80M	v53	12	384	60	2560	0.043	0.80
80M	v54	12	512	32	2560	0.058	1.50

$N_{\text{non-embed}}$	Variant	n_{layers}	d_{model}	n_{heads}	f_{size}	$d_{\text{model}}/\sqrt{N}$	$r_{\text{mlp/attn}}$
80M	v55	12	512	40	2176	0.057	1.02
80M	v56	12	512	44	1920	0.058	0.82
80M	v57	12	1024	20	1152	0.113	1.08
145M	v1	12	1024	16	3072	0.085	3.60
145M	v2	12	1024	8	3584	0.084	8.40
145M	v3	12	1024	24	2560	0.086	2.00
145M	v4	12	1024	32	2304	0.084	1.35
145M	v5	12	1024	40	1792	0.085	0.84
145M	v6	12	1024	48	1280	0.086	0.50
145M	v7	12	1024	64	512	0.085	0.15
145M	v8	12	512	32	6144	0.043	3.60
145M	v9	12	512	16	7168	0.042	8.40
145M	v10	12	512	48	5120	0.043	2.00
145M	v11	12	512	64	4608	0.042	1.35
145M	v12	12	512	80	3584	0.043	0.84
145M	v13	12	512	96	2560	0.043	0.50
145M	v14	12	512	128	1024	0.043	0.15
145M	v15	12	2048	8	1536	0.170	3.60
145M	v16	12	2048	4	1792	0.168	8.40
145M	v17	12	2048	12	1280	0.172	2.00
145M	v18	12	2048	16	1152	0.168	1.35
145M	v19	12	2048	20	896	0.170	0.84
145M	v20	12	2048	24	640	0.172	0.50
145M	v21	12	2048	32	256	0.170	0.15
145M	v22	12	768	24	3840	0.065	3.00
145M	v23	12	768	32	3584	0.063	2.10
145M	v24	12	768	40	3072	0.064	1.44
145M	v25	12	768	48	2560	0.065	1.00
145M	v26	12	768	56	2304	0.063	0.77
145M	v27	12	768	64	1792	0.064	0.53
145M	v28	12	1536	12	1920	0.129	3.00

$N_{\text{non-embed}}$	Variant	n_{layers}	d_{model}	n_{heads}	f_{size}	$d_{\text{model}}/\sqrt{N}$	$r_{\text{mlp/attn}}$
145M	v29	12	1536	16	1792	0.127	2.10
145M	v30	12	1536	20	1536	0.128	1.44
145M	v31	12	1536	24	1280	0.129	1.00
145M	v32	12	1536	28	1152	0.127	0.77
145M	v33	12	1536	32	896	0.128	0.53
145M	v34	12	4096	4	768	0.340	3.60
145M	v35	12	4096	16	128	0.340	0.15
145M	v48	12	1024	28	2368	0.086	1.59
145M	v49	12	1024	36	2048	0.085	1.07
145M	v50	12	512	52	5120	0.042	1.85
145M	v51	12	512	60	4800	0.042	1.50
145M	v52	12	512	68	4224	0.043	1.16
145M	v53	12	512	72	3968	0.043	1.03
145M	v54	12	768	44	2944	0.063	1.25
145M	v55	12	768	52	2432	0.064	0.88
297M	v1	12	1536	24	4096	0.089	3.20
297M	v2	12	1536	8	4864	0.090	11.4
297M	v3	12	1536	16	4608	0.088	5.40
297M	v4	12	1536	32	3584	0.090	2.10
297M	v5	12	1536	48	2816	0.089	1.10
297M	v6	12	1536	64	2048	0.088	0.60
297M	v7	12	1536	80	1024	0.090	0.24
297M	v8	12	768	48	8192	0.045	3.20
297M	v9	12	768	16	9728	0.045	11.4
297M	v10	12	768	32	9216	0.044	5.40
297M	v11	12	768	64	7168	0.045	2.10
297M	v12	12	768	96	5632	0.045	1.10
297M	v13	12	768	128	4096	0.044	0.60
297M	v14	12	768	160	2048	0.045	0.24
297M	v15	12	3072	12	2048	0.178	3.20
297M	v16	12	3072	4	2432	0.180	11.4

$N_{\text{non-embed}}$	Variant	n_{layers}	d_{model}	n_{heads}	f_{size}	$d_{\text{model}}/\sqrt{N}$	$r_{\text{mlp/attn}}$
297M	v17	12	3072	8	2304	0.177	5.40
297M	v18	12	3072	16	1792	0.180	2.10
297M	v19	12	3072	24	1408	0.178	1.10
297M	v20	12	3072	32	1024	0.177	0.60
297M	v21	12	3072	40	512	0.180	0.24
297M	v22	12	1024	36	6144	0.059	3.20
297M	v23	12	1024	12	7296	0.060	11.4
297M	v24	12	1024	24	6912	0.059	5.40
297M	v25	12	1024	48	5376	0.060	2.10
297M	v26	12	1024	72	4224	0.059	1.10
297M	v27	12	1024	96	3072	0.059	0.60
297M	v28	12	1024	120	1536	0.060	0.24
297M	v29	12	2048	12	3456	0.118	5.40
297M	v30	12	2048	24	2688	0.120	2.10
297M	v31	12	2048	48	1536	0.118	0.60
297M	v32	12	2048	60	768	0.120	0.24
297M	v45	12	1536	40	3200	0.089	1.50
297M	v46	12	1536	44	3072	0.089	1.31
297M	v47	12	1536	52	2688	0.088	0.97
297M	v48	12	1536	56	2432	0.089	0.81
297M	v49	12	768	80	6400	0.045	1.50
297M	v50	12	768	88	6016	0.045	1.28
297M	v51	12	768	104	5376	0.044	0.97
297M	v52	12	768	112	4736	0.045	0.79
297M	v53	12	3072	20	1664	0.177	1.56
297M	v54	12	3072	28	1152	0.180	0.77
297M	v55	12	1024	56	4864	0.060	1.63
297M	v56	12	1024	64	4608	0.060	1.35
297M	v57	12	1024	80	3840	0.059	0.90
297M	v58	12	1024	88	3328	0.060	0.71
297M	v59	12	2048	32	2432	0.117	1.43

$N_{\text{non-embed}}$	Variant	n_{layers}	d_{model}	n_{heads}	f_{size}	$d_{\text{model}}/\sqrt{N}$	$r_{\text{mlp/attn}}$
297M	v60	12	2048	36	2048	0.120	1.07
297M	v61	12	2048	40	1920	0.118	0.90
297M	v62	12	2048	44	1792	0.117	0.76
1B	v1	16	2048	32	8192	0.066	4.80
1B	v2	16	2048	72	5760	0.067	1.50
1B	v3	16	2816	92	2432	0.089	0.50
1B	v4	16	2816	76	3072	0.091	0.76
1B	v5	16	2816	68	3584	0.090	0.99
1B	v6	16	2816	60	4096	0.090	1.28
1B	v7	16	2816	56	4480	0.089	1.50
1B	v8	16	2816	24	6144	0.089	4.80
1B	v9	16	2816	48	4736	0.090	1.85
1B	v10	16	2816	40	5120	0.090	2.40
1B	v11	16	2816	36	5376	0.090	2.80
1B	v12	16	2560	64	4480	0.082	1.31
1B	v13	16	2560	72	4096	0.082	1.07
1B	v14	16	2560	80	3648	0.082	0.86
1B	v15	16	2560	56	4864	0.082	1.63
1B	v16	16	2560	88	3200	0.082	0.68
1B	v17	16	2560	48	5376	0.082	2.10

C.2 Inference FLOPs Analysis

Building on the inference FLOPs analysis from prior work [85], we begin with the following definition:

- d_{model}: hidden size
- f_{size}: intermediate (feed-forward) size
- n_{layers}: number of layers

- A: number of query heads
- K: number of key/value heads
- d_h: per-head hidden dimension (query and value)
- T: per-head hidden dim the KV length prior to token generation

Based on the above definition, we have $d_q = Ad_h$ and $d_{kv} = Kd_h$. We focus exclusively on non-embedding FLOPs, resulting in:

Attention: QKV and Project

$$n_{\text{layers}}(\underbrace{2d_{\text{model}}d_q}_{Q} + \underbrace{2d_{\text{model}}d_{kv}}_{K} + \underbrace{2d_{\text{model}}d_{kv}}_{V} + \underbrace{2d_{\text{model}}d_q}_{O})$$

Attention: Mask

$$n_{\text{layers}}(2Td_q)$$

Feedforward:

$$n_{\text{layers}}(3 \cdot 2d_{\text{model}}f_{\text{size}})$$

Total Inference non-embedding FLOPs:

$$\text{Total-FLOPs} = n_{\text{layers}}(\underbrace{2d_{\text{model}}d_q}_{Q} + \underbrace{2d_{\text{model}}d_{kv}}_{K} + \underbrace{2d_{\text{model}}d_{kv}}_{V} + \underbrace{2d_{\text{model}}d_q}_{O} + \underbrace{2Td_q}_{qK^\top} + \underbrace{3 \cdot 2d_{\text{model}}f_{\text{size}}}_{\text{up, gate, down}})$$

Since $P_{\text{non-emb}} \approx n_{\text{layers}}(2d_{\text{model}}d_q + 2d_{\text{model}}d_{kv} + 3d_{\text{model}}f_{\text{size}})$. Therefore, Total-FLOPs $= 2P_{\text{non-emb}} + 2n_{\text{layers}}Td_q$

We adopt the following three approaches to accelerate inference:

- Increasing the MLP-to-Attention ratio reduces the term $2Td_q$, thereby lowering the total FLOPs.
- Increasing the hidden size reduces the term $2Td_q$, thereby lowering the total FLOPs.

C.3 More Large-scale Training Results

In this section, we first show the detailed result over downstream tasks of large-scale models in Table C.2 and Table C.3.

Table C.2: Detailed Results on Downstream Tasks for 1B Models: In this table, we show detailed results of 1B models over 9 downstream tasks.

Downstream Tasks	LLaMA-3.2-1B	Panda-1B	Surefire-1B
Arc-Easy	58.8	60.9	59.7
Arc-Challenge	29.8	28.9	30.2
LAMBADA	52.8	55.1	52.0
HellaSwag	56.9	58.4	56.6
OpenBookQA	32.0	33.2	32.0
PIQA	73.6	75.2	73.0
SciQ	84.8	87.2	84.9
WinoGrande	57.1	58.6	57.5
COQA	48.7	55.3	52.7
Avg.	54.9	57.0	55.4

Table C.3: Detailed Results on Downstream Tasks for 3B Models: In this table, we show detailed results of 3B models over 9 downstream tasks.

Downstream Tasks	LLaMA-3.2-3B	Panda-3B	Surefire-3B	Panda-3B$^\circ$
Arc-Easy	66.4	65.5	67.6	66.8
Arc-Challenge	33.3	35.2	33.9	33.3
LAMBADA	60.6	61.8	61.4	61.5
HellaSwag	66.7	66.9	67.0	67.8
OpenBookQA	38.4	38.6	38.6	38.0
PIQA	76.8	76.9	77.4	76.8
SciQ	89.4	91.2	92.1	90.5
WinoGrande	62.5	63.2	60.5	62.7
COQA	63.3	63.4	65.4	64.9
Avg.	61.9	62.5	62.6	62.5

Bibliography

[1] Jeopardy. https://huggingface.co/datasets/jeopardy-datasets/jeopardy, 2022.

[2] Phi-2. https://huggingface.co/microsoft/phi-2, 2023.

[3] dclm baseline huggingface. https://huggingface.co/datasets/mlfoundations/dclm-baseline-1.0, 2024.

[4] grpc. https://grpc.io/, 2024.

[5] Llm foundry. https://github.com/mosaicml/llm-foundry, 2024.

[6] Nersc. https://www.nersc.gov/, 2024.

[7] Claude. https://www.anthropic.com/news/claude-opus-4-5, 2025.

[8] Grok 4.1. https://x.ai/news/grok-4-1, 2025.

[9] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

[10] Samira Abnar, Harshay Shah, Dan Busbridge, Alaaeldin Mohamed Elnouby Ali, Josh Susskind, and Vimal Thilak. Parameters vs flops: Scaling laws for optimal sparsity for mixture-of-experts language models. *arXiv preprint arXiv:2501.12370*, 2025.

[11] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems*, 6:114–127, 2024.

[12] Saurabh Agarwal, Amar Phanishayee, and Shivaram Venkataraman. Blox: A modular toolkit for deep learning schedulers. *arXiv preprint arXiv:2312.12621*, 2023.

[13] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. Adaptive gradient communication via critical learning regime identification. *Proceedings of Machine Learning and Systems*, 3:55–80, 2021.

[14] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

[15] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 348–363, 2023.

[16] Amey Agrawal, Nitin Kedia, Anmol Agarwal, Jayashree Mohan, Nipun Kwatra, Souvik Kundu, Ramachandran Ramjee, and Alexey Tumanov. On evaluating performance of llm inference serving systems. *arXiv preprint arXiv:2507.09019*, 2025.

[17] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[18] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[19] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[20] Ibrahim M Alabdulmohsin, Xiaohua Zhai, Alexander Kolesnikov, and Lucas Beyer. Getting vit in shape: Scaling laws for compute-optimal model design. *Advances in Neural Information Processing Systems*, 36:16406–16425, 2023.

[21] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017.

[22] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. The convergence of sparsified gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

[23] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[24] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[25] 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)*, pages 499–514, 2020.

[26] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[27] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[28] Song Bian, Dacheng Li, Hongyi Wang, Eric P Xing, and Shivaram Venkataraman. Does compressing activations help model parallel training? *Proceedings of Machine Learning and Systems*, 6:239–252, 2024.

[29] Song Bian, Minghao Yan, and Shivaram Venkataraman. Scaling inference-efficient language models. *arXiv preprint arXiv:2501.18107*, 2025.

[30] Song Bian, Tao Yu, Shivaram Venkataraman, and Youngsuk Park. Scaling laws meet model architecture: Toward inference-efficient llms. *arXiv preprint arXiv:2510.18245*, 2025.

[31] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR, 2023.

[32] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[33] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[34] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[35] 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. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[36] Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[37] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[38] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[39] Mengzhao Chen, Chaoyi Zhang, Jing Liu, Yutao Zeng, Zeyue Xue, Zhiheng Liu, Yunshui Li, Jin Ma, Jie Huang, Xun Zhou, et al. Scaling law for quantization-aware training. *arXiv preprint arXiv:2505.14302*, 2025.

[40] Andrew A Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. Reducing the carbon impact of generative ai inference (today and in 2035). In *Proceedings of the 2nd workshop on sustainable computer systems*, pages 1–7, 2023.

[41] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, et al. Mast: Global scheduling of ml training across geo-distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, 2024.

[42] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[43] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[44] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[45] Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. *Journal of the ACM (JACM)*, 68(1):1–39, 2021.

[46] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[47] 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)*, pages 613–627, 2017.

[48] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[49] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[50] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM Sigplan Notices*, 49(4):127–144, 2014.

[51] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

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

[53] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[54] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks VS Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. *arXiv preprint arXiv:2404.14618*, 2024.

[55] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[56] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[57] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[58] Samir Yitzhak Gadre, Georgios Smyrnis, Vaishaal Shankar, Suchin Gururangan, Mitchell Wortsman, Rulin Shao, Jean Mercat, Alex Fang, Jeffrey Li, Sedrick Keh, et al. Language models scale reliably with over-training and on downstream tasks. *arXiv preprint arXiv:2403.08540*, 2024.

[59] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

[60] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[61] Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

[62] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[63] Suchin Gururangan, Mitchell Wortsman, Samir Yitzhak Gadre, Achal Dave, Maciej Kilian, Weijia Shi, Jean Mercat, Georgios Smyrnis, Gabriel Ilharco, Matt Jordan, Reinhard Heckel, Alex Dimakis, Ali Farhadi, Vaishaal Shankar, and Ludwig Schmidt. open-lm: a minimal but performative language modeling (lm) repository, 2023. GitHub repository.

[64] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[65] 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*, pages 770–778, 2016.

[66] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[67] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[68] Geoffrey E Hinton and Richard Zemel. Autoencoders, minimum description length and helmholtz free energy. *Advances in neural information processing systems*, 6, 1993.

[69] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[70] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[71] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

[72] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter. *arXiv preprint arXiv:2403.07648*, 2024.

[73] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 457–472, 2023.

[74] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024.

[75] 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. *Advances in neural information processing systems*, 32, 2019.

[76] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739, 2021.

[77] Srinivasan Iyer, Xi Victoria Lin, Ramakanth Pasunuru, Todor Mihaylov, Daniel Simig, Ping Yu, Kurt Shuster, Tianlu Wang, Qing Liu, Punit Singh Koura, et al. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*, 2022.

[78] Peter Izsak, Moshe Berchansky, and Omer Levy. How to train bert with an academic budget. *arXiv preprint arXiv:2104.07705*, 2021.

[79] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 642–657, 2023.

[80] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[81] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[82] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[83] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

[84] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. Llamatune: Sample-efficient dbms configuration tuning. *arXiv preprint arXiv:2203.05128*, 2022.

[85] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[86] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

[87] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 39–48, 2020.

[88] Donghyun Kim, Chanyoung Park, Jinoh Oh, Sungyoung Lee, and Hwanjo Yu. Convolutional matrix factorization for document context-aware recommendation. In *Proceedings of the 10th ACM conference on recommender systems*, pages 233–240, 2016.

[89] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[90] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[91] Jakub Krajewski, Jan Ludziejewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, et al. Scaling laws for fine-grained mixture of experts. *arXiv preprint arXiv:2402.07871*, 2024.

[92] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[93] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[94] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[95] Neeraj Kumar, Pol Mauri Ruiz, Vijay Menon, Igor Kabiljo, Mayank Pundir, Andrew Newell, Daniel Lee, Liyuan Wang, and Chunqiang Tang. Optimizing resource allocation in hyperscale datacenters: Scalability, usability, and experiences. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 507–528, 2024.

[96] Tanishq Kumar, Zachary Ankner, Benjamin F Spector, Blake Bordelon, Niklas Muennighoff, Mansheej Paul, Cengiz Pehlevan, Christopher Ré, and Aditi Raghunathan. Scaling laws for precision. *arXiv preprint arXiv:2411.04330*, 2024.

[97] 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*, pages 611–626, 2023.

[98] Hector Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. In *Thirteenth international conference on the principles of knowledge representation and reasoning*, 2012.

[99] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[100] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7871–7880, 2020.

[101] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[102] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P Xing, and Hao Zhang. Mpcformer: fast, performant and private transformer inference with mpc. *arXiv preprint arXiv:2211.01452*, 2022.

[103] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. Amp: Automatically finding model parallel strategies with heterogeneity awareness. *arXiv preprint arXiv:2210.07297*, 2022.

[104] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, et al. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv preprint arXiv:2406.11794*, 2024.

[105] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850, 2023.

[106] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2023.

[107] 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)*, pages 583–598, 2014.

[108] Mu Li, David G Andersen, Alexander Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in neural information processing systems*, 27, 2014.

[109] 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. *arXiv preprint arXiv:2006.15704*, 2020.

[110] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient {GPU} memory sharing for concurrent {DNN} training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 161–175, 2021.

[111] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[112] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[113] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[114] 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. *arXiv preprint arXiv:1907.11692*, 2019.

[115] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[116] Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, et al. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*, 2024.

[117] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[118] Christopher D Manning. *Introduction to information retrieval*. Syngress Publishing,, 2008.

[119] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, 2018.

[120] MIG. *https://www.nvidia.com/en-us/technologies/multi-instance-gpu/*, 2023.

[121] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.

[122] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[123] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[124] MPS. *https://docs.nvidia.com/deploy/mps/index.html*, 2023.

[125] Niklas Muennighoff, Alexander Rush, Boaz Barak, Teven Le Scao, Nouamane Tazi, Aleksandra Piktus, Sampo Pyysalo, Thomas Wolf, and Colin A Raffel. Scaling data-constrained language models. *Advances in Neural Information Processing Systems*, 36:50358–50376, 2023.

[126] Aashiq Muhamed, Christian Bock, Rahul Solanki, Youngsuk Park, Yida Wang, and Jun Huan. Training large-scale foundation models on emerging ai chips. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5821–5822, 2023.

[127] 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*, pages 1–15, 2019.

[128] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

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

[130] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[131] 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*, pages 1–15, 2021.

[132] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[133] NVIDIA. *A100*, May 2025. `https://www.nvidia.com/en-us/data-center/a100`.

[134] NVIDIA. *V100*, May 2025. `https://www.nvidia.com/en-gb/data-center/tesla-v100/`.

[135] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[136] R OpenAI. Gpt-4 technical report. *arXiv*, pages 2303–08774, 2023.

[137] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. *arXiv preprint arXiv:1806.00187*, 2018.

[138] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

[139] Youngsuk Park, Kailash Budhathoki, Liangfu Chen, Jonas M Kübler, Jiaji Huang, Matthäus Kleindessner, Jun Huan, Volkan Cevher, Yida Wang, and George Karypis. Inference optimization of foundation models on ai accelerators. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6605–6615, 2024.

[140] 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. *Advances in neural information processing systems*, 32, 2019.

[141] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[142] Jackson Petty, Sjoerd van Steenkiste, Ishita Dasgupta, Fei Sha, Dan Garrette, and Tal Linzen. The impact of depth on compositional generalization in transformer language models. *arXiv preprint arXiv:2310.19956*, 2023.

[143] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[144] Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*, 2024.

[145] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[146] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[147] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[148] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[149] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[150] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.

[151] 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*, pages 1–16. IEEE, 2020.

[152] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–14, 2021.

[153] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[154] 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*, pages 3505–3506, 2020.

[155] Siva Reddy, Danqi Chen, and Christopher D Manning. Coqa: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019.

[156] 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 *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[157] Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *2011 AAAI spring symposium series*, 2011.

[158] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[159] Yangjun Ruan, Chris J Maddison, and Tatsunori Hashimoto. Observational scaling laws and the predictability of language model performance. *arXiv preprint arXiv:2405.10938*, 2024.

[160] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

[161] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[162] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[163] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3715–3734, 2022.

[164] Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws. *arXiv preprint arXiv:2401.00448*, 2023.

[165] Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superglue: Learning feature matching with graph neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4938–4947, 2020.

[166] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[167] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

[168] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth annual conference of the international speech communication association*. Citeseer, 2014.

[169] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[170] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[171] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[172] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, 2024.

[173] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

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

[175] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[176] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[177] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[178] Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, et al. Dolma: An open corpus of three trillion tokens for language model pretraining research. *arXiv preprint arXiv:2402.00159*, 2024.

[179] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. Optimus-cc: Efficient large nlp model training with 3d parallelism aware communication compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 560–573, 2023.

[180] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[181] Charles Spearman. The proof and measurement of association between two things. 1961.

[182] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

[183] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. *Advances in Neural Information Processing Systems*, 31, 2018.

[184] Chaofan Tao, Qian Liu, Longxu Dou, Niklas Muennighoff, Zhongwei Wan, Ping Luo, Min Lin, and Ngai Wong. Scaling laws with vocabulary: Larger models deserve larger vocabularies. *arXiv preprint arXiv:2407.13623*, 2024.

[185] Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. Scale efficiently: Insights from pre-training and fine-tuning transformers. *arXiv preprint arXiv:2109.10686*, 2021.

[186] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[187] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[188] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[189] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[190] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[191] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[192] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[193] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[194] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[195] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP workshop BlackboxNLP: Analyzing and interpreting neural networks for NLP*, pages 353–355, 2018.

[196] Chong Wang and David M Blei. Collaborative topic modeling for recommending scientific articles. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 448–456, 2011.

[197] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209*, 2023.

[198] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[199] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. *Advances in Neural Information Processing Systems*, 31, 2018.

[200] Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. Fine-tuning language models over slow networks using activation compression with guarantees. *arXiv preprint arXiv:2206.01299*, 2022.

[201] 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*, pages 1–17, 2019.

[202] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[203] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[204] Johannes Welbl, Nelson F Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. *arXiv preprint arXiv:1707.06209*, 2017.

[205] 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)*, pages 945–960, 2022.

[206] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[207] T Wolf. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[208] 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*, pages 38–45, 2020.

[209] Sewall Wright. Correlation and causation. *Journal of agricultural research*, 20(7):557, 1921.

[210] Carole-Jean Wu, Bilge Acun, Ramya Raghavendra, and Kim Hazelwood. Beyond efficiency: Scaling ai sustainably. *IEEE Micro*, 44(5):37–46, 2024.

[211] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. *arXiv preprint arXiv:2410.10819*, 2024.

[212] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[213] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[214] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020.

[215] Minghao Yan, Saurabh Agarwal, and Shivaram Venkataraman. Decoding speculative decoding. *arXiv preprint arXiv:2402.01528*, 2024.

[216] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[217] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[218] Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *Advances in neural information processing systems*, 37:115491–115522, 2024.

[219] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 2369–2380, 2018.

[220] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[221] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: A survey. *ACM Computing Surveys*, 56(6):1–38, 2024.

[222] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[223] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.

[224] 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)*, pages 521–538, 2022.

[225] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems*, 2:98–111, 2020.

[226] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[227] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International conference on machine learning*, pages 11328–11339. PMLR, 2020.

[228] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 2024.

[229] 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. *arXiv preprint arXiv:2205.01068*, 2022.

[230] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[231] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters. *arXiv preprint arXiv:2303.13803*, 2023.

[232] 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*, pages 428–440, 2022.

[233] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[234] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. 2023.

[235] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[236] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 703–723, 2023.

[237] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[238] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

[239] 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. *Proceedings of Machine Learning and Systems*, 5:526–540, 2023.