

UC Berkeley

UC Berkeley Electronic Theses and Dissertations

Title

Scalable and Efficient Systems for Large Deep Learning Models

Permalink

<https://escholarship.org/uc/item/97r5r746>

Author

Zheng, Lianmin

Publication Date

2024

Peer reviewed|Thesis/dissertation

Scalable and Efficient Systems for Large Deep Learning Models

By

Lianmin Zheng

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Co-chair
Professor Joseph E. Gonzalez, Co-chair
Professor Koushik Sen
Professor Alex Aiken

Summer 2024

Scalable and Efficient Systems for Large Deep Learning Models

Copyright 2024
by
Lianmin Zheng

Abstract

Scalable and Efficient Systems for Large Deep Learning Models

by

Lianmin Zheng

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Co-chair

Professor Joseph E. Gonzalez, Co-chair

Recent advancements in machine learning have primarily been driven by large-scale deep learning models, particularly large language models. The large scale and new capabilities of these models present challenges in designing infrastructure systems to support their entire lifecycle, from training and serving to evaluation. To meet the high computational and memory requirements of these models, while fully utilizing and accurately evaluating their capabilities, we need to redesign many system components, such as compilers, distributed computing platforms, programming systems, and evaluation methods.

In this dissertation, we introduce a suite of systems designed and built to support large models, covering training, serving, and evaluation phases. First, we discuss Alpa, a system for large-scale model-parallel training, which automatically generates distributed execution plans integrating both inter- and intra-operator parallelism. Moving on to serving, we introduce Ansor, a compiler that produces high-performance implementations of tensor programs for various hardware backends. We also explore SGLang, a system for deploying large language models that includes both a flexible front-end programming interface and an optimized back-end runtime for fast inference. Lastly, in the evaluation phase, we detail our efforts in model evaluation, which include Chatbot Arena, a crowdsourced live benchmark platform, and LLM-as-a-Judge, an automated evaluation pipeline. These tools collectively form a full-stack system for the continuous improvement of large models.

To my family.

Contents

Contents	ii
1 Introduction	1
1.1 Scalably Training Large Models	1
1.2 Efficiently Serving Large Models	2
1.3 Accurately Evaluating Large Models	2
1.4 Summary	3
2 Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning	4
2.1 Introduction	5
2.2 Background: Distributed Deep Learning	8
2.3 Overview	10
2.4 Intra-Operator Parallelism	12
2.5 Inter-Operator Parallelism	16
2.6 Parallelism Orchestration	20
2.7 Limitations and Discussion	21
2.8 Evaluation	22
2.9 Related Work	29
2.10 Conclusion	30
3 Generating High-Performance Tensor Programs for Deep Learning	32
3.1 Introduction	33
3.2 Background	34
3.3 Design Overview	37
3.4 Program Sampling	38
3.5 Performance Fine-tuning	44
3.6 Task Scheduler	46
3.7 Evaluation	49
3.8 Related Work	56
3.9 Limitations and Future work	58
3.10 Conclusion	58

4	Efficiently Programming Large Language Models using SGLang	59
4.1	Introduction	60
4.2	Background	62
4.3	Overview	66
4.4	Language, Interpreter, and Compiler	67
4.5	Runtime with RadixAttention	70
4.6	Evaluation	73
4.7	Discussion and Limitations	81
4.8	Related Work	82
4.9	Conclusion	82
5	Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena	83
5.1	Introduction	84
5.2	MT-Bench and Chatbot Arena	86
5.3	LLM as a Judge	87
5.4	Agreement Evaluation	92
5.5	Human Preference Benchmark and Standardized Benchmark	95
5.6	Discussion	96
5.7	Conclusion	96
5.8	Notes on the Appendix	97
6	Conclusion	98
6.1	Review	98
6.2	Impact	98
6.3	Future Directions	99
	Bibliography	101

Acknowledgments

This dissertation would not exist without the support and guidance of a phenomenal group of people.

First and foremost, I am incredibly grateful to my advisors, Ion Stoica and Joseph E. Gonzalez. Ion taught me the importance of thinking ambitiously and tackling problems with real-world significance. His vision and guidance have been invaluable throughout my Ph.D. journey. Joey, on the other hand, exemplifies the power of smart thinking when facing complex challenges. His ability to break down problems and find elegant solutions has been a constant source of inspiration. Together, they create a truly exceptional environment that fosters growth and success. Their unwavering support and visionary leadership have been instrumental in shaping me as a researcher.

My thesis committee, Koushik Sen and Alex Aiken, also deserve immense gratitude. Koushik’s expertise in programming languages has been a tremendous resource, providing invaluable insights during critical stages of my research. Alex Aiken’s pioneering work on compilers and distributed systems heavily influenced the direction of my research. Similarly, I would not have entered the fascinating world of parallel computing without the foundational knowledge gained from the classes of my preliminary exam committee, James Demmel and Katherine Yelick.

The seeds of my research career were sown during my undergraduate studies under the guidance of Tianqi Chen, Weinan Zhang, and Yong Yu. Their early mentorship played a crucial role in sparking my research interests, particularly with the TVM project. This foundation proved invaluable throughout my Ph.D.

My deepest gratitude extends to my core collaborators, Ying Sheng, Hao Zhang, Wei-Lin Chiang, and Zhuohan Li. Together, we have launched numerous successful projects like LMSYS and Alpa. The countless hours spent brainstorming, discussing, and debugging alongside them are memories I will forever cherish. These collaborations have not just yielded impactful research; they have fostered deep friendships.

The smooth sailing of my research journey would not have been possible without the fantastic lab support staff – Kattt Atchley, Jon Kuroda, Ivan Ortega, and Kailee Truong. Their efficiency and helpfulness made navigating administrative tasks and logistical hurdles significantly easier.

Even though miles separated me from them, my family’s unwavering support has been the bedrock of my success. To my parents, Shengqin Zheng and Yunmiao Ma, and my sister, Enci Zheng, thank you for your constant encouragement and endless love. A special thanks goes to my friend, Tiancheng Xie. Our adventures exploring the outdoors provided much-needed breaks and helped me navigate the inevitable emotional rollercoaster of research and life.

Finally, to my wife, Ying Sheng, words cannot express my gratitude. Your brilliance and insightful discussions have not only enriched my research but also broadened my perspective on the world itself. Your unwavering support and virtuous nature are the wind beneath my wings. You are the reason for my success.

In addition to those mentioned above, I would like to express my sincere appreciation to all my other collaborators: Ameer Haj Ali, Clark Barrett, Ruisi Cai, Shiyi Cao, Yukuo Cen, Jianfei Chen, Tianqi Chen, Tianlong Chen, Beidi Chen, Weize Chen, Zhifeng Chen, Alvin Cheung, Wei-Lin Chiang, Christopher Chou, Siyuan Feng, Daniel Y Fu, Joseph E. Gonzalez, Ameer Haj-Ali, Xu Han, Qirong Ho, Coleman Hooper, Bohan Hou, Yanping Huang, Jeff Huang, Chengfan Jia, Jiantao Jiao, Hongyi Jin, Xin Jin, Michael I. Jordan, Kurt Keutzer, Christos Kozyrakis, Woosuk Kwon, Ruihang Lai, Nicholas Lee, Zhuohan Li, Tianle Li, Dacheng Li, Percy Liang, Zi Lin, Wuwei Lin, Ruochen Liu, Zhiyuan Liu, Vincent Liu, Michael W. Mahoney, Max Ryabinin, Christopher Ré, Koushik Sen, Junru Shao, Ying Sheng, Zhao Song, Ion Stoica, Minmin Sun, Chuyue Sun, Jie Tang, Yuandong Tian, Dequan Wang, Zhangyang Wang, Yida Wang, Zhao Wu, Zhanghao Wu, Zhiqiang Xie, Eric P. Xing, Yuanzhong Xu, Shuo Yang, Jun Yang, Zhewei Yao, Zihao Ye, Liangsheng Yin, Cody Hao Yu, Yong Yu, Binhang Yuan, Ce Zhang, Hao Zhang, Zhenyu Zhang, Hexu Zhao, Yinmin Zhong, Tianyi Zhou, Banghua Zhu, Yonghao Zhuang, Siyuan Zhuang, and Danyang Zhuo.

This journey has been an incredible experience, and I am deeply grateful to everyone who has been a part of it.

Chapter 1

Introduction

The field of machine learning has witnessed a significant revolution driven by the emergence of large-scale deep learning models, particularly large language models (LLMs). These models, characterized by billions or even trillions of parameters, have demonstrated the ability to tackle a wide range of challenging and open-domain tasks. From basic natural language processing tasks like sentiment analysis to complex code generation and reasoning, large models are pushing the boundaries of what is possible with artificial intelligence.

However, their immense capabilities come at a cost. Training and deploying these models present unique systems challenges across their entire lifecycle, including training, serving, and evaluation. These challenges primarily stem from the substantial computational and memory resources required by these models, but also from their novel and evolving capabilities. Traditional system design approaches struggle to keep pace with the demands of LLMs, necessitating a fundamental rethink of core components like compilers, distributed systems, memory management, programming systems, and model evaluation methods. In this thesis, we work broadly and address these critical challenges by proposing a suite of systems that span the entire lifecycle of large-scale deep learning models.

1.1 Scalably Training Large Models

The scaling law of neural language models suggests that both model size and training data size need to be continuously scaled to achieve better model performance. To handle the large amount of parameters and data, parallel computing across a large number of accelerators is essential. This presents a significant challenge: parallelizing the complex computational graph for training a neural network. Numerous parallelization techniques exist, each with trade-offs in terms of compute utilization, communication overhead, and memory usage. Finding the optimal configuration is a complex search problem.

In chapter 2, we introduce Alpa, an automatic distributed training system designed to address this challenge. Alpa proposes a unified representation and runtime system for handling both inter-operator and intra-operator parallelism. It employs a hierarchical search

algorithm that leverages integer linear programming and dynamic programming to effectively explore the search space. This approach allows Alpa to automatically find a parallelization plan for a given neural network and a device cluster, achieving scalable training across distributed accelerators.

1.2 Efficiently Serving Large Models

While the capabilities of large models are impressive, using them for inference can be computationally expensive. This stems from the sheer number of operators involved in computations and the diverse hardware backends on which these models might be deployed. The challenge lies in generating high-performance implementations, often referred to as kernels, for these operators across various hardware platforms. Traditionally, this has involved significant manual optimization efforts utilizing techniques like tiling and fusion.

In chapter 3, to address this challenge, we introduce Ansor, a system that automates the generation of high-performance tensor programs for deep learning applications. Ansor tackles this problem by exploring a much larger space of potential optimizations compared to existing approaches. It achieves this by sampling programs from a hierarchical representation of the search space and then refining these programs using evolutionary search guided by a learned cost model. This approach allows Ansor to discover high-performance programs that might not be considered by traditional methods and effectively generate optimized kernels for diverse operators and hardware backends.

Furthermore, the usage patterns for large language models are evolving. We are witnessing a shift from simple, manually prompted chat interactions to complex programmatic workflows involving multiple chained generation calls. This necessitates a system that can not only flexibly express these intricate patterns but also execute them efficiently.

In chapter 4, we introduce SGLang to address this need. SGLang is a domain-specific language designed specifically for programming LLMs. It incorporates primitives that cater to common LLM programming patterns, allowing users to express complex workflows concisely and efficiently. SGLang is implemented as a domain-specific language embedded within Python. It includes an interpreter, a compiler, and a high-performance runtime system that work together to enable optimizations such as parallelism, batching, caching, and code sharing. Additionally, SGLang introduces novel techniques for the reuse of the KV cache across multiple generation calls at runtime, further enhancing the efficiency of LLM program execution. By providing a flexible and efficient programming environment, SGLang simplifies the development process for complex LLM applications while maximizing their performance.

1.3 Accurately Evaluating Large Models

The rapidly evolving capabilities of large language models, particularly their ability to handle open-domain tasks and generate lengthy outputs, pose new challenges for traditional

evaluation methods. Existing benchmarks often fall short in capturing the nuances of human preference, making it difficult to accurately assess the performance of these models. Additionally, the slow and expensive nature of human evaluation makes it impractical for large-scale assessments. Furthermore, automated evaluation methods can introduce biases that skew the results.

In chapter 5, we propose a combined approach utilizing LLM-as-a-Judge and Chatbot Arena. LLM-as-a-Judge leverages the capabilities of powerful LLMs to act as automated evaluators for other models. This approach offers scalability and the potential to capture human-like preferences. Chatbot Arena, on the other hand, is a crowdsourced platform that facilitates human evaluation of chatbots in a controlled environment. By combining these methods, we can leverage the strengths of both automated and human evaluation to achieve a more comprehensive understanding of LLM performance.

1.4 Summary

In this chapter, we introduced the challenges and opportunities presented by large-scale deep learning models. We highlighted the critical system bottlenecks encountered throughout the lifecycle of LLMs, including training, serving, and evaluation phases. To address these challenges, we proposed a suite of novel systems: Alpa for scalable training, Ansor and SGLang for efficient serving, and LLM-as-a-Judge and Chatbot Arena for accurate evaluation. These systems represent a comprehensive approach that leverages techniques from various fields, including compilers, distributed systems, high-performance computing, and machine learning itself. By working together, these systems create a virtuous cycle that fosters continuous improvement in the development and deployment of the next generation of large-scale deep learning models.

Chapter 2

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

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

This chapter is based on my previous paper [225], which was a collaborative effort with Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

2.1 Introduction

Several of the recent advances [79, 25, 170] in deep learning (DL) have been a direct result of significant increases in model size. For example, scaling language models, such as GPT-3, to hundreds of billions of parameters [25] and training on much larger datasets enabled fundamentally new capabilities.

However, training these extremely large models on distributed clusters currently requires a significant amount of engineering effort that is specific to both the model definition and the cluster environment. For example, training a large transformer-based language model requires heavy tuning and careful selection of multiple parallelism dimensions [128]. Training the large Mixture-of-Expert (MoE) transformers model [99, 49] on TPU clusters requires manually tuning the partitioning axis for each layer, whereas training the same model on an AWS GPU cluster calls for new pipeline schemes that can depend on the choices of partitioning (§2.8).

More generally, efficient large-scale model training requires tuning a complex combination of data, operator, and pipeline parallelization approaches at the granularity of the individual tensor operators. Correctly tuning the parallelization strategy has been shown [105, 98] to deliver an order of magnitude improvements in training performance, but depends on strong machine learning (ML) and system expertise.

Automating the parallelization of large-scale models would significantly accelerate ML research and production by enabling model developers to quickly explore new model designs without regard for the underlying system challenges. Unfortunately, it requires navigating a complex space of plans that grows exponentially with the dimensions of parallelism and the size of the model and cluster. For example, when all parallelism techniques are enabled, figuring out the execution plan involves answering a web of interdependent questions, such as how many data-parallel replicas to create, which axis to partition each operator along, how to split the model into pipeline stages, and how to map devices to the resulting parallel executables. The interplay of different parallelization methods and their strong dependence on model and cluster setups form a combinatorial space of plans to optimize. Recent efforts [130, 51, 195] to automatically parallelize model training are constrained to the space of a single model-parallelism approach, or rely on strong assumptions on the model and cluster specifications (§2.2).

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

With this observation in mind, in this paper, we take a different view from conventional data and model parallelisms, and re-categorize ML parallelization approaches as *intra-operator* and *inter-operator* parallelisms. Intra-operator parallelism partitions ML operators along one or more tensor axes (batch or non-batch) and dispatches the partitions

to distributed devices (Fig. 2.1c); inter-operator parallelism, on the other hand, slices the model into disjoint stages and pipelines the execution of stages on different sets of devices (Fig. 2.1d). They take place at two different granularities of the model computation, differentiated by whether to partition operators.

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

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

In summary, we make the following contributions:

- We construct a two-level parallel execution plan space (Fig. 2.1e) where plans are specified hierarchically using inter- and intra-operator parallelisms.
- We design tractable optimization algorithms to derive near-optimal execution plans at each level.
- We implement Alpa, a compiler system for distributed DL on GPU clusters. Alpa features: (1) a set of compilation passes that generate execution plans using the hierarchical optimization algorithms, (2) a new runtime architecture that orchestrates the inter-op parallelism between device meshes, and (3) a number of system optimizations that improve compilation and address cross-mesh communication.
- We evaluate Alpa on training large models with billions of parameters. We compare Alpa with state-of-the-art distributed training systems on an Amazon EC2 cluster of 8 p3.16xlarge instances with 64 GPUs. On GPT [25] models, Alpa can match the specialized system Megatron-LM [170, 128]. On GShard MoE models [99], compared to a hand-tuned system Deepspeed[152], Alpa achieves a $3.5\times$ speedup on 2 nodes and a $9.7\times$ speedup on 4 nodes. Unlike specialized systems, Alpa also generalizes to models without manual strategies and achieves an 80% linear scaling efficiency on Wide-ResNet [221] with 4 nodes. This means

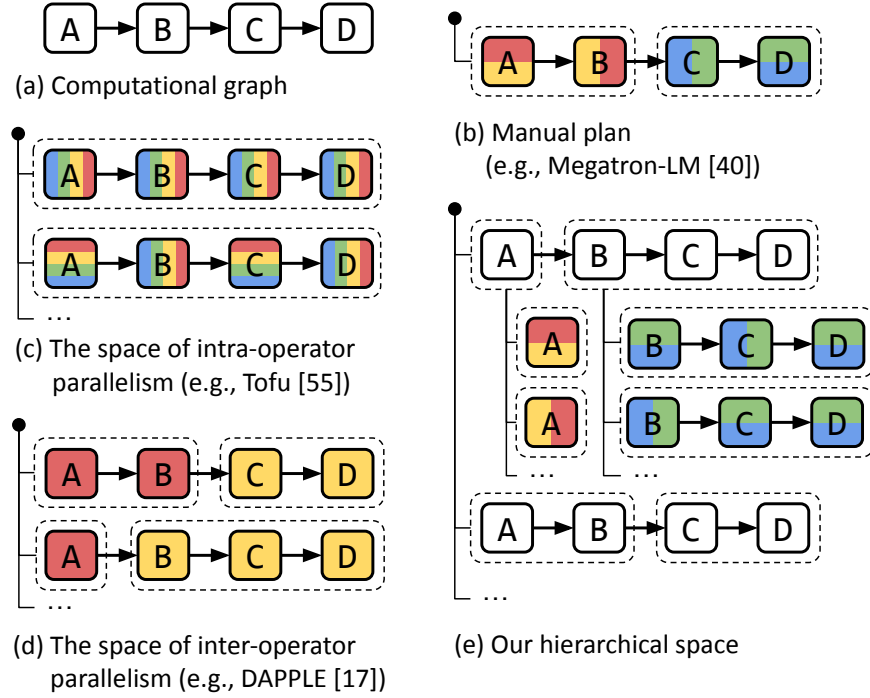


Figure 2.1: Generation of parallelization plans for a computational graph shown in (a). Different colors represent different devices, dashed boxes represent pipeline stages. (b) creates the plan manually. (c) and (d) automatically generate plans using only one of intra- and inter-operator parallelisms. (e) shows our approach that creates a hierarchical space to combine intra- and inter-operator parallelisms.

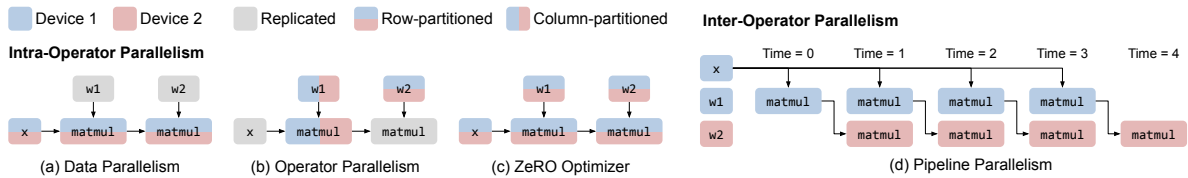


Figure 2.2: Common parallelization techniques for training a 2-layer Multi-layer Perceptron (MLP). Only the forward pass is shown. “x” is the input data. “w1” and “w2” are two weight matrices.

developers can get efficient model-parallel execution of large DL models out-of-the-box using Alpa.

2.2 Background: Distributed Deep Learning

DL computation is commonly represented by popular ML frameworks [1, 143, 23] as a dataflow graph. Edges in the graph represent multi-dimensional tensors; nodes are computational operators, such as matrix multiplication (matmul), that transform input tensors into output tensors. Training a DL model for one iteration consists of computing a loss by *forwarding* a batch of data through the graph, deriving the updates via a reverse *backward* pass, and applying the updates to the parameters via *weight update* operations. In practice, model developers define the dataflow graph. An execution engine then optimizes and executes it on a compute device.

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

Conventional View of ML Parallelism

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

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

Operator parallelism. When the model is too large to fit in one device, operator parallelism is an effective model parallelism option. Operator parallelism refers to approaches that partition the computation of a specific operator (abbreviated as *op* in the following text), such as matmul shown in Fig. 2.2b, along *non-batch* axes, and compute each part of the operator in parallel across multiple devices.

Because input tensors are jointly partitioned, when a device computes its *op* partition, the required portions of input tensors may not reside in its local memory. Communication is thus required to fetch the input data from other devices. When the tensors are partitioned evenly, i.e., SPMD[211], all devices will follow the same collective communication patterns such as all-reduce, all-gather, and all-to-all.

Pipeline parallelism. Instead of partitioning ops, pipeline parallelism places different groups of ops from the model graph, referred as *stages*, on different workers; meanwhile, it splits the training batch as a number of microbatches, and pipelines the forward and backward passes across microbatches on distributed workers, as Fig. 2.2d shows. Unlike operator parallelism, pipeline parallelism transfers intermediate activations at the forward and backward passes between different workers using point-to-point communication.

Manual combination of parallelisms. Recent development shows the approaches mentioned above need to be combined to scale out today’s large DL models[128, 211]. The state-of-the-art training systems, such as Megatron-LM [170, 128], manually design a specialized execution plan that combines these parallelisms for transformer language models,

which is also known as *3D Parallelism*. By assuming the model has the same transformer layer repeated, it assigns an equal number of layers to each pipeline stage and applies a hand-designed operator and data parallelism configuration uniformly for all layers. Despite the requirement of strong expertise, the manual plan cannot generalize to different models or different cluster setups (§2.8).

Automatic combination of parallelisms. The configurations of each individual parallelism, their interdependence, and their dependence on model and cluster setups form an intractable space, which prevents the trivial realization of automatically combining these parallelisms. For examples, when coupled with operator parallelism, each time adding a data-parallel replica would require allocating a new set of devices (instead of one single device) as the worker, and figuring out the optimal operator parallelism configurations within those devices. When including pipeline parallelism, the optimal pipelining scheme depends on the data and operator parallelism choices of each pipeline stage and how devices are allocated for each stage. With this conventional view, prior explorations [84, 195, 51, 223] of auto-parallelization are limited to combining data parallelism with at most one model parallelism approach, which misses substantial performance opportunities. We next develop our view of ML parallelisms.

Intra- and Inter-Operator Parallelisms

Different from the conventional view, in this paper, we re-catalog existing parallelization approaches into two orthogonal categories: intra-operator and inter-operator parallelisms. They are distinguished by if they involve partitioning operators along any tensor axis. We next use the examples in Fig. 2.2 to introduce the two types of parallelisms.

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

Fig. 2.2a-c illustrates the application of several typical instantiations of intra-op parallelism on an MLP. Data parallelism [93], by definition, belongs to intra-op parallelism – the input tensors and matmuls are partitioned along the batch dimension, and weight tensors are replicated. Alternatively, when the weights are very large, partitioning the weights (Fig. 2.2b) leads to the aforementioned operator parallelism adopted in Megatron-LM. Besides operators in the forward or backward passes, one can also partition the operators from the weight update phase, yielding the *weight update sharding* or equivalently the ZeRO [151, 210] technique, commonly comprehended as an optimization of data parallelism.

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

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

Fig. 2.2d illustrates the batch-splitting pipeline parallelism as a case of inter-operator parallelism.¹ The pipeline execution can follow different schedules, such as Gpipe [79], PipeDream [130], and synchronous 1F1B [129, 51]. We adopt the synchronous 1F1B schedule throughout this paper as it respects synchronous consistency, and has the same pipeline latency but lower peak memory usage compared to Gpipe.

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

By this categorization, the two parallelisms take place at different granularities of the DL computation and have distinct communication requirements, which happen to match the structure of today’s compute clusters. We will leverage these properties to design hierarchical algorithms and compilation passes to auto-generate execution plans. Several concurrent work [105, 129, 8, 178] have proposed similar categorization, but Alpa is the first end-to-end system that uses this categorization to automatically generate parallel plans from the full space.

2.3 Overview

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

To achieve this, Alpa implements three novel compilation passes as Fig. 4.3 shows. Given a model description, in the form of a Jax [23] intermediate representation (IR), and a cluster configuration, the inter-op compilation pass slices the IR into a number of stages, and slices the device cluster into a number of device meshes. The inter-op pass uses a Dynamic Programming (DP) algorithm to assign stages to meshes and invokes the intra-op compilation pass on each stage-mesh pair, to query the execution cost of this assignment. Once invoked, the intra-op pass optimizes the intra-op parallel execution plan of the stage running

¹Device placement [121] is another case of inter-op parallelism, which partitions the model graph and executes them on different devices but does not saturate pipelines using multiple microbatches. Hence pipeline parallelism is often seen as a better alternative to it because of less device idle time.

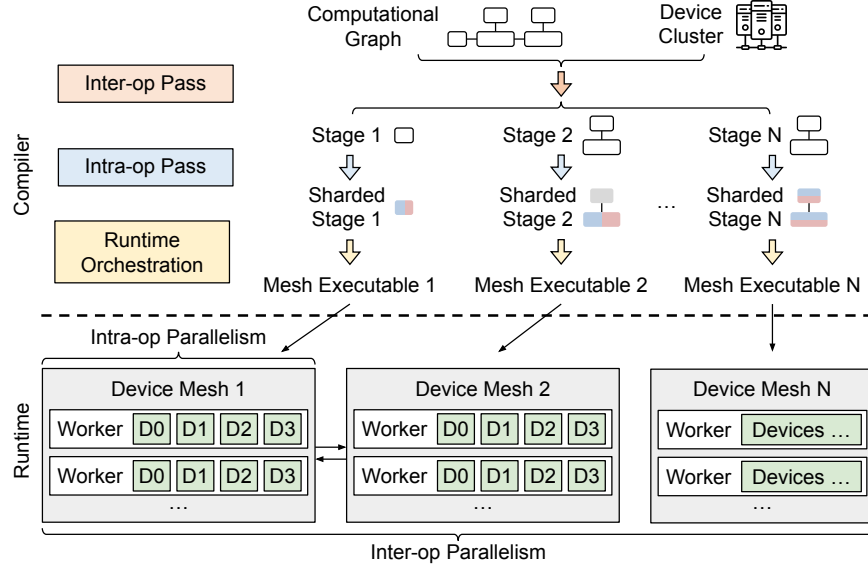


Figure 2.3: Compiler passes and runtime architecture. A sharded stage is a stage annotated with the sharding specs generated by intra-op pass.

on its assigned mesh, by minimizing its execution cost using an Integer Linear Programming (ILP) formulation, and reports the cost back to the inter-op pass. By repeatedly querying the intra-op pass for each allocation of a stage-mesh pair, the inter-op pass uses the DP to minimize the inter-op parallel execution latency and obtains the best slicing scheme of stages and meshes.

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

API. Alpa has a simple API shown in Fig. 2.4. Alpa requires developers to annotate functions to be parallelized, such as the `train_step()`, using a Python decorator `@parallelize`. Upon the first call to `train_step()`, Alpa traces the whole function to get the model IR, invokes the compilation, and converts the function to a parallel version.

Since the inter-op pass depends on the intra-op pass, in the following text, we first describe the intra-op pass, followed by the inter-op pass, and finally the runtime orchestration pass.

```

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

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

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

```

Figure 2.4: An example to demonstrate Alpa’s API for Jax. The developers uses a Python decorator `@parallelize` to annotate functions that need to be parallelized. The rest of the program is kept intact.

2.4 Intra-Operator Parallelism

Alpa optimizes the intra-operator parallelism plan within a device mesh. Alpa adopts the SPMD-style intra-op parallelism [99, 211] which partitions operators evenly across devices and executes the same instructions on all devices, as per the fact that devices within a single mesh have equivalent compute capability. This SPMD style significantly reduces the space of intra-op parallelism plans; meanwhile, it conveniently expresses and unifies many important approaches such as data parallelism, ZeRO, Megatron-LM’s operator parallelism, and their combinations, which are not fully covered by existing automatic operators parallelism systems, such as Tofu [195] and FlexFlow [84]. Unlike systems that perform randomized search [84] or assume linear graphs [195], Alpa formalizes the problem as an integer linear programming (ILP) and shows it can be solved efficiently for computational graphs with tens of thousands of operators. Next, we describe the space of intra-op parallelism and our solution.

The Space of Intra-Operator Parallelism

Given an operator in the computational graph, there are multiple possible parallel algorithms to run it on a device mesh. For example, a matrix multiplication $C_{ij} = \sum_k A_{ik} B_{kj}$ corresponds to a three-level for-loop. To parallelize it, we can parallelize the loop i , loop j , loop k , or combinations of them across devices, which would have different computation and communication costs, require different layouts for the input tensors, and result in output tensors with different layouts. If an input tensor does not satisfy the layout requirement, a

Table 2.1: Sharding specs of a 2-dimensional tensor on a 2×2 device mesh. A is a (N, M) tensor. The device mesh is $[[\text{Device 0}, \text{Device 1}], [\text{Device 2}, \text{Device 3}]]$. Each device stores a partition of A . The first column is the name of the sharding spec. The latter columns use Numpy syntax to describe the partitions stored on each device.

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

Table 2.2: Several cases of resharding. $all_gather(x, i)$ means an all-gather of x bytes along the i -th mesh axis. M is the size of the tensor. (n_0, n_1) is the mesh shape.

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

layout conversion is required, which introduces extra communication costs. The goal of the intra-op pass is to pick one parallel algorithm for every operator to minimize the execution time of the entire graph. Next, we formally define the device mesh and the layout of a tensor and discuss the cost of layout conversion.

Device mesh. A device mesh is a 2-dimensional logical view of a set of physical devices. Each device in the mesh has the same compute capability. Devices can communicate along the first mesh dimension and the second mesh dimension with different bandwidths. We assume different groups of devices along the same mesh dimension have the same communication performance. For a set of physical devices, there can be multiple logical views. For example, given 2 nodes and 8 GPUs per node (i.e., 16 devices in total), we can view them as a 2×8 , 1×16 , 4×4 , 8×2 , or 16×1 device mesh. The mapping between physical devices and the logical device mesh view is optimized by the inter-op pass (§2.5). In the rest of this section, we consider one fixed device mesh view.

Table 2.3: Several parallel algorithms for a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$. The notation $all-reduce(x, i)$ means an all-reduce of x bytes along the i -th mesh axis. M is the size of the output tensor. (n_0, n_1) is the mesh shape.

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

Sharding Spec. We use *sharding spec* to define the layout of a tensor. For an N -dimensional tensor, its sharding spec is defined as $X_0 X_1 \cdots X_{n-1}$, where $X_i \in \{S, R\}$. If $X_i = S$, it means the i -th axis of the tensor is partitioned. Otherwise, the i -th axis is replicated. For example, for a 2-dimensional tensor (i.e., a matrix), SR means it is row-partitioned, RS means it is column-partitioned, SS means it is both row- and column- partitioned. RR means it is replicated without any partitioning. After we define which tensor axes are partitioned, we then have to map the partitioned tensor axes to mesh axes. We only consider 2-dimensional device meshes, so a partitioned tensor axis can be mapped to either the first or the second axis of the device mesh, or both. We added a superscript to S to denote the device assignment. For example, S^0 means the partitions are along the 0-th axis of the mesh, S^{01} means the partitions take place along both mesh axes. $S^0 R$ means the tensor is row-partitioned into two parts – The first part is replicated on device 0 and device 1, and the second part is replicated on device 2 and device 3. Table 2.1 shows all possible sharding specs of a 2-dimensional tensor on a 2×2 mesh with 4 devices.

Resharding. When an input tensor of an operator does not satisfy the sharding spec of the chosen parallel algorithm for the operator, a layout conversion, namely *resharding*, is required, which might require cross-device communication. Table 2.2 lists several cases of resharding. For instance, to convert a fully replicated tensor to any other sharding specs (case #1), we can slice the tensor locally without communication; to swap the partitioned axis (case #4), we perform an all-to-all.

Parallel algorithms of an operator. With the definitions above, consider parallelizing a batched matmul $C_{b,i,j} = \sum_k A_{b,i,k} B_{b,k,j}$ on a 2D mesh – Table 2.3 lists several intra-op parallel algorithms for a batched matmul. Algorithm#1 maps loop i to the 0-th mesh axis and loop j to the 1-th mesh axis, resulting in the output tensor C with a sharding spec $RS^0 S^1$. As the LHS operand $A_{b,i,k}$ and RHS operand $B_{b,k,j}$ both have only one parallelized index, their sharding specs are written as $RS^0 R$ and RRS^1 , respectively. In this algorithm, each device has all its required input tiles (i.e., a partition of the tensor) stored locally to compute its output tile, so there is no communication cost. In Algorithm #2 in Table 2.3, when the reduction loop k is parallelized, all-reduce communication is needed to aggregate

the partial sum. Similarly, we can derive the sharding specs and communication costs of other parallel algorithms for a batched matmul.

For other primitive operators such as convolution and reduction, we can get a list of possible parallel algorithms following a similar analysis of their math expressions. In the intra-op pass, the model graph is represented in XLA’s HLO format[179], which summarizes common DL operators into less than 80 primitive operators, so we can manually enumerate the possible parallel algorithms for every primitive operator.

ILP Formulation

The total execution cost of a computational graph $G = (V, E)$ is the sum of the compute and communication costs on all nodes $v \in V$ and the resharding costs on all edges $e \in E$. We formulate the cost minimization as an ILP and solve it optimally with an off-the-shelf solver [54].

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

$$\min_s \sum_{v \in V} s_v^T (c_v + d_v) + \sum_{(v,u) \in E} s_v^T R_{vu} s_u, \quad (2.1)$$

where the first term is the compute and communication cost of node v , and the second is the resharding cost of the edge (v, u) . In this formulation, s is the variable, and the rest are constant values. The term $s_v^T R_{vu} s_u$ in Eq. 2.1 is quadratic, and cannot be fed into an ILP solver. We linearize [55] the quadratic term by introducing a new decision vector $e_{vu} \in \{0, 1\}^{k_v \cdot k_u}$ which represents the resharding decision between node v and u .

Although we can use profiling to get the accurate costs for c_v , d_v , and R_{vu} , we use the following methods to estimate them for simplicity. For communication costs c_v and R_{vu} , we compute the numbers of communicated bytes and divide them by the mesh dimension bandwidth to get the costs. For compute costs d_v , we set all of them as zero following the same motivation in [195]. This is reasonable because: (1) For heavy operators such as matmul, we do not allow replicated computation. All parallel algorithms always evenly divide the work to all devices, so all parallel algorithms of one operator have the same arithmetic complexity; (2) For lightweight operators such as element-wise operators, we allow replicated computation of them, but their computation costs are negligible.

To simplify the graph, we merge computationally-trivial operators, such as element-wise operators, transpose, and reduction, into one of their operands and propagate the sharding spec from the operand. This greatly reduces the number of nodes in the graph, thus the ILP

problem size. We do a breath-first-search and compute the depth of each node. The node is merged to the deepest operand.

Once the parallel plan is decided by ILP, we also apply a set of post-ILP communication optimizations, such as replacing all-reduce with reduce-scatter and all-gather, whenever applicable, because the latter reduces the number of replicated tensors and corresponding computations, while keeping the communication volume the same. This achieves the effect of weight update sharding [210] or ZeRO optimizer [151].

2.5 Inter-Operator Parallelism

In this section, we develop methods to slice the model and device cluster into stage-mesh pairs. Our optimization goal is to minimize the *end-to-end* pipeline execution latency for the entire computational graph. Previous works [105, 51] have considered simplified problems, such as assuming the device for each stage is pre-assigned, and all stages have fixed data or operator parallelism plan. Alpa rids these assumptions by jointly considering device mesh assignment and the existence of varying intra-op parallelism plans on each stage.

The Space for Inter-Operator Parallelism

Assume the computational graph contains a sequence of operators following the topology order of the graph², notated as $o_1, \dots, o_K$, where the inputs of an operator o_k are from operators $o_1, \dots, o_{k-1}$. We slice the operators into S stages $s_1, \dots, s_S$, where each stage s_i consists of operators $(o_{l_i}, \dots, o_{r_i})$, and we assign each stage s_i to a submesh of size $n_i \times m_i$, sliced from a computer cluster that contains devices, notated as the *cluster mesh* with shape $N \times M$. Let $t_i = t_{\text{intra}}(s_i, \text{Mesh}(n_i, m_i))$ be the latency of executing stage s_i on a submesh of $n_i \times m_i$, minimized by the ILP and reported back by the intra-op pass (§2.4). As visualized in Fig. 2.5, assuming we have B different input microbatches for the pipeline, the total minimum latency³ for the entire computation graph is written as:

$$T^* = \min_{\substack{s_1, \dots, s_S; \\ (n_1, m_1), \dots, (n_S, m_S)}} \left\{ \sum_{i=1}^S t_i + (B - 1) \cdot \max_{1 \leq j \leq S} \{t_j\} \right\}. \quad (2.2)$$

The overall latency contains two terms: the first term is the total latency of all stages, interpreted as the latency of the first microbatch going through the pipeline; the second term is the pipelined execution time for the rest of $B - 1$ microbatches, which is bounded by the slowest stage (stage 3 in Fig. 2.5).

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

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

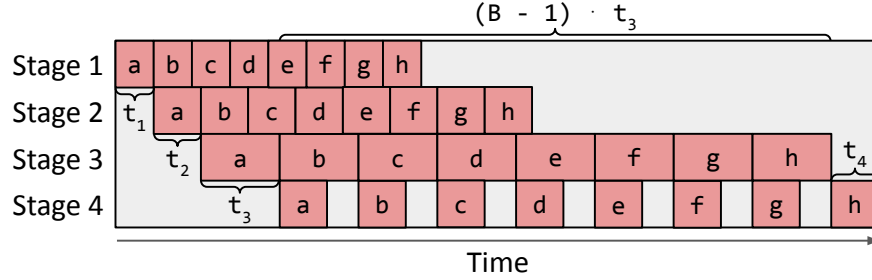


Figure 2.5: Illustration of the total latency of a pipeline, which is determined by two parts: the total latency of all stages ($t_1 + t_2 + t_3 + t_4$) and the latency of the slowest stage ($(B - 1) \cdot t_3$).

We aim to solve Eq. 2.2 with two additional constraints: (1) For an operator in the forward pass of the graph, we want to colocate it with its corresponded backward operator on the same submesh. Since backward propagation usually uses the similar set of tensors during forward propagation, this effectively reduces the amount of communication to fetch the required tensors generated at the forward pass to the backward pass. We use the sum of forward and backward latency for t_{intra} , so Eq. 2.2 reflects the total latency, including both forward and backward propagation. (2) We need the sliced submeshes $(n_1, m_1), \dots, (n_S, m_S)$ to fully cover the $N \times M$ cluster mesh – we do not waste any compute device resources. We next elaborate on our DP formulation.

DP Formulation

To ensure all submeshes $(n_1, m_1), \dots, (n_S, m_S)$ fully cover the $N \times M$ cluster mesh, we reduce the available submesh shapes into two options: (1) one-dimensional submeshes of sizes $(1, 1), (1, 2), (1, 4) \dots (1, 2^m)$ and (2) two-dimensional submeshes of size $(2, M), (3, M), \dots, (N, M)$ that fully use the second dimension of the cluster mesh (i.e., on a GPU cluster, this means using all compute devices in each physical machine). We include a theorem in the appendix⁴ that proves these submesh shapes can always fully cover the cluster mesh. To assign physical devices in the cluster to the resulting submeshes find by the DP algorithm, we enumerate by assigning devices to larger submeshes first and then to smaller ones. When there are multiple pipeline stages with the same submesh shape, we tend to put neighboring pipeline stages closer on the device mesh to reduce communication latency.

The simplification on submesh shapes works well for most available cloud deep learning setups: On AWS [9], the GPU instances have 1, 2, 4, or 8 GPUs; on GCP [63], the TPU instances have 8, 32, 128, 256 or 512 TPUs. The set of submesh shapes (n, m) excluded by the assumption is with $n > 1$ and $m < M$, which we observe lead to inferior results, since an alternative submesh with shape (n', M) where $n' \cdot M = n \cdot m$ has more devices that

⁴Please refer to the original paper [225]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

can communicate with high bandwidth. With this reduction, we only need to ensure that $\sum_{i=1}^S n_i \cdot m_i = N \cdot M$.

To find T^* in Eq. 2.2, we develop a DP algorithm. The DP first enumerates the second term $t_{max} = \max_{1 \leq j \leq S} t_j$ and minimizes the first term $t_{total}(t_{max}) = \sum_{1 \leq i \leq S} t_i$ for each different t_{max} . Specifically, we use the function $F(s, k, d; t_{max})$ to represent the minimal total latency when slicing operators o_k to o_K into s stages and putting them onto d devices so that the latency of each stage is less than t_{max} . We start with $F(0, K + 1, 0; t_{max}) = 0$, and derive the optimal substructure of F as

$$F(s, k, d; t_{max}) = \min_{\substack{k \leq i \leq K \\ n_s \cdot m_s \leq d}} \left\{ \begin{array}{l} t_{intra}((o_k, \dots, o_i), Mesh(n_s, m_s), s) \\ + F(s - 1, i + 1, d - n_s \cdot m_s; t_{max}) \end{array} \right\}, \quad (2.3)$$

and derive the optimal total latency as

$$T^*(t_{max}) = \min_s \{F(s, 0, N \cdot M; t_{max})\} + (B - 1) \cdot t_{max}. \quad (2.4)$$

The value of $t_{intra}((o_k, \dots, o_i), Mesh(n_s, m_s), s)$ is determined by the intra-op pass. It is the lowest latency of executing the subgraph $(o_k, \dots, o_i)$ on mesh $Mesh(n_s, m_s)$ with s subsequent stages. Note that $Mesh(n_s, m_s)$ is a set of physical devices – hence, we enumerate all the potential choices of logical device mesh shapes (n_l, m_l) satisfying $n_l \cdot m_l = n_s \cdot m_s$. For each choice, we query the intra-op pass with subgraph $(o_k, \dots, o_i)$, logical mesh (n_l, m_l) , and other intra-op options as inputs and get an intra-op plan. We then compile the subgraph with this plan and all other low-level compiler optimizations (e.g., fusion, memory planning) to get an executable for precise profiling. The executable is profiled in order to get the stage latency (t_l) and the memory required on each device to run the stage (mem_{stage}) and to store the intermediate activations (mem_{act}). We check whether the required memory fits the device memory (mem_{device}) according to the chosen pipeline execution schedule. For example, for 1F1B schedule [51, 129], we check

$$mem_{stage} + s \cdot mem_{act} \leq mem_{device}. \quad (2.5)$$

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

Our algorithm builds on top of that in TeraPipe [105]. However, TeraPipe assumes all pipeline stages are the same, and the goal is to find the optimal way to batch input tokens into micro-batches of different sizes. Instead, Alpa aims to group the operators of a computational graph into different pipeline stages, while assuming the input micro-batches are of the same size. In addition, Alpa optimizes the mesh shape in the DP algorithm for each pipeline stage in inter-op parallelism.

Complexity. Our DP algorithm computes the slicing in $O(K^3 NM(N + \log(M)))$ time for a fixed t_{max} . t_{max} has at most $O(K^2(N + \log(M)))$ choices: $t_{intra}((o_i, \dots, o_j), Mesh(n_s, m_s))$

for $i, j = 1, \dots, K$ and all the submesh choices. The complexity of this DP algorithm is thus $O(K^5 NM(N + \log(M))^2)$.

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

Performance optimization #1: early pruning. We use one optimization that is similar to that in TeraPipe [105]. We enumerate t_{max} from small to large. When $B \cdot t_{max}$ is larger than the current best T^* , we immediately stop the enumeration. This is because larger t_{max} can no longer provide a better solution. Also, during enumeration of t_{max} , we only evaluate a choice of t_{max} if it is sufficiently larger than the last t_{max} (by at least ϵ). This allows the gap between the solution found by the DP algorithm and the global optima to be at most $B \cdot \epsilon$. We empirically choose $\epsilon = 10^{-6}$ s, and we find that the solution output by our algorithm is the same as the real optimal solution ($\epsilon = 0$) for all our evaluated settings.

Performance optimization #2: operator clustering. Many operators in a computational graph are not computationally intensive (e.g., ReLU), and the exact placement of these operators has little impact on the total execution time. We develop another DP algorithm [10] to cluster neighboring operators to reduce the total size of the graph Eq. 2.2 works on. We cluster the operators $(o_1, \dots, o_K)$ into a series of layers⁵ $(l_1, \dots, l_L)$, where $L \ll K$. The goal of the algorithm is to merge two types of operators: (1) those that do not call for much computation but lengthen the computational graph and (2) neighboring operators that may cause substantial communication if put on different device meshes. We define function $G(k, r)$ as the minimum of maximal amount of data received by a single layer when clustering operators $(o_1, \dots, o_k)$ into r layers. Note that G has the following optimal substructure:

$$G(k, r) = \min_{1 \leq i \leq k} \left\{ \begin{array}{l} \max\{G(i-1, r-1), C(i, k)\} \\ FLOP(o_i, \dots, o_k) \leq \frac{(1+\delta)FLOP_{total}}{L} \end{array} \right\}, \quad (2.6)$$

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

Alg. 1 summarizes the workflow of the inter-op pass and illustrates its interactions with the intra-op pass in §2.4.

⁵Note that the clustering does not exactly reproduce the layers with original machine learning semantics in the model definition.

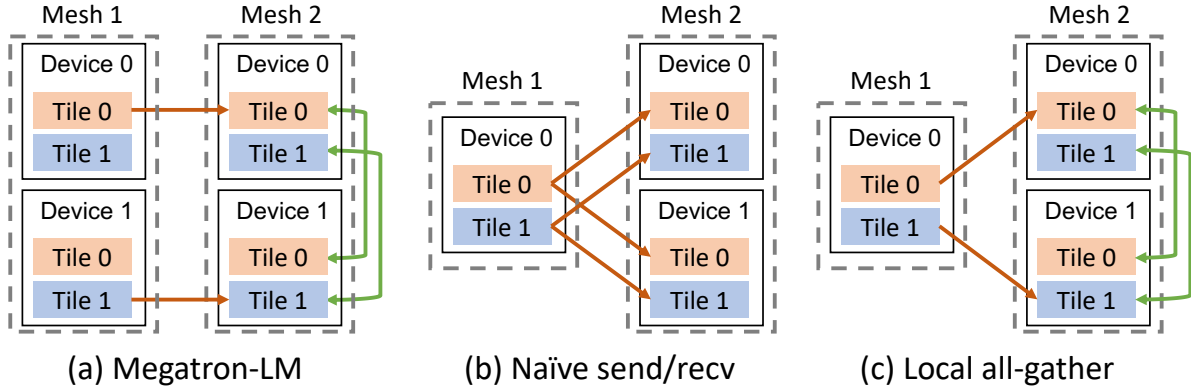


Figure 2.6: Cross-mesh resharding. Red arrows denote send/recv on slow connections. Green arrows denote all-gather on fast connections. (a) The scatter-gather optimization for equal mesh shapes in Megatron-LM. (b) The naïve send/recv for unequal mesh shapes. (c) The generalized local all-gather optimization for unequal mesh shapes.

2.6 Parallelism Orchestration

After stages, device meshes, and their assignments are decided, at the intra-op level, Alpa compiles each stage against its assigned device mesh, respecting the intra-op parallelism plan output by the ILP solver. The compilation depends on XLA [179] and GSPMD [211], and generates parallel executables for each stage-mesh pair. When needed, the compilation automatically inserts collective communication primitives (see §2.4) to address the *within-mesh* communication caused by intra-op parallelism.

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

Cross-mesh resharding. Existing manual systems, such as Megatron-LM [152, 170], constrain all pipeline stages to have the same degrees of data and tensor model parallelism, so the communication between pipeline stages is trivially realized by P2P send/recv between corresponded devices of two equivalent device meshes (Fig. 2.6a). In Alpa, the device meshes holding two adjacent stages might have different mesh shapes, and the tensor to communicate between two stages might have different sharding specs (Fig. 2.6b and Fig. 2.6c). We call this communication pattern as *cross-mesh resharding*, which is a many-to-many multicast problem.

Given the sharding specs of the tensor on the sender and receiver mesh, Alpa generates a communication plan to address cross-mesh sharding in two iterations. In the first iteration, Alpa calculates the correspondences between tensor partitions (a.k.a. tiles) on the source and destination mesh. Based on that, it generates P2P send/recv primitives between the source devices and destination devices to fulfill the communication. It then takes a second iteration to identify opportunities where the destination tensor has a replication in its sharding spec.

In this case, the tensor only needs to be transferred once between two meshes, then exchanged via all-gather across the devices on the destination mesh using its higher bandwidth (Fig. 2.6) – it rewrites send/recv generated at the first iteration into all-gather to avoid repeated communication.

We call this approach as *local all-gather* cross-mesh resharding. Since the communication between stages is normally small by our design, our experiments show that it performs satisfactorily well (§2.8). We defer the development of the optimal cross-mesh resharding plan to future work.

Generating pipeline execution instructions. As the final step, Alpa generates static execution instructions to launch the training on clusters. Since each stage has different sets of operators and may locate on meshes with different shapes, in contrast to many SPMD pipeline-parallel training systems [211, 128], Alpa adopts an MPMD-style runtime to orchestrate the inter-op parallel execution – Alpa generates distinct static execution instructions for each device mesh.

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

2.7 Limitations and Discussion

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

Compared to existing work that manually combines data, operator, and pipeline parallelism, such as 3D parallelism [152] and PTD-P [128], Alpa’s hierarchical view of inter- and intra-op parallelisms significantly advances them with three major flexibility: (1) pipeline stages can contain an uneven number of operators or layers; (2) pipeline stages in Alpa might be mapped to device meshes with different shapes; (3) within each stage, the data and operator parallelism configuration is customized non-uniformly on an operator-by-operator basis. Together, they allow Alpa to unify all existing model parallelism approaches and generalize to model architectures and cluster setups with more heterogeneity.

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

- Alpa does not model the communication cost between different stages because the cross-stage communication cost is *by nature small*. In fact, modeling the cost in either the DP or ILP is possible, but would require enumerating exponentially more intra-op passes and DP states.
- The inter-op pass currently has a hyperparameter: the number of micro-batches B , which is not optimized by our current formulation but can be searched by enumeration.

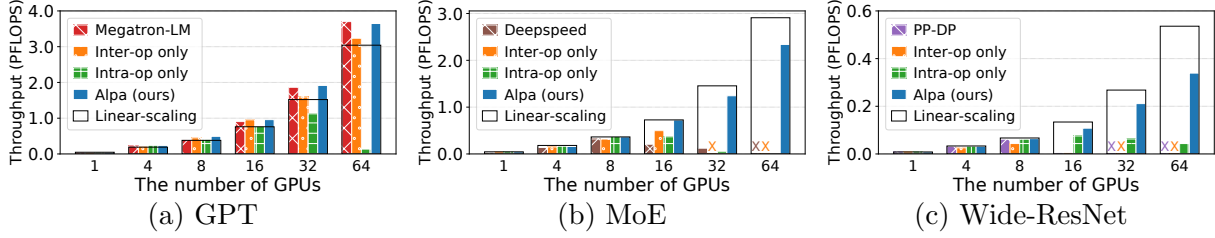


Figure 2.7: End-to-end evaluation results. “x” denotes out-of-memory. Black boxes represent linear scaling.

- The inter-op pass models pipeline parallelism with a static linear schedule, without considering more dynamic schedules that, for example, parallelize different branches in a computational graph on different devices.
- Alpha does not optimize for the best scheme of overlapping computation and communication; Alpha can only handle static computational graphs with all tensor shapes known at compilation time.

Nevertheless, our results on weak scaling (§4.6) suggest that Alpha is able to generate near-optimal execution plans for many notable models.

2.8 Evaluation

Alpha is implemented using about 16K LoC in Python and 6K LoC in C++. Alpha uses Jax as the frontend and XLA as the backend. The compiler passes are implemented on Jax’s and XLA’s intermediate representation (i.e., Jaxpr and HLO). For the distributed runtime, we use Ray [125] actor to implement the device mesh worker, XLA runtime for executing computation, and NCCL [133] for communication.

We evaluate Alpha on training large-scale models with billions of parameters, including GPT-3 [25], GShard Mixture-of-Experts (MoE) [99], and Wide-ResNet [221]. The testbed is a typical cluster consisting of 8 nodes and 64 GPUs. Each node is an Amazon EC2 p3.16xlarge instance with 8 NVIDIA V100 16 GB GPUs, 64 vCPUs, and 488 GB memory. The 8 GPUs in a node are connected via NVLink. The 8 nodes are launched within one placement group with 25Gbps cross-node bandwidth.

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

End-to-End Performance

Models and training workloads. We target three types of models listed in Table 2.4, covering models with both homogeneous and heterogeneous architectures. GPT-3 is a ho-

Table 2.4: Models used in the end-to-end evaluation. LM = language model. IC = image classification.

Model	Task	Batch size	#params (billion)	Precision
GPT-3 [25]	LM	1024	0.35, 1.3, 2.6, 6.7, 15, 39	FP16
GShard MoE [99]	LM	1024	0.38, 1.3, 2.4, 10, 27, 70	FP16
Wide-ResNet [221]	IC	1536	0.25, 1.0, 2.0, 4.0, 6.7, 13	FP32

homogeneous transformer-based LM by stacking many transformer layers whose model parallelization plan has been extensively studied [170, 128]. GShard MoE is a mixed dense and sparse LM, where mixture-of-experts layers are used to replace the MLP at the end of a transformer, every two layers. Wide-ResNet is a variant of ResNet with larger channel sizes. It is vastly different from the transformer models and there are no existing manually designed strategies.

To study the ability to train large models, we follow common ML practice to scale the model size along with the number of GPUs, with the parameter range reported in Table 2.4. More precisely, for GPT-3, we increase the hidden size and the number of layers together with the number of GPUs following [128], whereas for MoE we mainly increase the number of experts suggested by [99, 211]. For Wide-ResNet, we increase the channel size and width factor in convolution layers. For each model, we adopt the suggested global batch size per ML practice [128, 25, 99, 221] to keep the same statistical behavior. We then tune the best microbatch size for each model and system configuration that maximizes the system performance. The gradients are accumulated across microbatches. The detailed model specifications are provided in the appendix⁶.

Baselines. For each model, we compare Alpa against a strong baseline. We use Megatron-LM v2 [128] as the baseline system for GPT-3. Megatron-LM is the state-of-the-art system for training homogeneous transformer-based LMs on GPUs. It combines data parallelism, pipeline parallelism, and manually-designed operator parallelism (denoted as TMP later). The combination of these techniques is controlled by three integer parameters that specify the parallelism degrees assigned to each technique. We grid-search the three parameters following the guidance of their paper and report the results of the best configuration. Megatron-LM is specialized for GPT-like models, so it does not support other models in Table 2.4.

We use DeepSpeed [152] as the baseline for MoE. DeepSpeed provides a state-of-the-art implementation for training MoE on GPUs. It combines handcrafted operator parallelism for MoE layers and ZeRO-based [151] data parallelism. The combination of these techniques is controlled by several integer parameters that specify the parallelism degree assigned to each technique. We also grid-search them and report the best results. The performance of DeepSpeed on GPT-3 is similar to or worse than Megatron-LM, so we skip it on GPT-3. Note that original GShard-MoE [99] implementation is only available on TPUs, thus we do not include its results, though their strategies [99] are covered by Alpa’s strategy space.

For large Wide-ResNet, there is no specialized system or manually designed plan for it.

⁶Please refer to the original paper [225]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

We use Alpa to build a baseline “PP-DP” whose space only consists of data parallelism and pipeline parallelism, which mimics the parallelism space of PipeDream [130] and Dapple [51].

For all models, we also include the results of using Alpa with only one of intra- and inter-operator parallelism, which mimics the performance of some other auto-parallel systems. The open-source Flexflow [84] does not support the models we evaluate, as it lacks support for many necessary operators (e.g., layer normalization [11], mixed-precision operators). Tofu [195] only supports single node execution and is not open-sourced. Due to both theoretical and practical limitations, we do not include their results and we do not expect Flexflow or Tofu to outperform the state-of-the-art manual baselines in our evaluation.

Evaluation metrics. Alpa does not modify the semantics of the synchronous gradient descent algorithm, thus does not affect the model convergence. Therefore, we measure training throughput in our evaluation. We evaluate weak scaling of the system when increasing the model size along with the number of GPUs. Following [128], we use the aggregated peta floating-point operations per second (PFLOPS) of the whole cluster as the metric⁷. We measure it by running a few batches with dummy data after proper warmup. All our results (including those in later sections) have a standard deviation within 0.5%, so we skip the error bars in our figures.

GPT-3 results. The parallelization plan for GPT-3 has been extensively studied [25, 128, 105]. We observe in Fig. 2.7a that this manual plan with the best grid-searched parameters enables Megatron-LM to achieve super-linear weak scaling on GPT-3. Nevertheless, compared to Megatron-LM, Alpa automatically generates execution plans and even achieves slightly better scaling on several settings. If compared to methods that only use intra-operator parallelism, our results are consistent with recent studies – “Intra-op only” performs poorly on 16 GPUs because even the best plan has to communicate tensors heavily on cross-node connections, making communication a bottleneck. Surprisingly, “Inter-op only” performs well and maintains linear scaling on up to 64 GPUs.

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

⁷As the models are different for different numbers of GPUs, we cannot measure scaling on the system throughput such as tokens per second or images per second.

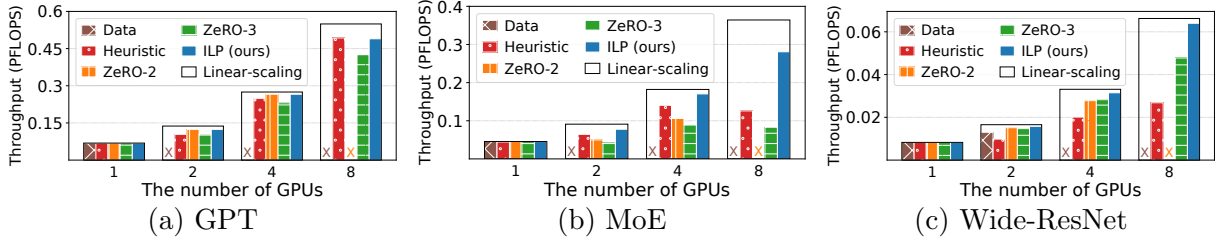


Figure 2.8: Intra-operator parallelism ablation study. “x” denotes out-of-memory. Black boxes represent linear scaling.

to the slight performance improvement over Megatron-LM. This attributes to the fact that Alpa, as a generic compiler system, can compose a wide range of parallelism approaches, while Megatron-LM, for now, misses weight update sharding support.

MoE results. DeepSpeed adopts a manual operator parallelism plan for MoE models, developed by GShard [99], called *expert parallelism*, which uses a simple rule: it partitions the expert axis for the operators in MoE layers, but switches back to data parallelism for non-expert layers. This expert parallelism is then combined with ZeRO data parallelism and TMP. All of these techniques belong to intra-operator parallelism. Unfortunately, DeepSpeed’s specialized implementation does not include any inter-operator parallelism approach, which is required for scaling across multiple nodes with low inter-node bandwidth. Therefore, DeepSpeed only maintains a good performance within a node (≤ 8 GPUs) on this cluster. “Intra-op only” fails to scale across multiple nodes due to the same reason. “Inter-op only” runs out of memory on 32 GPUs and 64 GPUs because it is not easy to equally slice the model when the number of GPUs is larger than the number of layers of the model. The imbalanced slicing makes some memory-intensive stages run out of memory.

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

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

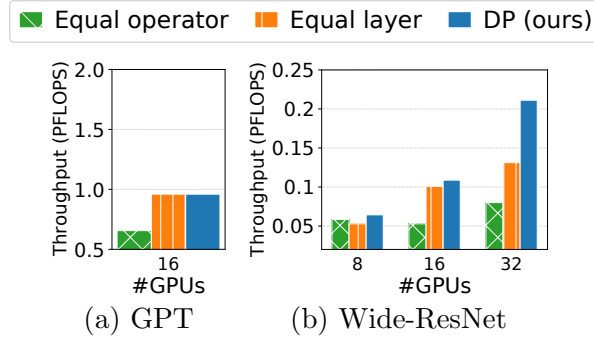


Figure 2.9: Inter-operator parallelism ablation study.

plan for Wide-ResNet is in §2.8.

Intra-Op Parallelism Ablation Study

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

Experimental setup. We run a weak scaling benchmark in terms of model size similar to §2.8, but disable pipeline parallelism and gradient accumulation to control variables. The benchmark is done on one AWS p3.16xlarge instance with 8 GPUs. In order to simulate an execution environment of large-scale training in one node, we use larger hidden sizes, smaller batch sizes, and smaller numbers of layers, compared to the model configurations in §2.8.

Baselines. We compare automatic solutions for intra-operator parallelism. “Data” is vanilla data parallelism. “ZeRO-2” [151] is a memory-efficient version of data parallelism which partitions gradients and optimizer states. “ZeRO-3” [151] additionally partitions parameters on top of “ZeRO-2”. “Heuristic” uses a rule combined with the sharding propagation in GSPMD. It marks the largest dimension of every input tensor as partitioned and runs sharding propagation to get the sharding specs for all nodes in the graph. “ILP” is our solution based on the ILP solver.

Results. As shown in Fig. 2.8, “Data” runs out of memory quickly and cannot train large models. “ZeRO-2” and “ZeRO-3” resolve the memory problem of data parallelism, but they do not optimize for communication as they always communicate the gradients. When the gradients are much larger than activations, their performance degenerates. “Heuristic” solves the memory issue by partitioning all tensors, but can be slowed down by larger communication. “Auto-sharding” performs best in all cases and maintains a near-linear scaling, because it figures out the correct partition plan that always minimizes the communication overhead.

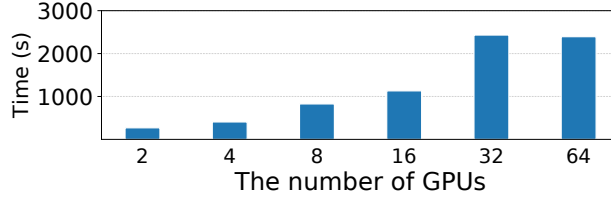


Figure 2.10: Alpha’s compilation time on all GPT models. The model size and #GPUs are simultaneously scaled.

Inter-Op Parallelism Ablation Study

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

Experimental setup. We report the performance of three variants of our DP algorithm on GPT and Wide-ResNet. The benchmark settings are the same as the settings in §2.8.

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

Results. Fig. 2.9 shows the result. “DP” always outperforms “Equal operator”. This is because “Equal operator” merges operator that should be put onto different device meshes. Alpha’s algorithm can cluster operators based on the communication cost and computation balance. Whether “DP” can outperform “Equal layer” depends on the model architecture. On homogeneous models like GPT, the solution of our DP algorithm uses the same number of layers for all stages, so “Equal layer” performs the same as “DP”. On Wide-ResNet, the optimal solution can assign different layers to different stages, so “Equal layer” is worse than the full flexible DP algorithm. For Wide-ResNet on 32 GPUs, our algorithm outperforms “Equal operator” and “Equal layer” by $2.6\times$ and $1.6\times$, respectively.

Compilation Time

Fig. 2.10 shows Alpha’s compilation time for all the GPT settings in §2.8. The compilation time is a single run of the full Alg. 1 with a provided number of microbatches B . According to the result, Alpha scales to large models or large clusters well, because compilation time grows linearly with the size of the model and the number of GPUs in the cluster. Table 2.5 reports the compilation time breakdown for the largest GPT model in our evaluation (39B, 64 GPUs). Most of the time is spent on enumerating stage-mesh pairs and profiling them. For the compilation part, we accelerate it by compiling different stages in parallel with distributed workers. For profiling, we accelerate it using a simple cost model built at the XLA instruction level, which estimates the cost of matrix multiplication and communication primitives with a piece-wise linear model. With these optimizations, the compilation and

Table 2.5: Compilation time breakdown of GPT-39B.

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

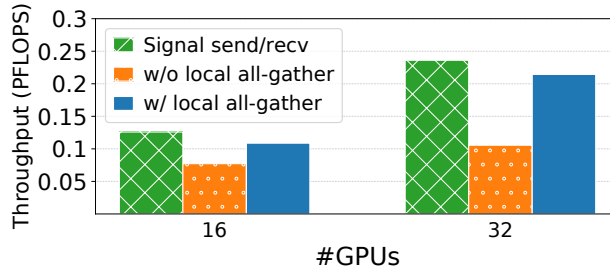


Figure 2.11: Cross-mesh resharding on Wide-ResNet.

search for a model take at most several hours, which is acceptable as it is much shorter than the actual training time, which can take several weeks.

Cross-Mesh Resharding

We evaluate our generalized local all-gather optimization for cross-mesh resharding between meshes with different shapes on Wide-ResNet, as shown in Fig. 2.11. “signal send/rcv” is a synthetic case where we only send 1 signal byte between stages, which can be seen as the upper bound of the performance. “w/o local all-gather” disables our local all-gather optimization and uses only send/rcv. “w/ local all-gather” enables our local all-gather optimization to move more communication from slow connections to fast local connections, which brings $2.0\times$ speedup on 32 GPUs.

Case Study: Wide-ResNet

We visualize the parallelization strategies Alpha finds for Wide-ResNet on 16 GPUs in Fig. 2.12. We also include the visualization of results on 4 and 8 GPUs in the appendix⁸.

On 4 GPUs, Alpha uses only intra-operator parallelism. The intra-operator solution partitions along the batch axis for the first dozens of layers and then switches to partitioning the channel axis for the last few layers. On 16 GPUs, Alpha slices the model into 3 stages and assigns 4, 4, 8 GPUs to stage 1, 2, 3, respectively. Data parallelism is preferred in the

⁸Please refer to the original paper [225]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

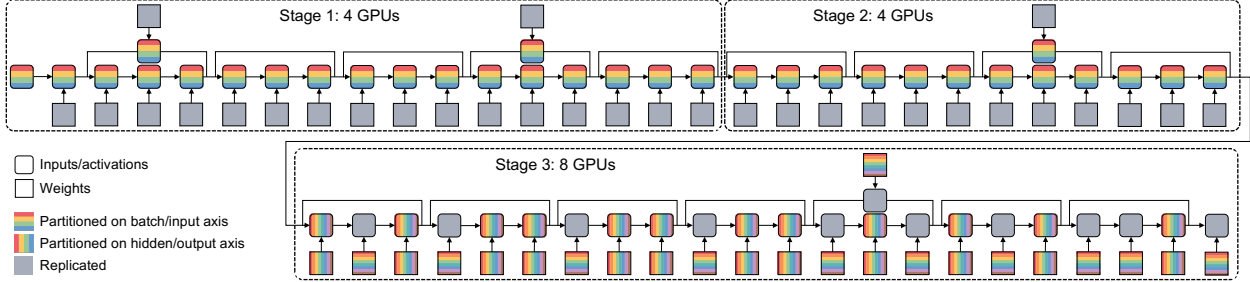


Figure 2.12: Visualization of the parallel strategy of Wide-ResNet on 16 GPUs. Different colors represent the devices a tensor is distributed on. Grey blocks indicate a tensor is replicated across the devices. The input data and resulting activation of each convolution and dense layer can be partitioned along the batch axis and the hidden axis. The weights can be partitioned along the input and output channel axis.

first two stages because the activation tensors are larger than weight tensors. In the third stage, the ILP solver finds a non-trivial way of partitioning the convolution operators. The result shows that it can be opaque to manually create such a strategy for a heterogeneous model like Wide-ResNet, even for domain experts.

2.9 Related Work

Systems for data-parallel training. Horovod [163] and PyTorchDDP [103] are two commonly adopted data-parallel training systems that synchronize gradients using all-reduce. BytePS [146, 86] unifies all-reduce and parameter servers and utilizes heterogeneous resources in data center clusters. AutoDist [223] uses learning-based approaches to compose a data-parallel training strategy. ZeRO [151, 210] improves the memory usage of data parallelism by reducing replicated tensors. MiCS [224] minimizes the communication scale on top of ZeRO for better scalability on the public cloud. In Alpa, data parallelism [89] reduces to a special case of intra-operator parallelism – partitioned along the batch axis.

Systems for model-parallel training. The two major classes of model parallelisms have been discussed in §4.2. Mesh-TensorFlow [164], GSPMD [211, 99] and OneFlow [220] provide annotation APIs for users to manually specify the intra-op parallel plan. ColocRL [121] puts disjoint model partitions on different devices *without pipelining*, thereby the concurrency happens only when there exist parallel branches in the model. In contrast, Gpipe [79] splits the input data into micro-batches and forms pipeline parallelisms. PipeDream [130, 129] improves GPipe by using asynchronous training algorithms, reducing memory usage, and integrating it with data parallelism. However, PipeDream is asynchronous while Alpa is a synchronous training system. TeraPipe [105] discovers a new pipeline parallelism dimension for transformer-based LMs. Google’s Pathway system [17] is a concurrent work of Alpa. Pathway advocates a single controller runtime architecture combining “single program multiple data” (SPMD) and “multiple program multiple data” (MPMD) model. This is similar

to Alpa’s runtime part, where SPMD is used for intra-op parallelisms and MPMD is used for inter-op parallelism.

Automatic search for model-parallel plans. Another line of work focuses on the automatic discovery of model-parallel training plans. Tofu [195] develops a dynamic programming algorithm to generate the optimal intra-op strategy for *linear* graphs on *a single node*. FlexFlow [84] proposes a “SOAP” formulation and develops an MCMC-based randomized search algorithm. However, it only supports device placement without pipeline parallelism. Its search algorithm cannot scale to large graphs or clusters and does not have optimality guarantees. TensorOpt [27] develops a dynamic programming algorithm to automatically search for intra-op strategies that consider both memory and computation cost. Varuna [8] targets low-bandwidth clusters and focuses on automating pipeline and data parallelism. Piper [178] also finds a parallel strategy with both inter- and intra-op parallelism, but it relies on manually designed intra-op parallelism strategies and analyzes on a uniform network topology and asynchronous pipeline parallel schedules.

Techniques for training large-scale models. In addition to parallelization, there are other complementary techniques for training large-scale models, such as memory optimization [33, 83, 28, 76, 90, 155], communication compression [14, 191], and low-precision training [120]. Alpa can incorporate many of these techniques. For example, Alpa uses rematerialization to reduce memory usage and uses mixed-precision training to accelerate computation.

Compilers for deep learning. Compiler techniques have been introduced to optimize the execution of DL models [180, 179, 34, 226, 85, 117, 193]. Most of them focus on optimizing the computation for a single device. In contrast, Alpa is a compiler that supports a comprehensive space of execution plans for distributed training.

Distributed tensor computation in other domains. Besides deep learning, libraries and compilers for distributed tensor computation have been developed for linear algebra [20] and stencil computations [44]. Unlike Alpa, they do not consider necessary parallelization techniques for DL.

2.10 Conclusion

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

Algorithm 1 Inter-op pass summary.

```

1: Input: Model graph  $G$  and cluster  $C$  with shape  $(N, M)$ .
2: Output: The minimal pipeline execution latency  $T^*$ .
3: // Preprocess graph.
4:  $(o_1, \dots, o_K) \leftarrow \text{Flatten}(G)$ 
5:  $(l_1, \dots, l_L) \leftarrow \text{OperatorClustering}(o_1, \dots, o_K)$ 
6: // Run the intra-op pass to get costs of different stage-mesh pairs.
7:  $\text{submesh\_shapes} \leftarrow \{(1, 1), (1, 2), (1, 4), \dots, (1, M)\} \cup \{(2, M), (3, M), \dots, (N, M)\}$ 
8: for  $1 \leq i \leq j \leq L$  do
9:    $\text{stage} \leftarrow (l_i, \dots, l_j)$ 
10:  for  $(n, m) \in \text{submesh\_shapes}$  do
11:    for  $s$  from 1 to  $L$  do
12:       $t_{\text{intra}}(\text{stage}, \text{Mesh}(n, m), s) \leftarrow \infty$ 
13:    end for
14:    for  $(n_l, m_l), \text{opt} \in \text{LogicalMeshShapeAndIntraOp Options}(n, m)$  do
15:       $\text{plan} \leftarrow \text{IntraOpPass}(\text{stage}, \text{Mesh}(n_l, m_l), \text{opt})$ 
16:       $t_l, \text{mem}_{\text{stage}}, \text{mem}_{\text{act}} \leftarrow \text{Profile}(\text{plan})$ 
17:      for  $s$  satisfies Eq. 2.5 do
18:        if  $t_l < t_{\text{intra}}(\text{stage}, \text{Mesh}(n, m), s)$  then
19:           $t_{\text{intra}}(\text{stage}, \text{Mesh}(n, m), s) \leftarrow t_l$ 
20:        end if
21:      end for
22:    end for
23:  end for
24: end for
25: // Run the inter-op dynamic programming
26:  $T^* \leftarrow \infty$ 
27: for  $t_{\text{max}} \in \text{SortedAndFilter}(t_{\text{intra}}, \varepsilon)$  do
28:  if  $B \cdot t_{\text{max}} \geq T^*$  then
29:    break
30:  end if
31:   $F(0, L + 1, 0; t_{\text{max}}) \leftarrow 0$ 
32:  for  $s$  from 1 to  $L$  do
33:    for  $l$  from  $L$  down to 1 do
34:      for  $d$  from 1 to  $N \cdot M$  do
35:        Compute  $F(s, l, d; t_{\text{max}})$  according to Eq. 2.3
36:      end for
37:    end for
38:  end for
39:   $T^*(t_{\text{max}}) \leftarrow \min_s \{F(s, 0, N \cdot M; t_{\text{max}})\} + (B - 1) \cdot t_{\text{max}}$ 
40:  if  $T^*(t_{\text{max}}) < T^*$  then
41:     $T^* \leftarrow T^*(t_{\text{max}})$ 
42:  end if
43: end for

```

Chapter 3

Generating High-Performance Tensor Programs for Deep Learning

High-performance tensor programs are crucial to guarantee efficient execution of deep neural networks. However, obtaining performant tensor programs for different operators on various hardware platforms is notoriously challenging. Currently, deep learning systems rely on vendor-provided kernel libraries or various search strategies to get performant tensor programs. These approaches either require significant engineering effort to develop platform-specific optimization code or fall short of finding high-performance programs due to restricted search space and ineffective exploration strategy.

We present Ansor, a tensor program generation framework for deep learning applications. Compared with existing search strategies, Ansor explores many more optimization combinations by sampling programs from a hierarchical representation of the search space. Ansor then fine-tunes the sampled programs with evolutionary search and a learned cost model to identify the best programs. Ansor can find high-performance programs that are outside the search space of existing state-of-the-art approaches. In addition, Ansor utilizes a task scheduler to simultaneously optimize multiple subgraphs in deep neural networks. We show that Ansor improves the execution performance of deep neural networks relative to the state-of-the-art on the Intel CPU, ARM CPU, and NVIDIA GPU by up to $3.8\times$, $2.6\times$, and $1.7\times$, respectively.

This chapter is based on my previous paper [226], which was a collaborative effort with Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, Ion Stoica. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

3.1 Introduction

Low-latency execution of deep neural networks (DNN) plays a critical role in autonomous driving [42], augmented reality [4], language translation[47], and other applications of AI. DNNs can be expressed as a directed acyclic computational graph (DAG), in which nodes represent the operators (*e.g.*, convolution, matrix multiplication) and directed edges represent the dependencies between operators. Existing deep learning frameworks (*e.g.*, Tensorflow [1], PyTorch [143], MXNet [32]) map the operators in DNNs to vendor-provided kernel libraries (*e.g.*, cuDNN [35], MKL-DNN [81]) to achieve high performance. However, these kernel libraries require significant engineering effort to manually tune for each hardware platform and operator. The significant manual effort required to produce efficient operator implementations for each target accelerator limits the development and innovation of new operators [16] and specialized accelerators [124].

Given the importance of DNNs’ performance, researchers and industry practitioners have turned to search-based compilation [34, 2, 229, 185, 113] for automated generation of *tensor programs*, *i.e.*, low-level implementations of tensor operators. For an operator or a (sub-)graph of multiple operators, users define the computation in a high-level declarative language (section 4.2), and the compiler then searches for programs tailored towards different hardware platforms.

To find performant tensor programs, it is necessary for a search-based approach to explore a large enough search space to cover all the useful tensor program optimizations. However, existing approaches fail to capture many effective optimization combinations, because they rely on either predefined manually-written templates (*e.g.*, TVM [31], FlexTensor [229]) or aggressive pruning by evaluating incomplete programs (*e.g.*, Halide auto-scheduler [2]), which prevents them from covering a comprehensive search space (section 4.2). The rules they use to construct the search space are also limited.

In this paper, we explore a novel search strategy for generating high-performance tensor programs. It can automatically generate a large search space with comprehensive coverage of optimizations and gives every tensor program in the space a chance to be chosen. It thus enables to find high-performance programs that existing approaches miss.

Realizing this goal faces multiple challenges. First, it requires automatically constructing a large search space to cover as many tensor programs as possible for a given computation definition. Second, we need to search efficiently without comparing incomplete programs in the large search space that can be orders of magnitude larger than what existing templates can cover. Finally, when optimizing an entire DNN with many subgraphs, we should recognize and prioritize the subgraphs that are critical to the end-to-end performance.

To this end, we design and implement *Ansor*, a framework for automated tensor program generation. Ansor utilizes a hierarchical representation to cover a large search space. This representation decouples high-level structures and low-level details, enabling flexible enumeration of high-level structures and efficient sampling of low-level details. The space is constructed automatically for a given computation definition. Ansor then samples complete programs from the search space and fine-tunes these programs with evolutionary search

and a learned cost model. To optimize the performance of DNNs with multiple subgraphs, Ansor dynamically prioritizes subgraphs of the DNNs that are more likely to improve the end-to-end performance.

We evaluate Ansor on both standard deep learning benchmarks and emerging new workloads against manual libraries and state-of-the-art search-based frameworks. Experiment results show that Ansor improves the execution performance of DNNs on the Intel CPU, ARM CPU, and NVIDIA GPU by up to $3.8\times$, $2.6\times$, and $1.7\times$, respectively. For most computation definitions, the best program found by Ansor is outside the search space of existing search-based approaches. The results also show that, compared with existing search-based approaches, Ansor searches more efficiently, generating higher-performance programs in a shorter time, despite its larger search space. Ansor can match the performance of a state-of-the-art framework with an order of magnitude less search time. Besides, Ansor enables automatic extension to new operators by only requiring their mathematical definitions without manual templates.

In summary, this paper makes the following contributions:

- A mechanism to generate a large hierarchical search space of tensor programs for a computational graph.
- An evolutionary strategy with a learned cost model to fine-tune the performance of tensor programs.
- A scheduling algorithm based on gradient descent to prioritize important subgraphs when optimizing the end-to-end performance of DNNs.
- An implementation and comprehensive evaluation of the Ansor system demonstrating that the above techniques outperform state-of-the-art systems on a variety of DNNs and hardware platforms.

3.2 Background

The deep learning ecosystem is embracing a rapidly growing diversity of hardware platforms including CPUs, GPUs, FPGAs, and ASICs. In order to deploy DNNs on these platforms, high-performance tensor programs are needed for the operators used in DNNs. The required operator set typically contains a mixture of standard operators (*e.g.*, `matmul`, `conv2d`) and novel operators invented by machine learning researchers (*e.g.*, `capsule conv2d` [73], `dilated conv2d` [217]).

To deliver portable performance of these operators on a wide range of hardware platforms in a productive way, multiple compiler techniques have been introduced (*e.g.*, TVM [34], Halide [149], Tensor Comprehensions [185]). Users define the computation in a form similar to mathematical expressions using a high-level declarative language, and the compiler generates optimized tensor programs according to the definition. Figure 3.1 shows the computation definition of matrix multiplication in the TVM tensor expression language. Users

Matrix Multiplication $C_{i,j} = \sum_k A_{i,k} B_{k,j}$

```
C = compute((N, M), lambda i, j: sum(A[i, k]*B[k, j], [k]))
```

Figure 3.1: The computation definition of matrix multiplication.

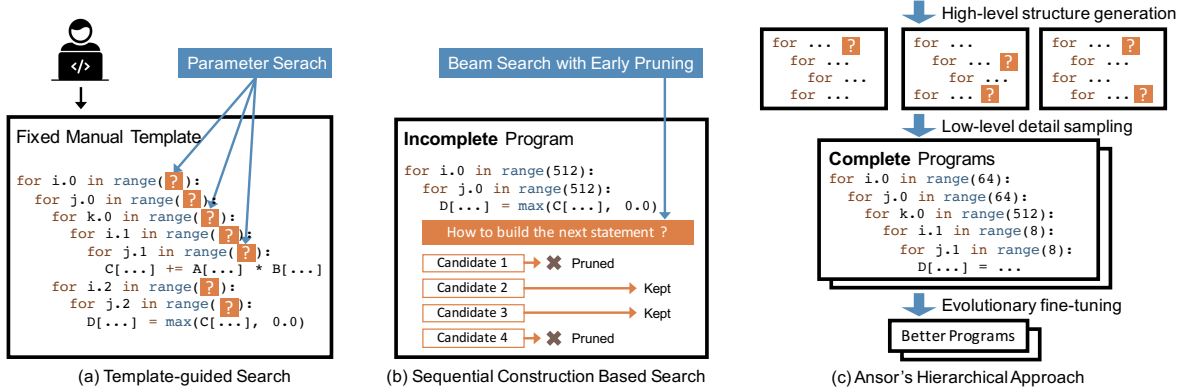


Figure 3.2: Search strategy comparison. The pseudo-code shows tensor programs with loop nests. The question marks in orange background denote low-level parameters.

mainly need to define the shapes of the tensors and how each element in the output tensor is computed.

However, automatically generating high-performance tensor programs from a high-level definition is extremely difficult. Depending on the architecture of the target platform, the compiler needs to search in an extremely large and complicated space containing combinatorial choices of optimizations (*e.g.*, tile structure, tile size, vectorization, parallelization). Finding high-performance programs requires the search strategy to cover a comprehensive space and explore it efficiently. We describe two recent and effective approaches in this section and other related work in section 3.8.

Template-guided search. In template-guided search, the search space is defined by manual templates. As shown in Figure 3.2a, the compiler (*e.g.*, TVM) requires the user to manually write a template for a computation definition. The template defines the structure of the tensor programs with some tunable parameters (*e.g.*, tile size and unrolling factor). The compiler then searches for the best values of these parameters for a specific input shape configuration and a specific hardware target. This approach has achieved good performance on common deep learning operators. However, developing templates requires substantial effort. For example, the code repository of TVM already contains more than 15K lines of code for these templates. This number continues to grow as new operators and new hardware platforms emerge. Besides, constructing a quality template requires expertise in both

tensor operators and hardware. It takes non-trivial research effort [113, 194, 229] to develop quality templates. Despite the complexity of template design, manual templates only cover limited program structures because manually enumerating all optimization choices for all operators is prohibitive. This approach typically requires defining one template for each operator. FlexTensor [229] proposes a general template to cover multiple operators, but its template is still designed for single operator granularity, which fails to include optimizations involving multiple operators (*e.g.*, operator fusion). The search space of optimizing a computational graph with multiple operators should contain different ways to compose the operators. A template-based approach fails to achieve this because it cannot break down their fixed templates and re-compose them during the search.

Sequential construction based search. This approach defines the search space by decomposing the program construction into a fixed sequence of decisions. The compiler then uses an algorithm such as beam search [118] to search for good decisions (*e.g.*, Halide auto-scheduler [2]). In this approach, the compiler constructs a tensor program by sequentially unfolding all nodes in the computational graph. For each node, the compiler makes a few decisions on how to transform it into low-level tensor programs (*i.e.*, deciding computation location, storage location, tile size, *etc.*). When all nodes are unfolded, a complete tensor program is constructed. This approach uses a set of general unfolding rules for every node, so it can search automatically without requiring manual templates. Because the number of possible choices of each decision is large, to make the sequential process feasible, this approach keeps only top- k candidate programs after every decision. The compiler estimates and compares the performance of candidate programs with a learned cost model to select the top- k candidates; while other candidates are pruned. During the search, the candidate programs are incomplete because only part of the computational graph is unfolded or only some of the decisions are made. Figure 3.2b shows this process.

However, estimating the final performance of incomplete programs is difficult in several respects: (1) the cost model trained on complete programs cannot accurately predict the final performance of incomplete programs. The cost model can only be trained on complete programs because we need to compile programs and measure their execution time to get the labels for training. Directly using this model to compare the final performance of incomplete programs will result in poor accuracy. As a case study, we train our cost model (section 3.5) on 20,000 random complete programs from our search space and use the model to predict the final performance of incomplete programs. The incomplete programs are obtained by only applying a fraction of loop transformations of the complete programs. We use two ranking metrics for evaluation: the accuracy of pairwise comparison and the recall@ k score of top- k programs ¹ ($k = 10$). As shown in Figure 3.3, the two curves start from 50% and 0% respectively, meaning that random guess with zero information gives 50% pairwise comparison accuracy and 0% top- k recall. The two curves increase quickly as the programs become complete, which means the cost model performs very well for complete programs but

¹recall@ k of top- $k = \frac{|G \cap P|}{k}$, where G is the set of top- k programs according to the ground truth and P is the set of top- k programs predicted by the model.

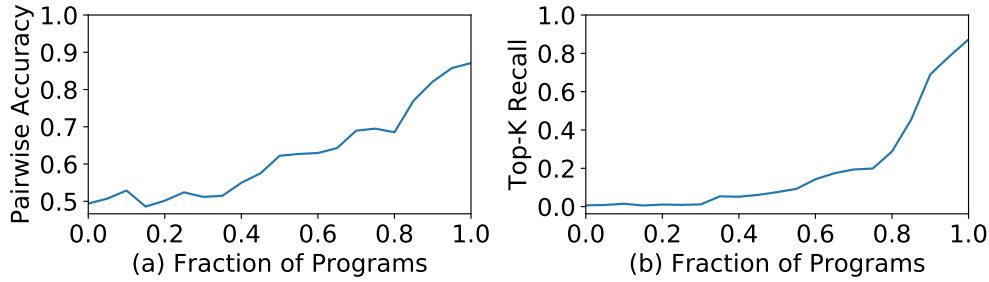


Figure 3.3: Pairwise comparison accuracy and top- k recall curve on random partial programs. In both subfigures, higher values are better.

fails to accurately predict the final performance of incomplete programs. (2) The fixed order of sequential decisions limits the design of the search space. For example, some optimization needs to add new nodes to the computational graph (*e.g.*, adding cache nodes, using `rfactor`[175]). The number of decisions for different programs becomes different. It is hard to align the incomplete programs for a fair comparison. (3) Sequential construction based search is not scalable. Enlarging the search space needs to add more sequential construction steps, which, however, leads to a worse accumulated error.

Ansor’s hierarchical approach As shown in Figure 3.2c, Ansor is backed by a hierarchical search space that decouples high-level structures and low-level details. Ansor constructs the search space for a computational graph automatically, eliminating the need to manually develop templates. Ansor then samples complete programs from the space and performs fine-tuning on complete programs, avoiding the inaccurate estimation of incomplete programs. Figure 3.2 shows the key difference between Ansor’s approach and existing approaches.

3.3 Design Overview

Ansor is an automated tensor program generation framework. Figure 3.4 shows the overall architecture of Ansor. The input of Ansor is a set of to be optimized DNNs. Ansor uses the operator fusion algorithm from Relay[156] to convert DNNs from popular model formats (*e.g.*, ONNX [13], TensorFlow PB) to partitioned small subgraphs. Ansor then generates tensor programs for these subgraphs. Ansor has three major components: (1) a program sampler that constructs a large search space and samples diverse programs from it; (2) a performance tuner that fine-tunes the performance of sampled programs; (3) a task scheduler that allocates time resources for optimizing multiple subgraphs in the DNNs.

Program sampler. One key challenge Ansor has to address is generating a large search space for a given computational graph. To cover diverse tensor programs with various high-level structures and low-level details, Ansor utilizes a hierarchical representation of

the search space with two levels: *sketch* and *annotation* (section 3.4). Ansor defines the high-level structures of programs as sketches and leaves billions of low-level choices (*e.g.*, tile size, parallel, unroll annotations) as annotations. This representation allows Ansor to enumerate high-level structures flexibly and sample low-level details efficiently. Ansor includes a program sampler that randomly samples programs from the space to provide comprehensive coverage of the search space.

Performance tuner. The performance of randomly sampled programs is not necessarily good. The next challenge is to fine-tune them. Ansor employs evolutionary search and a learned cost model to perform fine-tuning iteratively (section 3.5). At each iteration, Ansor uses re-sampled new programs as well as good programs from previous iterations as the initial population to start the evolutionary search. Evolutionary search fine-tunes programs by mutation and crossover which perform out-of-order rewrite and address the limitation of sequential construction. Querying the learned cost model is orders of magnitude faster than actual measurement, so we can evaluate thousands of programs in seconds.

Task scheduler. Using program sampling and performance fine-tuning allows Ansor to find high-performance tensor programs for a computational graph. Intuitively, treating a whole DNN as a single computational graph and generating a full tensor program for it could potentially achieve the optimal performance. This, however, is inefficient because it has to deal with the unnecessary exponential explosion of the search space. Typically, the compiler partitions the large computational graph of a DNN into several small subgraphs [156, 34]. This partition has a negligible effect on the performance thanks to the layer-by-layer construction nature of DNNs. This brings the final challenge of Ansor: how to allocate time resources when generating programs for multiple subgraphs. The task scheduler (section 3.6) in Ansor uses a scheduling algorithm based on gradient descent to allocate resources to the subgraphs that are more likely to improve the end-to-end DNN performance.

3.4 Program Sampling

The search space an algorithm explores determines the best programs it can find. The considered search spaces in existing approaches are limited by the following factors: (1) Manual enumeration (*e.g.*, TVM [31]). It is impractical to manually enumerate all possible choices by templates, so existing manual templates only cover a limited search space heuristically. (2) Aggressive early pruning (*e.g.*, Halide auto-scheduler [2]). Aggressive early pruning based on evaluating incomplete programs prevents the search algorithm from exploring certain regions in the space.

In this section, we introduce techniques to push the boundary of the considered search space by addressing the above limitations. To solve (1), we automatically expand the search space by recursively applying a set of flexible derivation rules. To avoid (2), we randomly sample complete programs in the search space. Since random sampling gives an equal chance to every point to be sampled, our search algorithm can potentially explore every program

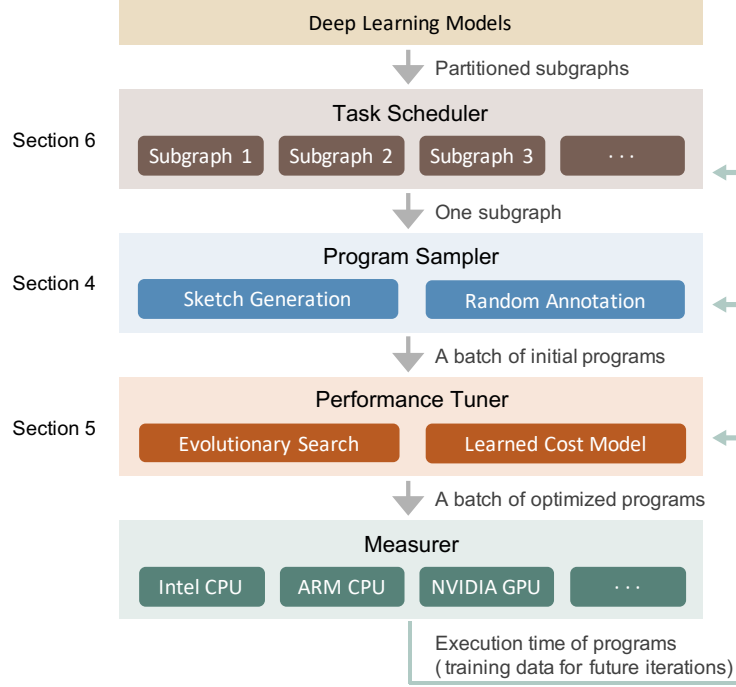


Figure 3.4: System Overview. The gray arrows show the flow of extracting subgraphs from deep learning models and generating optimized programs for them. The green arrows mean the measurer returns profiling data to update the status of all components in the system.

No	Rule Name	Condition	Application
1	Skip	$\neg IsStrictInlinable(S, i)$	$S' = S; i' = i - 1$
2	Always Inline	$IsStrictInlinable(S, i)$	$S' = Inline(S, i); i' = i - 1$
3	Multi-level Tiling	$HasDataReuse(S, i)$	$S' = MultiLevelTiling(S, i); i' = i - 1$
4	Multi-level Tiling with Fusion	$HasDataReuse(S, i) \wedge HasFusableConsumer(S, i)$	$S' = FuseConsumer(MultiLevelTiling(S, i), i); i' = i - 1$
5	Add Cache Stage	$HasDataReuse(S, i) \wedge \neg HasFusableConsumer(S, i)$	$S' = AddCacheWrite(S, i); i = i'$
6	Reduction Factorization	$HasMoreReductionParallel(S, i)$	$S' = AddRfactor(S, i); i' = i - 1$
...	User Defined Rule	...	...

Table 3.1: Derivation rules used to generate sketches. The condition runs on the current state $\sigma = (S, i)$. The application derives the next state $\sigma' = (S', i')$ from the current state σ . Note that some function (*e.g.*, *AddRfactor*, *FuseConsumer*) can return multiple possible values of S' . In this case we collect all possible S' , and return multiple next states σ' for a single input state σ .

in the considered space. We do not rely on random sampling to find the optimal program, because every sampled program is later fined-tuned (section 3.5).

To sample programs that can cover a large search space, we define a hierarchical search space with two levels: *sketch* and *annotation*. We define the high-level structures of programs as sketches and leave billions of low-level choices (*e.g.*, tile size, parallel, unroll annotations) as annotations. At the top level, we generate sketches by recursively applying a few derivation

rules. At the bottom level, we randomly annotate these sketches to get complete programs. This representation summarizes a few basic structures from billions of low-level choices, enabling the flexible enumeration of high-level structures and efficient sampling of low-level details.

While Ansor supports both CPU and GPU, we explain the sampling process for CPUs in section 3.4 and section 3.4 as an example. We then discuss how the process is different for GPU in section 3.4.

Sketch Generation

As shown in Figure 3.4, the program sampler accepts partitioned subgraphs as input. The first column in Figure 3.5 shows two examples of the input. The input has three equivalent forms: the mathematical expression, the corresponding naive program obtained by directly expanding the loop indices, and the corresponding computational graph (directed acyclic graph, or DAG).

To generate sketches for a DAG with multiple nodes, we visit all the nodes in a topological order and build the structure iteratively. For computation nodes that are compute-intensive and have a lot of data reuse opportunities (*e.g.*, conv2d, matmul), we build basic tile and fusion structures for them as the sketch. For simple element-wise nodes (*e.g.*, ReLU, element-wise add), we can safely inline them. Note that new nodes (*e.g.*, caching nodes, layout transform nodes) may also be introduced to the DAG during the sketch generation.

We propose a derivation-based enumeration approach to generate all possible sketches by recursively applying several basic rules. This process takes a DAG as an input and returns a list of sketches. We define the State $\sigma = (S, i)$, where S is the current partially generated sketch for the DAG, and i is the index of the current working node. The nodes in a DAG are sorted in a topological order from output to input. The derivation begins from the initial naive program and the last node, or the initial state $\sigma = (\text{naive program}, \text{index of the last node})$. Then we try to apply all derivation rules to the states recursively. For each rule, if the current state satisfies the application condition, we apply the rule to $\sigma = (S, i)$ and get $\sigma' = (S', i')$ where $i' \leq i$. This way the index i (working node) decreases monotonically. A state becomes a terminal state when $i = 0$. During enumeration, multiple rules can be applied to one state to generate multiple succeeding states. One rule can also generate multiple possible succeeding states. So we maintain a queue to store all intermediate states. The process ends when the queue is empty. All $\sigma.S$ in terminal states form a sketch list at the end of the sketch generation. The number of sketches is less than 10 for a typical subgraph.

Derivation rules. Table 3.1 lists derivation rules we used for the CPU. We first provide the definition of the used predicates and then describe the functionality of each rule. *IsStrictInlinable*(S, i) indicates if the node i in S is a simple element-wise operator that can always be inlined (*e.g.*, element-wise add, ReLU). *HasDataReuse*(S, i) indicates if the node i in S is a compute-intensive operator and has plentiful intra-operator data reuse opportunity (*e.g.*, matmul, conv2d). *HasFusableConsumer*(S, i) indicates if the node i in S has

only one consumer j and node j can be fused into node i (e.g., `matmul + bias_add`, `conv2d + relu`). *HasMoreReductionParallel*(S, i) indicates if the node i in S has little parallelism in space dimensions but has ample parallelism opportunity in reduction dimensions. (e.g., computing 2-norm of a matrix, `matmul` $C_{2 \times 2} = A_{2 \times 512} \cdot B_{512 \times 2}$). We perform static analysis on the computation definitions to get the values for these predicates. The analysis is done automatically by parsing the read/write pattern in the mathematical expressions. Next, we introduce the functionality of each derivation rule.

Rule 1 just simply skips a node if it is not strictly inlinable. Rule 2 always inlines strictly inlinable nodes. Since the conditions of rule 1 and rule 2 are mutually exclusive, a state with $i > 1$ can always satisfy one of them and continue to derive.

Rules 3, 4, and 5 deal with the multi-level tiling and fusion for nodes that have data reuse. Rule 3 performs multi-level tiling for data reusable nodes. For CPU, we use a “SSRSRS” tile structure, where “S” stands for one tile level of space loops and “R” stands for one tile level of reduction loops. For example, in the `matmul` $C(i, j) = \sum_k A[i, k] \times B[k, j]$, i and j are space loops and k is a reduction loop. The “SSRSRS” tile structure for `matmul` expands the original 3-level loop (i, j, k) into a 10-level loop $(i_0, j_0, i_1, j_1, k_0, i_2, j_2, k_1, i_3, j_3)$. Although we do not permute the loop order, this multi-level tiling can also cover some cases of reordering. For example, the above 10-level loop can be specialized to just a simple reorder (k_0, j_2, i_3) by setting the length of other loops to one. The “SSRSRS” tile structure is general for compute-intensive dense operators (e.g., `matmul`, `conv2d`, `conv3d`) in deep learning, because they all consist of space loops and reduction loops.

Rule 4 performs multi-level tiling and also fuses the fusible consumers. For example, we fuse the element-wise nodes (e.g., `ReLU`, `bias add`) into the tiled nodes (e.g., `conv2d`, `matmul`). Rule 5 adds a caching node if the current data-reusable node does not have a fusible consumer. For example, the final output node in a DAG does not have any consumer, so it directly writes results into main memory by default and this is inefficient due to the high latency of memory accesses. By adding a cache node, we introduce a new fusible consumer into the DAG, then rule 4 can be applied to fuse this newly added cache node into the final output node. With the cache node fused, now the final output node writes its results into a cache block, and the cache block will be written to the main memory at once when all data in the block is computed.

Rule 6 can use `rfactor` [175] to factorize a reduction loop into a space loop to bring more parallelism.

Examples. Figure 3.5 shows three examples of the generated sketches. The sketches are different from the manual templates in TVM, because the manual templates specify both high-level structures and low-level details while sketches only define high-level structures. For the example input 1, the sorted order of the four nodes in the DAG is (A, B, C, D) . To derive the sketches for the DAG, we start from output node $D(i = 4)$ and apply rules to the



Figure 3.5: Examples of generated sketches and sampled programs. This figure shows two example inputs, three generated sketches and four sampled programs. The code example is pseudo code in a python-like syntax.

nodes one by one. Specifically, the derivation for generated sketch 1 is:

$$\begin{aligned}
 \text{Input 1} &\rightarrow \sigma(S_0, i = 4) \xrightarrow{\text{Rule 1}} \sigma(S_1, i = 3) \xrightarrow{\text{Rule 4}} \\
 &\sigma(S_2, i = 2) \xrightarrow{\text{Rule 1}} \sigma(S_3, i = 1) \xrightarrow{\text{Rule 1}} \text{Sketch 1}
 \end{aligned}$$

For the example input 2, the sorted order of the five nodes is (A, B, C, D, E) . Similarly, we start from the output node $E(i = 5)$ and apply rules recursively. The generated sketch 2 is derived by:

$$\begin{aligned}
 \text{Input 2} &\rightarrow \sigma(S_0, i = 5) \xrightarrow{\text{Rule 5}} \sigma(S_1, i = 5) \xrightarrow{\text{Rule 4}} \\
 &\sigma(S_2, i = 4) \xrightarrow{\text{Rule 1}} \sigma(S_3, i = 3) \xrightarrow{\text{Rule 1}} \\
 &\sigma(S_4, i = 2) \xrightarrow{\text{Rule 2}} \sigma(S_5, i = 1) \xrightarrow{\text{Rule 1}} \text{Sketch 2}
 \end{aligned}$$

Similarly, the generated sketch 3 is derived by:

$$\begin{aligned} \text{Input 2} &\rightarrow \sigma(S_0, i = 5) \xrightarrow{\text{Rule 6}} \sigma(S_1, i = 4) \xrightarrow{\text{Rule 1}} \\ &\sigma(S_2, i = 3) \xrightarrow{\text{Rule 1}} \sigma(S_3, i = 2) \xrightarrow{\text{Rule 2}} \\ &\sigma(S_4, i = 1) \xrightarrow{\text{Rule 1}} \text{Sketch 3} \end{aligned}$$

Customization. While the presented rules are practical enough to cover the structures for most operators, there are always exceptions. For example, some special algorithms (*e.g.*, Winograd convolution [97]) and accelerator intrinsics (*e.g.*, TensorCore [134]) require special tile structures to be effective. Although the template-guided search approach (in TVM) can craft a new template for every new case, it needs a great amount of design effort. On the other hand, the derivation-based sketch generation in Ansor is flexible enough to generate the required structures for emerging algorithms and hardware, as we allow users to register new derivation rules and integrate them seamlessly with existing rules.

Random Annotation

The sketches generated by the previous subsection are incomplete programs because they only have tile structures without specific tile sizes and loop annotations, such as parallel, unroll, and vectorization. In this subsection, we annotate sketches to make them complete programs for fine-tuning and evaluation.

Given a list of generated sketches, we randomly pick one sketch, randomly fill out tile sizes, parallelize some outer loops, vectorize some inner loops, and unroll a few inner loops. We also randomly change the computation location of some nodes in the program to make a slight tweak to the tile structure. All “random” in this subsection means a uniform distribution over all valid values. If some special algorithms require custom annotations to be effective (*e.g.*, special unrolling), we allow users to give simple hints in the computation definition to adjust the annotation policy. Finally, since changing the layout of constant tensors can be done in compilation time and brings no runtime overhead, we rewrite the layouts of the constant tensors according to the multi-level tile structure to make them as cache-friendly as possible. This optimization is effective because the weight tensors of convolution or dense layers are constants for inference applications.

Examples of random sampling are shown in Figure 3.5. The sampled program might have fewer loops than the sketch because the loops with length one are simplified.

GPU Support

For GPU, we change the multi-level tiling structure from “SSRSRS” to “SSSRRSRS” to match the architecture of GPU. The loops in the first three space tiles are bound to BlockIdx, virtual thread (for reducing bank conflicts), and ThreadIdx, respectively. We add two sketch

derivation rules, one for utilizing shared memory by inserting a caching node (similar to Rule 5) and the other for cross-thread reduction (similar to Rule 6).

3.5 Performance Fine-tuning

The programs sampled by the program sampler have good coverage of the search space, but their qualities are not guaranteed. This is because the optimization choices, such as tile structure and loop annotations, are all randomly sampled. In this section, we introduce the performance tuner that fine-tunes the performance of the sampled programs via evolutionary search and a learned cost model.

The fine-tuning is performed iteratively. At each iteration, we first use evolutionary search to find a small batch of promising programs according to a learned cost model. We then measure these programs on hardware to get the actual execution time cost. Finally, the profiling data got from measurement is used to re-train the cost model to make it more accurate.

The evolutionary search uses randomly sampled programs as well as high-quality programs from the previous measurement as the initial population and applies mutation and crossover to generate the next generation. The learned cost model is used to predict the *fitness* of each program, which is the throughput of one program in our case. We run evolution for a fixed number of generations and pick the best programs found during the search. We leverage a learned cost model because the cost model can give relatively accurate estimations of the fitness of programs while being orders of magnitudes faster than the actual measurement. It allows us to compare tens of thousands of programs in the search space in seconds, and pick the promising ones to do actual measurements.

Evolutionary Search

Evolutionary search [190] is a generic meta-heuristic algorithm inspired by biological evolution. By iteratively mutating high-quality programs, we can generate new programs with potentially higher quality. The evolution starts from the sampled initial generation. To generate the next generation, we first select some programs from the current generation according to certain probabilities. The probability of selecting a program is proportional to its fitness predicted by the learned cost model (section 3.5), meaning that the program with a higher performance score has a higher probability to be selected. For the selected programs, we randomly apply one of the evolution operations to generate a new program. Basically, for decisions we made during sampling (section 3.4), we design corresponding evolution operations to rewrite and fine-tune them.

Tile size mutation. This operation scans the program and randomly selects a tiled loop. For this tiled loop, it divides a tile size of one tile level by a random factor and multiplies this factor to another level. Since this operation keeps the product of tile sizes equal to the original loop length, the mutated program is always valid.

Parallel mutation. This operation scans the program and randomly selects a loop that has been annotated with parallel. For this loop, this operation changes the parallel granularity by either fusing its adjacent loop levels or splitting it by a factor.

Pragma mutation. Some optimizations in a program are specified by compiler-specific pragma. This operation scans the program and randomly selects a pragma. For this pragma, this operation randomly mutates it into another valid value. For example, our underlying code generator supports auto unrolling with a maximum number of steps by providing an `auto_unroll_max_step=N` pragma. We randomly tweak the number N .

Computation location mutation. This operation scans the program and randomly selects a flexible node that is not multi-level tiled (*e.g.*, a padding node in the convolution layer). For this node, the operation randomly changes its computation location to another valid attach point.

Node-based crossover. Crossover is an operation to generate new offspring by combining the genes from two or more parents. The genes of a program in Ansor are its rewriting steps. Every program generated by Ansor is rewritten from its initial naive implementation. Ansor preserves a complete rewriting history for each program during sketch generation and random annotation. We can treat rewriting steps as the genes of a program because they describe how this program is formed from the initial naive one. Based on this, we can generate a new program by combining the rewriting steps of two existing programs. However, arbitrarily combining rewriting steps from two programs might break the dependencies in steps and create an invalid program. As a result, the granularity of crossover operation in Ansor is based on nodes in the DAG, because the rewriting steps across different nodes usually have less dependency. Ansor randomly selects one parent for each node and merges the rewriting steps of selected nodes. When there are dependencies between nodes, Ansor tries to analyze and adjust the steps with simple heuristics. Ansor further verifies the merged programs to guarantee the functional correctness. The verification is simple because Ansor only uses a small set of loop transformation rewriting steps, and the underlying code generator can check the correctness by dependency analysis.

The evolutionary search leverages mutation and crossover to generate a new set of candidates repeatedly for several rounds and outputs a small set of programs with the highest scores. These programs will be compiled and measured on the target hardware to obtain the real running time cost. The collected measurement data is then used to update the cost model. In this way, the accuracy of the learned cost model is gradually improved to match the target hardware. Consequently, the evolutionary search gradually generates higher-quality programs for the target hardware platform.

Unlike the search algorithms in TVM and FlexTensor that can only work in a fixed grid-like parameter space, the evolutionary operations in Ansor are specifically designed for tensor programs. They can be applied to general tensor programs and can handle a search space with complicated dependency. Unlike the unfolding rules in Halide auto-scheduler, these operations can perform out-of-order modifications to programs, addressing the sequential limitations.

Learned Cost Model

A cost model is necessary for estimating the performance of programs quickly during the search. We adopt a learned cost model similar to related works [2, 31] with newly designed program features. A system based on learned cost models has great portability because a single model design can be reused for different hardware backends by feeding in different training data.

Since our target programs are mainly data parallel tensor programs, which are made by multiple interleaved loop nests with several assignment statements as the innermost statements, we train the cost model to predict the score of one innermost non-loop statement in a loop nest. For a full program, we make predictions for each innermost non-loop statement and add the predictions up as the score. We build the feature vector for an innermost non-loop statement by extracting features in the context of a full program. The extracted features include arithmetic features and memory access features. A detailed list of extracted features is in the appendix².

We use weighted squared error as the loss function. Because we mostly care about identifying the well-performing programs from the search space, we put more weight on the programs that run faster. Specifically, the loss function of the model f on a program P with throughput y is $loss(f, P, y) = w_p(\sum_{s \in S(P)} f(s) - y)^2 = y(\sum_{s \in S(P)} f(s) - y)^2$ where $S(P)$ is the set of innermost non-loop statements in P . We directly use the throughput y as weight. We train a gradient boosting decision tree [30] as the underlying model f . A single model is trained for all tensor programs coming from all DAGs, and we normalize the throughput of all programs coming from the same DAG to be in the range of $[0, 1]$. When optimizing a DNN, the number of measured programs are typically less than 30,000. Training a gradient boosting decision tree is very fast on such a small data sets, so we train a new model every time instead of doing incremental updates.

3.6 Task Scheduler

A DNN can be partitioned into many independent subgraphs (*e.g.*, conv2d + relu). For some subgraphs, spending time in tuning them does not improve the end-to-end DNN performance significantly. This is due to two reasons: either (1) the subgraph is not a performance bottleneck, or (2) tuning brings only minimal improvement in the subgraph’s performance.

To avoid wasting time on tuning unimportant subgraphs, Ansor dynamically allocates different amounts of time resources to different subgraphs. Take ResNet-50 for example, it has 29 unique subgraphs after the graph partitioning. Most of these subgraphs are convolution layers with different shapes configurations (input size, kernel size, stride, etc). We need to generate different programs for different convolution layers because the best tensor program depends on these shape configurations. In reality, users may have multiple DNNs

²Please refer to the original paper [226]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

for all their applications. This leads to more subgraphs as well as more opportunities to reduce the total tuning time, because we can share and reuse knowledge between subgraphs. A subgraph can also appear multiple times in a DNN or across different DNNs.

We define a task as a process performed to generate high-performance programs for a subgraph. It means that optimizing a single DNN requires finishing dozens of tasks (*e.g.*, 29 tasks for ResNet-50). Ansor’s task scheduler allocates time resources to tasks in an iterative manner. At each iteration, Ansor selects a task, generates a batch of promising programs for the subgraph, and measures the program on hardware. We define such an iteration as one unit of time resources. When we allocate one unit of time resources to a task, the task obtains an opportunity to generate and measure new programs, which also means the chance to find better programs. We next present the formulation of the scheduling problem and our solution.

Problem Formulation

When tuning a DNN or a set of DNNs, a user can have various types of goals, for example, reducing a DNN’s latency, meeting latency requirements for a set of DNNs, or minimizing tuning time when tuning no longer improves DNN performance significantly. We thus provide users a set of objective functions to express their goals. Users can also provide their own objective functions.

Suppose there are n tasks in total. Let $t \in \mathbf{Z}^n$ be the allocation vector, where t_i is the number of time units spent on task i . Let the minimum subgraph latency task i achieves be a function of the allocation vector $g_i(t)$. Let the end-to-end cost of the DNNs be a function of the latency of the subgraphs $f(g_1(t), g_2(t), \dots, g_n(t))$. Our objective is to minimize the end-to-end cost:

$$\text{minimize } f(g_1(t), g_2(t), \dots, g_n(t))$$

To minimize the end-to-end latency of a single DNN, we can define $f(g_1, g_2, \dots, g_n) = \sum_{i=1}^n w_i \times g_i$, where w_i is the number of appearances of task i in the DNN. This formulation is straightforward because f is an approximation of the end-to-end DNN latency.

When tuning a set of DNNs, there are several options. Table 3.2 shows a number of example objective functions for tuning multiple DNNs. Let m be the number of DNNs, $S(j)$ is the set of tasks that belong to DNN j . f_1 adds up the latency of every DNN, which means to optimize the cost of a pipeline that sequentially runs all DNNs once. In f_2 , we define L_j as the latency requirement of DNN j , meaning that we do not want to spend time on a DNN if its latency has already met the requirement. In f_3 , we define B_j as the reference latency of a DNN j . As a result, our goal is to maximize the geometric mean of speedup against the given reference latency. Finally in f_4 , we define a function $ES(g_i, t)$ that returns an early stopping value by looking at the history of latency of task i . It can achieve the effect of per-task early stopping.

$$\begin{aligned}
 f_1 &= \sum_{j=1}^m \sum_{i \in S(j)} w_i \times g_i(t) \\
 f_2 &= \sum_{j=1}^m \max(\sum_{i \in S(j)} w_i \times g_i(t), L_j) \\
 f_3 &= -(\prod_{j=1}^m \frac{B_j}{\sum_{i \in S(j)} w_i \times g_i(t)})^{\frac{1}{m}} \\
 f_4 &= \sum_{j=1}^m \sum_{i \in S(j)} w_i \times \max(g_i(t), ES(g_i, t))
 \end{aligned}$$

Table 3.2: Examples of objective functions for multiple neural networks

Optimizing with Gradient Descent

We propose a scheduling algorithm based on gradient descent to efficiently optimize the objective function. Given the current allocation t , the idea is to approximate the gradient of the objective function $\frac{\partial f}{\partial t_i}$ in order to choose the task i such that $i = \operatorname{argmax}_i |\frac{\partial f}{\partial t_i}|$. We approximate the gradient by making an optimistic guess and considering the similarity between tasks.

The derivation is in the appendix³. We approximate the gradient by

$$\begin{aligned}
 \frac{\partial f}{\partial t_i} &\approx \frac{\partial f}{\partial g_i} \left(\alpha \frac{g_i(t_i) - g_i(t_i - \Delta t)}{\Delta t} + \right. \\
 &\quad \left. (1 - \alpha) \left(\min\left(-\frac{g_i(t_i)}{t_i}, \beta \frac{C_i}{\max_{k \in N(i)} V_k} - g_i(t_i)\right) \right) \right)
 \end{aligned}$$

where Δt is a small backward window size, $g_i(t_i)$ and $g_i(t_i - \Delta t)$ are known from the history of allocations. $N(i)$ is the set of similar tasks of i , C_i is the number of floating point operation in task i and V_k is the number of floating point operation per second we can achieve in task k . The parameter α and β control the weight to trust some predictions.

To run the algorithm, Ansor starts from $t = \mathbf{0}$ and warms up with a round of round-robin to get an initial allocation vector $t = (1, 1, \dots, 1)$. After the warm-up, at each iteration, we compute the gradient of each task and pick $\operatorname{argmax}_i |\frac{\partial f}{\partial t_i}|$. Then we allocate the resource unit to task i and update the allocation vector $t_i = t_i + 1$. The optimization process continues until we run out of the time budget. To encourage exploration, we adopt a ϵ -greedy strategy [176], which preserves a probability of ϵ to randomly select a task.

Taking the case of optimizing for a single DNN's end-to-end latency for example, Ansor prioritizes a subgraph that has a high initial latency because our optimistic guess says we can reduce its latency quickly. Later, if Ansor spends many iterations on it without observing a decrease in its latency, Ansor leaves the subgraph because its $|\frac{\partial f}{\partial t_i}|$ decreases.

³Please refer to the original paper [226]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

3.7 Evaluation

The core of Ansor is implemented in C++ with about 12K lines of code (3K for the search policy and 9K for other infrastructure). Ansor generates programs in its own intermediate representation (IR). These programs are then lowered to TVM IR for code generation targeting various hardware platforms. Ansor only utilizes TVM as a deterministic code generator.

We evaluate the performance of generated programs on three levels: single operator, sub-graph, and entire neural network. For each level of evaluation, we compare Ansor against the state-of-the-art search frameworks and hardware-specific manual libraries. We also evaluate the search efficiency and the effectiveness of each component in Ansor.

The generated tensor programs are benchmarked on three hardware platforms: an Intel CPU (18-core Platinum 8124M@3.0 GHz), an NVIDIA GPU (V100), and an ARM CPU (4-core Cortex-A53@1.4GHz on the Raspberry Pi 3b+). We use float32 as the data type for all evaluations.

Single Operator Benchmark

Workloads. We first evaluate Ansor on a set of common deep learning operators, including 1D, 2D, and 3D convolution (C1D, C2D, and C3D respectively), matrix multiplication (GMM), group convolution (GRP), dilated convolution (DIL) [217], depth-wise convolution (DEP) [74], transposed 2D convolution (T2D) [148], capsule 2D convolution (CAP) [73], and matrix 2-norm (NRM). For each operator, we select 4 common shape configurations and evaluate them with two batch sizes (1 and 16). In total, there are 10 operators $\times$ 4 shape configurations $\times$ 2 batch size = 80 test cases. The shape configurations used can be found in the appendix⁴. We run these test cases on the Intel CPU.

Baselines. We include PyTorch (v1.5)[143], Halide auto-scheduler (commit: 1f875b0)[2], FlexTensor (commit: 7ac302c)[229], and AutoTVM (commit: 69313a7)[31] as baselines. PyTorch is backed by the vendor-provided kernel library MKL-DNN [81]. Halide auto-scheduler is a sequential construction based search framework for Halide. AutoTVM and FlexTensor are template-guided search frameworks based on TVM. Since Halide auto-scheduler does not have a pre-trained cost model for AVX-512, we disabled AVX-512 for the evaluation in section 3.7 and section 3.7. For every operator, we use the best layout available in each framework, but the input and output tensors must not be packed.

Search settings. We let search frameworks (*i.e.*, Halide auto-scheduler, FlexTensor, AutoTVM, and Ansor) to run search or auto-tuning with up to 1,000 measurement trials per test case. This means each framework can measure at most $80 \times 1,000$ programs for auto-tuning in this evaluation. Using the same number of measurement trials makes it a fair comparison without involving implementation details. In addition, using 1,000 measurement trials per test case is typically enough for the search to converge in these frameworks.

⁴Please refer to the original paper [226]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

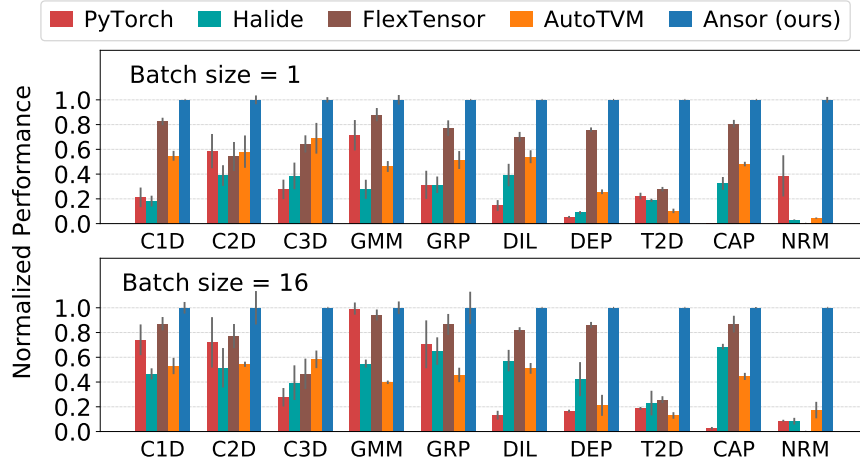


Figure 3.6: Single operator performance benchmark on a 20-core Intel-Platinum-8269CY. The y-axis is the throughput normalized to the best throughput for each operator.

Normalization. Figure 3.6 shows the normalized performance. For each test case, we normalize the throughputs to the best performing framework. We then plot the geometric mean of the four shapes of each operator. The geometric mean is also normalized to the best performing framework, so the best framework has a normalized performance of 1 in the figure. The error bar denotes the standard deviation of the normalized throughput of four shapes of each operator.

Results. As shown in the Figure 3.6, Ansor performs the best or equally the best in all operator and batch size settings. Ansor outperforms existing search frameworks by $1.1 - 22.5\times$. The performance improvements of Ansor come from both its large search space and effective exploration strategy. For most operators, we found the best program generated by Ansor is outside the search space of existing search frameworks because Ansor is able to explore more optimization combinations. For example, the significant speedup on NRM is because Ansor can parallelize reduction loops, while other frameworks do not. The large speedup on T2D is because Ansor can use correct tile structures and unrolling strategies to let the code generator simplify the multiplication of zeros in strided transposed convolution. In contrast, other frameworks fail to capture many effective optimizations in their search space, making them unable to find the programs that Ansor does. For example, the unfolding rules in Halide do not split the reduction loop in GMM and do not split reduction loops in C2D when padding is computed outside of reduction loops. The templates in AutoTVM have limited tile structures, as they cannot cover the structure of “Generated Sketch 1” in Figure 3.5. The template in FlexTensor does not change the computation location of padding. The template in FlexTensor fails to run for reduction operators like NRM.

Ablation study. We run four variants of Ansor on a convolution operator and report the performance curve. We pick the last convolution operator in ResNet-50 with batch size=16 as the test case, because its search space is sufficiently large to evaluate the search algorithms. Other operators share a similar pattern. In Figure 3.7, each curve is the median of

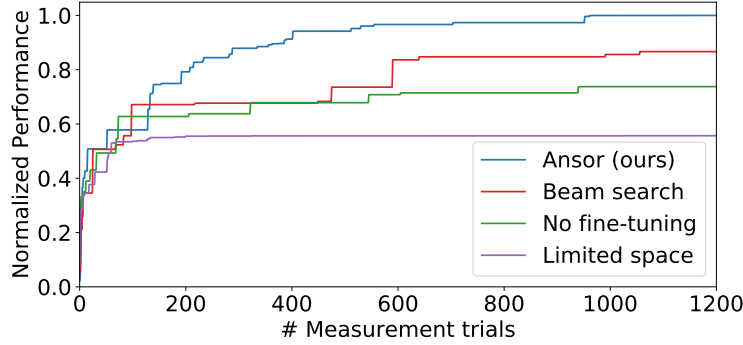


Figure 3.7: Ablation study of four variants of Ansor on a convolution operator. The y-axis is the throughput relative to the throughput of the best program.

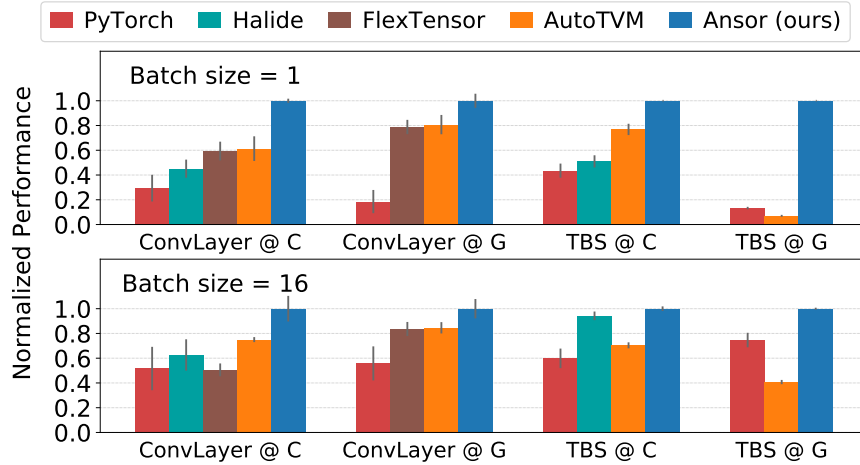


Figure 3.8: Subgraph performance benchmark on a 20-core Intel-Platinum-8269CY and an NVIDIA V100. “@C” denotes CPU results and “@G” denotes GPU results. The y-axis is the throughput normalized to the best throughput for each subgraph.

5 runs. “Ansor (ours)” uses all our introduced techniques. “Beam Search” means we prune incomplete programs with the cost model during the sampling process and do not use fine-tuning. “No fine-tuning” is based on “Ansor (ours)” but disables fine-tuning and only relies on random sampling. “Limited space” is also based on “Ansor (ours)” but limits the search space to make it similar to the space in existing manual templates (*e.g.*, limit tiling level, innermost tile sizes, and computation location). As demonstrated by Figure 3.7, dropping either the large search space or efficient fine-tuning decreases the final performance significantly. The aggressive early pruning in “Beam search” throws away incomplete programs with good final performance due to inaccurate estimation.

Subgraph Benchmark

We perform the subgraph benchmark on two common subgraphs in DNNs. The “ConvLayer” is a subgraph consisting of 2D convolution, batch normalization [82], and ReLU activation,

which is a common pattern in convolutional neural networks. The “TBS” is a subgraph consisting of two matrix transposes, one batch matrix multiplication, and a softmax, which is a pattern in the multi-head attention [187] in language models. Similar to the single operator benchmark (section 3.7), we select four different shape configurations and two batch sizes, run auto-tuning with up to 1,000 measurement trails per test case, and report the normalized performance. We use the same set of baseline frameworks and run the benchmark on the Intel CPU and the NVIDIA GPU. We do not report the performance of Halide auto-scheduler on GPU because as of writing the paper its GPU support is still in an experimental stage. FlexTensor fails to run on complicated subgraphs like “TBS”.

Figure 3.8 shows that Ansor outperforms manual libraries and other search frameworks by $1.1 - 14.2\times$. Ansor can generate high-performance programs consistently for these subgraphs on both platforms. FlexTensor performs well for single operators but shows less advantage for subgraphs because it lacks the support of operator fusion.

End-to-End Network Benchmark

Workloads. We benchmark the end-to-end inference execution time of several DNNs, which include ResNet-50 [71] and MobileNet-V2 [160] for image classification, 3D-ResNet-18 [70] for action recognition, DCGAN [148] generator for image generation, and BERT [47] for language understanding. We benchmark these DNNs on three hardware platforms. For the server-class Intel CPU and NVIDIA GPU, we report the results for batch size 1 and batch size 16. For the ARM CPU in the edge device, real-time feedback is typically desired, so we only report the results for batch size 1.

Baselines and Settings. We include PyTorch (v1.5 with torch script), TensorFlow (v2.0 with graph mode), TensorRT (v6.0 with TensorFlow integration) [135], TensorFlow Lite (V2.0), and AutoTVM as baseline frameworks. We do not include Halide auto-scheduler or FlexTensor because they lack the support of widely-used deep learning model formats (*e.g.*, ONNX, TensorFlow PB) and high-level graph optimizations. As a result, we expect that the end-to-end execution time they can achieve will be the sum of the latency of all subgraphs in a DNN. In contrast, AutoTVM can optimize a whole DNN with its manual templates and various graph-level optimizations (*e.g.*, graph-level layout search [113], graph-level constant folding [156]) which improve the performance significantly. Ansor also performs layout rewrite as described in section 3.4. We let both AutoTVM and Ansor run auto-tuning until they use to $1000 \times n$ measurement trials on each DNN, where n is the number of subgraphs in the DNN. This is typically enough for them to converge. We set the objective of the task scheduler as minimizing the total latency of one DNN and generate programs for these networks one by one. On the other hand, PyTorch, TensorFlow, TensorRT, and TensorFlow Lite are all backed by static kernel libraries (MKL-DNN on Intel CPU, CuDNN on NVIDIA GPU, and Eigen on ARM CPU) and do not need auto-tuning. We enable AVX-512 for all frameworks on the Intel CPU in this network benchmark.

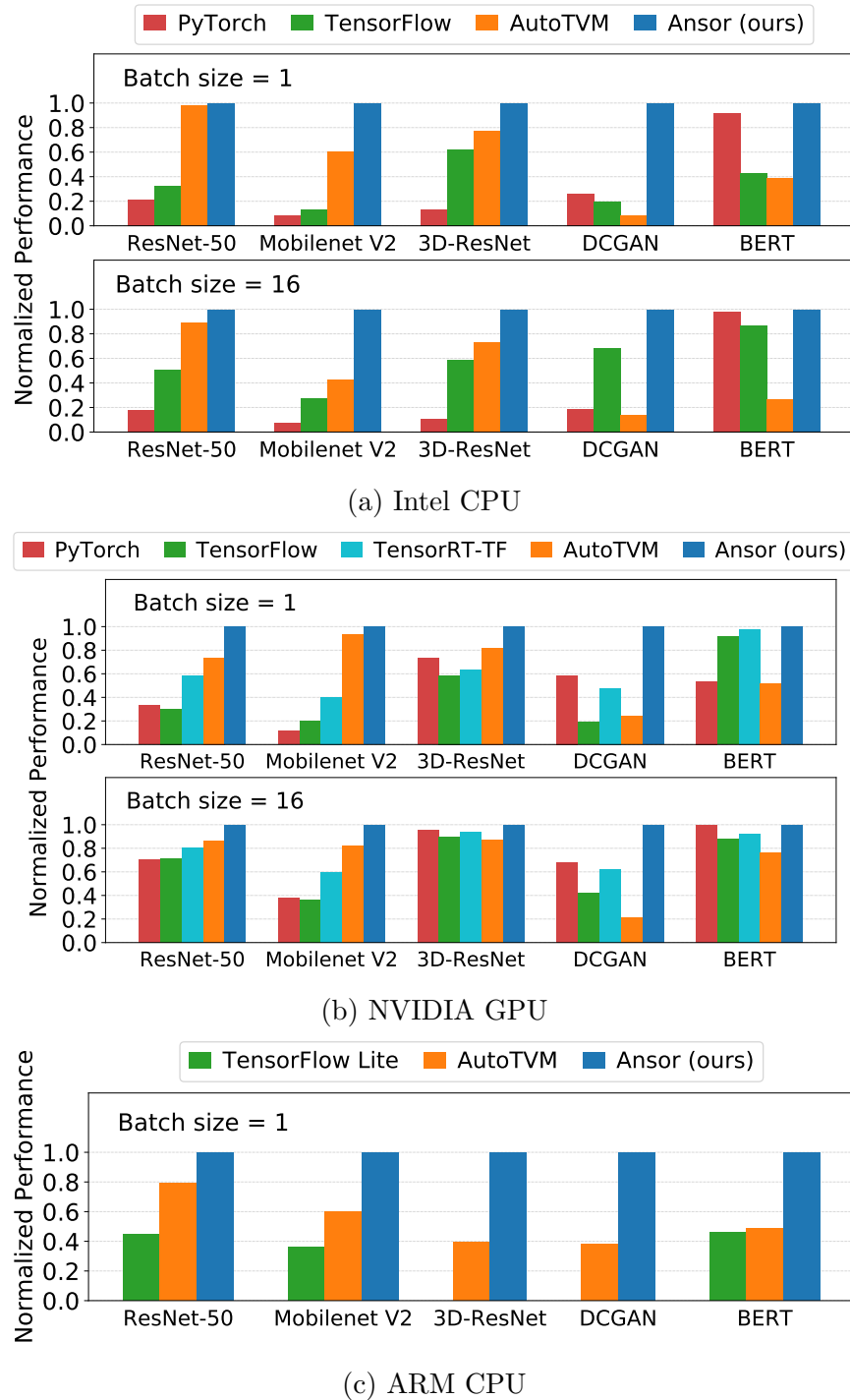


Figure 3.9: Network inference performance benchmark on three hardware platforms. The y-axis is the throughput relative to the best throughput for each network.

Results. Figure 3.9 shows the results on the Intel CPU, NVIDIA GPU and ARM CPU ⁵. Overall, Ansor performs the best or equally the best in all cases. Compared with search-based AutoTVM, Ansor matches or outperforms it in all cases with $1.0\text{--}21.8\times$ speedup. Compared with the best alternative, Ansor improves the execution performance of DNNs on the Intel CPU, ARM CPU, and NVIDIA GPU by up to $3.8\times$, $2.6\times$, and $1.7\times$, respectively. The reason for the significant speedup on DCGAN is that DCGAN mainly consists of transposed 2D convolution (T2D), which can be well optimized by Ansor, as shown and explained in the single operator benchmark (section 3.7). AutoTVM performs very well for ResNet-50 on the Intel CPU thanks to its highly-optimized templates for 2D convolution and global layout search [113]. Ansor does not run a global layout search but does rewrite the layout of weight tensors as described in section 3.4. Ansor uses more levels of tiling so it packs weight tensors into more levels. The layout rewrite brings about 40% improvement to ResNet-50 in Ansor. Compared with vendor-specific static libraries, Ansor has more advantages on uncommon shapes and small batch sizes, because it is not easy to manually optimize for these cases.

Ablation study. We run variants of Ansor on two test cases in Figure 3.10. In the left figure, we run four variants of Ansor to generate programs for a single mobilenet-V2. In the right figure, we run these variants for both mobilenet-V2 and ResNet-50. We set the objective function of the task scheduler to be the geometric mean of speedup against AutoTVM. As shown in Figure 3.10, “No task scheduler” means we use a round-robin strategy to allocate equal time resources to all subgraphs. “Limited space” is based on “Ansor (ours)” but limits the search space. “No fine-tuning” is also based on “Ansor (ours)” but disables fine-tuning and relies on random sampling only. As can be seen in Figure 3.10, “Limited space” performs the worst in terms of the final achieved performance, proving that the best programs are not included in the limited space. The final achieved performance can be improved by enlarging the search space, as depicted in “No fine-tuning”. However, in the right figure, randomly assigning tile sizes and annotations still cannot beat AutoTVM in the given time budget. After enabling fine-tuning, “No task scheduler” outperforms AutoTVM in both cases. Finally, “Ansor (ours)” employs the task scheduler to prioritize performance bottlenecks (*e.g.*, subgraphs contain 3×3 convolution), so it performs the best in both search efficiency and the final achieved performance.

Search Time

Ansor searches efficiently and can outperform or match AutoTVM with less search time. Ansor slices the time and utilizes the task scheduler to simultaneously optimize all subgraphs together. In contrast, AutoTVM and other systems do not have a task scheduler, so they generate programs for all subgraphs one by one with a predefined budget of measurement trials for each subgraph. Ansor saves the search time by prioritizing important subgraphs, while AutoTVM spends predefined time budget on every subgraph, which may be a waste on the unimportant subgraphs.

⁵3D-ResNet and DCGAN are not yet supported by TensorFlow Lite on the ARM CPU.

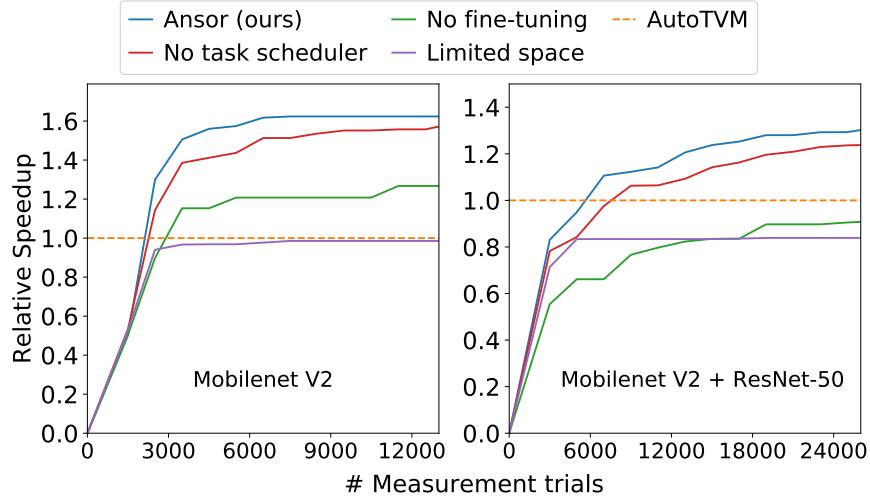


Figure 3.10: Network performance auto-tuning curve. The y-axis is the speedup relative to AutoTVM.

Table 3.3 shows the search time required for Ansor to match the performance of AutoTVM on the Intel CPU network benchmark (section 3.7). We list the search time in two metrics: number of measurements and wall-clock time. “Number of measurements” is a metric agnostic to the implementation of measurement and the overhead of search algorithm, while “Wall-clock time” takes these factors into account. As shown in the table, Ansor can match the performance of AutoTVM with an order of magnitude less search time. In Table 3.3a the saving in search time comes from the task scheduler, efficient fine-tuning, and comprehensive coverage of effective optimizations. In Table 3.3b, Ansor shows more time-saving in wall-clock time. This is because Ansor does not introduce much search overhead and has a better implementation of the measurement (on the Intel CPU, Ansor can get accurate measurement results with fewer repetitions by explicitly flushing the cache for some tensors). On other backends, Ansor can match the performance of AutoTVM with a similar saving in search time.

Typically, it takes several hours for Ansor to generate fully-optimized programs for a DNN on a single machine. This is acceptable for inference applications because it is a one-shot effort before deployment. In addition, the whole architecture of Ansor can be parallelized very easily.

Cost Model Evaluation

In this subsection, we evaluate the prediction quality of the learned cost model. We use 25,000 programs measured during tuning ResNet-50 on the Intel CPU as the data set. We randomly pick 20,000 programs as the training set and use the remaining 5,000 programs as the test set. We train the cost model and let it make predictions for the test set.

	AutoTVM	Ansor	Time-saving
ResNet-50	21,220	6,403	$3.3 \times$
Mobilenet-V2	31,272	1,892	$16.5 \times$
3D-ResNet	5,158	1,927	$2.7 \times$
DCGAN	3,003	298	$10.1 \times$
BERT	6,220	496	$12.5 \times$

(a) The number of measurements.

	AutoTVM	Ansor	Time-saving
ResNet-50	39,250	4,540	$8.6 \times$
Mobilenet-V2	58,468	660	$88.6 \times$
3D-ResNet	7,594	2,296	$3.3 \times$
DCGAN	4,914	420	$11.7 \times$
BERT	12,007	266	$45.1 \times$

(b) Wall-clock time (seconds)

Table 3.3: The number of measurements and wall-clock time used for Ansor to match the performance of AutoTVM on the Intel CPU (batch size=1).

Figure 3.11 plots the predicted throughputs vs. measured throughputs. The measured throughputs are normalized to the best performing programs in the test set. The predicted throughputs are the output of the model, so they can be negative. In Figure 3.11a, the points scatter around the diagonal line, meaning that the model makes accurate predictions. The distribution is not uniform because the data set is collected during the search. Good programs have a higher probability to be chosen for measurements, so most of the programs are in the top right corner. The points with measured throughput 0.0 are programs that are invalid or killed due to timeout during measurements. In Figure 3.11b, we sort the 5000 points according to the predictions from the slowest to the fastest, and use the relative ranking as x-axis. So the points are distributed uniformly over x-axis. It shows the distribution of performance of the explored programs better.

The model archives 0.079 RMSE, 0.958 R^2 correlation, 0.851 pairwise comparison accuracy, and 0.624 recall@30 of top-30 programs (see the definition at footnote 1) on the test set.

3.8 Related Work

Automatic tensor program generation based on scheduling languages. Halide [149] introduces a scheduling language that can describe loop optimization primitives. This language is suitable for both manual optimization and automatic search. Halide has three

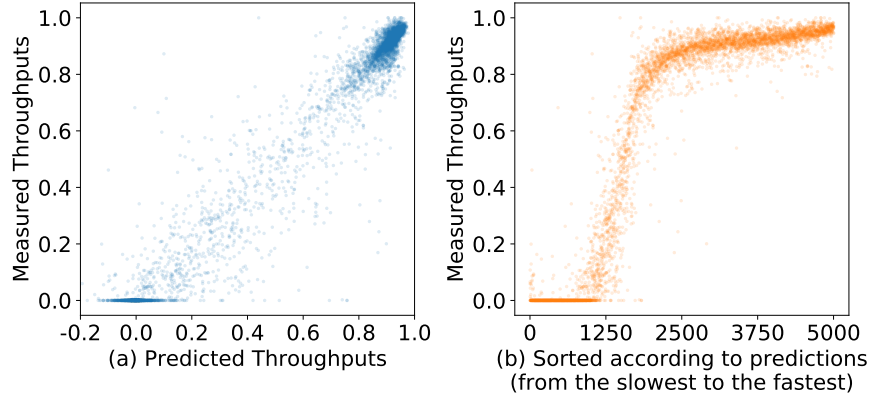


Figure 3.11: Measured throughputs vs. predicted throughputs.

versions of auto-scheduler based on different techniques [126, 104, 2]. The latest one with beam search and learned cost model performs the best among them, which is also used in our evaluation. TVM [34] utilizes a similar scheduling language and includes a template-guided search framework AutoTVM [31]. FlexTensor[229] proposes general templates that can target a set of operators, but its templates are designed for single operators. It is hard to use these templates for optimizations involving multiple operators (e.g., operator fusion). A concurrent work ProTuner [68] uses Monte Carlo tree search to solve the inaccurate estimation problem in Halide auto-scheduler. ProTuner mainly targets image processing workloads, while Ansor targets deep learning workloads and introduces new search space and other optimizations.

Polyhedral compilation models. The polyhedral compilation model [22, 189, 188] formulates the optimization of programs as an integer linear programming (ILP) problem. It optimizes a program with affine loop transformation that minimizes the data reuse distance between dependent statements. Tiramisu [12] and TensorComprehensions [185] are two polyhedral compilers that also target the deep learning domain. Tiramisu provides a scheduling language similar to the Halide language, and it needs manual scheduling. TensorComprehensions can search for GPU code automatically, but it is not yet meant to be used for compute-bounded problems [34]. It cannot outperform TVM on operators like conv2d and matmul [34, 181]. This is because of the lack of certain optimizations [186] and the inaccurate implicit cost model in the polyhedral formulation.

Graph-level optimization for deep learning. Graph-level optimizations treat an operator in the computational graph as a basic unit and perform optimization at graph level without changing the internal implementations of operators. The common optimizations at graph level include layout optimizations [113], operator fusion [34, 135, 230], constant folding [156], auto-batching [115], automatic generation of graph substitution [85] and so forth. The graph-level optimizations are typically complementary to operator-level optimizations. Graph-level optimizations can also benefit from high-performance implementations of oper-

ators. For example, general operator fusion relies on the code generation ability of Ansor. We leave the joint optimization of Ansor and more graph-level optimization as future work.

Search-based compilation and auto-tuning. Search based compilation and auto-tuning have already shown their effectiveness in domains other than deep learning. Stock [162] is a super-optimizer based on random search. Stock searches for loop-free hardware instruction sequences, while Ansor generates tensor programs with nests of loops. OpenTuner [7] is a general framework for program auto-tuning based on multi-armed bandit approaches. OpenTuner relies on user-specified search space, while Ansor constructs the search space automatically. Traditional high-performance libraries such as ATLAS[204] and FFTW [58] also utilize auto-tuning. More recent works NeuroVectorizer [67] and AutoPhase [78, 66] use deep reinforcement learning to automatically vectorize programs and optimize the compiler phase ordering.

3.9 Limitations and Future work

One of Ansor’s limitations is that Ansor cannot optimize graphs with dynamic shapes [165]. Ansor requires the shapes in the computational graph to be static and known in advance to do analysis, construct the search space, and perform measurements. How to generate programs for symbolic or dynamic shape is an interesting future direction. Another limitation is that Ansor only supports dense operators. To support sparse operators (*e.g.*, SpMM) that are commonly used in sparse neural networks [59] and graph neural networks [75], we expect that a large portion of Ansor can still be reused, but we need to redesign the search space. Lastly, Ansor only performs program optimizations at a high level but relies on other code generators (*e.g.*, LLVM and NVCC) to do platform-dependent optimizations (*e.g.*, instruction selection). Ansor comes short of utilizing the special instructions, such as Intel VNNI, NVIDIA Tensor Core, and ARM Dot for mixed-precision and low-precision operators, which are not handled well by the off-the-shelf code generators currently.

3.10 Conclusion

We propose Ansor, an automated search framework that generates high-performance tensor programs for deep neural networks. By efficiently exploring a large search space and prioritizing performance bottlenecks, Ansor finds high-performance programs that are outside the search space of existing approaches. Ansor outperforms existing manual libraries and search-based frameworks on a diverse set of neural networks and hardware platforms by up to 3.8×. By automatically searching for better programs, we hope that Ansor will help bridge the gap between the increasing demand in computing power and limited hardware performance. Ansor is integrated into the Apache TVM open-source project ⁶.

⁶<https://tvm.apache.org/>

Chapter 4

Efficiently Programming Large Language Models using SGLang

Large language models (LLMs) are increasingly used for complex tasks requiring multiple chained generation calls, advanced prompting techniques, control flow, and interaction with external environments. However, efficient systems for programming and executing these applications are lacking. To bridge this gap, we introduce SGLang, a Structured Generation Language for LLMs. SGLang is designed for the efficient programming of LLMs and incorporates primitives for common LLM programming patterns. We have implemented SGLang as a domain-specific language embedded in Python, and we developed an interpreter, a compiler, and a high-performance runtime for SGLang. These components work together to enable optimizations such as parallelism, batching, caching, sharing, and other compilation techniques. Additionally, we propose RadixAttention, a novel technique that maintains a Least Recently Used (LRU) cache of the Key-Value (KV) cache for all requests in a radix tree, enabling automatic KV cache reuse across multiple generation calls at runtime. SGLang simplifies the writing of LLM programs and boosts execution efficiency. Our experiments demonstrate that SGLang can speed up common LLM tasks by up to 5 $\times$, while reducing code complexity and enhancing control. SGLang’s source code is publicly available at <https://github.com/sgl-project/sglang>.

This chapter is based on my previous paper [227], which was a collaborative effort with Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, Ying Sheng. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

```

dimensions = ["Clarity", "Originality", "Evidence"]

@function
def essay_judge(s, essay):
    s += "Please evaluate the following essay. " + essay
    # Evaluate an essay from multiple dimensions in parallel
    forks = s.fork(len(dimensions)) ①
    for f, dim in zip(forks, dimensions):
        f += (
            "Evaluate based on the following metric: " +
            dim + ". End your judgement with the word 'END'"
        )
        f += "Judgment: " + f.gen("judgment", stop="END") ②
    # Merge judgments
    for f, dim in zip(forks, dimensions):
        s += dim + ": " + f["judgment"] ③
    # Generate a summary and give a score
    s += "In summary," + s.gen("summary")
    s += "I give the essay a letter grade of " +
    s += s.gen("grade", choices=["A", "B", "C", "D"]) ④

    ret = essay_judge.run(essay="A long essay ...") ⑤
    print(ret["grade"])

```

Figure 4.1: The implementation of a multi-dimensional essay judge in SGLang utilizes the branch-solve-merge prompting technique [158]. This function uses LLMs to evaluate the quality of an essay from multiple dimensions, merges the judgments, generates a summary, and assigns a final grade. The highlighted regions illustrate the use of SGLang APIs. (1) `fork` creates multiple parallel copies of a prompt. (2) `gen` invokes an LLM generation and stores the result in a variable. (3) `[variable_name]` retrieves the result of the generation. (4) `choices` imposes constraints on the generation. (5) `run` executes a SGLang function with its arguments.

4.1 Introduction

Recent increases in the capabilities of LLMs have broadened their utility, enabling them to tackle a wider range of general tasks and act as autonomous agents [137, 26, 142, 192]. In such applications, LLMs engage in *multi-round* planning, reasoning, and interaction with external environments. This is accomplished through tool usage [161], multiple input modalities [3], and a wide-range of prompting techniques [112], like few-shot learning [25], self-consistency [198], skeleton-of-thought [131], and tree-of-thought [214]. All of these new use cases require multiple, often dependent, LLM generation calls, and critically, the control flow of these programs may depend on the output of LLMs [213]. *The emergence of these complex patterns signifies a shift in our interaction with LLMs, moving from simple turn-based chatting to a more sophisticated form of programmatic usage of LLMs (see section 4.2).* Figure 4.1 shows an example of such an LLM program, where multiple generation calls are used to accomplish a task. The example uses SGLang, a new domain-specific language we introduce in this paper and explain in section 4.4.

To align with emerging trends, the community has seen the development and rising popularity of new interfaces such as LangChain [96], LMQL [19], Guidance [65], and DSPy [87]. These tools help users manage prompt templates and program pipelines with ease. However, their primary focus on front-end design often results in compromised runtime performance.

Conversely, while back-end inference engines like NVIDIA TensorRT-LLM [132], Hugging Face TGI [80], Orca [218], and vLLM [95] continue to reduce latency and improve throughput, their design largely remains anchored in traditional single-generation call interfaces. They typically support APIs similar to the OpenAI Completion API [138], where a user prompt is processed and a response is returned in a stateless manner.

While both front-end and back-end technologies are advancing rapidly, they are evolving along separate trajectories. The front end largely remains oblivious to the LLM inference processes, while the back end often lacks awareness of the application’s structure and is unable to optimize across multiple calls. This disjointed development leads to numerous missed opportunities for performance optimization, such as KV cache sharing and parallelism, as discussed in section 4.2.

In this paper, we present a holistic approach for an efficient LLM programming system by co-designing both the front-end language and the back-end runtime. The process involves three key challenges: (i) creating a flexible, domain-specific language with primitives suitable for LLM programming and ecosystem integration; (ii) ensuring efficient program execution via automatic parallelism, batching, caching, and sharing across multiple calls and multiple programs; and (iii) enhancing program performance through compiler optimizations like code movement and prefetching annotations.

On the front end, we introduce SGLang, a domain-specific language embedded in Python. It provides a set of primitives specifically designed to facilitate programming LLMs. These primitives enable the manipulation of prompts and generations (e.g., `extend`, `gen`, `select`) and the control of parallelism (e.g., `fork`, `join`). Furthermore, SGLang is compatible with Python’s control flow and libraries. Leveraging these primitives along with native Python syntax, users can intuitively develop sophisticated prompting strategies and agents, while obviating the need for manual optimization. Moreover, the unified language enhances portability, enabling users to transition easily between different open-source and proprietary models, such as those from OpenAI and Anthropic.

We develop both an interpreter and a compiler for SGLang. The interpreter is responsible for managing the prompt state and executing SGLang programs. Specifically, it treats a prompt as a stream and submits primitive operations to this stream for asynchronous execution, ensuring proper control over synchronization and intra-program parallelism. Additionally, we can trace a SGLang program and utilize the compiler to compile it into a computational graph. This approach opens opportunities for more aggressive optimizations. We explore several classical compiler optimizations, e.g., code movement, in the context of LLM programming.

On the back end, we propose a novel technique called RadixAttention to automatically reuse the KV cache across multiple generation calls. In existing systems, the KV cache of a request is discarded after processing is completed, which prevents KV cache from being

reused across multiple generation calls. Instead, our system maintains an LRU cache of the KV cache for all requests within a radix tree. This approach allows the runtime to efficiently achieve automatic sharing for various reuse patterns. Additionally, to increase the cache hit rate, we utilize a cache-aware scheduling policy.

In summary, we make the following contributions:

- We present SGLang, a domain-specific language with primitives for LLM programming, supporting parallelism, control flow, nested calls, and external calls.
- We develop an interpreter capable of executing SGLang programs in tandem with the Python interpreter and managing prompt states as asynchronous streams.
- We build a compiler capable of compiling programs into computational graphs and explore classical compiler techniques in the context of LLM programming.
- We propose RadixAttention, a novel technique for automatic KV cache reuse across multiple generation calls at runtime.
- Using SGLang, we implement various LLM applications, including agent control, logical reasoning, content generation, benchmarking, and long document processing. Our experiments demonstrate that SGLang can accelerate common LLM tasks by up to $5\times$, while reducing code complexity and enhancing control.

4.2 Background

The Inference Process of LLMs

Most LLMs in use today, such as GPT-3 [25], PaLM [38], and LLaMA [182], are based on the autoregressive Transformer architecture [187]. The models predict the probability of the next token in a sequence based on the preceding tokens. During inference, the model first processes a sequence of input tokens through a forward pass. It then sequentially decodes output tokens, with each token depending on prior tokens. We refer to the process of taking a sequence of input tokens and generating a sequence of output tokens as a single generation call. Throughout this process, each token generates some intermediate tensors, which are used for decoding further tokens. These intermediate tensors, known as the "KV Cache", are named for the key-value pairs in the self-attention mechanism. An observation that will be important when discussing optimizations later is that the computation of the KV cache only depends on all previous tokens, so different sequences with the same prefix can reuse the KV cache of the prefix tokens and avoid redundant computation.

The Paradigm of Programming LLMs

As LLMs become more powerful, they are increasingly employed as versatile task solvers in complex systems. In these contexts, it is imperative that LLMs operate as dependable and

Language	Syntax	Primitives	Runtime Backends	Execution Modes
LMQL	Custom	extend, gen, select	Hugging Face Transformers, llama.cpp, OpenAI	Compiler
Guidance	Python	extend, gen, select	Hugging Face Transformers, llama.cpp, OpenAI, VertexAI	Interpreter
SGLang	Python	extend, gen, select, fork, join	SGVM, OpenAI, Anthropic, Hugging Face TGI*, llama.cpp*	Interpreter and compiler

Table 4.1: Comparison among LMQL, Guidance, and SGLang. The backends marked with * are experimental.

intelligent components. This evolves the use of LLMs from single, straightforward generation calls to multiple, interdependent generation calls interleaved with control flow, leading to the emergence of the "Programming LLMs" paradigm. Programming LLMs refers to the development of computer programs that control the generation processes of LLMs. This concept can be exemplified in several ways.

First, advanced prompting techniques are examples of programming LLMs. An exemplary case is meta-reasoning [216]. To solve a problem, the LLM initially generates multiple solutions through chain-of-thought prompting. It then performs meta-reasoning across these solutions to derive the final resolution. Another example is skeleton-of-thought [131]. To answer a question, the LLM first generates a skeleton of the answer. It then issues multiple parallel calls to elaborate the skeleton into a detailed explanation. A more complex case is the tree-of-thought approach [214], which integrates LLM calls within tree search algorithms (e.g., breadth-first search).

Second, in LLM-based agents, the programmatic use of LLMs is crucial. Take the ReAct agent [213], for example. The behavior of the agent is controlled by loops and conditional statements. The agent performs one action per iteration. In each iteration, the agent chooses the action type (search or finish), generates action parameters, and processes observations from the external environment.

Third, simpler prompting forms also fall under the scope of programming LLMs. Techniques like few-shot prompting [25], chain-of-thought prompting [202], and retrieval-augmented generation [101] are basic elements in programming LLMs.

Challenges and Opportunities

Designing programming systems for LLMs presents unique challenges and opportunities. A good system should help a programmer be productive, offering an intuitive and expressive interface for common programming tasks, including LLM state management, decoding process control, and smooth integration with external tools. Furthermore, it is essential to ensure performance by optimizing for low latency and high throughput execution. In particular, the rich structure of LLM programs and the unique properties of LLM inference provide plenty of opportunities for runtime optimizations. These include:

- **Caching:** Often in LLM programs, multiple text segments and generation calls are appended to a single prompt. Caching the computed KV cache for previous tokens across multiple chained calls can reduce redundant computation. This optimization,

however, is neither free nor trivial, as it requires additional storage and more complex memory management.

- **Batching:** The decoding process in LLMs is primarily memory-bound, meaning increasing the batch size can significantly enhance throughput. Running a large number of LLM programs in a batch, and utilizing the contiguous batching technique [218] can be beneficial.
- **Sharing:** It is common in LLM programs to generate multiple outputs from a single prompt or to fork a new prompt from the current state. Basic prefix sharing has been investigated in vLLM [95]. More advanced sharing patterns like irregular tree-structured sharing can also be employed. Figure 4.2 shows four typical patterns of KV cache sharing across multiple calls; none of the existing systems can automatically handle all of them.
- **Parallelism:** An LLM program often includes multiple generation calls. By creating a dependency graph for these calls, we can identify independent calls and execute them in parallel. This creates intra-program parallelism, which adds additional parallelism to the straightforward inter-program parallelism. For example, in Figure 4.1, the generation of multiple judgments can be parallelized.
- **Compilation:** Once a full program is available, it can be compiled into an optimized intermediate representation for more efficient execution. More aggressive optimizations can also be implemented, such as automatically adjusting the original prompts based on user-provided test cases. These optimizations may include expanding, compressing, rearranging parts of the prompts, and selecting different models for different calls.

To systematically explore these opportunities, it is essential to co-design the programming interface and runtime. However, inference engines like Orca and vLLM fall short, as they optimize only for single generation calls. This limitation leads to redundant computations and missed opportunities when executing LLM programs that involve multiple calls.

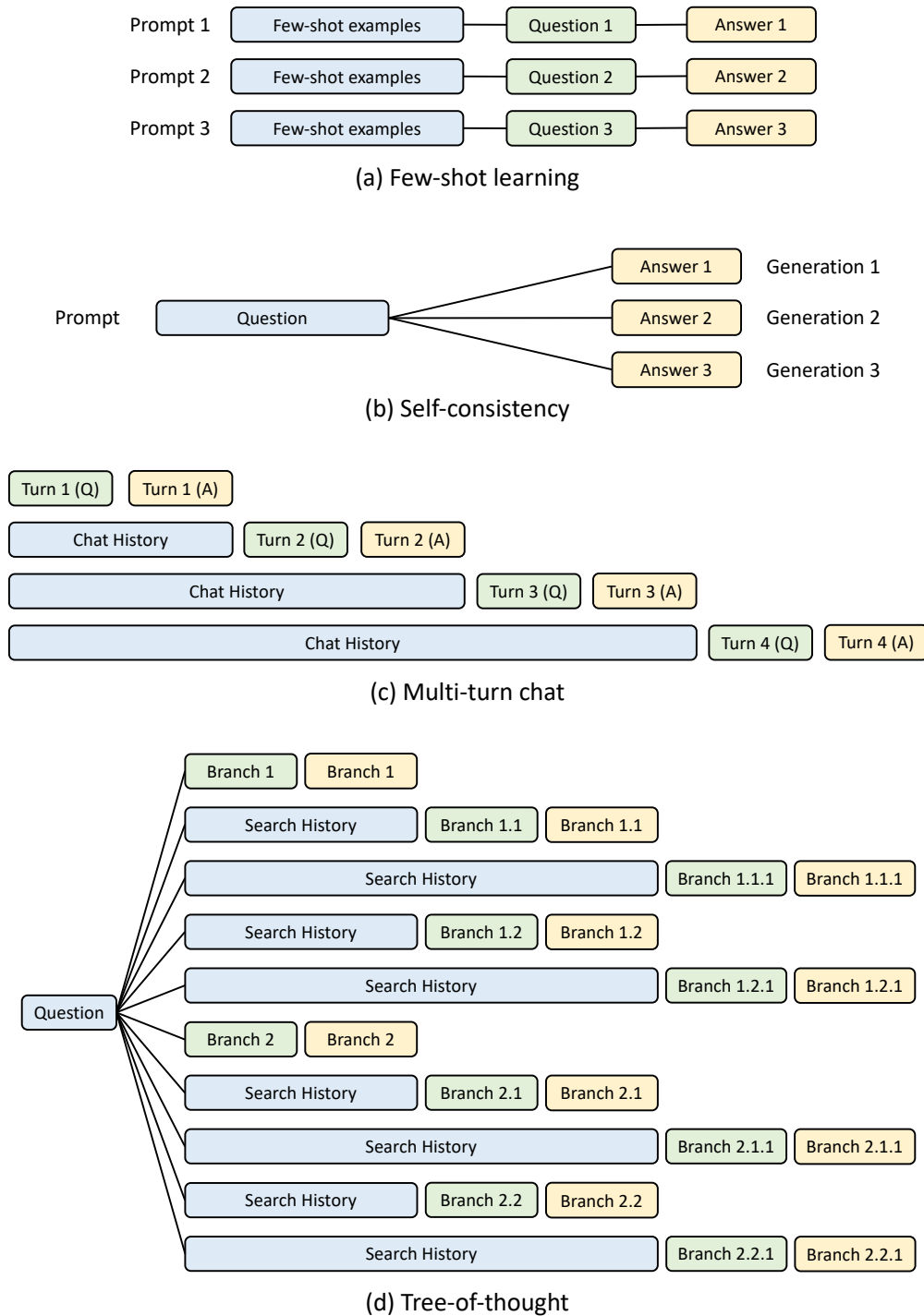


Figure 4.2: KV cache sharing examples. Blue boxes represent shareable prompt parts, green boxes indicate non-shareable parts, and yellow boxes mark non-shareable model outputs. Shareable elements include few-shot learning examples, questions in self-consistency [198], chat history in multi-turn chat, and search history in tree-of-thought [214].

Existing LLM Programming Systems

Given the significance of LLM programming, specialized programming systems have been developed. The top layer comprises high-level application development libraries with pre-built modules, such as LangChain [96]. The middle layer features more flexible pipeline programming frameworks like DSPy [87]. At the bottom layer, there are domain-specific languages designed for controlling a single prompt, including LMQL [19] and Guidance [65].

SGLang, uniquely positioned between the middle and bottom layers, shares similarities with LMQL and Guidance. However, unlike LMQL and Guidance, which primarily focus on language design while often overlooking runtime efficiency, SGLang thoughtfully integrates front-end language features with back-end runtime optimizations. This approach significantly enhances runtime efficiency, sometimes by an order of magnitude or more. For its front-end, LMQL employs an SQL-like custom syntax, contrasting with the more intuitive Python syntax. Guidance initially used the Handlebars template language and later transitioned to a Pythonic language embedded within Python. The front-end language of SGLang, inspired by Guidance, is being developed concurrently with the new version of Guidance.

SGLang introduces new primitives for enhanced expressiveness and runtime efficiency. It also offers the flexibility of both interpreter and compiler modes. A comparison among LMQL, Guidance, and SGLang is presented in Table 4.1. We omit the comparison with LangChain and DSPy because they are at a higher level. It is possible to compile high-level pipelines from LangChain and DSPy to SGLang.

4.3 Overview

Figure 4.3 shows an overview of our system. At the user level, SGLang programs are written in Python with embedded SGLang primitives for managing prompts and generation calls. The syntax and primitives of SGLang will be introduced in section 4.4. SGLang programs can be executed in two ways: either using an interpreter or a compiler. When using the interpreter, SGLang programs, which are also valid Python programs, are processed by both the Python and SGLang interpreters. The SGLang interpreter handles specific SGLang primitives. The design of this interpreter is explained in section 4.4. Alternatively, a limited set of SGLang programs can be compiled, enabling further optimizations. section 4.4 is dedicated to the SGLang compiler. Both the interpreter and the compiler work with the SGVM, which is our custom serving engine for LLM generation calls (section 4.5). A key aspect of SGVM is its RadixAttention feature, which automatically reuses KV caches across multiple calls.

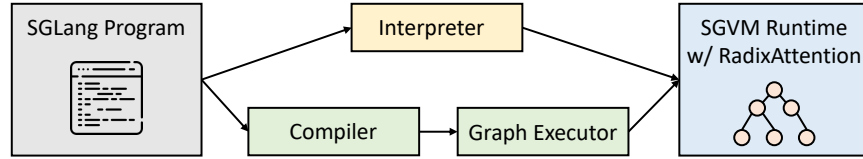


Figure 4.3: The overview of the system.

4.4 Language, Interpreter, and Compiler

Syntax and Primitives

SGLang is implemented as an embedded domain-specific language in Python.

Figure 4.1 shows an example of a SGLang program. This program evaluates the quality of an essay from multiple dimensions using the branch-solve-merge prompting method [158]. The function `essay_judge` takes two arguments: `s` and `essay`. The variable `s` manages the state of a prompt, allowing new strings and SGLang primitives to be appended using `+=`. In the function, it first adds the essay to the prompt, and then forks the prompt into three copies. These forks run in parallel, generating judgments from different dimensions. Finally, it merges these judgments, generates a summary, and assigns a letter grade to the essay.

Figure 4.4 presents another example. This program generates two tips in parallel for a given topic using the skeleton-of-thought prompting method [131]. It employs two functions: `expand`, which elaborates a short tip into a detailed paragraph, and `tip_suggestion`, which orchestrates the entire process. It first generates two short tips and then calls `expand` to elaborate them in parallel. Finally, it generates a summary.

These examples illustrate the use of SGLang. In SGLang, we provide the following primitives: `gen` for calling LLM generation, `select` for letting the LLM choose the option with the highest probability from a list, `+=` or `extend` for extending the current prompt, `fork` for forking the current prompt state, and `join` for rejoining the forked prompt states. Users can interleave these primitives with arbitrary Python control flow and libraries.

Interpreter

The most basic way to execute a SGLang program is through an interpreter. In SGLang, a prompt is treated as an asynchronous stream. Executing primitives such as `extend`, `gen`, and `select` on a prompt leads to the interpreter submitting these primitives into the stream for asynchronous execution. These submission calls are non-blocking, allowing the Python interpreter to continue executing Python code without waiting for the LLM generation to finish. This process is akin to launching CUDA kernels to CUDA streams asynchronously. Each prompt is managed by a stream executor in a background thread, enabling users to achieve parallelism.

When fetching the generation results from a prompt, the fetch action will be blocked until the desired generation results are ready. This is implemented using event mechanisms, where

```

@function
def expand(s, tip):
    s += (
        "Please expand the following tip into a "
        "detailed paragraph: " + tip + "\n"
    )
    s += s.gen("paragraph")

@function
def tip_suggestion(s, topic):
    s += "Here are 2 concise tips for " + topic + ".\n"

    # Generate a skeleton of two short tips
    s += "1." + s.gen("tip_1", stop=["\n", ":", "."]) + "\n"
    s += "2." + s.gen("tip_2", stop=["\n", ":", "."]) + "\n"

    # Expand the tips into detailed paragraphs in parallel
    detailed_tip1 = expand(tip=s["tip_1"])
    detailed_tip2 = expand(tip=s["tip_2"])

    # Merge the results and generate a summary
    s += "Tip 1: " + detailed_tip1["paragraph"] + "\n"
    s += "Tip 2: " + detailed_tip2["paragraph"] + "\n"
    s += "In summary" + s.gen("summary")

state = tip_suggestion.run(topic="staying healthy")
print(state.text())

```

Figure 4.4: The SGLang program for parallel tip suggestion with skeleton-of-thought prompting.

a generation call sets an event upon producing its output. Thus, the interpreter effectively manages intra-program parallelism and synchronization.

Compiler

Another way to run SGLang programs is to compile them as computational graphs and run them with a graph executor. This opens the opportunity for more compilation optimizations, as we can rewrite the graph.

Tracing and Graph Executor

We design an intermediate representation (IR) for SGLang, representing SGLang program structures and operations as a computational graph. This graph includes nodes for primitive operators and edges for dependencies. See Figure 4.5 for the graph corresponding to the program in Figure 4.4. In the program, each call to a decorated function and or fork will create a new prompt state, or a stream.

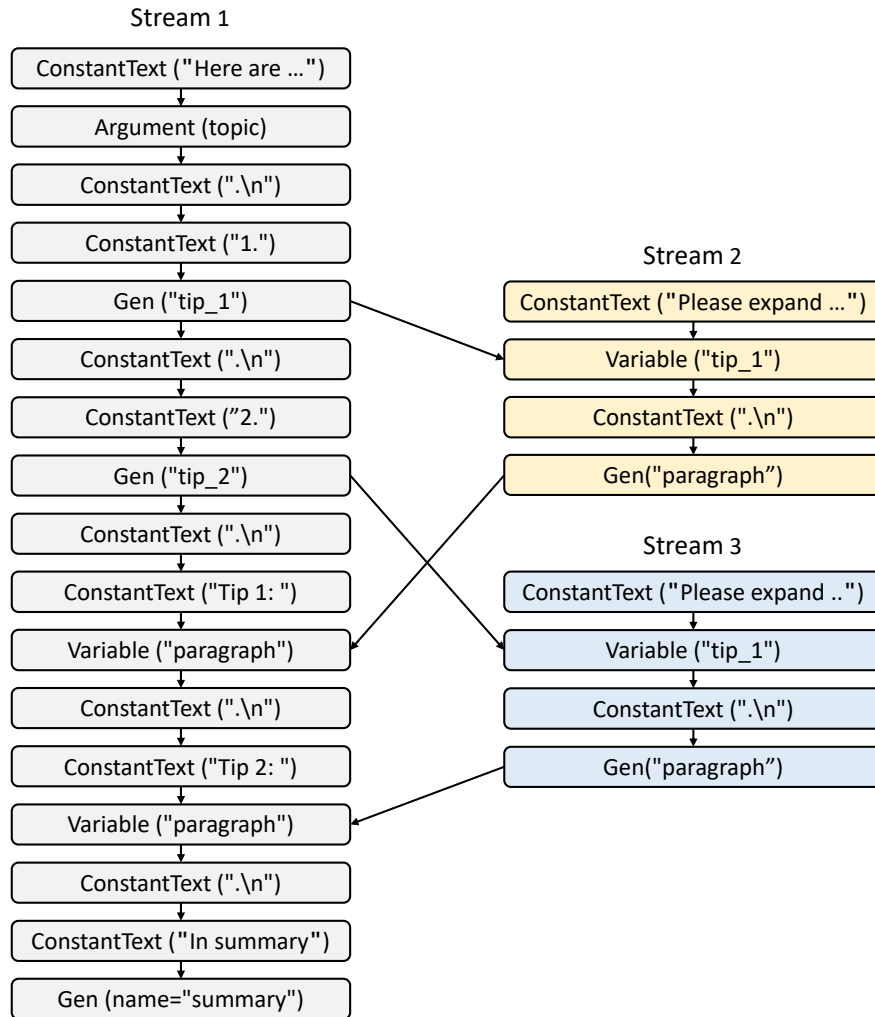


Figure 4.5: A computational graph for the program in Figure 4.4. The three streams correspond to three function calls.

There are several types of nodes. Each operand of `+=` and `+` in a SGLang program is represented by an IR node. These include `ConstantText`, `Argument`, `Gen`, `Select`, `Variable`, `Fork`, `GetForkItem`, and `Join`. There are two types of dependencies. The first is intra-stream dependency, where operations submitted into a stream using `+=` must be executed after all preceding operations in that stream. The second is inter-stream dependency, which occurs when one stream needs to fetch variable values from another stream, necessitating synchronization. Operations like `fork` manipulate multiple streams and thus introduce inter-stream dependencies.

To generate the graph, we use tracing to run the program with abstract arguments and construct the graph dynamically. This method is limited to programs without data-

dependent control flow, a limitation we plan to address in future work. Once constructed, the graph can be executed through a graph executor, eliminating the need to reinterpret the original Python program. This results in benefits like graph rewriting optimizations, reduced runtime overhead, and program serialization. For execution, stream executors are launched for each data stream, dispatching IR nodes to the streams in topological order.

Compilation Optimizations

We explore two compilation optimizations for SGLang IR. The first one is noteworthy as it integrates GPT-4 into compiler optimization processes. The second one is more conventional. We anticipate more classical compiler techniques can also be applied, such as auto-tuning and instruction selection.

Code Movement for Improving Prefix Sharing. This optimization aims to improve prefix sharing by reordering nodes in the graph to increase the length of the constant prefix. It does not strictly preserve the original computation, classifying it as an aggressive optimization. For instance, changing the prompt from “Here is a question + {question}. Please act as a math expert and solve the given question” to “Please act as a math expert and solve the given question. Here is a question + {question}.” results in a longer sharable prefix. This optimization is interesting because traditional program analysis cannot achieve it, due to the presence of natural language instructions in SGLang. Instead, we prompt GPT-4 to reorder graph nodes. We write a prompt with several examples to teach GPT-4 the concepts of SGLang IR, and we find that GPT-4 can successfully apply this optimization for some simple SGLang programs.

Prefetching Annotations. This involves automatically adding prefetching nodes to the graph, signaling the runtime to move the cached KV cache from the CPU to the GPU if the cache exists for a long prompt.

4.5 Runtime with RadixAttention

The primary challenge in optimizing the execution of SGLang programs lies in reusing the KV cache across multiple calls and programs, as discussed in section 4.2. While some existing systems can handle KV cache reuse in certain scenarios, this often requires manual configurations and ad-hoc adjustments. However, even with manual configurations, no existing system can automatically handle all scenarios due to the variety of possible reuse patterns.

This section introduces RadixAttention, a novel technique for automatic KV cache reuse during runtime. Instead of discarding the KV cache after finishing a generation request, our approach retains the KV cache for both prompts and generation results in a radix tree. This structure enables efficient prefix search, reuse, insertion, and eviction. We implement a Least Recently Used (LRU) eviction policy, complemented by a cache-aware scheduling policy, to enhance the cache hit rate. Furthermore, RadixAttention is compatible with existing techniques like continuous batching and paged attention.

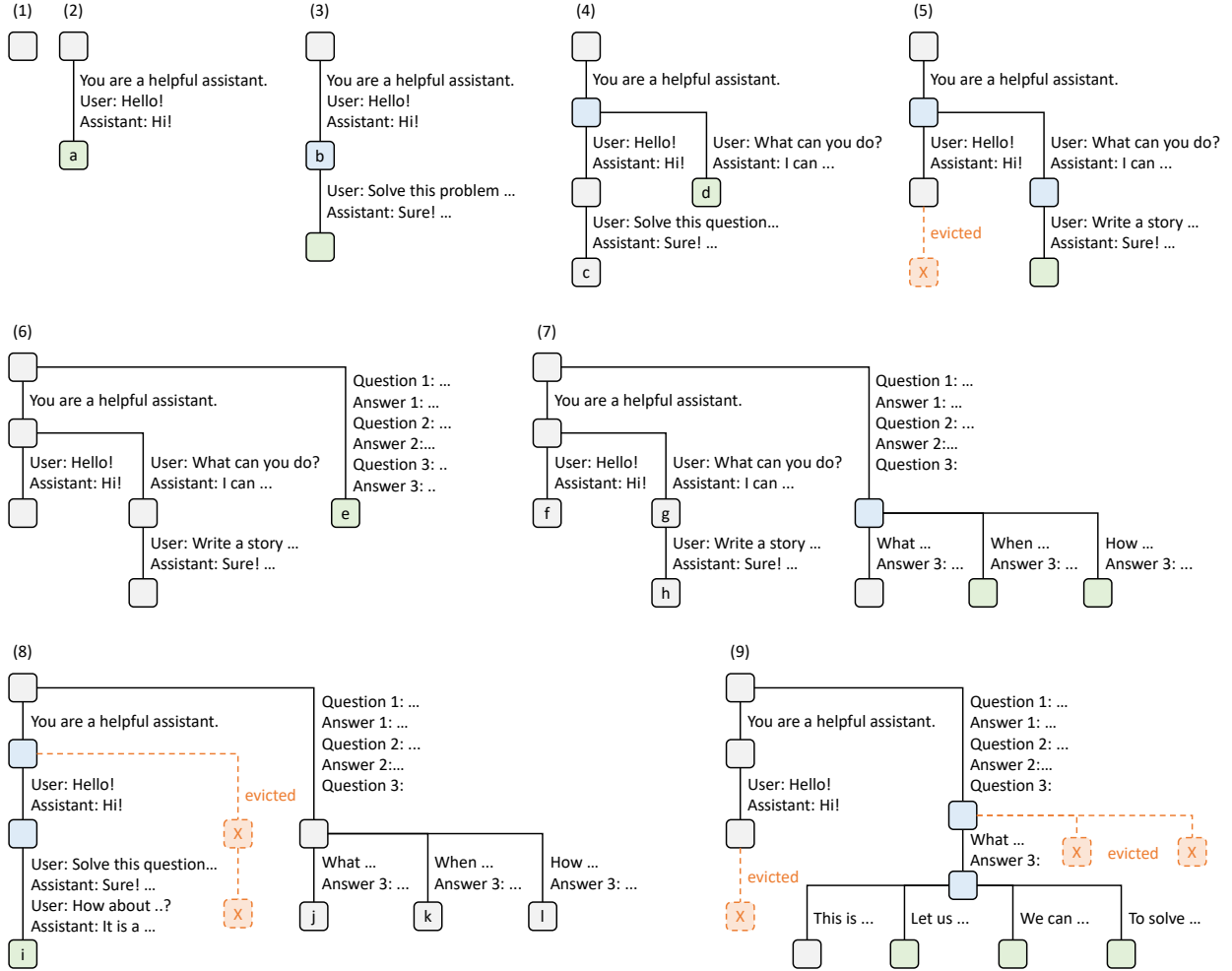


Figure 4.6: Examples of RadixAttention operations with an LRU eviction policy, illustrated across nine time points. The figure demonstrates the dynamic evolution of the radix tree in response to various requests. These requests include two chat sessions, a batch of few-shot learning inquiries, and a self-consistency sampling. Each tree edge carries a label denoting a substring or a sequence of tokens. The nodes are color-coded to reflect different states: green for newly added nodes, blue for cached nodes accessed during the time point, and red for nodes that have been evicted. In step (1), the radix tree is initially empty. In step (2), the server processes an incoming user message "Hello" and responds with the LLM output "Hi". The system prompt "You are a helpful assistant", the user message "Hello!", and the LLM reply "Hi!" are consolidated into the tree as a single edge linked to a new node. In step (3), a new prompt arrives and the server finds the prefix of the prompt (i.e., the first turn of the conversation) in the radix tree and reuses its KV cache. The new turn is appended to the tree as a new node. In step (4), a new chat session begins. The node "b" from (3) is split into two nodes to allow the two chat sessions to share the system prompt. In step (5), the second chat session continues. However, due to the memory limit, node "c" from (4) must be evicted. The new turn is appended after node "d" in (4). In step (6), the server receives a few-shot learning query, processes it, and inserts it into the tree. The root node is split because the new query does not share any prefix with existing nodes. In step (7), the server receives a batch of additional few-shot learning queries. These queries share the same set of few-shot examples, so we split node "e" from (6) to enable sharing. In step (8), the server receives a new message from the first chat session. It evicts all nodes from the second chat session (node "g" and "h") as they are least recently used. In step (9), the server receives a request to sample more answers for the questions in node "j" from (8), likely for self-consistency prompting. To make space for these requests, we evict node "i", "k", and "l" in (8).

RadixAttention

A radix tree is a data structure that serves as a space-efficient alternative to a trie (prefix tree). Unlike typical trees, the edges of a radix tree can be labeled with not just single elements, but also with sequences of elements of varying lengths. This feature significantly boosts the efficiency of radix trees. In our system, we utilize a radix tree to manage a mapping. This mapping is between sequences of tokens, which act as the keys, and their corresponding KV cache tensors, which serve as the values. These KV cache tensors are stored in a non-contiguous, paged layout, where the size of each page is equivalent to one token.

Considering the limited capacity of GPU memory, we cannot retrain infinite KV cache tensors, which necessitates an eviction policy. To tackle this, we implement an LRU eviction policy that recursively evicts leaf nodes. In the continuous batching setting, we cannot evict nodes that are being used by the currently running batch. Therefore, each node maintains a reference counter indicating how many running requests are using it. A node is evictable if its reference counter is zero.

The front end always sends full prompts to the runtime and the runtime will automatically do prefix matching, reuse, and caching. Figure 4.6 illustrates how the radix tree is maintained when processing several incoming requests. The tree structure is stored on the CPU and the maintenance overhead is small.

Cache-Aware Scheduling

To increase the cache hit rate, the scheduler must be cache-aware. Otherwise, when a large batch of requests arrives, the scheduler may switch between different requests, leading to cache thrashing. We introduce the algorithm¹ for cache-aware scheduling in RadixAttention with contiguous batching. The key idea is to sort the requests by matched prefix length, as opposed to adhering to a first-come-first-serve schedule. There are further opportunities for the implementation of more advanced scheduling strategies that assign priority based on the weights of subtrees. Essentially, this would involve counting the total number of requests within each sub-tree(prefix-sharing trees) and even considering requests that have been historically evicted. The weighted priority can fully utilize the tree-shape locality of some LLM tasks such as tree-of-thought [214], making the least redundant evictions.

Other Optimizations

The development of new CUDA kernels is essential to enhance the efficiency of KV cache reuse. Existing systems mostly implement CUDA kernels for two main functions: prefill and decoding. The prefill kernel assumes that the lengths of the query and key are identical. Conversely, the decode kernel assumes that the query length is one. However, in the case

¹Please refer to the original paper [227] for the pseudocode of the algorithm. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

of KV cache reuse, these kernels cannot be directly applied. We introduce a new “extend” kernel for this computation, capable of reading the KV cache from both contiguous buffers and non-contiguous memory pools. This fused extend kernel operates without extra copy overhead.

4.6 Evaluation

SGLang and SGVM are implemented in approximately 9K lines of Python code. SGVM reuses some components from LightLLM [123], an LLM serving system based on PyTorch [143] and Triton [181].

We evaluate the performance of SGLang across a diverse range of LLM workloads. Subsequently, we conduct case studies and ablation studies to demonstrate the effectiveness of specific components.

Setup

Models. We test the Llama 2 chat models [182], CodeLlama models [157], and vision language model LLaVA [110] models of various sizes. The number of parameters ranges from 7B to 33B. We use float16 precision for Llama and CodeLlama models, and 4-bit quantized precision for LLaVA models.

Hardware. Most experiments are conducted on AWS EC2 G5 instances unless otherwise stated. The GPUs on G5 instances are NVIDIA A10G GPUs, each with 24 GB of memory. The CPUs are AMD EPYC processors. For most experiments, we run 7B models with a single A10G GPU. To see how our system handles larger models, we also test a 33B model on an A100 GPU (80GB).

Baseline. We compare SGLang against both high-level programming systems with their respective languages and default runtimes, as well as low-level inference engines with standard OpenAI-like Completion APIs. The baselines include:

- Guidance [65], a language for controlling LLMs. We use Guidance v0.1.4 with the llama.cpp backend.
- LMQL [19], a query language for LLMs. We use LMQL v0.7.3 with the Hugging Face Transformers backend.
- vLLM [95], a high-throughput inference engine. We use vLLM v0.2.2 and its default API server.

Metrics. We report two performance metrics: throughput and latency. For throughput, we run a sufficiently large batch of program instances to compute the maximum throughput, comparing the number of program instances executed per second (programs per second, p/s). For latency, we execute a single program at a time without batching and report the average

latency for multiple instances. Unless otherwise stated, most of our optimizations do not change the computation results.

While our primary focus is on runtime efficiency, we also conduct several case studies to compare the number of lines of code for productivity.

End-to-End Benchmark

Few-Shot In-Context Learning

We start with the most basic few-shot learning prompting. We evaluate the maximum throughput of systems during batch processing of few-shot learning requests using three benchmarks, each with 5 shots. Results are shown in Figure 4.7.

MMLU [72] is a multi-choice knowledge benchmark where the LLM outputs a single token (A/B/C/D) for answer selection. LMQL exhibits the worst performance due to its slow backend and complex pre- and post-processing. Guidance, being faster than LMQL, benefits from reduced processing overhead and some KV cache reuse. However, both systems support only a batch size of one. vLLM performs similarly to Guidance, but it lacks automatic KV cache reuse for the shared prefix. Although the vLLM paper discusses this feature, the released code does not support it due to the absence of a reliable high-performance kernel. Even if a high-performance kernel is implemented for vLLM, manual configuration is required to use this feature. Conversely, SGLang automatically achieves sharing with RadixAttention and attains a throughput that is $4.4\times$ higher than its closest competitor.

HellaSwag [222] is a multi-choice benchmark where each option is a short string. Unlike MMLU, which decodes a single output token, HellaSwag requires an LLM to calculate the probability of each string choice by performing a forward pass. LMQL exhibits slowness due to high overhead. vLLM lacks a “select” primitive. Instead, it concatenates each choice with the prompt and computes the probability for the entire concatenated prompt for all choices, leading to redundant calculations. Guidance, although offering a “select” feature, is limited to a batch size of 1. SGLang outperforms others by $2\times$ with automatic KV cache sharing and batching.

GSM-8K [40] is a benchmark that requires LLMs to generate a reasoning process and produce an integer answer. Since the benchmark demands longer output lengths, batching is crucial for enhanced throughput. However, LMQL and guidance currently lack support for batching, leading to reduced performance. vLLM is also slow due to its inability to reuse the KV cache for the shared prefix. In contrast, SGLang achieves a $4.5\times$ increase in throughput, thanks to the integration of RadixAttention and contiguous batching.

Due to its overhead, LMQL often performs significantly worse than other systems in both short and long-generation tasks, so we will not include it in our future benchmarks.

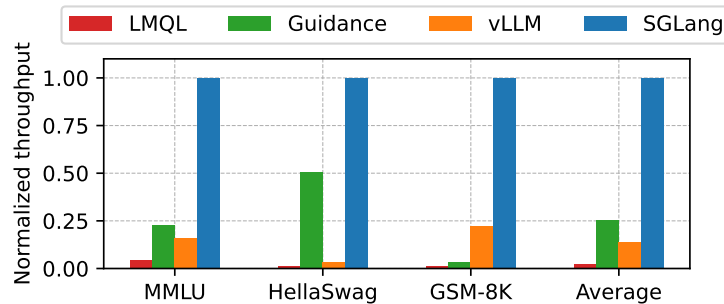


Figure 4.7: Throughput on few-shot in-context learning tasks.

LLM-Based Agents

We benchmark on two highly regarded agent frameworks: ReAct agent [213], which processes tasks in an interleaved manner by chaining multiple LLM calls with actions like web search; and generative agents [142], employed in a gaming environment where agents interact with each other and make decisions by invoking LLMs. Both original papers utilized OpenAI models. To accurately gauge potential performance gains, we mirrored the settings described in the papers and used GPT-3.5-turbo as the backbone model, and then traced the input and output of LLM calls to create a dataset. For ReAct agents, we used the HotpotQA [212] evaluation as reported in the paper. For generative agents, we rewrote the top five functions (out of forty traced) from the game using SGLang, which accounts for approximately 70% of all LLM calls. We then replayed these traces on our experimental platform.

As depicted in Figure 4.8, the ReAct agent, using SGLang, achieves a throughput that is $5.6\times$ higher than that of vLLM and has only 13% of its latency. This improvement primarily results from the agent’s process of appending current states (thoughts, actions, and observations) to the prompt for subsequent LLM calls. Our RadixAttention mechanism effectively identifies and minimizes redundant computations, a capability not present in the current inference engines. This mechanism also significantly reduces memory usage, allowing more questions to be addressed by a greater number of agents, thereby optimizing GPU utilization and enhancing throughput.

In the case of the generative agent task, the gaming setting’s constraints permit only a single LLM call per simulation to maintain game logic consistency. In this case, SGLang still outperforms vLLM by 30% in both throughput and latency. This efficiency is due to the caching of agents’ predefined routines, reducing redundant computations. The task prompt templates contain multiple arguments that vary at different speeds based on timestamps, resulting in complex prefix reuse patterns. Nevertheless, the SGLang runtime can automatically detect these patterns and reuse the KV cache.

Reasoning

We benchmark the performance of systems using advanced prompting techniques for reasoning tasks. These tasks involve solving GSM-8K problems in a zero-shot setting. Both

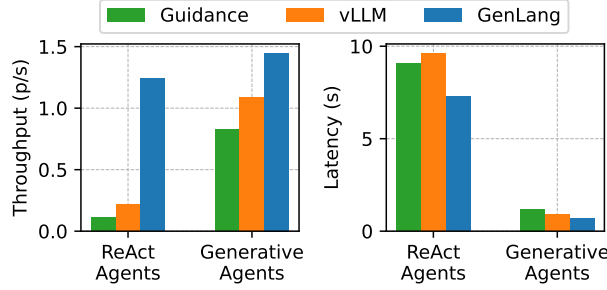


Figure 4.8: Throughput and latency on agent tasks.

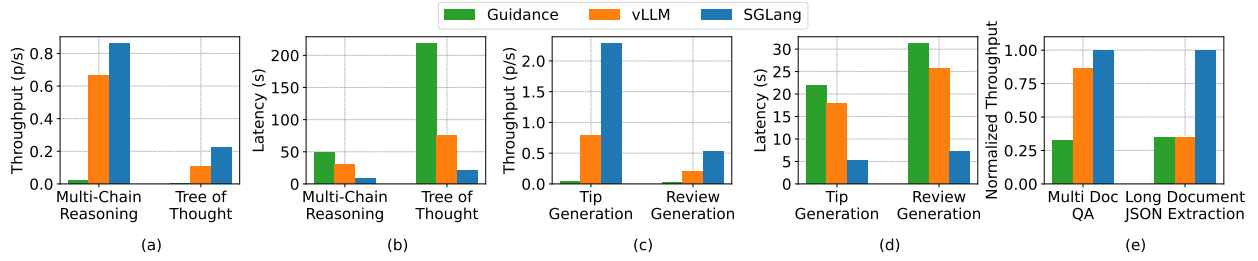


Figure 4.9: (a) Throughput on reasoning tasks. (b) Latency on reasoning tasks. (c) Throughput on content generation tasks. (d) Latency on content generation tasks. (e) Throughput on long-context tasks.

throughput and latency metrics are reported in Figure 4.9 (a)(b).

Multi-chain reasoning [216] involves generating several solutions to one question and using meta-reasoning across these solutions for the final answer. This process allows for sharing the question and parallelizing the generation of multiple solutions. For each solution generation, we use various eliciting prompts such as "Think step-by-step" and "Take a deep breath". Unlike simple parallel sampling, this flexible approach with different eliciting prompts is not supported by the vLLM API. Hence, vLLM exhibits lower throughput due to its inability to share prefixes flexibly. Guidance has lower throughput due to no batching. In contrast, SGLang allows users to intuitively implement this technique without worrying about manual optimizations, yet achieve high throughput. The improvement in throughput is not as substantial as in previous cases of few-shot learning, due to the zero-shot setting having a shorter shareable prefix.

Regarding latency, the SGLang interpreter can automatically resolve dependencies and submit requests in parallel. For a fair comparison, we directly translate the SGLang program into an equivalent Python program without explicitly using threading or asyncio to avoid added complexity. Consequently, the translated program does not parallelize parallelizable requests, resulting in higher latency.

Tree-of-thought [214] incorporates LLMs into the breadth-first tree-search algorithm. The LLMs must propose the next steps for branch exploration, execute these proposals, and evaluate the solutions. This method leads to complex sharing patterns beyond the capabilities of existing systems. Once again, SGLang surpasses others in terms of both higher throughput and lower latency in this benchmark.

Content Generation

We evaluate two advanced prompting methods for content generation: tip generation and article review. The skeleton-of-thought technique [131] first creates a brief outline of tips, then elaborates each into a detailed paragraph. The branch-solve-merge approach [158] assesses an article from multiple dimensions and then assigns a final grade. Results are shown in Figure 4.9 (c)(d). In both cases, there are opportunities for sharing and intra-program parallelism. SGLang systematically exploits them through an intuitive programming model, achieving higher throughput and reduced latency.

Long Document Processing

We benchmark the performance on long document processing tasks. When the document is long, KV cache reuse becomes more important as the forward pass to encode the prompt becomes significantly longer. We use CodeLlama models, which were fine-tuned on a context length of 16K tokens. On our setup, running a 7B model in fp16 precision on a 24 GB GPU, we can store at most 15K tokens. We select inputs exceeding 10K tokens in length, meaning the system can process only one request at a time at most. The first benchmark tests answering questions using multiple document segments from a paper, akin to a retrieval augmented generation setup. The second benchmark involves summarizing a city’s information from its Wikipedia page and outputting a JSON object.

For the first benchmark, we employ the fork/join primitives in SGLang to implement parallel-context window [153]. By using this technique, we can encode multiple documents separately in parallel and concatenate the resulting KV cache, without letting them attend to each other. This approach reduces the quadratic computational cost of self-attention for long prompts. While this optimization alters the computation, it does not negatively impact accuracy in our benchmark. For the second benchmark, we reuse the KV cache of the long prompts when decoding multiple fields of the JSON object. The results are shown in Figure 4.9 (e). SGLang achieves $1.2\times$ and $2.9\times$ improvement on these two benchmarks, respectively. Since the maximum batch size is one, latency improvements are the same as throughput improvements.

Case Study

Vision Language Models

We conduct a case study to demonstrate the extensibility of SGLang by supporting a new vision language model, LLaVA v1.5 [110], on a new hardware platform, Apple Silicon. To achieve this, we add a new primitive, `sgl.image`, to accept image inputs, and incorporate a new backend, `llama.cpp` [61], an inference engine compatible with a wide range of hardware.

We integrate `sgl.image` as a new IR node into the interpreter and reuse the SGVM protocol to build a server based on `llama.cpp`. This integration is achieved in around 400 lines of code. Additionally, the code that uses `sgl.image` can be executed on GPT-4 Vision

through the OpenAI back-end without code modification. We create a cache for image tokens based on the image’s hash ID, similar to how we maintain RadixAttention. We run the LLaVA-Bench and report the throughput using SGLang and the default llama.cpp server on an M2 Ultra. The LLaVA-Bench contains a total of 24 images and 60 questions, with 2.5 questions on each image on average. SGLang increases the throughput of llama.cpp by $1.7\times$. The speedup is mainly affected by the number of questions per shared image and the output length.

Compiler Optimizations

We evaluate the effectiveness of two compiler optimizations. The first is GPT-4-assisted code movement for prefix sharing. We collect 20 prompt templates from the internet and implement them in SGLang. We utilize 5 of these templates for few-shot training examples and the remaining 15 as test cases. Our evaluation shows that, for 12 out of the 15 templates, GPT-4 successfully reorders the graph nodes without altering the semantics, as confirmed by manual inspection of the modified prompts. On average, this optimization results in a 60-token increase in the sharable prefix length, showcasing GPT-4’s effectiveness. Failures in creating optimized prompt order come from an incorrect understanding of the semantic meaning behind the graph nodes. It is too aggressive and puts all constants upfront even when such ordering changes the original semantics. This case study aims to explore the use of GPT-4 for compiler optimizations. More work is needed to make these kinds of optimizations reliable in the future.

The second optimization involves prefetching annotation insertion. For frequently used long prompts, we cache them in CPU memory. The compiler automatically inserts prefetch nodes into the computational graph. During runtime, when the server encounters a prefetch node, it evicts nodes from the radix tree and swaps in the cached prefix. As the cost of swapping can be substantially overlapped, this reduces the first-token latency of the subsequent request. We test SGLang programs that contain function calls with a 4k prefix length. By prefetching the 4k prefix, the first token latency of the called function has been reduced from 1 second to 0.2 seconds, which is a 80% reduction.

Productivity

We conducted a case study to demonstrate SGLang’s productivity improvement by comparing its Lines of Code (LoC) with OpenAI APIs. We designed a prompting flow for article analysis: if it’s about a celebrity, the program generates a bio and introduction; otherwise, it summarizes the article. Implementing this flow in both OpenAI APIs and SGLang, we counted effective LoC, excluding comments and empty lines. The results show the OpenAI API required 206 lines, while SGLang needed only 91, marking a 55% reduction.

We delve into the implementations and summarize the insights regarding why SGLang is capable of using fewer LoC. First, whereas users must manually extract content from OpenAI API responses and chain them to form a multi-step flow, SGLang simplifies this process by

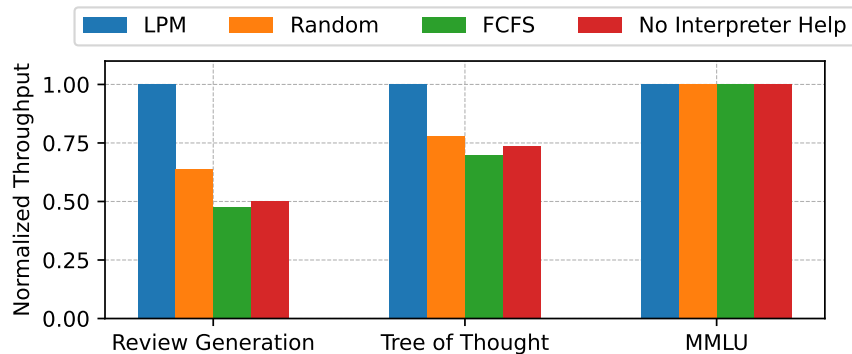


Figure 4.10: Performance of different scheduling policies.

leveraging the string concatenation operator. Second, since the flow involves several calls with restricted answers, such as true/false and integers, users are required to construct an appropriate logit bias to ensure OpenAI returns one of the expected answers. In contrast, SGLang provides primitives that automatically construct logit bias for this purpose, allowing users to merely specify the acceptable data type and choices. Third, in cases where skeleton-based prompting is used to reduce end-to-end latency, OpenAI API users need to initiate a thread pool for parallel request submission explicitly. However, SGLang’s built-in primitive `fork` automatically parallelizes requests.

Larger Models

While most of our experiments are conducted using a 7B model, our techniques also perform effectively on larger models. We test a 13B model on an A100 (40GB) and a 33B model on an A100 (80GB) using the ReAct agent benchmark, achieving a similar speedup ($5\times$).

Ablation Study

Scheduling Policy

We evaluate the effectiveness of our cache-aware scheduling by comparing it with several other policies. The results are shown in Figure 4.10. “LPM”, or longest prefix matching, is the default policy outlined in section 4.5. “Random” employs a random order. “FCFS” adheres to a first-come-first-serve policy. “No interpreter help” omits the prefix hints from the interpreter, which is explained in detail below. Although RadixAttention can automatically detect shared prefixes, our runtime implementation is simplified: we only compare a pending request with the existing cache tree. We do not construct a tree index for all requests in the waiting queue. This approach may lead to suboptimal performance when there is a large batch of pending requests that share a prefix but have yet to be processed and inserted into the tree. To mitigate this, we use the interpreter to help with the scheduling. When the

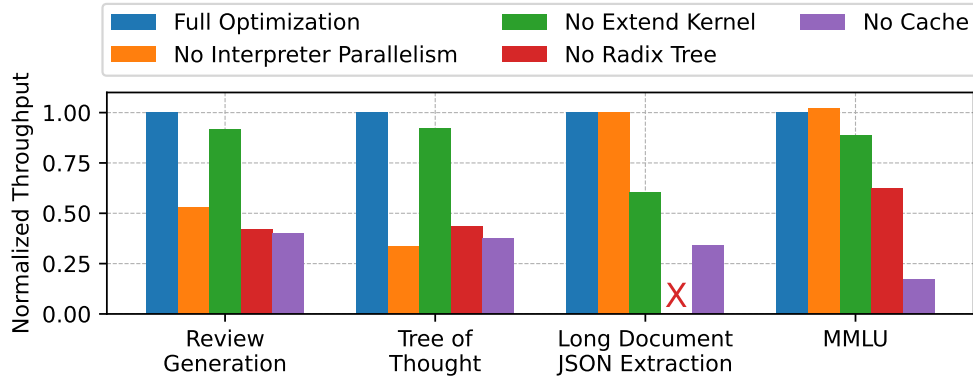


Figure 4.11: Optimization breakdown.

interpreter identifies a highly likely shared prefix (e.g., during a fork), it issues a forward request for this prefix first and stores it in the tree, and then it sends out the requests that can reuse the prefix. This method leverages hints from the interpreter to lessen the load on backend scheduling, underscoring the benefits of our co-design strategy. As depicted in Figure 4.10, “LPM” exhibits superior performance, while other policies struggle with a lower cache hit rate and reduced throughput. All policies perform similarly for simple tasks like MMLU.

Optimization Breakdown

We study the effectiveness of optimizations by disabling each one individually. Figure 4.11 illustrates the results. “No Extend Kernel” disables our optimized kernel, leading to degraded performance, particularly in long-context tasks. “No Interpreter Parallelism” turns off intra-program parallelism, resulting in reduced parallelism and cache locality. “No Cache” deactivates the cache, significantly deteriorating throughput across all benchmarks. “No Radix Tree” uses a list instead of a tree to manage the cache, leading to increased memory usage and copy overhead, and consequently, lower performance.

Improvements under Different Sharing Settings

We study how the improvement of prefix sharing is influenced by different parameters, including the lengths of shareable and unshareable inputs, and sharing structure configurations.

As shown in Figure 4.12, in the left subfigure, we establish a base setting with a shareable input length of 256, an unshareable input length of 256, and an output length of 256. We then gradually increase either the shareable input length or the unshareable length and observe the changes in throughput. The results demonstrate that the RadixAttention cache significantly enhances throughput compared to the “No Cache” scenario, with more notable improvements observed when the shareable length is longer.

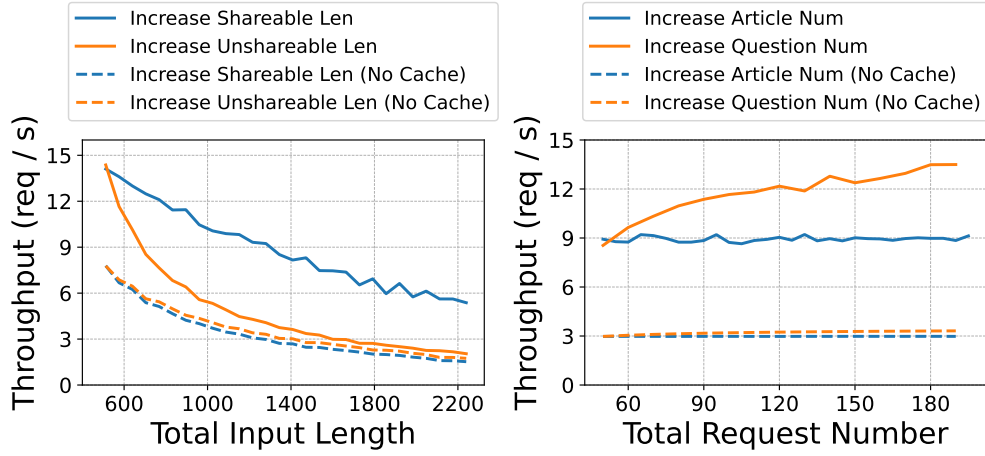


Figure 4.12: Improvements under Different Sharing Settings

In the right subfigure, we set a base scenario with 5 articles and 2 questions per article, where the article comprises 1000 tokens and each question 256 tokens. There are $5 \times 2 = 10$ requests in this base setting, with each article being shared by two questions. Subsequently, we increment either the number of articles or the number of questions per article. The findings indicate that the RadixAttention cache exhibits superior performance when the number of questions per article increases, as this creates more opportunities for sharing.

RadixAttention Overhead Analysis

We analyze the overhead of maintaining the radix tree at runtime. The tree structure is maintained on the CPU. In a typical scenario, which has a shareable input length of 256, an unshareable input length of 256, and 128 requests (organized into 16 subtrees, with 8 branches for each subtree), the forward pass in our setup takes 17.6 seconds. Meanwhile, the tree operation only takes 0.07 seconds. We also construct a request trace with no cache hits and observe the tree maintenance incurs only a 0.5% overhead. Therefore, the overhead is negligible. Furthermore, the overhead of tree maintenance can be further reduced by using a high-performance C++ implementation and by overlapping CPU and GPU operations.

4.7 Discussion and Limitations

Here are SGLang’s limitations and future development plans:

- Our interpreter runs arbitrary Python code, but the tracing and compiler lack support for data-dependent control flows. Introducing new control flow nodes to the SGLang IR, similar to TensorFlow’s dynamic flow [219], is required to address this.

- The current tokenization algorithm may create artifacts at prompt and generation boundaries. Implementing the token healing technique [65] could resolve this.
- We have not yet implemented grammar-constrained decoding. Efficient implementation would necessitate techniques from [205, 94] and careful batching.
- Plans include integrating more input and output modalities, such as incorporating text-to-image models into SGLang and exploring audio and video capabilities.

4.8 Related Work

Programming with Language Models The trend in programming LLM includes the development of better prompting techniques [202, 131, 214, 18], tools usage [161, 166, 144], agent frameworks [207, 171, 213, 169, 102, 140, 142], and programming systems [19, 65, 96, 87, 111, 205]. Among these, Guidance and LMQL are the most similar to our work. We discuss the differences between SGLang and existing LLM programming systems in section 4.2 and evaluate their performance in section 4.6. In short, the co-design approach of SGLang significantly enhances execution efficiency.

Inference Optimizations for LLMs. Common inference optimizations for LLMs include batching [218], memory optimizations [167, 95, 5], GPU kernel optimizations [43], model parallelism [147], parameter sharing [168], speculative execution [174, 100, 119], scheduling [206, 69], caching [172], quantization [208, 108, 45, 57], and sparsification [56]. These optimizations are complementary to our work and can be, or have already been, integrated into our backend.

Compilers for Machine Learning. Various domain-specific compilers for machine learning have been developed. These compilers are capable of optimizing a single tensor operator [34, 53, 226, 117, 181, 215, 193], a computational graph [85, 184], a distributed computation graph [225, 116], and classical algorithms [127]. Distinct from these more general-purpose or low-level compilers, SGLang is specifically designed for LLM applications and incorporates LLM-specific optimizations.

4.9 Conclusion

In this paper, we introduce SGLang, an efficient programming system for LLMs. SGLang greatly accelerates common LLM workloads by up to $5\times$ with reduced code complexity and improved control. Through the co-design of the language and runtime, we aim to present a more integrated approach to optimizing LLM programming paradigms.

Chapter 5

Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena

Evaluating large language model (LLM) based chat assistants is challenging due to their broad capabilities and the inadequacy of existing benchmarks in measuring human preferences. To address this, we explore using strong LLMs as judges to evaluate these models on more open-ended questions. We examine the usage and limitations of LLM-as-a-judge, including position, verbosity, and self-enhancement biases, as well as limited reasoning ability, and propose solutions to mitigate some of them. We then verify the agreement between LLM judges and human preferences by introducing two benchmarks: MT-bench, a multi-turn question set; and Chatbot Arena, a crowdsourced battle platform. Our results reveal that strong LLM judges like GPT-4 can match both controlled and crowdsourced human preferences well, achieving over 80% agreement, the same level of agreement between humans. Hence, LLM-as-a-judge is a scalable and explainable way to approximate human preferences, which are otherwise very expensive to obtain. Additionally, we show our benchmark and traditional benchmarks complement each other by evaluating several variants of LLaMA and Vicuna. The MT-bench questions, 3K expert votes, and 30K conversations with human preferences are publicly available at https://github.com/lm-sys/FastChat/tree/main/fastchat/llm_judge.

This chapter is based on my previous paper [228], which was a collaborative effort with Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, Ion Stoica. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

5.1 Introduction

There has been a proliferation of LLM-based chat assistants (chatbots) that leverage supervised instruction fine-tuning and reinforcement learning with human feedback (RLHF) to unlock new instruction following and conversational abilities [139, 15, 137, 37, 232, 209, 50]. Once aligned with humans, these chat models are strongly preferred by human users over the original, unaligned models on which they are built. However, the heightened user preference does not always correspond to improved scores on traditional LLM benchmarks – benchmarks like MMLU [72] and HELM [106] cannot effectively tell the difference between these aligned models and the base models. This phenomenon suggests that there is a fundamental discrepancy between user perceptions of the usefulness of chatbots and the criteria adopted by conventional benchmarks.

We argue that this discrepancy primarily arises due to existing evaluation that only measures LLMs’ core capability on a confined set of tasks (e.g., multi-choice knowledge or retrieval questions), without adequately assessing its alignment with human preference in open-ended tasks, such as the ability to accurately adhere to instructions in multi-turn dialogues. As a demonstration, we show conversation histories with two models on an MMLU question in Figure 5.1. The two models are LLaMA-13B [183], a pre-trained base model without fine-tuning, and Vicuna-13B, our fine-tuned model from LLaMA-13B on high-quality conversations (the training details are in section 5.8). Despite the base LLaMA models showing competitive performance on conventional benchmarks (Table 5.8), its answers to open-ended questions are often not preferred by humans. This misalignment of conventional benchmarks underscores the core problem driving this paper: *the need for a robust and scalable automated method to evaluate LLM alignment with human preferences*.

To study this, we introduce two benchmarks with human ratings as the primary evaluation metric: MT-bench and Chatbot Arena. MT-bench is a series of open-ended questions that evaluate a chatbot’s multi-turn conversational and instruction-following ability – two critical elements for human preference. MT-bench is also carefully constructed to differentiate chatbots based on their core capabilities, such as reasoning and math. In addition, we develop Chatbot Arena, a crowdsourced platform featuring anonymous battles between chatbots in real-world scenarios – Users engage in conversations with two chatbots at the same time and rate their responses based on personal preferences.

While human evaluation is the gold standard for assessing human preferences, it is exceptionally slow and costly. To automate the evaluation, we explore the use of state-of-the-art LLMs, such as GPT-4, as a surrogate for humans. Because these models are often trained with RLHF, they already exhibit strong human alignment. We call this approach “*LLM-as-a-judge*”. This approach has been tried in our earlier blog post [37] and other concurrent or follow-up work [26, 136, 50, 46, 232, 64, 145, 196, 36, 199]. However, there has not been a systematic study of this approach.

In this paper, we study the LLM-as-a-judge approach by comparing it to the gold standard of human evaluation. We examine several potential limitations of the LLM-as-a-judge approach including position bias, verbosity bias, self-enhancement bias, and limited reason-

Question: If the FED buys bonds in the secondary market

(A) the money supply will increase. (B) the money supply will decrease. (C) the money supply will not be affected. (D) the discount rate would be affected. Output the answer only.

Assistant A:

The Federal Reserve buys bonds in the secondary market to increase the money supply.

Assistant B:

(A) the money supply will increase.

Follow-up Question: How does it affect my daily life? Give 3 examples.

Assistant A:

The Federal Reserve buys bonds in the secondary market to increase the money supply. This affects your daily life by:

1. Increasing the money supply.
2. Increasing the money supply and lowering the discount rate.
3. Increasing the money supply and lowering the discount rate and increasing the money supply.

Assistant B:

When the Federal Reserve buys bonds in the secondary market, it can have several effects on the economy that can impact your daily life. Here are three examples:

1. Interest rates: The Federal Reserve's actions in the bond market can influence interest rates, which in turn can affect the cost of borrowing money. For example, if the Federal Reserve buys bonds and drives up demand for them, interest rates on things like mortgages, car loans, and credit cards may go up, making it more expensive to borrow money.
2. Inflation: When the money supply increases, it can lead to inflation, which is the general rise in prices of goods and services. This can impact your daily life by making it more expensive to buy things like groceries, gas, and other necessities.
3. Employment: The Federal Reserve's actions in the bond market can also have an impact on employment. For example, if the Federal Reserve's actions lead to economic growth and increased demand for goods and services, it can create more job opportunities and boost employment.

GPT-4 Judgment:

Assistant A provided an incorrect response to the user's question about how the Federal Reserve buying bonds in the secondary market affects daily life. The answer given is repetitive and lacks clear examples of how the action impacts daily life.

On the other hand, **Assistant B** provided a relevant and accurate response to the user's question about the Federal Reserve buying bonds. The answer includes three clear examples of how the action impacts daily life, such as interest rates, inflation, and employment.

Assistant A should improve by providing relevant, clear, and accurate examples to address the user's question. **Assistant B's** response is more helpful, relevant, and detailed.

Figure 5.1: Multi-turn dialogues between a user and two AI assistants—LLaMA-13B (Assistant A) and Vicuna-13B (Assistant B)—initiated by a question from the MMLU benchmark and a follow-up instruction. GPT-4 is then presented with the context to determine which assistant answers better.

ing ability. We show that some of the biases are minor or can be mitigated. Once addressed, our results from 3K controlled expert votes and 3K crowdsourced human votes in the wild verify that GPT-4 judge match human evaluations at an agreement rate exceeding 80%, achieving the same level of human-human agreement (§5.4, Table 5.4). Consequently, this suggests LLM-as-a-judge is a scalable method to swiftly evaluate human preference, serving as a promising alternative to traditional human evaluations.

This paper makes two contributions: (1) a systematic study of LLM-as-a-judge; and (2) human preference datasets with high-quality questions and diverse user interactions from MT-bench and Chatbot Arena. In addition, we argue for the adoption of a hybrid evalu-

ation framework for future LLM benchmarks: by combining the existing capability-based benchmarks and the new preference-based benchmarks with LLM-as-a-judge, one can swiftly and automatically evaluate both the core capabilities and human alignment of models. We publicly release 80 MT-bench questions, 3K expert votes, and 30K conversations with human preferences for future study.

Table 5.1: Sample multi-turn questions in MT-bench.

Category	Sample Questions	
Writing	1st Turn	Compose an engaging travel blog post about a recent trip to Hawaii, highlighting cultural experiences and must-see attractions.
	2nd Turn	Rewrite your previous response. Start every sentence with the letter A.
Math	1st Turn	Given that $f(x) = 4x^3 - 9x - 14$, find the value of $f(2)$.
	2nd Turn	Find x such that $f(x) = 0$.
Knowledge	1st Turn	Provide insights into the correlation between economic indicators such as GDP, inflation, and unemployment rates. Explain how fiscal and monetary policies ...
	2nd Turn	Now, explain them again like I'm five.

5.2 MT-Bench and Chatbot Arena

Motivation

With the recent advances of LLMs, LLM-based assistants start to exhibit artificial general intelligence across diverse tasks, from writing and chatting to coding [26, 137, 6, 173]. However, evaluating their broad capabilities also becomes more challenging. Despite the availability of numerous benchmarks for language models, they primarily focus on evaluating models on closed-ended questions with short responses. Given that these chat assistants can now precisely follow user instructions in multi-turn dialogues and answer open-ended questions in a zero-shot manner, current benchmarks are inadequate for assessing such capabilities. Existing benchmarks mostly fall into the following three categories.

- **Core-knowledge benchmarks**, including MMLU [72], HellaSwag [222], ARC [39], WinoGrande [159], HumanEval [29], GSM-8K [41], and AGIEval [231], evaluate the core capabilities of pre-trained LLMs using zero-shot and few-shot benchmark sets. They typically require LLMs to generate a short, specific answer to benchmark questions that can be automatically validated.
- **Instruction-following benchmarks**, such as Flan [114, 203], Self-instruct [200], NaturalInstructions [122], Super-NaturalInstructions [201], expand to slightly more open-ended questions and more diverse tasks and are used to evaluate LLMs after instruction fine-tuning.

- **Conversational benchmarks**, like CoQA [154], MMDialog [52] and OpenAssistant [92], are closest to our intended use cases. However, the diversity and complexity of their questions often fall short in challenging the capabilities of the latest chatbots.

While largely overlooked by existing LLM benchmarks, human preferences serve as a direct measure of a chatbot’s utility in open-ended, multi-turn human-AI interactions. To bridge this gap, we introduce two novel benchmarks expressly tailored to assess human preferences. Simultaneously, these benchmarks are designed to distinguish the core capabilities of state-of-the-art models.

MT-Bench

We create MT-bench, a benchmark consisting of 80 high-quality multi-turn questions. MT-bench is designed to test multi-turn conversation and instruction-following ability, covering common use cases and focusing on challenging questions to differentiate models. We identify 8 common categories of user prompts to guide its construction: writing, roleplay, extraction, reasoning, math, coding, knowledge I (STEM), and knowledge II (humanities/social science). For each category, we then manually designed 10 multi-turn questions. Table 5.1 lists several sample questions.

Chatbot Arena

Our second approach is Chatbot Arena, a crowdsourcing benchmark platform featuring anonymous battles. On this platform, users can interact with two anonymous models simultaneously, posing the same question to both. They vote for which model provides the preferred response, with the identities of the models disclosed post-voting. After running Chatbot Arena for one month, we have collected around 30K votes. Since the platform does not use pre-defined questions, it allows gathering a wide range of unrestricted use cases and votes in the wild, based on the diverse interests of users. A screenshot of the platform can be found at section 5.8.

5.3 LLM as a Judge

While our initial evaluations using MT-bench and Chatbot Arena rely on human ratings, collecting human preferences can be costly and laborious [200, 177, 139, 15, 48]. To overcome this, we aim to develop a more scalable and automated approach. Given that most questions in MT-bench and Chatbot Arena are open-ended without reference answers, devising a rule-based program to assess the outputs is extremely challenging. Traditional evaluation metrics based on the similarity between outputs and reference answers (e.g., ROUGE [107], BLEU [141]) are also ineffective for these questions.

As LLMs continue to improve, they show potential in replacing human annotators in many tasks [62, 77]. Specifically, we are interested in whether LLMs can effectively evaluate

the responses of chat assistants and match human preferences. Next, we discuss the use and limitations of LLM-as-a-judge.

Types of LLM-as-a-Judge

We propose 3 LLM-as-a-judge variations. They can be implemented independently or in combination:

- **Pairwise comparison.** An LLM judge is presented with a question and two answers, and tasked to determine which one is better or declare a tie. The prompt used is given in section 5.8 (Appendix).
- **Single answer grading.** Alternatively, an LLM judge is asked to directly assign a score to a single answer. The prompt used for this scenario is in section 5.8 (Appendix).
- **Reference-guided grading.** In certain cases, it may be beneficial to provide a reference solution if applicable. An example prompt we use for grading math problems is in section 5.8 (Appendix).

These methods have different pros and cons. For example, the pairwise comparison may lack scalability when the number of players increases, given that the number of possible pairs grows quadratically; single answer grading may be unable to discern subtle differences between specific pairs, and its results may become unstable, as absolute scores are likely to fluctuate more than relative pairwise results if the judge model changes.

Advantages of LLM-as-a-Judge

LLM-as-a-judge offers two key benefits: *scalability* and *explainability*. It reduces the need for human involvement, enabling scalable benchmarks and fast iterations. Additionally, LLM judges provide not only scores but also explanations, making their outputs interpretable, as shown in Figure 5.1.

Limitations of LLM-as-a-Judge

We identify certain biases and limitations of LLM judges. However, we will also present solutions later and show the agreement between LLM judges and humans is high despite these limitations.

Position bias is when an LLM exhibits a propensity to favor certain positions over others. This bias is not unique to our context and has been seen in human decision-making [21, 150] and other ML domains [91, 197].

section 5.8 (Appendix) shows an example of position bias. GPT-4 is tasked to evaluate two responses from GPT-3.5 and Vicuna-13B to an open-ended question. When GPT-3.5's answer is positioned first, GPT-4 considers GPT-3.5's answer more detailed and superior.

Table 5.2: Position bias of different LLM judges. Consistency is the percentage of cases where a judge gives consistent results when swapping the order of two assistants. “Biased toward first” is the percentage of cases when a judge favors the first answer. “Error” indicates wrong output formats. The two largest numbers in each column are in bold.

Judge	Prompt	Consistency	Biased toward first	Biased toward second	Error
Claude-v1	default	23.8%	75.0%	0.0%	1.2%
	rename	56.2%	11.2%	28.7%	3.8%
GPT-3.5	default	46.2%	50.0%	1.2%	2.5%
	rename	51.2%	38.8%	6.2%	3.8%
GPT-4	default	65.0%	30.0%	5.0%	0.0%
	rename	66.2%	28.7%	5.0%	0.0%

Table 5.3: Failure rate under “repetitive list” attack for different LLM judges on 23 answers.

Judge	Claude-v1	GPT-3.5	GPT-4
Failure rate	91.3%	91.3%	8.7%

Table 5.4: Judge failure rate on 10 math questions with different prompts. We test LLaMA-13B vs. Vicuna-13B and swap positions. A failure means when GPT-4 says an incorrect answer is correct.

	Default	CoT	Reference
Failure rate	14/20	6/20	3/20

However, upon switching the positions of the two responses, GPT-4’s judgement flips, favoring Vicuna’s answer.

To analyze the position bias, we construct two similar answers to each first-turn question in MT-bench by calling GPT-3.5 twice with a temperature of 0.7. We then try three LLMs with two different prompts: “default” is our default prompt in section 5.8 (Appendix). “rename” renames the assistants in our default prompt to see whether the bias is on positions or names. As in Table 5.2, we found all of them exhibit strong position bias. Most LLM judges favor the first position. Claude-v1 also shows a name bias which makes it favors “Assistant A”, as illustrated by the “rename” prompt. The position bias can be very significant. Only GPT-4 outputs consistent results in more than 60% of cases.

Note that this test is challenging because the answers are very similar and occasionally indistinguishable even to humans. We will show that position bias is less prominent in some cases in section 5.8. As for the origin of this bias, we suspect that it could be rooted in the training data or inherent to the left-to-right architecture of causal transformers, but leave a deeper study as future work.

Verbosity bias is when an LLM judge favors longer, verbose responses, even if they are not as clear, high-quality, or accurate as shorter alternatives.

To examine this bias, we design a “repetitive list” attack with model answers from MT-

bench. We first select 23 model answers from MT-bench that contain a numbered list. We then make them unnecessarily verbose by asking GPT-4 to rephrase the list without adding any new information and insert the rephrased new list to the beginning of the original list. For example, if the original response contains 5 items, then the new response will contain 10 items but the first 5 items are rephrased from the original 5 items. An example is shown in section 5.8 (Appendix). We define the attack is successful if an LLM judge thinks the new response is better than the old response. Table 5.3 shows the failure rate of LLM judges under this attack, demonstrating that all LLMs may be prone to verbosity bias though GPT-4 defends significantly better than others. As a calibration, we find LLM judges are able to correctly judge identical answers (i.e., they always return a tie for two identical answers) but cannot pass the more advanced “repetitive list” attack.

Self-enhancement bias. We adopt the term “self-enhancement bias” from social cognition literature [24] to describe the effect that LLM judges may favor the answers generated by themselves.

We examine this effect statistically. Figure 5.3(b) shows the win rate (w/o tie) of six models under different LLM judges and humans. Compared to humans, we do observe that some judges favor certain models. For example, GPT-4 favors itself with a 10% higher win rate; Claude-v1 favors itself with a 25% higher win rate. However, they also favor other models and GPT-3.5 does not favor itself. Due to limited data and small differences, our study cannot determine whether the models exhibit a self-enhancement bias. Conducting a controlled study is challenging because we cannot easily rephrase a response to fit the style of another model without changing the quality.

Limited capability in grading math and reasoning questions. LLMs are known to have limited math and reasoning capability [41], which results in its failure of grading such questions because they do not know the correct answers. However, what is more intriguing is that it also shows limitations in grading basic math problems which it is capable of solving. For instance, in Figure 5.8 (Appendix), we present an example of an elementary math question in which GPT-4 makes an incorrect judgment. It’s worth noting that although GPT-4 can solve the problem (when asked separately), it was misled by the provided answers, ultimately resulting in incorrect judgment. This pattern can also be seen in a reasoning question example in Figure 5.8 (Appendix). Both GPT-3.5 and Claude-v1 show a similar weakness. In section 5.3, we will introduce a reference-guided method to mitigate such issues.

Addressing limitations

We present a few methods to address position bias and the limited grading ability for math questions.

Swapping positions. The position bias can be addressed by simple solutions. A conservative approach is to call a judge twice by swapping the order of two answers and only declare a win when an answer is preferred in both orders. If the results are inconsistent after swapping, we can call it a tie. Another more aggressive approach is to assign positions ran-

domly, which can be effective at a large scale with the correct expectations. In the following experiments, we use the conservative one.

Few-shot judge. We assess whether few-shot examples can improve consistency in the position bias benchmark. We select three good judgment examples using MT-bench-like questions, GPT-3.5 and Vicuna for generating answers, and GPT-4 for generating judgments. The examples cover three cases: A is better, B is better, and tie. As shown in section 5.8 (Appendix), the few-shot judge can significantly increase the consistency of GPT-4 from 65.0% to 77.5%. However, high consistency may not imply high accuracy and we are not sure whether the few-shot examples will introduce new biases. Besides, the longer prompts make API calls $4\times$ more expensive. We use the zero-shot prompt by default in our following experiments but leave an additional study in section 5.8.

Chain-of-thought and reference-guided judge. In section 5.3, we have shown LLM’s limited capability in grading math and reasoning questions. We propose two simple methods to mitigate this issue: chain-of-thought judge and reference-guided judge. Chain-of-thought is a widely used technique to improve LLM’s reasoning capability [202]. We propose a similar technique to prompt an LLM judge to begin with answering the question independently and then start grading. Detailed prompt in section 5.8 (Appendix). However, even with the CoT prompt, we find that in many cases LLM makes exactly the same mistake as the given answers in its problem-solving process (See example in section 5.8 (Appendix), suggesting that LLM judge may still be misled by the context. Hence, we propose a reference-guided method, in which we first generate LLM judge’s answer independently, and then display it as a reference answer in the judge prompt. In Table 5.4, we see a significant improvement in failure rate (from 70% to 15%) over the default prompt.

Fine-tuning a judge model. We try fine-tuning a Vicuna-13B on arena data to act as a judge and show some promising preliminary results in section 5.8.

Multi-turn judge

In MT-bench, every question involves two turns to evaluate conversational abilities. Therefore, when comparing two assistants, it becomes necessary to present a total of two questions and four responses, complicating the prompt design. We explore two possible designs, (1) breaking the two turns into two prompts or (2) displaying complete conversations in a single prompt. Our finding is the former one can cause the LLM judge struggling to locate the assistant’s previous response precisely. We illustrate a case in section 5.8 (Appendix) where GPT-4 makes an inaccurate judgment due to a faulty reference. This suggests the necessity of displaying a complete conversation to enable the LLM judge to better grasp the context. We then consider the alternative design that presents two full conversations in a single prompt in which we ask the LLM judge to focus on the second question (section 5.8 (Appendix)). This approach has been found to significantly alleviate the aforementioned referencing issue.

5.4 Agreement Evaluation

We study the agreement between different LLM judges and humans on MT-bench and Chatbot Arena datasets. On MT-bench, we also study the agreement among humans. MT-bench represents a small-scale study with controlled human evaluation, while Chatbot Arena represents a larger-scale study with crowdsourced human evaluation in the wild.

Setup

MT-bench. We generate answers for all 80 questions with 6 models: GPT-4, GPT-3.5, Claude-V1, Vicuna-13B, Alpaca-13B [177], and LLaMA-13B [183]. We then use 2 kinds of judges: LLM judges and 58 expert-level human labelers. The labelers are mostly graduate students so they are considered experts and more skilled than average crowd workers. We let LLM judges evaluate all pairs and let each human evaluate at least 20 random multi-turn questions. This resulted in around 3K votes for all questions. The detailed data collection process is in section 5.8.

Chatbot Arena. We randomly sample 3K single-turn votes from 30K arena data, which covers models including GPT-4, GPT-3.5, Claude, Vicuna-7B/13B, Koala-13B [60], Alpaca-13B, LLaMA-13B, and Dolly-12B. We use two kinds of judges: LLM judges and collected crowd judges (2114 unique IPs).

Metrics. We define the *agreement* between two types of judges as the probability of randomly selected individuals (but not identical) of each type agreeing on a randomly selected question. See more explanation in section 5.8. *Average win rate* is the average of win rates against all other players. These metrics can be computed with or without including tie votes.

High agreement between GPT-4 and humans

We compute agreement on MT-bench data. In Table 5.5, GPT-4 with both pairwise comparison and single answer grading show very high agreements with human experts. The agreement under setup S2 (w/o tie) between GPT-4 and humans reaches 85%, which is even higher than the agreement among humans (81%). This means GPT-4’s judgments closely align with the majority of humans. We also show that GPT-4’s judgments may help humans make better judgments. During our data collection, when a human’s choice deviated from GPT-4, we presented GPT-4’s judgments to humans and ask if they are reasonable (details in section 5.8). Despite different views, humans deemed GPT-4’s judgments reasonable in 75% of cases and are even willing to change their choices in 34% of cases.

The data from Arena shows a similar trend, as illustrated by Table 5.6. Comparing GPT-4 and other LLM judges, we find they reach a similar non-tie agreement ratio between humans but the number of non-tied votes from GPT-4 is much larger. This means that GPT-4 is more affirmative and less suffered from position bias but other models also perform well when they give an affirmative answer.

In both tables, GPT-4 with single-answer grading matches both pairwise GPT-4 and human preferences very well. This means GPT-4 has a relatively stable internal rubric. Although it may sometimes perform slightly worse than pairwise comparison and give more tie votes, it is a more scalable method.

We then perform a breakdown analysis by computing agreement on different model pairs and categories. We only include non-tied votes. In Figure 5.2, we observe the agreement between GPT-4 and human progressively increases in line with the performance disparity of the model pairs (i.e., larger win rate difference), from 70% to nearly 100%. This suggests that GPT-4 aligns with humans better when significant performance differences exist between the models.

Table 5.5: Agreement between two types of judges on MT-bench. “G4-Pair” and “G4-Single” denote GPT-4 with pairwise comparison and single-answer grading respectively. The single-answer grading can be converted into pairwise comparison results for calculating the agreement. We report two setups: “S1” includes non-tie, tie, and inconsistent (due to position bias) votes and counts inconsistent as tie; “S2” only includes non-tie votes. The agreement between two random judges under each setup is denoted as “R=”. The top value in each cell is the agreement, and the bottom gray value is #votes.

Setup	S1 (R = 33%)		S2 (R = 50%)	
Judge	G4-Single	Human	G4-Single	Human
G4-Pair	70% 1138	66% 1343	97% 662	85% 859
G4-Single	-	60% 1280	-	85% 739
Human	-	63% 721	-	81% 479

(a) First Turn

Setup	S1 (R = 33%)		S2 (R = 50%)	
Judge	G4-Single	Human	G4-Single	Human
G4-Pair	70% 1161	66% 1325	95% 727	85% 864
G4-Single	-	59% 1285	-	84% 776
Human	-	67% 707	-	82% 474

(b) Second Turn

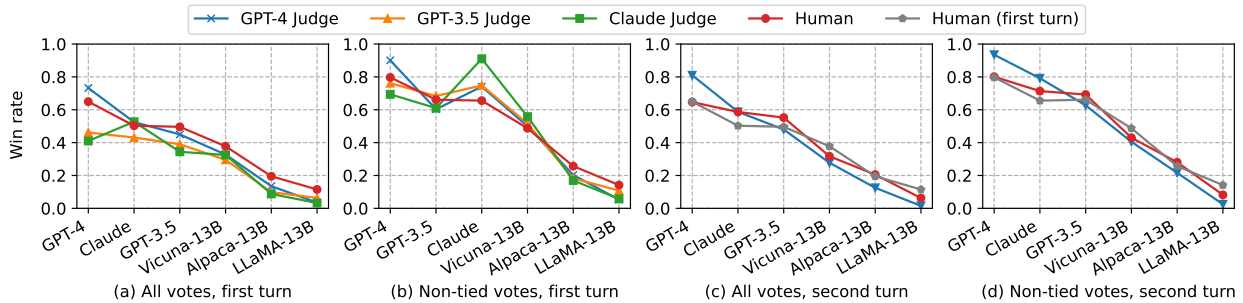


Figure 5.3: Average win rate of six models under different judges on MT-bench.

Table 5.6: Agreement between two types of judges on Chatbot Arena. “G4-S” denotes GPT-4 with single-answer grading. “G4”, “G3.5” and “C” denote GPT-4, GPT-3.5, and Claude with pairwise comparison, respectively. “H” denotes human. The remaining of table follows the same format as Chapter 5.5.

Setup	S1 (Random = 33%)				S2 (Random = 50%)			
Judge	G4-S	G3.5	C	H	G4-S	G3.5	C	H
G4	72% 2968	66% 3061	66% 3062	64% 3066	95% 1967	94% 1788	95% 1712	87% 1944
G4-S	-	60% 2964	62% 2964	60% 2968	-	89% 1593	91% 1538	85% 1761
G3.5	-	-	68% 3057	54% 3061	-	-	96% 1497	83% 1567
C	-	-	-	53% 3062	-	-	-	84% 1475

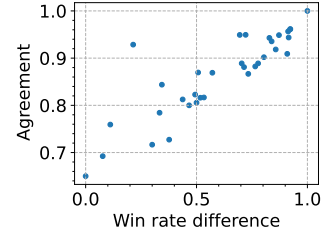


Figure 5.2: Agreement and win rate difference. Each point corresponds to a model pair and counts only the non-tie votes between the two models. The x-axis value is the win rate difference between the two models. The y-axis value is the GPT-4 and human agreement.

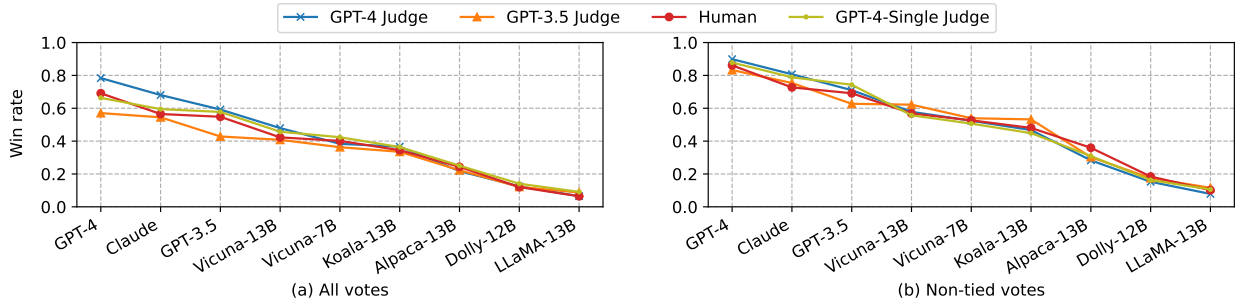


Figure 5.4: Average win rate of nine models under different judges on Chatbot Arena.

Table 5.7: Category-wise win rate of models.

Model	Writing	Roleplay	Reasoning	Math	Coding	Extraction	STEM	Humanities
GPT-4	61.2%	67.9%	49.3%	66.1%	56.3%	66.2%	76.6%	72.2%
GPT-3.5	50.9%	60.6%	32.6%	63.8%	55.0%	48.8%	52.8%	53.8%
Vicuna-13B	39.7%	39.2%	20.1%	18.0%	36.9%	29.2%	47.0%	47.5%
LLaMA-13B	15.1%	15.1%	7.8%	7.5%	2.1%	9.3%	6.8%	10.1%

Win rates under different judges

We plot the average win rate of models under different judges on MT-bench and Chatbot Arena in Figure 5.3 and Figure 5.4, respectively. The win rate curves from LLM judges closely

Table 5.8: Evaluation results of several model variants.

Model	#Training Token	MMLU (5-shot)	TruthfulQA (0-shot)	MT-Bench Score (GPT-4)
LLaMA-7B	1T	35.2	0.22	2.74
LLaMA-13B	1T	47.0	0.26	2.61
Alpaca-7B	4.4M	40.1	0.26	4.54
Alpaca-13B	4.4M	48.1	0.30	4.53
Vicuna-7B (selected)	4.8M	37.3	0.32	5.95
Vicuna-7B (single)	184M	44.1	0.30	6.04
Vicuna-7B (all)	370M	47.1	0.32	6.00
Vicuna-13B (all)	370M	52.1	0.35	6.39
GPT-3.5	-	70.0	-	7.94
GPT-4	-	86.4	-	8.99

match the curves from humans. On MT-bench second turn, proprietary models like Claude and GPT-3.5 are more preferred by the humans compared to the first turn, meaning that a multi-turn benchmark can better differentiate some advanced abilities of models. We also list the per-category win rate of representative models in Table 5.7 to show how MT-bench differentiates models, in which we see GPT-4 is significantly better than others. Vicuna-13B is noticeably worse than GPT-3.5/4 in reasoning, math, and coding categories. Note that in math/coding category, GPT-3.5 and GPT-4 have similar overall win-rate because they both failed to answer some hard questions, but GPT-4 is still significantly better than GPT-3 in the direct pairwise comparison or single-answer grading. Please see a performance breakdown of MT-bench score for each category in section 5.8.

5.5 Human Preference Benchmark and Standardized Benchmark

Human preference benchmarks such as MT-bench and Chatbot Arena serve as valuable additions to the current standardized LLM benchmarks. They focus on different aspects of a model and the recommended way is to comprehensively evaluate models with both kinds of benchmarks.

We evaluate several model variants derived from LLaMA on MMLU [72], Truthful QA [109] (MC1), and MT-bench (GPT-4 judge). The training details are in section 5.8. Since we have shown that GPT-4 single-answer grading also performs well in section 5.4, we use GPT-4 single-answer grading for MT-bench in favor of its scalability and simplicity. We ask GPT-4 to give a score for each turn on a scale of 10 by using our prompt templates (section 5.8, section 5.8) and report an average score of $160 = 80 \times 2$ turns. Table 5.8 shows the results. We find that fine-tuning on high-quality dialog datasets (i.e., ShareGPT) can consistently improve the model performance on MMLU and the improvement scales with fine-tuning data size. On the other hand, a small high-quality conversation dataset can quickly teach the model a style preferred by GPT-4 (or approximately human) but cannot improve MMLU

significantly, as shown by the Vicuna-7B (selected) which is trained with only 4.8M tokens or 3K conversations. In Table 5.8, no single benchmark can determine model quality, meaning that a comprehensive evaluation is needed. Our results indicate that using LLM-as-a-judge to approximate human preferences is highly feasible and could become a new standard in future benchmarks. We are also hosting a regularly updated leaderboard with more models¹. Notably, DynaBench [88], a research platform dedicated to dynamic data collection and benchmarking, aligns with our spirit. DynaBench addresses the challenges posed by static standardized benchmarks, such as saturation and overfitting, by emphasizing dynamic data with human-in-the-loop. Our LLM-as-a-judge approach can automate and scale platforms of this nature.

5.6 Discussion

Limitations. This paper emphasizes helpfulness but largely neglects safety. Honesty and harmlessness are crucial for a chat assistant as well [15]. We anticipate similar methods can be used to evaluate these metrics by modifying the default prompt. Additionally, within helpfulness, there are multiple dimensions like accuracy, relevance, and creativity, but they are all combined into a single metric in this study. A more comprehensive evaluation can be developed by analyzing and separating these dimensions. We propose preliminary solutions to address the limitations and biases of LLM-as-a-judge in section 5.3, but we anticipate more advanced methods can be developed.

Data collection and release. section 5.8 describes the detailed data collection and release processes, which include the instructions we give to users, the screenshots of the data collection interface, the information about participated users, and the content of the released data.

Societal impacts. The societal impact of this study is multi-faceted. Our evaluation methods can help enhance chatbot quality and user experiences. However, addressing biases in these methods is crucial. Our dataset enables better studies of human preferences and model behavior. Advanced chat assistants may replace certain human tasks, resulting in job displacements and new opportunities.

Future directions. 1) Benchmarking chatbots at scale with a broader set of categories 2) Open-source LLM judge aligned with human preference 3) Enhancing open models' math/reasoning capability.

5.7 Conclusion

In this paper, we propose LLM-as-a-judge for chatbot evaluation and systematically examine its efficacy using human preference data from 58 experts on MT-bench, as well as thousands of crowd-users on Chatbot Arena. Our results reveal that strong LLMs can achieve an

¹<https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard>

agreement rate of over 80%, on par with the level of agreement among human experts, establishing a foundation for an LLM-based evaluation framework.

5.8 Notes on the Appendix

For further information, including prompt templates, experimental details, figures, and additional training details, please see the appendix section of the original paper [228]. Readers are encouraged to consult the original paper as well for a better-formatted version of the content.

Chapter 6

Conclusion

6.1 Review

Large deep learning models, particularly large language models (LLMs), have revolutionized various fields due to their remarkable capabilities. However, their sheer size and complexity introduce unique challenges in designing systems to support their entire lifecycle, encompassing training, serving, and evaluation.

This dissertation presents a suite of open-source systems that collectively address these challenges and enable the scalable and efficient development of large models. Alpa (chapter 2) automates model-parallel training by generating efficient execution plans that integrate both inter- and intra-operator parallelism. Ansor (chapter 3) tackles the issue of performance optimization by generating high-performance tensor programs specifically tailored for various hardware backends. SGLang (chapter 4) offers a user-friendly programming system for LLMs, providing a high-performance runtime and optimizations like batching, caching, and parallelism. Finally, to address the challenge of model evaluation, the crowdsourced platform Chatbot Arena and automated pipeline LLM-as-a-Judge (chapter 5) are introduced.

6.2 Impact

The projects introduced in this dissertation have garnered significant recognition, accumulating over 2,500 citations in a short period during my Ph.D. studies. Their impact extends beyond academia, with widespread adoption in industry as well. Alpa has been integrated into the Google XLA compiler and the Jax deep learning framework, empowering researchers to train novel model architectures at scale. We also collaborated with NVIDIA to leverage Alpa’s capabilities, training a 175-billion parameter model on the Selene supercomputer. Ansor has been integrated into Apache TVM, a leading deep learning compiler framework. Additionally, SGLang has been deployed for practical use at Chatbot Arena, demonstrating cost reductions in serving large language models and vision language models. Chatbot Arena itself has been a resounding success. Operational for nearly a year, it has engaged millions of

users through crowdsourcing, collected over 10 million conversations, and received 1 million votes, becoming the industry standard for evaluating large language models as well as a valuable data source for understanding human-model interaction. Notably, Chatbot Arena, SGLang, and LLM-as-a-Judge were all developed under the LMSYS.org organization, a multi-university open research collaboration I co-founded to advance open large models. We also released the Vicuna model and FastChat system at LMSYS.org.

The success of these projects can be attributed to several factors: their timely and elegant solutions, open development approach, and focus on addressing significant, relevant problems. These systems have all been released as open-source projects, fostering collaboration and accelerating research in the field of large models. The open-source nature has fostered a vibrant community, with over 40,000 stars on GitHub and hundreds of active contributors.

6.3 Future Directions

Programming systems for large generative models. As LLMs become central to more complex and diverse applications, there is a pressing need to develop robust programming frameworks that can manage, chain, and orchestrate multiple model calls within a single workflow. This development involves creating programming systems that ensure the reliability and correctness of individual model interactions, potentially through the use of type systems and formal verification tools. Such frameworks would facilitate the construction of complex workflows, enabling seamless integration of different computational tasks and model interactions. To support these sophisticated systems, it is crucial to develop high-performance runtimes and compilers that can efficiently execute and optimize these workflows. Future research will explore how to compile these workflows dynamically, adjusting to the needs of different applications and ensuring optimal performance across various computational environments.

Further scaling of training with synthetic data generation. The continuous scaling of deep learning models necessitates an increasing volume of training data, a need that may soon outstrip the availability of human-generated text data. To address this impending data bottleneck, it is essential to explore methods for synthetic data generation, enabling models to train on artificially created, yet informative datasets. Leveraging techniques from this dissertation, such as efficient inference systems developed in SGLang, could facilitate the high-throughput generation and filtering of high-quality training examples through complex tree search algorithms. These algorithms would require multiple inference steps to sift and refine data, ensuring only high-quality examples are used for training. Additionally, integrating automatic feedback mechanisms, such as those explored in the LLM-as-a-Judge system, could provide a continuous loop of performance evaluation and improvement, enhancing the overall quality of the synthetic data generated.

Evaluation platform for human-AI collaboration. Current evaluation methods for LLMs need to evolve to keep pace with the models' growing capabilities and the complex scenarios in which they are deployed. Future research will aim to expand evaluation platforms like Chatbot Arena to create more immersive and interactive environments that can better simulate real-world applications. This might include gamifying the evaluation process to encourage more engagement from participants and to collect more nuanced data from multi-turn interactions. Further development of human-AI collaboration tools can provide deeper insights into how humans and AI can work together more effectively, exploring new forms of interaction that leverage the unique strengths of both. Such platforms would not only serve as benchmarks for model performance but also as incubators for new forms of AI-assisted human communication and decision-making.

Bibliography

- [1] Martín Abadi et al. “Tensorflow: a system for large-scale machine learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.
- [2] Andrew Adams et al. “Learning to optimize halide with tree search and random programs”. In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–12.
- [3] Jean-Baptiste Alayrac et al. “Flamingo: a visual language model for few-shot learning”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 23716–23736.
- [4] Hassan Abu Alhaija et al. “Augmented reality meets deep learning for car instance segmentation in urban scenes”. In: *British machine vision conference*. Vol. 1. 2017, p. 2.
- [5] Reza Yazdani Aminabadi 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*. IEEE. 2022, pp. 1–15.
- [6] Rohan Anil et al. “Palm 2 technical report”. In: *arXiv preprint arXiv:2305.10403* (2023).
- [7] Jason Ansel et al. “OpenTuner: an extensible framework for program autotuning”. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.
- [8] Sanjith Athlur et al. “Varuna: scalable, low-cost training of massive deep learning models”. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 472–487.
- [9] AWS Cluster Configuratoins. <https://aws.amazon.com/ec2/instance-types/p3/>.
- [10] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. “Distributed Balanced Partitioning via Linear Embedding”. In: *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 2016, pp. 387–396.
- [11] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

- [12] Riyadh Baghdadi et al. “Tiramisu: a polyhedral compiler for expressing fast and portable code”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 193–205.
- [13] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: open neural network exchange*. 2019.
- [14] Youhui Bai et al. “Gradient Compression Supercharged High-Performance Data Parallel DNN Training”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 2021, pp. 359–375.
- [15] Yuntao Bai et al. “Training a helpful and harmless assistant with reinforcement learning from human feedback”. In: *arXiv preprint arXiv:2204.05862* (2022).
- [16] Paul Barham and Michael Isard. “Machine learning systems are stuck in a rut”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 177–183.
- [17] Paul Barham et al. “Pathways: Asynchronous distributed dataflow for ML”. In: *Proceedings of Machine Learning and Systems 4* (2022).
- [18] Maciej Besta et al. “Graph of thoughts: Solving elaborate problems with large language models”. In: *arXiv preprint arXiv:2308.09687* (2023).
- [19] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. “Prompting is programming: A query language for large language models”. In: *Proceedings of the ACM on Programming Languages 7*.PLDI (2023), pp. 1946–1969.
- [20] L Susan Blackford et al. *ScaLAPACK users’ guide*. SIAM, 1997.
- [21] Niels J Blunch. “Position bias in multiple-choice questions”. In: *Journal of Marketing Research 21.2* (1984), pp. 216–220.
- [22] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008, pp. 101–113.
- [23] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: <http://github.com/google/jax>.
- [24] Jonathon D Brown. “Evaluations of self and others: Self-enhancement biases in social judgments”. In: *Social cognition 4.4* (1986), pp. 353–376.
- [25] Tom B Brown et al. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020).
- [26] Sébastien Bubeck et al. “Sparks of artificial general intelligence: Early experiments with gpt-4”. In: *arXiv preprint arXiv:2303.12712* (2023).
- [27] Zhenkun Cai et al. “TensorOpt: Exploring the tradeoffs in distributed DNN training with auto-parallelism”. In: *IEEE Transactions on Parallel and Distributed Systems 33.8* (2021), pp. 1967–1981.
- [28] Jianfei Chen et al. “ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training”. In: *International Conference on Machine Learning*. 2021.

- [29] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [30] Tianqi Chen and Carlos Guestrin. “Xgboost: a scalable tree boosting system”. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.
- [31] Tianqi Chen et al. “Learning to optimize tensor programs”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 3389–3400.
- [32] Tianqi Chen et al. “Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).
- [33] Tianqi Chen et al. “Training deep nets with sublinear memory cost”. In: *arXiv preprint arXiv:1604.06174* (2016).
- [34] Tianqi Chen et al. “TVM: an automated end-to-end optimizing compiler for deep learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.
- [35] Sharan Chetlur et al. “cudnn: efficient primitives for deep learning”. In: *arXiv preprint arXiv:1410.0759* (2014).
- [36] Cheng-Han Chiang and Hung-yi Lee. “Can Large Language Models Be an Alternative to Human Evaluations?” In: *arXiv preprint arXiv:2305.01937* (2023).
- [37] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: <https://lmsys.org/blog/2023-03-30-vicuna/>.
- [38] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *arXiv preprint arXiv:2204.02311* (2022).
- [39] Peter Clark et al. “Think you have solved question answering? try arc, the ai2 reasoning challenge”. In: *arXiv preprint arXiv:1803.05457* (2018).
- [40] Karl Cobbe et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).
- [41] Karl Cobbe et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).
- [42] Marius Cordts et al. “The cityscapes dataset for semantic urban scene understanding”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3213–3223.
- [43] Tri Dao et al. “Flashattention: Fast and memory-efficient exact attention with io-awareness”. In: *Advances in Neural Information Processing Systems* 35 (2022).
- [44] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. “Distributed halide”. In: *ACM SIGPLAN Notices* 51.8 (2016), pp. 1–12.

- [45] Tim Dettmers et al. “Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 30318–30332.
- [46] Tim Dettmers et al. “QLoRA: Efficient Finetuning of Quantized LLMs”. In: *arXiv preprint arXiv:2305.14314* (2023).
- [47] Jacob Devlin et al. “BERT: pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).
- [48] Shizhe Diao et al. “Lmflow: An extensible toolkit for finetuning and inference of large foundation models”. In: *arXiv preprint arXiv:2306.12420* (2023).
- [49] Nan Du et al. *GLaM: Efficient Scaling of Language Models with Mixture-of-Experts*. 2021. arXiv: 2112.06905 [cs.CL].
- [50] Yann Dubois et al. “AlpacaFarm: A Simulation Framework for Methods that Learn from Human Feedback”. In: *arXiv preprint arXiv:2305.14387* (2023).
- [51] Shiqing Fan et al. “DAPPLE: A pipelined data parallel approach for training large models”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 431–445.
- [52] Jiazhan Feng et al. “MMDialog: A Large-scale Multi-turn Dialogue Dataset Towards Multi-modal Open-domain Conversation”. In: *arXiv preprint arXiv:2211.05719* (2022).
- [53] Siyuan Feng et al. “Tensorir: An abstraction for automatic tensorized program optimization”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 804–817.
- [54] John Forrest and Robin Lougee-Heimer. “CBC user guide”. In: *Emerging theory, methods, and applications*. INFORMS, 2005, pp. 257–277.
- [55] Richard J Forrester and Noah Hunt-Isaak. “Computational comparison of exact solution methods for 0-1 quadratic programs: Recommendations for practitioners”. In: *Journal of Applied Mathematics* 2020 (2020).
- [56] Elias Frantar and Dan Alistarh. “Sparsegpt: Massive language models can be accurately pruned in one-shot”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 10323–10337.
- [57] Elias Frantar et al. “OPTQ: Accurate quantization for generative pre-trained transformers”. In: *The Eleventh International Conference on Learning Representations*. 2022.
- [58] Matteo Frigo and Steven G Johnson. “FFTW: an adaptive software architecture for the FFT”. In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP’98 (Cat. No. 98CH36181)*. Vol. 3. IEEE. 1998, pp. 1381–1384.

- [59] Trevor Gale, Erich Elsen, and Sara Hooker. “The state of sparsity in deep neural networks”. In: *arXiv preprint arXiv:1902.09574* (2019).
- [60] Xinyang Geng et al. *Koala: A Dialogue Model for Academic Research*. Blog post. Apr. 2023. URL: <https://bair.berkeley.edu/blog/2023/04/03/koala/> (visited on 04/03/2023).
- [61] Georgi Gerganov. *Llama.cpp*. <https://github.com/ggerganov/llama.cpp>. Accessed: 2023-11.
- [62] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. “Chatgpt outperforms crowd-workers for text-annotation tasks”. In: *arXiv preprint arXiv:2303.15056* (2023).
- [63] Google Cloud TPU Cluster Configurations. <https://cloud.google.com/tpu>.
- [64] Arnav Gudibande et al. “The False Promise of Imitating Proprietary LLMs”. In: *arXiv preprint arXiv:2305.15717* (2023).
- [65] Guidance AI. *A Guidance Language for Controlling Large Language Models*. <https://github.com/guidance-ai/guidance>. Accessed: 2023-11.
- [66] Ameer Haj-Ali et al. “AutoPhase: juggling HLS phase orderings in random forests with deep reinforcement learning”. In: *Third Conference on Machine Learning and Systems (ML-Sys)*. 2020.
- [67] Ameer Haj-Ali et al. “NeuroVectorizer: end-to-end vectorization with deep reinforcement learning”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 242–255.
- [68] Ameer Haj-Ali et al. “ProTuner: tuning programs with monte carlo tree search”. In: *arXiv preprint arXiv:2005.13685* (2020).
- [69] Mingcong Han et al. “Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 539–558.
- [70] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. “Can spatiotemporal 3D CNNs retrace the history of 2D CNNs and ImageNet?”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 6546–6555.
- [71] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [72] Dan Hendrycks et al. “Measuring massive multitask language understanding”. In: *arXiv preprint arXiv:2009.03300* (2020).
- [73] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. “Matrix capsules with EM routing”. In: (2018).
- [74] Andrew G Howard et al. “Mobilenets: efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

- [75] Yuwei Hu et al. “FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems”. In: *arXiv preprint arXiv:2008.11359* (2020).
- [76] Chien-Chin Huang, Gu Jin, and Jinyang Li. “Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1341–1355.
- [77] Fan Huang, Haewoon Kwak, and Jisun An. “Is chatgpt better than human annotators? potential and limitations of chatgpt in explaining implicit hate speech”. In: *arXiv preprint arXiv:2302.07736* (2023).
- [78] Qijing Huang et al. “Autophase: compiler phase-ordering for hls with deep reinforcement learning”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.
- [79] Yanping Huang et al. “Gpipe: Efficient training of giant neural networks using pipeline parallelism”. In: *Advances in neural information processing systems* 32 (2019), pp. 103–112.
- [80] Hugging Face. *Text Generation Inference*. <https://github.com/huggingface/text-generation-inference>. Accessed: 2023-11.
- [81] Intel. *Intel® math kernel library for deep learning networks*. 2017. URL: <https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation>.
- [82] Sergey Ioffe and Christian Szegedy. “Batch normalization: accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015).
- [83] Paras Jain et al. “Checkmate: Breaking the memory wall with optimal tensor materialization”. In: *arXiv preprint arXiv:1910.02653* (2019).
- [84] Zhihao Jia, Matei Zaharia, and Alex Aiken. “Beyond data and model parallelism for deep neural networks”. In: *arXiv preprint arXiv:1807.05358* (2018).
- [85] Zhihao Jia et al. “TASO: optimizing deep learning computation with automatic generation of graph substitutions”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 47–62.
- [86] Yimin Jiang et al. “A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 463–479.
- [87] Omar Khattab et al. “DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines”. In: *arXiv preprint arXiv:2310.03714* (2023).
- [88] Douwe Kiela et al. “Dynabench: Rethinking Benchmarking in NLP”. In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2021, pp. 4110–4124.

- [89] Soojeong Kim et al. “Parallax: Sparsity-aware data parallel training of deep neural networks”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–15.
- [90] Marisa Kirisame et al. “Dynamic Tensor Rematerialization”. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=Vfs_2RnOD0H.
- [91] Miyoung Ko et al. “Look at the first sentence: Position bias in question answering”. In: *arXiv preprint arXiv:2004.14602* (2020).
- [92] Andreas Köpf et al. “OpenAssistant Conversations—Democratizing Large Language Model Alignment”. In: *arXiv preprint arXiv:2304.07327* (2023).
- [93] Alex Krizhevsky. “One weird trick for parallelizing convolutional neural networks”. In: *arXiv preprint arXiv:1404.5997* (2014).
- [94] Michael Kuchnik, Virginia Smith, and George Amvrosiadis. “Validating large language models with relm”. In: *Proceedings of Machine Learning and Systems* 5 (2023).
- [95] Woosuk Kwon et al. “Efficient memory management for large language model serving with pagedattention”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 611–626.
- [96] LangChain AI. *LangChain*. <https://github.com/langchain-ai/langchain>. Accessed: 2023-11.
- [97] Andrew Lavin and Scott Gray. “Fast algorithms for convolutional neural networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4013–4021.
- [98] Woo-Yeon Lee et al. “Automating system configuration of distributed machine learning”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 2057–2067.
- [99] Dmitry Lepikhin et al. “Gshard: Scaling giant models with conditional computation and automatic sharding”. In: *arXiv preprint arXiv:2006.16668* (2020).
- [100] Yaniv Leviathan, Matan Kalman, and Yossi Matias. “Fast inference from transformers via speculative decoding”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 19274–19286.
- [101] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.
- [102] Guohao Li et al. “CAMEL: Communicative Agents for ”Mind” Exploration of Large Language Model Society”. In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.
- [103] Shen Li et al. “Pytorch distributed: Experiences on accelerating data parallel training”. In: *arXiv preprint arXiv:2006.15704* (2020).

- [104] Tzu-Mao Li et al. “Differentiable programming for image processing and deep learning in Halide”. In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), p. 139.
- [105] Zhuohan Li et al. “TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models”. In: *arXiv preprint arXiv:2102.07988* (2021).
- [106] Percy Liang et al. “Holistic evaluation of language models”. In: *arXiv preprint* (2022).
- [107] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. In: *Text summarization branches out*. 2004, pp. 74–81.
- [108] Ji Lin et al. “AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration”. In: *arXiv preprint arXiv:2306.00978* (2023).
- [109] Stephanie Lin, Jacob Hilton, and Owain Evans. “Truthfulqa: Measuring how models mimic human falsehoods”. In: *arXiv preprint arXiv:2109.07958* (2021).
- [110] Haotian Liu et al. “Improved baselines with visual instruction tuning”. In: *arXiv preprint arXiv:2310.03744* (2023).
- [111] Jerry Liu. *LlamaIndex*. Nov. 2022. DOI: 10.5281/zenodo.1234. URL: https://github.com/jerryjliu/llama_index.
- [112] Xiaoxia Liu et al. “Prompting Frameworks for Large Language Models: A Survey”. In: *arXiv preprint arXiv:2311.12785* (2023).
- [113] Yizhi Liu et al. “Optimizing CNN model inference on CPUs”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 1025–1040.
- [114] Shayne Longpre et al. “The flan collection: Designing data and methods for effective instruction tuning”. In: *arXiv preprint arXiv:2301.13688* (2023).
- [115] Moshe Looks et al. “Deep learning with dynamic computation graphs”. In: *arXiv preprint arXiv:1702.02181* (2017).
- [116] Wenyan Lu et al. “Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2017, pp. 553–564.
- [117] Lingxiao Ma et al. “Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 881–897.
- [118] Mark F. Medress et al. “Speech understanding systems: report of a steering committee”. In: *Artificial Intelligence* 9.3 (1977), pp. 307–316.
- [119] Xupeng Miao et al. “SpecInfer: Accelerating Generative LLM Serving with Speculative Inference and Token Tree Verification”. In: *arXiv preprint arXiv:2305.09781* (2023).
- [120] Paulius Micikevicius et al. “Mixed precision training”. In: *arXiv preprint* (2017).
- [121] Azalia Mirhoseini et al. “Device placement optimization with reinforcement learning”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2430–2439.

- [122] Swaroop Mishra et al. “Cross-task generalization via natural language crowdsourcing instructions”. In: *ACL*. 2022.
- [123] Model TC. *LightLLM*. <https://github.com/ModelTC/lightllm>. Accessed: 2023-11.
- [124] Thierry Moreau et al. “A hardware–software blueprint for flexible deep learning specialization”. In: *IEEE Micro* 39.5 (2019), pp. 8–16.
- [125] Philipp Moritz et al. “Ray: A distributed framework for emerging AI applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.
- [126] Ravi Teja Mullapudi et al. “Automatically scheduling halide image processing pipelines”. In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), p. 83.
- [127] Supun Nakandala et al. “A tensor compiler for unified machine learning prediction serving”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 899–917.
- [128] Deepak Narayanan et al. “Efficient large-scale language model training on GPU clusters using megatron-LM”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.
- [129] Deepak Narayanan et al. “Memory-efficient pipeline-parallel dnn training”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 7937–7947.
- [130] Deepak Narayanan et al. “PipeDream: generalized pipeline parallelism for DNN training”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 1–15.
- [131] Xuefei Ning et al. “Skeleton-of-thought: Large language models can do parallel decoding”. In: *arXiv preprint arXiv:2307.15337* (2023).
- [132] NVIDIA. *TensorRT-LLM*. <https://github.com/NVIDIA/TensorRT-LLM>. Accessed: 2023-11.
- [133] NVIDIA. *The NVIDIA Collective Communication Library*. 2018. URL: <https://developer.nvidia.com/nccl>.
- [134] Nvidia. *NVIDIA tensor cores*. 2017. URL: <https://www.nvidia.com/en-us/data-center/tensorcore/>.
- [135] Nvidia. *NVIDIA TensorRT: programmable inference accelerator*. 2017. URL: <https://developer.nvidia.com/tensorrt>.
- [136] OpenAI. *Evals is a framework for evaluating LLMs and LLM systems, and an open-source registry of benchmarks*. <https://github.com/openai/evals>.
- [137] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].
- [138] OpenAI. *OpenAI API Reference*. <https://platform.openai.com/docs/api-reference>. Accessed: 2023-11.

- [139] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 27730–27744.
- [140] Charles Packer et al. *MemGPT: Towards LLMs as Operating Systems*. 2023. arXiv: 2310.08560 [cs.AI].
- [141] Kishore Papineni et al. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.
- [142] Joon Sung Park et al. “Generative Agents: Interactive Simulacra of Human Behavior”. In: *In the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. UIST '23. San Francisco, CA, USA: Association for Computing Machinery, 2023.
- [143] Adam Paszke et al. “PyTorch: an imperative style, high-performance deep learning library”. In: *Advances in Neural Information Processing Systems*. 2019, pp. 8024–8035.
- [144] Shishir G. Patil et al. “Gorilla: Large Language Model Connected with Massive APIs”. In: *arXiv preprint arXiv:2305.15334* (2023).
- [145] Baolin Peng et al. “Instruction tuning with gpt-4”. In: *arXiv preprint arXiv:2304.03277* (2023).
- [146] Yanghua Peng et al. “A generic communication scheduler for distributed dnn training acceleration”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 16–29.
- [147] Reiner Pope et al. “Efficiently scaling transformer inference”. In: *Proceedings of Machine Learning and Systems* 5 (2023).
- [148] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised representation learning with deep convolutional generative adversarial networks”. In: *arXiv preprint* (2015).
- [149] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.
- [150] Priya Raghuram and Ana Valenzuela. “Center-of-inattention: Position biases in decision-making”. In: *Organizational Behavior and Human Decision Processes* 99.1 (2006), pp. 66–80.
- [151] Samyam Rajbhandari et al. “Zero: Memory optimizations toward training trillion parameter models”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–16.

- [152] Jeff Rasley et al. “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 3505–3506.
- [153] Nir Ratner et al. “Parallel context windows for large language models”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2023, pp. 6383–6402.
- [154] Siva Reddy, Danqi Chen, and Christopher D Manning. “Coqa: A conversational question answering challenge”. In: *Transactions of the Association for Computational Linguistics* 7 (2019), pp. 249–266.
- [155] Jie Ren et al. “Zero-offload: Democratizing billion-scale model training”. In: *arXiv preprint arXiv:2101.06840* (2021).
- [156] Jared Roesch et al. “Relay: a high-level compiler for deep learning”. In: *arXiv preprint arXiv:1904.08368* (2019).
- [157] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).
- [158] Swarnadeep Saha et al. “Branch-Solve-Merge Improves Large Language Model Evaluation and Generation”. In: *arXiv preprint arXiv:2310.15123* (2023).
- [159] Keisuke Sakaguchi et al. “Winogrande: An adversarial winograd schema challenge at scale”. In: *Communications of the ACM* 64.9 (2021), pp. 99–106.
- [160] Mark Sandler et al. “Mobilenetv2: inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.
- [161] Timo Schick et al. “Toolformer: Language models can teach themselves to use tools”. In: *arXiv preprint arXiv:2302.04761* (2023).
- [162] Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stochastic superoptimization”. In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 305–316.
- [163] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *arXiv preprint arXiv:1802.05799* (2018).
- [164] Noam Shazeer et al. “Mesh-tensorflow: Deep learning for supercomputers”. In: *arXiv preprint arXiv:1811.02084* (2018).
- [165] Haichen Shen et al. “Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference”. In: *arXiv preprint arXiv:2006.03031* (2020).
- [166] Yongliang Shen et al. “Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface”. In: *arXiv preprint arXiv:2303.17580* (2023).
- [167] Ying Sheng et al. “FlexGen: high-throughput generative inference of large language models with a single GPU”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31094–31116.

- [168] Ying Sheng et al. “S-LoRA: Serving Thousands of Concurrent LoRA Adapters”. In: *arXiv preprint arXiv:2311.03285* (2023).
- [169] Noah Shinn et al. “Reflexion: Language agents with verbal reinforcement learning”. In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023.
- [170] Mohammad Shoeybi et al. “Megatron-lm: Training multi-billion parameter language models using model parallelism”. In: *arXiv preprint arXiv:1909.08053* (2019).
- [171] Significant Gravitas. *AutoGPT*. URL: <https://github.com/Significant-Gravitas/AutoGPT>.
- [172] Xiaoni Song et al. “UGACHE: A Unified GPU Cache for Embedding-based Deep Learning”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 627–641.
- [173] Aarohi Srivastava et al. “Beyond the imitation game: Quantifying and extrapolating the capabilities of language models”. In: *arXiv preprint arXiv:2206.04615* (2022).
- [174] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. “Blockwise parallel decoding for deep autoregressive models”. In: *Advances in Neural Information Processing Systems* 31 (2018).
- [175] Patricia Suriana, Andrew Adams, and Shoaib Kamil. “Parallel associative reductions in halide”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 281–291.
- [176] Richard S Sutton and Andrew G Barto. *Reinforcement learning: an introduction*. MIT press, 2018.
- [177] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.
- [178] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. “Piper: Multidimensional Planner for DNN Parallelization”. In: *Advances in Neural Information Processing Systems* 34 (2021).
- [179] Google XLA Team. *XLA: Optimizing Compiler for Machine Learning*. 2017. URL: <https://www.tensorflow.org/xla>.
- [180] The Theano Development Team et al. “Theano: A Python framework for fast computation of mathematical expressions”. In: *arXiv preprint arXiv:1605.02688* (2016).
- [181] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. “Triton: an intermediate language and compiler for tiled neural network computations”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019, pp. 10–19.
- [182] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

- [183] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).
- [184] Colin Unger et al. “Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 267–284.
- [185] Nicolas Vasilache et al. “Tensor comprehensions: framework-agnostic high-performance machine learning abstractions”. In: *arXiv preprint arXiv:1802.04730* (2018).
- [186] Nicolas Vasilache et al. “The next 700 accelerated layers: from mathematical expressions of network computation graphs to accelerated GPU kernels, automatically”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16.4 (2019), pp. 1–26.
- [187] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).
- [188] Sven Verdoolaege. “Presburger formulas and polyhedral compilation”. In: (2016).
- [189] Sven Verdoolaege et al. “Polyhedral parallel code generation for CUDA”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), pp. 1–23.
- [190] Pradnya A Vikhar. “Evolutionary algorithms: a critical review and its future prospects”. In: *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*. IEEE. 2016, pp. 261–265.
- [191] Thijs Vogels, Sai Praneeth Karinireddy, and Martin Jaggi. “PowerSGD: Practical low-rank gradient compression for distributed optimization”. In: *Advances In Neural Information Processing Systems 32 (Nips 2019)* 32.CONF (2019).
- [192] Guanzhi Wang et al. “Voyager: An Open-Ended Embodied Agent with Large Language Models”. In: *arXiv preprint arXiv: Arxiv-2305.16291* (2023).
- [193] Haojie Wang et al. “PET: Optimizing tensor programs with partially equivalent transformations and automated corrections”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 37–54.
- [194] Leyuan Wang et al. “A unified optimization approach for CNN model inference on integrated GPUs”. In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.
- [195] Minjie Wang, Chien-chin Huang, and Jinyang Li. “Supporting very large models using automatic dataflow graph partitioning”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–17.
- [196] Peiyi Wang et al. “Large language models are not fair evaluators”. In: *arXiv preprint arXiv:2305.17926* (2023).
- [197] Xuanhui Wang et al. “Position bias estimation for unbiased learning to rank in personal search”. In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 2018, pp. 610–618.

- [198] Xuezhi Wang et al. “Self-Consistency Improves Chain of Thought Reasoning in Language Models”. In: *The Eleventh International Conference on Learning Representations*. 2022.
- [199] Yizhong Wang et al. “How Far Can Camels Go? Exploring the State of Instruction Tuning on Open Resources”. In: *arXiv preprint arXiv:2306.04751* (2023).
- [200] Yizhong Wang et al. *Self-Instruct: Aligning Language Model with Self Generated Instructions*. 2022.
- [201] Yizhong Wang et al. “Super-NaturalInstructions: Generalization via Declarative Instructions on 1600+ Tasks”. In: *EMNLP*. 2022.
- [202] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 24824–24837.
- [203] Jason Wei et al. “Finetuned language models are zero-shot learners”. In: *arXiv preprint arXiv:2109.01652* (2021).
- [204] R Clinton Whaley and Jack J Dongarra. “Automatically tuned linear algebra software”. In: *SC’98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE. 1998, pp. 38–38.
- [205] Brandon T Willard and Rémi Louf. “Efficient guided generation for large language models”. In: *arXiv e-prints* (2023), arXiv–2307.
- [206] Bingyang Wu et al. “Fast Distributed Inference Serving for Large Language Models”. In: *arXiv preprint arXiv:2305.05920* (2023).
- [207] Qingyun Wu et al. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework”. In: 2023. arXiv: 2308.08155 [cs.AI].
- [208] Guangxuan Xiao et al. “Smoothquant: Accurate and efficient post-training quantization for large language models”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 38087–38099.
- [209] Can Xu et al. “Wizardlm: Empowering large language models to follow complex instructions”. In: *arXiv preprint arXiv:2304.12244* (2023).
- [210] Yuanzhong Xu et al. “Automatic cross-replica sharding of weight update in data-parallel training”. In: *arXiv preprint arXiv:2004.13336* (2020).
- [211] Yuanzhong Xu et al. “GSPMD: General and Scalable Parallelization for ML Computation Graphs”. In: *arXiv preprint arXiv:2105.04663* (2021).
- [212] Zhilin Yang et al. “HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering”. In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2018.
- [213] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *The Eleventh International Conference on Learning Representations*. 2022.

- [214] Shunyu Yao et al. “Tree of thoughts: Deliberate problem solving with large language models”. In: *arXiv preprint arXiv:2305.10601* (2023).
- [215] Zihao Ye et al. “SparseTIR: Composable abstractions for sparse compilation in deep learning”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, pp. 660–678.
- [216] Ori Yoran et al. “Answering questions by meta-reasoning over multiple chains of thought”. In: *arXiv preprint arXiv:2304.13007* (2023).
- [217] Fisher Yu and Vladlen Koltun. “Multi-scale context aggregation by dilated convolutions”. In: *arXiv preprint arXiv:1511.07122* (2015).
- [218] Gyeong-In Yu et al. “Orca: A distributed serving system for {Transformer-Based} generative models”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 521–538.
- [219] Yuan Yu et al. “Dynamic control flow in large-scale machine learning”. In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–15.
- [220] Jinhui Yuan et al. “OneFlow: Redesign the Distributed Deep Learning Framework from Scratch”. In: *arXiv preprint arXiv:2110.15032* (2021).
- [221] Sergey Zagoruyko and Nikos Komodakis. “Wide residual networks”. In: *arXiv preprint arXiv:1605.07146* (2016).
- [222] Rowan Zellers et al. “HellaSwag: Can a Machine Really Finish Your Sentence?” In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 4791–4800.
- [223] Hao Zhang et al. “AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning”. In: *Advances in Neural Information Processing Systems 33* (2020).
- [224] Zhen Zhang et al. “MiCS: Near-linear Scaling for Training Gigantic Model on Public Cloud”. In: *arXiv preprint arXiv:2205.00119* (2022).
- [225] Lianmin Zheng et al. “Alpa: Automating inter-and intra-operator parallelism for distributed deep learning”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 559–578.
- [226] Lianmin Zheng et al. “Ansor: Generating high-performance tensor programs for deep learning”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 863–879.
- [227] Lianmin Zheng et al. “Efficiently programming large language models using SGLang”. In: *arXiv preprint arXiv:2312.07104* (2023).
- [228] Lianmin Zheng et al. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In: *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2023. URL: <https://openreview.net/forum?id=uccHPGDlao>.

- [229] Size Zheng et al. “FlexTensor: an automatic schedule exploration and optimization framework for tensor computation on heterogeneous system”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 859–873.
- [230] Zhen Zheng et al. “FusionStitching: boosting memory intensive computations for deep learning workloads”. In: *arXiv preprint arXiv:2009.10924* (2020).
- [231] Wanjun Zhong et al. “Agieval: A human-centric benchmark for evaluating foundation models”. In: *arXiv preprint arXiv:2304.06364* (2023).
- [232] Chunting Zhou et al. “Lima: Less is more for alignment”. In: *arXiv preprint* (2023).