Piper: Towards Flexible Pipeline Parallelism for PyTorch

Megan Frisella, Arvin Oentoro, Xiangyu Gao, Gilbert Bernstein, Stephanie Wang
University of Washington

Abstract

With the rise of increasingly large-scale ML systems, distributed execution across multiple devices is critical for efficient training and inference. Current ML systems have adopted pipeline-parallel distributed execution strategies to improve resource efficiency, but lack generality in the models and execution schedules they support. We designed and prototyped Piper, a compiler that automates pipeline-parallel execution for arbitrary PyTorch models and custom execution schedules. We present preliminary performance results that compare with existing state-of-the-art pipeline parallelism frameworks pipelining Llama and CLIP models, while aiming to support a wider range of PyTorch models and pipeline-parallel execution schedules.

ACM Reference Format:
Megan Frisella, Arvin Oentoro, Xiangyu Gao, Gilbert Bernstein, Stephanie Wang . 2025. Piper: Towards Flexible Pipeline Parallelism for PyTorch. In *Practical Adoption Challenges of ML for Systems (PACMI '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3766882.3767187

1 Introduction

Distributed execution is a key factor enabling efficient training and inference in today's ML systems. Strategies for distributing ML models are becoming increasingly complex. 4D parallelism, with pipeline, data, tensor, and context parallel dimensions, is popular for training and has many varieties in each dimension [3, 5–7, 15]. Pipeline parallelism [3] enables training large models by splitting them into stages across devices (Figure 2) and maximizing hardware utilization by concurrently processing different microbatches of data on different devices. However it is not frequently adopted due to its implementation complexity. There are two main challenges to implementing pipeline parallelism.

First, pipeline parallelism is difficult to adopt compared to other parallelism strategies because there are many possible *execution schedules*. Execution schedules define where and when forward and backward computations execute. For example, GPipe [3] schedules all the forward stages then all the backward stages. 1F1B [9] schedules the backward stages for each microbatch as soon as possible, requiring less memory to store intermediate activations. Research exploring execution schedules that balance tradeoffs between throughput, latency and memory remains important as new models with new execution characteristics emerge [2, 3, 9, 11]. However, implementing any one schedule takes significant care to get right; the timing and ordering of operations must be synchronized

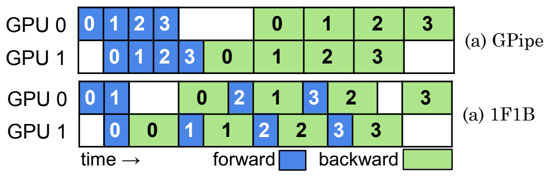


Figure 1: Two different pipeline-parallel execution schedules for a 2-stage model. Numbers identify data microbatches.

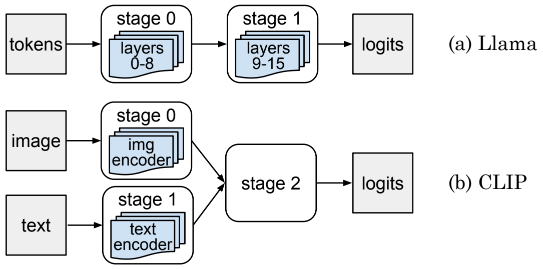


Figure 2: Pipeline stage dependencies for Llama and CLIP. Llama has linear dependencies while CLIP has a dependency DAG.

to respect the temporal dependencies of the execution schedule and the data dependencies of the model pipeline without introducing unnecessary pipeline bubbles (white space in Figure 1).

Second, pipeline parallelism is difficult to apply automatically to *arbitrary PyTorch models* because it is not always possible to statically determine the flow of data through the model. For example, it is impossible to analyze the behavior of data-dependent operations (e.g., data-dependent control flow) before running a model. Automatically partitioning a model requires understanding the model's data flow in order to identify the dependencies between pipeline stages and route data through the stages accordingly. This is different from other parallelism techniques that are not concerned with partitioning the overall data flow (e.g., data parallelism replicates the whole model across devices and tensor parallelism splits individual operations across devices).

Existing state-of-the-art frameworks have adopted pipeline parallelism but fail to achieve generality in the models and execution schedules they support. Megatron-LM [15] offers high-performance pipeline parallelism for a select set of transformer models and only two execution schedules. Pipeline Parallelism for PyTorch [6, 14] (PiPPy), automatically pipelines PyTorch models that have static computation graphs (excluding popular architectures like MoE), and requires additional user effort to support other models. PiPPy requires significant user effort to encode custom execution schedules. Current frameworks also do not support models with DAG pipeline dependencies, like CLIP [12] in Figure 2b.

We seek to build a framework that automatically pipelines arbitrary PyTorch models, gives the user full control over the execution schedule without the burden and error-proneness of low-level coordination, and achieves performance parity with equivalent execution strategies on state-of-the-art frameworks. To achieve these goals, we design and prototype Piper, a compiler that translates

This work is licensed under a Creative Commons Attribution 4.0 International License.
PACMI '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2205-9/25/10
https://doi.org/10.1145/3766882.3767187

PyTorch models into functionally equivalent distributed programs. We use a combination of just-in-time compilation and eager (interpreted) execution to support models whose data flow cannot be fully analyzed ahead of time. We use a centralized coordinator to enable flexible scheduling that can be modified with low user effort. However, centralization introduces synchronization points and limits how much the CPU can run ahead of the GPU on distributed devices. We apply optimizations to communication operations to mitigate these overheads. Thus, Piper:

- Automatically coordinates user-defined pipeline-parallel execution schedules that are easy to specify, and guarantees that user-defined schedules are valid respect the pipeline's data dependencies.
- Partitions arbitrary PyTorch models according to user annotations.
- Achieves competitive preliminary performance with state-of-the-art frameworks pipelining the Llama and CLIP models.

2 Related Work

2.1 Pipeline-Parallel Execution Schedules

There has been extensive research into developing pipeline-parallel execution schedules that balance throughput, latency, and memory for different model architectures. GPipe [3] processes forward and backward computations sequentially, incurring a large pipeline bubble. 1F1B [9] reduces intermediate activation lifetime compared to GPipe by scheduling backward computations right after forward computations. Interleaved 1F1B [10] interleaves the execution of multiple "virtual" stages to increase utilization by overlapping multiple forward-backward streams. ZeroBubble [11] improves 1F1B by decoupling backward logic into two phases to fill in pipeline bubbles. DualPipe [2] duplicates each layer so that it can feed microbatches into both ends of the pipeline and overlaps communication with computation to hide the all-to-all operations of expert parallelism in MoE transformer models.

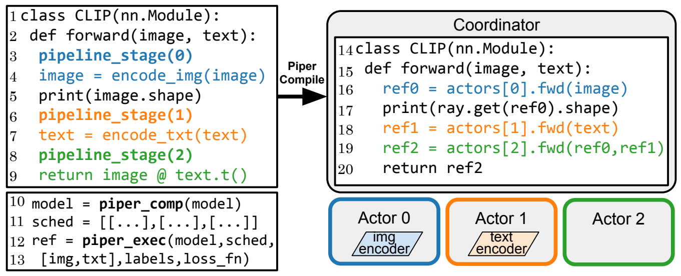
2.2 Training Frameworks

Megatron-LM The Megatron-LM [15] framework for LLM training hard-codes pipeline parallel support for a select set of models and offers two execution schedules (1F1B [9] and interleaved 1F1B [10]), trading off generality for high performance. Megatron-LM does not support many popular non-transformer models like StableDiffusion or multi-modal models like CLIP.

DeepSpeed The DeepSpeed [13] Python library for distributed training and inference supports PyTorch models that satisfy DeepSpeed's pipeline API: the model must be expressed as a list of layers such that the forward pass is equivalent to a composition of each layer's forward pass. DeepSpeed provides a 1F1B schedule but does not support custom schedules.

2.3 Pipeline Parallelism Libraries

PiPPy Pipeline Parallelism for PyTorch [6, 14] (PiPPy) automatically partitions PyTorch models that have static computation graphs.



```
1  class CLIP(nn.Module):
2   def forward(image, text):
3    pipeline_stage(0)
4    image = encode_img(image)
5    print(image.shape)
6    pipeline_stage(1)
7    text = encode_txt(text)
8    pipeline_stage(2)
9    return image @ text.t()
```

```
10 model = piper_comp(model)
11 sched = [[...],[...],[...]]
12 ref = piper_exec(model,sched,
13  [img,txt],labels,loss_fn)
```

```
14 class CLIP(nn.Module):
15  def forward(image, text):
16    ref0 = actors[0].fwd(image)
17    print(ray.get(ref0).shape)
18    ref1 = actors[1].fwd(text)
19    ref2 = actors[2].fwd(ref0,ref1)
20    return ref2
```

Figure 3: Piper workflow. User inputs are on the left and the compiled distributed program is on the right.

Models whose data flow cannot be analyzed ahead of time (e.g., models with data-dependent control flow, external library calls, or other untraceable operations) must be manually partitioned by the user. This precludes a large number of PyTorch models, including MoE architectures, StableDiffusion and multi-modal models (see Table 2). To use a custom execution schedule, the user must manually coordinate communication operations to respect the schedule's temporal dependencies and the pipeline's data dependencies.

JaxPP JaxPP [16] automatically partitions models according to pipeline stage annotations and coordinates user-defined execution schedules. JaxPP's support is limited to models written in the Jax framework, which requires computations to be expressed as pure, statically analyzable functions.

Pfeife Pfeife [4] provides automatic pipeline parallel support for PyTorch models by intercepting the computation graphs generated by TorchDynamo, similar to our system. Pfeife performs automatic search over model partitions and schedule parameters, limited to interleaved 1F1B schedules. Pfeife also interoperates with data parallelism. Pfeife supports models with static computation graphs, and does not support all models with non-static computation graphs.

3 Design and Implementation

Piper translates PyTorch models into distributed programs and automatically executes them according to user-defined pipeline-parallel execution schedules.

3.1 API

Figure 3 presents the user's workflow pipelining CLIP (Contrastive Language-Image Pre-training) [12], a multi-modal model that uses image and language encoders to connect images and text with contrastive learning. We split CLIP into three stages so that the stage dependencies form a DAG like Figure 2b. To achieve the 3-stage pipeline, the user adds `pipeline_stage` annotations on lines 3, 6, and 8 of Fig. 3. The user calls `piper_comp` (line 10) to compile the annotated model and produce the distributed program on the right side of Figure 3.

To define a custom execution schedule (line 11), the user builds a 2-D array that resembles an execution schedule diagram, like the diagrams in Figure 6. The entries in the schedule array must satisfy (1) one row per stage and one column per time step, (2) `None` entries represent pipeline bubbles, and (3) all other entries are tuples with the stage ID, microbatch index, and type (forward or backward) of a computation. To execute one iteration of the schedule, the user calls `piper_exec` with the compiled model, schedule, inputs, labels, and loss function (line 12). The result is a *future*, which is a reference that will eventually point to the result of a training iteration (training loss per microbatch). The result is distributed, so it remains on the actors.

3.2 Compiler

Our compiler, `piper_comp`, extends TorchDynamo [1], a PyTorch JIT compiler that works by tracing a model's forward function to extract computation graphs of PyTorch operations. TorchDynamo encodes computation graphs as `torch.fx` graphs and compiles them into optimized CPU or GPU kernels. "Graph breaks" occur when TorchDynamo hits an untraceable operation (e.g., data-dependent control flow, external library calls, or print statements). To handle this, TorchDynamo uses a continuation-passing style to pause execution when it reaches an untraceable operation, compile the `torch.fx` graph accumulated up to that point, replace the original code with a call into the compiled kernel, and then resume execution at the untraceable operation. The continuation runs the untraceable operation in eager-mode and then continues tracing the code that follows to produce the next `torch.fx` graph. This design enables flexible graph capture that does not require the entire model to be traceable.

Piper intercepts each `torch.fx` graph before TorchDynamo compiles it and instead sends it to be compiled and executed on a remote device. Piper also automatically sends the model parameters used by the computation graph to the same remote device on the first execution, to avoid repeatedly moving model parameters between the controller and the workers. `pipeline_stage` annotations force graph breaks in TorchDynamo so that each pipeline stage has a non-overlapping sequence of computation graphs. Piper keeps track of which graphs belong to which stage to track the data flow of the pipeline, for later use in `piper_exec`.

We use the Ray [8] distributed runtime as an RPC layer to manage distributed actors. The process running Piper becomes the distributed coordinator. Piper executes `torch.fx` graphs as tasks on remote actors by replacing the graph's original code with an RPC that runs the graph remotely (lines 16, 18 and 19 in Figure 3). Ray tasks return distributed futures, which eventually point to a result that remains on the actors. By operating on distributed futures, the coordinator never materializes data from a remote device unless the user reads the data (e.g., `ray.get` materializes the reference on line 17 because the user code called `print`). Thus, futures enable the coordinator to run ahead of the distributed actors by dispatching tasks asynchronously until there is a synchronization point on the coordinator that fetches data.

TorchDynamo uses continuation-passing style to execute untraceable operations, like the print statement on line 5 of Figure 3, eagerly. In Piper, these eager-mode operations that are not captured in a `torch.fx` graph are run on the coordinator (line 17). This ensures that the code run on distributed devices consists of static tensor operations, while the coordinator maintains a consistent view of potentially dynamic or stateful untraceable Python code.

TorchDynamo's continuation-passing style always runs a model's `torch.fx` graphs in succession. Naively applying this in Piper can result in incorrect executions for schedules that interleave forwards and backwards across different microbatches. To achieve interleaved execution, Piper modifies the continuation function call sites produced by TorchDynamo to return the arguments for the next continuation instead of directly calling it, allowing Piper to schedule the continuations for each stage separately.

Framework	Model	1F1B	GPipe
Piper	5	29	14
PiPPy	13	70	51

Table 1: Lines of Llama code changed to achieve a 2-stage partition and lines of code written to achieve 1F1B and GPipe execution schedules.

Model	PiPPy	Megatron-LM
Llama3	✓	✓
ViT	✓	✓
Sparsely-gated MoE	✗	✓
StableDiffusion	✗	✗
CLIP	✗	✗

Table 2: Models supported by PiPPy and Megatron-LM. Piper aims to support all of these models.

3.3 Distributed Execution

During compilation, Piper records the data flow between pipeline stages based on the traced `torch.fx` graphs. In `piper_exec`, Piper uses the data flow to route microbatches forward and backward through the pipeline in order to execute a distributed training step. Piper orders forward and backward microbatches according to a user-defined schedule. Before executing a schedule, Piper checks that it obeys the data dependencies of the model pipeline (e.g., Llama stage 1 happens after stage 0, backward for stage 1 happens after forward for stage 1, and backward for stage 0 happens after backward for stage 1). Then, for each timestep, Piper dispatches one task per pipeline stage and microbatch according to the schedule. Each task dispatch triggers a peer-to-peer communication between the actors corresponding to the current and previous pipeline stages.

`piper_exec` also automatically handles gradient accumulation and weight updates. Actors store a copy of the input and output activations per microbatch and propagate the gradients of intermediate activations backward through the pipeline in the backward pass. Each backward stage produces a partial gradient that must be accumulated over all the microbatches. After the last microbatch completes its final backward stage, Piper dispatches a weight update task on every actor to apply the accumulated partial gradients to the model weights on each device.

By automatically coordinating communication operations between forward and backward stages across devices, Piper frees the user from writing distributed coordination code. To achieve a different schedule, the user only needs to change the high-level schedule array. This makes it easy for users to experiment with different execution schedules in order to balance tradeoffs for their model's specific execution characteristics. Piper currently has experimental support for virtual stages, necessary for the interleaved 1F1B schedule. We plan to add support for finer-grained scheduling and overlapping communication with computation to support schedules like ZeroBubble and DualPipe.

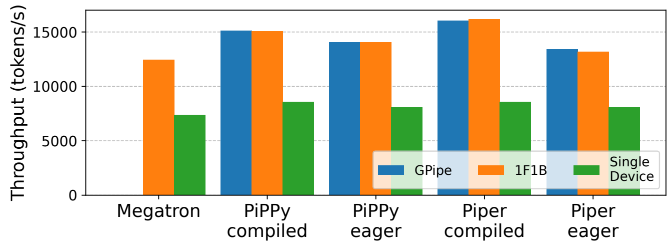


Figure 4: Training throughput for Llama-3B with GPipe and 1F1B execution schedules.

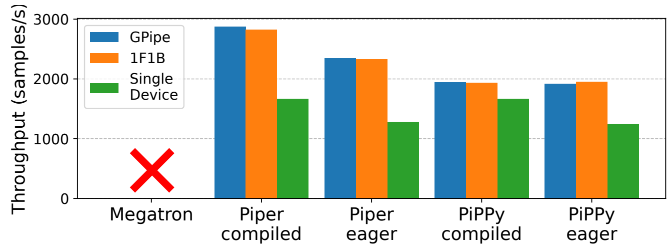
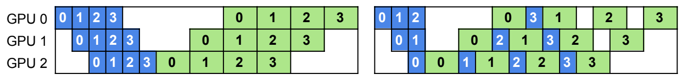
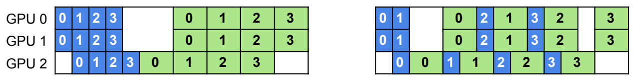


Figure 5: Training throughput for CLIP with GPipe- and 1F1B-like execution schedules.



(a) Linear stage dependencies.



(b) DAG stage dependencies.

Figure 6: GPipe- (left) and 1F1B-like (right) schedules for CLIP expressed with (a) linear stage dependencies and (b) DAG stage dependencies.

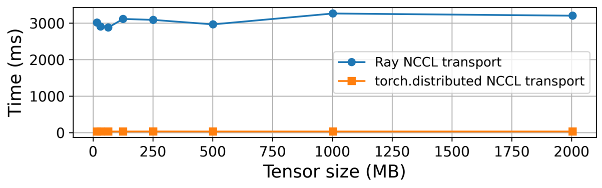
3.4 Performance Optimizations

The Piper coordinator is responsible for directing activations and gradients between distributed actors. Sending tensors through the coordinator would increase communication latency compared to sending tensors directly between distributed actors. Thus, we use Ray futures to transfer tensors directly between distributed actors using NVIDIA's Collective Communication Library (NCCL), and only materialize data on the coordinator if necessary (e.g., line 17 in Figure 3).

4 Evaluation

We compare Piper with two state-of-the-art pipeline parallelism frameworks, PiPPy and Megatron-LM, on two popular models, Llama and CLIP.

Distribution Effort We divide Llama-3B into 2 stages (Figure 2a). Table 1 describes the user effort required to pipeline Llama with Piper and PiPPy. Megatron-LM is not included because it does not support custom models. To achieve a 2-stage partition in Piper, we re-write the loop over transformer layers into two loops that each run half of the transformer layers and we add a `pipeline_stage` annotation between the loops. PiPPy's manual mode requires slightly more modifications to refactor the model in order to optionally load and run layers depending on the stage.



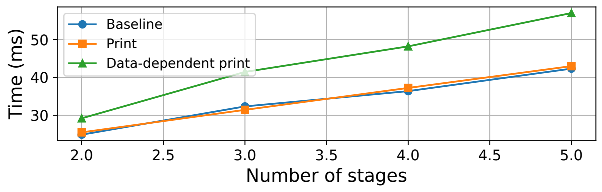


Figure 7: (Top) GPU-GPU tensor transfer time in Ray vs. PyTorch's distributed communication package. (Bottom) Running different types of untraceable code on the coordinator between stages.

To achieve 1F1B and GPipe execution schedules, Piper requires writing 29 and 14 lines of code to produce methods, parameterized over the number of microbatches and stages, that build 2-D schedule arrays for 1F1B and GPipe, respectively. Achieving the same schedules with PiPPy requires using `torch.distributed` operations to synchronize forward and backward stages according to the schedule's temporal dependencies and the pipeline's data dependencies. This requires more code and significant care to ensure that distributed operations are synchronized correctly.

Training Throughput Piper and PiPPy "compiled" run compiled PyTorch code using the TorchInductor [1] backend for TorchDynamo. Figure 4 presents the Llama throughput results. Piper has slightly lower throughput than PiPPy in eager mode because Ray NCCL tensor transfers incur a 3ms overhead compared to PyTorch's default NCCL tensor transfers, due to the Ray coordinator dynamically dispatching send and receive RPCs to facilitate the transfer. Piper achieves slightly better throughput than PiPPy in compiled mode, indicating that TorchDynamo interoperates favorably with Piper compared to PiPPy. Piper achieves better throughput than Megatron-LM without Megatron's other parallelism dimensions enabled (DP, TP, CP).

We evaluate CLIP training throughput using Piper and PiPPy. Megatron-LM does not support any non-transformer architectures, so we do not include it. We partition CLIP into 3 stages (Figure 2b) and measure training throughput using the GPipe- and 1F1B-like schedules in Figure 6. Piper exploits CLIP's dependency DAG by scheduling stages 0 and 1 in parallel (Figure 6b). This reduces pipeline bubbles compared to PiPPy, which can only express linear stage dependencies (Figure 6a). Figure 5 presents throughput results. Piper achieves better throughput than PiPPy by reducing pipeline bubbles. Compilation via TorchInductor further improves Piper's throughput compared to PiPPy.

Scalability We evaluate the scalability of our centralized coordinator distributed runtime. In Figure 7 (top) we measure GPU-GPU tensor transfer time (using the NCCL backend) in Ray vs. PyTorch's native distributed communication package. Ray tensor transfer adds roughly 3ms of overhead from dynamic dispatch. However, this overhead remains constant with increasing tensor size. In future work, we plan to explore fusing Ray RPCs ahead of time to reduce this overhead.

In Figure 7 (bottom) we evaluate the cost of running untraceable code on the coordinator between stages. We measure forward iteration time with (i) no computation, (ii) printing "hello world" and (iii) printing the first value of the previous stage's activation on the coordinator between stages. We observe linear overhead from the data-dependent operation due to materializing activations on the coordinator between stages. The data-independent print does not add measurable overhead because the operation is fast enough to not delay the coordinator in dispatching the next stage's RPC. In future work, we plan to further evaluate how these overheads translate to real-world PyTorch models.

Supported Models We evaluate whether PiPPy and Megatron-LM support a set of popular PyTorch models and present our findings in Table 2. Megatron-LM only has native support for transformer models. It does not support popular models like StableDiffusion or multi-modal models like CLIP. PiPPy only supports models with linear pipeline dependencies, which precludes CLIP. PiPPy automatically supports traceable models, precluding architectures like MoE (non-static computation graph) and StableDiffusion (external library calls). It would require additional user effort to pipeline these models with PiPPy. Piper aims to support all the models in Table 2 and more.

5 Discussion

Piper decouples model and schedule specification from distributed implementation by using `torch.fx` subgraphs as an intermediate representation (IR) for PyTorch models. PyTorch and the Piper scheduling interface allow for flexible model and schedule specification, respectively. The TorchDynamo compiler translates these into a series of `torch.fx` subgraphs. Finally, Piper distributes the subgraphs using the Ray distributed runtime. We think that Piper's design will enable further improvements to usability and flexibility of pipeline-parallel support by enabling improvements to the components above (compiler) and below (distributed runtime) the IR independently of each other. Decoupling also enables portability to other backends and frontends, e.g., alternative frontends that produce `torch.fx` graphs or distributed runtimes other than Ray.

We plan to extend Piper to interoperate with additional parallelism dimensions, including tensor, data and expert parallelism. We believe the centralized coordinator design will adapt well to new parallelism dimensions. For example, to support data parallelism, we imagine replicating the Piper coordinator n times for n-way DP, and adding all-to-all gradient synchronization steps to the pipeline schedule before weight updates.

6 Conclusion

We designed and prototyped Piper, a compiler that applies pipeline parallelism to arbitrary PyTorch models. Piper automatically partitions arbitrary PyTorch models and coordinates user-defined pipeline-parallel execution schedules. We demonstrate competitive preliminary performance with existing state-of-the-art pipeline parallelism frameworks while aiming to support a wider range of PyTorch models and execution schedules. We plan to build out our prototype to support all the models in Table 2 and more, as well as more pipeline-parallel execution schedules (e.g., ZeroBubble and DualPipe). We also plan to explore interoperability with other parallelism strategies (e.g., data and tensor parallelism) and ML frameworks (e.g., Megatron-LM and DeepSpeed).

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[2] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[3] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019.

[4] Ho Young Jhoo, Chung-Kil Hur, and Nuno P. Lopes. Pfeife: Automatic pipeline parallelism for pytorch. In *Forty-second International Conference on Machine Learning*, 2025.

[5] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[6] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. Torchtitan: One-stop pytorch native solution for production ready LLM pretraining. In *The Thirteenth International Conference on Learning Representations*, 2025.

[7] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

[8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In

13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, 2018. USENIX Association.

[9] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[10] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism, 2023.

[12] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, 2021.

[13] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019.

[14] James Reed, Pavel Belevich, Ke Wen, Howard Huang, and Will Constable. Pippy: Pipeline parallelism for pytorch. https://github.com/pytorch/PiPPy, 2022.

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

[16] Anxhelo Xhebraj, Sean Lee, Hanfeng Chen, and Vinod Grover. Scaling deep learning training with MPMD pipeline parallelism. In *Eighth Conference on Machine Learning and Systems*, 2025.