TorchTitan: A PyTorch Native Platform for Training Generative AI Models

Tianyu Liu [*1] **Wanchao Liang** [*2]

Abstract

TorchTitan is a PyTorch native open-source platform designed for scalable and flexible training of generative AI models[1]. Integrated tightly with PyTorch's distributed stack while offering efficient optimizations and modular configurations, TorchTitan showcases elastic training of LLMs with composable 4-D parallelism. Moreover, TorchTitan supports extensible abstractions to experiment with new model architectures (e.g., diffusion models) or infrastructure techniques (e.g., a compiler-first FSDP implementation), while biasing towards a clean, minimal codebase. This paper presents the motivation, system architecture, and demonstrated impact of TorchTitan, underscoring its alignment with the CODEML mission to advance open, sustainable machine learning development.

1. Introduction

Large Language Models (LLMs) (Devlin, 2018; Liu et al., 2019; Radford et al., 2019; Chowdhery et al., 2023; Anil et al., 2023; Achiam et al., 2023; Dubey et al., 2024; Jiang et al., 2024; Abdin et al., 2024) power modern NLP applications — from translation and content generation to education and research.

State-of-the-art models like Llama 3.1 (405B, 15T tokens, 16K H100s) (Dubey et al., 2024) and PaLM (540B, 6144 TPUv4s) (Chowdhery et al., 2023) require massive scale and resources. Training them demands careful orchestration of parallelism, memory, and compute budgets, while being robust to GPU failures (Eisenman et al., 2022; Wang et al., 2023; Gupta et al., 2024).

Achieving efficiency at large scale involves combining many parallelisms, Data (Li et al., 2020; Zhao et al., 2023), Tensor (Narayanan et al., 2021), Context (Liu et al., 2023), and Pipeline (Huang et al., 2019), with techniques like activation recomputation (Chen et al., 2016), mixed precision (Micikevicius et al., 2018), and compiler optimizations (Ansel et al., 2024).

However, current systems often fall short due to poor composability of multi-dimensional parallelisms, lack of modularity and extensibility hinder the application and innovation of new technologies. In addition, existing solutions do not come with robust checkpoint save/load solutions, failure recovery mechanisms, or debugging tools for production scale workflows.

TorchTitan addresses these limitations by unifying parallelism and optimization into a PyTorch-native system built on top of DTensor and DeviceMesh (Wanchao Liang, 2023). It supports 4D parallelism strategies, composable training techniques, and efficient distributed checkpoint save/load – all with strong production and research usability. It provides simple and extensible abstractions, allowing fast experimentation with new model architecture and infrastructure innovations. TorchTitan provides multiple performance optimization techniques out of box, including Async Tensor Parallelism, selective activation recomputation, Float8 mixed precision training, torch.compile integration, etc. It also encapsulates communication debugging and failure recovery tools, training receipts and guidelines for real world generative AI models training.

It is worth noting that TorchTitan has received strong community adoptions with good momentum since it became public (currently around 4k GitHub stars).

2. Composable LLM training

2.1. Composable N-D parallelism training

To overcome GPU memory constraints and training efficiency problems, we want to parallelize the computation and storage as much as possible. Modern LLM training usually employs multi-dimensional parallelisms to tackle this. TorchTitan offers simple and performant N-D parallelism solutions that composes with each other, allow model to train efficient on thousands of accelarators without Out of Memory (OOM) issues.

[1]Meta [2]Thinking Machines Lab. Correspondence to: Tianyu Liu <lty@meta.com>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25).

[1]GitHub: https://github.com/pytorch/torchtitan

2.1.1. Meta Device Initialization

At large scale, even initiating models that exceed CPU/GPU memory becomes challenging. TorchTitan uses meta device initialization to avoid this issue: models are first created on a meta device that holds only metadata, enabling ultra-fast setup. Parameters are then sharded into DTensors, with local shards residing on the meta device, and initialized via user-defined functions to ensure correct sharding and RNG behavior.

2.1.2. Fully Sharded Data Parallel (FSDP2)

Original PyTorch FSDP (FSDP1) has limited composability due to its FlatParameter design. TorchTitan integrates FSDP2, which uses per-parameter DTensor sharding, improving memory efficiency (7% lower usage) and performance (1.5% gain). FSDP2 is TorchTitan 's default 1D parallelism, with auto-sharding based on world size. To scale further, TorchTitan includes Hybrid Sharded Data Parallel (HSDP) via a 2D DeviceMesh.

2.1.3. Tensor and Sequence Parallel (TP/SP)

Tensor Parallel (TP) (Narayanan et al., 2021) and Sequence Parallel (SP) (Korthikanti et al., 2023) enable model parallelism across Linear and normalization/dropout layers, respectively. TorchTitan implements TP using PyTorch's `RowwiseParallel` and `ColwiseParallel` APIs on DTensors, requiring no model code changes.

SP shards normalization and dropout layers over the sequence dimension, reducing memory footprint. TP and SP are jointly controlled via the TP degree.

Loss Parallel Cross-entropy loss computation with TP/SP typically incurs high memory and communication costs due to output gathering. TorchTitan implements Loss Parallel by default to compute loss in a sharded fashion, reducing overhead and accelerating training.

2.1.4. Pipeline Parallel (PP)

Pipeline Parallelism (PP) splits models into S stages across device groups, each computing a subset of the model. TorchTitan overlaps computation and communication via microbatching and supports multiple pipeline schedules (Narayanan et al., 2019; Huang et al., 2019; Narayanan et al., 2021; Qi et al., 2023), including recent ones like ZeroBubble and Flexible-Interleaved-1F1B (PyTorch Team, 2024d) via pipeline IR.

TorchTitan simplifies PP integration by introducing a shared `loss_fn` to coordinate gradient scaling, shard/unshard operations, and final reductions.

2.1.5. Context Parallel (CP)

Context Parallelism (CP) (Liu et al., 2023; Liu & Abbeel, 2024; NVIDIA, 2023) splits the sequence dimension across GPUs, enabling 4D parallelism. This allows training with extremely long context lengths. For example, Llama 3.1 8B trained with CP on 8 H100s reached 262K token contexts with minimal MFU drop (PyTorch Team, 2025a). CP integrates seamlessly with existing DP, TP, and PP.

2.2. Optimizing training efficiencies

2.2.1. Activation Checkpointing

To reduce peak memory, TorchTitan supports activation checkpointing (AC) (Chen et al., 2016; He & Yu, 2023) and selective AC (SAC) (Korthikanti et al., 2023) using `torch.utils.checkpoint` at the `TransformerBlock` level. Options include full AC, op-level SAC, and layer-level SAC (every x blocks), enabling configurable memory-compute trade-offs.

2.2.2. Regional `torch.compile`

TorchTitan applies `torch.compile` (Ansel et al., 2024) regionally to each `TransformerBlock`, improving performance through fusion and reuse while ensuring compatibility with DTensor, FSDP2, and TP. This reduces compile time and boosts throughput and memory efficiency.

2.2.3. Asynchronous Tensor Parallelism

AsyncTP (Wang et al., 2022) overlaps TP communication with computation via chunked matmuls and micro-pipelining. Implemented with `SymmetricMemory` buffers (PyTorch Team, 2024a), it leverages `torch.compile` for speedups on modern hardware (e.g., H100).

2.2.4. Mixed Precision and Float8

FSDP2 supports mixed precision (`bfloat16` compute, `float32` reduce). TorchTitan extends this with Float8 for linear layers, supported by `torchao.float8` with dynamic scaling (Micikevicius et al., 2022; PyTorch Community, 2023). It integrates cleanly with `torch.compile`, FSDP2, and TP (PyTorch Team, 2024c).

2.3. Production ready training

To enable production-grade training, TorchTitan integrates many features out of the box, including efficient checkpoint save/load and robust debugging tools.

2.3.1. Scalable Distributed Checkpointing

Checkpointing is critical for fault recovery and model publish. Traditional methods either save unsharded states (easy to reuse but slow) or sharded local states (fast but inflexible). TorchTitan leverages PyTorch Distributed Checkpointing (DCP), which uses DTensor to decouple tensor data from parallelism. DCP stores internal metadata and restores shards based on the DTensor layouts, enabling fast, parallelism-agnostic loading.

DCP also supports asynchronous checkpointing, overlapping save operations with training via background threads. This reduces checkpointing overhead by 5–15× on models like Llama 3.1 8B (PyTorch Team, 2024b).

2.3.2. Flight Recorder for Debugging

Debugging collective timeouts at scale is difficult due to asynchronous communications. In TorchTitan, we use PyTorch's Flight Recorder to log all collective operations with timing and metadata (e.g., tensor sizes, ranks, stack traces), which helps localize communication failures in FSDP, TP, and PP.

2.3.3. Fault Tolerance

In additional to efficient checkpoint save/load for failure recovery, TorchTitan is experimenting many fault tolerant training strategies, including local SGD (Stich, 2018), DiLoCo (Douillard et al., 2023)

2.4. Experimentation

To showcase the elasticity and scalability of TorchTitan, we conducted training on Llama 3.1 family of models on a wide range of scales (from 8 to 512 GPUs) on a custom built NVIDIA H100 cluster. We gradually increase the model size (8B, 70B, and 405B) with different parallelism strategies (up to 4D). We demonstrate accelerations ranging from 65.08% on Llama 3.1 8B at 128 GPU scale (1D), 12.59% on Llama 3.1 70B at 256 GPU scale (2D), to 30% on Llama 3.1 405B at 512 GPU scale (3D) over optimized baselines, and stay converging. For a full detailed experiment results, please see Appendix A.

3. Extensions

As one of TorchTitan's core design philosophy is to make the codebase easy to understand, use, and extend for different training purposes, it provides basic reusable / swappable components while aiming to keep the codebase as clean and minimal as possible.

In order to facilitate innovations in new modeling architectures and infrastructure techniques and support flexible configurations of training components, while reusing the core training script, TorchTitan introduces the concept of `TrainSpec` (see the code snippet below) where each instance of the class is a training specification at the coarse level of model class and model parallelization functions, and the functions to build optimizer, learning rate scheduler, data loader, tokenizer, loss function, etc.

In this section, we will see how `TrainSpec` is used to support new model architecture (FLUX, a diffusion model), and innovations in infrastructure (SimpleFSDP, a compiler-first FSDP implementation).

```
@dataclass
class TrainSpec:
    name: str
    cls: type[nn.Module]
    config: Mapping[str, BaseModelArgs]
    parallelize_fn
    pipelining_fn
    build_optimizers_fn
    build_lr_schedulers_fn
    build_dataloader_fn
    build_tokenizer_fn
    build_loss_fn
    build_metrics_processor_fn

def register_train_spec(train_spec:
    TrainSpec) -> None:
    ...

def get_train_spec(name: str) -> TrainSpec:
    ...
```

In addition, we also introduce `ModelConvert` as a general interface for applying modification to a PyTorch model (see the code snippet below). Typical use cases include

- applying quantization: using QAT, Float8, or other specialized linear layers;
- swapping to fused optimized layers, such as FlashAttention, various norm layers, etc.

```
class ModelConverter(Protocol):
    def __init__(self, job_config:
        JobConfig, parallel_dims:
        ParallelDims):
        ...

    def convert(self, model: nn.Module):
        """Inplace convertion of the
            model."""
        ...

    def post_optimizer_hook(self, model:
        Union[nn.Module, List[nn.Module]]):
        """Post-optimizer (optional) hook
            (e.g. compute weights
            statistics)."""
        ...
```

With these extension points, TorchTitan aims to support both in-repo experiments and out-of-repo usage as a submodule.

3.1. Modeling architecture

In addition to language models, another important class of generative AI models is diffusion models (Ho et al., 2020; Rombach et al., 2022; Peebles & Xie, 2023; Podell et al., 2024; Esser et al., 2024). In TorchTitan, we present diffusion model training of the FLUX.1 model (Labs, 2024) which achieves state-of-the-art performance on text-to-image tasks (PyTorch Team, 2025b).

All FLUX.1 models are based on a hybrid architecture of multimodal and parallel diffusion transformer blocks and scaled to 12B parameters. TorchTitan supports FLUX model training via FSDP / HSDP and activation checkpointing. This is done by specifying a `TrainSpec` where we reuse the functions for building optimizers and learning rate schedulers, but create new model definitions and configurations, and corresponding functions to parallelize model and build data loader, tokenizers, and loss function.

Moreover, to show case TorchTitan's capability of doing end-to-end pretraining of diffusion models, we perform real-scenario training of the FLUX.1 [schnell] model on the Conceptual 12M dataset (Changpinyo et al., 2021). We train the model for on 256 NVIDIA H100 GPUs for 60000 steps (about 5 epochs), global batch size 1024, with a linear warmup learning rate schedule peaked at $1e-4$ and mixed precision training (`bfloat16` computation, `float32` reduction). Interestingly, the training converges and output meaningful pictures. See Figure 1.

For a concrete training recipe to reproduce the results, please refer to the `flux` folder in TorchTitan.

3.2. Infrastructure technique

With the `TrainSpec` abstraction, TorchTitan can host not only new models but also infrastructure innovations applied to existing models.

SimpleFSDP (Zhang et al., 2024) is a compiler-based Fully Sharded Data Parallel (FSDP) framework, which has a simple implementation for maintenance and composability, allows full computation-communication graph tracing, and brings performance enhancement via compiler back-end optimizations.

TorchTitan includes the front-end implementation for SimpleFSDP, which is compatible with other training techniques such as TP, CP, PP, AC, and mixed precision. Compared with the default `TrainSpec` of Llama 3.1 training, the implementation only substitutes `parallelize_fn` and reuses others.

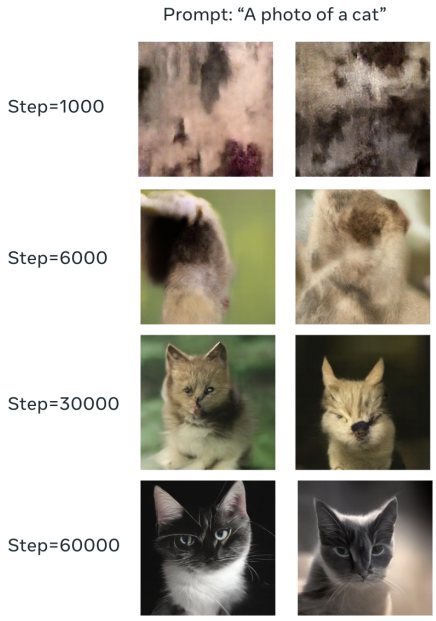

Figure 1. Example image output of FLUX model training.

We believe that it provides an ideal playground for innovations such as communication optimizations (e.g. reordering and bucketing of FSDP communications covered in (Zhang et al., 2024)) on the back-end of the compiler, given a fully traced graph containing both computation nodes and communication nodes.

4. Ongoing and future work

TorchTitan is still under active development of new models and features. We are hosting experimental folders on MoE LLMs and multimodal models, and building necessary training techniques such as Expert Parallel and parallelism-compatible multimodal data loader, respectively.

In addition, as Reinforcement Learning becomes more important, there are requests to integrate TorchTitan with post-training frameworks, so that it could be used as a post-training capable trainer.

5. Conclusion

As a PyTorch native platform that is revolutionizing rapid experimentation and large-scale training of generative AI models, TorchTitan has achieved widespread adoption across top research institutions and industry labs.

We hope TorchTitan's cutting-edge training techniques, lean implementation, and flexible extension points could empower researchers and developers to push the boundaries of what is possible in generative AI.

References

Abdin, M., Jacobs, S. A., Awan, A. A., Aneja, J., Awadallah, A., Awadalla, H., Bach, N., Bahree, A., Bakhtiari, A., Behl, H., et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., and Team, G. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. 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*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Changpinyo, S., Sharma, P., Ding, N., and Soricut, R. Conceptual 12M: Pushing web-scale image-text pre-training to recognize long-tail visual concepts. In *CVPR*, 2021.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training Deep Nets with Sublinear Memory Cost, 2016. URL https://arxiv.org/abs/1604.06174.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. PaLM: Scaling language modeling with Pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Devlin, J. BERT: Pre-training of deep bidirectional Transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Douillard, A., Feng, Q., Rusu, A. A., Chhaparia, R., Donchev, Y., Kuncoro, A., Ranzato, M., Szlam, A., and Shen, J. Diloco: Distributed low-communication training of language models. *arXiv preprint arXiv:2311.08105*, 2023.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Eisenman, A., Matam, K. K., Ingram, S., Mudigere, D., Krishnamoorthi, R., Nair, K., Smelyanskiy, M., and Annavaram, M. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 929–943, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/eisenman.

Esser, P., Kulal, S., Blattmann, A., Entezari, R., Müller, J., Saini, H., Levi, Y., Lorenz, D., Sauer, A., Boesel, F., Podell, D., Dockhorn, T., English, Z., and Rombach, R. Scaling rectified flow transformers for high-resolution image synthesis. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Gupta, T., Krishnan, S., Kumar, R., Vijeev, A., Gulavani, B., Kwatra, N., Ramjee, R., and Sivathanu, M. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, pp. 1110–1125, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3650085. URL https://doi.org/10.1145/3627703.3650085.

He, H. and Yu, S. Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. *Proceedings of Machine Learning and Systems*, 5:414–427, 2023.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 6840–6851. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 341–353. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/80083951326cf5b35e5100260d64ed81-Paper-mlsys2023.pdf.

Labs, B. F. Flux. https://github.com/black-forest-labs/flux, 2024.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. PyTorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Liu, H. and Abbeel, P. Blockwise parallel Transformers for large context models. *Advances in Neural Information Processing Systems*, 36, 2024.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise Transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: A robustly optimized BERT pretraining approach, 2019. URL https://arxiv.org/abs/1907.11692.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training, 2018. URL https://arxiv.org/abs/1710.03740.

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., and Wu, H. FP8 formats for deep learning, 2022. URL https://arxiv.org/abs/2209.05433.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. 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. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

NVIDIA. Megatron Core API Guide: Context Parallel, 2023. URL https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html. Accessed: 2023-09-25.

Peebles, W. and Xie, S. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4195–4205, October 2023.

Podell, D., English, Z., Lacey, K., Blattmann, A., Dockhorn, T., Müller, J., Penna, J., and Rombach, R. SDXL: improving latent diffusion models for high-resolution image synthesis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=di52zR8xgf.

PyTorch Community. Float8 in PyTorch 1.x, 2023. URL https://dev-discuss.pytorch.org/t/float8-in-pytorch-1-x/1815. PyTorch Discussion Thread.

PyTorch Team. Introducing Async Tensor Parallelism in PyTorch. https://discuss.pytorch.org/t/distributed-w-torchtitan-introducing-async-tensor209487, 2024a. PyTorch Forum Post.

PyTorch Team. Optimizing checkpointing efficiency with PyTorch DCP. https://discuss.pytorch.org/t/distributed-w-torchtitan-optimizing-checkpointing211250, 2024b. PyTorch Forum Post.

PyTorch Team. Enabling Float8 all-gather in FSDP2. https://discuss.pytorch.org/t/distributed-w-torchtitan-enabling-float8-all-gath209323, 2024c. PyTorch Forum Post.

PyTorch Team. Training with zero-bubble Pipeline Parallelism. https://discuss.pytorch.org/t/distributed-w-torchtitan-training-with-zero-bubbl214420, 2024d. PyTorch Forum Post.

PyTorch Team. Breaking barriers: Training long context llms with 1M sequence length in PyTorch using Context Parallel. https://discuss.pytorch.org/t/distributed-w-torchtitan-breaking-barriers-training-long-context-llms-with-1m-sequence-length-in-pytorch-using-context-parallel/215082, 2025a. PyTorch Forum Post.

PyTorch Team. FLUX is here: Experience diffusion model training on TorchTitan. https://discuss.pytorch.org/t/distributed-w-torchtitan-flux-is-here-experience-diffusion-model-training-on-torchtitan/221119, 2025b. PyTorch Forum Post.

Qi, P., Wan, X., Huang, G., and Lin, M. Zero bubble pipeline parallelism, 2023. URL https://arxiv.org/abs/2401.10241.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text Transformer. *J. Mach. Learn. Res.*, 21(1), January 2020. ISSN 1532-4435.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10684–10695, June 2022.

Stich, S. U. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Wanchao Liang. PyTorch DTensor RFC, 2023. URL https://github.com/pytorch/pytorch/issues/88838. GitHub Issue.

Wang, S., Wei, J., Sabne, A., Davis, A., Ilbeyi, B., Hechtman, B., Chen, D., Murthy, K. S., Maggioni, M., Zhang, Q., et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 93–106, 2022.

Wang, Z., Jia, Z., Zheng, S., Zhang, Z., Fu, X., Ng, T. S. E., and Wang, Y. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 364–381, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613145. URL https://doi.org/10.1145/3600006.3613145.

Zhang, R., Liu, T., Feng, W., Gu, A., Purandare, S., Liang, W., and Massa, F. Simplefsdp: Simpler fully sharded data parallel with torch.compile, 2024. URL https://arxiv.org/abs/2411.00284.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. PyTorch FSDP: Experiences on scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, aug 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611569. URL https://doi.org/10.14778/3611540.3611569.

A. Experiment results

The experiments are conducted on a cluster of NVIDIA H100 GPUs[2] with 95 GiB memory, where each host is equipped with 8 GPUs and NVSwitch. Two hosts form a rack connected to a TOR switch. A backend RDMA network connects the TOR switches.

In TorchTitan we integrate a checkpointable data loader and provide built-in support for the C4 dataset (`en` variant), a colossal, cleaned version of Common Crawl's web crawl corpus (Raffel et al., 2020). We use the same dataset for all experiments in this section. For the tokenizer, we use the official one (tiktoken) released together with Llama 3.1.

A.1. Performance

To showcase the elasticity and scalability of TorchTitan, we experiment on a wide range of GPU scales (from 8 to 512), as the underlying model size increases (8B, 70B, and 405B) with a varying number of parallelism dimensions (up to 4D). To demonstrate the effectiveness of the optimization techniques introduced in the paper, we show how training throughput improves when adding each individual technique on appropriate baselines. In particular, when training on a higher dimensional parallelism with new features, the baseline is always updated to include all previous techniques.

Table 1. 1D parallelism (FSDP) on Llama 3.1 8B model, 8 GPUs. Mixed precision training. Selective activation checkpointing. Local batch size 2, global batch size 16. (Stats per GPU)

Techniques	Throughput (Tok/Sec)	Comparison	Memory (GiB)
FSDP	6,258	100%	81.9
+ `torch.compile`	6,674	+ 6.64%	77.0
+ `torch.compile` + Float8	9,409	+ 50.35%	76.8

Table 2. 1D parallelism (FSDP) on Llama 3.1 8B model, 128 GPUs. Mixed precision training. Selective activation checkpointing. Local batch size 2, global batch size 256. (Stats per GPU)

Techniques	Throughput (Tok/Sec)	Comparison	Memory (GiB)
FSDP	5,645	100%	67.0
+ `torch.compile`	6,482	+ 14.82%	62.1
+ `torch.compile` + Float8	9,319	+ 65.08%	61.8

Table 3. 2D parallelism (FSDP + TP) + `torch.compile` + Float8 on Llama 3.1 70B model, 256 GPUs. Mixed precision training. Full activation checkpointing. FSDP degree 32, TP degree 8. Local batch size 16, global batch size 512. (Stats per GPU)

Techniques	Throughput (Tok/Sec)	Comparison	Memory (GiB)
2D	897	100%	70.3
+ AsyncTP	1,010	+ 12.59%	67.7

Table 4. 3D parallelism (FSDP + TP + PP) + `torch.compile` + Float8 + AsyncTP on Llama 3.1 405B model, 512 GPUs. Mixed precision training. Full activation checkpointing. FSDP degree 4, TP degree 8, PP degree 16. Local batch size 32, global batch size 128. (Stats per GPU)

Schedule	Throughput (Tok/Sec)	Comparison	Memory (GiB)
1F1B	100	100%	78.0
Interleaved 1F1B	130	+ 30.00%	80.3

[2]The H100 GPUs used for the experiments are non-standard. They have HBM2e and are limited to a lower TDP. The actual peak TFLOPs should be between SXM and NVL, and we don't know the exact value.

Table 5. FSDP + CP + `torch.compile` + Float8 on Llama 3.1 8B model, 8 GPUs. Mixed precision training. Full activation checkpointing. Local batch size 1. (Stats per GPU)

Schedule	Sequence Length	Throughput (Tok/Sec)	Memory (GiB)
FSDP 8, CP 1	32,768	3,890	83.9
FSDP 4, CP 2	65,536	2,540	84.2
FSDP 2, CP 4	131,072	1,071	84.0
FSDP 1, CP 8	262,144	548	84.5

Table 6. 4D parallelism (FSDP + TP + PP + CP) + `torch.compile` + Float8 + AsyncTP + 1F1B on Llama 3.1 405B model, 512 GPUs. Mixed precision training. Full activation checkpointing. TP degree 8, PP degree 8. Local batch size 8. (Stats per GPU)

Schedule	Sequence Length	Throughput (Tok/Sec)	Memory (GiB)
FSDP 8, CP 1	32,768	76	75.3
FSDP 4, CP 2	65,536	47	75.9
FSDP 2, CP 4	131,072	31	77.1
FSDP 1, CP 8	262,144	16	84.9

A.2. Loss converging

TorchTitan has ensured the loss converging of individual techniques as well as their various combinations of parallelisms and optimizations.

For example, below is a series of loss-converging tests covering both parallelisms and training optimizations. We assume the correctness of FSDP, which can be further verified by comparing it with DDP or even single-device jobs.

Table 7. Loss-converging tests setup.

Parallelism	Techniques
FSDP 8 (ground truth)	default
FSDP 8, TP 2, PP 2	torch.compile, Float8, async TP, Interleaved 1F1B
FSDP 8, TP 2, CP 2, PP 2	torch.compile, Float8, async TP, Interleaved 1F1B
FSDP 8, CP 8	default

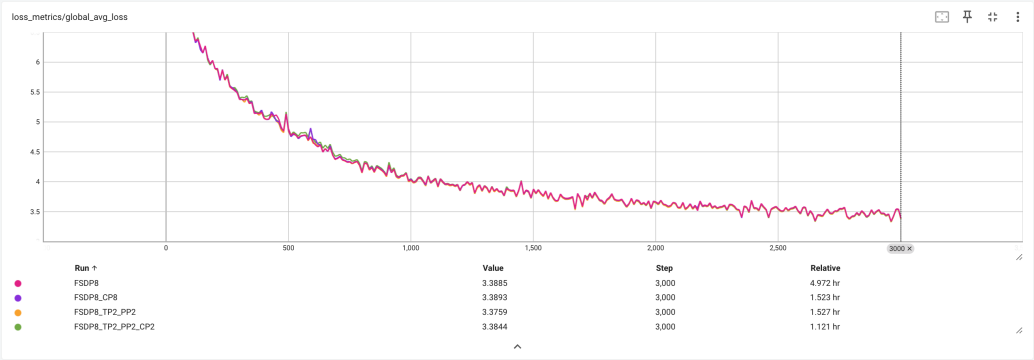

Figure 2. Loss converging tests on Llama 3.1 8B. C4 dataset. Local batch size 4, global batch size 32. 3000 steps, 600 warmup steps.