Summary of "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM"

Haotian Gong (haotiang), Zhongwei Xu (xzw), Zheng Li (jimmyli), Siyuan Dong (dougdong)

Problem and Motivation

Training large language models efficiently on GPU clusters faces challenges due to limited GPU memory capacity and the massive number of computations required. Although emerging parallelism techniques such as tensor and pipeline parallelism can boost the training efficiency, naive usage of these methods will lead to scale issues beyond a certain number of GPUs. The motivation of this paper is to combine pipeline, tensor, and data parallelism (PTD-P) to achieve efficient large language models training performance. The hypothesis is that achieving a balance between pipeline, tensor, and data parallelism can improve training efficiency.

Related Works

Data parallelism distributes the full model across all workers. The input dataset is sharded and workers aggregate their gradient periodically to ensure the version of weights among all workers are consistent. Data parallelism can increase training efficiency but the communication overhead for gradient synchronization is a bottleneck.

Pipeline Model Parallelism involves splitting the model into several segments and processing different segments on different devices sequentially. Each device processes a part of the data through its segment of the model before passing it to the next device. Notable implementations include GPipe and PipeDream-Flush, which address the latency issues inherent in pipeline execution by optimizing the scheduling of forward and backward passes and minimizing idle times through efficient batching strategies. To reduce the size of the pipeline bubble, each device in the pipeline is assigned multiple pipeline stages (called a model chunk), instead of a single stage of a contiguous set of layers.

Tensor Model Parallelism splits the computation of large models across multiple devices to overcome memory limitations per device. For instance, the computation of MLP blocks and Self-Attention can be split into multiple devices. Megatron-LM further refined this approach by optimizing the distribution of transformer layers across devices.

Solution Overview

The paper introduces an optimized method combining tensor, pipeline, and data parallelism to scale training to thousands of GPUs efficiently. The tradeoffs between different parallelisms are analyzed for optimizing both communication and computation. For communication improvements, the paper reduces the overhead of cross-node communication through scatter/gather optimization that combines both tensor parallelism and pipeline parallelism. The tensor is splitted into equal-size chunks on the send side and the receive side performs an

all-gather over NVLink to re-materialize the full tensor. The computation optimizations are: 1) the data layout in the transformer layer is changed to avoid memory-intensive transpose operations; 2) the paper implements customized fused kernels (operations) to reduce overheads.

Limitations

The system greatly depends on specific hardware and software designs. For example, part of the optimization is based on transformer models, which may limit its probability of being applied for other ML tasks.

Future Research Directions

The implementation is based on some simplified situations and future research may consider more complex designs. For example, some designs such as PipeMare do away with flushes completely, which can be considered as future research directions. Besides, implementations can be further improved by making it generalizable for other hardwares and ML models.

Summary of Class Discussion

Technical terms such as interleaving (alternating communication + computation), all-reduce (broadcasting updates to other machines) were discussed.
The class also talked about hypothetical scenarios, such as

1. What would happen when inter-node communication (e.g. NVLink) is slower?
   Tensor parallelism would have worse performance.
2. What will happen if we have very large models?
   It will be costly to use data parallelism, and we have to swap GPU memory with host memory or even disk data.

Interesting observations/extensions are raised:

1. GPUs in earlier stages need to hold more activations, leading those in later stages to have memory underutilized. The key takeaway is that it is important for designers to consider the balance between computation bubbles and memory bubbles.
2. There are other techniques such as parallelizing optimizer states and from the papers regarding FSDP, and ZeRO. The key is to balance the system.
3. We prefer Data parallelism as long as Tensor/pipeline parallelism can fit in a single node.

Summary of "Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates"

Problem and Motivation

Training large DNN models with billions of parameters requires the deployment of multi-GPU clusters. As a result, one must address the aspect of fault-tolerance and ensure that the training process will not be severely affected by the failure of one or more GPUs. The stakes are higher with larger models as we have higher probability of a GPU failing and pay a higher price as thousands of GPUs must wait for recovery.

The hypothesis of this work is that formulating the DNN fault-tolerance problem formally and optimizing it results in a general framework that has provable fault tolerance.

Related Works

Existing methods for hybrid parallelism mainly fall into two categories:
Checkpointing: Periodically saves training progress to allow recovery from the most recent checkpoint after a failure.
Redundant Computation: Introduces computational redundancy during the training process to recovery from potential failures.
Representative works include Varuna and Bamboo. Varuna uses checkpointing and dynamic reconfiguration to handle failures but incurs high recovery overhead in high failure rate scenarios. Bamboo reduces recovery time through redundant computation but continuously introduces additional resource overhead during the training process.

Solution Overview

The core idea is to use pipeline templates and plan ahead before training begins, allowing for quick adaptation to failures. Key points include:
Plan-execute co-design: Before training begins, Oobleck generates a set of heterogeneous pipeline templates with varying amounts of nodes by solving a Frobenius problem. These templates are reused throughout the training process.
Multi-replica fault tolerance: Oobleck instantiates at least f+1 logically equivalent pipeline replicas to tolerate any f simultaneous failures.
Fast recovery: When failures occur, Oobleck utilizes the inherent redundancy in hybrid parallel execution, and leverages existing model replicas to achieve fast recovery by copying missing model states from other pipelines, without restarting from checkpoints.
Resource utilization maximization: Oobleck proves that for any f or fewer failures, there is always a combination of pre-generated pipeline templates that can cover all available resources, ensuring no resources are idle at any time. It estimates iteration time using dynamic programming and picks the best one.

Limitation

Planning complexity: Although pipeline template generation only needs to be done once at the start of training, the planning algorithm may incur computational overhead for very large models and cluster sizes. While the paper mentions that the actual overhead is small, further optimization may be needed at larger scales.
Hardware heterogeneity: Although Oobleck supports heterogeneous pipeline templates, load balancing and resource utilization may be affected in cases of significant hardware performance differences (e.g., different types of GPUs), requiring additional mechanisms to handle.

Future Research Direction

Further improve algorithms for generating pipeline templates to support more complex hardware environments. Explore methods to improve fault tolerance in resource-constrained scenarios, such as introducing incremental model redundancy or more flexible replica management strategies.

Summary of Class Discussion

What is hybrid-parallel? Hybrid-parallel is using the different parallel paradigms in conjunction (data parallel, pipeline parallel, tensor parallel)
Does it make sense to retrieve state from other pipelines when they are working on different data? Yes, because the model is uniform from the outside and their parameters are updated in conjunction over despite working on different data.