Distributed Training Strategies on Commodity Hardware

1st Can Kerem Akbulut
Dept. of Computer Science
Columbia University
New York, NY, USA
cka2115@columbia.edu

2nd Rakene Chowdhury
Dept. of Computer Science
Columbia University
New York, NY, USA
rc3574@columbia.edu

3rd Juan Gutierrez
Dept. of Computer Science
Columbia University
New York, NY, USA
jmg2048@columbia.edu

***Abstract*—Training large language models (LLMs) increasingly requires distributed computing, yet most empirical analyses assume high-bandwidth interconnects unavailable to many academic or resource-constrained environments. This work presents a controlled evaluation of data, tensor, and pipeline parallelism, as well as ZeRO, on PCIe-connected NVIDIA RTX A6000 GPUs. Using models from 10M to 1B parameters and fixed-shape synthetic data, we measure throughput, memory consumption, and scaling behavior across 1–4 GPUs. Results show that only data parallelism and ZeRO Stage 1 achieve meaningful scaling on two GPUs, while tensor and pipeline parallelism exhibit limited benefits due to communication bottlenecks. All strategies experience significant scaling degradation on four GPUs, where communication dominates computation. These findings highlight the constraints of commodity hardware and provide practical guidance for selecting parallelism strategies in low-bandwidth environments.**

***Index Terms*—distributed training, pipeline parallelism, tensor parallelism, data parallelism, memory optimization, Megatron, ZeRO**

I. Introduction & Motivation

The rapid scaling of large language models (LLMs) has made distributed training a fundamental requirement rather than an optimization. Modern transformer models routinely contain upwards of billions of parameters [1], [2] and training them efficiently demands exploitation of their inherently parallelizable design [3]. While state-of-the-art training infrastructures often rely on specialized accelerators and high-bandwidth interconnects, a large fraction of academic labs, startups and independent researchers may find it difficult to operate on the same scale as enterprise labs given the expensive cost-per-accelerator and the architecture necessary for large-scale distributed training. Understanding how different distributed training strategies behave under resource-constrained environments is critical for making informed training decisions.

Distributed training efficiency directly impacts the feasibility of LLM scaling. Poor parallelization choices can lead to excessive communication overhead, GPU under-utilization, memory fragmentation, or diminishing returns as the number of devices increases. Conversely, well-chosen strategies can unlock near-linear scaling even on relatively modest hardware. As models grow larger and training budgets remain constrained, the difference between effective and ineffective distributed training can determine whether an experiment is, at all, practical.

Despite the importance of this problem, there exists a gap in systematic, empirical comparisons of distributed training strategies on commodity hardware. Much of the existing literature focuses either on large-scale clusters with specialized interconnects (e.g. using Nvidia's NVLink and/or InfiniBand) or evaluates individual methods in isolation. Moreover, the limited availability of alternative compute-competitive GPUs outside the current NVIDIA ecosystem further complicates cost–performance trade-offs, making it difficult for practitioners to identify more economical yet effective training configurations. As a result, practitioners lack clear guidance on the trade-offs between commonly used parallelism approaches and hybrid schemes when deployed on real-world hardware with limited memory bandwidth and interconnect performance.

In this work we aim to close this gap by conducting a controlled evaluation of distributed training strategies on a multi-GPU setup consisting of NVIDIA RTX A6000 GPUs. We study the performance characteristics, memory efficiency, and communication overhead of several widely used approaches in low-resource environments, focusing on how they behave as model size and GPU count increase. Rather than proposing a new algorithm, our goal is to provide practical insights into when and how certain strategies outperform others under realistic constraints.

Our contributions are threefold. First, we present a systematic comparison of distributed training strategies under identical hardware and software conditions, isolating the effects of parallelization choices. Second, we analyze the interaction between throughput and peak memory usage on commodity GPUs connected via PCIe, highlighting key bottlenecks and providing insight into factors that limit scaling efficiency. Finally, we distill these findings into concrete recommendations for practitioners training large models on similar hardware.

By grounding our evaluation in widely accessible GPU configurations, we hope this work serves as a practical reference for researchers and engineers seeking to train large language models efficiently without access to specialized infrastructure. Our results show that on PCIe-connected commodity GPUs, data parallelism and ZeRO Stage 1 dominate tensor and pipeline parallelism on two devices, with most strategies exhibiting negative scaling at four GPUs.

II. Models and Data Description

All experiments were run on RTX A6000 (48 GB VRAM) GPUs connected via PCIe, provided by the cloud service Runpod.io.

The software stack used

- Python 3.12.3
- PyTorch 2.8.0
- Megatron-Core 0.12.3
- DeepSpeed 0.18.2
- NVIDIA-ml-py 13.580.82

A. The Dataset

Our models were trained on a custom dataset we crafted which generates sequences across 8000 uniformly distributed token IDs. All tokens are simply integers. Consistent with LLM training pipelines, all tensors generated by this dataset are a fixed-shape which then also has the side-effect of a fixed compute cost per step, eliminating FLOPs variability. By removing variability in the training data, it allowed us to focus on overall system behavior and have more confidence in our scaling measurements. Given the deterministic nature of the dataset, every batch and sequence across strategies for a given model size is guaranteed to always be identical. In other words: the n_{th} batch of the 10M model in a single GPU run will be identical to the n_{th} batch of the 10M model in a 4 GPU data parallel run or the 2 GPU ZeRO-stage1 run.

Total token counts are fixed for fairness across strategies and device counts such that, for a given model size, the total tokens seen by a single GPU will be the total number of tokens seen by a group of GPUs in a given strategy. The batch size in our configurations is the ”global batch size.” For data parallelism and ZeRO, the effective batch size that any one GPU sees is always the batch size configured divided by the number of devices, aka the world size. For tensor parallelism, all devices need to see the same input, so all devices use the global batch size. Finally for pipeline parallelism, each device sees microbatches calculated from the global batch.

B. Tensor, Pipeline, Data Parallelism

To measure scaling efficiency on commodity hardware, we ran models ranging from 10M parameters up to 1B parameters. The model configurations were not tuned for accuracy or convergence. Instead, we configure them in such a way to reflect realistic scaling behavior (activation sizes, parameter layout, FLOPs/GPU) that stress distributed training systems in ways consistent with GPT-like workloads [6], [7]. The 10M model uses data type *float32*, but all other models used *bfloat16*. *bfloat16* ensured we could fit the 1B model on a single RTX A6000 GPU.

We used NVIDIA's Megatron-LM [8] framework which supplies a *GPTModel*. We configure the model and data loaders using the hyperparameters seen in Table I. We chose this framework and model because they provide a fairly straightforward configuration of tensor, pipeline, and data parallelism [1]. The optimizer used is PyTorch's Adam and the learning rate was Megatron's default value of `1e-4`.

C. Alternative Model for ZeRO Experiments

We attempted to integrate ZeRO [9] with Megatron's *GPTModel* for a fourth strategy, but discovered Megatron-LM builds parameters in such a way that prevents DeepSpeed from intercepting and sharding parameters via ZeRO. While there is Megatron-DeepSpeed [10], using it would require significant restructuring of our training loop. Instead, we opted to create a simple transformer decoder that ZeRO could operate with. We took a separate set of single GPU baseline measurements with this lightweight, GPT-like model and provide an independent analysis of ZeRO stages 1, 2, 3, and 3-offload [14] on 2 and 4 GPUs for completeness. Because of these fundamental differences, we do not compare the results of ZeRO against the other three distributed training strategies.

The lightweight *SimpleTransformerDecoder* model follows standard decoder structures using PyTorch primitives. Unlike Megatron's *GPTModel*, this implementation uses standard PyTorch *nn.Module* parameter initialization that allows DeepSpeed's ZeRO to intercept and shard parameters during model construction.

The model architecture consists of:

- Token embeddings (*nn.Embedding*) and learned positional embeddings
- A dropout layer with rate 0.1
- n transformer layers (driven by model configuration)
- Layer normalization and linear output projection to vocabulary size

Each transformer layer implements:

- Multi-head self-attention using PyTorch's *nn.MultiheadAttention*, with the number of heads driven by model configuration
- Pre-layer normalization (LayerNorm applied before attention and feedforward)
- Feed-forward network:
  - Linear(d_{model}, d_{ff})
  - GELU
  - Linear(d_{ff}, d_{model})
- Residual connections around both attention and feedforward blocks

The model uses identical hyperparameters to the Megatron experiments (Table I). The configuration differences between ZeRO and the Megatron experiments are only in the batch size and sequence lengths used, due to ZeRO's microbatching strategy, see Appendix H. The optimizer used for the ZeRO experiments was once again Adam, with a learning rate of `2e-4`.

[1] We did not use their ”distributed optimizer” for data parallelism. This would have the effect of using stage 1 ZeRO as opposed to the traditional approach to ”data parallelism” which is what we were more interested in measuring.

TABLE I: Model configurations used in Megatron experiments

Model Size	d_{model}	n_{heads}	n_{layers}	d_{ff}	Seq Len	Batch Size
10M	320	8	4	1536	128	16
100M	768	12	12	3072	512	16
300M	1024	16	24	4096	1024	8
500M	1280	20	24	5120	1024	8
1B	1536	24	36	6144	1024	4

III. Training and Profiling Methodology

For the purposes of this study, model convergence is not a priority. Each experiment trained a model for 200 steps. A baseline experiment is run on each model by training on a single GPU to understand single device memory and throughput behavior. Given that we run five models on 1, 2, and 4 GPUs across tensor, pipeline, data parallelism, and 4 ZeRO stages, we made heavy use of a Makefile to simplify experiment execution, as we have over 70 experimental runs.

200 training steps were determined adequate because our primary concern is steady-state and peak memory performance. After the first few dozen steps, the devices would reach stable throughput, memory, and utilization patterns in the different strategies. The remaining steps provide enough samples for stable averages and meaningful statistical sampling. GPU memory would peak early and then stabilize. This stability should provide deeper credence to our scaling analysis.

To properly capture the end-to-end iteration step-time of a particular strategy, our primary measurement is wall-clock time. Using PyTorch's device synchronization mechanism allows us to measure both compute and communication overhead of a strategy, per device, per step, by forcing the CPU to wait until all queued CUDA operations have completed. This approach captures the total cost of computation, coordination, and communication. Furthermore, given the benefits of our fixed-shape dataset, we can focus on the question of: what is the true training time for a particular strategy given a fixed set of tokens, devices, and FLOPs? Training loop details can be found in Appendix E.

To compute throughput, we snapshot a timestamp just before the training loop begins and immediately after it ends, providing the wall-clock time of the 200 iterations. With the number of training steps, batch size, and sequence length, we know the total number of tokens a strategy will see. This is the basis of our throughput computation.

GPU memory and utilization were measured using NVIDIA's pyNVML [11]. Throughput and peak GPU memory provide the most meaningful metrics to our study, but average GPU utilization also showed interesting results, e.g. when a GPU has high utilization but very low throughput compared to a single GPU baseline.

Final metrics are computed using aggregated per-device measurements following standard distributed training practices; full aggregation definitions are provided in Appendix F.

Our survey does not perform hyperparameter or kernel-level performance tuning in order to isolate the effects of parallelization strategy under identical configurations.

It is worth mentioning some of the nuances and differences of model and strategy. The following sections go into these details.

TABLE II: Model and Batch Partitioning Across Strategies

Parallelism	Batch Behavior	Model Partitioning
Data	Split	Replicated
Tensor	Unchanged	Shard weights
Pipeline	Microbatches	Shard layers
ZeRO-stage1	Split	Shard optimizer
ZeRO-stage2	Split	Shard optimizer+gradients
ZeRO-stage3	Split	Shard opt+grads+params

A. Megatron Strategies

As stated earlier, the tensor, pipeline, and data parallelism experiments leveraged Megatron's *GPTModel*. These experiments are run using PyTorch's parallel process runner *torchrun*. The `--nproc-per-node` argument controls the number of devices to use in the experiment, also referred to as the "world size." Tensor parallelism shards the weights across all available devices. This requires an all-gather or reduce-scatter of activations and an all-reduce or reduce-scatter of gradients. Data parallelism replicates the model across all devices and only requires an all-reduce of gradients. Finally, pipeline parallelism assigns specific layers to specific devices, in order, and does peer-to-peer communication of both activations and gradients. Configuration and implementation details can be found in Appendix G

TABLE III: Communication Patterns Across Strategies[2]

Parallelism	Forward Pass	Backward Pass
Data	None	AR(grads)
Tensor	AG(acts) / RS(acts)	AR(grads) / RS(grads)[3]
Pipeline	P2P(acts)	P2P(grads)
ZeRO-1	None	AR(grads)
ZeRO-2	None	RS(grads)
ZeRO-3	AG(params)	RS(grads)

B. ZeRO

Fundamentally, ZeRO is a form data parallelism, but with a focus on various forms of memory optimization. The underlying framework that provides ZeRO's capabilities is *deep-*

[2] AG = all-gather, RS = reduce-scatter, AR = all-reduce, P2P = point-to-point send/recv, acts = activations, grads = gradients, params = parameters.

[3] Forward/backward communication operation depends on layer type (column vs row parallel).

speed. *deepspeed* operates similar to PyTorch's *DistributedDataParallel*, such that you *deepspeed.initialize* your model and the library returns a wrapped instance of it. The training loop interacts with this wrapped "engine." The initialization also takes a configuration which, at a minimum dictates which ZeRO stage you want to use: 1, 2, 3, or 3-offload (stage 3 with CPU offloading).

Each stage offers increasing memory reduction given that they work together additively (II): stage 1 shards the optimizer states across devices; stage 2 also shards gradients; stage 3 also shards model parameters. Stage 3-offload offloads the optimizer states, gradients, and parameters to the CPU. Of course, each memory saving technique incurs a cost of throughput.

ZeRO stage 1 mimics data parallelism communication, where there needs to be an all-reduce of the gradients on the backward pass. In stage 2, because both gradient and optimizer states are sharded, the backward pass all-reduce becomes a reduce-scatter of the gradients. The reduce-scatter is a communication optimization because it is mathematically identical to doing an all-reduce and then sharding the result gradient the GPU is responsible for, but less data is transferred. Finally, in stage 3, the forward pass now needs to do an all-gather of parameters along with the previous stages backward pass communication. Note, stage 2 and stage 3 is doing a reduce-scatter on every microstep, whereas stage 1 delays the communication of the all-reduce until the optimizer step. More implementation details about ZeRO-specific configurations can be found in Appendix H.

C. Single GPU Baselines

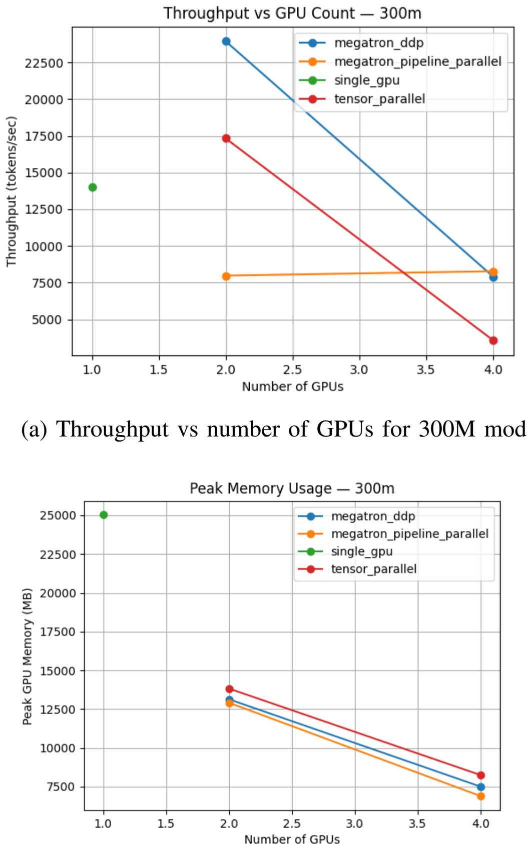
To generate the *GPTModel* baseline we used our *tensor_parallel.py* experiment using PyTorch's *torchrun* command with `--nproc-per-node=1`. This sets the world size and the tensor parallel size to 1.

For the ZeRO baselines, we had a separate python script with an identical training loop to the ZeRO experiment which we run directly with *python*.

IV. Experimental Results with analysis

While all of the training strategies are exceptional in their own right, they each rely on highly optimized infrastructure to achieve their full potential. Full-mesh GPU interconnects like NVSwitch that provide high-bandwidth GPU communication come at high cost: upwards of tens of thousands to sometimes hundreds of thousands of dollars, often not available to all GPUs; the more expensive systems require the higher end GPUs. Without significant investment, most research is inherently limited to 3rd and 4th generation PCIe connections. In most cases, this provides manageable GPU-to-GPU peer communication latency. However, PCIe bridges are sometimes needed to facilitate communication, which naturally incurs a greater latency overhead, see Appendix D.

Our results show that on relatively cost-accessible commodity hardware, data parallelism (including ZeRO) on 2 devices shows consistent near 80% throughput scaling efficiency across medium to large models, while the rest of the strategies on 2 devices provide no better than 60% throughput scaling efficiency and at times significantly worse. The throughput efficiency of most distributed training strategies collapses on 4 devices, compared to the 2 device experiments and the single GPU baselines, indicating that communication overhead dominates computation at this scale. On the smallest model, 10M parameters, all distributed training strategies across 2 or more devices always perform poorly. This is because the model does not computationally stress the GPUs enough. Distributed training strategies are only beneficial when compute outweighs communication. That said, like the other models, using 4 GPUs showed considerably worse performance. In the case of ZeRO, the 4 GPU case does not work well for small models (10M, 100M) but becomes viable with mid-to-larger sized models (≥300M).



(a) Throughput vs number of GPUs for 300M model.

(b) Peak memory usage vs number of GPUs.

Fig. 1: Megatron scaling summary for the 300M model.

A. Tensor, Data, and Pipeline Parallelism in Megatron

Fig 1a demonstrates the 4 GPU collapse of any given distributed strategy's throughput. Results for the 300M model are shown, but all model sizes exhibit similar patterns (see Appendix B). Tensor and data parallelism at least provide some benefit on 2 GPUs. Pipeline parallelism performs poorly on 2 GPUs but remains consistent moving to 4 GPUs. On 2

GPUs, tensor parallelism shows ∼60% throughput efficiency which is not ideal, while data parallelism exceeds 80%. While data parallelism performs better, both show that there is some benefit to using 2 GPUs in a PCIe constrained setup. 4 GPUs only offers negative scaling and slow training times.

We also show the effect of each strategy on peak memory (Fig 1b). This metric is more important to highlight because if one over-optimizes for the average memory usage of a strategy, they are guaranteed to incur out-of-memory errors. Peak memory occurs early on in the training loop before stabilizing. This was true across all runs. Tensor and pipeline parallelism showed dramatically lower peak memory as expected, given how they decompose the model weights, showing even more dramatic peak memory savings at 4 GPUs.

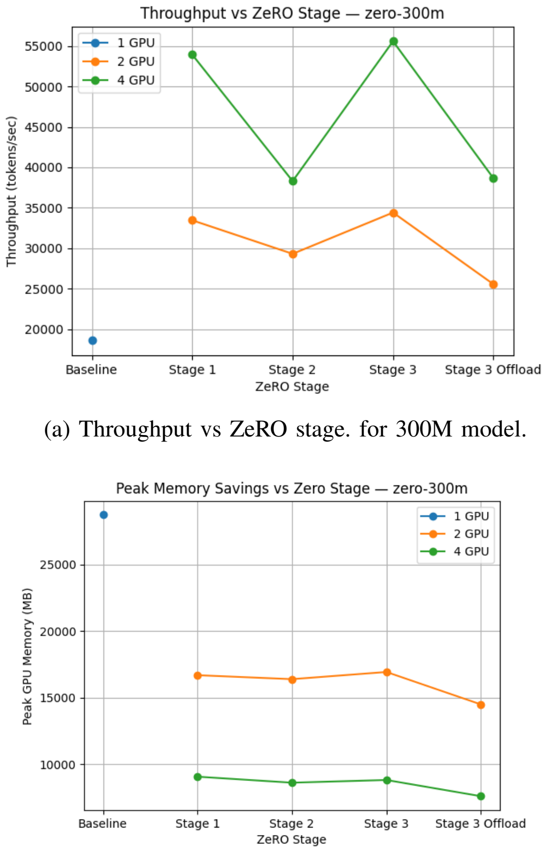
While one would expect a higher memory footprint from data parallelism, due to its replication of the model, the lower memory footprint is due to the fact that we split the batch size by world size in order to keep the total number of tokens in the experiment consistent. By decreasing the batch size, this decreases activations and ultimately less memory overhead. This speaks to the substantial effect activations and batch size also have on peak memory. The reduction we observe reflects activation memory scaling rather than reduced model replication, which remains unchanged under data parallelism.

B. Pipeline Parallelism Microbatch Analysis

Pipeline parallelism performance is dependent on two things: microbatch count and balancing compute load across the divided layers. While there's minimal control of the load balancing, Megatron provides very explicit control of the microbatch count, which directly affects the bubble overhead [4], [15]. Our main experiments ran with a configuration that maximized the bubble overhead, the worst-case configuration. See Appendix G-C for more details about pipeline parallelism in Megatron and how microbatches affect the bubble overhead.

With our experimental setup, we were able to do an extra handful of pipeline parallelism runs to determine if its poor performance was PCIe related or due to our maximum bubble overhead configuration. We found that GPU utilization stayed mostly the same. On 2 GPUs, the throughput behaved as expected: using a microbatch size of $m = 4p$ consistently outperformed the maximum bubble $m = p$ configuration. However, to our surprise, the maximum bubble configuration performed better than the recommended one on 4 GPUs on the medium sized 100M and 300M model parameter models. We believe this is due to the fact that with more devices and more microbatches, there is a multiplicative effect on the amount of communication overhead that PCIe amplifies; alternatively, setting the microbatch size to the minimal allowed amount also minimizes the amount of communication needed. See Appendix B-D for the how the different microbatch sizes behave on 2 and 4 GPUs.

C. ZeRO Results



(a) Throughput vs ZeRO stage. for 300M model.

(b) Peak memory usage vs ZeRO stage.

Fig. 2: ZeRO scaling summary for the 300M model.

Unlike Megatron's data parallelism behavior, the effect of PCIe is less prominent at mid-to-large models when using ZeRO. Fig 2a shows the general throughput pattern exhibited by models across the different stages ZeRO offers. Results for the 300M model are shown, but all models ≥300M exhibit a similar pattern (see Appendix C-A)

1) Stage 1 offers more throughput compared to the single GPU baseline.
2) Stage 2 diminish slightly on 2 GPUs and more prominently on 4 GPUs.
3) Stage 3 offers comparable throughput to Stage 1.
4) Stage 3-offload is comparable or worse than Stage 2 throughput.

Before reviewing the observed throughput behavior, it would be useful to first discuss peak memory behavior. While average GPU memory shows consistent savings as we progress through the stages (see Appendix C-D), peak memory behaves differently. Stage 1 consistently shows a meaningful drop in peak memory compared to the single GPU baseline. Stage 1 shards the optimizer states, which is usually the most memory intense in training loops. Stage 2 and 3 do not further significantly reduce peak memory relative to stage 1. Stage 2 shards the gradients but this does not impact peak memory given that all of the same memory allocations from stage 1 will occur in stage 2. Stage 3 shows a slight increase from stage 2, likely due to the overhead of memory buffers that need to be allocated for the forward pass's all-gather. Stage 3-offload once again shows more peak memory savings due to offloading all of the sharded data onto the CPU.

While stage 1 shards the optimizer states, it still mirrors traditional data parallelism and we see an increase in throughput relative to the number of devices. In our experiments, throughput of any stage on 4 GPUs consistently exceeds 2 GPUs. This behavior differs from Megatron's data parallelism. While we cannot compare metrics directly, the difference in behavior is noteworthy. We believe this is likely due to differences in model implementation and communication patterns between Megatron's *GPTModel* and the PyTorch-based model used with ZeRO.

In Stage 2, throughput drops for both 2 and 4 GPUs. Earlier we mentioned that Stage 2 switches from an all-reduce in the backward pass to a reduce–scatter as a communication optimization, but this optimization only reduces the total number of bytes communicated per device. As described by Rajbhandari et al. [9], ”*...as each gradient of each layer becomes available during backward propagation, we only reduce them on the data-parallel process responsible for updating the corresponding parameters...gradients corresponding to different parameters are reduced to different processes. To make this more efficient in practice, we use a bucketization strategy, where we bucketize all the gradients corresponding to a particular partition, and perform reduction on the entire bucket at once.*”

In other words, when gradients are sharded across devices, they are usually grouped into multiple gradient buckets and Stage 2 performs one reduce–scatter per bucket. This contrasts with Stage 1, which issues fewer and larger all-reduce operations per backward pass. Issuing many reduce–scatter calls likely increases the communication overhead on PCIe, which we believe contributes to Stage 2's reduced throughput. The number of gradient buckets can be controlled with DeepSpeed's *reduce_bucket_size* parameter. We used the default `5e8`. This parameter and a number of other factors affect the number of buckets, but the number of buckets is not equal to world size, so ZeRO stage-2 will usually require more reduce-scatter calls than stage 1 all-reduce calls.

While stage 3 has the same communication requirements in the backward pass as stage 2, it is likely that this communication and the parameter all-gather communications are better overlapped within a single training step, across layers and gradient buckets. Stage 3 increases communication, yet appears to optimize and hide the communication overhead on the critical path with concurrent compute operations. Stage 3-offload once again diminishes in throughput due to the GPU → CPU → GPU round trip latency of offloading and reloading all the sharded data the data parallel process is responsible for to and from the CPU.

V. Conclusion & Future Work

A. Conclusion

We have shown that multiple training strategies simply cannot operate as advertised due to bandwidth limitations that are a reality when working with real world cost constraints. Under our tested configurations and PCIe topology, using NVIDIA's Megatron framework for tensor and pipeline parallelism is not a viable option under PCIe-constrained commodity interconnects when using more than 2 GPUs. Without the highly specialized interconnects, training time suffers drastically. While data parallelism in Megatron remains viable on 2 GPUs, ZeRO would be our preferred data parallelism approach: even though its 4 GPU scaling efficiency is $\sim$60%, it still provides some throughput benefits along with providing immediate memory relief using stage 1. The memory reduction has the added effect of allowing researchers and engineers to increase the size of the models they wish to experiment with, without increasing their hardware costs. That said, in most cases, 10M and 100M parameter models are simply too small to gain any meaningful benefit from distributed training techniques in a PCIe setting.

Lastly, the ZeRO stage 2 and 3 throughput results suggest that in PCIe-constrained environments with small microbatches, restructuring communication to reduce its presence on the training-step critical path can be more important for throughput than minimizing total communication volume. This observation highlights the importance of communication placement, a consideration that becomes a top priority for designers and users of distributed training frameworks operating under PCIe constraints.

B. Future Work

During our experiments, many questions came up regarding these training strategies and the effect of various system and model parameters on their behavior.

- How does 3D [10], which requires at minimum 8 GPUs, behave in a PCIe environment?
- Does an optimal configuration for a single strategy translate directly to optimality in the context of 3D?
- Do optimal ”per strategy” configurations translate directly in a hybrid strategy setting, e.g. combining Megatron's tensor and pipeline parallelism?
- Could Megatron's *DataDistributedParallelConfig* be configured in a way that provides better performance in PCIe environments?
- Could ZeRO's *reduce_bucket_size* be tuned to give better Stage 2 performance? If large enough, could it approach Stage 1 performance?
- Could our ZeRO microbatches be too small and unfairly penalizing Stage 2 performance?
- ZeRO stage 3 offers more parameters for tuning: how sensitive is memory, communication, and GPU utilization for these different parameters?
- Could different kinds of pipeline parallelism load balancing have any meaningful effect on throughput?

- PCIe vs NVL - When is a training strategy no longer viable on PCIe and NVL becomes a requirement for training?
- As the model sizes grow, so does the training sequence length. We were very careful with our batch sizes to ensure we could still fit a 1B model in memory. How does sequence length interact with parallelism strategy choice?
- While we did not have time, a more granular analysis using PyTorch profiler results focused on communication and its overlap with compute could provide insight into total communication time versus optimized kernels that overlap communication and compute
- We saw a difference in data parallelism behavior between Megatron and our ZeRO experiments. Would pipeline parallelism also behavior differently using GPipe in PyTorch vs using the Megatron framework in a PCIe environment?
- PyTorch now supports *FullyShardedDataParallel* which has ZeRO-like features. How does this design/implementation fare in a PCIe environment?
- Could different architectures be better suited for PCIe environments, e.g. MoE?
- How sensitive is each training strategy to batch size changes?

References

[1] A. Grattafiori et al., “The Llama 3 Herd of Models,” arXiv preprint arXiv:2407.21783, July 2024.

[2] T. Scao et al., “BLOOM: A 176B-Parameter Open-Access Multilingual Language Model,” arXiv preprint arXiv:2211.05100, November 2022.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, I. Polosukhin, “Attention is All You Need,” arXiv preprint arXiv:1706.03762, June 2017.

[4] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. Chen, D. Chen, H. Lee, J. Ngiam, Q. Le, Y. Wu, Z. Chen, “GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism,” arXiv preprint arXiv:1811.06965, November 2018.

[5] C. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, G. Dahl, “Measuring the effects of data parallelism on neural network training,” arXiv preprint arXiv:1811.03600, November 2018.

[6] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” arXiv preprint arXiv:1810.04805, October 2018.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” OpenAI Blog: https://openai.com/index/better-language-models, February 2019.

[8] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, “Megatron- LM: Training Multi-Billion Parameter Language Models Using Model Parallelism,” arXiv preprint arXiv:1909.08053, September 2019.

[9] S. Rajbhandari, J. Rasley, O. Ruwase, Y. He, “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models”, arXiv preprint arXiv:1910.02054, October 2019.

[10] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, B. Catanzaro, “Using DeepSpeed and Megatron to Train Megatron- Turing NLG 530B, A Large-Scale Generative Language Model,” arXiv preprint arXiv:2201.11990, January 2022.

[11] NVIDIA, “Python Bindings for the NVIDIA Management Library,” https://pypi.org/project/nvidia-ml-py/.

[12] NVIDIA, “Run Simple Megatron Core Training Loop,” https://github.com/NVIDIA/Megatron-LM/blob/v0.12.0rc3/examples/run_simple_mcore_train_loop.py.

[13] NVIDIA, “Write Your First Training Loop,” https://docs.nvidia.com/megatron-core/developer-guide/0.15.0/user-guide/index.html#write-your-first-training-loop.

[14] J. Ren, S. Rajbhandari, R. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, Y. He, “ZeRO-Offload: Democratizing Billion-Scale Model Training”, arXiv preprint arXiv:2101.06840, January 2021.

[15] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, M. Zaharia, “Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM,” arXiv preprint arXiv:2104.04473, April 2021.

Appendix A
ZeRO Configurations

ZeRO experiments use the same model dimensions but fix the global batch size to 16 for all model sizes and use a reduced sequence length of 512 for the 1B model to avoid out-of-memory errors

TABLE IV: Model configurations for ZeRO experiments

Model Size	d_{model}	n_{heads}	n_{layers}	d_{ff}	**Seq Len**	**Batch Size**
10M	320	8	4	1536	128	16
100M	768	12	12	3072	512	16
300M	1024	16	24	4096	1024	16
500M	1280	20	24	5120	1024	16
1B	1536	24	36	6144	512	16

The following appendices provide detailed plots supporting the results discussed in Section V.

Appendix B
Megatron Plots

A. Throughput

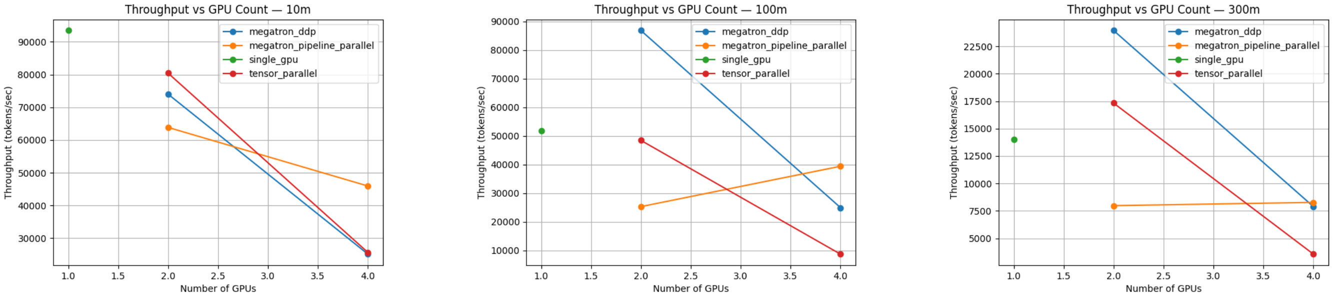


Fig. 3: Megatron throughput vs number of GPUs (10M–300M models)

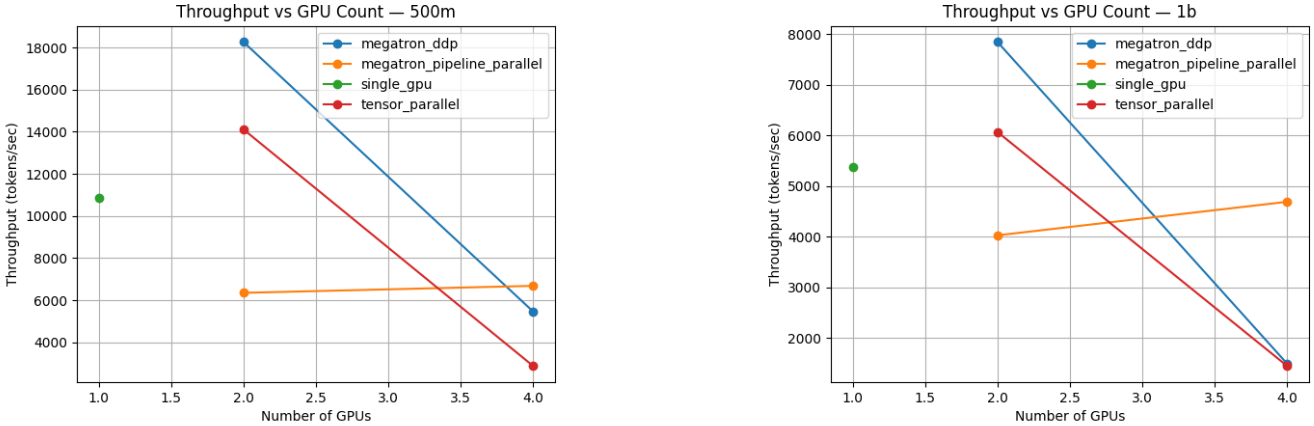


Fig. 4: Megatron throughput vs number of GPUs (500M–1B models)

B. Throughput Efficiency Plots

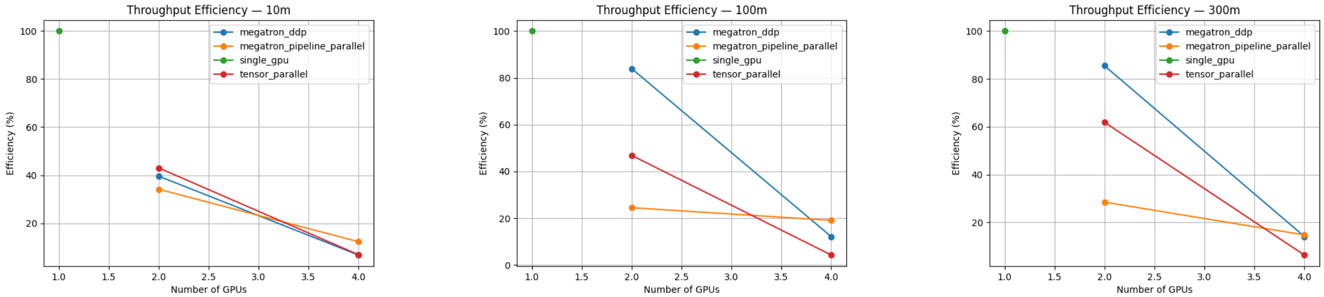


Fig. 5: Megatron throughput efficiency vs number of GPUs (10M–300M models)

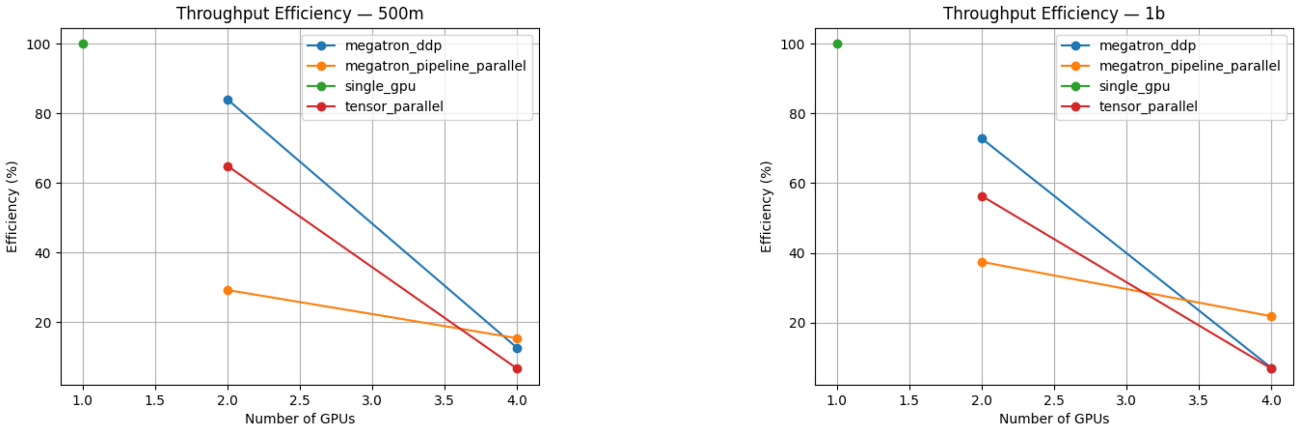


Fig. 6: Megatron throughput efficiency vs number of GPUs (500M–1B models)

C. Peak Memory Usage Plots

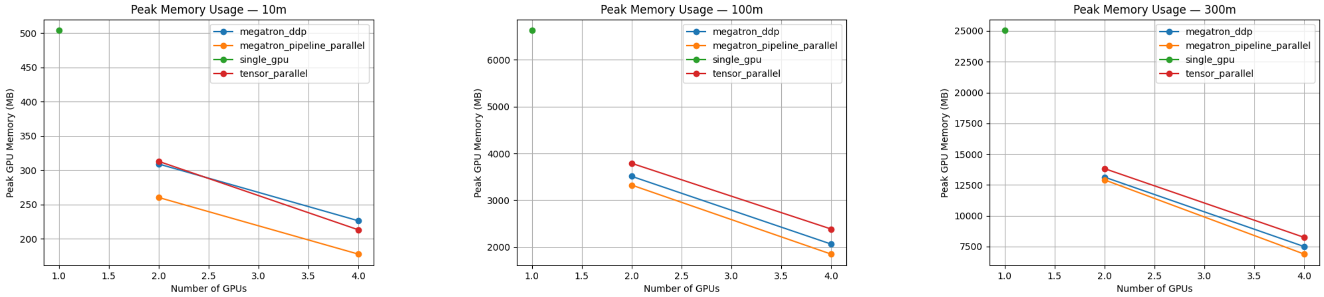


Fig. 7: Megatron peak memory vs number of GPUs (10M–300M models)

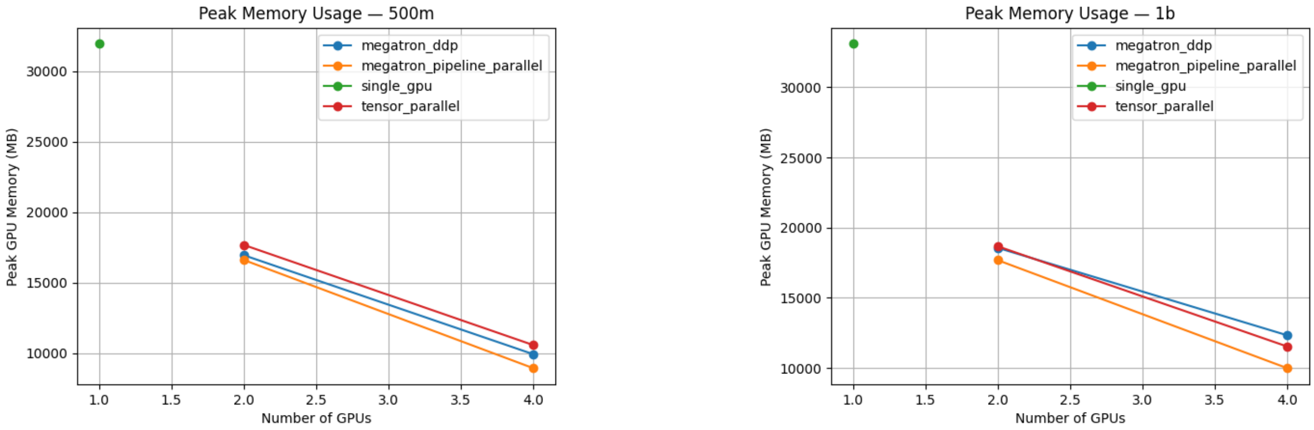


Fig. 8: Megatron peak memory vs number of GPUs (500M–1B models)

D. Pipeline Parallelism Analysis Plots

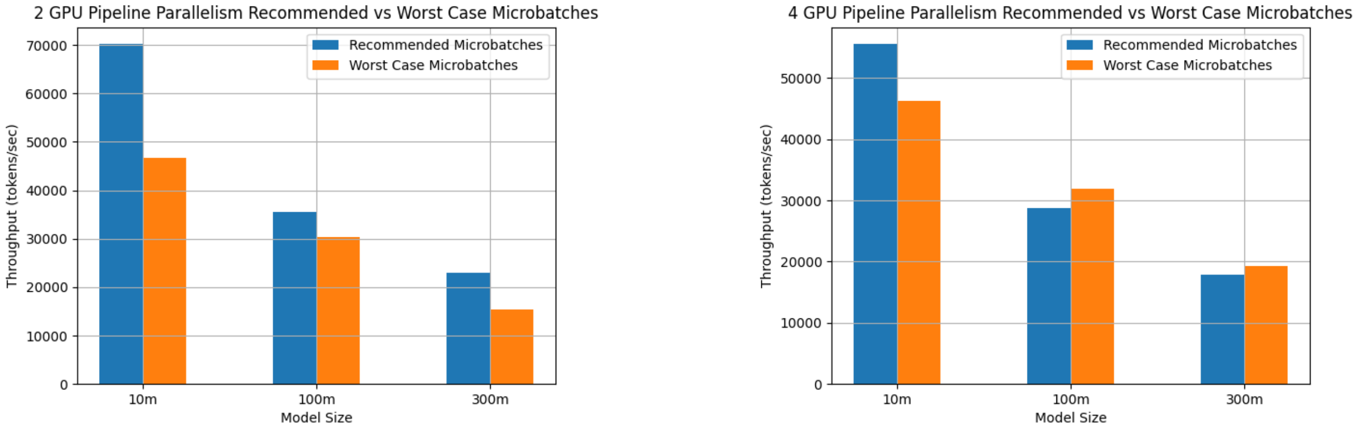


Fig. 9: Pipeline parallelism throughput: recommended vs worst-case microbatch configurations

Appendix C
ZeRO Plots

A. Throughput

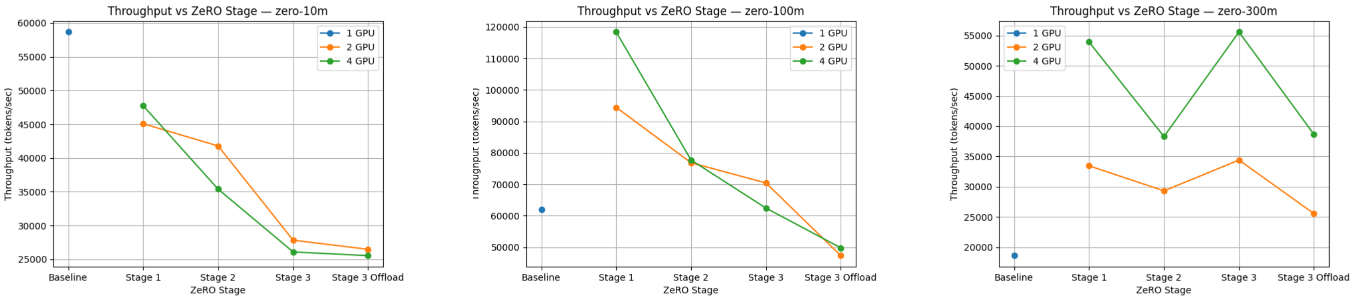


Fig. 10: ZeRO throughput vs stage (10M–300M models)

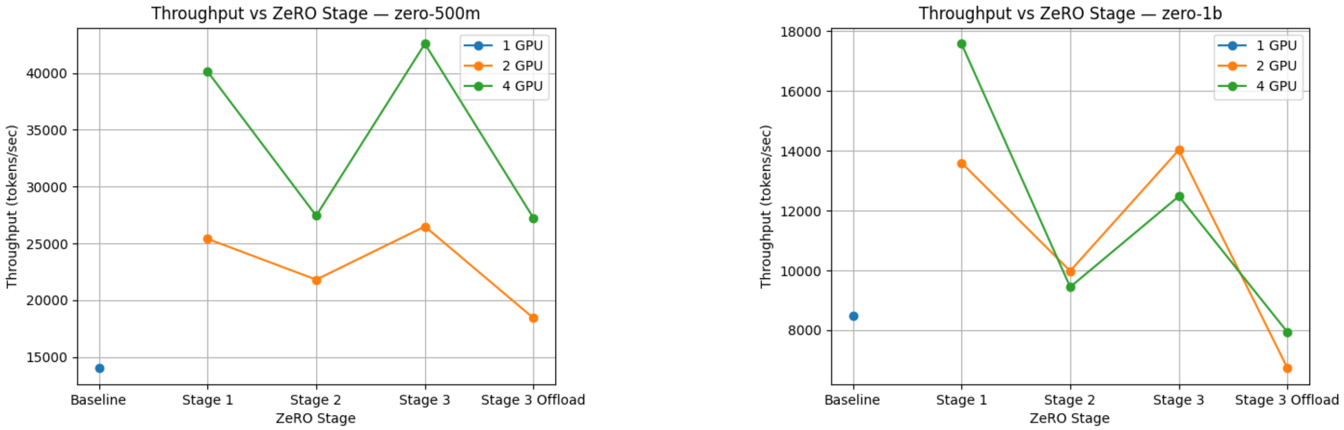


Fig. 11: ZeRO throughput vs stage (500M–1B models)

B. Throughput Efficiency Plots

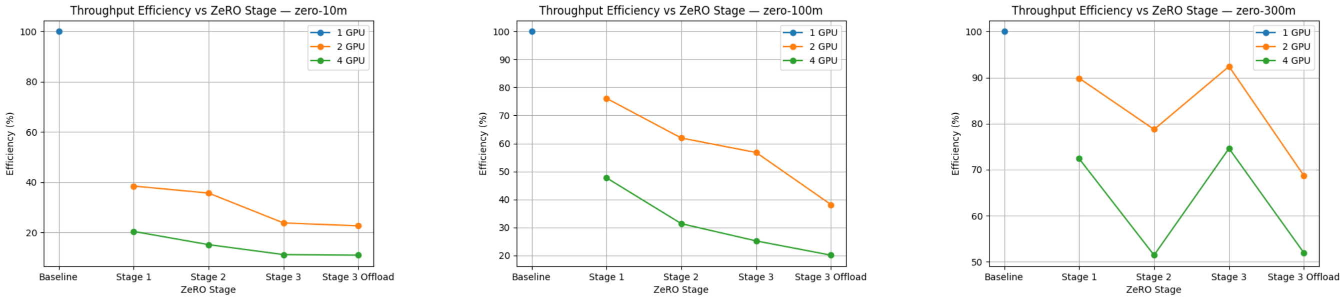


Fig. 12: ZeRO throughput efficiency vs stage (10M–300M models)

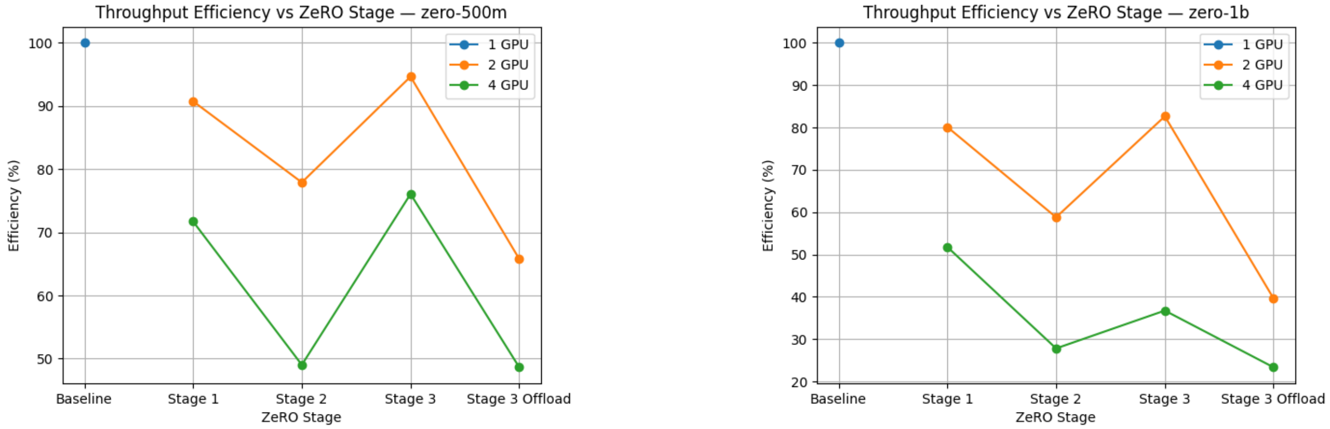


Fig. 13: ZeRO throughput efficiency vs stage (500M–1B models)

C. Peak Memory Usage

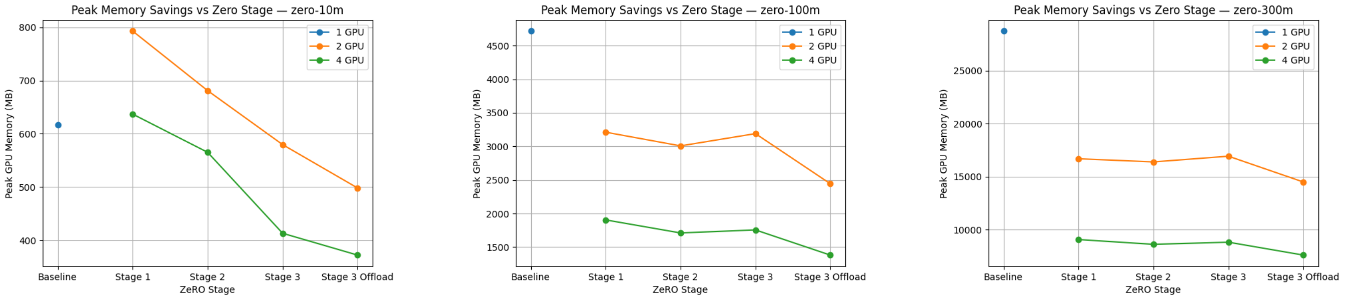


Fig. 14: ZeRO peak memory usage vs stage (10M–300M models)

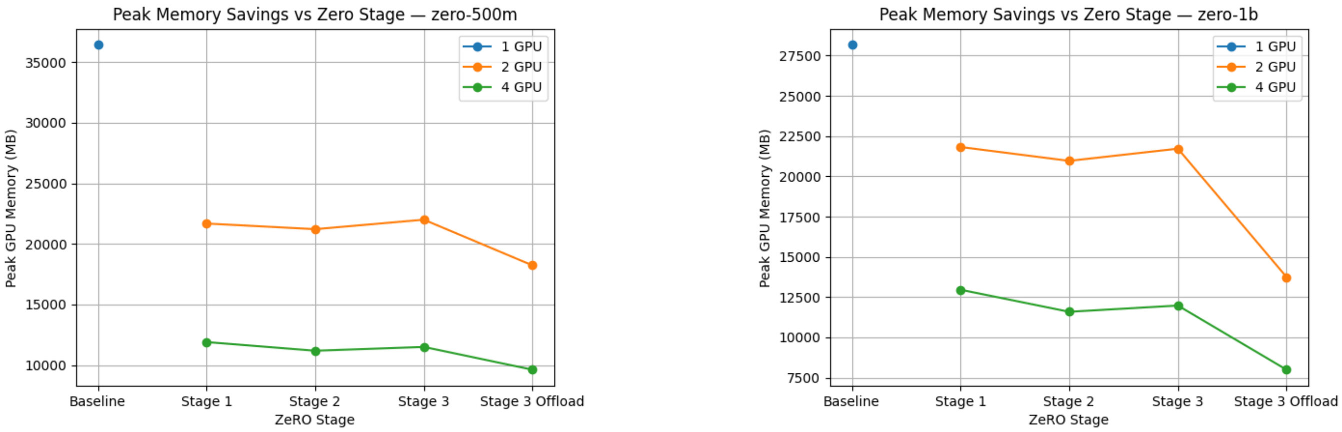


Fig. 15: ZeRO peak memory usage vs stage (500M–1B models)

D. Average Memory Usage

It may be expected that ZeRO should consistently reduce average/steady-state GPU memory usage, the fact is when world size is sufficiently small, e.g. 2, the sharded state(s) are likely not exceeding the activation memory. So while peak memory is meaningfully affected, average GPU memory benefits are seen at $\geq$ 4 GPUs.

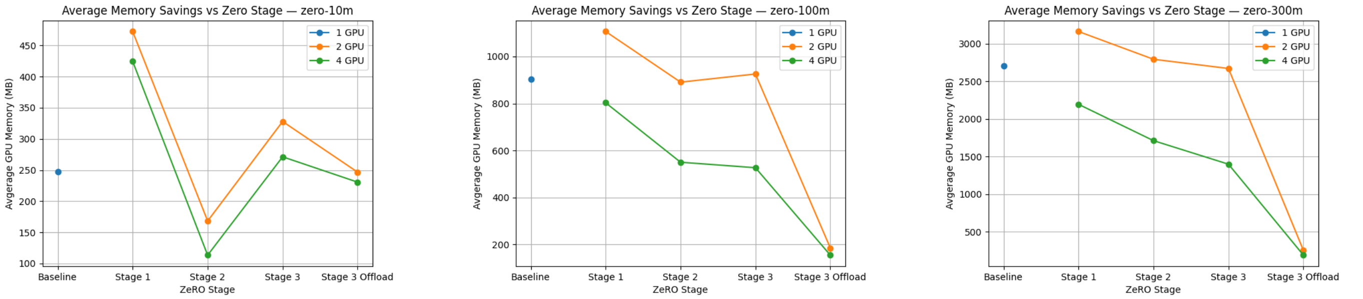


Fig. 16: ZeRO average memory usage vs stage (10M–300M models)

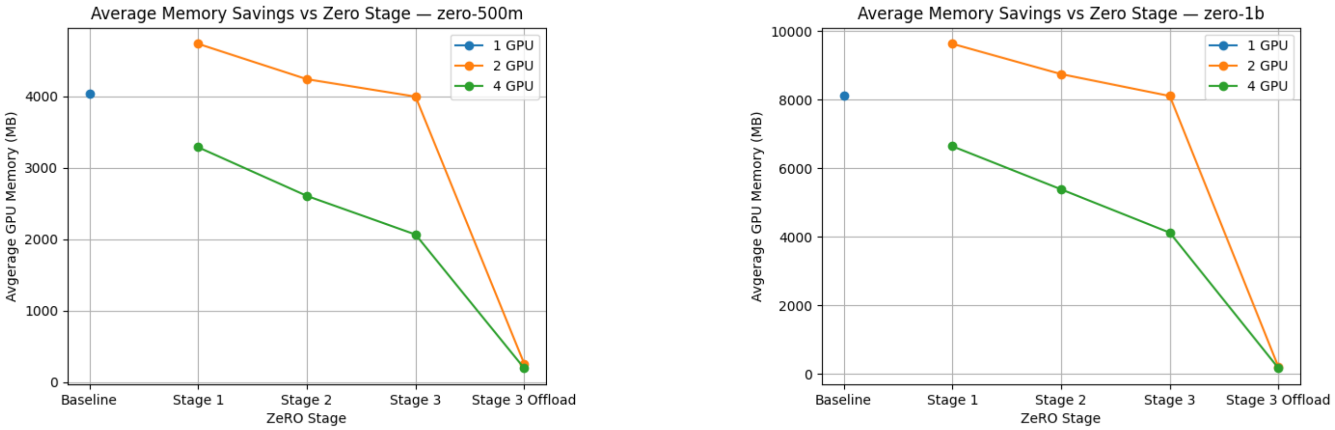


Fig. 17: ZeRO average memory usage vs stage (500M–1B models)

Appendix D
GPU Interconnect Topology

This is the interconnect topology of the RTX A6000 GPUs used in the RunPod.io cloud infrastructure environment for our 2 and 4 GPU experiments. In the environment, most GPUs must traverse multiple PCIe bridges rather than a single PCIe bridge on the same host, amplifying communication overhead, which could help explain why multiple strategies failed on 4 GPUs.

```
# nvidia-smi topo -m
    GPU0   GPU1   GPU2   GPU3   NIC0   NIC1    CPU Affinity  NUMA Affinity   GPU NUMA ID
GPU0  X     PXB    PXB    PXB    SYS    SYS     0-23,48-71         0             N/A
GPU1 PXB     X     PXB    PXB    SYS    SYS     0-23,48-71         0             N/A
GPU2 PXB    PXB     X     PIX    SYS    SYS     0-23,48-71         0             N/A
GPU3 PXB    PXB    PIX     X     SYS    SYS     0-23,48-71         0             N/A
```

Legend:
X = Self
SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
PIX = Connection traversing at most a single PCIe bridge

Appendix E
Training Loop and Metrics Capture

All experiments follow a similar workflow on every device:

1) Launch with *torchrun* and a model configuration reference (e.g. ”10M”, ”300M”)
2) Create a logger for the particular run
3) Configure the model with the given configuration reference
4) Initialize any required distributed protocol mechanisms
5) Prepare a data loader with our synthetic dataset
6) Initialize the optimizer
7) Run a training loop for 200 steps
8) Every iteration, take a snapshot of GPU memory and utilization

After 200 iterations, total throughput, average GPU memory usage, peak GPU memory, and average GPU utilization per device is logged. These per-device results are the foundational metrics that all of our results are derived from. Finally, 8 more steps are executed using PyTorch’s profiler, to monitor all CPU and GPU activity. While our experiments still generate the traces for more granular measurements, we were unable to incorporate this data due to time restrictions.

We took care early on to make sure that the log file paths were clear and that the per device logs could be easily aggregated. This was achieved by using properties of the experiment as nested directories. The files containing raw metrics followed the `cuda_N.log` naming pattern, where *N* is the device’s assigned ”rank.” Using the model size, strategy, and a unique timestamp for the folder structure allowed for clearly identifiable experimental results. The folder structure template is `/log/path/<strategy>/<model_size>/<timestamp>/`, e.g. `megatron_ddp/10m/1234567890/cuda_1.log`

In the training loop, all experiments follow the same general pattern:

1) Zero out gradients of the optimizer
2) Synchronize devices via torch.cuda's *synchronize()*
3) Run a forward pass
4) Compute loss
5) Run the optimizer's step
6) Synchronize devices once again via torch.cuda's *synchronize()*

Appendix F
Metrics Aggregation Derivations

As mentioned in §III, experiment results were logged in a very specific manner: it guarantees that all *cuda_N.log* files belong to the same experiment. We accomplished consistent aggregations by implementing 3 data classes

1) *TrainingResults* - the metrics we log after 200 iterations, per device.
2) *ExperimentSummary* - aggregate metrics given a list of *TrainingResults* from a single experiment
3) *ComparisonSummary* - comparative metrics given an *ExperimentSummary* and a target baseline *TrainingResults*

§III went over the per device metrics that would be populated in *TrainingResults*. Our final results represent a combination of data from the *ExperimentSummary* and *ComparisonSummary* aggregates. We now provide more depth into *ExperimentSummary* and *ComparisonSummary*.

1) ExperimentSummary: As mentioned across the different strategies in §III, the computation of "total tokens" is strategy dependent: in the case of data parallelism and ZeRO, all devices see different tokens, so we take the sum; for pipeline parallelism, the last device contains the "total" (the others have 0, so we can still take the sum); for tensor parallelism, all devices see the same tokens, so we choose the max of all devices, which has the same effect as picking a single device.

The rest of the aggregates are defined as:

- Total time: the maximum time across the set of *TrainingResults*
- Total throughput: still total tokens ÷ total time, as just defined in this context
- Average GPU memory: the average across the *TrainingResults* average GPU memory values
- Total GPU memory: the sum across all *TrainingResults* average GPU memory values
- Peak GPU memory: the maximum peak across the set of *TrainingResults*
- Average GPU utilization: the average across the *TrainingResults* average GPU utilization values

2) ComparisonSummary: Given an *ExperimentSummary* of multiple devices for a specific experiment type and a specific single-GPU baseline, we will now define the following comparative metrics:

- Relative runtime overhead: how much slower or faster the multi-GPU run is compared to the single GPU run
- Throughput and throughput scaling efficiency
- Peak memory usage

"Relative runtime overhead" tells us how much slower or faster the multi-GPU run is compared to single GPU. We also compute throughput scaling efficiency

$$(\frac{TotalTime_{MultiGPU}}{TotalTime_{SingleGPU}} - 1) \times 100$$

When >0%, this indicates that the multi-GPU strategy is slower than the single GPU. When the overhead percentages is <0%, then the multi-GPU strategy is operating faster than the single GPU. This captures effective end-to-end overhead, including overlapped communication and computation of the strategy/model/ framework/etc. This metric is not meant to be a scaling metric. We introduce it as a direct way to measure wall-clock speedup or slowdown.

While Shallue et al. measured batch sizes [5], it sets the basis for computing scaling throughput and efficiencies. The ideal scaling throughput is straightforward, representing the throughput expected under perfect linear scaling: $Throughput_{SingleGPU} \times N_{devices}$. To compute the throughput efficiency, we first need the throughput scaling factor

$$\frac{Throughput_{MultiGPU}}{Throughput_{SingleGPU}}$$

The interpretation of the throughput scaling factor is

- 1 means the strategy has the same speed as a single GPU
- <1 means the strategy is slower than a single GPU
- >1 means the strategy is faster than a single GPU

- ≈ the number of devices is equivalent to ideal scaling

Throughput scaling efficiency is then defined by

$$\frac{ThroughputScalingFactor}{N_{devices}} \times 100$$

When this percentage is 100%, it means the throughput achieves perfect linear scaling. Efficiency values ∼80% are desirable whereas values <70% indicate poor throughput performance.

We also computed a number of memory scaling factors: average GPU memory, total GPU memory, and peak GPU memory. All of them are computed in the same manner

$$\frac{MemoryType_{MultiGPU}}{MemoryType_{SingleGPU}}$$

We interpret these metrics similar to the throughput scaling factor, such that values less than 1 indicate less memory usage than a single GPU, values larger than 1 indicate more memory usage, and values near 1 indicate comparable memory usage.

Appendix G
Megatron Implementation Details

Following NVIDIA's example [12], the first thing any Megatron driven experiment does is tear down any previous parallel states. Our experiments only ever run one type of Megatron model-parallel grouping, so this is more a matter of correctness and defensiveness. Next, we initialize the distributed process group to use the NCCL backend. Now, with each device aware of the world size and their respective rank, the specific model-parallel grouping can be initialized for the particular Megatron parallelism (Table V). By setting tensor or pipeline parallel size to 1 effectively disables it from the experiment.

At this point the *GPTModel* can be initialized with a corresponding *TransformerConfig*. The initialization values are driven by the configuration reference provided as an argument to the experiment runner. The *TransformerConfig* handles defining the hidden size, number of attention heads, and the number of layers. The *TransformerConfig* is then given to the *GPTModel* along with the vocabulary size and maximum sequence length.

One other important note is how Megatron's training script configures the forward and backward passes. Normally, in PyTorch, the model is called with an input batch, the logits are returned, and a loss function is applied against the result and the expected values. The idea is similar in Megatron, but they have a wrapper you must use (*forward_back_func*) in which you must provide the model, the data iterator, seq len, "micro batch size" (in the case of tensor parallelism, this is the global batch size; data parallelism is $batch\_size \div worldsize$) and a *forward_step_func* you define. This callable takes the data iterator and an instance of your model as arguments. This is where you call the model to get a resulting "output" tensor. You then return this output tensor and a python partial of your loss function and the loss mask, which later, Megatron will call to allow you to compute the loss. This loss function finally returns what your main training loop sees as the loss. This wrapping is likely due to how Megatron is sharding the model across devices and how/when these functions can be called in order to get a meaningful result back to the main training loop.

The three experimental files that leverage the Megatron *GPTModel* are

- `tensor_parallel.py`
- `megatron_pipeline_parallel.py`
- `megatron_ddp.py`

These scripts were heavily inspired by the `run_simple_mcore_train_loop.py` [12] in NVIDIA's Megatron User Guide [13]. This was necessary given how drastically differently Megatron prefers to operate compared to PyTorch.

TABLE V: Megatron strategy configurations

Parallelism	Tensor Parallel Size	Pipeline parallel size
Data	1	1
Tensor	world size	1
Pipeline	1	world size

The following sections describe the slight differences of each Megatron strategy.

A. Tensor Parallelism

This was the first experiment we built using the *GPTModel*. One might also notice that it is the only file that deviates from the `megatron_*.py` naming convention. This was because we originally started experimenting with the other strategies on the lightweight transformer decoder model first. When we realized how drastically different Megatron-core behaved and how we could not simply ”wrap” our model with Megatron, we pivoted to an ”all Megatron” approach, given the ”out of the box” support for the other distributed training strategies[4].

In retrospect, tensor parallelism was one of the most straightforward strategies to implement, once we understood how Megatron-core prefers to configure models and how it handles the forward and backward passes. This experiment uses a PyTorch *DataLoader* with batch size set to the global batch size, per the experiment configuration. Every device must see the identical inputs at every step: each layer is sharded across multiple devices and thus still needs the entire input set to be able to process the forward pass. The tensor parallel size is set to the world size while the pipeline parallel size is set to 1 for all tensor parallelism experiments (Table V). The Megatron framework handles how the weights of a layer are sharded across the devices, but they do support some level of configuration as to how how the weights can be sharded.

For tensor parallelism to function given the weight sharding, forward and backward passes require communication:

- Forward: activations are communicated via all-gather or reduce-scatter
- Backward: gradients are communicated via all-reduce or reduce-scatter

The communication type is dependent on the layer type.

When computing ”total tokens” for throughput across multiple devices, because all devices see the same batch size, but each step requires that all devices process the batch, we can take the total number of tokens from any device in the experiment.

B. Data Parallelism

Megatron follows a very similar pattern to PyTorch's data parallelism approach; they both provide a *DataDistributedParallel* wrapper that wraps your model to support data parallelism. The primary difference is Megatron has a separate *DataDistributedParallelConfig* that provides many features to tweak and optimize how the data parallelism behaves. One important note is that Megatron provides a ”Distributed Optimizer” which you can enable in the configuration. This has the effect of making the strategy behave similar to ZeRO-1 where the optimizer states is sharded across devices. We made sure not to use this feature, to ensure that the data parallel experiment behaves in a traditional manner.

Aside from this, the main difference between the tensor and data parallelism experiments is the data loader used and its configured batch size. When doing distributed data parallelism, a PyTorch *DistributedSampler* must be used and provided to the data loader. The data loader's batch size is set to $batch\_size \div world\_size$. This has the effect of lower memory (fewer activations are stored compared to a single GPU baseline) and, in the case of smaller models, reducing GPU utilization. This guarantees that the total number of tokens seen by the ”world” matches the total seen by the single GPU baseline, given that each device is processing a unique set of tokens. For data parallelism, Megatron is configured with both tensor parallel and pipeline parallel sizes set to 1 (Table V).

Unlike tensor parallelism, there is no communication necessary on the forward pass, as each device has the entire model replicated and can operate completely independently of the other. However, on the backward pass, an all-reduce on each parameter's gradients is required so that all devices apply the same optimizer update.

While we split the global batch size across the number of devices in data parallelism, each device processes a unique set of tokens, so when we compute total tokens for the strategy, we actually take the sum of ”total tokens” across all devices.

C. Pipeline Parallelism

Pipeline parallelism is a strategy that differs quite a bit from the other two discussed strategies. First, pipeline parallelism splits the layers in their entirety across available devices, e.g. layer 0 and 1 are on device 1, layer 2 and 3 are on device 2, etc. The grouping of layers are called ”stages.” The Megatron framework handles how to separate the stages, although they do support customizing the stages[5]. However, this is only for the transformer layers; you must define which device will handle the embeddings and the output layer. We always make the first rank responsible for embeddings and the last rank responsible for the output layer.

Second, pipeline parallelism operates on ”microbatches” of the original batch size. For both the forward and backward pass of training, each device trains the model on these microbatches sequentially, i.e. device 1 sees the first microbatch, then, while device 1 sees the second microbatch, device 2 begins work on the first microbatch. After all devices complete the forward pass, the same sequence is done in reverse for the backward pass.

This inherently requires a scheduling algorithm. The first of such algorithms was GPipe [4] but Megatron does not use this. Instead, the scheduler used is 1F1B [15][6]. The primary difference between 1F1B and GPipe is

- GPipe: All devices do forward microbatches of batch, then all devices do backward microbatches of batch.
- 1F1B: After reaching a ”steady-state” of microbatches at the start, the algorithm alternates ”one forward, one backward” pass until the end of the batch.

[4] With the exception of ZeRO, where we were forced to reuse our lightweight model, given ZeRO and Megatron's incompatibility.

[5] We originally had a PyTorch implementation of GPipe with the *SimpleTransformerDecoder* model. PyTorch allows much more flexibility with how to load balance stages, but that also comes with its own complexity cost - getting it right is not easy.

[6] This is the default scheduler provided, which we used for our experiments. There is another interleaved 1F1B variant available.

For the purposes of our study, this has no effect, as both GPipe and 1F1B are subject to the same "bubble overhead" ratio: $(p-1)/m$ where p is the number of stages and m is the number of microbatches. In theory, pipeline performance increases when $m >> p$, but more practically the recommendation is generally $m \approx 4p$. Conversely, the configuration that maximizes the bubble overhead is $m = p$, which we later realized was the configuration we used in our experiments. With this in mind, we ran another set of pipeline parallel experiments on 10M/100M/300M parameter models with a fixed sequence length of 512 and batch size of 32 across all model sizes to determine if the performance results were due to our configuration or because of the PCIe infrastructure. The 500M and 1B parameter models were not included, as they generated out-of-memory errors with these sequence length and batch size values. We modified these values to guarantee the configuration generated enough compute related stress on the GPU to make communication beneficial and give the strategy the best opportunity to demonstrate its capabilities. §IV-B shows the results of this experiment.

Given that each device is involved in the backward and forward pass, both passes require communication. That said, pipeline parallelism only requires that each device communicates with its immediate neighboring rank.

- On the forward pass, the i_n device sends activations to device i_{n+1}.
- On the backward pass, the i_n device sends gradients to device i_{n-1}.

We only compute the total number of tokens seen by the strategy at the last rank, as this rank is responsible for starting the backward pass. Our logic was that the moment the backward pass begins on the last rank, then all tokens have flowed from beginning to end. All previous ranks have their total tokens value set to 0. The global batch size is preserved across microbatches; increasing the number of microbatches reduces per-microbatch size but does not change total tokens processed per optimizer step.

Appendix H
ZeRO Configuration Reasons

During the initial set of ZeRO experiments, we observed that memory consumption appeared nearly unchanged across ZeRO stages for smaller models. To further stress the optimizer-state sharding behavior and make memory differences more apparent, we increased the global batch size used by all ZeRO configurations to 16. Achieving this global batch size on 2 and 4 GPU configurations required setting *gradient_accumulation_steps* (GAS) to 4, which produced per-GPU microbatch sizes of 2 (2 GPUs) and 1 (4 GPUs). This adjustment also required maintaining a separate set of ZeRO configurations from the one's used with in our Megatron models. Note that ZeRO's "microbatches" are unrelated to pipeline parallelism's microbatches.

We later discovered a bug in our initial ZeRO training loop: we were executing $200 \times GAS$ iterations instead of the planned 200 optimizer steps. After fixing the step-counting logic, we reran all ZeRO experiments using the same configurations (e.g., batch size, GAS) without modifying any hyperparameters. All results reported in this paper reflect the corrected implementation.