Optimizing Distributed Training on Frontier for Large Language Models

Sajal Dash
Oak Ridge National Laboratory
dashs@ornl.gov

Isaac R Lyngaas
Oak Ridge National Laboratory
lyngaasir@ornl.gov

Junqi Yin
Oak Ridge National Laboratory
yinj@ornl.gov

Xiao Wang
Oak Ridge National Laboratory
wangx2@ornl.gov

Romain Egele
Université Paris-Saclay
romainegele@gmail.com

J. Austin Ellis
AMD
austin.ellis@amd.com

Matthias Maiterth
Oak Ridge National Laboratory
maiterthm@ornl.gov

Guojing Cong
Oak Ridge National Laboratory
congg@ornl.gov

Feiyi Wang
Oak Ridge National Laboratory
fwang2@ornl.gov

Prasanna Balaprakash
Oak Ridge National Laboratory
pbalapra@ornl.gov

***Abstract*—Large language models (LLMs) have demonstrated remarkable success as foundational models, benefiting various downstream applications through fine-tuning. Loss scaling studies have demonstrated the superior performance of larger LLMs compared to their smaller counterparts. Nevertheless, training LLMs with billions of parameters poses significant challenges and requires considerable computational resources. For example, training a one trillion parameter GPT-style model on 20 trillion tokens requires a staggering 120 million exaflops. This research explores efficient distributed training strategies to extract this computation from Frontier, the world's first exascale supercomputer. We enable and investigate various model and data parallel training techniques, such as tensor parallelism, pipeline parallelism, and sharded data parallelism, to facilitate training a trillion-parameter model on Frontier. We empirically assess these techniques and their associated parameters to determine their impact on memory footprint, communication latency, and GPU's computational efficiency. We analyze the complex interplay among these techniques and find a strategy to combine them to achieve high throughput through hyperparameter tuning. We have identified efficient strategies for training large LLMs of varying sizes through empirical analysis and hyperparameter tuning. For 22 Billion, 175 Billion, and 1 Trillion parameters, we achieved GPU throughputs of 38.38%, 36.14%, and 31.96%, respectively. For the training of the 175 Billion parameter model and the 1 Trillion parameter model, we achieved 100% weak scaling efficiency on 1024 and 3072 MI250X GPUs, respectively. We also achieved strong scaling efficiencies of 89% and 87% for these two models. We trained these models only tens of iterations instead of training till completion.**

I. Introduction

Large language models (LLMs) leverage an attention mechanism to learn language structure and can generate natural language responses to many prompts. Once trained on a large corpus of text, these models can be fine-tuned to perform many downstream tasks; thus, LLMs are very successful as foundational models. Recent studies demonstrated that LLM models with a large number of parameters outperform LLM models with a smaller number of parameters [1]. Large LLMs such as GPT3-175B [2], BLOOM-176B [3], OPT-175B [4], and Turing NLG-530B [5] have shown remarkable success as foundational models and outperform their smaller counterparts in many NLP tasks. Some studies also reported the loss scaling law, which states that an LLM model can keep learning from data up to 20x-200x of its parameter count [1], [6], [7]. Training large models using large data requires a tremendous amount of computing resources. Cost and energy-efficient utilization of these resources is always challenging.

These models' success stories demonstrate that open-sourced large models can serve as state-of-the-art foundation models. With the advent of RedPajama datasets with one Trillion and 30 Trillion tokens [8], and the Dolma dataset with three Trillion tokens [9], [10], a model of size 1 Trillion parameter must be within the horizon. A rough estimate [11] tells us that training a Trillion parameter model on 1-30 Trillion tokens will require $6 \times 10^{12} \times [1-30] \times 10^{12} = 6-180$ Million exa-flops (floating point operations).

This paper details our experience training such large LLM models with billions to trillion parameters on Oak Ridge National Laboratory's (ORNL) Frontier supercomputer, one of the world's most advanced HPC systems. Central to our narrative is the acknowledgment of HPC systems as more than mere facilitators of computing resources. The Frontier supercomputer, powered by advanced AMD GPUs, represents a paradigm shift in computational capabilities. However, training AI models at the trillion-parameter scale introduces unique challenges. These include balancing the extreme computational demands with memory constraints and optimizing inter-node communication to mitigate performance bottlenecks. Our research provides a detailed analysis of these challenges and the strategies employed to overcome them, offering insights into the intricacies of large-scale model training in an HPC environment.

Large language models often hit GPU memory walls, and training a trillion parameter model requires approximately 20 Terabytes of memory. So, to fit this model, we need to break it

down into parts and distribute them across hundreds of GPUs. LLMs are transformer models whose shapes are determined linearly by the depth (number of layers) and quadratically by the width (hidden dimension). Various model parallelization approaches distribute the model along these two dimensions. We can also use data parallelism to speed up training by utilizing more GPUs for training on large datasets.

Tensor parallelism proposed by Megatron-LM [12] partitions the layer weights (Tensors), performs computation on the smaller matrices, and combines the activation results. This approach splits the model across the width dimension. Pipeline parallelism breaks the model across layer dimensions and places groups of layers on individual GPUs. The micro-batches are consumed in a pipelined fashion, and backward and forward propagation of these micro-bathes are overlapped so that the communication latency can be hidden [13]–[15]. The most traditional way for data parallelism is to replicate the entire model across GPU groups and train these replicas in parallel while averaging the loss after every forward pass. A novel direction of data parallelism, namely sharded data parallelism, achieves data parallelism by sharding the model parameters across available memory, reducing the amount of resources required for the model.

Megatron-DeepSpeed [16] supports tensor, pipeline, data, and sharded data parallelism. Megatron-LM supports the first three. DeepSpeed ZeRO [17] and Fully Sharded Data Parallelism (FSDP) [18] support sharded data parallelism. Since models with trillion parameters require many modes of parallelism, Megatron-DeepSpeed is the most complete framework in terms of the different modes it supports. We explore this framework to find an optimal strategy through an investigative understanding of the complex interplay between these modes. However, these frameworks are primarily developed to run on NVIDIA GPUs and have yet to be tested extensively on a large scale or run on AMD platforms. So, we performed a feasibility study of running these frameworks on Frontier, ported the framework to Frontier to identify the limiting issues and opportunities, and prepared a training workflow with an optimized AI software stack.

After porting the Megatron-DeepSpeed framework to Frontier, we focus on finding the best strategies to train large models on Frontier using a combination of these parallelizations. The next challenge for a particular model is what combinations of these modes we should select and to what extent. For example, using tensor parallelism for wide models is beneficial. Still, because this incurs frequent communication after every layer, expanding tensor parallelism beyond the GPUs within a single node is not advised. For pipeline parallelism, the pipeline bubble (a measure of GPU idle time (Section II-C) can become an issue, making the communication latency a bottleneck. Finding the right pipeline parallelism parameters is crucial for hiding communication latency. So, we explore these modes and their right configuration in isolation and in combination.

In our work, we delve into the specifics of how these tools are optimized for Frontier's AMD GPU architecture. This involves an in-depth exploration of their adaptability in managing extensive computational loads and memory optimization techniques necessary for training trillion-parameter models. The trade-offs between memory, compute, and communication are critical in the HPC context. Our research examines the interplay of these elements in the Frontier environment. We investigate how adjustments in distributed model training frameworks can be finely tuned to leverage the full potential of AMD GPUs, focusing on achieving an optimal balance between these components to maximize training efficiency and model accuracy.

A pivotal aspect of our study is the exploration of pipeline parallelism, tensor parallelism, micro-batch size, and gradient accumulation steps. These elements are fundamental in distributed training, particularly at the scale of trillion-parameter models. Our research provides a detailed examination of how each component is optimized for Frontier's infrastructure, focusing on how these parallelism strategies can be effectively implemented in an HPC setting to enhance computational efficiency and reduce training times.

In this paper, our primary focus lies in improving the training performance of these large-scale LLMs, particularly emphasizing the computational aspects and efficiency of various training strategies. It is important to note that our objective was not to train these models to completion for the purpose of achieving the highest possible accuracy. Instead, our approach was centered around understanding and enhancing the performance characteristics of training processes on HPC systems. This involved exploring the scalability, efficiency, and resource utilization of different training methodologies, especially in the context of the Frontier supercomputer's capabilities. We seek to gain insights into how different parallelization techniques, model configurations, and system architectures impact the training dynamics of LLMs with billions to trillion parameters. Through this exploration, we sought to contribute practical strategies, which could aid in the efficient training of large-scale models in future research and practical applications.

A. Paper Outline

Section II discusses various distribution strategies and cost evaluation of training large LLMs on Frontier. Section III provides an empirical analysis of multiple distribution strategies and associated parameters. We identify some valuable observations for training a 22B model from our experiments. In Section IV, we report hyperparameter tuning for training a 175B model to understand the combinations of these distribution strategies. Section V combines the lessons from Sections 3 and 4 and performs further experiments to devise a training recipe for 175B and 1T models. In that section, we also report GPU throughput, three different-sized models, and strong and weak scaling performance.

B. Contributions

The contributions of the paper are:

1) Distributed training techniques on AMD Hardware with ROCM software platform: This work contributes to

enabling state-of-the-art distributed training algorithms and frameworks for large LLMs on AMD hardware using the ROCM software platform. This advancement serves as a blueprint for efficient training of LLMs on non-NVIDIA and non-CUDA platforms, such as the AMD-powered Frontier supercomputer.

2) Development of an optimized distributed training strategy through hyperparameter search: The research presents strategies to effectively manage the GPU memory wall and communication latency in the training of LLMs with billions to trillions of parameters. By performing empirical analysis and hyperparameter search we identified a strategy that combines model parallelism techniques, such as tensor parallelism and pipeline parallelism, along with data parallelism to efficiently train large models of size 175 billion and 1 trillion parameters on Frontier. This study demonstrates how to optimize memory usage across multiple GPUs and minimize the communication latency.

II. Distributed Training Techniques and Frameworks

Distributed training of large language models has seen many innovations and advances in recent times. Much of the focus on large language models has been on their ability to create more accurate models using an ever-increasing number of parameters. Due to this focus, models have become too large to fit in a single GPU's memory, accelerating the research into model parallelism techniques. Tensor parallelism, pipeline parallelism, and sharded data parallelism are the most popular techniques for model parallelism.

A. GPT-style Model Architecture and Model Sizes

Transformer models can consist of two distinct parts, a stack of encoder blocks and a stack of decoder blocks (Figure 1) [19]. Encoder blocks help capture non-causal self-attention where each token in a sentence can attend to tokens from left and right. On the other hand, decoder blocks are useful for capturing causal self-attention, where a token can attend to only past tokens in the sequence. In a recent body of works, the encoder part has been co-opted for building BERT-like [20] models, which are useful for classification and regression types of work. On the other hand, the decoder block has been used for GPT-like [21] models for generative tasks.

The simplest GPT-like models consist of a stack of similar layers. Each layer has one attention block, followed by a Feed Forward Network (FFN) (Figure 2). The attention block has three sets of parameters $W_K, W_Q, W_V \in \mathbb{R}^{d \times d}$, where d is the hidden dimension of the models. K, Q, and V matrices required for attention computation are calculated by multiplying the input $X \in \mathbb{R}^{s \times d}$ with these weights ($K = XW_K$, $Q = XW_Q$, $V = XW_V$), here s is the input sequence length. The FFN block has two layers, with weights $W_1 \in \mathbb{R}^{d \times 4d}$ and $W_2 \in \mathbb{R}^{4d \times d}$. So, a layer contributes to $11d^2$ parameters. For multi-head head attention, we use h as the number of attention heads.

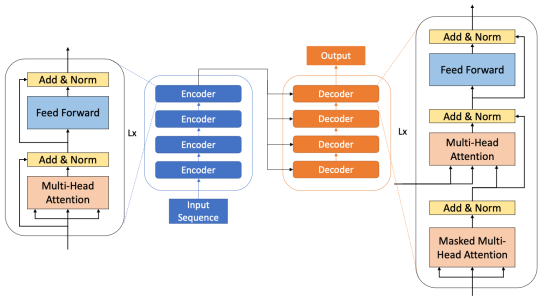


Fig. 1: Transformer architecture.

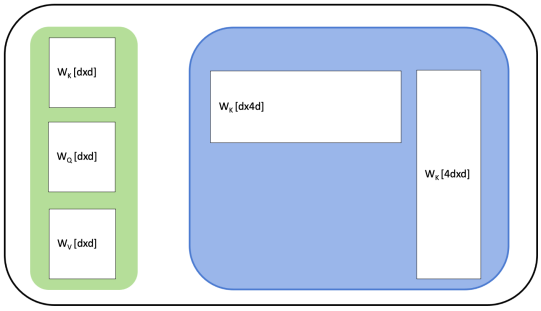


Fig. 2: Model parameters of a GPT layer.

With the embedding layer at the beginning of the model, the number of parameters becomes roughly $12Ld^2$, where L is the number of layers, and d is the hidden dimension. With this formula, we can define three models with sizes 22B, 175B, and 1T in Table I.

Model	#Layers	Hidden size	#Attention heads
1.4B	24	2114	24
22B	48	6144	48
175B	96	12288	96
1T	128	25600	128

TABLE I: Architecture specification of GPT-style LLMs.

Most memory requirements come from model weights, optimizer states, and gradients. However, the memory required for forward activation can also become significant depending on the batch size. In mixed precision training, we need 6 bytes for each model parameter, 4 to save the model in fp32, and 2 to use in computation in fp16. We need 4 bytes per parameter for Optimizer states to save the momentum in fp32 (Adam Optimizer). We need to save one fp32 gradient value for each parameter. So, in a mixed precision training with Adam optimizer, the minimum memory requirement is listed in Table II.

Each Frontier node has 8 MI250X GPUs[1], each with 64 GB of HBM memory. So from Table I's memory requirement, we can conclude that model parallelism is necessary to fit even

[1]Each node has 4 MI250X cards, and each card has two Graphics Compute Dies (GCDs) or effective GPUs. Hence, going forward, we will use the term GPU to refer to the GCDs. Each GPU has a theoretical fp16 peak of 191.5 TFLOPS.

Values	Memory Requirement		
	22B Model	175B Model	1T Model
Parameters (6x)	132 GB	1050 GB	6 TB
Gradients (4x)	88 GB	700 GB	4 TB
Optimizer States (8x)	176 GB	1.4 TB	8 TB
Total Memory (20x*)	440 GB	3.5 TB	20 TB

TABLE II: Memory Requirement for Training a 22B, 175B, and 1T Models in mixed precision. We have not included the activation memory or the overhead memory for various distribution frameworks.

one replica of the model. Model parallelism can happen in the hidden dimension via Tensor and Sharded data parallelism or in the layer dimension via Pipeline parallelism.

B. Tensor Parallelism

Tensor parallelism splits the weight tensor of a layer along the row or column dimension [22]. The attention block has three tensors $K, Q, V \in \mathbb{R}^{s\times d}$ for a given layer. These tensors are split column-wise (Figure 3) and $K = [K_1, K_2, \ldots, K_h]$, $Q = [Q_1, Q_2, \ldots, Q_h]$, and $V = [V_1, V_2, \ldots, V_h]$ where $K_i, Q_i, V_i \in \mathbb{R}^{s\times d/h}$. For the FFN block, W_1 is split along column, and W_2 is split along row.

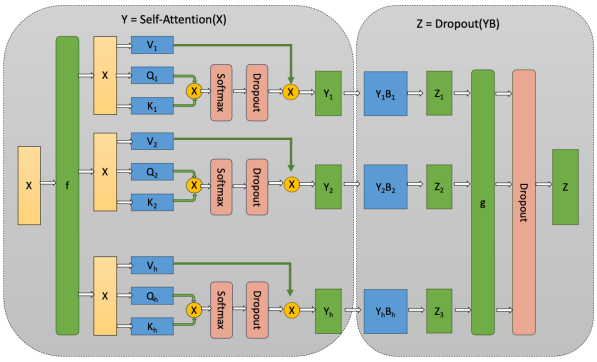


Fig. 3: Tensor parallelism on Attention block. V, K, Q tensors are split along the attention heads (column dimension). $V = [V_1, V_2, \ldots V_h]$, $Q = [Q_1, Q_2, \ldots Q_h]$, and $K = [K_1, K_2, \ldots K_h]$. B is split along its row-dimension.

For attention computation, the original formula is $softmax(\frac{KQ^T}{\sqrt{d}})V$. For each split, the partial attention is computed as $softmax(\frac{K_iQ_i^T}{\sqrt{d/h}})V_i$. These partial attentions are then appended along column dimension to find the final attention $A = [A_1, A_2, \ldots, A_h]$ where $A \in \mathbb{R}^{s\times d}, A_i \in \mathbb{R}^{s\times d/h}$ (Figure 3(b)).

For the FFN block, the computed attention goes through two FFN layers, where A is first multiplied by $W_1 \in \mathbb{R}^{d\times 4d}$, then the result is multiplied by $W_2 \in \mathbb{R}^{4d\times d}$. W_1 is partitioned along column dimension, whereas W_2 is partitioned along row dimension (Figure 3(b)).

C. Pipeline Parallelism

Pipeline parallelism splits the model into p stages, each having roughly L/p layers. Then, the batch is split into micro-batches, and at every execution step, one micro-batch is passed through a stage. Each stage is placed on a GPU. Initially, only the first GPU can process the first micro-batch. At the second execution step, the first micro-batch progresses to the second stage, and the first micro-batch can now go to the first stage. This is repeated until the last micro-batch reaches the last stage. Then, the backward propagation starts, and the whole process continues in the reverse direction. GPipe [13] proposes this simple method. Synchronization points are introduced after every batch to maintain the correct order of computation, requiring flushing pipeline stages. So, at the beginning and the end of a batch's processing, GPUs hosting earlier and later stages stay idle, resulting in wasted compute time or a *pipeline bubble*. The pipeline bubble fraction is $\frac{p-1}{m}$, where m is the number of micro-batches in a batch.

The simple GPipe scheduling creates a large pipeline bubble. Some additional methods are in place to reduce the pipeline bubbles. One such way is 1F1B scheduling proposed by PipeDream [14], wherein during the forward pass, initially, the micro-batches are allowed to flow forward until the last group receives the first micro-batch. But then the backward propagation of the first batch starts, and from then, a forward pass is always accompanied by a backward pass, hence the name 1F1B. An interleaving schedule has been proposed to reduce the bubble size even further where instead of placing one pipeline group on one GPU, multiple smaller pipeline groups are placed on a single GPU.

The pipeline bubble size for the 1F1B schedule is roughly p/m, where p is the number of pipeline groups, and m is the number of micro-batches. For the 1f1B schedule with interleaving, the bubble size is $\frac{p-1}{m\times v}$, where v is the number of interleaved groups placed on a single GPU.

D. Sharded Data Parallelism

Sharded data parallelism shards model parameters, optimizer states, and gradients and places one partition on each GPU [17], [18]. Since training advances one layer at a time, the computing devices need to have only one full layer and associated values (optimizer states, gradients, and parameters) in the memory. Sharded data parallelism takes advantage of this; before execution of a layer, that layer is materialized in all the GPUs by performing all gather across all the GPUs for that layer (Figure 4(a)). Now, all the GPUs have replicas of the same layer. Then, the layer is executed on different data batches on different GPUs. After that, each GPU deletes all the gathered parts of that layer and prepares for the next layer's materialization through all-gather. This way, it emulates data parallelism, but instead of every GPU hosting a complete replica of the entire model, it just hosts a replica of the currently active layer.

Sharded data parallelism can facilitate data parallel training of a large model across GPUs, even if the model is too large to fit in a single GPU's memory. DeepSpeed's ZeRO

optimizers [17] support sharded data parallelism in varying degrees. ZeRO-1 only shard optimizer states, ZeRO-2 shards gradients along with optimizer states, and ZeRO-3 shards optimizer states, gradients, and model parameters. On the other hand, PyTorch FSDP (Fully Sharded Data Parallelism) [18] shards all three and also supports a hybrid data parallelism by combining sharded data parallelism with traditional data parallelism.

E. 3D Parallelism and Megatron-DeepSpeed

Using only a single parallelism strategy to implement model parallelism can be an inefficient approach. For example, if we use only tensor parallelism to slice the model horizontally, the tensors can be too thin, requiring frequent all-reduce communication that can slow down the training. On the other hand, if we partition the model into too many pipeline stages, each stage will have tiny amounts of computation, which will require frequent communication. A known issue is that performing tensor parallel training across multiple nodes requires slow tree-like allreduce.

The use of multiple of these modes of parallelism in a hybrid fashion can minimize the areas of poor performance. 3D parallelism combines tensor, pipeline, and data (traditional and sharded) parallelism techniques to utilize resources. Through a proper setup, 3D parallelism can reduce communication latency by overlapping communication with computation. The standard code base for 3D parallelism used across the AI landscape is based on the Megatron-LM [22]. Megatron-DeepSpeed [16] extends on the features from Megatron-LM by adding DeepSpeed features such as the ZeRO-1 sharded data parallelism and a pipeline parallelism with overlapped 1F1B schedule. However, these standard codebases are all developed for NVIDIA GPUs and the CUDA platform.

F. Porting Megatron-DeepSpeed to Frontier

Megatron-DeepSpeed codebase is forked from NVIDIA's Megatron-LM codebase, and Microsoft then added DeepSpeed ZeRO optimizers, pipeline parallelism, and Mixture of Experts into this. NVIDIA develops Megatron-LM; hence, its codebase is developed with NVIDIA GPUs and CUDA environment as the target platform. Porting this codebase to run on the AMD platform presents some challenges.

1) CUDA Code: CUDA code doesn't run on AMD hardware; however, HIP, a CUDA-like C/C++ extension language, does. We converted the CUDA source code to HIP code using the `hipify` tool, built the shareable objects (so files) using `hipcc`, and then used `pybind` to make these shareable objects accessible from Python code.

2) DeepSpeed Ops: Most of the DeepSpeed ops are built during the execution of the training pipeline through JIT (Just in time) compilation. However, the JIT compilation of DeepSpeed ops didn't work on the ROCM platform, so we prebuilt all the ops when we installed DeepSpeed. We disabled all JIT functionalities from the Megatron-DeepSpeed codebase to avoid any runtime error.

3) Initialization of PyTorch Distributed Environment: Megatron-DeepSpeed utilized PyTorch Distributed initialization for creating various data and model parallel groups. This initialization process requires dedicating one compute node as the "master" node, and all the distributed processes require its IP address. We modify the codebase to accept $MASTER\_ADDR$ as an argument. We prepared a launch script to read the first node's IP address from the SLURM node list and pass this as an argument to all the processes launched using srun. The initialization code then uses this $MASTER\_ADDR$ for PyTorch Distributed initialization.

4) Libraries/Packages provided through ROCM Platform Software: We worked with AMD developers to get the ROCM version of some of the essential CUDA packages, such as APEX [23]. APEX is NVIDIA's mixed precision library, which is heavily leveraged by the Megatron-DeepSpeed code base for mixed precision training. We also adapted ROCM-enabled versions of FlashAttention [24] and FlashAttention2 [25] libraries for use with available compilers on Frontier. The Flash-Attention operations are ported to AMD GPUs using kernels from the Composable Kernel library [26].

III. Empirical Analysis of Various Distribution Strategy

In this section, we report our experiments exploring various distribution strategies and their optimal parameter values (Table III).

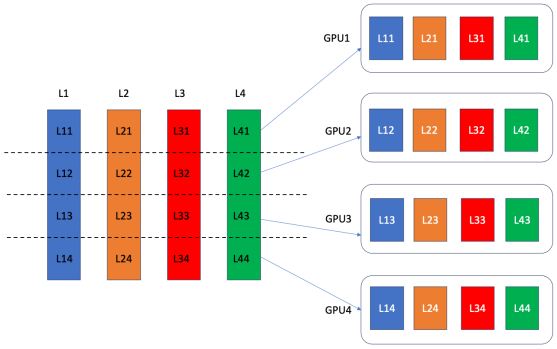
Distribution Strategy	Tunable Parameters
Tensor Parallelism	Tensor Parallel Size (TP)
Pipeline Parallelism	Pipeline Parallel Size (PP), #Microbatches (m)
Sharded Data Parallelism	ZeRO-1
Common	Micro Batch Size
Mixed Precision Training	FP16, BF16

TABLE III: Distribution Strategies and relevant tunable parameters

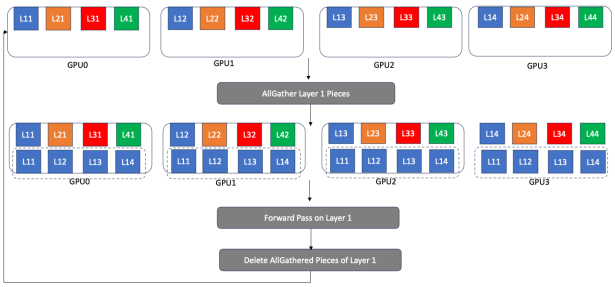
A. Tensor Parallelism

Tensor parallelism partitions model layers [22], and after every layer, the partial activation values need to be aggregated via allreduce. AllReduce after every layer execution is costly, and this depends on communication bandwidth between GPUs in a tensor-parallel group, communication volume which depends on hidden size and micro-batch size.

Figure 5 shows the communication bandwidth between Frontier GPUs. There are 8 GPUs in a node, and the GPUs in a single die are connected via four (50+50 GB/s) infinity fabrics. The bandwidth between GPUs across the die is half of it. But, the bandwidth between GPUs across nodes is 25+25 GB/s. So, from the network topology and configuration, TP = 2 would provide the fastest communication, and TP = 4 or 8 would be the second fastest. But, for TP > 8, the communication will happen over slower slingshot, and the communication will be much slower. So, keeping TP in $\{2, 4, 8\}$ should be the optimal strategy.



(a) Model is sharded vertically, and each shard is placed on a GPU.



(b) Sharded Data Parallelism, illustrated for the first layer.

Fig. 4: Sharded data parallelism.

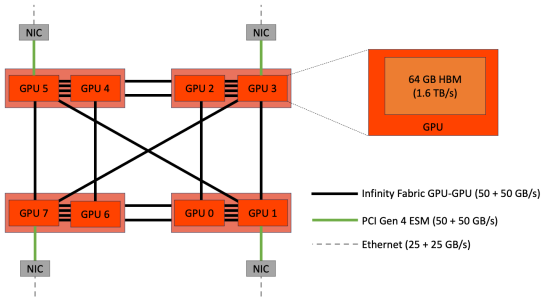


Fig. 5: Communication Bandwidth between GPUs in Frontier.

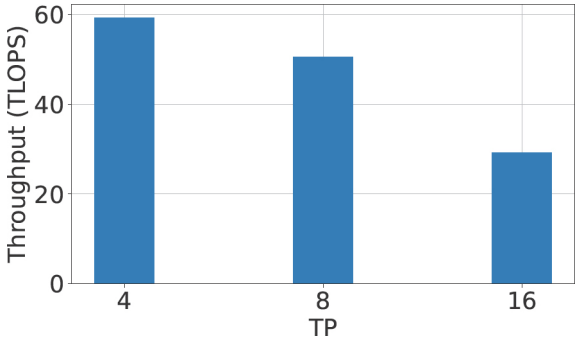


Fig. 6: GPU throughput vs TP for a 22B model.

We train a 1.4B model using 8 GPUs by varying TP from 1 to 8 and see that the smaller the value of TP, the higher the throughput (Figure 6).

Observation III.1 *Larger values of TP requiring inter-node communication deteriorate training performance.*

B. Pipeline Parallelism

Pipeline parallelism partitions the model along the layer dimension and groups consecutive layers into pipeline stages. The execution of a micro-batch flows from one stage to the next one.

A pipeline bubble would be a limiting factor for efficient training with this parallelism. The bubble size is roughly $\frac{TP}{v \times M}$, where TP is the number of pipeline stages, M is the number of micro-batches, and v is the number of overlapped pipeline stages on a single GPU. A large M can ensure the bubble size is minimal; however, it might need to utilize gradient accumulation. A large M results in a sizeable global batch size (GBS).

We saw the effect of large M or large GBS to see the impact on GPU throughput for two models of size 22B parameters and 1T parameters (Figure 7).

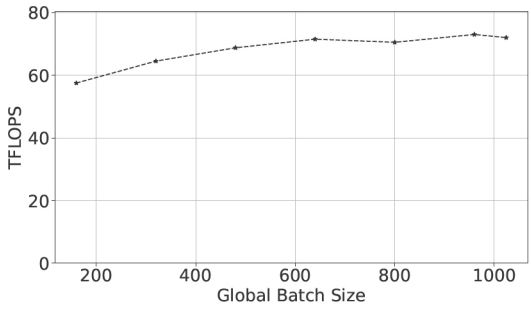
Observation III.2 *Saturating pipeline stages with large global batch sizes or many micro-batches minimizes pipeline bubble size.*

1) Impact of Number of Pipeline Stages: Next, we investigate the impact of the number of pipeline stages on training performance. Intuitively, more pipeline stages mean less computation before the communication happens. With a fixed global batch size (number of micro-batches), the pipeline bubble size increases with the number of pipeline stages. We also experiment with increasing the number of pipeline stages while keeping the $\frac{PP}{M}$ fixed by increasing the global batch size proportionally.

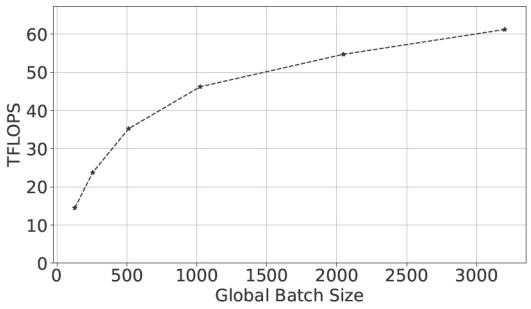
Observation III.3 *Increasing the number of pipeline stages while keeping the global batch size fixed increases the pipeline bubble size and deteriorates training performance.*

Observation III.4 *The training performance can be maintained with an increasing number of pipeline stages if the ratio of the number of pipeline stages to the number of micro-batches is kept constant.*

From the first experiment (Figure 8a), the training performance deteriorates with the increasing pipeline stages. However, scaling global batch size to fix the bubble ratio maintains the throughput (Figure 8b).

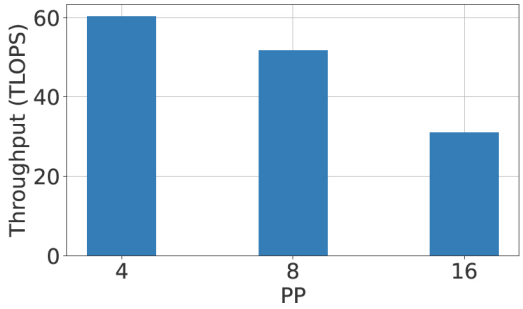


(a) Throughput vs. global batch-size for 22B model.

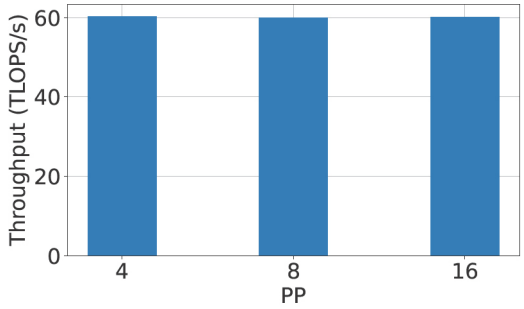


(b) Throughput vs global batch-size for 1T model.

Fig. 7: GPU throughput as a function of global batch size. Throughput increases with global batch size since it increases the number of micro-batches (M) and reduces the bubble size. However, using a large global batch size for a single replica has implications for how much data parallelism we can use and what part of the entire machine we will be able to use.



(a) Throughput vs. PP while keeping global batch size fixed at 128.



(b) Throughput vs. PP while scaling global batch size to keep the pipeline bubble ratio fixed.

Fig. 8: Impact of pipeline parallelism on training performance.

IV. Hyperparameter Tuning Using DeepHyper

DeepHyper is a Python package developed to automate the development of machine learning workflows with algorithms such as hyperparameter optimization (HPO) [27], neural architecture search [28], and uncertainty quantification [29]. Hyperparameters are the parameters of the optimized learning workflow that define it but cannot be inferred during the so-called "training" phase.

To find the best distributed training strategy determined by tensor, data pipelining, data sharding, and data parallelism settings, we use the asynchronous HPO from DeepHyper, which is based on a Bayesian optimization solver.

For the execution aspect, DeepHyper offers multiple parallelization schemes [30] (e.g., centralized or decentralized for the search execution; threads, processes, or MPI for the execution of our black box). We focused on a centralized architecture with processes for black-box evaluations, as our setting did not reach any parallelism bottleneck. Then, as each evaluation also requires a set of parallel resources (in particular, different nodes), we pass a queue of all nodes available to DeepHyper and ask each job to use 16 nodes. When DeepHyper suggests a hyperparameter configuration to be evaluated, a local process is created to schedule its evaluation; within this process, we set up the distributed training application given the passed hyperparameters and launch it through SLURM via `srun`.

We define the space of allowed hyperparameters through DeepHyper, which can be of categorical, discrete, or continuous types (a.k.a. mixed types). Table IV describes the hyperparameter space we used to tune the distributed training of a 175 Billion parameter model through FLOPs maximization. In this space, hyperparameter configurations can exist, triggering failures of the distributed training, such as memory exhaustion (out-of-memory). We handle such failures by catching the exception and returning the special `F`-objective value to DeepHyper, which internally penalizes such evaluations to discourage future evaluations.

Hyperparameters	Range
Pipeline-parallel-size (PP)	$PP \in \{1, 2, 4, 8, 12, 16\}$
Tensor-parallel-size (TP)	$TP \in \{1, 2, 4, 8\}$
Micro-batch-size (MBS)	$MBS \in [4, 20]$
Gradient accumulation steps (GAS)	$GAS \in \{5, 10\}$
ZeRO-1 Optimizer	$ZeRO - 1 \in \{True, False\}$
Number of Nodes (NNODES)	$NNODES \in \{12, 16\}$

TABLE IV: Hyperparameter Tuning for 175B Model

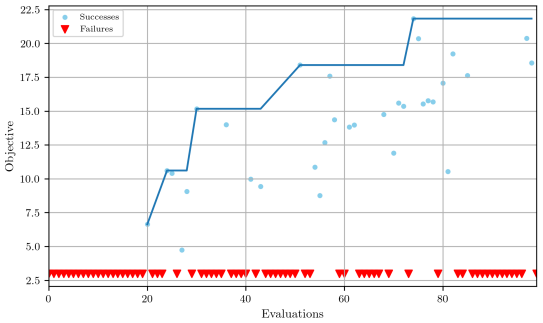


Fig. 9: Search Trajectory of DeepHyper search

We ran these hyperparameter tunings on 128 nodes, and each search job utilized 16 nodes. Each job picks 16 nodes from the queue. However, some of the jobs will use 12 nodes or 16 nodes. We dynamically create a srun launch command with these hyperparameters and submit the job for a maximum of 20 minutes.

In Figure 9, we present the results of our experiment. We observe many failures (red arrows), mostly out-of-memory errors. However, we observe that the frequency of such failures decreases with time. For successful evaluations, the best value of FLOPS improves with time to reach a final value of 22 TFLOPS.

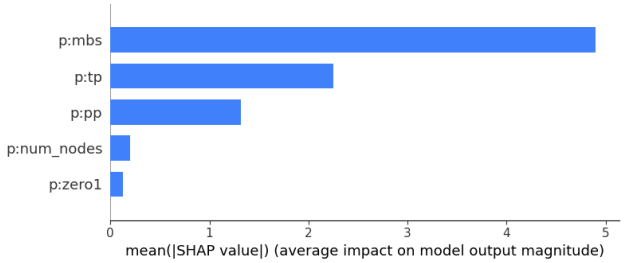


Fig. 10: Impact of various hyperparameters on training performance in terms of GPU throughput.

Using tuning run results of the DeepHyper search, we conduct SHAP (SHapley Additive exPlanations) [31] sensitivity analysis to assess the impact of the hyperparameters on the performance. SHAP is a game theory-based approach for explaining the output of machine learning models. It assigns each feature an importance value for a particular prediction. These values are derived from the Shapley values in cooperative game theory and provide a measure of the contribution of each feature to the prediction. SHAP values are instrumental because they are consistent and locally accurate, meaning they sum up the difference between the model and baseline outputs.

In the context of hyperparameter sensitivity analysis, SHAP can be used to evaluate the impact of different hyperparameters on performance by fitting a regression model that predicts the performance (outputs) with hyperparameters (inputs). In this analysis, various hyperparameters, such as micro-batch size (p:mbs), tensor-parallel-size (p:tp), pipeline-parallel-size (p:pp), number of nodes (p:num_nodes), and another parameter (p:zero1), are assessed to determine their influence on the FLOPS. When varying these hyperparameters, the SHAP values indicate the average impact on the model output magnitude. The bar chart in Figure 10 presents these hyperparameters' mean absolute SHAP values, providing insight into which parameters have the most substantial effect on the model's computational efficiency.

From the sensitivity analysis of all the hyperparameters (Figure 10), we find that the micro-batch size is the most impactful hyperparameter. Then, tensor-parallel-size, and pipeline-parallel-size. We see that utilizing ZeRO-1 has the least impact.

V. Training a Trillion Parameter Model

From the experiments, hyperparameter tuning, and analysis, we have identified an efficient strategy for training a Trillion-parameter model on Frontier by combining various distribution strategies and software optimizations. In this subsection, we list them with their contribution and best configurations (Table V).

A. An Efficient Strategy for Training a Trillion Parameter Model

Saturate Pipeline Stages by Increasing the Number of Micro-batches: We use pipeline parallelism provided by DeepSpeed (from DeepSpeed-Megatron, but not the Megatron's version). This pipeline parallelism algorithm is PipeDream's algorithm, where multiple stages are overlapped, and the 1F1B algorithm is followed to reduce the bubble size. However, the bubble size will increase if the pipeline stages are not saturated. To ensure saturation, the number of micro-batches must equal or exceed the number of pipeline stages.

Limit Tensor-Parallelism to A Single Node / Eight GPUs: Since the AllReduce operation is too frequent and needs to be performed for every layer, a layer spread across nodes causes tree-based AllReduction between GPUs across nodes, and the communication latency becomes a significant bottleneck.

Use Flash-Attention v2: We observed up to 30% throughput improvement using Flash-attention compared to the regular attention implementation.

Use ZeRO-1 Optimizer For Data Parallelism: We use ZeRO-1 for data parallelism to reduce memory overhead.

Use RCCL Plugin by AWS to Improve Communication Stability: AWS OFI RCCL plugin enables EC2 developers to use libfabric as a network provider while running AMD's RCCL-based applications. On Frontier, usage of this plugin shows communication stability.

B. Training Performance of a Trillion Parameter Model

From the lessons learned from hyperparameter tuning, we identified a set of combinations for models of size 22 Billion parameters and 175 Billion parameters. Encouraged by the GPU throughput of these two models, we finally trained a trillion parameter model using the combination of distribution strategies listed in Table V for ten iterations to see the

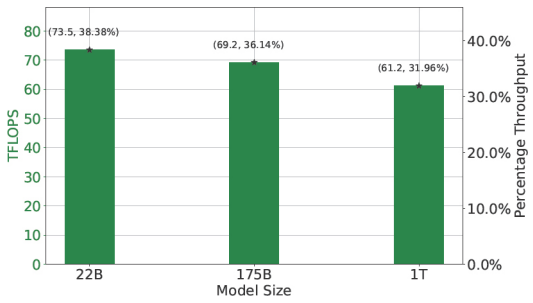


Fig. 11: MI250X Throughput for various model sizes. We report the hardware FLOPS, which are in close agreement with the model FLOPS.

training performance. For the 22B parameter model, we could extract 38.38% (73.5 TFLOPS) of its peak throughput (191.5 TFLOPS). For the 175B model training, we achieved 36.14% (69.2 TFLOPs) of peak throughput. Finally, for the 1T model, we achieved 31.96% (61.2 TFLOPs) of peak throughput 11.

Hyperparameters	Value	
	175B Model	1T Model
TP	4	8
PP	16	64
MBS	1	1
GBS	640	1600
ZeRO Stage	1	1
Flash Attention	v2	v2
Precision	fp16	fp16
checkpoint-activations	True	True

TABLE V: Best parameters for training a 175B model and a 1T model.

a) Composite Roofline Analysis: We collected the total hardware flops and bytes read and written throughout the training. From there, we computed the arithmetic intensities for training 22B and 175B parameter models to understand the limiting factors of the kernels. For these models, our achieved FLOPS were 38.38% and 36.14%, and arithmetic intensities of 180+. The memory bandwidth roof and the peak flops-rate roof meet close to the arithmetic intensity of 1. Hence, our training is not memory-bound.

C. Scaling Performance

Sustaining the performance of model-parallel training through data parallelism to engage a large number of GPUs in a system is a challenging task. Frontier GPUs are connected via communication links of various speeds, and stressing the larger part of the network can result in lost performance. So, we scale the training up to 1024 GPUs for a 175B model and 3072 GPUs for a 1T model through data parallelism to measure the scaling efficiency of our training strategy.

1) Weak Scaling: We perform a weak scaling experiment for the 1T model by performing data-parallel training on 1024, 2048, and 3072 GPUs using global batch sizes 3200, 6400, and 9600. The data-parallel training achieves 100% weak scaling efficiency (Figure 12(a)).

2) Strong Scaling: We perform strong scaling experiments by keeping the global batch size at 8000 and then varying the number of GPUs. We achieved 89.93% strong scaling performance for a 175B model on 1024 GPUs. We achieved 87.05% strong scaling performance for a 1 Trillion parameter model on 3072 GPUs (Figure 13b(b)).

The 100% weak scaling performance tells us that if we can keep increasing the global batch-size proportionally with the increasing number of GPUs, we will be able to maintain the training efficiency of the smallest training workload. For example, for training a model with 1 Trillion parameter on 1024 GPUs (with two replicas or data parallels size = 2), we achieve 61.2 TFLOPS GPU throughput. We can expect to achieve roughly the same throughput even when we train the model on tens of thousands of GPUs by increasing the data parallelism proportionally. This is due to infrequent and efficient allreduce operations at the end of the forward pass. However, there is a major caveat regarding the large batch size training. With increasing global batch size, the training accuracy decreases.

The strong scaling efficiency tells us the most significant challenge with training an LLM across tens of thousands of GPUs. Here, we are keeping the global batch size fixed, that means with increasing number of GPUs, one replica of the model needs to span more and more GPUs. Consequently, there are increasing number of frequent and inefficient collective communications between various components of the same model replica.

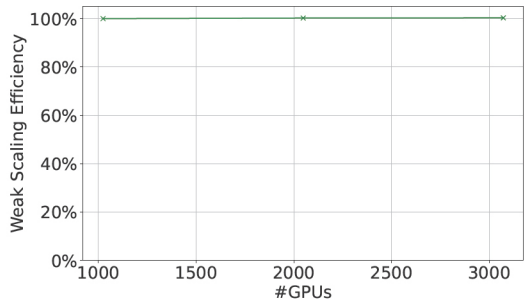
D. Power and Energy Analysis

We used rocm-smi to measure GPU level energy consumption. Figure 13 shows GPU power during training of 22B and 175B model parameters. Idle GPU power is approximately 90 Watts.

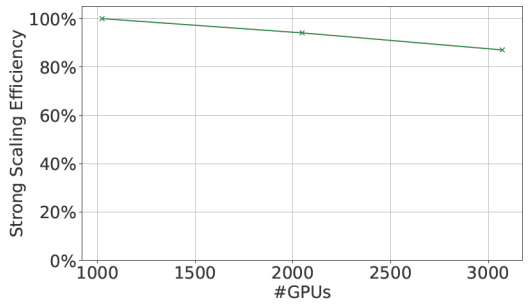
Taking iteration time, consumed tokens per iterations, average active power, and total number of MI250X GPU cards, we can estimate how much energy each model training will require to train the model on 20X tokens (compared to model parameters). For 22B, 175B, and 1T parameter model, the estimated energy consumption is 284 Gigajoules, 17.65 Terajoules, and 662 Terajoules.

VI. Conclusion and Discussion

Training an LLM with Billions to Trillions of parameters is a challenging task since we have to orchestrate overcoming the GPU memory wall, minimizing communication latency, and building a software stack with state-of-the-art distribution algorithms. A Trillion parameter model requires a minimum of 14 Terabytes of memory, while an MI250X GPU has only 64 Gigabytes. So, to overcome the memory wall problem, we have explored a combination of model parallelism strategies. However, model parallel training requires communication within groups of GPUs either sharing the parts of the same tensor (Tensor parallelism) or the groups of GPUs hosting neighboring components (pipeline parallelism). We also needed to utilize data parallelism to consume a large

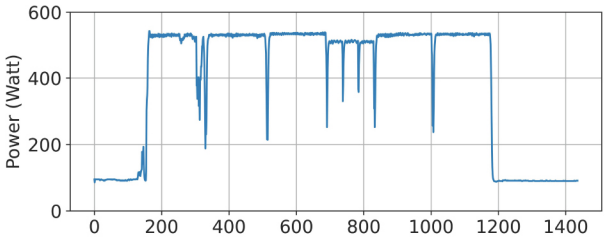


(a) Weak scaling of 1T model training by keeping per replica batch-size fixed at 1600.

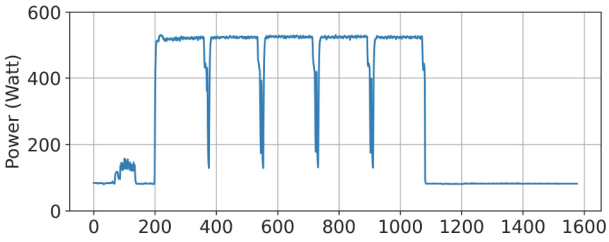


(b) Strong scaling of 1T model training by keeping a total batch· fixed at 8016. The strong scaling efficiency at 3072 GPUs is 87.(

Fig. 12: Weak and strong scaling performance of 1T model training on 3072 GPUs.



(a) GPU power for training a 22B parameter model.



(b) GPU power of an a 175B parameter model training.

Fig. 13: Strong scaling performance of 175b model and 1T model training.

number of tokens simultaneously and use a larger number of computing resources to achieve faster time to solution. All of these model parallelism and data parallelism incur communication at various frequencies and of various volumes. The communication latency can significantly increase the training time and reduce the training performance regarding GPU throughput.

We must select the right combinations of these parallelization and distributed techniques to overlap computation and communication to hide or minimize the latency. We prepared a software stack on Frontier for training LLM models by porting state-of-the-art distributed training frameworks such as Megatron-DeepSpeed and FSDP. We then used this framework to experiment with various distribution strategies and their impact on training performance. Starting with the results of these experiments, we also performed hyperparameter tuning to understand how these distribution strategies can work together to get high GPU throughput on the Frontier system with AMD hardware and the ROCM software platform. With the lessons learned from these experiments, we further tuned the distribution strategies to develop recipes for training large models such as 175 billion and 1 trillion parameters. We achieved high GPU throughput and 100% weak scaling efficiency for both models on thousands of GPUs. We also achieved 89% and 87% strong scaling efficiencies for these two models on thousands of GPUs.

One major challenge in reducing time-to-solution using a large part of Frontier will be loss divergence due to large batch size. To the best of our knowledge, the largest global batch size used in training an open-source LLM is 16 Million tokens, and most large LLMs have used much fewer tokens than the global batch size. With a sequence length of 2048, this translates to 8000 samples. We observed that at least one sample per GPU significantly boosts GPU throughput. To this extent, we must improve large-batch training and model parallel training performance with smaller per-replica batch sizes.

Most state-of-the-art distributed training frameworks target NVIDIA GPUs, and large-scale model training is done on CUDA-supported platforms. There needs to be more work exploring efficient training performance on AMD GPUs, and the ROCM platform is sparse. In this work, we have developed a training system of large LLMs of 175B and 1T on AMD hardware and the ROCM platform. This work can serve as the blueprint for efficient training of LLMs on non-NVIDIA and non-CUDA platforms such as AMD-powered Frontier supercomputer and Intel-powered Aurora supercomputer.

ACKNOWLEDGEMENTS

This research was sponsored by and used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility at the Oak Ridge National Laboratory supported by the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

CODE

The code is available at https://github.com/sajal-vt/Megatron-DeepSpeed-ORNL/tree/FA2.

REFERENCES

[1] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[4] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[5] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[6] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[7] J. Yin, S. Dash, F. Wang, and M. Shankar, "Forge: Pre-training open foundation models for science," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[8] T. Computer, "Redpajama: an open dataset for training large language models," 2023. [Online]. Available: https://github.com/togethercomputer/RedPajama-Data

[9] L. Soldaini, R. Kinney, A. Bhagia, D. Schwenk, D. Atkinson, R. Authur, K. Chandu, J. Dumas, L. Lucy, X. Lyu, I. Magnusson, A. Naik, C. Nam, M. E. Peters, A. Ravichander, Z. Shen, E. Strubell, N. Subramani, O. Tafjord, E. P. Walsh, H. Hajishirzi, N. A. Smith, L. Zettlemoyer, I. Beltagy, D. Groeneveld, J. Dodge, and K. Lo, "Dolma: An Open Corpus of 3 Trillion Tokens for Language Model Pretraining Research," Allen Institute for AI, Tech. Rep., 2023, released under ImpACT License as Medium Risk artifact, https://github.com/allenai/dolma.

[10] Soldaini, Luca and Lo, Kyle and Kinney, Rodney and Naik, Aakanksha and Ravichander, Abhilasha and Bhagia, Akshita and Groeneveld, Dirk and Schwenk, Dustin and Magnusson, Ian and Chandu, Khyathi, "The Dolma Toolkit," 2023, Apache 2.0 License, Version `0.9.0`, https://github.com/allenai/dolma.

[11] J. Yin, S. Dash, J. Gounley, F. Wang, and G. Tourassi, "Evaluation of pre-training large language models on leadership-class supercomputers," *The Journal of Supercomputing*, pp. 1–22, 2023.

[12] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[13] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[14] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[15] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[16] https://github.com/microsoft/Megatron-DeepSpeed.

[17] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[18] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer *et al.*, "Pytorch fsdp: experiences on scaling fully sharded data parallel," *arXiv preprint arXiv:2304.11277*, 2023.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[21] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[22] https://github.com/NVIDIA/Megatron-LM.

[23] https://github.com/NVIDIA/apex.

[24] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[25] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[26] https://github.com/ROCm/composable_kernel.

[27] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, "Deephyper: Asynchronous hyperparameter search for deep neural networks," in *2018 IEEE 25th international conference on high performance computing (HiPC)*. IEEE, 2018, pp. 42–51.

[28] R. Maulik, R. Egele, B. Lusch, and P. Balaprakash, "Recurrent neural network architecture search for geophysical emulation," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[29] R. Egele, R. Maulik, K. Raghavan, B. Lusch, I. Guyon, and P. Balaprakash, "Autodeuq: Automated deep ensemble with uncertainty quantification," in *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE, 2022, pp. 1908–1914.

[30] R. Egelé, I. Guyon, V. Vishwanath, and P. Balaprakash, "Asynchronous decentralized bayesian optimization for large scale hyperparameter optimization," in *2023 IEEE 19th International Conference on e-Science (e-Science)*. IEEE, 2023, pp. 1–10.

[31] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf