Distributed Training of Large Language Models on AWS Trainium

Xinwei Fu
Amazon Web Services
fuxinwe@amazon.com

Zhen Zhang
Amazon Web Services
zhzhn@amazon.com

Haozheng Fan
Amazon Web Services
fanhaozh@amazon.com

Guangtai Huang
Amazon Web Services
guangtai@amazon.com

Mohammad El-Shabani
Amazon Web Services
elshaban@amazon.com

Randy Huang
Amazon Web Services
renfu@amazon.com

Rahul Solanki
Amazon Web Services
rhsoln@amazon.com

Fei Wu
Amazon Web Services
fewu@amazon.com

Ron Diamant
Amazon Web Services
diamant@amazon.com

Yida Wang
Amazon Web Services
wangyida@amazon.com

ABSTRACT

Large language models (LLMs) are ubiquitously powerful but prohibitively expensive to train, often requiring thousands of compute devices, typically GPUs. To reduce the cost of training LLMs for customers, Amazon Web Services (AWS) launched the Amazon EC2 *trn1* instances, powered by AWS TRAINIUM, an Amazon's homegrown deep learning accelerator, as an alternative to distributed LLM training. The *trn1* instances provide a high-performance LLM training solution at a lower cost compared to their GPU-based counterpart, the *p4d* instances, which are powered by Nvidia A100 GPUs. This paper describes the design and development of the Neuron Distributed Training Library, a component of the AWS Neuron SDK, which enables distributed training of large language models on AWS TRAINIUM. Neuron Distributed Training Library supports a variety of existing distributed training techniques with unified interfaces, and provides features to address *trn1*-specific challenges as well. Our evaluation shows that *trn1* instances, specifically the *trn1.32xlarge*, achieve better or comparable performance (up to 24.6% improvement) while offering significant lower costs (up to 46.3% cost saving) in selected workloads when compared to *p4d.24xlarge* instances. As a result, AWS TRAINIUM has been adopted for training numerous external and internal models, showcasing its high-performance and cost-effectiveness. Several supported open-source LLMs are accessible via HuggingFace Optimum Neuron.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Large Language Model, Distributed Training, AWS Trainium, Neuron SDK

ACM Reference Format:
Xinwei Fu, Zhen Zhang, Haozheng Fan, Guangtai Huang, Mohammad El-Shabani, Randy Huang, Rahul Solanki, Fei Wu, Ron Diamant, and Yida Wang. 2024. Distributed Training of Large Language Models on AWS Trainium. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3698038.3698535

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated their effectiveness in a variety of challenging problems in natural language processing and related fields, including text/code generation, business consultation, and medical applications. As performance continues to rise, the model sizes of LLMs are also on the ascent. PaLM [13] has 540 billion parameters, which is a 360× increase over GPT2 [40] that was released three years earlier. As model sizes increase, training LLMs

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SoCC '24, November 20–22, 2024, Redmond, WA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698535

demands significant hardware resources and time costs. A typical way to train an LLM is to rely on the cloud providers. For example, Falcon 180B foundation model was trained on Amazon Web Services (AWS) using a cluster of up to 4096 Nvidia A100 GPUs for about 7 million GPU hours [21]. While technically feasible, training LLMs on GPUs becomes prohibitively expensive and may encounter capacity issues due to insufficient GPU supplies. Specialized deep learning accelerators [11, 12, 24] are the alternative way to train models, with hopefully better performance and less cost compared to the general-purpose GPUs. However, very few of them become popular and helpful in the LLM regime.

To better help customers process deep learning workloads, AWS started to build its own deep learning accelerators back in 2016. The homegrown accelerators are equipped to Amazon EC2 instances to accelerate deep learning workloads. Most recently, AWS launched *trn1* instances, powered by AWS Trainium, to tackle LLM training. This paper shares the experience of developing and optimizing distributed LLM training on Amazon EC2 *trn1.32xlarge* (*trn1*) instances, to fully utilize the available compute power for high-performance LLM training with lower cost. The *trn1* was designed to be the counterpart of *p4d.24xlarge* (*p4d*) with NVIDIA 40GB A100 GPUs. Both *trn1* and *p4d* offer comparable per-instance computational power and onboard memory capacity, as detailed in §3.2. We compare *trn1* against *p4d* throughout the paper.

Each *trn1* instance consists of 16 AWS Trainium accelerators, the AWS-homegrown deep learning accelerator optimized for high-performance training. AWS Trainium adopts the design of leveraging systolic arrays [28], which are well known to provide great compute throughput for matrix operations, the dominating operation in deep learning models. AWS Trainium also pairs systolic arrays with local scratchpad memory for their high performance, simple regular design and good data reuse. Similar designs have also been used by some existing popular deep learning accelerators [12, 25]. The details of the Trainium architecture and the *trn1* specification will be discussed in §3.

Along with AWS Trainium, AWS Neuron comes as a Software Development Kit (Neuron SDK) to facilitate the development on the accelerators. It supports customers throughout their end-to-end deep learning development cycle, enabling them to build, train, optimize, and deploy models for production. In the context of distributed LLM training, Neuron SDK takes in a user-provided model, makes distributed decisions based on the available model and cluster specifications, and compiles the model for execution on *trn1*. The current Neuron SDK consists of four components: a Distributed Training Library, a Framework Extension, a Compiler, and a Runtime. Neuron Distributed Training Library (NxDT) provides the distributed training interfaces to the users, while Neuron Framework Extension (NFE), Compiler, and Runtime manage model code conversion, compilation, and execution seamlessly. The primary emphasis of this paper, distributed LLM training, centralizes the design and implementation of NxDT, with additional attention given to introducing essentials from NFE, Compiler, and Runtime.

We make the following design goals of NxDT by working backwards from our customers' needs and the *trn1*'s hardware specifications:

Support existing distributed training techniques. Most of our distributed training customers target transformer-based [52] LLMs, such as LLAMA2 [51] and GPT [7, 40]. Existing distributed training techniques, such as 3D parallelism [34, 60], are proved to be effective to distributed LLM training on GPUs and should be supported by NxDT.

Require minimum porting efforts from GPU. A significant number of our customers require the porting of their custom models and training scripts, initially designed for GPU instances, to run on *trn1*. The customers prefer minimizing the efforts involved in porting the models and training scripts from GPU to AWS Trainium, ideally reusing their own models and training scripts used on GPUs.

Address hardware-specific challenges. While *trn1* is comparable to *p4d* in terms of per-instance computation power and memory capacity (details in §3), it's worth noting that *trn1* is powered by a greater number of accelerators, each with less individual computation power and memory capacity. The hardware difference results in distributed training features that are tailored specifically to *trn1*.

We build the NxDT based on the above design goals. NxDT supports a variety of existing distributed training techniques, *e.g.*, 3D parallelism, selective activation checkpoint and sequence parallelism [27]. To minimize the porting efforts from GPU, NxDT provides unified interfaces to port users' models and training scripts on *trn1*. Customers can access the various supported distributed training techniques by wrapping the model and optimizer with corresponding customized configurations in a unified manner. Furthermore, NxDT provides distributed training features to address hardware-specific challenges. For example, NxDT supports Tensor Parallelism (TP) with mixed degrees to shard models with certain parts that are not compatible with large TP degree.

Our design and implementation of the NxDT enable distributed LLM training via AWS Trainium. We have utilized *trn1* to train a number of external and internal models, showcasing its high-performance and cost-effectiveness compared to *p4d*. In addition, we offer support for a variety of open-source models through HuggingFace Optimum Neuron[1]. In this paper, we report the performance results on five distributed training workloads of LLAMA2 and GPT models

[1] https://huggingface.co/docs/optimum-neuron/index

with the number of model parameters up to 175 billion. We compare the performance of running those workloads on *trn1* instances against that on *p4d* instances. In the 32-instance distributed training experiments, the Model Floating Point Operations per Second per Instance of *trn1* outperforms that of *p4d* by up to 24.6%, and furthermore *trn1* achieves 29.1% – 46.3% cost savings across the five workloads. In terms of scaling efficiency, *trn1* achieves comparably or better strong scaling efficiencies compared to *p4d* when scaling up to 64 instances, i.e., 1024 AWS Trainium accelerators.

In summary, this paper shares the experience of developing and optimizing distributed LLM training on Amazon EC2 *trn1* instances. In the rest of the paper, we first introduce the background of existing distributed training techniques used on GPUs in §2. The same techniques are applied to AWS Trainium. We then introduce the architecture of Trainium and the specifications of Amazon EC2 *trn1* instance in §3. We then provide an overview of Neuron SDK, along with the introduction of its essential components in §4. We discuss the design and implementation of NxDT in §5. After that, we present the evaluation result in §6. We discuss the lessons learned, limitations, future works, customers and other workloads in §7. In the end, we discuss the related works in §8 and conclude in §9.

2 BACKGROUND

In this section, we discuss the existing distributed training techniques that facilitate the efficient training of LLMs on GPUs, which are also applicable to AWS Trainium.

3D Parallelism. Megatron-LM's 3D parallelism [34] combines data, pipeline, and tensor parallelism to efficiently train LLMs on distributed systems. With *Data Parallelism* (DP), each accelerator has a replicated full model weight and processes a different subset of the input dataset concurrently. The gradients are aggregated among accelerators periodically to ensure the consistency of the model weights. With *Pipeline Parallelism* (PP), the layers of a model are sharded sequentially into distinct stages, each of which is executed on a different accelerator simultaneously. With *Tensor Parallelism* (TP), individual layers of a model are sharded over multiple accelerators, thus certain tensor operations are distributed and computed across multiple accelerators in parallel. When implementing TP in attention layers, attention heads are distributed across different TP ranks [47], thus necessitating that the number of heads be divisible by the TP degree.

Reducing Activation Memory Footprint. Selective Activation Checkpoint and Sequence Parallelism [27] are proposed to reduce the activation memory footprint during LLM distributed training. Instead of storing activations for all layers during forward pass or recomputing activations for all layers during backward pass, *selective activation checkpoint* selectively stores and recomputes activations for certain layers. Selective activation checkpoint reduces training memory footprint while preserving training performance simultaneously. *Sequence parallelism* parallelizes certain computation in the sequence dimension to reduce the activation memory footprint. Specifically, sequence parallelism targets the computation in the non-tensor parallel regions of transformer layers, such as `LayerNorm` and `Dropout`.

ZeRO. Zero Redundancy Optimizer (ZeRO) [41] extends Data Parallelism by sharding training states, such as model weights, gradients and optimizer states, onto multiple accelerators. ZeRO supports different levels of sharding and memory reduction. When ZeRO-1 is applied, the optimizer states are partitioned across the processes, and each process updates only its own optimizer partition. Since 3D Parallelism shards the model and ZeRO-1 shards the optimizer states, users can apply both 3D Parallelism and ZeRO-1 simultaneously as needed.

3 NEURON HARDWARE ARCHITECTURE

This paper focuses on the distributed LLM training on Amazon EC2 *trn1* instances, which are powered by AWS Trainium accelerators. In this section, we first introduce the architecture of a Trainium accelerator (§3.1), and then present the specification of an EC2 *trn1* instance with the comparison with that of its GPU Counterpart, *i.e.*, EC2 *p4d* instance (§3.2).

3.1 Trainium Architecture

Figure 1 shows the hardware architecture of a Trainium accelerator. Each Trainium accelerator consists of two *NeuronCores*. Each *NeuronCore* is a fully-independent heterogeneous compute unit, which consists of four compute engines (Tensor/Vector/Scalar/GPSIMD Engines), and one on-chip software-managed scratchpad memory.

- *Tensor Engine* is based on a power-optimized systolic-array, which is highly optimized for tensor computations, *e.g.*, general matrix multiply, convolution, reshape, and transpose. The Tensor Engine also supports mixed-precision computations. Each Tensor Engine delivers over 90 TFLOPS of FP16/BF16 tensor computations.
- *Vector Engine* is optimized for vector computations, in which every element of the output is dependent on multiple input elements, *e.g.*, layer normalization and pooling operations. Each Vector Engine delivers 2.3 TFLOPS of FP32 computations.
- *Scalar Engine* is optimized for scalar computations, in which every element of the output is dependent on one element of the input, *e.g.*, activation functions. Each Scalar Engine delivers 2.9 TFLOPS of FP32 computations.
- *General-Purpose-Single-Instruction-Multiple-Data (GPSIMD) Engine* consists of eight fully programmable 512-bit wide

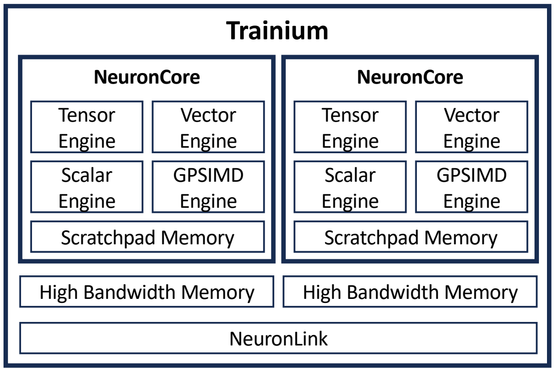


Figure 1: Trainium accelerator hardware architecture. Each Trainium accelerator has two NeuronCores. Each NeuronCore consists of four compute engines and one scratchpad memory. The scratchpad memory of each NeuronCore connects with a high bandwidth memory. The NeuronLink is responsible for device-to-device interconnection.

general-purpose processors, which can execute straight-line C-code, and have direct access to the other *NeuronCore* engines, as well as the on-chip scratchpad memory. With the GPSIMD engine, developers can implement custom-operators and execute them on the *NeuronCore* engines.

- *Scratchpad Memory* is a 24MB software-managed on-chip Static Random Access Memory (SRAM) in each *NeuronCore*. Maintaining data locality in the scratchpad memory is the key to maximize the computation performance.

Inside each Trainium accelerator, two *High Bandwidth Memory* with 16 GB capacity, are connected to *NeuronCores'* scratchpad memory through direct memory access with 820 GB/s bandwidth. The NeuronLink, responsible for accelerator-to-accelerator interconnect, enables scale-out training via communication between different Trainium accelerators.

3.2 *trn1* Specification

trn1 and *p4d* are two popular Amazon EC2 instances used for LLM training in AWS. Table 1 shows the instance specification comparison between *trn1* and *p4d*. The accelerators in *trn1* and *p4d* are Trainium (with two NeuronCores each) and NVIDIA A100 GPU, respectively. While each Trainium has less computation performance and memory capacity than A100 GPU (190 vs 312 BF16/FP16 TFLOPS), the total per-instance accelerator computation performance and memory capacity of *trn1* is larger than that of *p4d*: each *trn1* consists of 16 Trainium accelerators, *i.e.*, 3040 BF16/FP16 TFLOPS and 512 GB accelerator memory per instance, while each *p4d* consists of 8 A100 GPUs, *i.e.*, 2496 BF16/FP16 TFLOPS and 320 GB accelerator memory per instance. The accelerator-to-accelerator bandwidths of Trainium and the NVIDIA A100 GPU are 384 and 600 GB per second per accelerator, respectively. It is noteworthy that the ratio of BF/FP16 TFLOPS to accelerator-to-accelerator bandwidth for Trainium is similar to that of the NVIDIA A100 GPU. The network bandwidth of *trn1* (800 Gbps) also doubles that of *p4d* (400 Gbps). Furthermore, the on-demand price per hour of *trn1* ($21.5) is around two thirds of that of *p4d* ($32.77), making *trn1* more cost effective.

Table 1: EC2 instance specification comparison between *trn1* and *p4d*.

Accelerator Type	**Trainium Device**	**A100 GPU**
BF16/FP16 TFLOPS	190	312
Onboard Memory (GB)	32	40
Accelerator-to-accelerator (GB/sec/accelerator)	384	600
Instance Type	***trn1***	***p4d***
# Accelerator	16	8
BF16/FP16 TFLOPS	3040	2496
Onboard Memory (GB)	512	320
Network Bandwidth (Gbps)	800	400
Price ($/hour)	21.5	32.77

4 AWS NEURON SDK

In this section, we first provide an overview of Neuron SDK (§4.1), then introduce the following three components: Neuron Framework Extension (§4.2), Compiler (§4.3), and Runtime (§4.4). NxDT will be discussed in the next section (§5).

4.1 Overview

AWS Neuron is a software development kit (SDK)[2] enabling high-performance deep learning acceleration using AWS Trainium. With Neuron SDK, one can develop, profile, and deploy high-performance deep learning workloads on top of EC2 instances with AWS Trainium such as *trn1*. Distributed LLM training workload is a major focus in Neuron SDK.

Neuron SDK consists of four major components in a top-down order: *Neuron Distributed Training Library (NxDT)*, *Neuron Framework Extension (NFE)*, *Neuron Compiler* and *Neuron Runtime*. Figure 2 illustrates the overview of distributed LLM training workflow on Trainium using Neuron SDK, showcasing the close collaboration of all four components. The user's training script first invokes NxDT to apply distributed training techniques (step ❶); NFE traces the processed script (step ❷), generates computation graphs, and sends them to Neuron Compiler (step ❸); Neuron Compiler compiles the computation graphs into Neuron Executable File Format files (NEFFs) and sends NEFFs back to NFE (step

[2] Neuron SDK have been publicly available at https://github.com/aws-neuron, including the libraries for distributed training and inference.

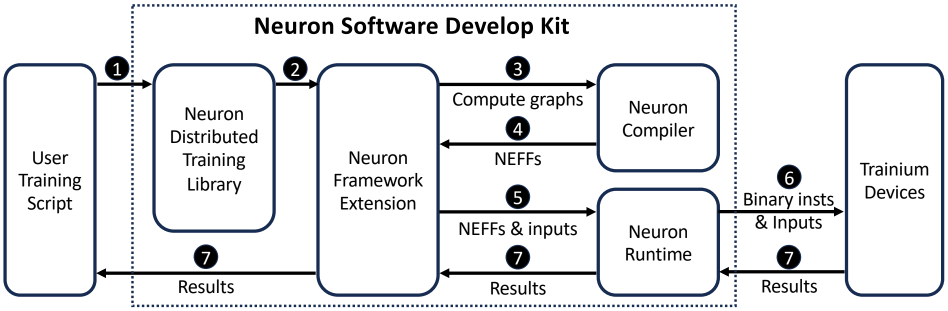


Figure 2: Neuron distributed training overview.

❹); NFE then sends the compiled NEFFs along with the inputs to Neuron Runtime (step ❺); after that, Neuron Runtime loads the NEFFs, generates binary instructions, and executes those binary instructions with the inputs on Trainium (step ❻); at last the results will be propagated to the user (step ❼). In this section, we briefly discuss NFE, Compiler, and Runtime, while leaving NxDT, the one that is directly related to distributed LLM training, to be elaborated in details in §5.

4.2 Neuron Framework Extension

NFE allows different existing deep learning frameworks, *e.g.*, PyTorch [39], Tensorflow [1], and JAX [6], to interact with Neuron software components and to operate AWS Trainium. To support different frameworks, models implemented in different frameworks are first lowered to unified computation graph representations, which then are sent to Neuron Compiler. In our current implementation, NFE leverages the computation graph representations in XLA [38]. One of the main advantages of XLA is that it has been already natively integrated into the leading deep learning frameworks, *e.g.*, PyTorch, TensorFlow, and JAX, and thus allows Neuron SDK to enable multiple entry points into Trainium.

Some frameworks, such as PyTorch, execute their computations in "eager" mode [50], *i.e.*, computations are executed operation-by-operation sequentially in a define-by-run manner, which makes compiler-based solutions less efficient. More specifically, the eager execution model does not allow for significant compiler-based optimization opportunities across operations. To address the issue incurred by eager mode execution, NFE inherits the LazyTensor [49] technique from PyTorch. With LazyTensor, the whole computation is split into multiple computation graphs by inserting barriers automatically or manually. Operations are traced into a computation graph first, and the compilation and execution of the traced computation graph are triggered when encountering a barrier. The LazyTensor allows Neuron Compiler to get triggered when hitting a barrier, and then optimize the entire traced computation graph across multiple operators. In addition, NFE employs a NEFF cache of compiled computation graphs to avoid redundant compilations. The NEFF cache stores compiled NEFFs on disk during the just-in-time (JIT) compilation process. Given the repetitive nature of deep learning training workloads, the majority of JIT compilations occur during the initial training steps. The NEFF cache indexes these compiled NEFFs using the hash value of the computation graph and operates without an eviction policy. If a traced computation graph has been compiled before, *i.e.*, the compiled NEFF exists in the cache, NFE just fetches the NEFF in the cache instead of recompilation.

4.3 Neuron Compiler

In order to achieve high performance, Neuron SDK uses JIT compilation to process the model for a full stack optimization. Neuron Compiler processes computation graphs as soon as they become available from the framework, without an explicit compilation step. The first compilation phase is a graph optimizer for hardware-agnostic optimizations, like operator fusion and common sub-expression elimination. After graph compilation, each operator in the computation graph is represented in a loop format, and the compiler moves into its loop optimization phase. In this phase, the most optimal layout of each tensor is identified, followed by a sequence of loop transformations like tiling, vectorization, pipelining, and more. At the end of this phase, the inner loops are mapped into hardware intrinsics, which are executable by the accelerator. Finally, in the last step, memory allocation and instruction scheduling are performed to efficiently manage hardware resources. The memory allocator aims to minimize data movement by keeping data local in the on-chip memories, while the instruction scheduling reorders instructions to achieve a high level of instruction level parallelism. Ultimately, the Neuron compiler compiles the computation graphs generated by NFE into NEFFs.

4.4 Neuron Runtime

Neuron Runtime loads compiled NEFFs for execution on our accelerators. Running on instances, Neuron Runtime consists of kernel driver and C/C++ libraries, which provide APIs to access the Trainium accelerator. In addition to NEFF execution, Neuron Runtime also provides interfaces for monitoring hardware health and performance to profile compute performance and collect execution statistics.

It is worth noting that Neuron Runtime employs asynchronous execution to reduce the framework level overhead incurred by the current LazyTensor implementation in PyTorch. In the current PyTorch LazyTensor implementation, the execution of the current computation graph can overlap with the tracing of the next computation graph to reduce the tracing overhead. However, only after the execution of the current computation graph finishes can the execution of the next computation graph start, even if the barrier of the next computation graph has been hit and the graph tracing finishes. In other words, while there is an executing graph, at most one more graph can be traced. If the current executing graph is large and the next graph is small, the LazyTensor system remains idle after tracing the next small graph and waits for current large graph execution, which wastes the opportunity of tracing more graphs further. To mitigate this limitation, Neuron Runtime returns to the LazyTenor system immediately and executes the graph in background, thus the LazyTensor system keeps tracing following graphs regardless of the completion of the current executing graph.

The asynchronous execution achieves up to 119% performance improvement in our evaluation, as shown in §6.5. The total number of computation graphs that can be traced while a graph is executing in background is configurable to users. Based on our experience, we trace a larger number of graphs in the background when pipeline parallelism is employed. However, we must ensure that this number does not lead to CPU out-of-memory errors, as tracing more graphs increases CPU memory consumption.

5 NEURON DISTRIBUTED TRAINING LIBRARY

On top of NFE, Neuron Compiler, and Neuron Runtime, we build NxDT to support distributed LLM training. This section introduces the design and implementation of NxDT. We first present the supported features (§5.1) of NxDT. Then we especially discuss two unique features in NxDT: Automatic Pipeline Stage Partition (§5.2) and Tensor Parallelism with Mixed Degrees (§5.3). Lastly, we show a script example that uses NxDT for training (§5.4).

Table 2: Neuron Distributed Training Library Features

Categories	Features	Model Change?
Existing Distributed Training Techniques	Data Parallelism	**NO**
	Tensor Parallelism	**YES**
	Pipeline Parallelism	**NO**
	ZeRO-1	**NO**
	Selective Activation Checkpoint	**YES**
	Sequence Parallelism	**YES**
Unified Interfaces	Model and Optimizer Wrappers	**NO**
	Automatic Pipeline Stage Partition	**NO**
Trn1-specific Features	TP with Mixed Degrees	**YES**
	Model Initialization on Trainium	**NO**
Other	Attention Head Padding	**NO**

5.1 Features

NxDT supports a variety of existing distributed training techniques mainly targeted transformer-based models [52]. Our library supports 3D parallelism [34], *i.e.*, Tensor Parallelism (TP), Pipeline Parallelism (PP), and Data Parallelism (DP), which is used by the most of existing LLM training [7, 51, 54]. To reduce the activation memory footprint during training, selective activation checkpoint and sequence parallelism [27] are naturally supported with the 3D parallelism. TP and PP shard the model parameters and gradients. We also support Zero Redundancy Optimizer Stage 1 (ZeRO-1) [41] to shard optimizer states. 3D parallelism and ZeRO-1 can be applied simultaneously during training.

NxDT provides unified interfaces to port users' models and training scripts on *trn1*. To facilitate this, NxDT offers model and optimizer wrappers. Users can apply the required distributed training techniques by wrapping their customized models and optimizers with the corresponding configurations. Different parallelism strategies may necessitate distinct optimization boundary operations and checkpoint saving/loading schemes. For instance, when ZeRO-1 is applied, optimizer state synchronization and sharding are required at the optimization boundary and during checkpoint operations. Leveraging the model and optimizer wrappers, NxDT automatically handles these optimization boundary operations and checkpoint procedures based on the applied distributed training techniques. Furthermore, NxDT features automatic pipeline stage partitioning, which will be discussed in §5.2, to minimize the effort required when implementing pipeline parallelism.

NxDT also provides distributed training features that are tailored specifically to *trn1*. For example, NxDT supports TP with mixed degrees, which will be discussed in §5.3, *i.e.*, users can employ multiple TP degrees to shard different model parameters. As a *trn1* is equipped with 32 NeuronCores, the distributed training on *trn1* may set TP degree to 32 for large models, *e.g.*, LLAMA2-70B and GPT-175B in our practice, to maximize the intra-instance bandwidth for TP. TP with mixed degrees is used to handle the case when the LLM has some model parts are not compatible with a large unified TP degree, *e.g.*, when the Grouped Query Attention (GQA) [2] is applied, a unified TP degree of 32 to the whole model may not be applicable.

Additionally, *trn1* is significantly more prone to encountering host Out-of-Memory (OOM) issues, as a single *trn1* instance has 32 NeuronCores, meaning 32 models may be initialized on the host simultaneously. To address this, NxDT supports model initialization on Trainium, which materializes the model weights after the empty model in the host has been moved to Trainium, thus avoiding host OOM.

NxDT also supports attention head padding to make the number of attention heads divisible by the TP degree. This is important for achieve the best TP efficiency but the model architecture may not design with it in mind. For example, Falcon-7B [3] has 71 attention heads in a transformer layer, which prevents applying TP with degree 8. In our practice of training Falcon-7B, the number of attention heads are padded to 72.

We implemented NxDT by extending PyTorch due to its widely usage in existing libraries, *e.g.*, Megatron-LM/NeMo [27, 34, 47], DeepSpeed [41, 42], and FSDP [57]. Table 2 lists the features supported by NxDT as discussed above. The first column in Table 2 shows the four categories of the features, the second column shows each feature, and the last column indicates whether the feature requires manual model changes by users. Among all the supported features, only TP (with Mixed Degrees), Selective Activation Checkpoint, and Sequence Parallelism require code changes of models, while all the other features are invoked in training scripts directly. To use TP (with Mixed Degrees), users are required to replace certain non-parallel layers with parallel layers [47], such as the query, key, value, and dense layers in an attention module [52]. To use Selective Activation Checkpoint, users need to wrap the attention core computation as a module and pass the module class to the model wrapper. To use Sequence Parallelism, users need to mark the corresponding `LayerNorm` layers, and implement the required dimension transposition and collective communication of intermediate states based on the proposed algorithm [27].

5.2 Automatic Pipeline Stage Partition

The PP designs of most existing deep learning frameworks require users to implement the models or training scripts in specific formats, which harms user experience and limits the flexibility. For examples, Megatron-LM/NeMo only supports their own model implementation; PyTorch and DeepSpeed require models to be implemented as a list of layers by using `torch.nn.Sequential` to apply pipeline parallelism.To minimize the changes of models and training scripts when applying pipeline parallelism, NxDT uses a trace-based solution to automatically partition the model into pipeline stages similar to Slapo [9].

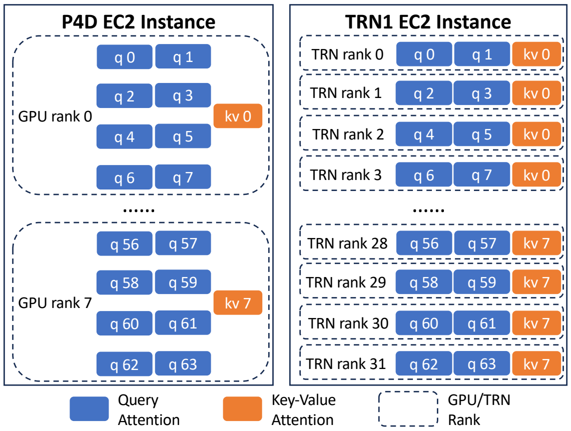


Figure 3: An example of using TP with mixed degrees for group query attention. The P4D instance uses TP degree 8 for query, key and value attention heads, while TRN1 instance uses TP degree 32 for query attention heads and TP degree 8 for key and value attention heads.

We first trace the entire model source code into a static graph representation, then partition the graph into a list of sub-graphs based on user annotations, and lastly convert each sub-graph to the source code of each pipeline stage. We leverage `torch.fx` [43] to perform the transformation between model source code and graph representation. However, `torch.fx` has its own limitations. First, `torch.fx` traces models by using symbolic execution, which cannot trace dynamic control flow [43]. Fortunately, most of existing transformer-based language models, such as LLAMA2 and GPT, do not contain dynamic control flow and can be traced symbolically. Second, `torch.fx` only traces `torch` functions, *i.e.* built-in Python functions can not be traced by `torch.fx`. `torch.fx` provides an interface to wrap a module as a symbol to avoid tracing certain blocks of codes. We by default mark all transformer layers as symbols during the tracing, so that the complicated computation inside transformer layers is not traced. Uses can also provide a list of modules to wrap as symbols to avoid tracing `non-torch` functions. Furthermore, HuggingFace has native `torch.fx` tracing support for most existing transformer-based language models, which then can be naively applied with our pipeline parallelism.

5.3 Tensor Parallelism with Mixed Degrees

As each *trn1* instance has 32 NeuronCores, when using *trn1* for training extremely large models, *e.g.*, a model with tens or hundreds billions parameters, it is ideal to set TP degree to 32 for maximizing the performance. This follows the same takeaway from training experiences using GPUs [47]. However, in many cases, the model architecture prevents us from using TP degree 32 for certain parts of the model. For example, LLAMA2-70B with GQA enabled has eight attention groups (most likely as the model was designed for the GPU instance topology with 8 GPUs per node), which is not divisible with 32. To handle such cases, NxDT supports TP with mixed degrees.

We use a real case of LLAMA2-70B with GQA in our production to illustrate the feature of TP with mixed degrees, as shown in Figure 3. In each attention layer of a LLAMA2-70B model, there are 64 query attention heads and 8 key-value attention heads; and every 8 query attention heads share one key-value attention head. The left side of Figure 3 shows how one attention layer is sharded using TP degree 8 in a *p4d* instance with 8 GPUs, each of which perfectly holds 8 query attention heads and one key-value attention heads. The right side of Figure 3 illustrates how mixed TP degrees are utilized within a *trn1* instance, which contains 32 NeuronCores. A TP degree of 32 is applied for sharding query attention heads, meaning each NeuronCore holds two query attention heads. For key-value attention heads, a TP degree of 8 is used, with each NeuronCore storing one key-value attention head, while every four NeuronCores replicate the same key-value attention head. For instance, ranks 0–3 replicate the same key-value attention head, and ranks `0,4,8,...,28` form a TP group to shard the 8 key-value attention heads. Note that using a unified TP degree 8 for LLAMA2-70B on *trn1* leads to larger numbers of pipeline stages, which hurts the performance, as will be shown in our evaluation (§6.5).

5.4 Running Example

Figure 4 shows a script example that uses NxDT for training. This example assumes that the model changes, which are required by the features as shown in Table 2, have already been applied. This example also assumes that PP has been enabled. This example first creates a configuration (line 4-13) that indicates what features listed in Table 2 will be applied to the distributed training. The configuration is self-explanatory, and its specification is publicly available[3]. After creating the configuration of the distributed training, this example shards and initializes the model and the optimizer based on the configuration by wrapping the model (line 15-17) and the optimizer (line 19-22) separately. By wrapping the model, inside a training loop, model forward and backward passes and gradient accumulation are all wrapped together in one function (line 32) with applying the features defined in the configuration. The optimization boundary operations, which includes but are not limited to the gradient synchronization and clipping and optimization step, are also wrapped in one function (line 33). Training checkpoints, including model and optimizer states and user-defined information, are loaded and saved at line 24-28 and line 36-42 based on the configuration, respectively.

```
 1  import neuronx_distributed as nxd
 2
 3  # create config
 4  nxd_config = nxd.neuronx_distributed_config(
 5      tensor_parallel_size,
 6      pipeline_parallel_size,
 7      pipeline_config,
 8      optimizer_config,
 9      activation_checkpoint_config,
10      sequence_parallel_enabled,
11      model_init_config,
12      pad_model_enabled,
13  )
14  # wrap model
15  model = nxd.initialize_parallel_model(
16      nxd_config, model_constructor
17  )
18  # wrap optimizer
19  optimizer = nxd.initialize_parallel_optimizer(
20      nxd_config,
21      AdamW, model.parameters(), lr=1e-3
22  )
23  # loading checkpoint
24  user_content = nxd.load_checkpoint(
25      "ckpts",
26      model=model,
27      optimizer=optimizer,
28  )
29  ...
30  # training loop
31  for inputs in train_dataloader:
32     loss = model.run_train(inputs)
33     optimizer.step()
34     ...
35  # saving checkpoint
36  nxd.save_checkpoint(
37      "ckpts",
38      nxd_config=nxd_config,
39      model=model,
40      optimizer=optimizer,
41      user_content={"total_steps": total_steps},
42  )
43  ...
```

Figure 4: NxDT Interface Example

In production, some users requested to port model training with Megatron-LM/NeMo on GPUs to Trainium. Our library supports Megatron-LM/NeMo as a front-end. In such cases, users only need to provide a training configuration file of Megatron-LM/NeMo to run distributed training of the supported models on Trainium without any code changes.

[3] https://awsdocs-neuron.readthedocs-hosted.com/en/latest/libraries/neuronx-distributed/api-reference-guide-training.html#initialize-nxd-core-config

Model Name	**NL**	**HS**	**IS**	**NAH**	**NQG**	**SL**	**TFPS**	***p4d***	***trn1***
LLAMA2-7B	32	4096	11008	32	32	4096	175	TP=8, PP=1	TP=8, PP=1
LLAMA2-13B	40	5120	13824	40	40	4096	336	TP=8, PP=2	TP=8, PP=4
LLAMA2-70B	80	8192	28672	64	8	4096	1828	TP=8, PP=8	TP=(32, 8), PP=4
GPT-23B	28	8192	32768	64	64	2048	292	TP=8, PP=2	TP=8, PP=4
GPT-175B	96	12288	49152	96	96	2048	2202	TP=8, PP=16	TP=32, PP=8

NL: number of layers **HS**: hidden size **IS**: intermediate size **NAH**: number of attention heads **NQG**: number of query groups
SL: input sequence length **TFPS**: teraFLOPs per sequence **TP**: tensor parallel size **PP**: pipeline parallel size

Table 3: Evaluated models and parallelism strategies.

We provide publicly available example tutorials demonstrating the utilization of our distributed training library for training selected HuggingFace models[4] and Megatron-LM/NeMo models[5].

6 EVALUATION

Since being launched, *trn1* has been supporting internal and external customers on their distributed LLM training as an effective alternate to the GPU instance *p4d*. In addition, a variety of supported open-source LLMs are available at HuggingFace Optimum Neuron. In this section, as a generic evaluation, we first present our evaluation methodology (§6.1), then present the following evaluation results:

- **Performance and scalability**: We compare end-to-end training performance (§6.2) and scalability (§6.3) of popular LLM workloads on *trn1* and *p4d*.
- **Usability**: We demonstrate the advantage of Neuron SDK usability (§6.4).
- **Ablation study**: We provide various ablation experiments to study performance characteristics of Neuron SDK (§6.5).
- **Fidelity**: We check the fidelity of distributed training on *trn1* using Neuron SDK (§6.6).

6.1 Evaluation Methodology

Hardware and software setups. We conduct all experiments on Amazon EC2 *trn1* and *p4d* instances. Table 1 summarizes the hardware specifications of both instances. We conduct our experiments using Neuron SDK on *trn1* and using NVIDIA NeMo on *p4d*, respectively. Despite differences in data ingestion methods between Neuron SDK and NeMo, our profiling results indicate that the data ingestion cost is negligible compared to the computational expenses for both platforms. The software stacks include: Neuron SDK 2.16, NeMo 23.08 (available inside NeMo docker container with tag 23.08), EFA 2.1.1, aws-ofi-nccl 1.5.0-aws.

Workloads. We evaluate Neuron SDK with popular and representative transformer-based language models, including LLAMA2-style [51], and GPT-style models. Model sizes are varied according to public benchmark configurations [7]. Table 3 lists out the model configurations in details. We name each model variant with the convention "`<model>-<size>`". The model sizes are counted in billions. For example, the "LLAMA2-7B" denotes the LLAMA2 model architecture with 7-billion parameters. Table 3 also provides the input configurations, which follow the conventions [7, 51]. Throughout the evaluation, we use BFloat16 [26] precision. AdamW [31] optimizer is employed for updating model weights of each training iteration.

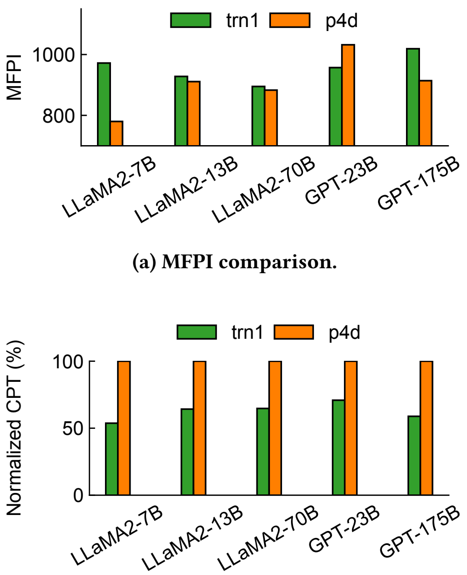
Parallelism. The last two columns of Table 3 include the parallelism configurations used for training on *trn1* and *p4d* instances. The data parallel (DP) size depends on the size of the cluster. The product of DP size, tensor parallel (TP) size, and pipeline parallel (PP) size equals to the number of accelerators in a cluster. For example, in the case of training GPT-175B on 32 *trn1* instances, the DP size is 8, given by $(32 \times 32)/(32 \times 4)$, as each *trn1* has 32 NeuronCores. The parallelism configuration of each model for either Trainium or GPU follows the takeaways from practices [34]. Specifically, we limit the TP size less or equal to the number of cores/devices on each instance. As described in §5.3, we use tensor parallelism with mixed degrees for LLAMA2-70B. We use the smallest PP size to enable training to reduce the pipeline bubble overhead [34]. The difference in parallelization schemes between *trn1* and *p4d* arises from the factors that each *trn1* has a larger number of accelerators and the onboard memory capacity of each *trn1* is also larger. Taking LLAMA2-70B and GPT-175B on *trn1* as examples, when TP degree is 32, each pipeline stage resides in one *trn1* instance; since the permitted memory consumption of one pipeline stage on *trn1* is higher than that on *p4d*, so the PP degrees of LLAMA2-70B and GPT-175B on *trn1* are smaller than that on *p4d*.

Metrics. We mainly use three metrics to evaluate the system performances: achieved Model Floating point operations

[4] https://awsdocs-neuron.readthedocs-hosted.com/en/latest/libraries/neuronx-distributed/index.html

[5] https://awsdocs-neuron.readthedocs-hosted.com/en/latest/libraries/nemo-megatron/index.html

per second (FLOPS) per Instance (MFPI), Cost per 4-million tokens (CPT), and scalability, in which the scalability is evaluated by measuring the system throughput (sequence per second, *i.e.*, seq/sec) performance on different sized clusters. The MFPI metric provides evaluation of the floating point processing throughput we can achieve in distributed setting. The formulation of MFPI is $t \times f/n$, where n is the number of instances in a cluster, t is the throughput in sequence per second, f denotes the floating point operations (FLOPs) per sequence. And the CPT metric provides the sense of financial costs when using *trn1* and *p4d* instances. The value of CPT is calculated as $\mathrm{CPT} = (p \times n/3600) \times (4194304/(s \times t))$, where p is the hourly rate in dollar, n denotes the number of instances in the cluster, s is the input sequence length, and t is the measured throughput in sequence per second. The use of 4-million tokens is chosen following a common practice. From the literature of LLM trainings [32, 51], 4-million tokens are commonly used for each iteration, i.e., updating the model parameters, to get high quality models. Other than that, we also report time cost of model compilation, and model code changes to enable distributed training with NEURON SDK to demonstrate our usability advantages.



(a) MFPI comparison.

(b) Cost of every 4-million tokens.

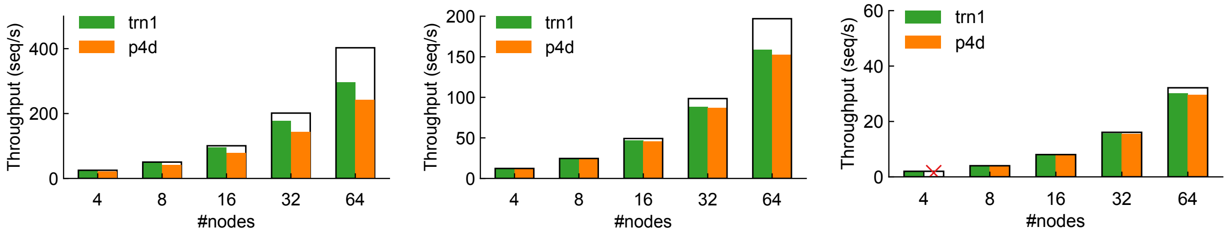
Figure 5: Training performance comparison between *trn1* and *p4d* on 32 instances.

6.2 Training Performance

In this subsection, we compare the end-to-end TFLOPS performance and corresponding financial cost of *trn1* against those of *p4d*.

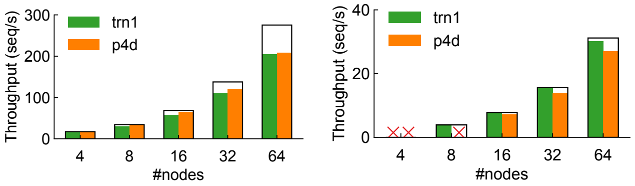
The evaluations are conducted on 32 instances (i.e., either *trn1* or *p4d* instances). We evaluated five different sized models, listed in the Table 3. Figure 5a shows the MFPI comparison of *trn1* and *p4d* instances. In most cases, the *trn1* can outperform *p4d* noticeably. The performance advantage is by up to 24.6%. For the cases which *trn1* is less performant, the gap is within 7.3%. The MFPI performance benefit is especially obvious in settings where the model sizes are either relatively small (*i.e.*, we can train models by only applying TP) or relatively large (*i.e.*, the model size requires a large PP size on GPU platform). For relatively larger models, we can apply larger TP size to reduce the PP size, because on *trn1* we have 32 NeuronCores instead of 8 devices on GPU platform. With reduced PP size, the pipeline bubble overhead is effectively reduced [34], which results in better performance when comparing to training on *p4d*. For these relatively smaller models like LLAMA2-7B, the iteration time is relatively smaller due to smaller number of float point operations. As a result, training performance of these smaller models are more sensitive to network bandwidth. At the end of each iteration, there is a collective all-reduce communication to synchronize model replicas. This all-reduce communication is bandwidth bounded operation due to large network traffic volume, *e.g.*, for LLAMA2-7B model all-reduce transits about 14GB data across computing instances when training in half-precision mode. With 800Gbps network (Table 1), the all-reduce operation overhead is smaller, and thus these relatively smaller models are favor for *trn1*. For the GPT-23B models, *trn1* did not demonstrate a performance advantage. One primary reason for this is the use of a larger PP degree on *trn1*, which incurs increased PP runtime overhead.

The *trn1* instance has a clear advantage for financial cost in training, shown in Figure 5b. The *trn1* achieves 29.1% – 46.3% cost savings across five training workloads. And the system throughput is even better or on par with *p4d*, which means no compromise of training time to get quality models. The cost advantage is more obvious for LLAMA2-7B and GPT-175B models. This observation is consistent to the performance advantages discussed above. The larger the performance gains we have on *trn1* the more savings for processing every 4-million tokens. Note that the CPT metric is a function of both performance and instance price. While instance pricing is influenced by business and market factors, *trn1* demonstrates better or comparable performance on selected workloads when compared to *p4d*. With its lower instance price and competitive performance, *trn1* provides clear cost advantages.



(a) LLAMA2-7B **(b) LLAMA2-13B** **(c) LLAMA2-70B**

Figure 6: Strong scaling of LLAMA2 models; the outer rectangular denotes the linear scaling with respect to the throughput of the smallest cluster.



(a) GPT-23B **(b) GPT-175B**

Figure 7: Strong scaling of GPT models; the outer rectangular denotes the linear scaling with respect to the throughput of the smallest cluster.

6.3 Scalability

In this subsection, we present the strong scaling results of *trn1* and *p4d*. The strong scaling measures the system efficiency with fixed amount of computations when scaling up the size of the cluster. In our experiments, we fixed the number of tokens as 4 million for each model updating iteration. As the number of tokens and the structure of the model are determined, the computation for each iteration is thus fixed. We scale up the cluster size from 4 to 64 instances. We measure the training throughput in terms of sequence per second of each setup. Ideally, the system throughput scales linearly, in which when doubling the cluster size, the throughput of the system is also doubled. However, this is particularly difficult to achieve in practice when the amount of the computation work is fixed. To achieve linear scaling, the system needs to keep the overhead roughly the same when increasing the cluster size, while the system overhead grows due to various sources as the cluster grows up [34, 56].

The *trn1* instance can scale up the training almost linearly for large models like LLAMA2-70B (Figure 6c) and GPT-175B models (Figure 7b). Outer rectangles in figures denote the linear scaling w.r.t. the maximum throughput of either GPU or Trainium that is obtained on the smallest cluster. For GPT-175B model, training on eight *p4d* runs into out-of-memory errors (noted with "×" in Figure 7b) due to insufficient memory. The *trn1* instance has more on-device memory per instance, which provides the capability of training the model on eight instances. For GPT-175B case specifically, we used pipeline parallel (PP) size 16, and micro-batch size 2 for benchmark. In comparison, on *trn1* PP size 8 is used. At the scale of the 64 instances, both platforms use 256 accumulation steps to process 4-million tokens. With the same accumulation steps, the *trn1* instance has about half of the pipeline bubble (2.7% vs 5.8%), according to formulation of $(p-1)/m$ [34] where p denotes the PP size and m denotes the accumulation steps. Other than the advantage of less pipeline bubbles *trn1* doubles the network bandwidths across instance for necessary communications of distributed training. For LLAMA2-70B model, training on *trn1* leverages tensor parallelism with mixed degrees, and it incurs extra computation overhead for key value attention. We will discuss the effect of using tensor parallelism with mixed degrees in §6.5. For the scalability of other models, Figure 6 and Figure 7 show that the *trn1* instance can achieve comparable or better strong scaling efficiencies. It is worth noting that the scalability and the FLOPS performance of the GPU platform reported here do not completely match what was reported by other vendors [27, 48, 51]. This is due to hardware differences, which mainly includes network bandwidth, on-device memory, and memory bandwidth differences [37].

6.4 Usability

Compilation overhead. JIT compilation enables high-performance executables but may also introduce compilation overhead at runtime. Across our evaluation in §6.2, the compilation overhead is under 9 minutes. Compared to the long LLM training time, *e.g.*, weeks or months, we believe our compilation overhead is negligible.

Model code changes. Users often come with models available in popular open source communities such as HuggingFace. Porting them to Trainium for distributed training

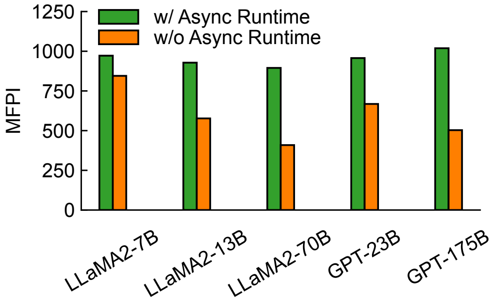


Figure 8: The comparison of MFPI with and without enabling Async Runtime on 32 instances.

requires necessary code change. We use the HuggingFace LLAMA2-70B model, which consists of 1,009 lines of codes, as a case study. The total model code changes to train the HuggingFace LLAMA2-70B model on *trn1* are 82 lines, which includes applying TP with mixed degrees (21 lines), applying sequence parallelism (27 lines), applying selective checkpoint (25 lines), and importing packages (9 lines). It is worth noting that converting a HuggingFace model to execute 3D parallelism on GPU requires a similar amount of work. On the other hand, training models written in NeMo-Megatron on *trn1* requires no model code changes.

6.5 Ablation Studies

In this subsection, we conduct ablation studies to showcase the Trainium-unique optimizations we have done.

TP with Mixed Degrees. We use the 32-instance LLAMA2-70B workload in §6.2 as a case study to show the effect of TP with Mixed Degrees presented in 5.3. When the TP with mixed degrees (32 and 8) is enabled in §6.2, PP degree is 4 and selective activation checkpoint is also enabled. To disable the TP with mixed degrees, we use a unified TP degree 8, which is the largest TP degree can be applied to the number of key-value attention heads. We then set PP degree to 16, so that each DP replica is sharded among four instances, same as when the TP with mixed degrees is enabled. In the setting of unified TP degree 8, we have to disable selective activation checkpoint and use full-transformer-layer activation checkpoint [27] instead to avoid the device OOM during the training, due to the activation memory pressure caused by the TP degree 8. The end-to-end throughput of the setting with unified TP degree decreases by 25% compared to the throughput of the setting with mixed TP degrees. The main root cause of the throughput degradation comes from the larger number of pipeline stages and the overhead of full-transformer-layer activation checkpoint imposed by using a unified TP degree 8.

Asynchronous Execution in Neuron Runtime. We use the five 32-instance workloads in §6.2 as case studies to show the effect of asynchronous execution in Neuron Runtime discussed in §4.4. The number of graphs traced in the background was empirically determined: 3 for LLAMA2 7B, 7 for LLAMA2 13B and 70B, and 5 for GPT 23B and 175B. The comparison of MFPI with and without enabling asynchronous execution in Neuron Runtime is show in Figure 8. When enabling asynchronous execution in Neuron Runtime, we observe MFPI improvement for all the workloads. The MFPI improvement ratio due to asynchronous execution in Neuron Runtime enlarges as increasing the size of LLAMA2 and GPT models, respectively. Specifically, the MFPI improvement ratio increases from 15% to 119% as the model size of LLAMA2 increases from 7B to 70B, and the MFPI improvement ratio increases from 43% to 102% as the model size of GPT increases from 23B to 175B. This is because the overhead of computation graph tracing and NEFF loading becomes more significant as the model size increases. Asynchronous execution in Neuron Runtime successfully reduces the overhead so that it improves the MFPI.

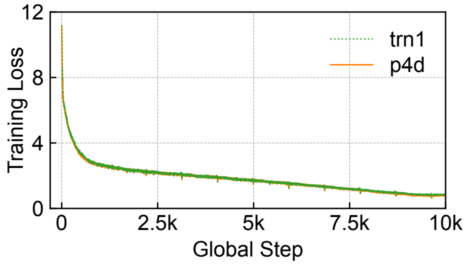


Figure 9: LLAMA2-7B training loss curves from *trn1* and *p4d* instances.

6.6 Fidelity Check

To check the fidelity of the distributed training on *trn1*, we first train a LLAMA2-7B model from scratch using a small dataset on *trn1* and *p4d*, respectively, and compare the loss curves of the trained models. We use the wikicorpus dataset [44] and apply the same training hyperparameters for both training on *trn1* and *p4d*. The training loss curves of the first `10k` global steps are shown in Figure 9. During the `10k` global steps of the LLAMA2-7B training, the training loss curves from *trn1* and *p4d* both decrease monotonically and follow each other without a consistent winner. For a more thorough study, NxDT is used to train a LLAMA2-like model over 1.8 trillion tokens [16]. The performance of the trained model is benchmarked against popular open source baseline models. Our model trained on *trn1* achieves the same quality as public models trained via GPU, *i.e.*, matching the evaluation values across tasks including commonsense reasoning, world knowledge, math, coding, etc.

7 DISCUSSION

7.1 Lessons, Limitations and Future works

Enabling distributed LLM training on homegrown deep learning accelerators is never easy. Making it product-ready for customers is even harder. In this subsection, we would like to share some lessons learned, limitations to be conquered, and works to be done.

We would like to highlight two key insights from the lessons learned:

- **Leveraging XLA v.s. Building from Scratch.** Many distributed LLM training users use PyTorch. To compile PyTorch scripts to be executable on Trainium, we have attempted to convert PyTorch scripts to our own build-from-scratch computation graph representations [55] as the inputs of Neuron Compiler. Building from scratch demands significant efforts in terms of development and maintenance, especially when PyTorch upgrades with numerous new features. Leveraging XLA that is natively supported by PyTorch minimizes the efforts, allowing us to concentrate on the development and optimizations specific to our accelerators. One of our future works is to adopt the PyTorch 2.0 compiler [5].
- **Framework Level v.s. Compiler Level Optimizations.** Based on our practice, framework level optimizations yield a higher return-on-investment than compiler level optimizations. After several years' development, Neuron compiler has done substantial optimization (*e.g.* [58]) to generate highly efficient NEFFs in most of the cases, making a diminishing return of further enhancement. Empirically, in the context of distributed LLM training, each optimization within Neuron Compiler, such as improving the overlap between collective operations and computations, typically generate 1-3% speedup after considerable efforts. On the other hand, framework level optimizations potentially have much higher gain if applicable. For example, applying sequence parallelism and selective activation checkpoint together achieves around 45% throughput improvement in one of our LLAMA2 7B workloads. Given the maturity of CUDA kernels, we believe the same observation also applies to GPU.

While Neuron SDK provides comprehensive support to workloads running on top of AWS Trainium, it has some limitations. For example, Neuron Compiler currently only accepts computation graphs with static tensor shapes as inputs and does not support dynamic tensor shapes. Any change in tensor shape within a computation graph triggers a new compilation to generate a new NEFF, resulting in considerable overhead. This is manageable in LLM training as the tensor shapes between training iterations are mostly static. For the seldom case of dynamism, we make a workaround to apply the bucketing technique [46] to pad a dynamic tensor size to its corresponding fixed-size bucket. However, extending to other workloads like MoE may introduce performance challenges. Addressing these concerns is part of our planned future work.

Moving forward, the scope of Amazon EC2 instances powered by Amazon homegrown accelerators is rapidly expanding. AWS Trainium2 has been announced and expects to be launched in 2024 with 4× faster than AWS *trn1*, which powers the next generation of Amazon EC2 ultra clusters with 65 exaFLOPS of on-demand supercomputing performance. Meanwhile, we are actively maintaining and adding new features into Neuron SDK based on customer requests. On one hand, we are broadening the scope of supported model types to include Mixture-of-Experts [17], Multi-Modal [35], *et al.* On the other hand, we are developing techniques to improve the cluster reliability and reduce the recovery time upon a failure [53].

7.2 Customers and Other Workloads

The *trn1* has been adopted internally and externally. External customers include companies such as Anthropic, Databricks, RICOH, and many others. Meanwhile, we collaborate with open-source communities like PyTorch, Jax, XLA, HuggingFace, and others to continually expand our support for various ML workloads.

Although this paper primarily focuses on distributed training of LLMs on *trn1*, *trn1* is versatile and can be applied to a variety of other workloads. *trn1* is well-suited for inferencing LLMs, such as GPT and LLAMA, and can also be used for other model types, including image and video encoders, as well as diffusion models for image and video generation.

8 RELATED WORKS

Deep learning accelerators. To accelerate Deep Learning training and to save costs, many customized chips are manufactured in industry. Google builds Cloud TPU [19] since 2016. Meta release their first-generation inference chip, MTIA [33], in 2023. Intel has Habana [22] chips for training and inference. AMD has data center GPU [4] for AI workloads. Other than these large companies, many startups join the competitions building advanced AI chips, *e.g.*, SambaNova [45], GraphCore [20], Cerebras [8], etc. Similar to AWS Trainium, these chips need dedicated software stacks for maximizing the performance and user experiences.

Distributed training libraries. DeepSpeed [42], MiCS [56], and PyTorch FSDP [57] adopt the partitioned data parallelism [41] techniques, which shards the model states onto multiple accelerators to reduce the memory footprint of the standard data parallelism. Megatron-LM/NeMo [27, 34] mainly uses 3D parallelism techniques to distribute computation and memory consumption in training. Colossal-AI [29] integrates various parallelism strategies and features a flexible system design, allowing for the easy combination of different parallelism strategies. These software libraries are implemented for GPU platform. NxDT follows the same design principles as these libraries do. It provides unified interfaces to support those techniques, and is tailored specifically to AWS Trainium.

Optimizations for computations. Other than libraries to support distributed training capabilities, another direction for accelerating the training aims to generate efficient computing operations at the lower level. FlashAttention [14, 15] improves the computation efficiency and reduces the memory footprints of transformer's attention computation by tiling computations. RingAttention [30] distributes long sequences across multiple devices by leveraging blockwise computation of self-attention and feedforward layers, and fully overlaps the communication of key-value blocks with the computation of blockwise attention. Ansor [59], TVM [10], TASO [23], and NVFuser [36] can generate high-performance tensor programs for multiple backends. Sputnik [18] provides efficient sparse matrix operations for GPUs. These works are orthogonal to the design of NxDT. Some of these techniques are implemented in the compiler part of Neuron SDK, *e.g.*, operator fusions. We are actively adding these optimizations that are beneficial on Trainium.

9 CONCLUSION

This paper presents our design choices and experience in developing and optimizing distributed LLM training on Amazon EC2 *trn1* instances. Our evaluation demonstrates that *trn1* achieves better or comparable performance to *p4d* while incurring lower costs. Leveraging AWS Neuron SDK, the additional work required for users to adapt existing GPU-based training scripts to AWS Trainium is reasonable. Consequently, *trn1* emerges as a compelling alternative for AWS customers seeking distributed LLM training solutions. More broadly, it serves as an affirmation within the community, signaling a promising trajectory for the widespread adoption of deep learning accelerators in the near future. We foresee a landscape marked by the flourishing growth of deep learning accelerators, which will likely handle a considerable portion of LLM workloads in cloud environments such as AWS.

ACKNOWLEDGEMENTS

We thank the AWS Neuron team for their support of making this work possible. We also thank the anonymous reviewers and our shepherd Prof. Luo Mai for their insightful comments and feedback.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[3] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. Falcon-40B: an open large language model with state-of-the-art performance. (2023).

[4] AMD. 2024. AMD Instinct™ MI300 Series Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi300.html

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Cerebras. 2024. Cerebras is the go-to platform for fast and effortless AI training. https://www.cerebras.net/

[9] Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. 2024. Slapo: A schedule language for progressive optimization of large deep learning model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1095–1111.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11, 20 (2018).

[11] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.

[12] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[13] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[14] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[15] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with

io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[16] Haozheng Fan, Hao Zhou, Guangtai Huang, Parameswaran Raman, Xinwei Fu, Gaurav Gupta, Dhananjay Ram, Yida Wang, and Jun Huan. 2024. HLAT: High-quality Large Language Model Pre-trained on AWS Trainium. *arXiv preprint arXiv:2404.10630* (2024).

[17] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research* 23, 1 (2022), 5232–5270.

[18] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*.

[19] Google. 2017. An in-depth look at Google's first Tensor Processing Unit (TPU). https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[20] Graphcore. 2024. Designed for AI: Intelligence Processing Unit. https://www.graphcore.ai/products/ipu

[21] HuggingFace. 2023. Spread Your Wings: Falcon 180B is here. https://huggingface.co/blog/falcon-180b

[22] Intel. 2024. Built for the Large-Scale Era of AI. https://habana.ai/

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[24] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–14.

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[26] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. 2019. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019).

[27] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023).

[28] Hsiang Tsung Kung and Charles E Leiserson. 1979. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, Vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA, 256–282.

[29] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*. 766–775.

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

[31] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[32] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162* (2018).

[33] Meta. 2023. MTIA v1: Meta's first-generation AI inference accelerator. https://ai.meta.com/blog/meta-training-inference-accelerator-AI-MTIA/

[34] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. 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*. 1–15.

[35] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. 2011. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*. 689–696.

[36] NVIDIA. 2023. A Fusion Code Generator for NVIDIA GPUs. https://github.com/NVIDIA/Fuser.

[37] NVIDIA. 2024. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/

[38] OpenXLA. 2024. A machine learning compiler for GPUs, CPUs, and ML accelerators. https://github.com/openxla/xla

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[41] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[42] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[43] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. Torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems* 4 (2022), 638–651.

[44] Samuel Reese, Gemma Boleda, Montse Cuadros, Lluís Padró, and German Rigau. 2010. Wikicorpus: A word-sense disambiguated multilingual wikipedia corpus. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*.

[45] SambaNova. 2024. Generative AI built for the Enterprise. https://sambanova.ai/

[46] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems* 3 (2021), 208–222.

[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[48] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990* (2022).

[49] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. 2021. LazyTensor: combining eager execution with domain-specific compilers. *arXiv preprint*

arXiv:2102.13267 (2021).

[50] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2002–2011.

[51] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[53] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 364–381.

[54] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[55] Cody Hao Yu, Haozheng Fan, Guangtai Huang, Zhen Jia, Yizhi Liu, Jie Wang, Zach Zheng, Yuan Zhou, Haichen Shen, Junru Shao, et al. 2023. RAF: Holistic Compilation for Deep Learning Model Training. *arXiv preprint arXiv:2303.04759* (2023).

[56] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. 2022. MiCS: near-linear scaling for training gigantic model on public cloud. *Proceedings of the VLDB Endowment* 16, 1 (2022), 37–50.

[57] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch FSDP: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[58] Hongbin Zheng, Sejong Oh, Huiqing Wang, Preston Briggs, Jiading Gai, Animesh Jain, Yizhi Liu, Rich Heaton, Randy Huang, and Yida Wang. 2020. Optimizing memory-access patterns for deep learning accelerators. *arXiv preprint arXiv:2002.12798* (2020).

[59] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[60] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.