Overhead-communication Exploration in Large-Scale Machine Learning Frameworks

Wai Cheuk Chadwick Leung

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-130
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-130.html

May 15, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Overhead-communication Exploration in Large-scale Machine Learning Frameworks

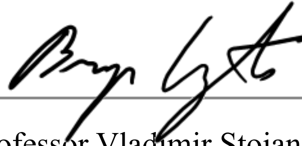
by Author Chadwick Leung

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Vladimir Stojanovic

Research Advisor

5/13/2022

(Date)

* * * * * * *

Professor Sophia Shao

Second Reader

5/14/2022

(Date)

Overhead-communication Exploration in Large-scale Machine Learning Frameworks

Chadwick Leung

13 May 2022

Abstract

This project studies the impact of overhead-communication in large-scale machine learning models on throughput and resource utilization. Recent machine learning scale-out frameworks such as ZionEX and ZeRO-Infinity, have shown the importance of increasing interconnect bandwidth on computation efficiency; in this project, we will measure the impact of overhead communication and its interconnect bandwidth on the GShard Mixture-of-Experts architecture. We measured and analyzed the performance of the training model using the Google Cloud Platform and version-3 TPUs, and its profiling tool, Tensorboard. The results showed that the communication portion of the training process increases as we increase the model size and scale-out the model. As a result, given the trend of increasing model size in machine learning to improve accuracy, it is important to scale interconnect bandwidth with respect to model size to maintain computation efficiency.

Table of Contents

1. Introduction
    1.1. Recent Related Frameworks
        1.1.1. ZionEX
        1.1.2. ZeRO-Infinity
        1.1.3. Evaluation Framework

2. GShard Mixture-of-Experts Model
    2.1. Model Architecture
    2.2. Key Operations

3. Evaluation Structure and Framework
    3.1. Google Cloud Platform
    3.2. Tensorboard Platform
    3.3. Run and Capture the model training

4. GShard Mixture-of-Expert Code Profiling
    4.1. Training Flow and Structure
    4.2. Modules Descriptions
    4.3. Key Modules and Params

5. Data and Result
    5.1. Tensorboard raw data
    5.2. Parsed Result

6. Conclusion and Future Direction

7. References

1. Introduction

Machine learning is widely used nowadays in different areas, from server level search engines to smart home devices; it has become an integral part of our lives and it is essential to have better predictions from the machine learning models. To improve model performance and accuracy, we need bigger models with more parameters to capture extra details of the training samples and consider them in predictions. However, increasing the model size and the number of parameters is limited by hardware capabilities. One compute node does not have the memory space required to store a large-scale model. Therefore, different forms of parallelism emerged to solve the scaling problem. Data parallelism can be used to scale training to distributed devices, with each device working on a different piece of data and combining their result at the end. Model parallelism can partition the large-scale model into multiple devices, such that each partition agrees with the device memory capacity. Pipeline parallelism breaks tasks into a sequence of processing stages to increase throughput. Combining the three parallelisms, we can distribute a large-scale model into multiple devices and solve the scaling problem.

While using parallelism to construct a large-scale machine learning model solves the memory capacity problem, it brings us other challenges. First, in order to implement parallelisms, machine learning scientists would need to refactor their code to partition the model. This involves changing the model architecture and sharding the model wisely (choosing which parts to be partitioned and what not to). However, changing the model architecture would require users to also provide an effective communication strategy in order to maintain the model throughput.

Second, the nodes that hold the distributed model may have sequential dependencies and cannot achieve true parallelism. For example, a simple partitioned model that uses model parallelism would not utilize the resources efficiently, since parts of the model would have to stall and wait for the output from its parent parts. This leads to under-utilization of the resources and would require users to either change the model architecture or migrate the code to a specific framework.

Third, for each of the nodes to communicate with each other, it requires overhead communication to transfer data between nodes such as dispatching data to the devices or aggregating the results computed by the devices. This communication is essential since it allows the model to be distributed into multiple devices. However, the overhead communication is hindering the model performance in terms of training time and prediction time. While the nodes

are communicating with each other using the overhead communication network, the nodes are under-utilized as they are waiting for the data to be sent or received. However, current frameworks do not size communication bandwidth as fast as model size, and this would lead to data congestion/traffic in the overhead communication network, which results in extended latency.

There are many existing large-scale frameworks that use the parallelism strategies, and each has a unique way of tackling the challenges that it brings. Learning about the recent related large-scale machine learning model training frameworks allows us to compare the computation efficiency of different frameworks. Then, we would study the overhead communication of the GShard Mixture-of-Experts machine learning model. Investigation of this model can help us understand the overall compute efficiency and learn what scaling problems that model can solve. Furthermore, it would illustrate the importance of having higher communication bandwidth in an efficient machine learning framework.

1.1 Recent Related Frameworks

1.1.1 ZionEX

In the ZionEX paper, it introduces NEO, a software-hardware co-designed system for high-performance distributed training of large-scale deep learning recommendation models. It combines table-wise, row-wise, column-wise, and data parallelism when training massive models and uses ZionEX, a new hardware platform based on the co-designed with NEO's 4D parallelism for optimizing communications for large-scale model training.

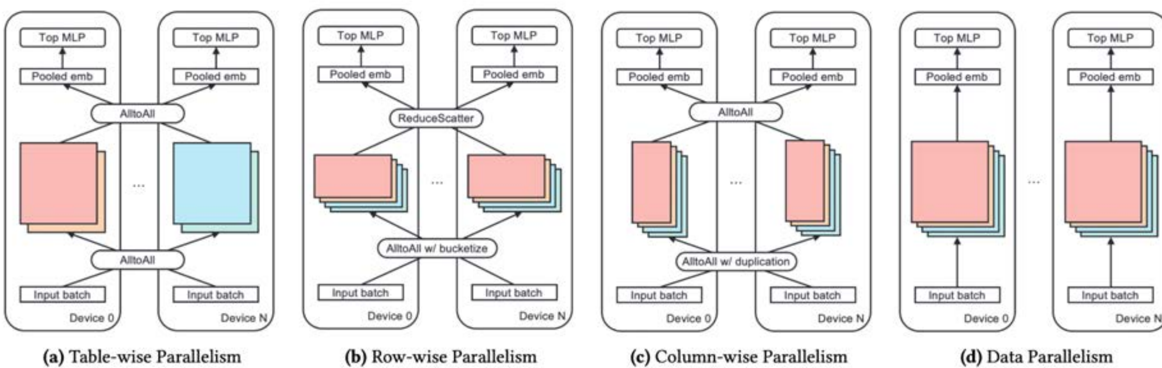


Figure 1 [5]. 4D parallelism introduced in the ZionEX paper.

The system provides the above parallelism strategies, and users can test different combinations of them to determine the best partition strategy when constructing large-scale machine learning models. Note that this hybrid 4D parallelism strategy requires all-to-all communications for results sharing and input distributions. The all-to-all communication is often the bottleneck for computation performance due to limited communication bandwidth and data dependencies among devices. Therefore, the ZionEX paper also introduced ZionEX, a revised version of their previous hardware platform design, Zion, to improve the communication bandwidth.

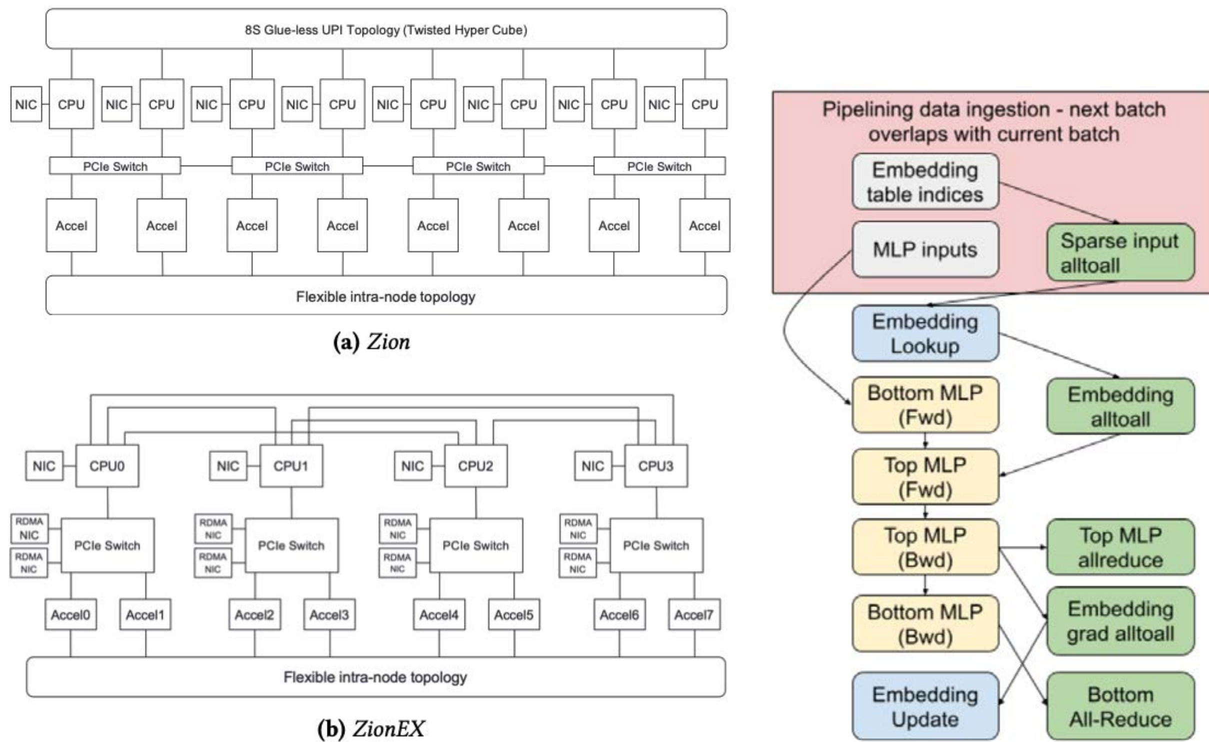


Figure 2&3 [5].

2: ZionEX framework and its parent framework, Zion.

3: Data dependency graph using the ZionEX framework.

The tasks would be divided into GPUs and CPUs based on their operation, with compute heavy tasks allocated to GPUs and relatively cheaper tasks allocated to CPUs; CPUs and GPUs would then need to transfer data to each other to perform a complete run. The ZionEX architecture aims to solve the problem of increased overhead communication such as data transfers between CPUs and GPUs and inter-socket communication.

One optimization that ZionEX made is to increase the communication bandwidth of the GPUs by introducing RDMA NIC (network interface cards) to each of the GPUs connected to the PCIe switch (peripheral component interconnect express, an interface standard for connecting high-speed components switch). This allows extra communication between nodes compared to the old architecture, Zion, where inter-node communication had to go through the PCIe, then interrupt the CPU in order to gain access to the normal NIC before transferring data. In other words, transferring data between nodes would require CPU intervention and additional GPU-CPU communications.

Furthermore, the communication of this RDMA NIC uses an isolated network from the common communication network (original NICs). Therefore, "each GPU can communicate with GPUs on a different node through the dedicated low-latency high-bandwidth RoCE NIC, without involving host CPUs." [5]

1.1.2 ZeRO-Infinity

In the ZeRO-Infinity paper, it introduces a novel heterogeneous system that supports large-scale model sizes on limited GPU resources by exploiting CPU and NVMe memory simultaneously. Users can use fewer GPU nodes to construct a large model since parts of the model will be distributed to CPU or NVMe memory. The entire model is partitioned into multiple devices using model parallelism and parameters such as tensor weight could be distributed among GPUs and CPUs/NVMe memory.

The ZeRO-Infinity framework uses a GPU-CPU memory architecture. This architecture favors overhead communication since it increases the bandwidth by allowing communication between CPU or NVMe to GPUs, where GPUs would be able to talk to other nodes using the GPU network, and at the same time can talk to its connected CPU and NVMe using a separate network. The distributed model utilizes data 3D parallelism; data parallelism to distribute the training, model parallelism to partition the model to fit device memory, pipeline parallelism to pipeline the training process. Similar to the ZionEX paper introduced in the previous section, the distributed devices would need to communicate with each other and transfer data to make a complete training step.

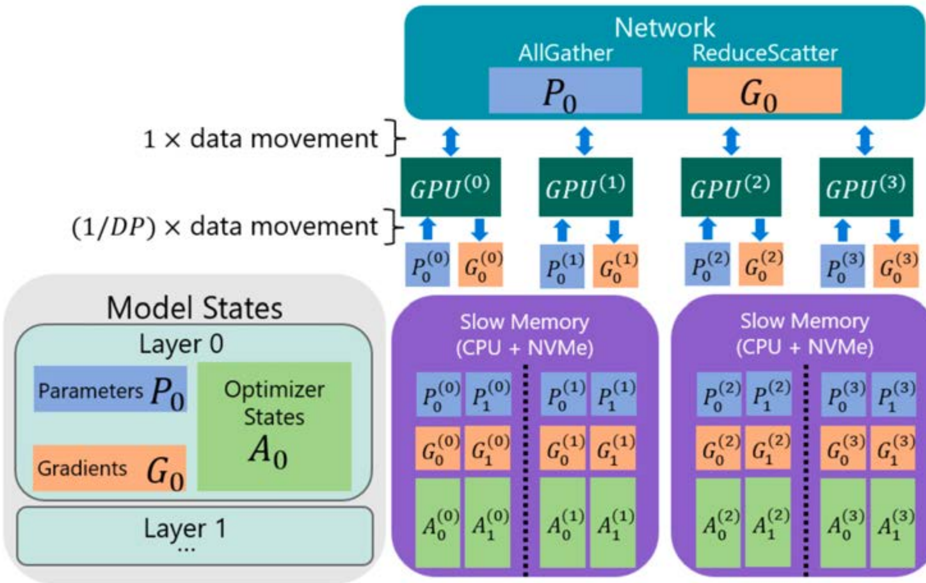


Figure 3 [4]. ZeRO-Infinity framework.

The data would be transferred using a broadcast-based/all-gather approach; since each piece of data is owned by one of the devices, the data must be first read from the NVMe to CPU memory, then copy the read data to GPU memory, and finally allow inter-node data transfer. Therefore, the total communication time would be the sum of the above three data transfer times. To optimize the communication bandwidth, the paper uses a bandwidth-centric partitioning design, where each GPU is allowed to request data from its CPU/NVMe in parallel, thus, each adding 1/DP data to the communication process, whereas DP represents the degree of data parallelism. Moreover, with the overlap centric design, ZeRO-Infinity overlaps GPU-GPU communication with GPU computation, NVMe-CPU communication, and CPU-GPU communication to hide the sequential nature of the NVMe to GPUs data movements. The following bandwidth table shows that the communication bandwidth increases with the above architecture, and “the effective communication bandwidth between NVMe or CPU to the GPU, increases linearly with the DP degree.” [4]

Nodes	GPUs	Aggregate Memory (TB)			GPU-GPU Bandwidth (GB/s)	Memory Bandwidth/GPU (GB/s)		
		GPU	CPU	NVMe		GPU	CPU	NVMe
1	1	0.032	1.5	28.0	N/A	600-900	12.0	12.0
1	16	0.5	1.5	28.0	150-300	600-900	3.0	1.6
4	64	2.0	6.0	112.0	60-100	600-900	3.0	1.6
16	256	8.0	24.0	448.0	60-100	600-900	3.0	1.6
64	1024	32.0	96.0	1792.0	60-100	600-900	3.0	1.6
96	1536	48.0	144.0	2688.0	60-100	600-900	3.0	1.6

Table 1 [4]. The table shows the extra bandwidth that the architecture provides by providing CPU and NVMe bandwidth for communication.

1.1.3 Evaluation Framework

From the above examples, we can see that the current trend to have improve scale-out efficiency is to increase interconnection bandwidth; adding switches to nodes can increase the interconnect communication and increase the computation efficiency since less time would be spent on talking to other nodes and would allow us to better utilize the resources.

In the ZionEX paper, it introduces a way to estimate the overall per-iteration latency [5],

$$T_{fwd} = max(BotMLP_{fwd}, Embedding_lookup + alltoall_{fwd}) + Interaction_{fwd} + TopMLP_{fwd}$$

$$T_{bwd} = max[TopMLP_{bwd} + Interaction_{bwd} + max(alltoall_{bwd} + Embedding_update, BotMLP_{bwd}), TopMLP_Allreduce + BotMLP_Allreduce]$$

$$T_{total} = T_{fwd} + T_{bwd}$$

We can see that the main components of the estimation are computation time and communication time.

On the other hand, the ZeRO-Infinity paper introduces an interesting way to quantify the relationship between computation and communication [4], it uses a function of ait (arithmetic intensity, the ratio between the total computation and the data required by the computation), compute time, and communication time to measure the system's efficiency.

$$compute_time = \frac{total_computation}{peak_{tp}}$$

$$ait = \frac{total_computation}{total_data_movement}$$

$$communication_time = \frac{total_data_movement}{bw}$$

$$efficiency = \frac{compute_time}{compute_time + communication_time}$$

In this project, we will use the efficiency calculation to measure the computation efficiency. Both compute time and communication time measurements would be introduced in a later section.

2. GShard Mixture-of-Experts Model

2.1 Model Architecture

The GShard Mixture-of-Experts (MoE) paper introduced us GShard, a lightweight annotation API as the XLA compiler extension that provides an easier way to achieve parallel computation with little to no changes to the model source code. This essentially solves the problem of refactoring model codes discussed in the previous section. Users no longer need to focus on customizing the model architecture to fit the framework, instead, they can just "plug and play".

The GShard MoE model uses the Transformer [2] architecture as its building block. By stacking multiple Transformer layers, we can form an encoder or a decoder. An encoder is formed by using a self-attention layer [2] followed by a position-wise feed-forward layer [reference number]; a decoder layer is formed by appending a cross-attention layer to the encoder, which purpose is to attend over the encoder's outputs. The GShard Mixture-of-Experts model replaces every other feed-forward layer with a Top-2 gating layer followed by a Position-wise Mixture-of-Experts (MoE) layer.

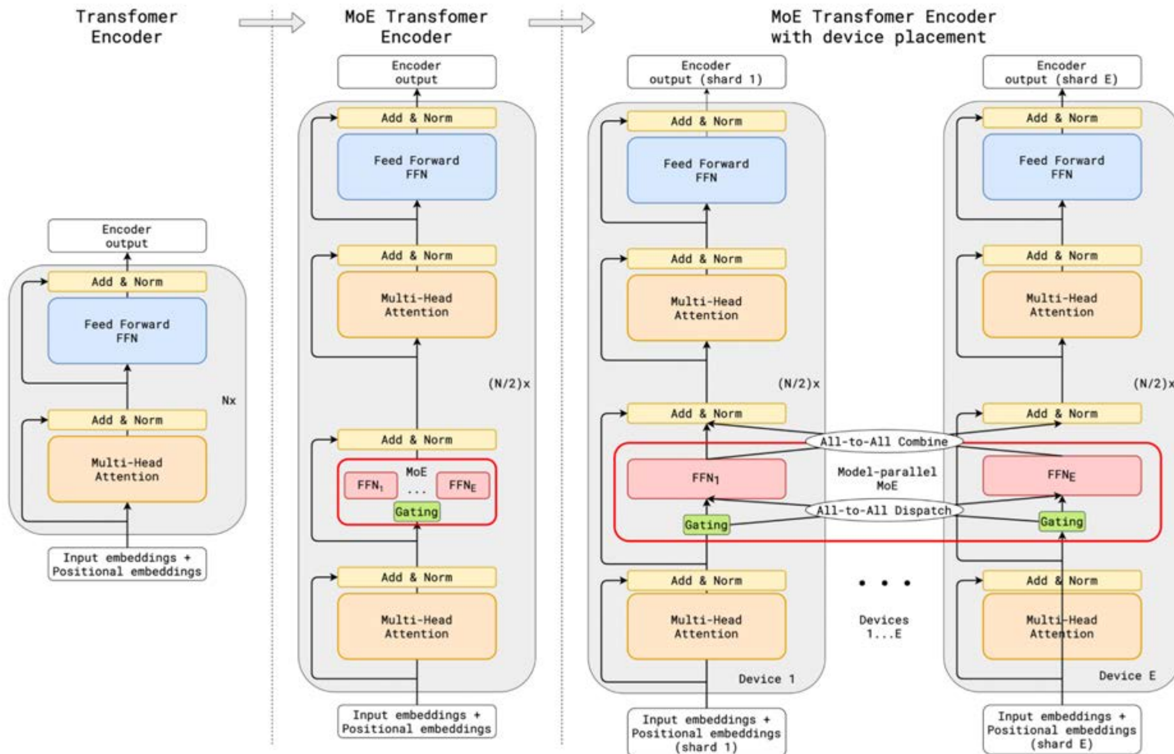


Figure 4 [6]. GShard MoE architecture, with alternating FFN-MoE layers.

The Top2-gating layer serves as a distributor, which chooses the best expert to dispatch the input token to. The ideal scenario is that each expert focuses on one aspect of the training, and the gating function would then choose the best expert to work on the input token based on the expert's expertise. Therefore, it is possible that a second expert is not chosen for the input token, as they might have found the most suitable expert to deploy the job already. One thing to notice is that in order to prevent the gating function from always choosing the same few experts, the paper introduced an auxiliary loss function, which keeps track of expert capacities and the input tokens they received and aims to average the amount of input token received by each expert.

Algorithm 1: Group-level top-2 gating with auxiliary loss

Data: x_S, a group of tokens of size S
Data: C, Expert capacity allocated to this group
Result: $\mathcal{G}_{S,E}$, group combine weights
Result: ℓ_{aux}, group auxiliary loss

(1) $c_E \leftarrow 0$ ▷ gating decisions per expert
(2) $g_{S,E} \leftarrow softmax(wg \cdot x_S)$ ▷ gates per token per expert, wg are trainable weights
(3) $m_E \leftarrow \frac{1}{S} \sum_{s=1}^{s} g_{s,E}$ ▷ mean gates per expert
(4) **for** $s \leftarrow 1$ **to** S **do**
(5) $\quad g1, e1, g2, e2 = top_2(g_{s,E})$ ▷ top-2 gates and expert indices
(6) $\quad g1 \leftarrow g1/(g1 + g2)$ ▷ normalized $g1$
(7) $\quad c \leftarrow c_{e1}$ ▷ position in $e1$ expert buffer
(8) $\quad$ **if** $c_{e1} < C$ **then**
(9) $\quad\quad \mathcal{G}_{s,e1} \leftarrow g1$ ▷ $e1$ expert combine weight for x_s
(10) $\quad$ **end**
(11) $\quad c_{e1} \leftarrow c + 1$ ▷ incrementing $e1$ expert decisions count
(12) **end**
(13) $\ell_{aux} = \frac{1}{E} \sum_{e=1}^{E} \frac{c_e}{S} \cdot m_e$
(14) **for** $s \leftarrow 1$ **to** S **do**
(15) $\quad g1, e1, g2, e2 = top_2(g_{s,E})$ ▷ top-2 gates and expert indices
(16) $\quad g2 \leftarrow g2/(g1 + g2)$ ▷ normalized $g2$
(17) $\quad rnd \leftarrow uniform(0, 1)$ ▷ dispatch to second-best expert with probability $\propto 2 \cdot g2$
(18) $\quad c \leftarrow c_{e2}$ ▷ position in $e2$ expert buffer
(19) $\quad$ **if** $c < C \land 2 \cdot g2 > rnd$ **then**
(20) $\quad\quad \mathcal{G}_{s,e2} \leftarrow g2$ ▷ $e2$ expert combine weight for x_s
(21) $\quad$ **end**
(22) $\quad c_{e2} \leftarrow c + 1$
(23) **end**

Figure 5 [6]. Note that on line 17, the second expert would be chosen based on a probability proportional to the second expert gating weight.

After getting the gating weights, it would then dispatch the input tokens to the respective experts based on the gating weights and the expert capacities. Most of the computations are covered by the Einum Tensorflow operation. The Einsum operation is a powerful operation that can do matrix reordering and matrix multiplication. After the dispatch operation (line 3&4), each expert would then compute the expert results using a series of Einsum and ReLU operations (line 5-7). Then, the experts results would be aggregated through the combine operation, which is also an Einsum operation that combines the expert results with respect to their gating weights.

Algorithm 2: Forward pass of the Positions-wise MoE layer. The underscored letter (e.g., G and E) indicates the dimension along which a tensor will be partitioned.

```
1 gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
2 combine_weights, dispatch_mask = Top2Gating(gates)
3 dispatched_expert_inputs = einsum(
4     "GSEC,GSM->EGCM", dispatch_mask, reshaped_inputs)
5 h = einsum("EGCM,EMH->EGCH", dispatched_expert_inputs, wi)
6 h = relu(h)
7 expert_outputs = einsum("EGCH,EHM->GECM", h, wo)
8 outputs = einsum(
9     "GSEC,GECM->GSM", combine_weights, expert_outputs)
```

Figure 6 [6]. Algorithm representation of the GShard MoE architecture.

The paper then proceeded to evaluate the training efficiency of the mentioned architecture. The evaluation model is designed to have the number of experts equal to the number of cores used, and each expert would have the same dimension as the regular Feed Forward Network. The MoE layers are distributed in every other Transformer layer (Figure 4), and there are 12/36 layers in total (18 FFN-MoE combo layers). The following figures show the evaluation framework and the results.

Id	Model	Experts Per-layer	Experts total	TPU v3 Cores	Enc+Dec layers	Weights
(1)	MoE(2048E, 36L)	2048	36684	2048	36	600B
(2)	MoE(2048E, 12L)	2048	12228	2048	12	200B
(3)	MoE(512E, 36L)	512	9216	512	36	150B
(4)	MoE(512E, 12L)	512	3072	512	12	50B
(5)	MoE(128E, 36L)	128	2304	128	36	37B
(6)	MoE(128E, 12L)	128	768	128	12	12.5B
*	MoE(2048E, 60L)	2048	61440	2048	60	1T

Table 2 [6]. The evaluation framework from the GShard paper. Note that for (1) and (2), the correct experts total should have been 36864 and 12288.

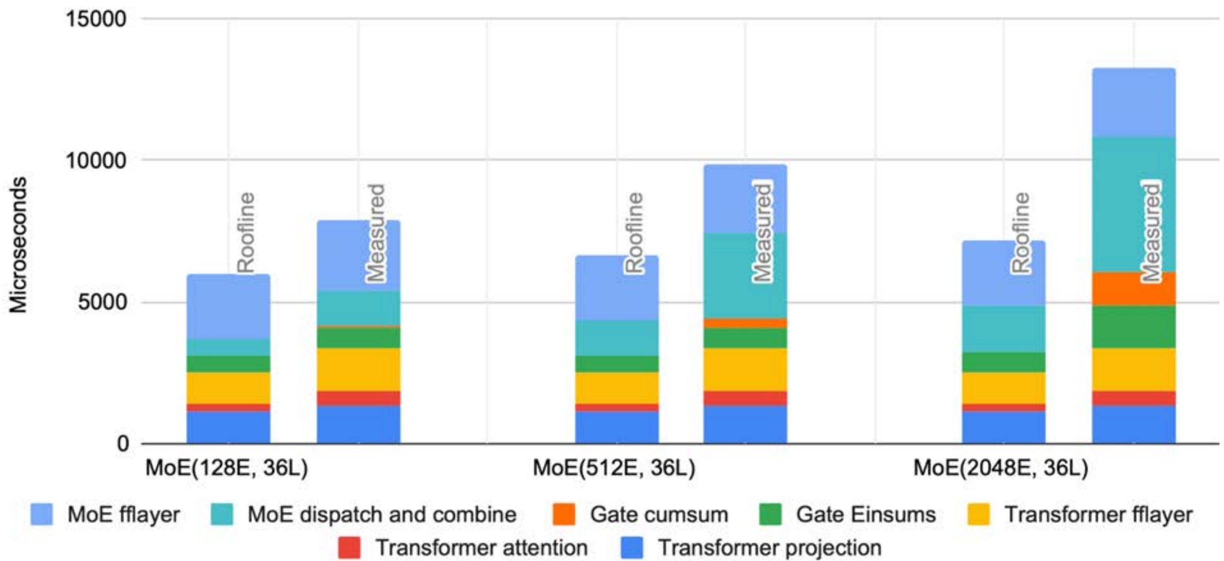


Figure 7 [6]. The timing result for each operation as an aggregate model with 36 layers. Note that MoE dispatch and combine represents the all-to-all communication.

The roofline data is an estimation of the training time; we can see that the measured model underperformed the roofline model, this is because as the model size increases, the overhead communication between nodes becomes harder, as there are more nodes need to talk to each other with an unchanged communication throughput between nodes. The most critical communication operation is all-to-all (MoE dispatch and combine). This result agrees with the results found in the two papers introduced in the previous section (ZionEX and ZeRO-Infinity), where communication bandwidth has become the bottleneck for large-scale machine learning models.

In this project, we would like to quantify the effect of communication time on machine learning model performance and computation efficiency, especially the ratio between compute and interconnect.

2.2 Key operations

Replication and sharding are two key operations for this model to perform parallelisms. Replication is required to copy the data in the form of tensors across different nodes, for instance, replication is used to replicate the weights for the Top-2 gating function as the gating function of each node must have the same gating results and agree upon the best expert for a certain input token. Sharding is required to partition the tensor into smaller parts along some dimension and place them onto different devices. For example, the Mixture-of-Experts layer will have to be sharded across the TPU cores (one expert per core) to achieve model parallelism. The replication and sharding operations are achieved using the following XLA operations,

Einsum

> Einsum allows defining Tensors by defining their element-wise computation. "It is the most critical operator in implementing the MoE model. They are represented as a Dot operation in XLA HLO." [6] For example, when doing resharding, the dispatch operation requires the tensor to change its partitioning dimension, therefore, a Einsum is done locally first before performing the all-to-all operation.

CollectivePermute

> "This operator specifies a list of source-destination pairs, and the input data of a source is sent to the corresponding destination. It is used in two places: changing a sharded tensor's device order among partitions, and halo exchange." [6]

AllGather

> "This operator concatenates tensors from all participants following a specified order. It is used to change a sharded tensor to a replicated tensor." [6]

AllReduce

> "This operator performs elementwise reduction (e.g., summation) over the inputs from all participants. It is used to combine partially reduced intermediate tensors from different partitions. In a TPU device network, AllReduce has a constant cost when the number of partitions grows." [6]

AllToAll

> "This operator logically splits the input of each participant along one dimension, then sends each piece to a different participant. On receiving data pieces from others, each participant concatenates the pieces to produce its result. It is used to reshard a sharded tensor from one dimension to another dimension. AllToAll is an efficient way for such resharding in a TPU device network, where its cost increases sublinearly when the number of partitions grows." [6] This operator is used extensively in the dispatch (distribution of input token to respective experts) and combine (aggregate experts results based on respective gating weights) operations, and thus, the domination communication primitive in the overhead interconnect.

3. Evaluation Structure and Framework

3.1 Google Cloud Platform

Resources required:

- Virtual Machine: To store our code and give commands
- TPU cores: Optimized for machine learning operations such as vector operations and matrix multiplication
- Google Cloud storage: To store the intermediate training weights and the model information such as params/hyperparams
- Cloud network: To connect all the above tools

1. Permissions setup

Menu bar -> IAM & Admin -> IAM

Required roles:

- Editor
- Project IAM Admin
- Project Lien Modifier
- Project Mover
- Storage Admin: Required to access Cloud storage (detailed in Cloud Storage setup)
- TPU Admin

2. Cloud network setup

Menu bar -> VPC network -> VPC networks -> Create a VPC network

- Name: network name
- Subnet creation mode: Automatic
- Firewall rules: Create the firewall rules (detailed below)
- Dynamic routing mode: Regional
- Maximum transmission unit: Default, 1460

3. Firewall setup

Menu bar -> VPC network -> Firewall -> CREATE FIREWALL RULE

The firewall is set to allow ssh-ing to the VM instance

- Logs: Whether we want to generate log or not
- Network: Choose the created network
- Priority: Default (1000)
- Direction of traffic: One each: Ingress (incoming traffic, for connecting to VM through ssh); Egress (outbound traffic, for outputting data when using tensorboard)
- Action on match: Allow
- Targets: All instances in the network
- Source filter: IPv4 ranges
- Source IPv4 ranges: 0.0.0.0/0, 192.168.2.0/24
- Second source filter: None
- Protocols and ports: Allow all (for simplicity, allow all; tensorboard needs access to port 6006)

4. Cloud Storage setup

Menu bar -> Cloud Storage -> Browser -> CREATE BUCKET

- Name your bucket: bucket name (We will have to create 2 buckets, one for storing model information/intermediate values, and one for tensorboard profiling data)
- Location type: Region, choose the region that you plan to use the resources, (detailed in Virtual Machine setup); the Region location type is also the cheapest
- Default storage class: Standard
- Control access to objects: DO NOT prevent public access, as we need to access data using Tensorboard
- Access control: Uniform
- Protection tools: None

5. Virtual Machine setup

Menu bar -> Compute Engine -> VM instances (Under VIRTUAL MACHINES) -> CREATE INSTANCE

- Name
- Region: For v3 (required to use v3) 16-core or more, choose europe-west4 only, as it is the only region that provides 16 or more cores
- Zone: For v3 16-core or more, choose europe-west4-a only, it is the only zone that provides 16 or more cores
- Series: Default
- Machine type: Default
- Display device: Default
- Confidential VM service: Default

6. TPU setup

Note that we are using Cloud TPU nodes but not TPU VMs.

Menu bar -> Compute Engine -> TPUs (Under VIRTUAL MACHINES) -> CREATE TPU NODE

- Name
- Zone: see above section for zone selection
- TPU settings: TPU node architecture
- TPU type: v3-# (has to use v3; note that devices greater than 8 nodes cannot be stopped, it can only be deleted after done running)
- TPU software version: refer to the required Tensorflow version in the README in https://github.com/tensorflow/lingvo.git
- Network: Select the created network
- IP address range: Leave empty

7. Virtual Machine + TPU alternative setup

Instead of doing step 5 & 6, we can utilize the Cloud Shell to create a VM + TPU execution group. This method is preferred as the TPU and VM created are put into the same execution group.

Cloud shell command:

gcloud compute tpus execution-groups create --name=[TPU_NAME] --zone=[ZONE] --tf-version=[TENSORFLOW_VERSION] --accelerator-type=[v3-NUMBER_OF_CORES] --network=[NETWORK_NAME]

- TENSORFLOW_VERSION should match the required tensorflow version stated in the README in https://github.com/tensorflow/lingvo.git
- NETWORK_NAME should match the created network name

3.2 Tensorboard Platform

3.2.1 Tensorboard Setup

Tensorboard is a profiling platform that is installed as part of Tensorflow; by using tensorboard, we can capture the TPU node information such as the TPU FLOP utilization and the time taken for each tensorflow operation.

To use tensorboard, we first have to ssh to the Virtual Machine through the 6006 port, using the following gcloud command on GCP shell,

gcloud compute ssh [TPU_NAME] --zone=[ZONE] --ssh-flag=-L6006:localhost:6006

To capture the TPU usage data, we have to capture the data at the model training stage, in which we have added custom logging statements to indicate the current training step of the training procedure. Note that it is important to capture the data at the right time, otherwise it wouldn't be able to capture any useful data. The preferred moment to capture the training data is when the custom logging statement prints "done training." When the "done training" flag is seen, we can capture the TPU usage information using the following command,

capture_tpu_profile --tpu=[TPU_NAME] --logdir=gs://[PROFILE_BUCKET] --duration_ms=[DURATION]

Note that PROFILE_BUCKET must not be the same bucket used for storing intermediate training data when running the model, and users can specify DURATION to capture more training steps.

To access the tensorboard data via the Tensorboard platform, we first have to delete the event file in the GCP cloud bucket, which indicates that the tensorboard data is empty (this file is auto-generated after each capture and must be deleted before accessing the tensorboard data).

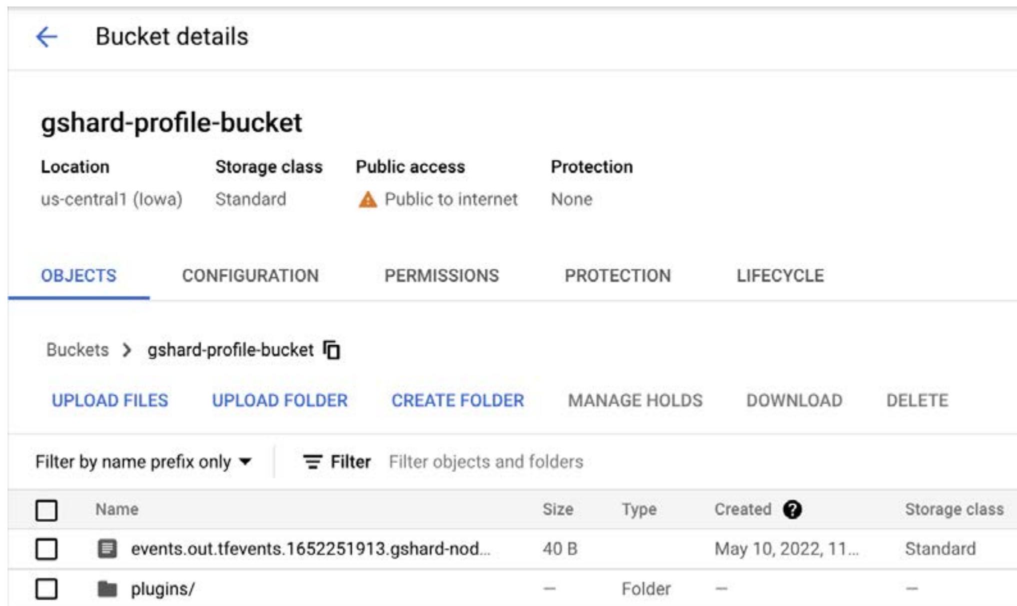


Figure 8. Must delete events.out.tf.events… before accessing Tensorboard.

The **Trace Viewer** tool shows a detailed breakdown on when and how the operation is called and executed. For example, it shows how long each training step took and the parent-child relationship of the Tensorflow operation being called and executed. Note that in model training, there are tensorflow operations and XLA operations, where the tensorflow operation represents the operation initialized by the model, and the XLA operations represents the operations initialized by the compiler. When inspecting the operations and communication pattern, it is more accurate to observe the XLA operations as some of the tensorflow operations would be pipelined as multiple XLA operations.

The **Op Profile** tool displays the performance of the XLA operations, which shows how well the model training uses the TPU resources in terms of TPU FLOPS utilization. It also shows the average step time of the training procedure and provides details of individual operations, such as shape, padding and expressions that use the operation. However, small operations such as

all-reduce in smaller nodes would not show up due to its small utilization. We would use a custom parser to retrieve and analyze the result.

3.3 Run and Capture the model training

1. Start both the Virtual Machine and TPU node

    We can either

    - Use the GCP Console to start the resources

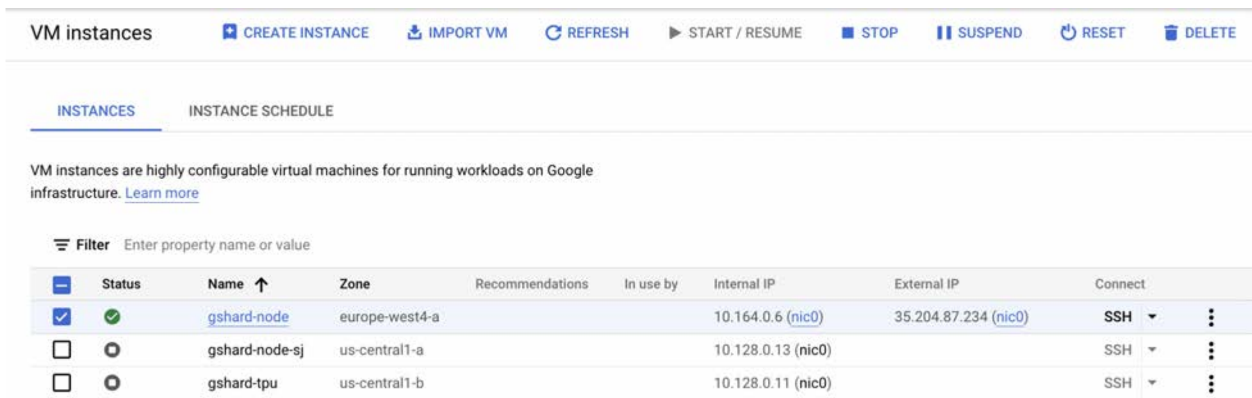


Figure 9. Starting Virtual Instance using the GCP console.

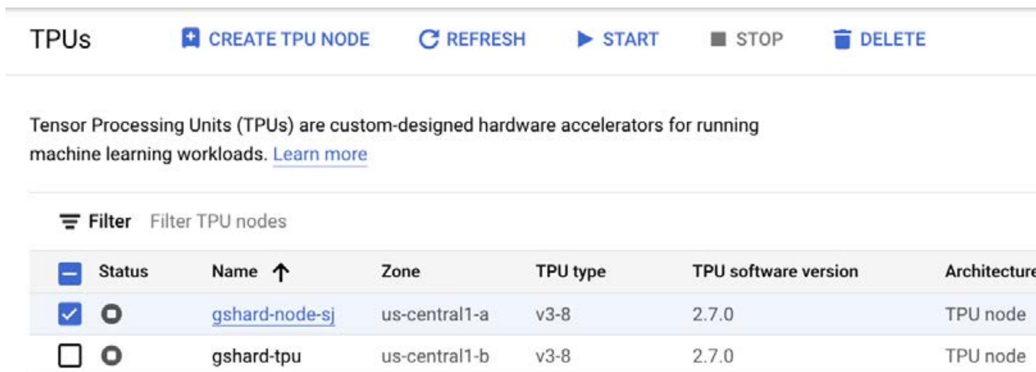


Figure 10. Starting the TPU node using the GCP console.

    - Use the Cloud Shell command to start the resources

      gcloud alpha compute tpus start [TPU_NAME] --zone=[ZONE]

2. SSH to Virtual Instance

We can either

- Use GCP Console to connect to the Virtual Instance
- Use Cloud Shell to connect to the Virtual Instance

3. Check available resources

In the Virtual Instance shell, the command to check the availability is

gcloud alpha compute tpus list --zone=[ZONE]

4. Resources configuration setup

This step is required every time we connect to the Virtual Instance; each Virtual Instance shell (if opening many) would need to be configured in order to utilize the resources (i.e. TPU nodes) outside of our region/zone.

gcloud config set project [PROJECT_NAME]

gcloud config set compute/region [REGION]

gcloud config set compute/zone [ZONE]

5. Code setup

This step is only required when the Virtual Instance is first created, skip this step if the following has been done.

In the Virtual Instance shell, the setup commands are

- sudo apt-get update
- sudo apt-get install git
- sudo apt-get install python3-pip
- git clone https://github.com/tensorflow/lingvo.git
- # Get docker
- curl -fsSL https://get.docker.com -o get-docker.sh
- chmod +x get-docker.sh
- ./get-docker.sh
- gcloud auth configure-docker
- sudo usermod -aG docker $(whoami)
- sudo systemctl restart docker

- # Lingvo docker installation
- sudo docker build --tag tensorflow:lingvo_dev_lm - < lingvo/docker/dev.dockerfile

6. Run the GShard training model

   In Virtual Machine:

   - **export TPU_NAME=[TPU_NAME]**
   - **sudo docker run --rm -it -v /home/$(whoami)/lingvo:/tmp/lingvo -e TPU_NAME=${TPU_NAME} --name lingvo tensorflow:lingvo_dev_lm bash**

   In Docker:

   - **export LOGDIR=gs://[TRAIN_BUCKET]**
   - **bazel run -c opt //lingvo:trainer -- --mode=sync --alsologtostderr --model=lm.synthetic_packed_input.[MODEL_NAME] --logdir=${LOGDIR} --tpu=${TPU_NAME} --worker_split_size=[NUMBER_OF_CORES] --ps_replicas=[NUMBER_OF_CORES / 8] --job=executor_tpu --disable_tf2=true**

7. Stop the resources after running

   Note that TPU nodes greater than 8 can only be deleted.

Figure 11. Stop or delete the TPU node using the GCP console.

4. GShard Mixture-of-Expert Code Profiling

4.1 Training Flow and Structure

The *trainer.py* module would parse the command line flags, determine which training model to be used, and define the Google Cloud bucket to be used to store intermediate training data as checkpoints.

It will first initialize the infrastructure to train the model, for example, it would establish a connection to the provided TPU cluster and configure the TPU cluster such as defining the number of devices to be used and provide the model distribution details to the cluster sharder. Then, it will fetch the params based on the flags it received by calling the *get_params.py* method. It will find the training schedule parameters to determine how to train the model, such as when to make a forward and backward propagation, define the learning rate of each training step, and how many steps it should train the model based on the maximum training step parameter.

It will construct the model based on the hyperparameters provided in *synthetic_packed_input.py*. It specifies the model

After building the model and having the training schedule set up, *trainer.py* will start the training process and record the intermediate training data to the specified Google Cloud bucket.

4.2 Modules Descriptions

Key modules detailed descriptions:

trainer.py:

1. Take in command line params
2. Configure the Cloud TPUs
3. Call helper function to get the params needed to instantiate the TPU executor
4. Create TPU executor/runner
5. Run the TPU executor

executor.py:

1. Helper function to retrieve the params needed to instantiate the TPU executor

2. Create an runner that does arbitrary multi-program execution on TPU
    a. During the construction process, all programs would construct their subgraphs, which would be aggregated to form a mega-graph
    b. After creating the graph, a sequence of programs is then executed associated with the task specified in the params retrieved
3. Execute the runner after done constructing the model

synthetic_packed_input.py:

1. Set up model architecture parameters such as model dimension on Feed-Forward layer and batch size
2. Set up training inputs
3. Set up training parameters such as learning rates and number of training iterations/steps

gshard_builder.py:

1. Defines the model architecture based on the training parameters. For example, it defines the number of layers and model dimension of the FFN. This module then construct the TPU graph that is specified in the GShard research paper using the building blocks specified in *gshard_layer.py*
2. Supply default sharding annotations to building blocks in *gshard_layer.py*

gshard_layer.py:

1. This module contains the building blocks of the model. It contains blocks such as Top-2 Gating, Feed Forward Neural Network layer, Mixture-of-Expert layer, Attention layer
2. Depending on the training parameters, this module provides a sharding function for the above layers to be distributed to multiple devices and achieve model parallelism.
3. When running the model, this module provides the forward and backward propagation method for each of the layers

program.py:

1. Initialize the training program and supply random data as training input to the model. Since the purpose of the project is to investigate the communication pattern, we decided to use random inputs as training data.

2. We have added custom logging statements to indicate the current training step; this is to notify users when to capture the TPU information using the tensorboard command

4.3 Key Params

The *synthetic_packed_input.py* file defines the model parameters; by changing the parameters introduced in the following part of this section, users can change the model architecture and training behavior.

The *Task* method in this file defines the model hyperparameters such as model dimension for the Feed-Forward layer and batch size. The important parameters are:

NUM_DEVICES_PER_SPLIT:

> Defines the number of TPUV3 cores used for the model; it is the maximum number of partitions for the model.

DEVICE_MESH_SHAPE:

> A tuple of two integers *(x, y)*, each of which defines the number of partitions along some dimension of a tensor or variable. The product *x* and *y* must equal to *NUM_DEVICES_PER_SPLIT*.

BATCH_DIM_PER_DEVICE:

> The batch size that each device (core) would take; we have chosen the batch size per core to be 0.125, and the batch size per TPU node to be 0.125 * 8 = 1. Note that the total batch size would equal to the product of *BATCH_DIM_PER_DEVICE* and *NUM_DEVICES_PER_SPLIT*.

NUM_TRANSFORMER_LAYERS:

> Defines the number of transformer layers in the model. The GShard MoE architecture is defined to have an alternating layer of normal Feed Forward Network (FFN) layer and MoE layer. Since we are more interested in the communication pattern within the FFN-MoE combo layer, we have chosen the number of transformer layers to be 2, one each for FFN and MoE.

model_dim:

The model dimension for the Feed Forward Neural Network. It is the key component of the Transformer layer; a larger model dimension would be able to capture more features of the training data and potentially give a more accurate training model. When changing the *model_dim* parameter, note that it is constrained by the TPU's high-bandwidth memory.

moe:

An boolean flag that indicates the use of Mixture-of-Experts (MoE) layers. A MoE layer will be used in every other Transformer layer, which replaces the normal Feed Forward Network with a MoE layer.

num_groups:

Defines the number of groups the input batch would be divided into; the groups are processed in parallel. Each group is assigned a fractional capacity of each expert, this way, we can ensure that the input tokens dispatched to the experts is balanced

e_dim:

Defines the number of experts existing in the training model. To match the research paper, we chose to make the number of experts match the number of cores used in the training model.

c_dim:

Defines the input token capacity that each expert can take. The expert capacity is used to ensure that the input tokens received by the experts is balanced. This is to avoid the situation where all the input tokens are dispatched to the same expert and under-utilize the other experts.

5. Data and Result

5.1 Tensorboard raw data

One key finding is that we found that the combined operation would not show up due to random routing of the second expert; we confirmed this finding by referencing other papers [1] and [3]. The idea behind this is that the second expert would only be chosen by a small random probability based on the gating weights; if the weight for the second expert is small enough, we can ignore the second expert being chosen and thus, save expert capacity for the second expert. Since this project is interested in the communication pattern and utilization of the TPUs but not the accuracy of the training model, we have set our maximum training steps to be around 100, it is very likely that the second expert would not be chosen at all, and therefore, the all-to-all combine operation would not be seen.

The captured data can be viewed using the Trace Viewer tool, which gives an interactive view of the time taken for each operation. Since the tensorflow operations would often be pipelined as multiple XLA operations, it is more accurate to account for the XLA operations than the original tensorflow operations. For example, the Einsum operation would be pipelined into multiple XLA Fusion operations.

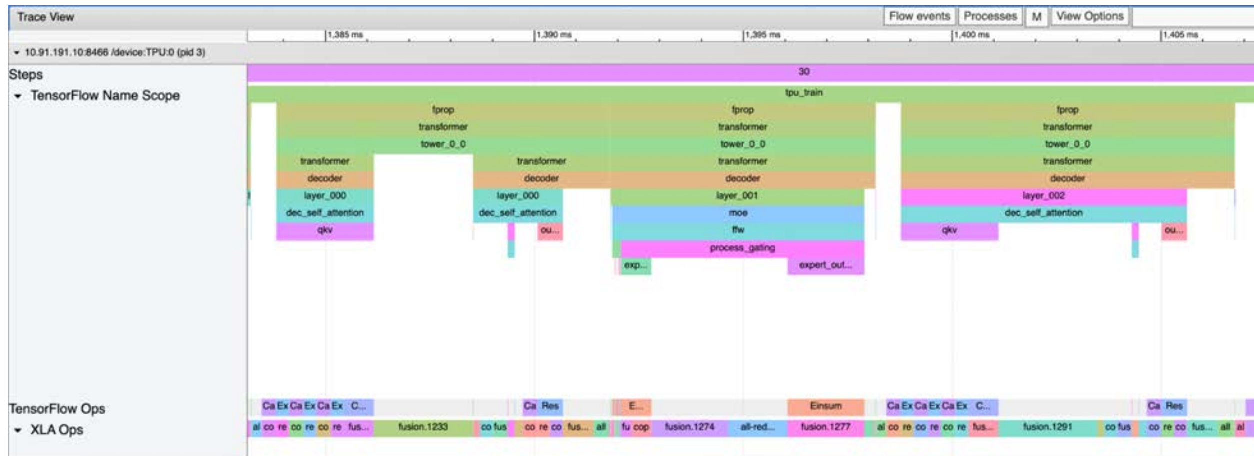


Figure 12. Trace viewer that shows the timeline for the model training.

The captured result is also available in json format, where we would later use a custom parser to process the data.

```
{"byCategory":{"children":[{"children":[{"children":[{"children":[{"children":[],"metrics":
{"flops":0,"memoryBandwidth":0,"rawBytesAccessed":0,"rawFlops":4398046511104,"rawTime":0,"t
ime":0},"name":"convolution-base-dilated.750","numChildren":0,"xla":
{"category":"convolution base-dilated","expression":"%convolution-base-dilated.750 =
bf16[128,16,256,16,256]{4,2,3,0,1:T(8,128)(2,1)} convolution(bf16[16,1024,128,256,1]
{1,3,4,2,0:T(8,128)(2,1)} %fusion.4417, bf16[16,1024,16,256,1]{3,1,2,4,0:T(8,128)(2,1)}
%fusion.4416), window={size=16x1x16 stride=15x1x1 pad=0_0x0_0x15_15 lhs_dilate=16x1x1
rhs_reversal=0x0x1}, dim_labels=0f1b2_0i2o1-\u003e10b2f","layout":{"dimensions":
[{"alignment":128,"semantics":"feature","size":1024},
{"alignment":8,"semantics":"batch","size":256},
{"alignment":0,"semantics":"spatial","size":1},
{"alignment":0,"semantics":"spatial","size":128},
{"alignment":0,"semantics":"spatial","size":16}]},"op":"","provenance":"tpu_train/fprop/
transformer/tower_0_0/transformer/decoder/layer_009/moe/ffw/process_gating/
expert_inputs_egcm/Einsum:"}},{"children":[{"children":[],"metrics":
{"flops":0,"memoryBandwidth":0,"rawBytesAccessed":0,"rawFlops":0,"rawTime":0,"time":0},"nam
e":"bitcast.4041","numChildren":0,"xla":{"category":"bitcast","expression":"%bitcast.4041 =
bf16[16,1024,16,256,1]{3,1,2,4,0:T(8,128)(2,1)} bitcast(bf16[16,1024,16,256,1]
{3,1,2,4,0:T(8,128)(2,1)} %bf16input.187)","op":"","provenance":"tpu_train/fprop/
transformer/tower_0_0/transformer/decoder/layer_009/moe/ffw/process_gating/
expert_inputs_egcm/Einsum:"}}],"metrics":
{"flops":0,"memoryBandwidth":0,"rawBytesAccessed":0,"rawFlops":0,"rawTime":0,"time":0},"nam
e":"fusion.4416","numChildren":1,"xla":{"category":"loop fusion","expression":"%fusion.4416
= bf16[16,1024,16,256,1]{3,1,2,4,0:T(8,128)(2,1)} fusion(bf16[16,1024,16,256,1]
{3,1,2,4,0:T(8,128)(2,1)} %param_1.12170), kind=kLoop,
calls=%bitcast_fusion.150.clone","op":"","provenance":"tpu_train/fprop/transformer/
tower_0_0/transformer/decoder/layer_009/moe/ffw/process_gating/expert_inputs_egcm/
Einsum:"}},{"children":[{"children":[],"metrics":
{"flops":0,"memoryBandwidth":0,"rawBytesAccessed":0,"rawFlops":0,"rawTime":0,"time":0},"nam
e":"bitcast.4042","numChildren":0,"xla":{"category":"bitcast","expression":"%bitcast.4042 =
```


Figure 13. Screenshot of parts of the json raw result.

5.2 Parsed Result

We have constructed a custom parser to better interpret the collected data. The custom result parser allows us to collect more accurate results, as we can look into the pipelined XLA operations instead of the Tensorflow operations. The custom parser can be found in the following repo,

https://github.com/sunjin-choi/gshard_eval.git

First, we looked into the relationship between the size of the model, in terms of the number of TPU cores used in the training model, to the ratio of overhead communication during model training. As shown in the table below, when the size of the model increases, where we use more TPU cores to construct our model, the percentage of overhead communication, in terms of all-to-all communication, increases superlinearly. The most significant increase is when we doubled the number of cores from 64 to 128, the time taken for all-to-all communication increased by 2.19x. It is expected that as we increase the number of nodes to 512 and 1024, the all-to-all communication time would increase by a greater extent. We then calculated the efficiency using the previously mentioned evaluation method,

$$efficiency = \frac{compute_time}{compute_time + communication_time}$$

num_cores	average_step_time (s)	compute (%)	all-to-all (%)	all-gather (%)	all-reduce (%)	miscellaneous (%)	tpu_flop_utilization
32	0.4146	40%	9%	13%	9%	29%	35%
64	0.6571	40%	12%	15%	5%	28%	36%
128	0.7495	37%	23%	12%	4%	24%	33%

num_cores	average_step_time (s)	compute (s)	all-to-all (s)	all-gather (s)	all-reduce (s)	miscellaneous (s)	efficiency
32	0.4146	0.16584	0.03731	0.05390	0.03731	0.12023	82%
64	0.6571	0.26284	0.07885	0.09857	0.03286	0.18399	77%
128	0.7495	0.27732	0.17239	0.08994	0.02998	0.17988	62%

Table 3. Key operations' contributions to one training step time, with overall TPU FLOP utilization as reference.

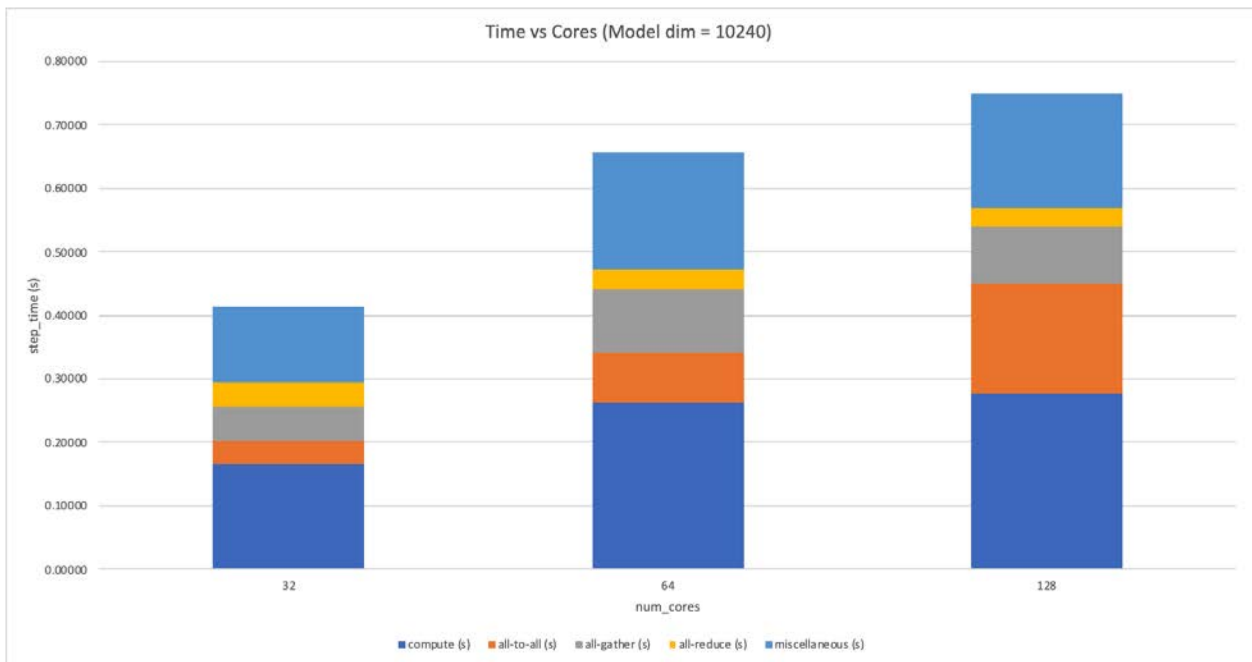


Figure 14. Relationship between operation latencies and number of cores used in the model.

The result is as expected, since the interconnection throughput between TPU nodes does not increase proportionally to the number of nodes as we increase the scale-out the training model, therefore, when there are more nodes trying to transfer data to each other, the overhead communication would have to stall for some of the transfers since the data transaction throughput did not scale fast enough.

We then inspected the relationship between model sizes and timing. A larger model dimension in the Feed Forward Network means there are more weights in the network and would require more computation per layer to complete one training pass. We have chosen the hidden layer dimension to be eight times the model dimension for simplicity. The maximum TPU FLOP utilization we were able to push to was 33%; model sizes larger than 10240 would result in TPU high-bandwidth memory exhaustion.

num_cores	model_dim	hidden_dim	average_step_time (s)	compute (%)	all-to-all (%)	all-gather (%)	all-reduce (%)	miscellaneous (%)	tpu_flop_utilization (%)
128	2048	16384	0.1282	21%	28%	16%	7%	28%	15%
128	4069	32768	0.256	27%	28%	15%	5%	25%	21%
128	8192	65536	0.5626	34%	25%	13%	4%	24%	30%
128	10240	81920	0.7495	37%	23%	12%	4%	24%	33%

num_cores	model_dim	hidden_dim	average_step_time (s)	compute (s)	all-to-all (s)	all-gather (s)	all-reduce (s)	miscellaneous (s)	efficiency
128	2048	16384	0.1282	0.026922	0.035896	0.020512	0.008974	0.035896	43%
128	4069	32768	0.256	0.069120	0.071680	0.038400	0.012800	0.064000	49%
128	8192	65536	0.5626	0.191284	0.140650	0.073138	0.022504	0.135024	58%
128	10240	81920	0.7495	0.277315	0.172385	0.089940	0.029980	0.179880	62%

Table 4. Relationship between model dimensions and operation latencies.

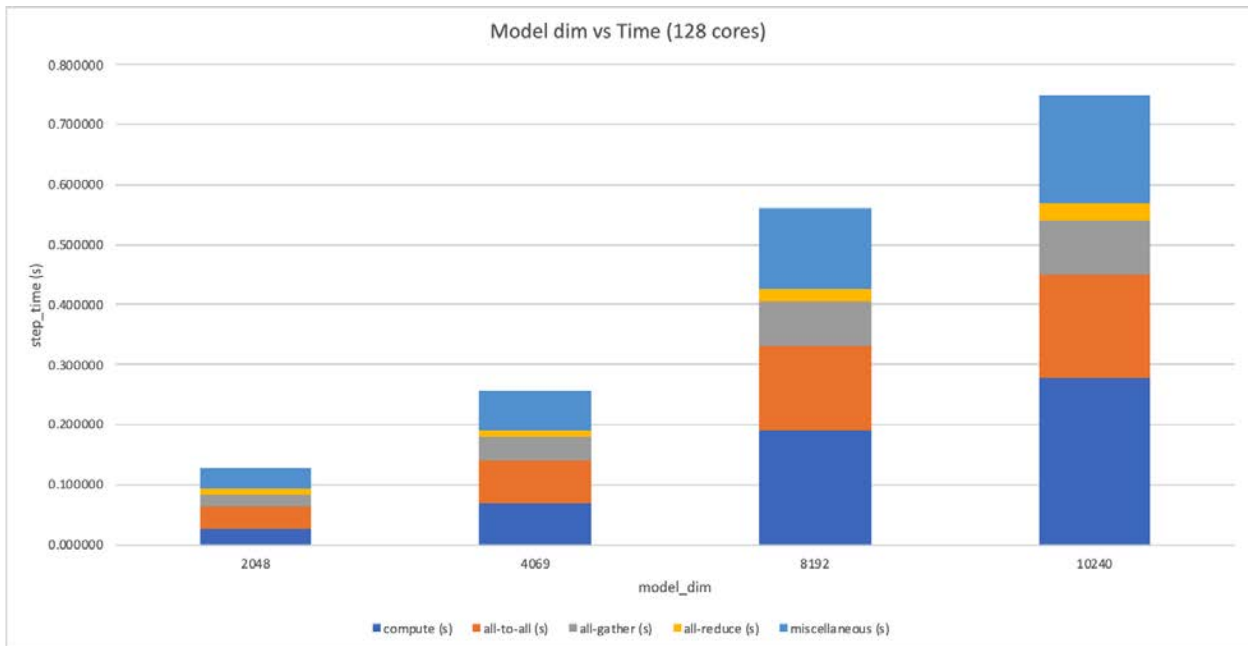


Figure 15. Relationship between operation latencies and model dimensions.

From the result, we can observe that a larger model size would have higher node utilization, and this can hide some of the overhead communication cost in the overall latency calculation.

However, due to memory capacity, we have a maximum model size to respect and could not further improve the efficiency.

6. Conclusion and Future Direction

To conclude, both this project and the recent large-scale machine learning framework papers have shown that interconnection bandwidth and overhead communication have become the bottleneck of efficient computation and resource utilization in large-scale machine learning frameworks. As machine learning models scale-out, they require efficient overhead communication to reduce node idle time and reduce the overall training or computing latency.

For example, in the ZeRO-Infinity, it presented the bandwidth requirement for its framework if we want to achieve more efficient computation in the future using the same topology,

	V100	10x	100x
Total devices	512	512	512
Achievable peak (pflops/device)	0.07	0.70	7.00
Slow memory bw requirement (GB/s per device)	3.0	30.0	300.0
Slow memory aggregate bw (TB/s)	1.5	15.0	150.0
GPU-to-GPU bw (GB/s)	70.0	700.0	7000.0

Table 5 [4]. Table of bandwidth requirements for ZeRO-Infinity to remain efficient on a cluster of 512 accelerator devices with 10x and 100x more achievable compute than NVIDIA V100 GPUs [4].

Furthermore, it also showed that as model size increases, the throughput decreases. This is due to interconnection bandwidth not sizing fast enough as the model size and lead to data conjunction in the overhead communication network.

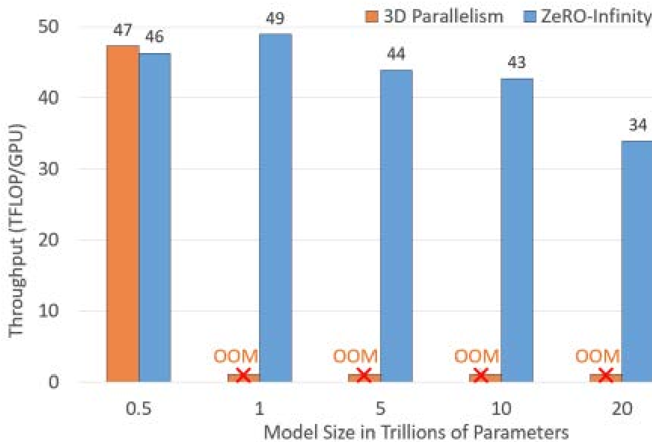


Figure 16 [4]. Throughput decreases as model size increases, using the same hardware framework.

The ZeRO-Infinity paper also showed the increase in efficiency as bandwidth increases, which again illustrates the importance of scaling bandwidth,

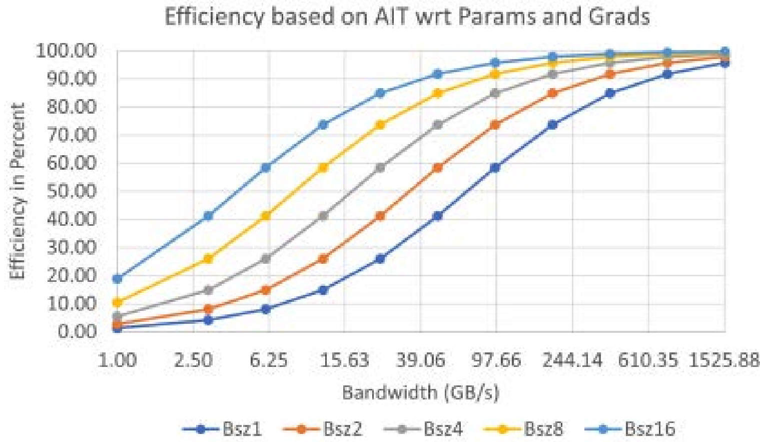


Figure 17 [4]. The impact of bandwidth on efficiency.

The ZionEX paper also showed the importance of increasing the bandwidth, as it can greatly reduce the overall latency and allow better resource utilization. We can see that as model size increases (from 8 GPUs to 128 GPUs), the communication primitives dominate the overall latency.

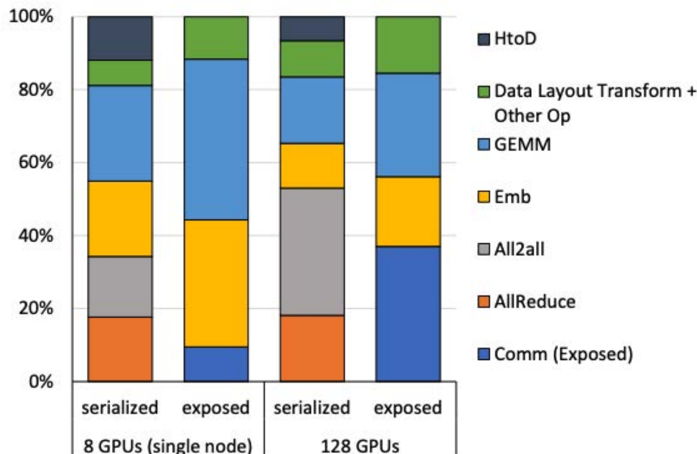


Figure 18 [5]. Dominant operator time breakdown (serialized and exposed time) per GPU. [5]

In the future, we hope to construct an architecture that maximizes the node abilities based on the data collected in this project. It would require sophisticated interconnect/routing networks and increase in communication bandwidth such as adding additional switches to the existing hardware frameworks.

7. References

[1] Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer." arXiv preprint arXiv:1701.06538 (2017).

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

[3] Nie, Xiaonan, Shijie Cao, Xupeng Miao, Lingxiao Ma, Jilong Xue, Youshan Miao, Zichao Yang, Zhi Yang, and Bin Cui. "Dense-to-Sparse Gate for Mixture-of-Experts." arXiv preprint arXiv:2112.14397 (2021).

[4] Rajbhandari, Samyam, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-14. 2021.

[5] Mudigere, Dheevatsa, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu et al. "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models." arXiv preprint arXiv:2104.05158 (2021).

[6] Lepikhin, Dmitry, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. "Gshard: Scaling giant models with conditional computation and automatic sharding." arXiv preprint arXiv:2006.16668 (2020).

[7] Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand 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, pp. 1-15. 2021.

[8] Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin et al. "{TensorFlow}: A System for {Large-Scale} Machine Learning." In 12th USENIX symposium on operating systems design and implementation (OSDI 16), pp. 265-283. 2016.