Efficient Giant Models Benchmarks and Performance Guide

YANPING HUANG, YUANZHONG XU, DEHAO CHEN, DMITRY LEPIKHIN, ZHIFENG CHEN, SHIBO WANG, TAO WANG, NOAM SHAZEER, MAXIM KRIKUN, and BLAKE HECHTMAN

1 INTRODUCTION

With GShard infrastructure, we are pioneering giant model training and serving infrastructure at Google. In this report we provide systems benchmarks of giant models and share the performance optimization techniques to achieve them.

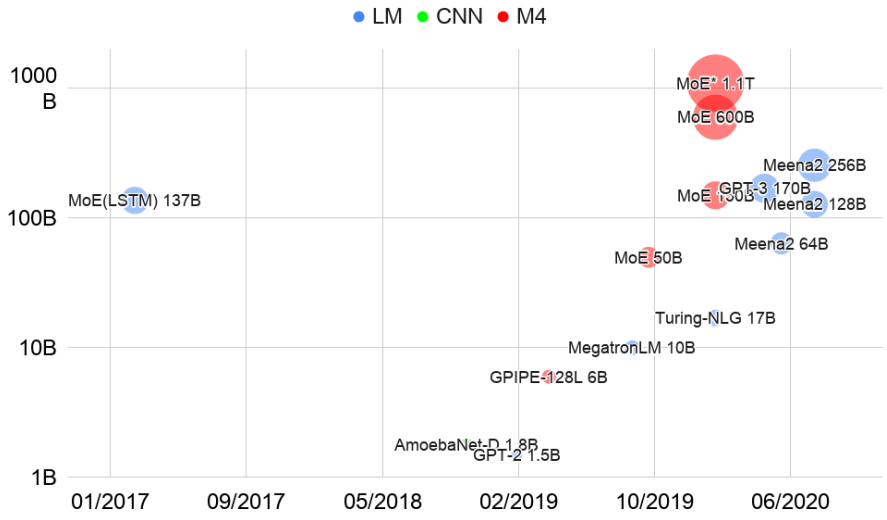

Fig. 1. Example giant mmodels between 2017 and 2020.

Giant models have been actively studied at Google and started to show direct impacts in various production areas. For example, the massively massive multilingual machine translation models (go/m4-projects) have been helping hundreds of millions of Google translate active users with better translation quality, especially those who spoke low resource languages (languages with smaller number of training examples for translation). Models distilled from giant m4 models with billions of parameters are now being served for 81 out

Authors' address: Yanping Huang; Yuanzhong Xu; Dehao Chen; Dmitry Lepikhin; Zhifeng Chen; Shibo Wang; Tao Wang; Noam Shazeer; Maxim Krikun; Blake Hechtman.

of a total of 107 supported languages by Translate. Moreover, training with a single multilingual model instead of hundreds of bilingual models saved years of TPU training time.

As shown in Figure 2, the BLEU score improvements from giant multilingual models are high for all languages. Especially the improvements are even higher for low-resource languages whose speakers represent billions of people in some of the world's most marginalized communities on Internet. Each rectangle on the figure represents languages with 1B speakers. The larger and deeper the model, the larger the BLEU score improvements were across all languages. Our 2.3B GPipe model in 2019 improved average quality by 6.1 BLEU, and the 600B GShard model in 2020 further increase the average quality by additional 7.4 BLEU. To put these gains into perspective, the 2016 Google neural machine translation launch (that transitioned from a statistical phrase-based translation system to neural network based system) improved the average quality by 5 BLEU, at the time one of the largest improvements in quality in product's history.

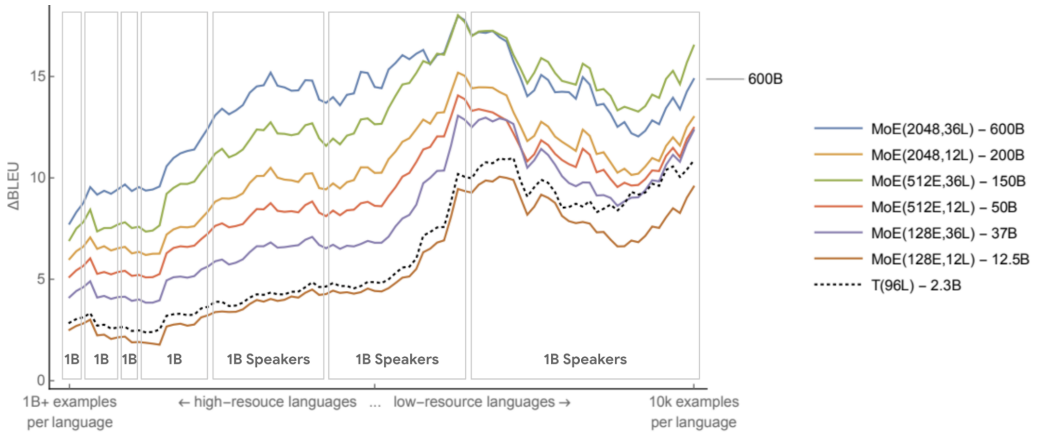

Fig. 2. Broader impact of giant M4 models.

There are growing need to train larger and larger models over the years, as shown in Figure 1. Cloud customers like ByteDance and AI21 started to pilot giant model training on cloud TPU using the GShard open source implementation (go/gshard-oss). Internally there are also multiple giant model initiatives such as M4 (go/m4-project), Meena GLM (go/glm-256b, go/glm-500b) and MuM (go/mum).

At the same time, we can't neglect the environmental impacts of training and serving giant models. The vast amount of cost for enabling giant model called for efficient infrastructure and tools. A 10% improvement in the step time could result in 10% less CO_2 emission.

2 DENSE TRANSFORMER SCALING

2.1 Einsum/Matrix multiplication case study.

Matrix multiplication (Matmul) and the more general Einsum are common operations widely used in language models. For example, the input projection computation in the transformer feed-forward layer involves Einsum of shape $[B, S, M] \times [M, H] \rightarrow [B, S, H]$ where B is the input batch size, S is the sequence length, M is the model dimension and H is the hidden dimension. This operation alone will require more than $O(BSMH) = 2.8 \times 10^{14}$ FLOPs to compute with typical dimensions of $B = 512$, $S = 1024$, $M = 8192$, and $H = 65536$, which will take 16 seconds on a DFC with 140 TFLOPS. Therefore, distributing computation over multiple devices is required to accelerate computation of this scale. Below we show a few sharding strategies for large scale Einsum/Matmul.

Figure 3 shows an example of partitioning a matrix multiplication operation over K devices, i.e., dividing operand matrices on both sides along the contracting dimension. In this way, each device will only compute $1/K$ of flops and save the partial local result on the device. Thus we need use an AllReduce to do the element-wise reduction to combine the partial results and introduce a communication overhead proportional to the size of the output matrix. This sharding strategy is ideal for matrices whose contracting dimension is much larger than the non-contracting dimension. However, each device needs to allocate the full sized tensor for the local result, which would cause device out of memory when the non-contracting dimensions are large.

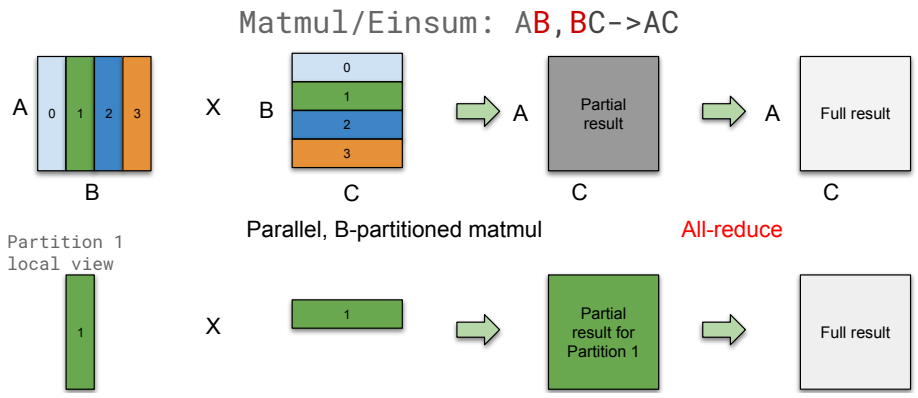

Fig. 3. A simple Einsum/Matmul partitioned on different contracting dimension over K devices. It required allocating a full-sized output tensor during the process.

Figure 4 shows Einsum/Matmul operations whose operands and output tensors are sharded by a group of K by K devices. Using this strategy, matrices will be partitioned into smaller pieces so that they can be fit in the device memory. First, operand tensors are subgrouped AllGather along the contracting dimension so that tensors become partially sharded along the non-contracting dimensions only. After AllGather, matrices

are partitioned into K pieces, each of which are replicated K times among K devices. Device (i, j) holds i-th row partition from LHS and j-th column partition from RHS and computes Einsum/matmul locally. The result buffer becomes the (i, j) partition of the output tensor. Using this 2D sharding strategy, there is no redundant computation and the communication cost comes from two subgrouped allgather operations each for one input operand.

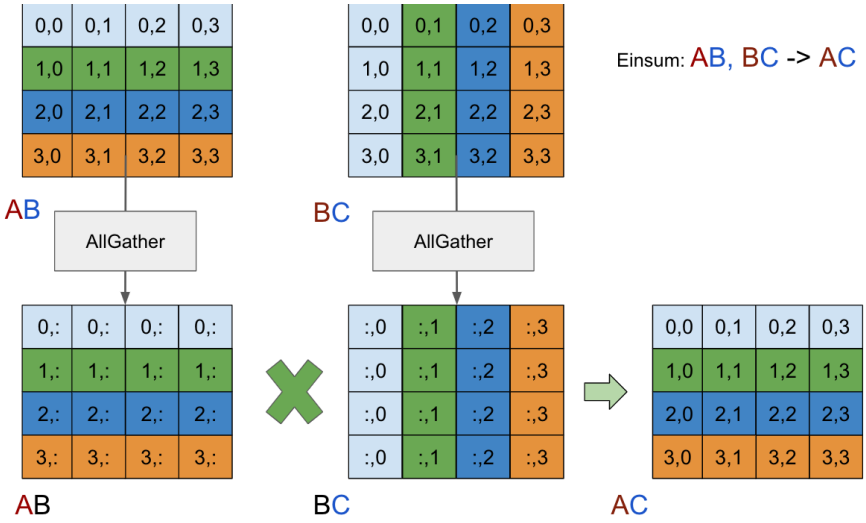

Fig. 4. Einsum/Matmul with 2D sharding. Both input operands and output tensor are sharded by a group of two-dimensional K by K devices. Per device

Figure 5 illustrated the case when input/output tensors of Einsum need to be partitioned along different non-contracting dimensions with 1D devices. Since operands have different non-contracting dimensions, we cannot compute the local Einsum directly. Replicating one of the operands would not cause redundant computation, but it required the replicated operand to fit in device memory. Instead, if the size of the operand is too large, we implemented an algorithm similar to Cannon's algorithm by keeping both operands partitioned and using a loop to iterate over each slice of the result. Finally `CollectivePermute` operations are used to circularly shift the input slices among the devices. A communication overhead proportional to the size of the output tensor is only required in this case.

2.2 Grouping and recursive partitioning

In Section 2.1 we discussed a few typical cases of Einsum partitioning. In practice, these cases could be applied to a single Einsum op in a nested way. We developed a framework for recursive partitioning to support nested cases.

First, we define a device context object for the partitioner, which defines how cross-partition collective operators are created based on given subgroups of partitions, and how partition ID is calculated. With

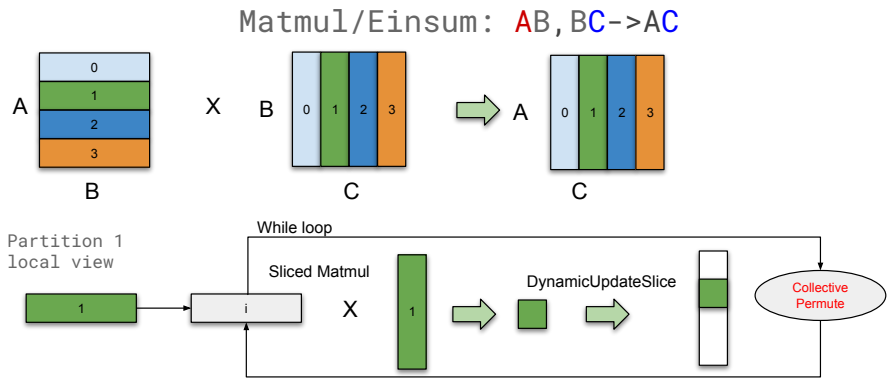

Fig. 5. An Einsum/Matmul partitioned on different non-contracting dimensions of operands over K devices. We use collective-permute in a loop to compute one slice at a time. There is no full-sized tensor during the entire process.

this context, we can virtualize the concept of partitions by overwriting the creators for collective ops and partition IDs.

Second, we introduce the concept of partition grouping. It allows us to divide the partitions into equally sized groups, and treat each group as a virtual device. Once we have such grouping, we can create a new partitioner context, where each logical partition is mapped to a group of devices, and the collective operator creator will rewrite the logical partition IDs to subgroups of original parttition IDs.

With this approach, we can perform recursive pattern matching on an Einsum op. For example, the partitioner detects if there is a matching pattern on how the batch dimensions are partitioned in the inputs and the output. If such a batch-partitioned pattern exists, it could create nested partitioner context by grouping the partitions across the batch dimensions, reduce the shape sizes, and recursively call the partitioner to handle other dimensions. Recursive partitioning allows us to handle nested partitioning cases without having to enumerate all possible scenarios.

2.3 Transformer layer case study

Given an input tensor x of shape (B, S, M), the transformer feed-forward layer computes

$$y = W_{out} \times \text{Relu}(W_{in} \times x)$$

where W_{in} and W_{out} are input and output projection weights with shape (M, H) and (H, M), respectively. With dimension $B = 512$, $S = 1024$, $M = 8192$, and $H = 65536$, tensors involved in the computation will have size $8.6GB$, $68.6GB$, and $2.14GB$ for bfloat16 input and hidden activations and the float32 W_{in}/W_{out} weights, respectively.

We can shard those above large tensors on 2D mesh of devices, as illustrated in the left panel of Figure 6. First, input and output weights are fully sharded but will be subgrouped all-gathered along M dimension on demand. The resulting weight tensor will be partially sharded along the H dimension. The input activation is sharded along the B and M dimensions. We apply the strategy in Figure 4 to compute the Einsum of $BSM, MH \rightarrow BSH$. The resulting hidden activation tensor are sharded along the B and H dimension. Since the output weight is partially sharded along H after all-gather, we can then apply a variant of strategy in Figure 3 to compute the Einsum of $BSH, HM \rightarrow BSM$ as the contracting dimension is sharded. We first compute the partial result of shape BSM that's sharded along the B dimension. Then instead of AllReduce, we use ReduceScatter to accumulate the result and extra sub-arrays so that the resulting output tensor becomes sharded along the B and M dimensions. Logically ReduceScatter is equivalent to AllReduce followed by DynamicSlice, but with only half of the cost of AllReduce. Note that During this feed-forward layer, the largest tensor BSH is fully 2D sharded. And there will be temporary partially tensors such as all-gathered weights and the output tensor BSM. Those temporary tensors have short live range within the layer and will be free after crossing the layer boundary. With a device mesh shape of 32×64 on a dragonfish pod and

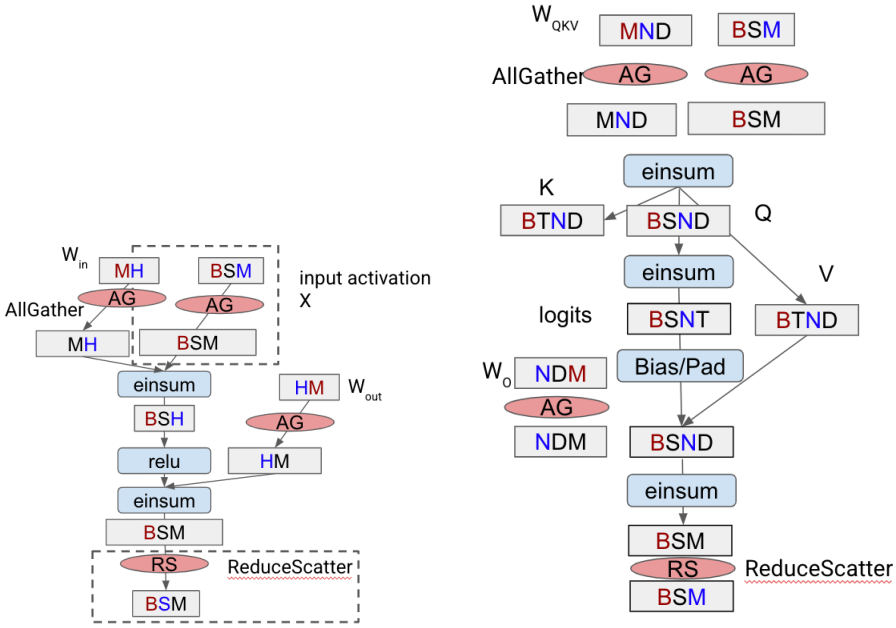

Fig. 6. 2D sharding of transformer feed-forward layer (left) and attention layer (right) over the $K_X \times K_Y$ device mesh. Red color indicates sharding arcoss K_X devices and blue color indicates sharding across K_Y devices.

the same dimension used in the Meena-128B model, each device will allocate 1.1MB for each projection weight, 4.4MB for the BSM activation, 33.5MB for the hidden activation BSH, 34MB for each partially sharded weight, and 269MB for the partially sharded BSM tensor. Only the memory usage for the project

weights and activation tensors at the layer boundary will grow with the number of layers in the model, other activation tensors and partially sharded weights within the layer will be free once cross the layer boundary.

The transformer attention layer computes:

$$y = \text{Attention}(W_Q \times x, W_K \times x, W_V \times x) \times W_O$$
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The Einsum operation also makes up the majority of computation, as shown in the right panel of Figure 6. Similar to the feed-forward layer, the input activation $[B, S, M]$ and all the weights are first all-gathered along the model dimension. Then the Einsum $BSM, MND \rightarrow BSND$ that projects input tensor X to the multi-headed key, value and query outputs a 2D sharded tensor along the B and multi-heads N dimensions. Einsums in the Attention(q, k, v) function like $BTND, BSND \rightarrow BSNT$ and $BSNT, BTND \rightarrow BSND$ are 2D sharded and have no sharding along the contracting dimension. No extra communication operation is required there. The computation of the output project is also similar to that in the feed-forward layer, a ReduceScatter is needed to accumulate and shard the final output tensor. In summary, on the $K_X \times K_Y$ device mesh, we recommend shard the weights and activations of transformer using the following rules:

- Split the batch dimension (B) across K_X devices.
- Split the feed-forward hidden dimension (H) and attention multi-heads dimension (N) across K_Y devices.
- Split the model dimension (M) across K_Y devices for activations and across K_X devices for the weights.

2.4 Meena 128B model performance optimizations

Here we estimate the communication and computation cost of the 64-layer Meena 128B model with dimensions $B = 512$, $S = 1024$, $M = 8192$, and $H = 65536$. The total number of parameter in this model is $137,702,416,384$. Additional communication needed over the $K_X \times K_Y$ mesh includes:

- 2x All-Gather for each projection weight across K_X in bfloat16 (one from the forward pass and the other from the backward pass.).
- 1x Reduce-scatter of all gradients in float32 across K_X, which is the backprop op of All-Gather.
- Up to 6x All-Gather of $[B, S, M]$ tensor per layer across K_Y in bfloat16 (forward/remat forward/backward inputs to self-attention and feedforward layers).

- 4x Reduce-scatter of $[B, S, M]$ tensor per layer across K_Y (forward/backward activations in both self-attention and feedforward layers) of type bfloat16.

Assuming 85 GB/s bandwidth, $K_X = 32$, $K_Y = 64$, the total communication time for the Meena 128B model per ship per step on a dragonfish pod is:

$$(\text{sizeof(fp32)} + 2 * \text{sizeof(bp16)}) \times \frac{\text{\# params}}{K_X \times \text{Bandwidth}} = 8 \times 138e9/32/85e9 = 0.4s$$

for 1x Reduce-scatter and 2x All-Gather across K_X devices and

$$(4 + 6) \times LBSM \times \frac{\text{sizeof(bp16)}}{K_Y \times \text{Bandwidth}} = 64 * 10 * 512 * 1024 * 8192 * 2/64/85e9 = 1.01s$$

for 4x Reduce-scatter and 6x All-Gather across K_Y devices.

The total MXU computation time per step including forward pass, remat forward, and backward pass on a dragonfish pod:

$$2 \times 3 \times BS \times \frac{\text{\# params}}{\text{FLOPS}} = 2 * 3 * 512 * 1024 * 138e9/126e15 = 3.44s$$

assuming 2 flops per multiplication/addition and peak 126 PFLOPS from a dragonfish pod. With everything optimized to 100% as assumed above, we should get to $\frac{3.44}{3.44+0.4+1.01} = 70.9\%$ utilization. However, due to imperfect data layout for Matmul, MXU computation utilization is actually around 85%. Not all AllReduce/DynamicSlice patterns got fused into Reduce-Scatter, failure to fuse would increase the communication time 2x. Assuming 2/3 communication bandwidth utilization, a more realistic estimation for the utilization is $\frac{3.44}{3.44/0.85+0.4*1.5+1.01*1.5} = 55.8\%$

Here are a list of existing and potential optimization techniques to improve utilization.

- Using approximate Relu (tf.nn.relu(approximate=True) or Silu (tf.nn.silu) instead of Relu.

- Combine QKV weights. The original implementation the input X is projected through weights W_Q, W_K and W_V separately to get Q, K and V. One optimization we can do to improve MXU efficiency is to combine the weights first and do the Einsum once, then split the results to get Q, K and V.

- Layout optimization to reduce communication time. Patterns of AllReduce/DynamicSlice Reshape activation model dimension M to 2 dimensions, then partition on the first. This gives XLA layout assignment more freedom to incur Reduce-Scatter fusion.

- Controlling device order for multi-dimensional sharding. To best utilize the ICI, each partitioned dimension is better to be aligned in TPU topology rows or columns. This is particularly important for slice smaller than a full pod since there will be no wrap-around ICI links. We need to arrange the device ids of the (K_X, K_Y) mesh carefully so that the communication neighbors can avoid long links.

- 2D AG/RS. TODO: more details
- go/collective-matmul. TODO: more details.

3 BENCHMARKS

3.1 Training

The systems performance of GShard-based transformers scales linearly from 64B to 1T. The communication bottleneck started to dominate when scaling further to 4T as compute/communication ratio is lower due to small batch size. Larger batch size is possible by scaling out the model to more TPU cores, or by enabling gradient accumulation. For the purpose of apple-to-apple comparison to other models in Table 1, we did not include the above optimizations in the 4T model.

# of params	# of layers	model dim	hidden dim	SeqLen	Batch size	MXU utilization	k-tokens/sec
64B	32	8192	65536	1024	512	52.2 %	151.9
128B	64	8192	65536	1024	512	55.3%	77.8
1T	128	16384	131072	1024	128	50.0%	9.2
4T	128	32768	262144	1024	32	28.8%	1.4

Table 1. Dense language models benchmarks on a Dragonfish pod. The model parameters are sharded across 2048 cores. We measured the TPU matrix unit utilization and throughput for dense models with various configurations.

Step times of GShard models roughly doubles when the number of layers doubles. Moreover, they are almost linear to the batch size as well, as shown in figure 7.

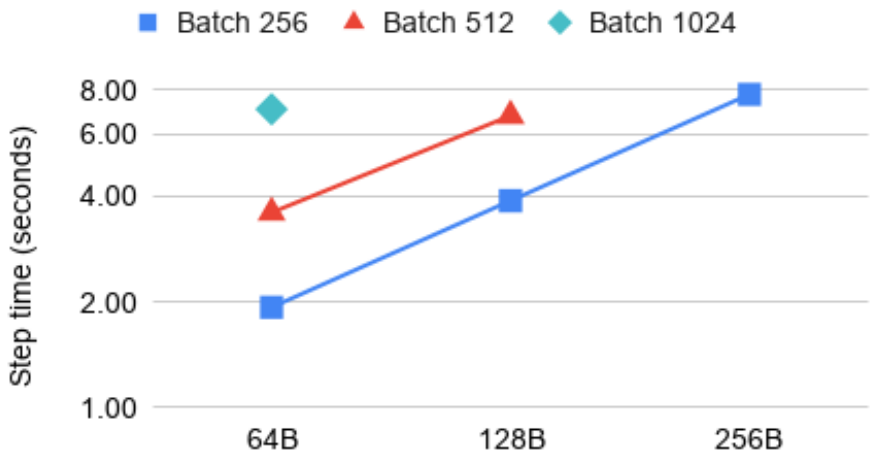

Fig. 7. Step times of GShard models are linear functions of both the model size and the batch size.

3.2 Serving

We are pioneering giant model serving infrastructure at Google.

Inference with Meena64B.

Inference with an autoregressive Transformer language model consists of 3 parts.

1) Input to the model is provided as a "prefix" of fixed size (1024 tokens). First part runs forward pass over the prefix in one pass and saves decoder self-attention key and value tensors. This part is very similar to forward pass during training and has the same performance characteristics.

2) Second part samples continuation of the prefix. It contains a loop where at each step decoder stack runs forward for 16 additional tokens. This loop repeats 64 times and generates 16 independent samples. Here again, decoder self-attention key and value tensors are saved at each iteration.

3) Third part takes all generated samples, appends a fixed "rescoring suffix" to each sequence and runs forward pass over this suffix (again in one pass). The log-perplexity of the suffix determines "quality score" of each sample. The highest-scoring sample is eventually shown to the user.

"Flat sampling"

In order to simplify the underlying implementation, all three parts (prefix, decoder loop, and suffix) operate over a common "buffer" which contains the prefix, 16 samples, and rescoring suffix for each sample in a single long sequence (2048 tokens = 1024 prefix + 16 samples * 64 decoder steps)

At each forward pass, the decoder self-attention mask is adjusted so that each token has attention to the prefix and to previously generated tokens from the same sample, but not to the tokens from alternative samples. (Decoder self-attention mask is applied to attention logits and is set to 0 for "visible" positions and -1e6 for "invisible" positions)

Memory constraints

The chief memory constraint for inference comes from the need to preserve attention keys and values across decoder loop iterations. As such we have "key/value state" tensors of size [2048,128,128] or 64MB each (total buffer length, number of heads, head dimension) in each layer. Total memory requirement is 32 * 2 * 64MB = 4GB for each element of the input batch.

Latency constraints

Decoder loop has to run forward the decoder stack (32 layers) multiple times. In order to make each iteration as fast as possible, dense layers and attention layers need to be sharded. This introduces additional all-reduce operations (one for attention and one for dense layer) which contribute to overall latency. The softmax

Model Size	Serving slice	dtype	Latency
64B	4x2	bp16	3.6s
64B	4x8	bp16	1.2s

Table 2. Prefix length=1024, decoded length=64. Flat beam search decoding.

computation (which happens at each iteration of decoder loop) also needs to be sharded. Because on each iteration the decoder stack runs for only 16 tokens (significantly less compared to training mode) it makes sense to shard the softmax over the vocabulary dimension. The decoder embedding layer is sharded the same way.

Latency

QPS and 100+ freebie 4x2s. https://screenshot.googleplex.com/6EC8itGf6E4cDJC

4 APPLICATIONS AND FURTHER WORK

We aim at achieving human-like multitasking with one giant model (go/meena2-paper-plan), this is the joint goal among the TWM/Meena/T5/OGM teams. We are working together on the downstream fine-tuned/few-shot tasks (go/meena-multitask), trying not only to beat GPT3 results, but to reach human-level performance as much as possible. This Giant Language Model can also be used as a base pre-train model and fine-tuned (using a fine-tune API recipe we are developing) by any researcher to conduct their research.

4.1 Application: Open Domain QA

We evaluate the model using open domain question answer (QA) tasks. Open domain QA tasks are among the most difficult generative NLP tasks in general in that for a given question, there is no additional context to be given from which the answer can be deduced. Without the explicit searching process for the related contexts and evidence, it requires the model to implicitly encode the knowledge into its parameters from general purpose pretraining corpus. Table 3 shows our current results of 1-shot learning on three public open domain qa benchmarks: TriviaQA, Natural Questions (NQ), and Web Questions (WQ), respectively. The performance is measured by the exact matching metric which requires that the generated answer must be exactly the same as the true answer on the dev set of each dataset.

4.2 Google Applications

We (Brain Applied Research team) are working closely with production teams on fine-tuning giant LM to their domain tasks. Currently we are in contact with Search (DeepRank, WebAnswer, etc. go/gems-for-search), Ads (Automated Targeting/Creative/Bidding), Translate, QueryBert, Cloud (CloudContactCenter) et al for potential product applications. We have a strong track record of working with these teams on past

Model Size	Step	Train Loss	Eval Loss	TriviaQA	NQ	WQ
64B	784k	1.98	n/a	44.3	n/a	n/a
64B	1.14M	1.91	2.31	48.7	15.7	19.1
128B	685k	1.91	2.36	48.0	n/a	n/a
128B	915K	1.88	2.3	50.7	n/a	n/a
128B	1M	1.85	2.22	52.3	17.3	18.3
128B	1.1675M	1.795	-	51.9	n/a	n/a
256B	262k	2.02	2.42	43.3	13.6	14.4

Table 3. Meena2 Model Quality

production launches. This Giant Language Model can also be used as a base pre-train model and fine-tuned (using a fine-tune API recipe we are developing) by any product team to improve their product task.