EFFICIENTLY SCALING TRANSFORMER INFERENCE

Reiner Pope [1] **Sholto Douglas** [1] **Aakanksha Chowdhery** [1] **Jacob Devlin** [1] **James Bradbury** [1]
Anselm Levskaya [1] **Jonathan Heek** [1] **Kefan Xiao** [1] **Shivani Agrawal** [1] **Jeff Dean** [1]

ABSTRACT

We study the problem of efficient generative inference for Transformer models, in one of its most challenging settings: large deep models, with tight latency targets and long sequence lengths. Better understanding of the engineering tradeoffs for inference for large Transformer-based models is important as use cases of these models are growing rapidly throughout application areas. We develop a simple analytical model for inference efficiency to select the best multi-dimensional partitioning techniques optimized for TPU v4 slices based on the application requirements. We combine these with a suite of low-level optimizations to achieve a new Pareto frontier on the latency and model FLOPS utilization (MFU) tradeoffs on 500B+ parameter models that outperforms the FasterTransformer suite of benchmarks. We further show that with appropriate partitioning, the lower memory requirements of multiquery attention (i.e. multiple query heads share single key/value head) enables scaling up to 32× larger context lengths. Finally, we achieve a low-batch-size latency of 29ms per token during generation (using int8 weight quantization) and a 76% MFU during large-batch-size processing of input tokens, while supporting a long 2048-token context length on the PaLM 540B parameter model.

1 INTRODUCTION

Scaling Transformer-based models to 100B+ (Brown et al., 2020; Kaplan et al., 2020; Rae et al., 2021; Hoffmann et al., 2022) and later 500B+ parameters (Chowdhery et al., 2022; Smith et al., 2022) has led to state of the art results on natural language processing benchmarks. The practical utility of these large language models (LLMs) in a variety of applications makes them compelling for widespread use. While the sequence parallelism of the Transformer architecture enables highly parallel training, efficient deployment of these models is challenging in practice because generative inference proceeds one token at a time and the computation for each token sequentially depends on the previously generated tokens. Thus, models that support efficient training at scales of thousands of chips require careful attention to parallel layout and memory optimizations to unlock the scalability needed for efficient, low-latency inference. This paper focuses on a simple set of engineering principles that enable serving large-scale Transformer-based models efficiently in a variety of challenging production settings.

We consider the requirements of downstream applications for LLMs. Some applications, including interactive workloads like chatbots, involve tight latency constraints (Thoppilan et al., 2022). Others, including offline inference for scoring or distillation, emphasize high throughput and low cost per token at any latency.

We discuss briefly what makes generative inference of LLMs challenging. First, large models have a large memory footprint both due to the trained model parameters as well as the transient state needed during decoding. The model parameters generally do not fit in the memory of a single accelerator chip. The attention key and value tensors of each layer, which we refer to as the *KV cache*, must also be stored in memory for the duration of decoding. Second, tight latency targets become especially challenging for generative inference given the much lower parallelizability of Transformer generation relative to training. The large memory footprint gives rise to a large amount of memory traffic to load the parameters and KV cache from high-bandwidth memory (HBM) into the compute cores for each step, and hence a large total memory bandwidth required to meet a given latency target. Finally, inference cost from the attention mechanism scales quadratically with input sequence length (Sukhbaatar et al., 2019; Choromanski et al., 2020; Dao et al., 2022).

We found two keys to optimize LLMs for inference efficiency. First, we found it useful to build a powerful and abstract *partitioning framework* to enable reaching the limits of model parallel scaling given the limited parallelizability of Transformer inference. Within this framework, we analytically solve for the best partitioning strategy for a given model size with specific application requirements. This enables the user to intuitively understand the tradeoffs and

[1]Google. Correspondence to: Sholto Douglas <sholto@google.com>, Aakanksha Chowdhery <chowdhery@google.com>.

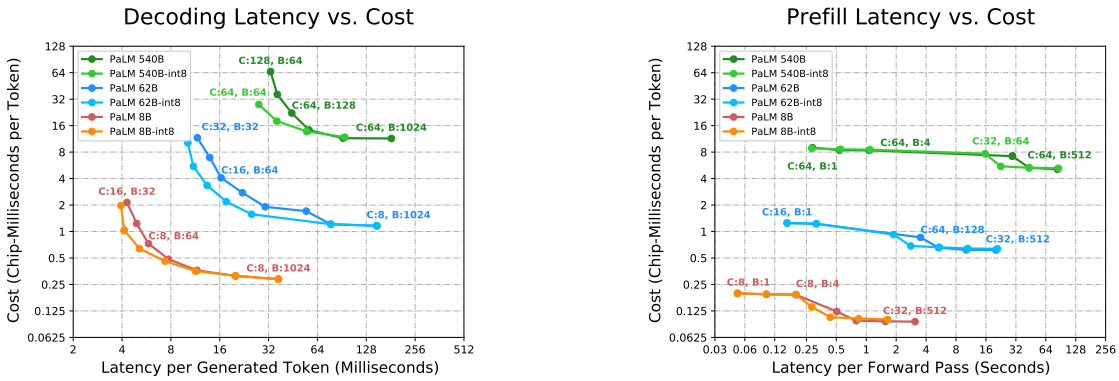

Figure 1: Cost vs. latency for PaLM models. We use a context length of 2048. Points in each line represent the Pareto frontier of efficiency versus latency. Chip count is C, batch size is B. Left: latency per token for generating 64 tokens, assuming the context has already been processed. Right: time to process 2048 input tokens; excludes the time to generate any output tokens. Tables 2 and 3 show details on a few specific scenarios from the Pareto frontier where the applications have low-latency or high-throughput requirements.

select the best multi-axis tensor partitioning strategy, batch size and chip configuration for their application, in contrast to a black-box exhaustive search over partitioning strategies (Zheng et al., 2022; Xu et al., 2021). To fully realize the performance in practice, we use additional fine-grained control over cross-chip collective operations and low-level scheduling optimizations. Second, we apply memory optimizations and take full advantage of PaLM's multiquery attention to reduce unnecessary tensor overheads and maximize the batch size that fits on a given number of chips, enabling higher throughput.

The primary goal of this paper is to provide a set of engineering principles for how best to partition a model in order to scale Transformer inference. In other words, how is the performance of different partitioning strategies affected by changes in model size, sequence length, and number of hardware chips? How does the optimal partitioning strategy change when trading off between latency and throughput? What is the intuitive and mathematical reasoning behind these effects? As we show in later sections, the right tradeoffs and strategies change as model size, sequence length, and application requirements for latency and throughput targets change, so having a framework that enables easy expression of different strategies and choices is important.

In Section 2, we describe the specific metrics and tradeoffs we use to compare different partitioning strategies. In Section 3.1, we provide an overview of partitioning principles for large language models. In the remainder of Section 3, we describe a number of specific partitioning strategies, with an empirical validation on the PaLM family of large language models in Section 4.

For a state-of-the-art 540B parameter dense model running on 64 TPU v4 chips, we achieve a low-batch-size latency of 29ms per token during generation (with int8 weight quantization) and a 76% MFU during large-batch-size processing of input tokens while supporting a large context length of 2048 tokens. Figure 1(left) shows our performance for generating text using the PaLM models. For an interactive application such as a chatbot running on PaLM 540B with int8 weights, our implementation on 64 TPU v4 chips can process 64 tokens of text from a user, consult a cached conversation history of 1920 tokens, and generate a 64-token response in a total of 1.9 seconds. For an offline throughput-oriented application, our implementation can process 1984 tokens of input and generate 64 tokens of output, for huge numbers of examples, with an overall FLOPS efficiency of 73%. Table 2 shows more details on a few specific scenarios.

2 Inference Cost Tradeoffs

Scaling up model sizes can unlock new capabilities and applications but has fundamental tradeoffs in terms of inference cost. We measure the inference cost in terms of the following metrics: latency, throughput, and model FLOPS utilization. The *latency* is the total time for an inference and can be broken down into the time to process the input tokens present at the start of the inference (which we call "prefill") and the time to autoregressively generate output tokens (which we term "decode"). The decode latency can also be measured "per step", i.e. divided by the number of tokens in each sequence. The *throughput* of prefill or decode is the number of tokens processed or generated per second. The *model FLOPS utilization (MFU)* is the ratio of the observed throughput to the theoretical maximum throughput if the benchmarked hardware setup were operating at peak FLOPS with no memory or communication overhead.

Larger models do not fit on a single accelerator chip and

need to be partitioned across many accelerator chips to fit in memory. This also enables us to divide the memory and compute costs described below over all the chips, but comes at the cost of introducing chip-to-chip communication.

Memory costs. We store tensors such as weights and the KV cache in on-device high-bandwidth memory (HBM). While there are other tensors that pass through the HBM, their memory footprint is much smaller, so we focus on just these two largest groups of tensors. These tensors need to be transferred from HBM to the compute cores of the chip once per forward pass (prefill or decode step) of the model. This takes a certain amount of time, which we call the "memory time." At small batch sizes and sequence lengths, the time to load weights dominates. At larger batch sizes and sequence lengths (e.g. 2048+ tokens with batch size 512+), the time to load the KV cache dominates.

Compute costs. An N-parameter decoder-only model requires $2N$ matmul FLOPs in the forward pass per token seen because each matmul performs one multiplication and one addition per pair of input token and parameter values in the forward pass (Kaplan et al., 2020). If all chips were running at peak FLOPS, these matmuls would take a certain amount of time, which we call the "compute time." The matmuls in the attention mechanism typically add a much smaller number of FLOPs per token for large models and can often be excluded. Even though the computational cost of attention is relatively small, it can still account for a significant fraction of memory capacity and bandwidth costs, since (unlike the weights) the KV cache is unique for each sequence in the batch.

2.1 Expected tradeoffs and challenges

Both the weight loading part of the memory time and the non-attention compute time are proportional to the model size and inversely proportional to the number of chips. However, for a given partitioning layout, the time needed for chip-to-chip communication decreases less quickly (or not at all) with the number of chips used, so it becomes an increasingly important bottleneck as the chip count grows. We consider some scenarios where these tradeoffs become especially challenging.

If an application requires the *lowest possible latency*, we need to apply more chips and partition the model in as many ways as we profitably can. Lower latency can often be achieved with smaller batch sizes, but smaller batch sizes also result in worse MFU, resulting in a higher total cost (in terms of chip-seconds or dollars) per token.

If an application requires generating text with *long attention contexts*, it substantially increases the inference time. For a 500B+ model with multihead attention, the attention KV cache grows large: for batch size 512 and context length 2048, the KV cache totals 3TB, which is 3 times the size of the model's parameters. The on-chip memory needs to load this KV cache from off-chip memory once for every token generated during which the computational core of the chip is essentially idle.

If an applications requires *offline inference* and latency is not a concern, the primary goal is to maximize per-chip throughput (i.e., minimize total cost per token). It is most efficient to increase the batch size because larger batches typically result in better MFU, but certain partitioning strategies that are not efficient for small batch sizes become efficient as the batch size grows larger.

2.2 Inference Setup

We briefly introduce the inference setup and notation. We consider a Transformer model with n_{params} parameters laid out for inference on n_{chips} chips. The model has model (or embed) dimension d_{model} (or E), feedforward intermediate dimension d_{ff} (or F), and n_{heads} (or H) heads.

Each example in a batch of B sequences has L_{input} tokens of input text, and generates L_{gen} tokens of output text. Since the input tokens are all present at the start of the inference, we can run the model over all $B \times L_{\text{input}}$ many tokens in parallel, in a single forwards pass over all the tokens. We call this step *prefill*. The output tokens are generated autoregressively, with a sequential loop of L_{gen} steps. Each step consists of a single forwards pass through the model, after which we sample one new token for each of the B examples in the batch. This loop is known as *generation* or *decode*.

Since prefill can run in parallel over L_{input}, but decode must run sequentially over L_{gen}, the two phases have different performance characteristics and we analyze them separately.

3 Partitioning for Inference Efficiency

We must partition large models over many chips in order to fit weight and activation tensors in memory and fit compute and memory time within latency requirements. Model partitioning introduces communication between chips, and different partitioning strategies for a given model involve different patterns and amounts of communication. In this section, we detail several high-level strategies for partitioning a large Transformer language model for cost-effective and latency-effective inference.

3.1 Partitioning notation and communication collectives

We describe the partitioning layouts in this section based on a TPU v4 system with 3D torus topology $X \times Y \times Z$.

Following (Xu et al., 2021), we use subscripts to specify the tensor dimension that is partitioned. For example, notation BLE_{xyz} means that the last dimension E of a tensor of logical shape BLE is split into $X \times Y \times Z$ partitions, where x, y and z refer to the physical TPU v4 axes, and the per-chip tensor is of shape $[B, L, E/(X \times Y \times Z)]$. Here B, E and F refers to batch, model embed and MLP feedforward dimension. We use L to refer to the sequence length and explicitly specify prefill or generation phase.

If a tensor is replicated over an axis x, that axis is omitted from the notation. We also use a suffix "partialsum-x" to indicate that a given tensor has been contracted (summed) locally on each chip (over some axis not represented in the shape), but still needs to be summed across the chips in the TPU x axis (creating a tensor replicated over x) before the result is meaningful.

We use several communication collectives originating from MPI (Clarke et al., 1994). The *all-reduce(x)* primitive sums a partialsum tensor such as BLE_{yz}(partialsum-x) across sets of chips in the x axis of the torus and broadcasts the sum back to all the involved chips, returning output of shape BLE_{yz}. For the reasons outlined in Rajbhandari et al. (2020), we typically split all-reduce into two phases: a reduction phase and a broadcast phase. The reduction phase is called *reduce-scatter(x)*, and it sums BLE_{yz}(partialsum-x) tensors across sets of chips in the x axis but produces an output that's sharded rather than replicated over the chips in that axis, in a layout such as B_xLE_{yz} or BLE_{xyz}. The broadcast phase is called *all-gather(x)*, and it broadcasts and concatenates the tensor BLE_{xyz} to all chips in the x axis, producing an output X times larger than its input, replicated over the x axis: BTE_{yz}. The *all-to-all* collective shifts sharding from one tensor dimension to another, e.g. $BLH_xQ \rightarrow B_xLHQ$ by using direct communication between every (source, destination) pair. Figure A.1 illustrates these primitives.

3.2 Partitioning the feedforward layer

3.2.1 Feedforward layer, 1D weight-stationary layout

Overview. When a model doesn't fit on a single chip, the simplest partitioning strategy is *1D weight-stationary*, where each $E \times F$ weight matrix is partitioned (or sharded) among n_{chips} along the E *or* F axis. Each weight shard is multiplied by the appropriate activation shard on each chip, and the results are aggregated between the chips with an all-gather and/or reduce-scatter. Additionally, when computing two consecutive matrix multiplications (as in a Transformer MLP block), there is a "trick" (Shoeybi et al., 2019) to avoid any cross-chip communication between the matmuls: if the first matmul is partitioned by the output axis, the resulting activation shard on each chip will be the exact one needed to compute the second matmul partitioned by the input axis. As we parallelize the computation across more chips, the memory latency and compute latency does decrease, often near-linearly. However, the communication latency remains roughly constant independent of the number of chips used, since the entire activation matrix is aggregated across chips for every pair of matrix multiplications. As the number of chips grows larger, communication becomes a bottleneck.

Details. We consider as a baseline the layout where the weights and activations of the feedforward layer are partitioned over n_{chips} along the d_{ff} dimension, as in Megatron (Shoeybi et al., 2019). Figure 2(a) shows the partitioning layout for this case. On the TPU v4's 3D torus topology the partition layout for weights is EF_{xyz} and $F_{xyz}E$, i.e. they are partitioned in to $X \times Y \times Z = n_{\text{chips}}$ partitions with X, Y, and Z partitions across physical TPU axes. The weights are kept stationary in each chip, and the activations are transferred between chips to match the weight layout, requiring one all-gather and one reduce-scatter.

In this 1D weight-stationary partitioning strategy, each chip gets inputs and outputs of shape BLE in the reduce-scatter and all-gather respectively. We derive the the communication cost of these operations in Appendix A.1. The resulting communication time is

$$T_{\text{comm}} = \frac{2BLE}{\text{network bandwidth}}.$$

3.2.2 Feedforward layer, 2D weight-stationary layout

Overview. For a larger number of chips, a more economical strategy involves partitioning each $E \times F$ weight matrix along both the E *and* F axes, such that each shard is roughly square. For example, if $E = 1024$, $F = 4096$, and $n_{\text{chips}} = 64$, then we would shard 4-ways among E and 16-ways among F, so that each of the 64 chips stores a 256-by-256 chunk of the weight matrix, and activations are transferred between chips. This is called *2D weight-stationary*. The total compute cost is the same as 1D weight-stationary, but communication is much more efficient: when multiplying an activation matrix through a set of consecutive weight matrices, we can *alternate* which of the two axes we perform the activation aggregation on between each multiplication. With the correct partitioning, each chip will always have the necessary activation shard to multiply with its weight shard, without ever having a fully replicated copy of the activation tensor. Since each axis is partitioned on $O(\sqrt{n_{\text{chips}}})$, the communication time scales as $O(\frac{1}{\sqrt{n_{\text{chips}}}})$ rather than remaining constant. This means that even if the 2D layout is communication-limited at a certain chip count and batch size, we can continue to reduce latency by adding more chips, because communication time continues to reduce.

However, while the 1D weight-stationary "trick" requires

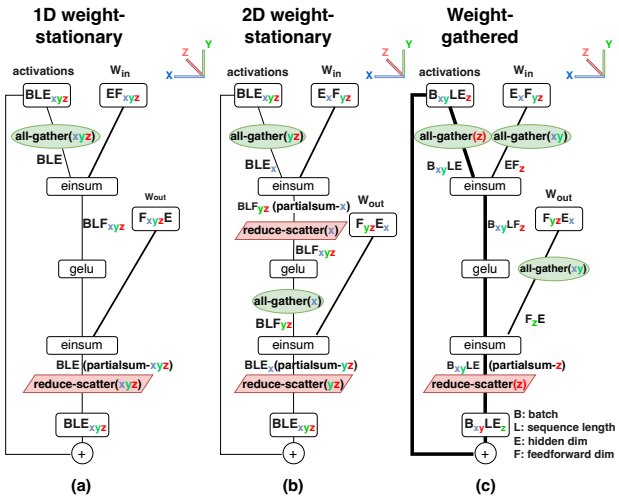

Figure 2: Partitioning layouts for feedforward layer.

us to only aggregate over the d_{model} dimension, 2D weight-stationary requires alternating aggregation over the d_{model} and d_{ff} dimensions. Therefore, 2D weight-stationary becomes more communication-efficient when $\sqrt{n_{\text{chips}}} > \frac{d_{\text{ff}}}{d_{\text{model}}}$. Since typically $d_{\text{ff}} = 4d_{\text{model}}$, this occurs when $n_{\text{chips}} > 16$.

Details. Figure 2(b) shows the partitioning layout. Whereas the 1D weight-stationary layout runs its all-gather and reduce-scatter with unsharded shape BLE per chip, this 2D weight-stationary layout partitions d_{model} so that the communication volume for d_{ff} partitioning is reduced from BLE to $\frac{BLE}{X}$. This comes at the cost of introducing a second pair of reduce-scatter and all-gather operations, whose cost must be balanced with the existing communication.

The partitioning layout for weights is $E_x F_{yz}$, i.e. they are partitioned along the d_{model} dimension into X partitions and along the d_{ff} dimension into $Y \times Z$ partitions, where $X \times Y \times Z = n_{\text{chips}}$. The partitioning layout for the input activations is the same as the previous section. Note that we again keep the partitioned weights stationary on their chips, but because of their 2D layout, the activation communication includes two all-gathers and reduce-scatters.

We derive the optimal values of X, Y and Z to minimize total communication time in Appendix A.2.1. Assuming $d_{\text{ff}} = 4 \times d_{\text{model}}$, we achieve the minimum communication time with $X = 0.5 \times \sqrt{n_{\text{chips}}}$ and $YZ = 2 \times \sqrt{n_{\text{chips}}}$. The resulting total communication time is:

$$T_{\text{comm}} = \frac{8BLE}{\sqrt{n_{\text{chips}}} \times \text{network bandwidth}}.$$

3.2.3 *Feedforward layer, weight-gathered layout*

Overview. In the previously described weight-stationary strategies, each chip stores one shard of each weight matrix in memory, and that chip is responsible for multiplying it's "stationary" weight shard with each corresponding activation shard. The output of each per-chip matrix multiplication must then be aggregated between chips to be used as input to the subsequent operations.

However, as the batch size (and sequence length) grows larger, the size of the output activations may become significantly larger than the size of the weights. When this happens, it can become more economical to keep the activations stationary on each chip, and instead transfer the weights between chips. For very large batch sizes, it is best to keep the activations fully stationary between sequential matrix multiplications, requiring that we fully transfer the weights between all chips. We call this approach *XYZ-weight-gathered*. For moderate batch sizes, it is beneficial to use a "hybrid" approach where both weights and activations are partially transferred along different axes. We refer to these approaches as *X-weight-gathered* and *XY-weight-gathered*.

Details. Figure 2(c) shows the XY-weight-gathered layout. A key aspect of the specific layout we choose is that weights start in the same $E_x F_{yz}$ layout as in 2D weight-stationary, so that we can use the same weight layout for weight-gathered (duing prefill) and weight-stationary (during decoding). Just before the einsums, the weight tensors are all-gathered over the X and Y axes, with communication volume $\frac{EF}{Z}$. This is additional communication relative to weight-stationary layout, but in return we reduce the communication on activations: one reduce-scatter/all-gather pair for activations is skipped, and the communication volume on the other pair drops from $\frac{BLE}{X}$ to $\frac{BLE}{XY}$.

By changing the relative sizes of the X, Y, and Z axes, we can trade off weight communication against activation communication, and thereby minimize the total communication volume. But we choose to share the weights between weight-stationary and weight-gathered layouts, which means we are required to match the choices of X, Y and Z made for the weight-stationary layout. What we do instead is pick between a few variants of the weight-gathered layout. The variant shown in Figure 2(c) uses all-gather(xy) for the weights and $B_{xy}LE_z$ partitioning of batch for the activations. Our other variants use all-gather(x) or all-gather(xyz) for weights, and correspondingly use $B_x LE_{yz}$ or $B_{xyz}LE$ partitioning of the activations. Figure A.2 shows the three weight-gathered layouts.

Figure 3 shows how the communication-optimal configuration switches between these layouts as batch size grows – while the 2D weight-stationary strategy minimizes communication at low tokens per batch, different weight-gathered layouts are optimal at larger number of tokens per batch. This highlights the importance of choosing different infer-

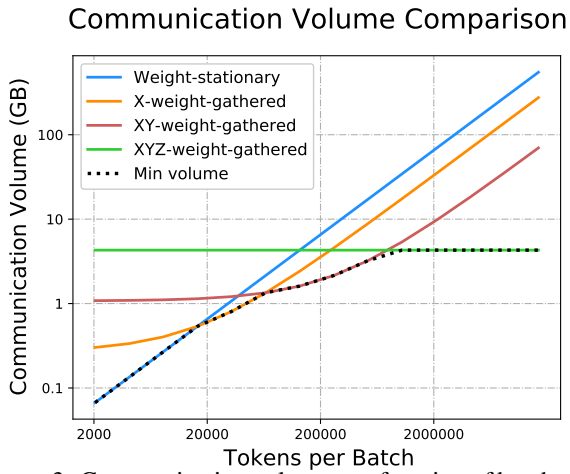

Figure 3: Communication volume as a function of batch size, for a feedforward layer. As batch size (in tokens) grows, it is better to switch to a layout that all-gathers the weights over increasingly more chips to minimize the communication volume. Communication volumes estimated for $X = Y = Z = 4$, $d_{\text{model}} = 16384$, and $d_{\text{ff}} = 65536$.

ence configurations depending on application goals.

We now show the asymptotic scaling of weight-gathered layouts. Let N be the number of chips that weights are all-gathered over: $N = X$ in X-weight-gathered, $N = XY$ in XY-weight-gathered, $N = XYZ$ in XYZ-weight-gathered. Total communication is minimized by the choice $N = \sqrt{\frac{BLn_{\text{chips}}}{F}}$ which we derive in Appendix A.2.2. The total communication time is

$$T_{\text{comm}} = 4E\frac{\sqrt{BLF}}{\sqrt{n_{\text{chips}}} \times \text{network bandwidth}}$$

Note that BL corresponds to the total batch size in tokens. The communication time for the weight-stationary layout is linear in BL, while the communication time for the weight-gathered layout is linear in $\sqrt{BL}$. Therefore, the weight-gathered layout becomes cheaper when the batch size and prefill sequence length are sufficiently large.

3.3 Partitioning the attention layer

Multihead attention can be parallelized in essentially the same ways as a feedforward layer, with n_{heads} replacing d_{ff}. But inference with multihead attention incurs significant memory capacity and bandwidth costs to store and load the KV cache, and these costs can dominate the rest of the inference at large batches or long context lengths.

An alternative approach, called *multiquery attention* (Shazeer, 2019; Chowdhery et al., 2022), still emits n_{heads} for the query tensor, but only a single head for the key and value tensors. This key and value head is shared across the n_{heads} query heads. This reduces the size of the KV cache tensors by a factor of n_{heads} and hence the

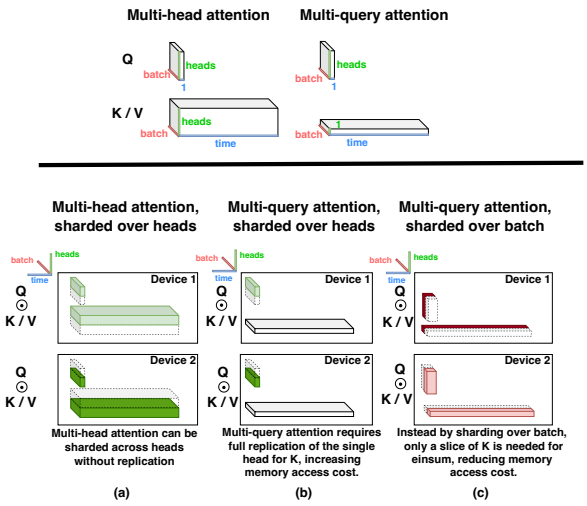

Figure 4: Multiquery attention has lower memory cost to load the KV cache when sharded over batch.

memory time spent loading them. But it also removes an axis otherwise used for parallelism, so the KV cache and related computations need to be partitioned differently.

Partitioning strategy. The key design consideration is to minimize the memory time of repeatedly loading the KV cache that dominates the inference cost. The partitioning layout of projection matrices that have a n_{heads} dimension (W_Q and W_O in multiquery attention, and those two plus W_K and W_V in multihead attention) should match the layout used in the feedforward layer.

Figure 4(a) shows a typical partitioning layout for multihead attention, matching the 2D weight stationary feedforward layout. Here the Q, K, and V activations are partitioned over the n_{heads} dimension into n_{chips} partitions when n_{heads} is a multiple of n_{chips}. For n_{chips} greater than n_{heads}, the attention heads are partially replicated. The most similar partitioning layout for multiquery attention (shown in Figure 4(b)) treats the KV cache the same as in multihead attention. Even though the key and value tensors are shared across all heads, they must be replicated on each chip and the memory cost savings of multiquery attention are lost.

We instead propose a partitioning strategy for the multiquery attention where the Q, K, and V matrices are partitioned over the batch B dimension into n_{chips} partitions. Figure 4(c) shows that this reduces the memory cost of loading the KV cache per chip by a factor of n_{chips}, thereby reducing the memory time by the same factor. The proposed partitioning strategy incurs additional communication cost of resharding the input activation tensors using an all-to-all collective as shown in Figure 5(b) in comparison to the multiquery attention sharding strategy shown in Figure 5(a) where the Q, K,

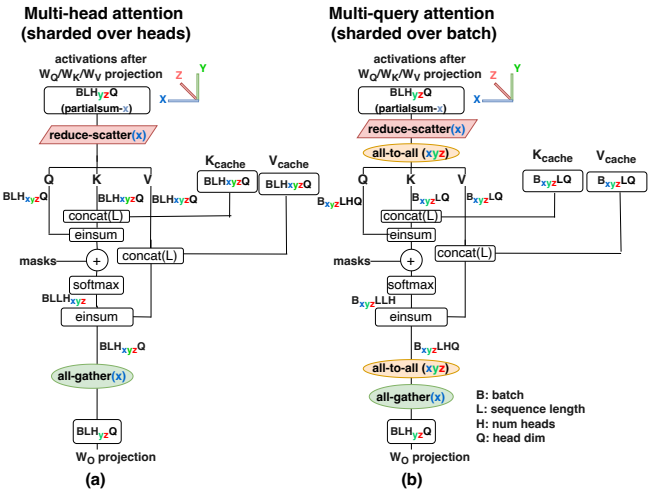

Figure 5: Comparison of partitioning layouts for attention layer: multihead attention sharded over heads versus multiquery attention sharded over batch.

and V matrices are partitioned over the heads dimension.

During autoregressive generation, there is only one token per example of Q, K, and V tensors, whereas the KV cache has many (perhaps 2048) tokens. Since the KV cache is orders of magnitude larger than the Q, K, and V tensors, it is very profitable to spend the all-to-all communication time on the small tensors to save the memory time on the large tensors.

During prefill, it is typically not profitable to shard attention over batch. The Q tensor has many (perhaps 2048) tokens, all of which are queried against the same K and V tensors. The memory load of the K and V tensors is amortized over all tokens in the Q tensor, and so this memory load is typically not a bottleneck during prefill. Therefore for prefill we use the sharded-over-heads layout.

With the proposed partitioning layout, multiquery attention enables using larger batch sizes and sequence lengths, thereby increasing throughput in addition to the latency reduction from reduced memory time. As shown in Section 4.2, the savings are an order of magnitude compared to multihead attention.

3.4 Parallel attention/feedforward layers

We discuss the inference latency gains from the "parallel" formulation of each Transformer block (Wang and Komatsuzaki, 2021) as used in PaLM (Chowdhery et al., 2022) instead of the standard "serialized" formulation, where the feedforward layer and attention layer are computed in parallel from the layernormed input and summed to get the output.

The benefits from the parallel formulation are as follows. First, there is only one layernorm per layer instead of two, which reduces latency at small batch sizes. Second, the input matrices of the feedforward layer can be fused with the query projection matrix W_Q of the attention layer, the key/value projection matrices W_K and W_V can be fused in the attention layer, and the output matrix of the feedforward layer can be fused with the output projection matrix W_O of the attention layer. This fusion results in higher FLOPS utilization because larger matrix-multiplications run more efficiently on accelerators. More importantly, it also eliminates one of the two all-reduce operations in each Transformer layer needed for $d_{\text{ff}}/n_{\text{heads}}$ parallelism, cutting communication time over this axis in half.

3.5 Low-level optimizations

We use the Looped CollectiveEinsum technique from (Wang et al., 2023) to run communication concurrently with computation. This allows us to partially or fully hide the communication time of most of the reduce-scatter and all-gather operations in Figures 2 and 5. For all reduce-scatter operations in Figures 2 and 5, we had a choice of whether to reduce-scatter into a batch or sequence dimension (B or L) or into the hidden dimension (E or F). We chose the latter, because it exposes more effective opportunities for Looped CollectiveEinsum, whereas Korthikanti et al. (2022) chose the former, to avoid communication in layernorm.

The CollectiveEinsum loops are the overwhelming majority of the inference latency, so we invested considerable effort to maximize their performance. First, we used the underlying "async CollectivePermute" APIs of Wang et al. (2023) to develop a suite of variants of the CollectiveEinsum concept, to optimize for different scenarios: latency versus throughput, different numbers of torus axes, fusing with different input/output collectives. Second, we explicitly match up communication collectives with the matrix multiplies that they should be fused with, to maximize the potential for overlap. Through such optimizations, we achieved about 1.4 times better performance than the simpler compiler-partitioned-and-scheduled implementation that we started with. Some of the weight-gathered layouts would exhaust memory without these optimizations.

We also included the following low-level optimizations: better in-memory layout of tensors to minimize padding and copying during matrix multiplies, faster top-k/top-p implementations for decode sampling, faster log-base-2 implementations of Softmax and Swish, and support for incremental processing of sequences during prefill (FasterTransformer).

3.6 Quantization

We use the AQT library (Lew et al., 2022) to reduce the memory cost of 16-bit weights by converting them to int8

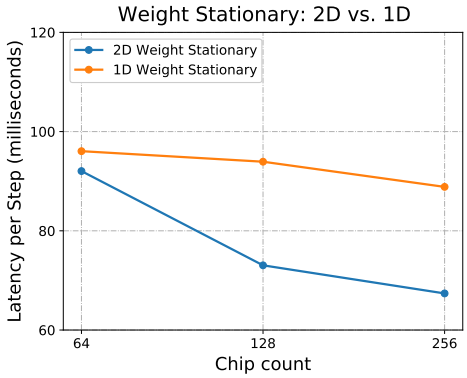

Figure 6: Latency per token doing text generation of PaLM 540B for 2D and 1D weight stationary layouts on 64 chips.

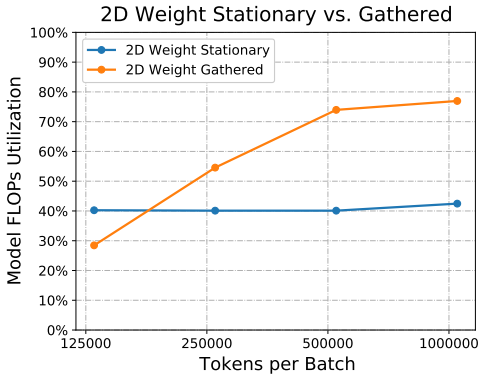

Figure 7: Model FLOPS utilization running prefill on PaLM 540B on 64 chips, with sequence length 2048. We report batch size measured in tokens: number of sequences multiplied by sequence length. As batch size (in tokens) grows, note that it is better to switch from the 2D weight stationary to the weight gathered approach to improve MFU.

without noticeable quality loss. This enables memory time savings from weight loading, which is especially helpful in the low batch size regime, and it reduces communication volume in weight-gathered layouts. We have not implemented *activation* quantization (Abdolrashidi et al., 2021), but we are hopeful that it could reduce compute time in large-batch configurations and reduce communication volume of activations in weight-stationary layouts.

4 Case Study for PaLM Models

Methodology We now conduct an empirical study of our techniques on the PaLM family of models (Chowdhery et al., 2022), which we select since the model architecture incorporates the techniques of multiquery attention and parallel attention and feedforward layers.

Our inference framework is based on JAX (Bradbury et al., 2018) and XLA (XLA, 2019), and our original high-level implementation was based on T5X (t5x, 2021). We use up to 256 TPU v4 chips (Google, 2022) for our benchmarks. Each TPU v4 chip can run bfloat16 matrix arithmetic at 275 TFLOPS, has 32 GiB of High Bandwidth Memory (HBM) at 1200 GB/s of bandwidth, and has 270 GB/s of interconnect bandwidth in a 3D torus topology (TPUv4).

For the PaLM 540B model we padded the number of attention heads up from 48 to 64 in order to partition more effectively on 64+ chips. This adds 18B parameters to the model, which comes at a 3% MFU cost, which was more than recovered by being able to partition more effectively.

4.1 Partitioning feedforward layer

We evaluate the relative performance of our feedforward layer partitioning strategies. First we evaluate performance of decoding. We use batch size 512 to balance latency and MFU. Figure 6 shows the performance of 1D and 2D weight-stationary layouts as we increase the chip count. Both layouts start to become communication-limited, but the 2D layout performs better because of its asymptotically better scaling with chip count.

Next we consider the prefill phase. We consider batch sizes from 2048 tokens (1 example, 2048 tokens) to 1 million tokens (512 examples, 2048 tokens per example). Figure 7 shows that the optimal partitioning layout switches from the 2D weight-stationary layouts to the weight-gathered layouts as the batch size increases. The weight-gathered layouts are inefficient at low batch sizes, but eventually they become the most efficient at high batch sizes, achieving 76% MFU when the communication overhead is almost negligible. Such large batch sizes would fail from memory exhaustion without multiquery attention, as shown in Section 4.2. This highlights the importance of flexibility in configuring the inference system with different choices depending on the application setting and goals.

These results give us our basic strategy for selecting partitioning layout: during the prefill phase, we select from weight-stationary and weight-gathered layouts based on the current number of tokens in the batch. During the generate phase, we select the 2D weight-stationary layout because the batch size in tokens is always small.

4.2 Partitioning Attention layer

We now evaluate the partitioning layout for multiquery attention proposed in Section 3.3. We consider PaLM with multiquery attention in both the baseline layout that partitions by attention heads and the optimized layout that partitions by batch. We also create a modified variant of PaLM 540B which uses multihead attention instead of multiquery. To keep parameter count in the attention layer constant, we shrink d_{head} from 256 in the multiquery variant to 128 in the

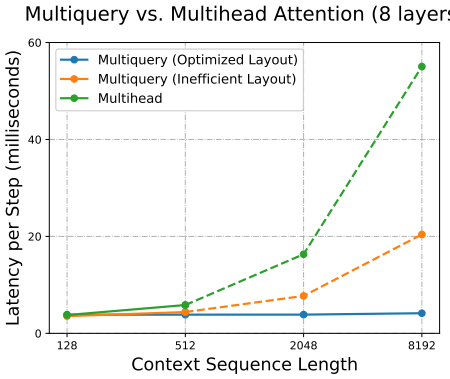

Figure 8: Latency per generated token vs. sequence length, for an 8-layer version of PaLM 540B on 64 chips with batch size 256. The dotted line represents that on the full 118-layer model and context lengths longer than 512, the KV cache will not fit in memory when using multihead attention or the baseline multiquery partitioning.

Model variant	d_{head}	Max context length	
		batch=128	batch=512
Multihead	128	1320	330
Baseline multiquery	256	660	165
Optimized multiquery	256	43,000	10,700

Table 1: Maximum context length supported for different attention variants of PaLM 540B on 64 chips. We reserve 30% of the total memory for KV cache. Optimized multiquery attention enables up to 32x larger context lengths.

multihead variant.

At large batch sizes and context lengths, the KV cache can become very large, putting us at the risk of running out of memory. Table 1 shows that the optimized multiquery layout can fit up to 32–64 times longer context lengths than the multihead and baseline multiquery variant.

During prefill, multiquery and multihead attention incur similar inference latencies because we compute many attention queries in parallel and the attention computation becomes compute-limited on the attention matrix multiplies. During generation, Figure 8 shows that the optimized multiquery layout improves speed. The speed improvement is small when the context length is short because almost all of the time is spent on the feedforward layer. As the context length grows longer, the time to load the KV cache in the attention layer becomes a much larger portion of overall inference time. Multiquery attention scales up to sequence lengths of 8192–32,768 tokens (batch sizes 512 and 128 respectively) with attention taking only 8–31% of total runtime.

4.3 Parallel attention/feedforward layers

We consider a variant of PaLM 540B with the parallel formulation of Transformer block replaced by serial attention/feedforward layers. During generation, we use 2D weight-stationary layout, 64 chips, and batch size 512. The serial formulation incurs 14% higher inference latency per step than the parallel version because of the increased communication time for activations. In the prefill phase, this difference shrinks because the weight-gathered layouts incur less activation communication.

4.4 End-to-end results on PaLM

We find the Pareto frontier between efficiency and latency as we scale the model size for the PaLM family of models: 8B, 62B and 540B, with weights in either bfloat16 or int8. We use a context length 2048 and sweep over the batch size and chip count.

To meaningfully compare throughput across multiple model sizes with different chip count and batch sizes, we report the *cost* of an inference in terms of *chip-seconds per token* calculated as

$$\text{cost (chip-seconds per token)} = \frac{n_{\text{chips}} \times \text{time}}{BL}.$$

This is directly proportional to operational cost and inversely proportional to MFU.

Figure 1(left) shows the relationship between model size, latency, and cost in the generate phase, at the Pareto frontier of optimal batch size, chip count, and partitioning strategy. The lowest cost is achieved at batch sizes larger than about 512, where the cost is proportional to the number of parameters. As we decrease the batch size, we improve the latency but incur higher cost per token. The minimum latency for generation is 3 times lower than the batch-512 latency.

We observe that int8 weight quantization achieves the minimum latency in Figure 1 (left): for example, we achieve 28.5ms/token with int8 weights at batch size 64 on PaLM 540B, while we achieve 36.9ms/token with bfloat16 weights. At low latency targets the cost is improved just over a factor of 2, because low-batch-size cost is dominated by weight loading time. At large batch size, cost is more neutral between int8 and bfloat16, because large-batch cost is dominated by the compute time and the matmuls still use bfloat16 arithmetic. We believe that quantization of *activations* to int8 could enable a further cost improvement.

Figure 1 (right) shows the relationship between model size, latency, and cost in the prefill phase. The tradeoff between batch size and latency is less severe in the prefill phase than the generate phase and even batch size 1 runs with fairly low cost. Further, the cost of batch-512 prefill is 2 times lower than batch-512 generate because of the increased MFU of the weight-gathered layouts we use during prefill. More details on the relationship between model size and MFU are presented in Figure C.1 and Section C in the Appendix.

	Low-latency		High-throughput	
	Prefill	Decode	Prefill	Decode
Chips	64	64	64	64
Batch	1	64	512	512
FFN	WS 2D	WS 2D	WG XYZ	WS 2D
Attention sharding	Head	Batch	Batch	Batch
Weights format	int8	int8	bfloat16	bfloat16
MFU	43%	14%	76%	33%
Latency	0.29s	1.82s	85.2s	6.0s

Table 2: Example configurations for PaLM 540B, in the same setting as Figure 1. Prefill latency is for processing 2048 tokens; decode latency is for generating 64 tokens. Feedforward network (FFN) layouts are Weight Stationary 2D (WS 2D, Section 3.2.2) and Weight Gathered XYZ (WG XYZ, Section 3.2.3). Attention layouts are from Section 3.3.

	Low-latency		High-throughput	
	Prefill	Decode	Prefill	Decode
Chips	16	16	32	8
Batch	1	32	512	512
FFN	WS 2D	WS 2D	WG XYZ	WS 2D
Attention sharding	Head	Batch	Batch	Batch
Weights format	int8	int8	bfloat16	bfloat16
MFU	36%	8%	73%	37%
Latency	0.16s	0.73s	20.2s	5.1s

Table 3: Example configurations for PaLM 62B, in the same setting as Figure 1. Prefill latency is for processing 2048 tokens; decode latency is for generating 64 tokens. Feedforward network (FFN) layouts are Weight Stationary 2D (WS 2D, Section 3.2.2) and Weight Gathered XYZ (WG XYZ, Section 3.2.3). Attention layouts are from Section 3.3.

Tables 2 and 3 show some key configurations from the Pareto frontiers of Figure 1, on PaLM 540B and PaLM 62B. In the low-latency scenarios we combine batch-1 prefill with batch 32-to-64 decode: batch size 1 achieves best latency in the prefill phase, but for the generate phase we can increase the batch size up to 64 with negligible latency impact, and doing so is dramatically better for generate MFU. This mixture of batch sizes is possible in practice either by generating multiple samples from the same input text, or by pipelining a batch-1 prefill server into a batch-64 decoding server.

In the high-throughput scenarios of Tables 2 and 3, we use larger batch sizes and we switch partitioned layouts between prefill and decode. We use bfloat16 weights for high-throughput scenario, because the weight-loading time is unimportant at large batch sizes, and because our software is missing some optimizations for large-batch int8 mode.

Comparing 62B (Table 3) vs. 540B models (Table 2), we find that we use more chips for the 540B model, but similar batch sizes and the same partitioned layouts. High-throughput MFUs are similar between the model sizes. The low-batch-size latencies grow *sublinearly* with model size at the Pareto frontier: even though larger models load proportionally more weights from memory, we can partition them across more chips before becoming communication-limited. We estimate an approximately square-root relationship between model size and latency based on Figure 1 (left).

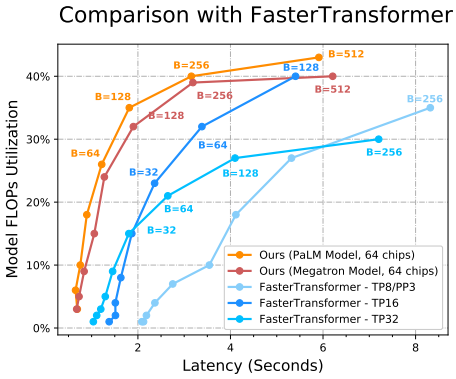

Figure 9: Model FLOPS utilization (MFU) versus total latency for running a 60-input-token, 20-output-token inference, at a range of batch sizes.

5 FasterTransformer Benchmarks

We now compare with the FasterTransformer benchmarks (FasterTransformer) across a wide range of batch sizes and configurations of prefill and generate. There are multiple differences between our benchmark setup and the FasterTransformer benchmark. In particular, we use different types of chips and chip counts – FasterTransformer uses 16–32 NVIDIA A100s with 80GiB HBM, while we use 64 Google TPU v4 chips with 32GiB HBM. Therefore, we report throughput numbers in terms of MFU, which normalizes for both chip count and chip FLOPS.

Figure 9 shows the performance of our implementation relative to three FasterTransformer configurations. We benchmark the Megatron 530B model (Smith et al., 2022) and the similarly-sized PaLM 540B model, which has architectural optimizations including multiquery attention and parallel attention/feedforward layers (full list of differences in Table D.1). Our implementation of PaLM 540B achieves the best absolute latency, and our implementation also offers the best MFU for the Megatron model for all but one latency target. Our PaLM implementation outperforms our Megatron implementation by up to 10% MFU in this benchmark primarily because of the parallel attention/ffn layers. Compared to Section 4.2, the advantage of parallel layers is partially offset by Megatron's larger d_{model} and d_{ff} sizes. The advantage of multiquery attention is not noticeable in this benchmark because the attention context length is too short.

FasterTransformer reports results with 8-, 16-, and 32-way tensor parallelism. Their 32-way tensor parallelism achieves a maximum of 33% MFU across all reported benchmarks, compared to 46% MFU in their 16-way tensor parallel configuration. This likely indicates a communication bottleneck of scaling tensor parallelism beyond this point. In contrast, our implementation is able to scale up to 64-way tensor parallelism while still achieving 44% MFU, suggesting superior scalability of our 2D weight-stationary partitioning strategy on TPU v4's larger high-speed interconnect domains.

We provide results on all the configurations used in the FasterTransformer baseline in Appendix D. We also note that our benchmarks throughout the paper attempt to include more challenging inference scenarios, such as context lengths in the range 1024–4096, and report the inference latency for the generate phase and the prefill phase separately (since they have different characteristics).

6 Related Work

Parallelism approaches. Prior works propose several approaches for efficient partitioning to train large models efficiently, for e.g., NeMo Megatron (Korthikanti et al., 2022), GSPMD (Xu et al., 2021) and Alpa (Zheng et al., 2022). FasterTransformer establishes a benchmark suite for multi-GPU multi-node inference for a range of different model sizes, including Megatron–Turing NLG 530B. The key inference speedups come from combining tensor parallelism and pipeline parallelism in conjuction with memory optimizations. DeepSpeed Inference (Aminabadi et al., 2022) further enables ZeRO offload to use CPU and NVMe memory in addition to the GPU memory. For larger batch sizes, EffectiveTransformer packs consecutive sequences together to minimize padding. Zheng et al. (2022) generalizes the search through parallelism strategies via integer-linear programming. In comparison, this paper derives the partitioning strategies based on intuitive empirically-backed analytical tradeoffs to meet the application requirements that scale well with model size, context length and chip count.

ML inference efficiency. Several approaches (Gupta and Agrawal, 2020) to improve the inference efficiency of Transformer models focus on model architecture improvements, for example efficient attention layers (Roy et al., 2020; Choromanski et al., 2020; Kitaev et al., 2020; Sukhbaatar et al., 2019; Child et al., 2019), distillation (Sanh et al., 2019; Sun et al., 2020), and model compression techniques, such as pruning (Li et al., 2020b; Brix et al., 2020; Zhou et al., 2021; Li et al., 2020a; Wang et al., 2020), or quantization (Dettmers et al., 2022; Abdolrashidi et al., 2021; Zafrir et al., 2019; Zhang et al., 2018). This paper reuses the prior work on model quantization to add to the inference speedups, and the techniques we describe could also be coupled with other model compression methods.

7 Conclusions

Large Transformer-based models are unlocking new capabilities and applications in several domains, but we need significant advances to democratize their access as we scale up the model size. This paper investigates the scaling properties of Transformer inference workloads and proposes practical partitioning approaches to meet challenging application requirements such as tight latency targets (on the order of seconds for 500B+ parameter models). We show that the best latencies are achieved by going far beyond the traditional paradigm of single-server inference, and scaling inference up to 64+ chips. Longer context lengths incur higher memory costs, but multiquery attention with appropriate partitioning reduces this cost and makes long-context inference practical. The proposed partitioning strategies generalize to many topologies, including single- and multi-node NVLink networks in GPU systems.

Although we achieve our goal of pushing the boundaries of scale for inference workloads, we observe that FLOP count and communication volume can fundamentally limit inference performance of dense Transformer models. Sparsity techniques, such as task-based mixture of expert architectures (Fedus et al., 2022; Kudugunta et al., 2021; Lepikhin et al., 2020; Shazeer et al., 2017), and adaptive computation techniques that allocate different amounts of compute per input and generation timestep (Jaszczur et al., 2021; Schuster et al., 2022), promise to reduce FLOPs per token of Transformer models. We are hopeful that such techniques that reduce FLOPs per token, as well as techniques that compress chip-to-chip communication, will enable further gains in both cost and latency.

8 Acknowledgments

Our work builds on top of the work of many, many teams at Google. We'd especially like to recognize the PaLM team, T5X team, the Pathways infrastructure team, the JAX team, the Flaxformer team, the XLA team, and the AQT team. We are grateful to Blake Hechtman, Marcello Maggioni, Zongwei Zhou, and Shibo Wang for XLA support and performance optimizations. We would like to thank our colleagues for valuable inputs and discussion on the project – Jacob Austin, Yuanzhong Xu, Lukasz Lew, Sharan Narang, Adam Roberts, Noah Fiedel, and Mike Gunter. We also thank Hyeontaek Lim, James Laudon, George Necula, Martin Abadi and Chandu Thekkath for their review and feedback in improving the presentation of this work, and Erica Moreira for the support of compute resources.

REFERENCES

T5x, 2021. URL https://github.com/google-research/t5x.

AmirAli Abdolrashidi, Lisa Wang, Shivani Agrawal, Jonathan Malmaud, Oleg Rybakov, Chas Leichner, and Lukasz Lew. Pareto-optimal quantized resnet is mostly 4-bit. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3091–3099, 2021.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.

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. JAX: Composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Christopher Brix, Parnia Bahar, and Hermann Ney. Successfully applying the stabilized lottery ticket hypothesis to the transformer architecture. *arXiv preprint arXiv:2005.03454*, 2020.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Charles Sutton Hyung Won Chung, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022. URL https://arxiv.org/abs/2204.02311.

Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*, 2022.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

EffectiveTransformer. Effective transformer. https://github.com/bytedance/effective_transformer). [Online; accessed October-2022].

FasterTransformer. Fastertransformer: Gpt guide. https://github.com/NVIDIA/FasterTransformer/blob/main/docs/gpt_guide.md). [Online; accessed October-2022].

William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning. *arXiv preprint arXiv:2209.01667*, 2022.

Google. Cloud TPU. https://cloud.google.com/tpu, 2022. [Online; accessed October-2022].

Manish Gupta and Puneet Agrawal. Compression of deep learning models for text: A survey. *arXiv preprint arXiv:2008.05221*, 2020.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Sebastian Jaszczur, Aakanksha Chowdhery, Afroz Mohiuddin, Lukasz Kaiser, Wojciech Gajewski, Henryk Michalewski, and Jonni Kanerva. Sparse is enough in scaling transformers. *Advances in Neural Information Processing Systems*, 34:9895–9907, 2021.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distillation: Task-level mixture-of-experts for efficient inference. *arXiv preprint arXiv:2110.03742*, 2021.

Dmitry Lepikhin, 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. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=qrwe7XHTmYb.

Lukasz Lew, Vlad Feinberg, Shivani Agrawal, Jihwan Lee, Jonathan Malmaud, Lisa Wang, Pouya Dormiani, and Reiner Pope. Aqt: Accurate quantized training), 2022. URL http://github.com/google/aqt.

Bingbing Li, Zhenglun Kong, Tianyun Zhang, Ji Li, Zhengang Li, Hang Liu, and Caiwen Ding. Efficient transformer-based large scale language representations using hardware-friendly block structured pruning. *arXiv preprint arXiv:2009.08065*, 2020a.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning*, pages 5958–5968. PMLR, 2020b.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, H. Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training Gopher. *CoRR*, abs/2112.11446, 2021.

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

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *arXiv preprint arXiv:2003.05997*, 2020.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q Tran, Yi Tay, and Donald Metzler.

Confident adaptive language modeling. *arXiv preprint arXiv:2207.07061*, 2022.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR (Poster)*. OpenReview.net, 2017. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2017.html#ShazeerMMDLHD17.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

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

Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. *arXiv preprint arXiv:1905.07799*, 2019.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, 2020.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022. URL https://arxiv.org/pdf/2201.08239.

TPUv4. Google Cloud unveils world's largest publicly available ML hub with Cloud TPU v4, 90% carbon-free energy. https://cloud.google.com/blog/products/compute/google-unveils-worlds-largest-publicly-available-ml-cluster.

Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. Hat: Hardware-aware transformers for efficient natural language processing. *arXiv preprint arXiv:2005.14187*, 2020.

Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *To appear in the Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

XLA. XLA: Optimizing compiler for TensorFlow. https://www.tensorflow.org/xla, 2019. [Online; accessed September-2019].

Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=K9bw7vqp_s.

A Partitioning Strategies: Deriving Communication Costs

A.1 Cost of all-gather/reduce-scatter

Figure A.1 shows typical collective operations we use in partitioning strategies and their communication patterns across three devices. For an all-gather over K partitions, where each chip produces an output of size D, the communication pattern requires chunks of size $\frac{D}{K}$ to be transferred over $(K-1)$ interconnect links in the process of getting copied to $(K-1)$ chips. The resulting communication time for the all-gather is

$$T_{\text{comm(all-gather)}} = \frac{D}{\text{(network bandwidth)}} \frac{K-1}{K}.$$

This is a general cost model that holds true for most real-world network topologies (Chan et al., 2007), not just the TPU's torus topology.

The communication time for a reduce-scatter $T_{\text{comm(reduce-scatter)}}$ is the same, except that D is the size of the (larger) input buffer rather than the (smaller) output buffer. Thus, the total communication time for an all-reduce is $T_{\text{comm(all-reduce)}} = 2 \times T_{\text{comm(all-gather)}}$.

In most formulas, we will disregard the $(K-1)/K$ term, approximating it as 1 under the assumption $K \gg 1$, in order to simplify the algebra. This yields a simple approximation: reduce-scatter time is proportional to the size of the per-chip input, and all-gather time is proportional to the size of the per-chip output.

A.2 Details for communication time calculations

A.2.1 Feedforward layer, 2D weight-stationary layout

Figure 2(b) shows the partitioning layout. The partitioning layout for weights is $E_x F_{yz}$, i.e. they are partitioned along the d_{model} dimension into X partitions and along the d_{ff} dimension into $Y \times Z$ partitions, where $X \times Y \times Z = n_{\text{chips}}$. We now show how to size the X, Y and Z axes of the torus to minimize total communication time in 2D weight-stationary layout. The communication time is:

$$T_{\text{comm}} = \frac{2BL}{\text{network bandwidth}} \left(\frac{E}{X} + \frac{F}{YZ} \right)$$

We have a free choice of X, Y and Z subject to available TPU v4 slice shapes and $X \times Y \times Z = n_{\text{chips}}$. Assuming $d_{\text{ff}} = 4 \times d_{\text{model}}$, we achieve the minimum communication time with $X = 0.5 \times \sqrt{n_{\text{chips}}}$ and $YZ = 2 \times \sqrt{n_{\text{chips}}}$. The resulting total communication time is:

$$T_{\text{comm}} = \frac{8BLE}{\sqrt{n_{\text{chips}}} \times \text{network bandwidth}}.$$

A.2.2 Feedforward layer, weight-gathered layout

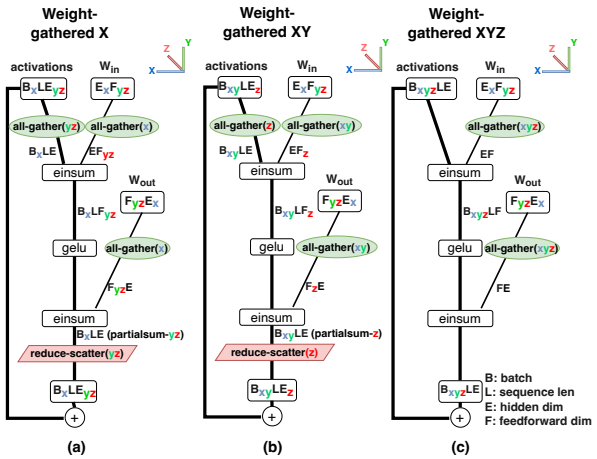

Figure A.2: Weight-gathered layouts for Feedforward layer.

Figure A.2 shows the different weight-gathered layouts, while Figure 2(c) shows one instance of XY weight-gathered layout. A key aspect of the specific layout we choose is that weights start in the same $E_x F_{yz}$ layout as in 2D weight-stationary, so that we can instantly switch between weight-gathered layout and weight-stationary layout. Just before the einsums, the weight tensors are all-gathered over the X and Y axes, with communication volume EF/Z.

By changing the relative sizes of the X, Y, and Z axes, we can trade off weight communication against activation communication, and thereby minimize the total communication volume. We now show the asymptotic scaling of weight-gathered layouts. Let N be the number of chips that weights are all-gathered over: $N = X$ in X-weight-gathered, $N = XY$ in XY-weight-gathered, and $N = XYZ$ in XYZ-weight-gathered.

Weight communication is:

$$T_{\text{comm(weights)}} = \frac{2EF \times N}{n_{\text{chips}} \times \text{network bandwidth}}.$$

Activation communication is:

$$T_{\text{comm(acts)}} = \frac{2BLE}{N \times \text{network bandwidth}}.$$

Total communication is minimized by the choice $N = \sqrt{BSn_{\text{chips}}/F}$, which yields total communication time

$$T_{\text{comm}} = 4E \frac{\sqrt{BLF}}{\sqrt{n_{\text{chips}}} \times \text{network bandwidth}}$$

Figure 3 shows how the communication-optimal configuration switches between these layouts as batch size grows. While the 2D weight-stationary strategy minimizes communication at low tokens per batch, different weight-gathered layouts are optimal at larger number of tokens per batch.

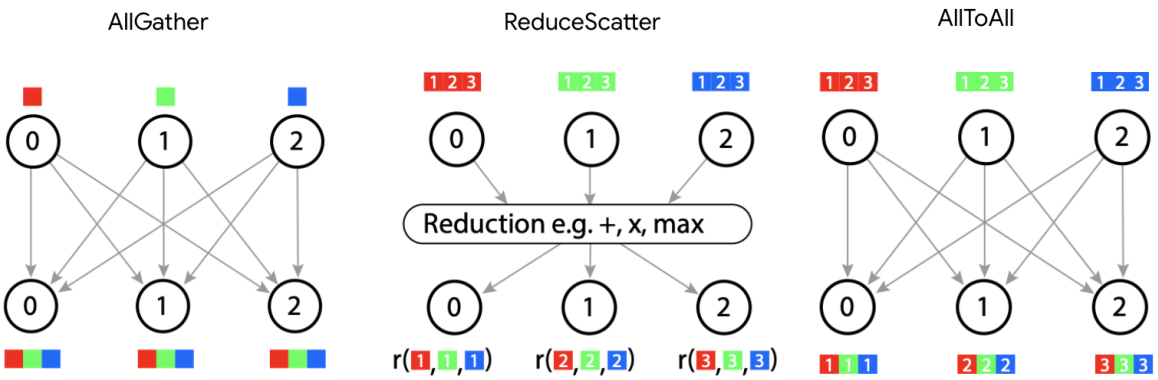

Figure A.1: Communication patterns of collective operations: all-gather, reduce-scatter, and all-to-all across three devices.

B Minimum prefill latency

We report here the minimum latency required for prefill. Figure B.1 shows the Pareto frontier of cost vs. latency as we sweep sequence length from 32 to 1024 at batch size 1.

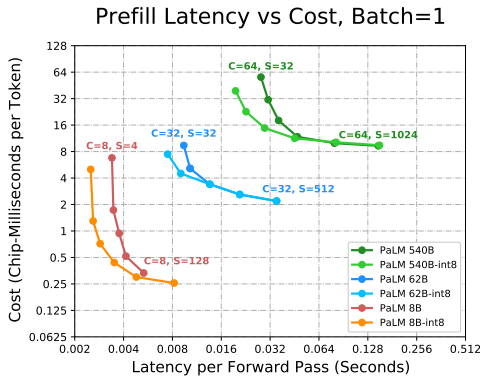

Figure B.1: Prefill cost vs. latency for PaLM models over a range of sequence lengths S. C indicates chip count.

C MFU vs latency tradeoff

We report here the relationship between model size, latency, and MFU. Figure C.1 shows the Pareto frontier of MFU vs. latency as we sweep the batch size and the number of chips same as Figure 1. The MFU for decode is typically much lower than for prefill. In the prefill phase, the "jumps" in MFU show the transition point from weight stationary 2D layout to XYZ weight gathered layout.

In most cases, the larger models achieve higher MFUs than the smaller models, because larger matrix multiplies are more efficient. However, at long-latency decodes, PaLM 62B achieves higher MFU than PaLM 540B, because the former uses 8-way model parallelism and the latter uses 64-way model parallelism. We may be able to further optimize PaLM 540B by reducing the model parallelism in the high-throughput (latency-tolerant) regime.

D Full comparison to FasterTransformer

In this section, we report the latency and MFU of our implementations of both the PaLM 540B model and the Megatron-Turing NLG 530B model run on 64 TPU v4 chips, in comparison to FasterTransformer baselines. We first note the model architecture differences in Table D.1.

Then, we report the the full set of comparisons for the three configurations in the FasterTransformer benchmarks: 20 input tokens and 8 output tokens in Table D.2, 60 input tokens and 20 output tokens in Table D.3, and 128 input tokens and 8 output tokens in Table D.4.

For each table we report the *Pareto frontier* of latency and MFU with **bold font** (frontier across all 500B-class results) and underline (frontier across MT-NLG specifically). This frontier is *not* a per-row comparison, but instead is defined globally across the table. It is defined as follows: a benchmark result $(\text{latency}, \text{MFU})$ is on the Pareto frontier if, for all other benchmark results $(\text{latency}_2, \text{MFU}_2)$, either $\text{latency} \leq \text{latency}_2$ or $\text{MFU} \geq \text{MFU}_2$ (or both) is true. Visually, this corresponds to being "up and to the left" in Figure 9.

We do not report batch sizes below 4 because our partitioning strategy partitions multiquery attention over batch and achieves no speedup for a batch size smaller than 4 (the minimum size of a TPU v4 torus axis).

	PaLM 540B	**Megatron 530B**
n_{params}	540B	530B
n_{layers}	118	105
d_{model}	18432	20480
d_{ff}	73728	81920
n_{heads}	48	128
d_{head}	256	160
Attention	Multiquery	Multihead
Parallel ffn/attn	Yes	No

Table D.1: Hyperparameters for PaLM and Megatron-Turing NLG inference.

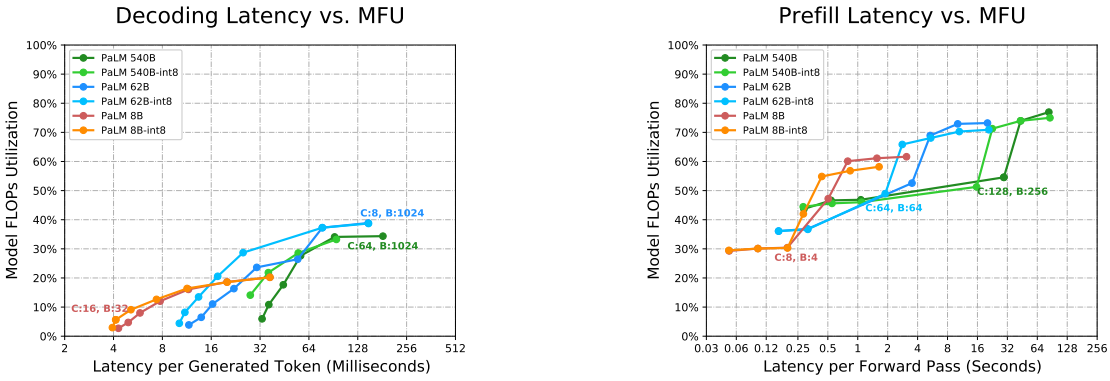

Figure C.1: MFU vs. latency for PaLM models. We use a context length of 2048. Points in each line represent the Pareto frontier of efficiency versus latency. Chip count is C, batch size is B. Left: latency per token for generating 64 tokens, assuming the context has already been processed. Right: time to process 2048 input tokens; excludes the time to generate any output tokens. The corresponding cost vs latency numbers are shown in Figure 1.

	FasterTransformer MT-NLG 530B total						Ours (530B/540B on 64 TPU v4 with 2D partitioning)							
	TP16		TP32		PP3/TP8		PaLM prefill		PaLM generate		PaLM total		MT-NLG total	
batch	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU
1	565	1%	431	1%	842	0%	-	-	-	-	-	-	-	-
2	598	2%	455	1%	860	1%	-	-	-	-	-	-	-	-
4	616	4%	493	2%	867	2%	34	14%	255	1%	**289**	**2%**	**289**	**2%**
8	660	7%	523	5%	929	3%	40	25%	226	2%	**265**	**5%**	304	4%
16	730	13%	575	8%	1049	6%	58	34%	234	3%	**292**	**9%**	339	8%
32	865	22%	672	14%	1283	10%	99	40%	235	7%	**334**	**16%**	420	13%
64	1191	32%	942	20%	1722	15%	186	42%	265	12%	**451**	**24%**	532	20%
128	1862	41%	1431	27%	2124	24%	356	44%	312	20%	**668**	**33%**	740	29%
256	**3341**	**46%**	2483	31%	3140	32%	668	47%	415	30%	**1083**	**41%**	1151	38%
512	-	-	-	-	-	-	1366	46%	671	37%	**2037**	**43%**	2151	40%
1024	-	-	-	-	-	-	2785	45%	1257	40%	4041	44%	4082	42%

Table D.2: Results for the 20-input-token, 8-output-token benchmark. All times in milliseconds. The **bold** and underline annotations are *not* per row, but instead show the Pareto frontier of time vs. MFU. See Section D for full explanation.

	FasterTransformer MT-NLG 530B total						Ours (530B/540B on 64 TPU v4 with 2D partitioning)							
	TP16		TP32		PP3/TP8		PaLM prefill		PaLM generate		PaLM total		MT-NLG total	
batch	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU
1	1379	1%	1037	1%	2085	1%	-	-	-	-	-	-	-	-
2	1515	2%	1110	2%	2122	1%	-	-	-	-	-	-	-	-
4	1512	4%	1198	3%	2184	2%	50	29%	640	1%	690	3%	678	3%
8	1631	8%	1295	5%	2367	4%	80	37%	574	2%	**653**	**6%**	728	5%
16	1868	15%	1454	9%	2753	7%	153	39%	602	3%	**755**	**10%**	838	9%
32	2361	23%	1804	15%	3543	10%	270	44%	626	6%	**896**	**18%**	1058	15%
64	3383	32%	2646	21%	4117	18%	501	47%	717	11%	**1218**	**26%**	1275	24%
128	5406	40%	4099	27%	5319	27%	985	48%	829	19%	**1814**	**35%**	1902	32%
256	OOM	-	7203	30%	8318	35%	2041	46%	1114	28%	**3155**	**40%**	3189	39%
512	-	-	-	-	-	-	4167	45%	1743	36%	**5910**	**43%**	6210	40%
1024	-	-	-	-	-	-	8349	45%	3260	39%	11608	43%	12390	40%

Table D.3: Results for the 60-input-token, 20-output-token benchmark. All times in milliseconds. The **bold** and underline annotations are *not* per row, but instead show the Pareto frontier of time vs. MFU. See Section D for full explanation.

	FasterTransformer MT-NLG 530B total						Ours (530B/540B on 64 TPU v4 with 2D partitioning)							
	TP16		TP32		PP3/TP8		PaLM prefill		PaLM generate		PaLM total		MT-NLG total	
batch	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU	time	MFU
1	585	5%	451	3%	866	2%	-	-	-	-	-	-		
2	667	9%	508	6%	932	4%	-	-	-	-	-	-		
4	765	15%	606	10%	1097	7%	81	39%	258	1%	343	10%	**338**	**10%**
8	990	23%	766	15%	1434	11%	149	42%	234	2%	403	17%	**384**	**16%**
16	1377	34%	1074	22%	2104	15%	287	44%	253	3%	586	23%	**540**	**23%**
32	2251	41%	1741	27%	2623	23%	536	47%	263	6%	**796**	**34%**	799	33%
64	4002	46%	3114	30%	3578	34%	1056	48%	317	10%	**1329**	**40%**	1372	39%
128	OOM	-	5784	32%	5512	45%	2202	46%	381	17%	**2343**	**46%**	2583	45%
256	OOM	-	11232	33%	**9614**	**51%**	4479	45%	431	29%	4710	45%	4911	45%
512	-	-	-	-	-	-	8913	45%	734	34%	9673	44%	9647	43%
1024	-	-	-	-	-	-	17766	45%	1370	37%	19723	43%	19136	43%

Table D.4: Results for the 128-input-token, 8-output-token benchmark. All times in milliseconds. The **bold** and underline annotations are *not* per row, but instead show the Pareto frontier of time vs. MFU. See Section D for full explanation.