AIRND.CENTER
artificial intelligence Innovation

إعداد الفريق العلمي:

بمركز أبحاث الذكاء الاصطناعي (آيرند)

إشراف المهندس: عبدالله بن إبراهيم الحجي

AIRND.CENTER

مركز آيرند - تعزيز أبحاث الذكاء الاصطناعي

اسم البحث:

توسيع النماذج العملاقة باستخدام الحوسبة المشروطة والتقسيم التلقائي :GShard

GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding

إعداد الفريق العلمي:

بمركز أبحاث الذكاء الاصطناعي (آيرند)

إشراف المهندس: عبدلله بن إبراهيم الحجي

تاريخ التقرير:12/16/2024

تاريخ البحث:6/30/2020

اختار مركز أبحاث الذكاء الاصطناعي (أيرند) هذا البحث لتقديم تلخيص عنه يبرز أهميته ويقربه للباحثين

يُقدم هذا البحث إطار عمل يُدعى **GShard** تم تصميمه لتوسيع نطاق النماذج الضخمة في التعلم العميق بطريقة فعّالة ومرنة. يعتمد GShard على الحوسبة المشروطة والتقسيم التلقائي للحوسبة عبر أعداد كبيرة من الأجهزة) مثل .(TPU تم استخدام GShard لتدريب نموذج ترجمة آلي متعدد اللغات يحتوي على 600 مليار معلمة باستخدام بنية Transformer مع طبقات "مزيج الخبراء" (Mixture-of-Experts). يُظهر البحث كيف يمكن تدريب مثل هذه النماذج العملاقة بكفاءة على 2048 وحدة TPU خلال 4 أيام فقط، مع تحقيق جودة ترجمة فائقة مقارنة بالأعمال السابقة.

النقاط الرئيسية في البحث:

التحديات في توسيع النماذج:

1. **التكاليف الحاسوبية :**النماذج الضخمة تتطلب موارد حاسوبية ضخمة وزمن تدريب طويل.
2. **تعقيد البرمجة :**يتطلب تقسيم النموذج إلى أجزاء قابلة للتنفيذ على أجهزة متعددة جهدًا برمجيًا كبيرًا.
3. **كفاءة التنفيذ :**يجب تقليل الفاقد في الأداء الناتج عن الاتصال بين الأجهزة والتبعيات المتسلسلة.

حلول GShard:

1. **الحوسبة المشروطة :**يتم تنشيط أجزاء محددة من الشبكة بناءً على المدخلات، مما يقلل التكاليف الحاسوبية.
2. **التقسيم التلقائي :**يقوم GShard بتقسيم العمليات الحاسوبية تلقائيًا عبر أجهزة متعددة، مما يجعل التنفيذ أكثر كفاءة.
3. **تعزيز الأداء :**يستخدم GShard تحسينات مثل تحويل SPMD (Single Program Multiple Data) لتحسين وقت الترجمة وتقليل تكاليف الاتصال بين الأجهزة.

أهمية البحث:

تحسين جودة النماذج:

- باستخدام GShard ، تم تحسين جودة الترجمة (BLEU) بنسبة كبيرة مقارنة بالنماذج السابقة.
- تم تحقيق هذه التحسينات مع تقليل زمن التدريب مقارنة بالطرق التقليدية.

كفاءة التدريب:

- نموذج GShard يمكنه تدريب نموذج بحجم 600 مليار معلمة في 4 أيام فقط باستخدام 2048 وحدة TPU.
- تكلفة التدريب باستخدام GShard أقل من تدريب 100 نموذج ترجمة فردي.

سهولة الاستخدام:

- يقلل GShard من الجهد البرمجي المطلوب لتقسيم النماذج الكبيرة، مما يسمح للمطورين بالتركيز على تصميم النموذج.

التطبيقات المحتملة:

الترجمة الآلية متعددة اللغات:

- يمكن استخدام GShard لتطوير نماذج ترجمة قادرة على التعامل مع مئات اللغات في نموذج واحد.

النماذج اللغوية الضخمة:

- يمكن توسيع نطاق النماذج مثل GPT و BERTلتصبح أكبر وأكثر كفاءة.

التعلم متعدد المهام:

- يمكن تدريب نماذج قادرة على التعامل مع مهام متعددة بفضل بنية "مزيج الخبراء."

الرؤية الحاسوبية:

- يمكن استخدام GShard في تطبيقات مثل تصنيف الصور ومعالجة الفيديو باستخدام تقسيم الأبعاد المكانية.

القيود والتحديات:

1. **الاستقرار العددي:**
   - تدريب النماذج ذات الأحجام الكبيرة جدًا (مثل 1 تريليون معلمة) قد يواجه تحديات في الاستقرار العددي.
2. **التكاليف الأولية:**
   - رغم تقليل زمن التدريب، فإن استخدام وحدات TPU على نطاق واسع قد يكون مكلفًا.
3. **تعقيد التنفيذ:**
   - يتطلب GShard فهمًا عميقًا لتقسيم البيانات والاتصالات بين الأجهزة.

الإنجازات الرئيسية للبحث:

- **تدريب نموذج 600 مليار معلمة :**يُعد هذا النموذج أحد أكبر النماذج التي تم تدريبها بكفاءة حتى الآن.
- **تحسين جودة الترجمة :**حقق النموذج زيادة كبيرة في جودة الترجمة مقارنة بالنماذج السابقة .

- **كفاءة التدريب :**تم تقليل التكلفة الزمنية لتدريب النماذج بشكل كبير.

البحث :GShard: توسيع النماذج العملاقة باستخدام الحوسبة المشروطة والتقسيم التلقائي

الكلمات المفتاحية:

#النماذج_العملاقة #التعلم_العميق #الترجمة_الآلية# الذكاء_الاصطناعي
#مركز_أبحاث_الذكاء_الاصطناعي #آيرند

Tags:
#AI #Artificial_Intelligence #Airnd_Center #DeepLearning
#MachineTranslation #ScalableAI #ConditionalComputation
#NeuralNetworks

GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding

Dmitry Lepikhin
lepikhin@google.com

HyoukJoong Lee
hyouklee@google.com

Yuanzhong Xu
yuanzx@google.com

Dehao Chen
dehao@google.com

Orhan Firat
orhanf@google.com

Yanping Huang
huangyp@google.com

Maxim Krikun
krikun@google.com

Noam Shazeer
noam@google.com

Zhifeng Chen
zhifengc@google.com

Abstract

Neural network scaling has been critical for improving the model quality in many real-world machine learning applications with vast amounts of training data and compute. Although this trend of scaling is affirmed to be a sure-fire approach for better model quality, there are challenges on the path such as the computation cost, ease of programming, and efficient implementation on parallel devices. GShard is a module composed of a set of lightweight annotation APIs and an extension to the XLA compiler. It provides an elegant way to express a wide range of parallel computation patterns with minimal changes to the existing model code. GShard enabled us to scale up multilingual neural machine translation Transformer model with Sparsely-Gated Mixture-of-Experts beyond 600 billion parameters using automatic sharding. We demonstrate that such a giant model can efficienctly be trained on 2048 TPU v3 accelerators in 4 days to achieve far superior quality for translation from 100 languages to English compared to the prior art.

1 Introduction

Scaling neural networks brings dramatic quality gains over a wide array of machine learning problems [1, 2, 3, 4, 5, 6]. For computer vision, increasing the model capacity has led to better image classification and detection accuracy for various computer vision architectures [7, 8, 9]. Similarly in natural language processing, scaling Transformers [10] yielded consistent gains on language understanding tasks [4, 11, 12], cross-lingual down-stream transfer [4, 13] and (massively-)multilingual neural machine translation [14, 15, 16]. This general tendency motivated recent studies to scrutinize the factors playing a critical role in the success of scaling [17, 18, 19, 20, 3], including the amounts of training data, the model size, and the computation being utilized as found by past studies. While the final model quality was found to have a power-law relationship with the amount of data, compute and model size [18, 3], the significant quality gains brought by larger models also come with various practical challenges. *Training efficiency* among the most important ones, which we define as the amount of compute and training time being used to achieve a superior model quality against the best system existed, is oftentimes left out.

Preprint. Under review.

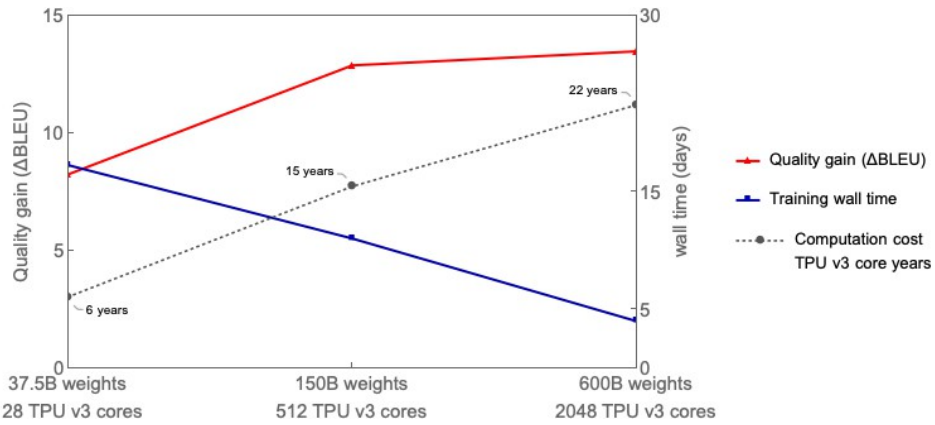


Figure 1: Multilingual translation quality (average ΔBLEU comparing to bilingual baselines) improved as MoE model size grows up to 600B, while the end-to-end training cost (in terms of TPU v3 core-year) only increased sublinearly. Increasing the model size from 37.5B to 600B (16x), results in computation cost increase from 6 to 22 years (3.6x). The 600B parameters model that achieved the best translation quality was trained with 2048 TPU v3 cores for 4 days, a total cost of 22 TPU v3 core-years. In contrast, training all 100 bilingual baseline models would have required 29 TPU v3 core-years. Our best quality dense single Transformer model (2.3B parameters) achieving ΔBLEU of 6.1, was trained with GPipe [15] on 2048 TPU v3 cores for 6 weeks or total of 235.5 TPU v3 core-years.

1.1 Practical Challenges for Scaling

Here we enumerate major practical challenges faced especially when training massive-scale models that are orders of magnitude larger than the capacity limit of a single accelerator memory (e.g., GPUs or TPUs).

Architecture-specific model parallelism support There is a lack of support for efficient model parallelism algorithms under commonly used deep learning frameworks such as TensorFlow [21] and PyTorch [22]. Naive model parallelism with graph partition is supported but it would lead to severe under-utilization due to the sequential dependency of the network and gradient based optimization. In order to scale up the existing models efficiently, users typically need to invest a lot of engineering work, for example, migrating the model code to special frameworks [23, 15].

Super-linear scaling of computation cost vs model size Straightforward scaling of the mode size by increasing the depth or width [6, 15] generally results in at least linear increase of training step time. Model parallelism by splitting layer weights and computation across multiple devices generally becomes necessary, leading to network communication overhead and device under-utilization. Device under-utilization stems from imbalanced assignment and sequential dependencies of the underlying neural network. This super-linear relationship between the computation cost and the model size can not be resolved by simply using more devices, making training massive models impractical.

Infrastructure scalability for giant model representation A naive graph representation for the massive-scale model distributed across thousands of devices may become a bottleneck for both deep learning frameworks and their optimizing compilers. For example, adding D times more layers with inter-op partitioning or increasing model dimensions with intra-op partitioning across D devices may result in a graph with $O(D)$ nodes. Communication channels between devices could further increase the graph size by up to $O(D^2)$ (e.g., partitioning gather or transpose). Such increase in the graph size would result in an infeasible amount of graph building and compilation time for massive-scale models.

Non-trivial efforts for implementing partitioning strategies Partitioning a model to run on many devices efficiently is challenging, as it requires coordinating communications across devices. For graph-level partitioning, sophisticated algorithms [15, 24] are needed to reduce the overhead

introduced by the sequential dependencies between different partitions of graphs allocated on different devices. For operator-level parallelism, there are different communication patterns for different partitioned operators, depending on the semantics, e.g., whether it needs to accumulate partial results, or to rearrange data shards. According to our experience, manually handling these issues in the model requires substantial amount of effort, given the fact that the frameworks like TensorFlow have a large sets of operators with ad-hoc semantics. In all cases, implementing model partitioning would particularly be a burden for practitioners, as changing model architecture would require changing the underlying device communications, causing a ripple effect.

1.2 Design Principles for Efficient Training at Scale

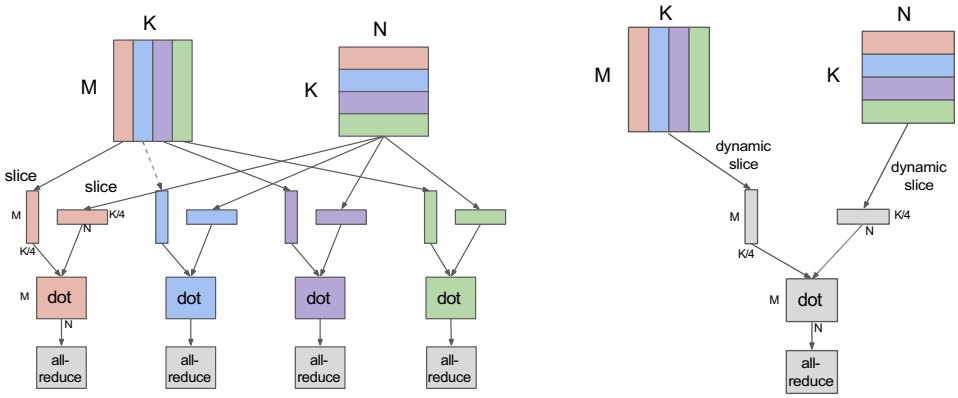
In this paper, we demonstrate how to overcome these challenges by building a 600 billion parameters sequence-to-sequence Transformer model with Sparsely-Gated Mixture-of-Experts layers, which enjoys sub-linear computation cost and $O(1)$ compilation time. We trained this model with 2048 TPU v3 devices for 4 days on a multilingual machine translation task and achieved far superior translation quality compared to prior art when translating 100 languages to English with a single non-ensemble model. We conducted experiments with various model sizes and found that the translation quality increases as the model gets bigger, yet the total wall-time to train only increases sub-linearly with respect to the model size, as illustrated in Figure 1. To build such an extremely large model, we made the following key design choices.

Sub-linear Scaling First, model architecture should be designed to keep the computation and communication requirements sublinear in the model capacity. Conditional computation [25, 16, 26, 27] enables us to satisfy training and inference efficiency by having a sub-network activated on the per-input basis. Scaling capacity of RNN-based machine translation and language models by adding Position-wise Sparsely Gated Mixture-of-Experts (MoE) layers [16] allowed to achieve state-of-the-art results with sublinear computation cost. We therefore present our approach to extend Transformer architecture with MoE layers in Section 2.

The Power of Abstraction Second, the model description should be separated from the partitioning implementation and optimization. This separation of concerns let model developers focus on the network architecture and flexibly change the partitioning strategy, while the underlying system applies semantic-preserving transformations and implements efficient parallel execution. To this end we propose a module, GShard, which only requires the user to annotate a few critical tensors in the model with partitioning policies. It consists of a set of simple APIs for annotations, and a compiler extension in XLA [28] for automatic parallelization. Model developers write models as if there is a single device with huge memory and computation capacity, and the compiler automatically partitions the computation for the target based on the annotations and their own heuristics. We provide more annotation examples in Section 3.2.

Scalable Compilers Third, the system infrastructure, including the computation representation and compilation, must scale with thousands of devices for parallel execution. For example, Figure 2 illustrates two different ways of partitioning a dot-product operation across 4 devices (color-coded). Notice that with the usual MPMD (Multiple Program Multiple Data) approach in Figure 2a scaling becomes more challenging since the number of nodes in the graph increases linearly with the number of devices. Instead, we developed a compiler technique for SPMD (Single Program Multiple Data) transformation that generates a single program to run on all devices, keeping the compilation time constant independent of the number of devices, as illustrated in Figure 2b. We will discuss our SPMD framework in more details in Section 3.3.

The rest of the paper is organized as the following. Section 2 describes our Transformer architecture with Sparsely-Gated MoE layer in more details. Section 3 introduces our development module GShard. Section 4 demonstrates the application of our mixture of expert models on the multilingual machine translation task over 100 language pairs. Section 5 has performance and memory measurements of our implementation. Section 6 discusses related work.



(a) MPMD Partition (b) SPMD Partition

Figure 2: Comparison between MPMD and our proposed SPMD partitioning of a Dot operator ($[M, K] \times [K, N] = [M, N]$) across 4 devices. In this example, both operands are partitioned along the contracting dimension K, where each device computes the local result and globally combines with an AllReduce. MPMD partitioning generates separate operators for each device, limiting its scalability, whereas SPMD partitioning generates one program to run on all devices. Note that the compilation time with our SPMD partitioning is not-dependent of the number of devices being used.

2 Model

2.1 Sparse scaling of the Transformer architecture

The Transformer [10] architecture has been widely used for natural language processing. It has become the de-facto standard for many sequence-to-sequence tasks, such as machine translation. Transformer makes use of two computational blocks, an encoder and a decoder, both implemented by stacking multiple Transformer layers. Transformer encoder layer consists of two consecutive layers, namely a self-attention layer followed by a position-wise feed-forward layer. Decoder adds third cross-attention layer, which attends over encoder output. We sparsely scale Transformer with conditional computation by replacing every other feed-forward layer with a Position-wise Mixture of Experts (MoE) layer [16] with a variant of top-2 gating in both the encoder and the decoder (Figure 3). We vary the number of Transformer layers and the number of experts per MoE layer in order to scale the model capacity.

Each training example consists of a pair of sequences of subword tokens. Each token activates a sub-network of the MoE Transformer during both training and inference. The size of the sub-network is roughly independent of the number of experts per MoE Layer, allowing sublinear scaling of the computation cost as described in the previous section. Computation complexity is further analyzed in Section 3.1 and training performance in Section 5.

2.2 Position-wise Mixture-of-Experts Layer

The Mixture-of-Experts (MoE) layer used in our model is based on [16] with variations in the sparse gating function and the auxiliary loss being used. A MoE layer for Transformer consists of E feed-forward networks $\text{FFN}_1 \ldots \text{FFN}_E$:

$$\mathbf{G}_{s,E} = \text{GATE}(x_s) \tag{1}$$

$$\text{FFN}_e(x_s) = wo_e \cdot \text{ReLU}(wi_e \cdot x_s) \tag{2}$$

$$y_s = \sum_{e=1}^{E} \mathbf{G}_{s,e} \cdot \text{FFN}_e(x_s) \tag{3}$$

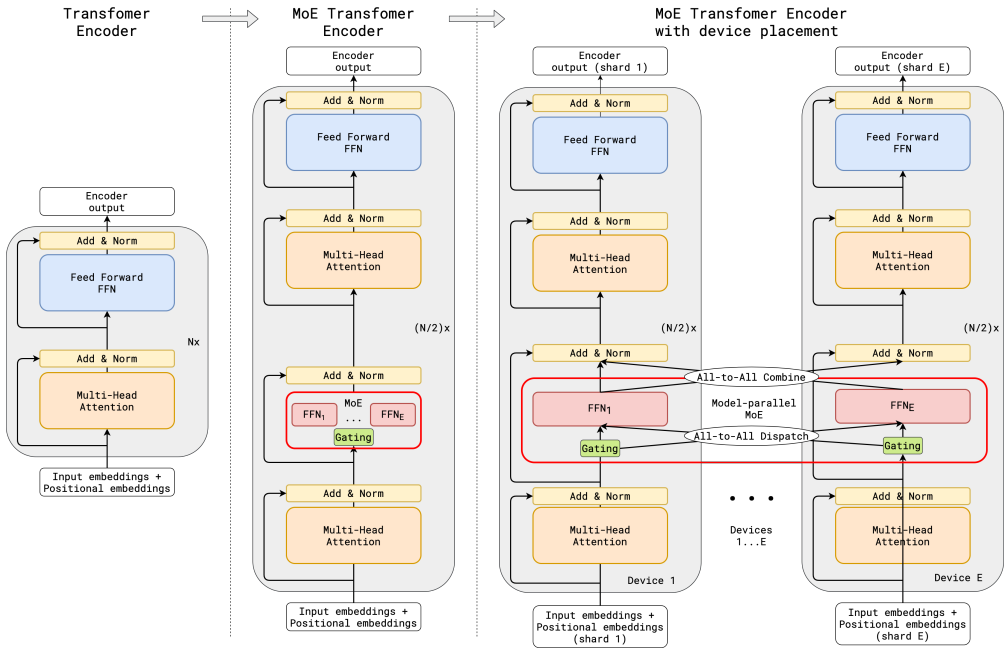


Figure 3: Illustration of scaling of Transformer Encoder with MoE Layers. The MoE layer replaces the every other Transformer feed-forward layer. Decoder modification is similar. (a) The encoder of a standard Transformer model is a stack of self-attention and feed forward layers interleaved with residual connections and layer normalization. (b) By replacing every other feed forward layer with a MoE layer, we get the model structure of the MoE Transformer Encoder. (c) When scaling to multiple devices, the MoE layer is sharded across devices, while all other layers are replicated.

where x_s is the input token to the MoE layer, wiand wobeing the input and output projection matrices for the feed-forward layer (an expert). Vector$G_{s,E}$ is computed by a gating network.$G_{s,E}$ has one non-negative for each expert, most of which are zeros meaning the token is not dispatched to that expert. The token is dispatched to a very small number of experts. We choose to let each token dispatched to at most two experts. The corresponding entries in$G_{s,E}$ are non-zeros, representing how much an expert contributes to the final network output. Every expert FFN_e applies to x_s a fully-connected 2-layer network using ReLU [29] activation function. The output of the MoE layer, y_s, is the weighted average of outputs from all the selected experts.

The gating function GATE(.) is critical to the MoE layer, which is modeled by a softmax activation function to indicate the weights of each expert in processing incoming tokens. In other words, to indicate how good an expert is at processing the incoming token. Furthermore, the gating function must satisfy two goals:

- **Balanced load** It is desirable that the MoE layer to sparsely activate the experts for a given token. A naive solution would be just to choose the top-k experts according to the softmax probability distribution. However, it is known that this approach leads to load imbalance problem for training [16]: most tokens seen during training would have been dispatched to a small number of experts, amassing a very large input buffer for only a few (busy) experts leaving other experts untrained, slowing down the training. Meanwhile many other experts do not get sufficiently trained at all. A better design of the gating function would distribute processing burden more evenly across all experts.
- **Efficiency at scale** It would be rather trivial to achieve a balanced load if the gating function is done sequentially. The computation cost for the gating function alone is at least $O(NE)$ for all N tokens in the input batch given E experts. However, in our study, N is in the order of millions and E is in the order of thousands, a sequential implementation of the gating function would keep most of the computational resources idle most of the time. Therefore, we need an efficient parallel implementation of the gating function to leverage many devices.

We designed the following mechanisms in the gating function GATE(.) to meet the above requirements (details illustrated in Algorithm 1):

- **Expert capacity** To ensure the load is balanced, we enforce that the number of tokens processed by one expert is below some uniform threshold, which we define as expert capacity. Assuming that the total number of tokens in a training batch is N, and each token is dispatched to at most two experts, then the expert capacity is set to be $O(N/E)$. GATE(.) keeps a running counter c_e for how many tokens are dispatched to an expert. When both experts selected by a token already exceed their capacity, the token is considered as an *overflowed* token, where $G_{s,E}$ degenerates into a zero vector. Such tokens have their representation x_s passed on to the next layer via residual connections.
- **Local group dispatching** GATE( ) partitions all tokens in a training batch evenly into G groups, i.e., each group contains $S = N/G$ tokens. All groups are processed independently in parallel. Each group is given a fractional capacity of each expert, $2N/(G\ E)$. Each group ensures that at most this many tokens are dispatched to an expert. In this way, we can ensure that expert capacity is still enforced and the overall load is balanced.
- **Auxiliary loss** It is important that the gating function does not always choose the same few experts, as this would lead to a capacity overflow for only a few experts and under-utilization for the remaining ones. Following [16], we define an auxiliary loss term l_{aux} to enforce this constraint. It is added to the overall loss function of the model $l = l_{nll} + k * l_{aux}$ with a constant multiplier k. The particular form of the auxiliary loss term l_{aux} in line (13) of algorithm 1 is motivated by the following consideration: the term c_e/S represents the fraction of input routed to each expert, and we want to minimize mean square of c_e/S. But because c_e is derived from top-2 operation and is not differentiable, we use the mean gates per expert m_e as a differentiable approximation and replace $(c_e/S)^2$ with $m_e(c_e/S)$, which can now be optimized with gradient descent.
- **Random routing** Intuitively, because y_s is a weighted average of what selected experts return, if the weight for the 2nd expert is very small, we can simply ignore the 2nd expert to conserve the overall expert capacity. Hence, in addition to respecting the expert capacity constraint, GATE(.) dispatches to the 2nd-best expert with the probability proportional to its weight g_2.

3 Highly Parallel Implementation using GShard

This section describes the implementation of the model in Section 2 that runs efficiently on a cluster of TPU devices.

The first step is to express the model in terms of linear algebra operations, in which our software stack (TensorFlow [21]) and the hardware platform (TPU) are highly tailored and optimized. It is readily easy to code up most of the model in terms of linear algebra in the same way as the original Transformer. However, it requires some effort to express the MoE Layer, in particular GATE(.) function presented in Algorithm 1 due to its sequential nature, and we describe the details in Section 3.1.

Next, we annotate the linear algebra computation to express parallelism. Each tensor in the computation can be annotated for replication or distribution across a cluster of devices using sharding APIs in Section 3.2. Using sharding annotations enables separation of concerns between the model description and the efficient parallel implementation, and allows users to flexibly express diverse parallelization strategies. For example, (1) the attention layer is parallelized by splitting along the batch dimension and replicating its weights to all devices. On the other hand, (2) experts in the MoE layer are infeasible to be replicated in all the devices due to its sheer size and the only viable strategy is to shard experts into many devices. Furthermore, the whole model alternates between these two modes (1)-(2). Using annotations frees model developers from the system optimization efforts and avoids baking the parallel implementation and low-level details into the model code.

Finally, the compiler infrastructure takes a (partially) annotated linear algebra computation and produces an efficient parallel program that scales to thousands of devices. As will be described in Section 3.3, the compiler applies SPMD (Single Program Multiple Data) partitioning transformation to express per-device computation, inserts necessary cross-device communication, handles irregular

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

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

$c_E \leftarrow 0$ — d gating decisions per expert
$g_{S,E} \leftarrow softmax(wg \cdot x_S)$ — d gates per token per expert, wg are trainable weights
$m_E \leftarrow \frac{1}{S} \sum_{s=1}^{S} g_{s,E}$ — d mean gates per expert
for $s \leftarrow 1$ **to** S **do**
  $g1, e1, g2, e2 = top\_2(g_{s,E})$ — d top-2 gates and expert indices
  $g1 \leftarrow g1/(g1 + g2)$ — d normalized $g1$
  $c \leftarrow c_{e1}$ — d position in $e1$ expert buffer
  if $c_{e1} < C$ **then**
    $G_{s,e1} \leftarrow g1$ — d $e1$ expert combine weight for x_s
  end
  $c_{e1} \leftarrow c + 1$ — d incrementing $e1$ expert decisions count
end
$l_{aux} = \frac{1}{E} \sum_{e=1}^{E} \frac{S}{c_e} \cdot m_e$
for $s \leftarrow 1$ **to** S **do**
  $g1, e1, g2, e2 = top\_2(g_{s,E})$ — d top-2 gates and expert indices
  $g2 \leftarrow g2/(g1 + g2)$ — d normalized $g2$
  $rnd \leftarrow uniform(0, 1)$ — d dispatch to second-best expert with probability $\propto 2 \cdot g2$
  $c \leftarrow c_{e2}$ — d position in $e2$ expert buffer
  if $c < C \wedge 2 \cdot g2 > rnd$ **then**
    $G_{s,e2} \leftarrow g2$ — d $e2$ expert combine weight for x_s
  end
  $c_{e2} \leftarrow c + 1$
end

patterns such as uneven partitions, and finally generates a single program to be launched on all devices for parallel execution.

3.1 Positions-wise Mixture-of-Expert Layer Expressed in Linear Algebra

Our model implementation (Algorithm 2) views the whole accelerator cluster as a single device and expresses its core mathematical algorithm in a few tensor operations independent of the concrete setup of the cluster. Einstein summation notation [30] (i.e., tf.einsum) is a powerful construct to concisely express the model and we use it extensively in our implementation. The softmax gates computation is trivially expressed by one einsum followed by the softmax function. Dispatching of inputs to selected experts is expressed by a single einsum between the dispatching mask and the input. All FFN_e weights are combined into single 3-D tensors wi amd wo and the computation by $FFN_1 \ldots FFN_E$ is expressed using 3 operators (two einsum and one relu). Finally, taking weighted average of all experts output into the final output is expressed in another einsum.

Top2Gating in Algorithm 2 computes the union of all group-local $G_{S,E}$ described in Algorithm 1. combine_weights is a 4-D tensor with shape [G, S, E, C]. The value combine_weights[g, s, e, c] is non-zero when the input token s in group g is sent to the input buffer of expert e at buffer position c. For a specific g and s, a slice combine_weight[g, s, :, :] contains at most two non-zero vaules. Binary dispatch_mask is produced from combine_weights by simply setting all non-zero values to 1.

We need to choose the number of groups G and the number of experts E properly so that the algorithm can scale to a cluster with D devices. It is worthwhile to analyze its overall computation complexity (the total number of floating point operations) for a training step given a training batch of N tokens.

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

```
gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
combine_weights, dispatch_mask = Top2Gating(gates)
dispatched_expert_inputs = einsum(
    "GSEC,GSM->EGCM", dispatch_mask, reshaped_inputs)
h = einsum("EGCM,EMH->EGCH", dispatched_expert_inputs, wi)
h = relu(h)
expert_outputs = einsum("EGCH,EHM->GECM", h, wo)
outputs = einsum(
    "GSEC,GECM->GSM", combine_weights, expert_outputs)
```

We analyze Algorithm 2 computation complexity scaling with number the of devices D with the following assumptions: *a)* number of tokens per device $\frac{N}{D} = O(1)$ is constant[1]; *b)* $G = O(D)$, $S = O(1)$ and $N = O(GS) = O(D)$; *c)* $M = O(1)$, $H = O(1)$; *d)* $E = O(D)$; and *e)* $C = O(\frac{2S}{E}) = O(\frac{1}{D})$, $D < S$ and is a positive integer[2] .

The total number of floating point operations $FLOPS$ in Algorithm 2:

$$
\begin{array}{llll}
FLOPS_{\text{Softmax}} & +FLOPS_{\text{Top2Gating}} & +FLOPS_{\text{Dispatch|Combine}} & +FLOPS_{\text{FFN}} = \\
O(GSME) & +O(GSEC) & +O(GSMEC) & +O(EGCHM) = \\
O(D \cdot 1 \cdot 1 \cdot D) & +O(D \cdot 1 \cdot D \cdot \frac{1}{D}) & +O(D \cdot 1 \cdot 1 \cdot D \cdot \frac{1}{D}) & +O(D \cdot D \cdot \frac{1}{D} \cdot 1 \cdot 1) = \\
O(D^2) & +O(D) & +O(D) & +O(D)
\end{array}
$$

and consequently per-device $FLOPS/D = O(D) + O(1) + O(1) + O(1)$. Per-device softmax complexity $FLOPS_{\text{softmax}}/D = O(D)$ is linear in number of devices, but in practice is dominated by other terms since $D << H$ and $D < S$. As a result $FLOPS/D$ could be considered $O(1)$, satisfying sublinear scaling design requirements. Section 5 verifies this analysis empirically.

In addition to the computation cost, we have non-constant cross-device communication cost, but it grows at a modest rate $O(\sqrt{D})$ when we increase D (Section 5).

3.2 GShard Annotation API for Parallel Execution

Due to the daunting size and computation demand of tensors in Algorithm 1, we have to parallelize the algorithm over many devices. An immediate solution of how to shard each tensor in the algorithm is illustrated by underscored letters in Algorithm 2. The *sharding* API in GShard allows us to annotate tensors in the program to selectively specify how they should be partitioned. This information is propagated to the compiler so that the compiler can automatically apply transformations for parallel execution. We use the following APIs in TensorFlow/Lingvo [31] in our work.

- **replicate(tensor)** annotates tensor to be replicated across partitions, and returns the annotated tensor. This is often used for the non-MoE layers in our model to replicate the weights.
- **split(tensor, split_dimension, num_partitions)** annotates tensor to be partitioned along split_dimension, and returns the annotated tensor. Partition *i* is placed on the *i*'th device, and num_partitions must not exceed the number of devices on the system.
- **shard(tensor, device_assignment)** generalizes split() to allow partitioning multiple dimensions and specifying the placement of each partition. Appendix A.3 describes this API with more details.

[1]This is oftentimes necessary in practice to avoid overflowing device memory.

[2]Scaling $D > S$ would require different use of fractional expert capacity.

Note that the invocations to split or shard only adds annotations and does not change the logical shape in the user program. The user still works with full shapes and does not need to worry about issues like uneven partitioning.

GShard is general in the sense that the simple APIs apply to all dimensions in the same way. The sharded dimensions could include batch (data-parallelism), feature, expert, and even spatial dimensions in image models, depending on the use cases. Also, since the sharding annotation is per tensor, different parts of the model can be partitioned in different ways. This flexibility enables us to partition the giant MoE weights and switch partition modes between MoE and non-MoE layers, as well as uses cases beyond this paper, e.g., spatial partitioning of large images [32] (Appendix A.4).

With the above sharding APIs, we can express the sharding strategy shown in Algorithm 2 as below. The input tensor is split along the first dimension and the gating weight tensor is replicated. After computing the dispatched expert inputs, we apply split to change the sharding from the group (G) dimension to the expert (E) dimension. D is device count.

```
1    # Partition inputs along group (G) dim.
2  + inputs = split(inputs, 0, D)
3    # Replicate the gating weights
4  + wg = replicate(wg)
5    gates = softmax(einsum("GSM,ME->GSE", inputs, wg))
6    combine_weights, dispatch_mask = Top2Gating(gating_logits)
7    dispatched_expert_inputs = einsum(
8      "GSEC,GSM->EGCM", dispatch_mask, reshaped_inputs)
9    # Partition dispatched inputs along expert (E) dim.
10 + dispatched_expert_inputs = split(dispatched_expert_inputs, 0, D)
11   h = einsum("EGCM,EMH->EGCH", dispatched_expert_inputs, wi)
12   ...
```

Per-tensor sharding assignment As shown in the example above, users are not required to annotate every tensor in the program. Annotations are typically only required on a few important operators like Einsums in our model and the compiler uses its own heuristics to infer sharding for the rest of the tensors [3]. For example, since the input tensor is partitioned along G and the weight tensor is replicated, the compiler chooses to partition the einsum output along the same G dimension (Line 5). Similarly, since both inputs are partitioned along the G dimension for the input dispatch einsum (Line 7), the output sharding is inferred to be split along the G dimension, and then we add the split annotation on the output to reshard along the E dimension. Some annotations in the above example could also be determined by the compiler (e.g., replicate(wg)) but it is recommended to annotate the initial input and final output tensors of the computation.

The compiler currently uses an iterative data-flow analysis to propagate sharding information from an operator to its neighbors (operands and users), starting from the user-annotated operators. The analysis tries to minimize the chance of resharding by aligning the sharding decisions of adjacent operators. There could be other approaches such as integer programming or machine-learning methods, but improving the automatic sharding assignment is not the focus of this paper and we leave it as future work.

Mixing manual and automatic sharding Automatic partitioning with sharding annotations is often enough for common cases, but GShard also has the flexibility to allow mixing manually partitioned operators with auto-partitioned operators. This provides users with more controls on how operators are partitioned, and one example is that the user has more run-time knowledge beyond the operators' semantics. For example, neither XLA's nor TensorFlow's Gather operator definition conveys information about the index bounds for different ranges in the input, but the user might know that a specific Gather operator shuffles data only within each partition. In this case, the user can trivially partition the operator by simply shrinking the dimension size and performing a local Gather; otherwise, the compiler would need to be conservative about the index range and add unnecessary communication overhead. For example, the dispatching Einsum (Line 3) in Algorithm 2

[3] It is also important for the compiler to infer missing shardings since the backpropagation computation is often automatically generated by the frontend framework and users don't have access to those tensors.

in Algorithm 2, which uses an one-hot matrix to dispatch inputs, can be alternatively implemented with a Gather operator using trivial manual partitioning, while the rest of the model is partitioned automatically. Below is the pseudocode illustrating this use case.

```
# input has shape [G, S, M]. split() does not change logical shape.
input = split(input, 0, num_devices)
# s_indices has shape [E, G, C, 1]. Values: indices to S in input.
s_indices = split(s_indices, 1, num_devices)

# Begin manual partitioning.
# partitioned_input has shape [G/num_devices, S, M]
partitioned_input = auto_to_manual_spmd_partition(input)
# partitioned_s_indices has shape [E, G/num_devices, C, 1]
partitioned_s_indices = auto_to_manual_spmd_partition(s_indices)
# Concat with G indices in partitioned_input: Iota on G dimension.
partitioned_gs_indices = concat(
    iota([E, G/num_devices, C, 1], 1), partitioned_s_indices, 3)
# partitioned_data has shape [E, G/num_devices, C, M]
partitioned_data = gather(
    partitioned_input, partitioned_gs_indices)

# Switch back to auto partitioning.
# data has shape [E, G, C, M]
data = manual_to_auto_spmd_partition(partitioned_data)
...
```

3.3 The XLA SPMD Partitioner for GShard

This section describes the compiler infrastructure that automatically partitions a computation graph based on sharding annotations. Sharding annotations inform the compiler about how each tensor should be distributed across devices. The SPMD (Single Program Multiple Data) partitioner (or "partitioner" for simplicity) is a compiler component that transforms a computation graph into a single program to be executed on all devices in parallel. This makes the compilation time near constant regardless of the number of partitions, which allows us to scale to thousands of partitions. [4]

We implemented the partitioner in the XLA compiler [28]. Multiple frontend frameworks including TensorFlow, JAX, PyTorch and Julia already have lowering logic to transform their graph representation to XLA HLO graph. XLA also has a much smaller set of operators compared to popular frontend frameworks like TensorFlow, which reduces the burden of implementing a partitioner without harming generality, because the existing lowering from frontends performs the heavy-lifting to make it expressive. Although we developed the infrastructure in XLA, the techniques we describe here can be applied to intermediate representations in other machine learning frameworks (e.g., ONNX [33], TVM Relay [34], Glow IR [35]).

XLA models a computation as a dataflow graph where nodes are operators and edges are tensors flowing between operators. The core of the partitioner is per-operation handling that transforms a full-sized operator into a partition-sized operator according to the sharding specified on the input and output. When a computation is partitioned, various patterns of cross-device data transfers are introduced. In order to maximize the performance at large scale, it is essential to define a core set of communication primitives and optimize those for the target platform.

3.3.1 Communication Primitives

Since the partitioner forces all the devices to run the same program, the communication patterns are also regular and XLA defines a set of collective operators that perform MPI-style communications [36]. We list the common communication primitives we use in the SPMD partitioner below.

[4] An alternative is MPMD (Multiple Program Multiple Data), which does not scale as shown in Figure 2.

CollectivePermute This operator specifies a list of source-destination pairs, and the input data of a source is sent to the corresponding destination. It is used in two places: changing a sharded tensor's device order among partitions, and halo exchange as discussed later in this section.

AllGather This operator concatenates tensors from all participants following a specified order. It is used to change a sharded tensor to a replicated tensor.

AllReduce This operator performs elementwise reduction (e.g., summation) over the inputs from all participants. It is used to combine partially reduced intermediate tensors from different partitions. In a TPU device network, AllReduce has a constant cost when the number of partition grows (Section 5.2). It is also a commonly used primitive with efficient implementation in other types of network topology [37].

AllToAll This operator logically splits the input of each participant along one dimension, then sends each piece to a different participant. On receiving data pieces from others, each participant concatenates the pieces to produce its result. It is used to reshard a sharded tensor from one dimension to another dimension. AllToAll is an efficient way for such resharding in a TPU device network, where its cost increases sublinearly when the number of partitions grows (Section 5.2).

3.3.2 Per-Operator SPMD Partitioning

The core of the partitioner is the per-operator transformation from a full-sized operator into a partition-sized operator according to the specified sharding. While some operators (e.g., elementwise) are trivial to support, we discuss several common cases where cross-partition communications are required.

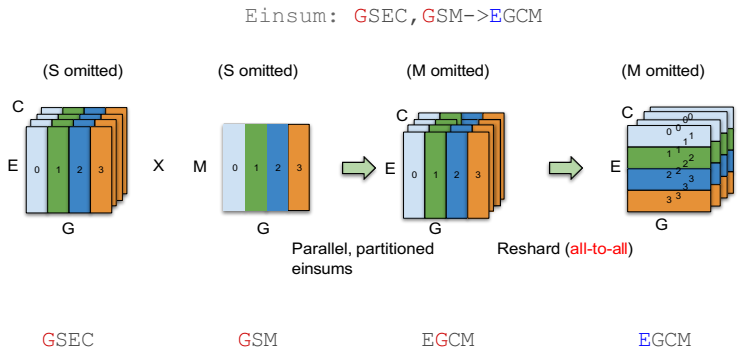
There are a few important technical challenges in general cases, which we will cover in Section 3.3.3. To keep the discussion more relevant to the MoE model, this section focuses on Einsum partitioning to illustrate a few communication patterns. And to keep it simple for now, we assume that all tensors are evenly partitioned, which means the size of the dimension to partitition is a multiple of the partition count.

Einsum Case Study Einsum is the most critical operator in implementing the MoE model. They are represented as a Dot operation in XLA HLO, where each operand (LHS or RHS) consists of three types of dimensions:

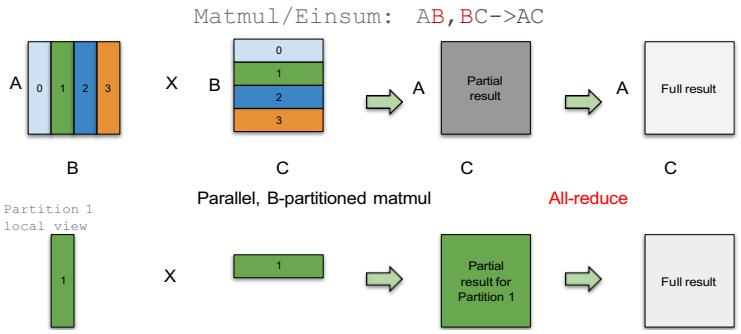
- **Batch dimensions** are the embarrassingly parallel dimensions. The same set of batch dimensions must exist in all of LHS, RHS and the output, and each element in the output only depends on the corresponding batch in LHS and RHS.
- **Contracting dimensions** only exist in the operands. LHS and RHS must have the same set of contracting dimensions, and they are summed up and collapsed in the output.
- **Non-contracting dimensions** are also parallel dimensions that exist in one of the operands and the output. Each of LHS and RHS has its own set of non-contracting dimensions, which are inherited by the output.

Sharding propagation prioritizes choosing the same sharding on batch dimensions of LHS, RHS and output, because that would avoid any cross-partition communication. However, that is not always possible, and we need cross-partition communication in the following three cases.

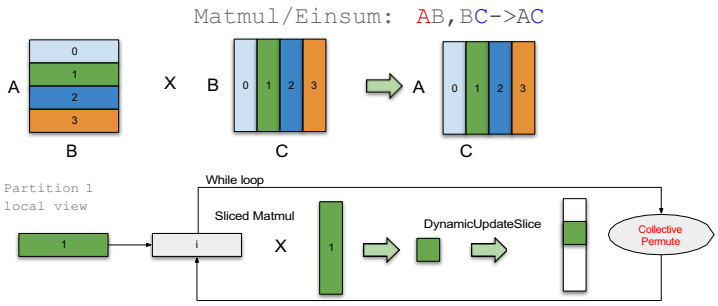
- **Resharding.** In the MoE model we built, the expert dispatching logic (Line 3 in Algorithm 2) requires switching the partitioned dimension after an Einsum. Since resharding is efficient (Section 5.2) with AllToAll, we first execute the Einsum locally, then reshard it to the desired dimension, as shown in Figure 4a.
- **Accumulating partial results.** If the inputs are partitioned along contracting dimensions, the local result is partial and we need to use an AllReduce to combine them and produce the final result, as shown in Figure 4b.
- **Slicing in a loop.** For certain scenarios, we also implemented an algorithm similar to Cannon's algorithm [38], in order to limit the size of tensors on each partition. For example,



(a) A partitioned Einsum operator. Colored letters (*G* and *E*) represent the partitioned dimension of each tensor. The partitioner decides to first execute a batch-parallel Einsum along the *G* dimension, then reshard the result to the *E* dimension.



(b) A simple Einsum (Matmul) partitioned on the contracting dimension.



(c) An Einsum (Matmul) where we use collective-permute in a loop to compute one slice at a time. There is no full-sized tensor during the entire process.

Figure 4: Examples of Einsum partitioning with cross-device communication.

if both operands are partitioned on a non-contracting dimension, we cannot compute the local Einsum directly since operands have different non-contracting dimensions. Replicating one of the operands would not cause redundant computation, but it requires the replicated operand to fit in device memory. Therefore, if the size of the operand is too large, we instead keep both operands partitioned and use a loop to iterate over each slice of the result, and use CollectivePermute to communicate the input slices (Figure 4c).

3.3.3 Supporting a Complete Set of Operators

We solved several additional challenges to enable the SPMD partitioner to support a complete set of operators without extra constraints of tensor shapes or operator configurations. These challenges often involve asymmetric compute or communication patterns between partitions, which are particularly

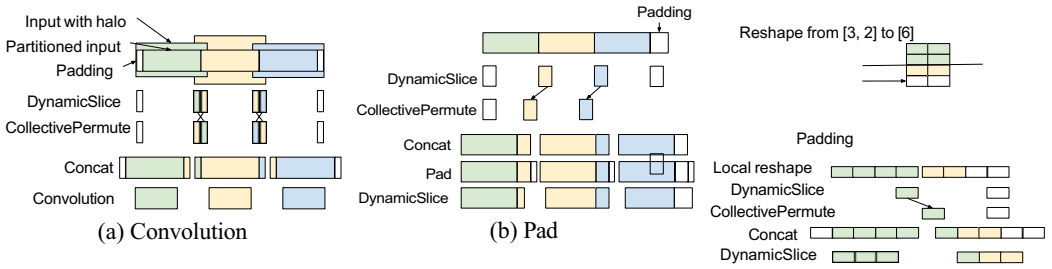


Figure 5: Halo exchange examples.

hard to express in SPMD, since the single program needs to be general enough for all partitions. We cannot simply create many branches in the single program based on the run-time device ID, because that would lead to an explosion in program size.

Static shapes and uneven partitioning XLA requires tensor shapes to be static. [5] However, when a computation is partitioned, it's not always the case that all partitions have the same input/output shapes, because dimensions may not be evenly divisible by the number of partitions. In those cases, the size of the shape is rounded up to the next multiple of partition count, and the data in that padded region can be arbitrary.

When computing an operator, we may need to fill in a known value to the padded region for correctness. For example, if we need to partition an Reduce-Add operator, the identity value of zero needs to be used. Consider an example where the partitioned dimension (15) cannot be divided into 2 (partition count), so Partition 1 has one more column than needed. We create an Iota operator of range [0, 8), add the partition offset (calculated from $\mathit{PartitionId} \times 8$), and compare with the full shape offset (15). Based on the predicate value, we select either from the operand or from zero, and the result is the masked operand.

Static operator configurations XLA operators have static configurations, like the padding, stride, and dilation defined in Convolution. However, different partitions may not execute with the same operator configuration. E.g., for a Convolution, the left-most partition applies padding to its left while the right-most partition applies padding to its right. In such cases, the partitioner may choose configurations that make some partitions to produce slightly more data than needed, then slice out the the irrelevant parts. Appendix A.4 discusses examples for Convolution and similar operators.

Halo exchange Certain operators have a communication pattern which involves partial data exchange with neighboring partitions, which we call *halo exchange*. We use the CollectivePermute operator to exchange halo data between partitions.

The most typical use case of halo exchange is for partitinoning window-based operators (e.g., Convolution, ReduceWindow), because neighboring partitions may require overlapping input data (Figure 5a). In practice, halo-exchange for these operator often needs to be coupled with proper padding, slicing, and masking due to advanced use of window configurations (dilation, stride, and padding), as well as uneven halo sizes. We describe various scenarios in Appendix A.4.

Another use of halo exchange is for data formatting operators that change the size of the shape. For example, after a Slice or Pad operator, the shape of the tensor changes, and so do the boundaries between partitions. This requires us to realign the data on different partitions, which can be handled as a form of halo exchange (Figure 5b).

Other data formatting operators, although logically not changing the size of the shape, may also need halo exchange, specifically due to the static shape constraint and uneven partitioning. For example, the Reverse operator reverses the order of elements in a tensor, but if it is partitioned unevenly, we need to shift data across partitions to keep the padding logically to the right of the result tensor. Another example is Reshape. Consider reshaping a tensor from [3, 2] to [6], where the input is

[5]The limited dynamism in the intermediate representation is often necessary to efficiently target accelerators.

unevenly partitioned in 2 ways on the first dimension (partition shape [2, 2]), and the output is also partitioned in 2 ways (partition shape [3]). There is padding on the input due to uneven partitioning, but after Reshape, the output tensor no longer has padding; as a result, halo exchange is required in a similar way to Slice (Figure 5c).

Compiler optimizations The SPMD partitioner creates various data formatting operators in order to perform slicing, padding, concatenation, masking and halo exchange. To address the issue, we leverage XLA's fusion capabilities on TPU, as well as code motion optimizations for slicing and padding, to largely hide the overhead of data formatting. As a result, the run-time overhead is typically negligible, even for convolutional networks where masking and padding are heavily used.

4 Massively Multilingual, Massive Machine Translation (M4)

4.1 Multilingual translation

We chose multilingual neural machine translation (MT) [39, 40, 41] to validate our design for efficient training with GShard. Multilingual MT, which is an inherently multi-task learning problem, aims at building a single neural network for the goal of translating multiple language pairs simultaneously. This extends our line of work [15, 14, 16] towards a universal machine translation model [42], i.e. a single model that can translate between more than hundred languages, in all domains. Such massively multilingual translation models are not only convenient for stress testing models at scale, but also shown to be practically impactful in real-world production systems [43].

In massively multilingual MT, there are two criteria that define success in terms of the model quality, 1) improvements attained on languages that have large amounts of training data (high resourced), and 2) improvements for languages with limited data (low-resource). As the number of language pairs (tasks) to be modeled within a single translation model increases, *positive language transfer* [44] starts to deliver large gains for low-resource languages. Given the number of languages considered, M4 has a clear advantage on improving the low-resource tasks. On the contrary, for high-resource languages the increased number of tasks limits per-task capacity within the model, resulting in lower translation quality compared to a models trained on a single language pair. This *capacity bottleneck* for high resourced languages can be relaxed by increasing the model size to massive scale in order to satisfy the need for additional capacity [14, 15].

Massively multilingual, massive MT consequently aims at striking a balance between increasing *positive transfer* by massive multilinguality and mitigating the *capacity bottleneck* by massive scaling. While doing so, scaling the model size and the number of languages considered have to be coupled with a convenient neural network architecture. In order to amplify the *positive transfer* and reduce the *negative transfer*[6], one can naturally design a model architecture that harbours shared components across languages (shared sub-networks), along with some language specific ones (unshared, language specific sub-networks). However, the search space in model design (deciding on what to share) grows rapidly as the number of languages increase, making heuristic-based search for a suitable architecture impractical. Thereupon the need for approaches based on learning the wiring pattern of the neural networks from the data emerge as scalable and practical way forward.

In this section, we advocate how conditional computation [45, 46] with sparsely gated mixture of experts [16] fits into the above detailed desiderata and show its efficacy by **scaling neural machine translation models beyond 1 trillion parameters**, while keeping the training time of such massive networks practical. E.g. a 600B GShard model for M4 can process 1T tokens[7] in 250k training steps in under 4 days. We experiment with increasing the model capacity by adding more and more experts into the model and study the factors playing role in convergence, model quality and training efficiency. Further, we demonstrate how conditional computation can speed up the training [25] and how sparsely gating/routing each token through the network can efficiently be learned without any prior knowledge on task or language relatedness, exemplifying the capability of learning the routing decision directly from the data.

[6]Negative transfer is the notion of sharing the model capacity by unrelated tasks which in return hurts the quality of such *interfering* tasks.

[7]Source side tokens after sub-word segmentation.

4.2 Dataset and Baselines

The premise of progressively larger models to attain greater quality necessitates large amounts of training data to begin with [3]. Following the prior work on dense scaling for multilingual machine translation [15, 14], we committed to the realistic test bed of MT *in the wild*, and use a web-scale in-house dataset. The training corpus, mined from the web [47], contains parallel documents for 100 languages, to and from English, adding up to a total of 25 billion training examples. A few characteristics of the training set is worth mentioning. Having mined from the web, the joint corpus is considerably noisy while covering a diverse set of domains and languages. Such large coverage comes with a heavy imbalance between languages in terms of the amount of examples per language pair. This imbalance follows a sharp power law, ranging from billions of examples for high-resourced languages to tens of thousands examples for low-resourced ones. While the above mentioned characteristics constitute a challenge for our study, it also makes the overall attempt as realistic as possible. We refer reader to [15, 14] for the additional details of the dataset being used.

We focus on improving the translation quality (measured in terms of BLEU score [48]) from all 100 languages to English. This resulted in approximately 13 billion training examples to be used for model training[8]. In order to form our baselines, we trained separate bilingual Neural Machine Translation models for each language pair (e.g. a single model for German-to-English), tuned depending on the available training data per-language[9]. Rather than displaying individual BLEU scores for each language pair, we follow the convention of placing the baselines along the x-axis at zero, and report the ΔBLEU trendline of each massively multilingual model trained with GShard (see Figure 6). The x-axis in Figure 6 is sorted from left-to-right in the decreasing order of amount of available training data, where the left-most side corresponds to high-resourced languages, and low-resourced languages on the right-most side respectively. To reiterate, our ultimate goal in universal machine translation is to amass the ΔBLEU trendline of a single multilingual model above the baselines for all languages considered. We also include a variant of dense 96 layer Transformer Encoder-Decoder network T(96L) trained with GPipe pipeline parallelism on the same dataset as another baseline (dashed trendline in Figure 6). Training to convergence took over 6 weeks on 2048 TPU v3 cores [10], outperforming the original GPipe T(128L)[11] [15] and is the strongest single dense model baseline we use in our comparisons.

4.3 Sparsely-Gated MoE Transformer: Model and Training

Scaling Transformer architecture has been an exploratory research track recently [49, 50, 51]. Without loss of generality, emerging approaches follow scaling Transformer by stacking more and more layers [49, 15], widening the governing dimensions of the network (i.e. model dimension, hidden dimension or number of attention heads) [4, 11] and more recently learning the wiring structure with architecture search [52] [12]. For massively multilingual machine translation, [15] demonstrated the best practices of scaling using GPipe pipeline parallelism; in which a 128 layer Transformer model with 6 billion parameters is shown to be effective at improving high-resource languages while exhibiting the highest *positive transfer* towards low-resource languages. Although very promising, and satisfying our desiderata for universal translation, dense scaling of Transformer architecture has practical limitations which we referred in Section 1 under *training efficiency*.

We aim for practical training time and seek for architectures that warrant training efficiency. Our strategy has three pillars; increase the depth of the network by stacking more layers similar to GPipe [15], increase the width of the network by introducing multiple replicas of the feed-forward networks (experts) as described in Section 2.2 and make use of learned routing modules to (sparsely) assign tokens to experts as described in Section 2.1. With this three constituents, we obtain an

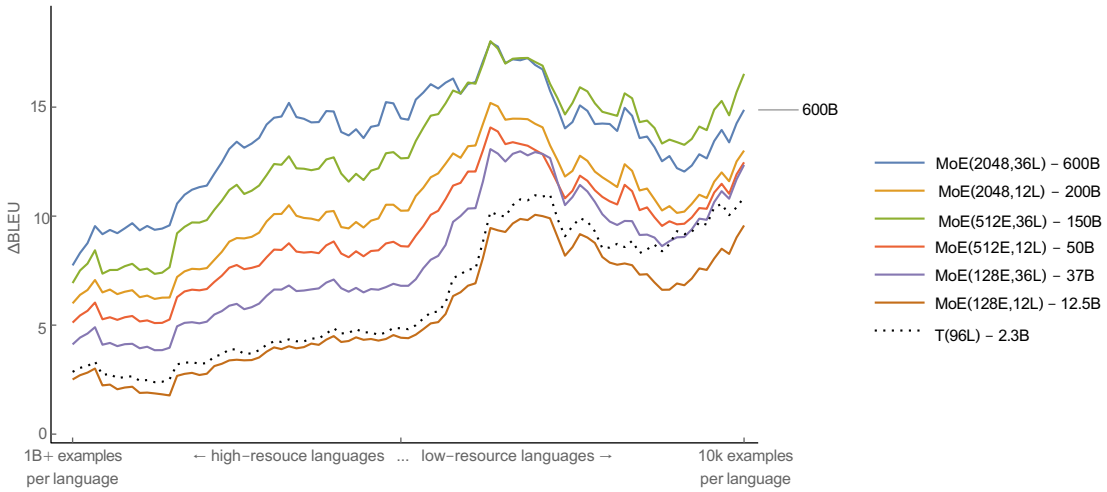
[8]Compared to prior work using the same dataset, Kazakh and Latin to English language pairs were excluded from evaluation.

[9]We tuned batch-size and different values of regularization methods (e.g. dropout) in a Transformer-Big or Transformer-Base layout, for high or low-resourced languages respectively.

[10]T(96L) measured to be processing 1+ trillion tokens at 300k steps, processing around 4M tokens/step, total budget of 235.5 TPU v3 core years

[11]64 encoder + 64 decoder layers, 16384 hidden dim, 32 attention heads

[12]Since the approaches utilizing architecture search are compute intensive, they are not considered within the scope of this work.



Id	Model	BLEU avg.	ΔBLEU avg.	Weights
(1)	MoE(2048E, 36L)	**44.3**	13.5	600B
(2)	MoE(2048E, 12L)	41.3	10.5	200B
(3)	MoE(512E, 36L)	43.7	12.9	150B
(4)	MoE(512E, 12L)	40.0	9.2	50B
(5)	MoE(128E, 36L)	39.0	8.2	37B
(6)	MoE(128E, 12L)	36.7	5.9	12.5B
*	T(96L)	36.9	6.1	2.3B
*	Baselines	30.8	-	100×0.4B

Figure 6: Translation quality comparison of multilingual MoE Transformer models trained with GShard and monolingual baselines. Positions along the x-axis represent languages, raging from high- to low-resource. ΔBLEU represents the quality gain of a single multilingual model compared to a monolingual Transformer model trained and tuned for a specific language. MoE Transformer models trained with GShard are reported with solid trend-lines. Dashed trend-line represents a single 96 layer multilingual Transformer model T(96L) trained with GPipe on same dataset. Each trend-line is smoothed by a sliding window of 10 for clarity. (Best seen in color)

easy to scale, efficient to train and highly expressive architecture, which we call Sparsely-Gated Mixture-of-Experts Transformer or MoE Transformer in short.

Model Details To detail the model specifics, each expert is designed to have the same shape of a regular Transformer feed-forward network, and experts (MoE layers) are distributed once in every other Transformer layer. We tied the number of devices used for training to the number of experts per MoE layer for simplicity, although this is not a requirement. During training, we use float32 for both model weights and activations in order to ensure training stability. We ran additional scalability experiments with MoE(2048E, 60L) with bfloat16 [53] activations with total of 1 trillion model weights. Although trainable by careful and manual diagnostics, with deep 1 trillion model we encountered several trainability issues with numerical stability, hence did not include the results for the sake of reproducibility. For more model and training details, please see Appendix A.2.

4.4 Results

Before going into the details of *training efficiency*, we first investigate the effect of various design choices on building MoE Transformer. In order to prune the search space, we explored varying two

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

Table 1: MoE Transformer model family. To achieve desired capacity we *i)* increased the depth by stacking more layers, *ii)* increased the width of the network by scaling the number of experts per MoE layer along with number of cores used for training.

variables, number of layers in the Transformer encoder-decoder stack (L) and the total number of experts used for every other MoE layer (E). For depth, we tested three different options, 12 (original Transformer depth, which consists of 6 encoder and 6 decoder layers), 36 and 60 layers. For the number of experts that replaces every other feed-forward layer, we also tested three options, namely 128, 512 and 2048 experts. Note that, the number of devices used for training, is fixed to be equal to the number of experts per-layer, using 128, 512 and 2048 cores respectively independent of the depth being experimented. Please also see the detailed description in Table 1 for model configurations.

For each experiment (rows of the Table 1), we trained the corresponding MoE Transformer model until it has seen 1 trillion (10^{12}) tokens. The model checkpoint at this point is used in the model evaluation. We did not observe any over-fitting patterns by this point in any experiment. Instead, we observed that the training loss continued to improve if we kept training longer. We evaluated BLEU scores that the models achieved for all language pairs on a held-out test set. Figure 6 reports all our results.

Here we share a qualitative analysis for each experiment and discuss the implication of each setup on high- and low-resource languages in order to track our progress towards universal translation. To ground the forthcoming analysis, it is worth restating the expected behavior of the underlying quality gains. In order to improve the quality for both high- and low-resource languages simultaneously within a single model, scaled models must mitigate *capacity bottleneck* issue by allocating enough capacity to high-resource tasks, while amplifying the *positive transfer* towards low-resource tasks by facilitating sufficient parameter sharing. We loosely relate the expected learning dynamics of such systems with the long-standing memorization and generalization dilemma, which is recently studied along the lines of width vs depth scaling efforts [54]. Not only do we expect our models to generalize better to the held-out test sets, we also expect them to exhibit high transfer capability across languages as another manifestation of generalization performance [55].

Deeper Models Bring Consistent Quality Gains Across the Board We first investigate the relationship between the model depth and the model quality for both high- and low-resource languages. Three different experiments are conducted in order to test the generalization performance, while keeping the number of experts per-layer fixed. With an increasing number of per-layer experts for each experiment (128, 512 and 2048), we tripled the depth of the network for each expert size, from 12 to 36. This resulted in three groups where experts per-layer fixed but three times the depth within each group:

For each configuration shown in Fig. 6, we observed that increasing the depth (L) while keeping the experts per-layer (E) fixed, brings consistent gains for both low and high resourced languages (upwards Δ shift along the y-axis), almost with a constant additive factor every time we scale the depth from 12L to 36L (2-to-3 BLEU points on average as shown in the last column of Table 3).

Relaxing the Capacity Bottleneck Grants Pronounced Quality Gains Earlier in Section 4.1 we highlighted the influence of the *capacity bottleneck* on task interference, resulting in degraded quality especially for high resourced languages. Later we alleviated this complication by increasing the number of experts per-layer, which in return resulted in a dramatic increase in the number of parameters (weight) of the models studied. Here we investigate whether this so called *capacity*

bottleneck is distinctly observable and explore the impact on model quality and efficiency once it is relaxed. To that end, we first consider three models with identical depths (12L), with increasing number of experts per-layer: 128, 512 and 2048. As we increase the number of experts per-layer from 128 to 512 by a factor of four, we notice a large jump in model quality, +3.3 average BLEU score across 100 languages. However again by four folds scaling of the number of experts per-layer, from 512 to 2048, yields only +1.3 average BLEU scores. Despite the significant quality improvement, this drop in gains hints the emergence of diminishing returns.

Speculatively, the capacity bottleneck is expected to be residing between 128 to 512 experts, for the particular parametrization, number of languages and the amount of training data used in our experimental setup. Once the bottleneck is relaxed, models enjoy successive scaling of the depth, which can be seen by comparing 12 versus 36 layer models both with 128 experts. Interestingly increasing the depth does not help as much if the capacity bottleneck is not relaxed.

Having More Experts Improve Quality Especially for High-Resourced Tasks Another dimension that could shed light on the quality gains of scaling in multi-task models is the contrast between high and low resource language improvements. As mentioned before, low resourced languages benefit from transfer while high resource languages seek for added capacity. Next we examine the effect of increasing the experts per-layer while fixing the depth.

As can be seen in Figure 6, for 12 layer models increase in the expert number yields larger gains for high resourced languages as opposed to earlier revealed diminishing returns for low-resourced languages. A similar pattern is observed also for 36 layer models. While adding more experts relaxes the capacity bottleneck, at the same time it reduces the amount of transfer due to a reduction of the shared sub-networks.

Deep-Dense Models are Better at Positive Transfer towards Low-Resource Tasks Lastly we look into the impact of the depth on low-resourced tasks as a loose corollary to our previous experiment. In order to do so, we include a dense model with 96 layers T(96L) trained with GPipe on the same data into our analysis. We compare T(96L) with the shallow MoE(128E, 12L) model. While the gap between the two models measured to be almost constant for the majority of the high-to-mid resourced languages, the gap grows in favor of the dense-deep T(96L) model as we get into the low-resourced regime. Following our previous statement, as the proportion of the shared sub-networks across tasks increase, which is 100% for dense T(96L), the bandwidth for transfer gets maximized and results in a comparably better quality against its shallow counterpart. Also notice that, the same transfer quality to the low-resourced languages can be achieved with MoE(36E, 128L) which contains 37 billion parameters.

We conjecture that, increasing the depth might potentially increase the extent of transfer to low-resource tasks hence generalize better along that axis. But we also want to highlight that the models in comparison have a disproportionate training resource requirements. We again want to promote the importance of *training efficiency*, which is the very topic we studied next.

4.5 Training Efficiency

In this section we focus on the *training efficiency* of MoE Transformer models. So far, we have seen empirical evidence how scaling the models along various axes bring dramatic quality gains, and studied the factors affecting the extent of the improvements. In order to measure the *training efficiency*, we first keep track of the number of tokens being processed to reach a certain training loss and second we keep track of the wall-clock time for a model to process certain number of tokens. Note that, we focus on the training time and training loss[13] while varying other factors, as opposed to test error, which we analyzed in the previous section.

Deeper models are more sample efficient, converge faster with fewer examples It has been shown that, deeper models are better at sample efficiency, reaching better training/test error given the same amount of training examples [15, 56], commonly attributed to the acceleration effect of over-parametrization [1]. We empirically test the hypothesis again using GShard with MoE Transformers and share trade-offs for models that are not only deep, but also sparsely activated.

[13]Training loss reported in this section corresponds to cross-entropy loss and excludes the auxiliary loss term introduced in Section 2.2

For this purpose, we compare number of tokens being processed by each model to reach a preset training loss. A general trend we observe from Table 2 is that, MoE Transformer models with 3 times the depth need 2 to 3 times fewer tokens to reach the preset training loss thresholds. For example MoE(128E, 12L) takes 3 times the number of tokens to reach 0.7 training cross-entropy compared to MoE(128E, 36L), (6) vs (5). We observe a similar trend for models with 512 and 2048 experts, (4) vs (3) and (2) vs (1).

Id	Model	Cores	Billion tokens to cross-entropy of		
			0.7	0.6	0.5
(1)	MoE(2048E, 36L)	2048	82	175	542
(2)	MoE(2048E, 12L)	2048	176	484	1780
(3)	MoE(512E, 36L)	512	66	170	567
(4)	MoE(512E, 12L)	512	141	486	-
(5)	MoE(128E, 36L)	128	321	1074	-
(6)	MoE(128E, 12L)	128	995	-	-

Table 2: The number of tokens have been seen by a model during training to reach three different cross-entropy loss. A general trend is that deeper models are more sample efficient and converge faster than the comparable shallow ones.

Another intriguing observation from Table 2, is again related to the presence of *capacity bottleneck*. Comparing the models with same depth, (5), (3) and (1), we notice a significant drop in the number of tokens required to reach training loss of 0.7, as we transition from 128 to 512 number of experts. Practically that is where we observed the capacity bottleneck was residing, aligning with the hypothesis in Section 4.4. After this phase shift, models with ample capacity tend to exhibit similar sample efficiency characteristics, as in models (3) and (1).

Largest model (600B) can be trained under 4 days achieving the best quality Next we delve deeper into the interaction between model size and wall-clock time spent for training. We monitor number of TPU cores being used, training steps per-second, total number of tokens per batch, TPU core years[14], and actual wall-clock time spent in days for training (see Table 3 columns respectively).

We start with investigating one of the largest models we trained, MoE(2048E, 36L) with 600 billion parameters, model with id (1). Having utilized 2048 TPU cores for 4 days, this model achieves the best translation quality in terms of average BLEU, but also takes a total of 22.4 TPU years to train. While we have not seen any signs that the quality improvements plateau as we scale up our models, we strive for finding cost-effective solutions for scaling.

Results in Table 3 again validates scaling with conditional computation is way more practical compared to dense scaling. Given the same number of TPU cores used by (1), the dense scaling variant, T(96L), appears to be taking more than ten times to train (235 TPU core years), while trailing behind in terms of model quality compared to models trained with GShard.

Id	Model	Cores	Steps per sec.	Batch sz. (Tokens)	TPU core years	Training time (days)	BLEU avg.
(1)	MoE(2048E, 36L)	2048	0.72	4M	22.4	**4.0**	**44.3**
(2)	MoE(2048E, 12L)	2048	2.15	4M	7.5	1.4	41.3
(3)	MoE(512E, 36L)	512	1.05	1M	15.5	11.0	43.7
(4)	MoE(512E, 12L)	512	3.28	1M	4.9	3.5	40.0
(5)	MoE(128E, 36L)	128	0.67	1M	6.1	17.3	39.0
(6)	MoE(128E, 12L)	128	2.16	1M	1.9	5.4	36.7
*	T(96L)	2048	-	4M	~235.5	~42	36.9

Table 3: Performance of MoE models with different number of experts and layers.

In this section, we benchmarked GShard with MoE Transformers applications to multilingual machine translation (in particular to M4). We identified variables that are affecting the end result, such as

[14]TPU core years is simply measured by the product of number of cores and wall-clock time in years.

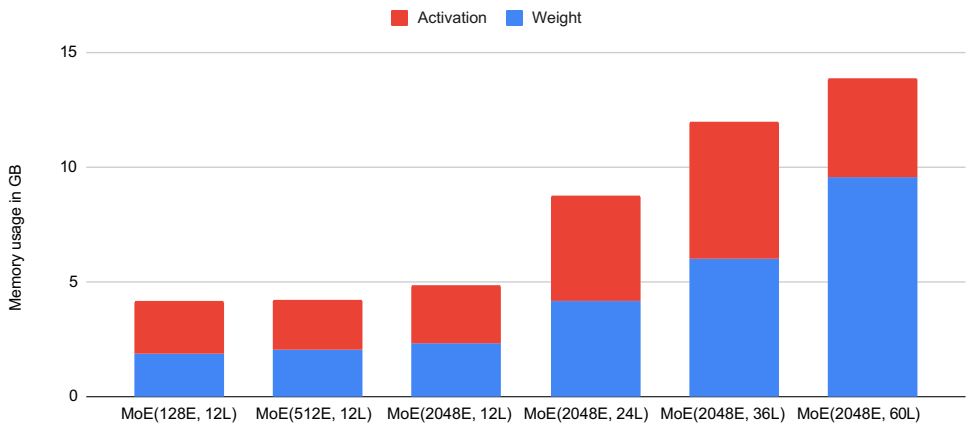


Figure 7: Per-device memory consumption in gigabytes.

capacity bottleneck, *positive transfer* and *training efficiency*, and provided experimental results in order to reveal the interplay between them. Next we will delve deep into performance related topics of GShard, such as memory and runtime efficiency and communication benchmarks.

5 Performance and Memory Consumption

This section discusses how well GShard achieves computation and memory efficiency on the TPU platform. Our measurement and analysis show that the device memory consumption is roughly constant when we increase the number of devices and experts, and the step time grows sublinearly, i.e., 1.7x execution time increase when we scale the model by 16x from 128 devices to 2048 devices. We also provide microbenchmarks and analyses for a variety of partitioned operators, which could guide use cases beyond this paper.

5.1 Memory Efficiency and Scalability

In the GShard model, there are mainly three types of memory usage, all of which have constant per-device sizes after SPMD partitioning, when the number of experts increases.

- Replicated weights (e.g. transformer feed-forward layers).
- Distributed weights (MoE feed-forward layers[15]).
- Activations (output of each layer that is used in both forward and backward pass).

The $O(1)$ memory scaling is demonstrated in Figure 7, which shows the per-device memory usage distribution for different models. With a fixed number of layers, both weight memory and activation memory stay constant when the number of experts increases.

On this other hand, weight memory and activation memory both scale linearly with the number of layers. When the memory requirement exceeds available memory on each device, compiler-based rematerialization will automatically recompute part of the activations in the backward pass in order to reduce peak activation memory. This is why the activation size for MoE(2048E, 60L) is smaller than MoE(2048E, 36L). The overhead of rematerialization is also optimized, e.g. only 28% and 34% of the total cycles are spent on recomputation for 36L and 60L models respectively, and 0% for 12L and 24L since they fit in device memory without rematerialization.

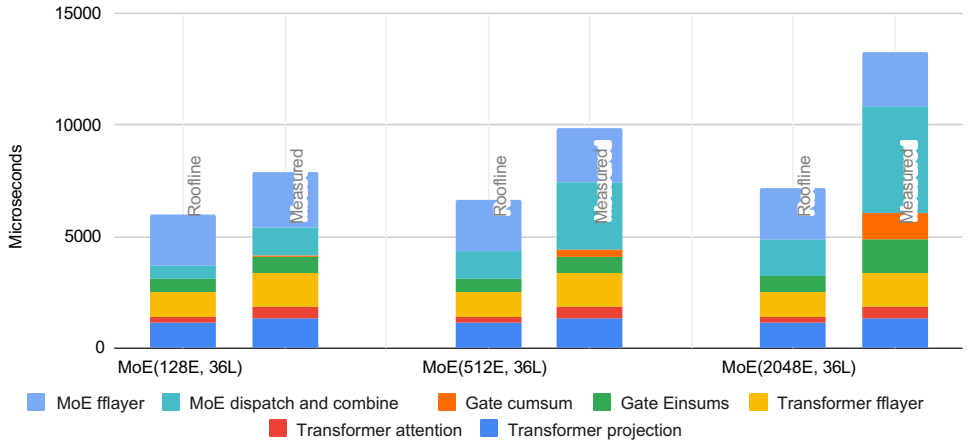


Figure 8: Measured vs roofline execution time breakdown. Only the forward pass is shown, and the backward pass has similar breakdown. "MoE dispatch and combine" represents cross-partition communication with AllToAll.

5.2 Runtime Efficiency and Scalability

Figure 8 shows the breakdown of execution time for an MoE layer and its adjacent Transformer layer. It also compares the achieved performance to a roofline, which is estimated by assuming compute-, memory-, or communication-bounded operations can achieve 100% of the peak FLOPS, memory bandwidth, or interconnect bandwidth. This is a very optimistic estimate as many operators are bounded by a mixed set of resources. At a smaller scale (128 experts), our model can achieve > 70% of the roofline performance. The device time increases by 1.7x when we scale the model to 16x larger (2048 experts), and can still achieve 48% of the roofline performance.

Before analyzing performance scalability, we recall the size scaling of relevant tensor dimensions as discussed in Section 3.1. With D devices, the number of experts E and the group count G are both set to $O(D)$. The fractional per-group expert capacity C is set to $O(1/D)$. This setup cannot scale indefinitely, since C needs to be at least 1, but it is good enough to scale to thousands of experts.

Transformer layers and MoE feed-forward layer These are the dense parts of the model, which are designed to achieve peak TPU utilization. On each device, these computations also have a constant cost when we scale to more experts. Feed-forward layers and Transformer projections are mainly large matrix multiplications that utilize the TPU's matrix unit well. These operations have achieved > 85% peak FLOPS in our experiment. The attention operations are composed of mainly batch matmuls, which are bounded by memory bandwidth when sequence lengths are small. As a result, in our experiments attention operations only achieved > 30% peak FLOPS.

Gate computation In Figure 8, "Gate Einsum" represents the first two and the last Einsums in Algorithm 2. The first Einsum is the projection that calculates per-expert input to softmax. It has an $O(D)$ cost, but it is a very small part of the layer. The other two Einsums are dispatching tokens and combining expert results. They effectively implement Gather with one-hot matrices, which are more expensive, but with constant $O(GC) = O(1)$ cost that is independent from the number of experts. The execution time of these Einsums increases by around 2x when we scale from 128 to 2048 experts (16x).

The remaining per-device gating computation involves many general-purpose computations like ArgMax and Cumsum, which are either memory-bound or even sequential in nature, thus not designed to utilize TPUs well. The majority of the time is spent on sequential Cumsum operations to invert one-hot matrices that represent selected experts for each token to one-hot matrices that represent

[15]Gate projection weights are $O(E)$ in size and could be partitioned, but in practice they are small enough to be replicated and only have negligible effect on peak memory usage.

selected tokens for each expert. The linear complexity of Cumsum is demonstrated in Figure 8. This part of the gating computation also has an $O(D)$ cost, but fortunately, similar to the Einsum before softmax, it has a very small constant factor. It has negligible execution time with 128 experts, and takes less than 10% of the total time spent in the MoE and Transformer layers with 2048 experts.

The most significant part of gating is communication, shown as "MoE dispatch and combine" in Figure 8. These are AllToAll operators, and as we will discuss in Section 5.3, their cost is $O(\sqrt{D})$. When the number experts grows 16x from 128 to 2048, the execution time increases by about 3.75x, and their proportion of execution time in the MoE and Transformer increases from 16% to 36%.

5.3 Communication Microbenchmarks and Per-Operator Scalability

In this section, we measure and analyze the performance scalability of the SPMD partitioner for basic operators, which can be used to guide use cases beyond the MoE model presented in this paper.

Performance scaling of communication primitives Two critical collective communication operators in the MoE model are AllReduce and AllToAll. AllReduce is used in accumulating partial results, and AllToAll is used in resharding (Section 3.3.2). Figure 9 shows their performance scalability from 16 to 2048 partitions. AllReduce on TPU has an execution time independent from the number of devices. The variance in Figure 9 is due to specifics of each topology, e.g., whether it is a square or a rectangle, and whether it is a torus or a mesh.

AllToAll, on the other hand, gets more expensive as the number of partitions grows, but in a sublinear manner. On our 2D TPU cluster, AllToAll cost is roughly $O(\sqrt{D})$, where D is the number of partitions. This is because with a fixed amount of data each partition sends (8MB or 32MB in Figure 9), the total amount of data that all partitions send is $d = O(D)$. Meanwhile, each data piece needs to travel $h = O(\sqrt{D})$ hops on average, and there are overall $l = O(D)$ device-to-device links in the network. Therefore, if it is bandwidth-bound, the execution time of an AllToAll is

$$t = \frac{dh}{l} = O(\frac{D\sqrt{D}}{D}) = O(\sqrt{D}).$$

Even if it is latency-bound, the execution time will still be $O(h) = O(\sqrt{D})$. Comparing 2048 partitions and 16 partitions, while D grows by 128 times, the execution time of AllToAll only increases by 9 times. This enables us to use resharding to efficiently implement cross-partition dispatching (Figure 4a).

AllGather and CollectivePermute are easier to analyze. AllGather's output is D larger than the input, and if we fix input size, then its communication cost is $O(D)$. CollectivePermute has a one-to-one communication pattern, and with reasonable device arrangement where the source-destination pairs are close, its cost is $O(1)$ for a fixed input size.

	$O(D)$ Dimensions	Total Compute	Per-partition	
			Compute	Communication
Add(A,A->A)	A	$O(D)$	$O(1)$	0
Matmul(AB,BC->AC)	B	$O(D)$	$O(1)$	$O(1)$ AR
Matmul(AB,BC->AC)	A	$O(D)$	$O(1)$	0
Matmul(AB,BC->AC)	A,B	$O(D^2)$	$O(D)$	$O(D)$ AG or CP
Matmul(AB,BC->AC)	A,C	$O(D^2)$	$O(D)$	$O(D)$ AG or CP
Reduce(AB->A)	A	$O(D)$	$O(1)$	0
Reduce(AB->B)	A	$O(D)$	$O(1)$	$O(1)$ AR
Einsum(GSEC,GSM->EGCM)	G,E *	$O(D)$	$O(1)$	$O(\sqrt{D})$ AA
Convolution(BIXY,xyIO->BOXY)	X **	$O(D)$	$O(1)$	$O(1)$ CP

Table 4: Scalability of partitioned operators. Abbreviation for communication primitives: AR: AllReduce, AG: AllGather, CP: CollectivePermute, AA: AllToAll. *This is the dispatch Einsum in our model, where we set C to $O(1/D)$. **I/O are the input/output feature dimensions, B is the batch dimension, X/Y are input spatial dimensions, and x/y are the kernal spatial dimensions.

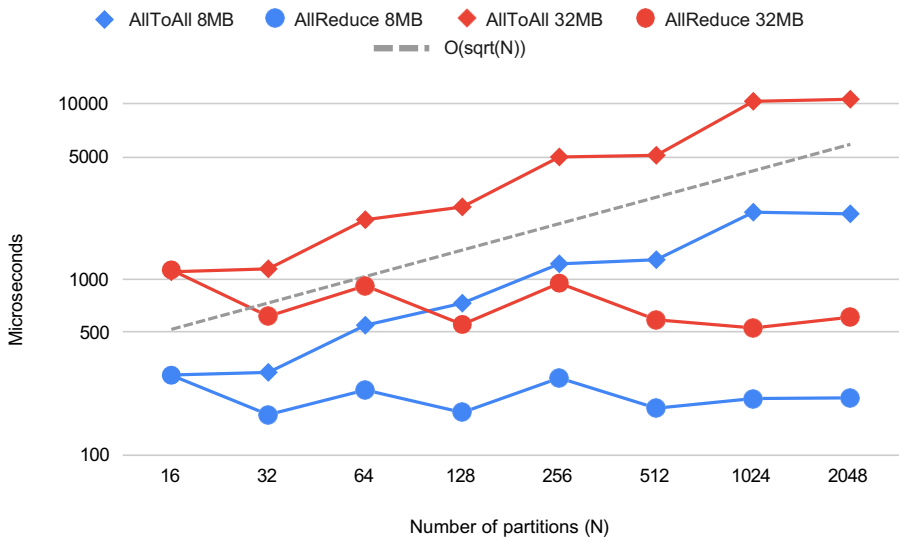


Figure 9: Performance scaling of communication, AllReduce and AllToAll. Log scale on both axes. AllReduce cost is roughly $O(1)$, and AllToAll cost is roughly $O(\sqrt{D})$, where D is the number of partitions. We measure their performance with 8MB and 32MB data. For AllToAll, that means each partition initially has 8MB (or 32MB) data, then divides it to D pieces, and sends each piece to a different receiving partition.

Partitioned operator scalability We summarize the performance scalability for common operators using GShard in Table 4. It contains the Einsum/Matmul examples in Section 3.3.2, and also other common operators like Convolution and Reduce. The table includes the local compute on each partition, as well as the required communication based on our analysis above.

Most operators in Table 4 have sublinear scalability in terms of both compute and communication, which is consistent with our performance measurement of the MoE model. The $O(1)$ scaling of spatially partitioned convolutions also demonstrates the efficiency of GShard for image partitioning (Appendix A.4).

However, the last two Matmul operators in Table 4 have $O(D)$ scaling of per-partition compute and communication, where they have unmatched sharding in the operands. This is not due to inefficiency in the partitioning algorithm, but because the total compute in the full operator is very large ($O(D^2)$). Different partitioning strategies can be used for these cases, producing different communication primitives: replicating one operand will result in AllGather (requiring the replicated operand to fit in device memory), while slicing in a loop (Figure 4c) will result in CollectivePermute.

6 Related Work

Neural networks Deep learning models have been very successful in advancing sub-fields of artificial intelligence. For years, the fields have been continuously reporting new state of the art results using varieties of model architectures for computer vision tasks [57, 58, 7], for natural language understanding tasks [59, 60, 61], for speech recognition and synthesis tasks [62, 63, 64, 65, 66]. More recently, attention-based Transformer models further advanced state of the art of these fields [10, 4].

Model scaling Both academic research and industry applications observed that larger neural networks tend to perform better on large enough datasets and for complex tasks. Within a single model family, simply making the network wider or deeper often improves the model quality empirically. E.g., deeper ResNets performed better [8], bigger Transformer models achieved better translation quality [10], models with larger vocabulary, or embedding or feature crosses work better, too [14, 13]. Across

different model families, it has also been observed that bigger models with larger model capacities not only fit the training data better but also generalize better on test time [67, 68, 15]. This observation motivated many research efforts to build much bigger neural networks than those typically used in deep learning research models or production models. Shazeer et al. showed that a recurrent language model with 69 billion parameters using mixture-of-expert layers achieved much lower test perplexity for the one billion words (LM1B) benchmark [16]. Brown et al. showed that a non-sparse 175 billion parameters model is capable of exhibiting highly accurate few-shot performance on several downstream NLP tasks.

Hardware Neural networks demand non-negligible amounts of computation power. To address such a demand, special hardware (chips and networked machines) built for neural network training and inference can be dated back to 25 years ago [69]. Since late 2000s, researchers started to leverage GPUs to accelerate neural nets [70, 57, 71]. More recently, the industry also invested heavily in building more dedicated hardware systems chasing for more cost-effective neural network hardware [72]. Because the core computation of neural networks (various forms of summation of multiplications: convolution, matrix multiplication, einsum) are highly parallelizable numerical calculations, these chips are equipped with huge number of floating processing units (FPUs). Hence, the compute power of these specially designed hardware grew dramatically. It is reported that GPU price per flops dropped a factor of ten in just the last 4 years [73] and flops per watts increased by 2 magnitude over the past 12 years [74]. The widely available low-cost computation power is a major enabler for the success of neural networks.

Software Software systems supporting neural networks evolved together with the advancement of the underlying hardware [75, 76, 21, 77]. While the accelerators are highly parallel compute machines, they are significantly more difficult to program directly. The frameworks made building neural networks easier and abstracted away many hardware specific details from the practitioners. They in turn rely on lower-level libraries to drive special hardware (accelerators) efficiently. E.g., CUDA [78] for Nvidia's GPUs, or XLA for Google's TPUs [28]. These lower-level libraries are critical for achieving high efficiency using these special hardware.

Parallelism in model training and inference Modern neural networks make extensive use of a cluster of machines for training and inference, each of which equiped with several accelerators. Data parallelism [57] is the most commonly used approach and is supported by major frameworks (TensorFlow [21], PyTorch [22], JAX [79, 80]), where devices run the same program with different input data and combine their local gradients before the weight updates. Model parallelism on the other hand, partitions computation beyond the input batch, which is needed to build very large models. For example, pipelining [15, 24] splits a large model's layers into multiple stages, while operator-level partitioning [23, 81] splits individual operators into smaller parallel operators. GShard used a type of operator-level partitioning to scale our model to a large number of parallel experts.

Automated parallelism Because programming in a distributed heterogeneous environment is challenging, particularly for high-level practitioners, deep-learning frameworks attempt to alleviate the burden of their users from specifying how the distributed computation is done. For example, TensorFlow [21] has support for data parallelism, and basic model parallelism with graph partitioning by per-node device assignment. Mesh TensorFlow [23] helps the user to build large models with SPMD-style per-operator partitioning, by rewriting the computation in a Python library on top of TensorFlow; in comparison, our approach partitions the graph in the compiler based on light-weight annotations without requiring the user to rewrite the model. FlexFlow [81] uses automated search to discover the optimal partition of operators in a graph for better performance; while it focuses on determining the partitioning policy, our SPMD partitioner focuses on the mechanisms to transform an annotated graph. Weight-update sharding [82] is another automatic parallelization transformation based on XLA, which mostly focuses on performance optimizations for TPU clusters, and conceptually can be viewed as a special case for GShard. Zero [83] presents a set of optimizations to reduce memory redundancy in parallel training devices, by partitioning weights, activations, and optimizer state separately, and it is able to scale models to 170 billion parameters; in comparison, GShard is more general in the sense that it does not distinguish these tensors, and all of those specific partitioning techniques can be supported by simply annotating the corresponding tensors, allowing us to scale to over 1 trillion parameters and explore more design choices.

Conditional Computation and Machine Translation Conditional computation [25, 16, 26, 27] premises that the examples should be routed within the network by activating an input dependent sub-

network. The routing depends (or conditions) on certain criterion and without the loss of generality, can be any of the following: estimated difficulty of the example [84], available computation budget [26, 27], or more generally a learned criterion with sparsity induced mixture of experts [16]. We extend sparsely gated mixture of experts [16] due to its flexibility and ease of scaling to state of the art neural sequence models, Transformers [10], to satisfy training efficiency.

7 Conclusion

In this paper, we introduced GShard, a deep learning module that partitions computation at scale automatically. GShard operates with lightweight sharding annotations required in the user model code only and delivers an easy to use and flexible API for scaling giant neural networks. We applied GShard to scale up Transformer architecture with Sparsely-Gated Mixture-of-Experts layers (MoE Transformer) and demonstrated a 600B parameter multilingual neural machine translation model can efficiently be trained in 4 days achieving superior performance and quality compared to prior art when translating 100 languages to English with a single model. In addition to the far better translation quality, MoE Transformer models trained with GShard also excel at *training efficiency*, with a training cost of 22 TPU v3 core years compared to 29 TPU years used for training all 100 bilingual Transformer baseline models. Empirical results presented in this paper confirmed that scaling models by utilizing conditional computation not only improve the quality of real-world machine learning applications but also remained practical and sample efficient during training. Our proposed method presents a favorable scalability/cost trade-off and alleviates the need for model-specific frameworks or tools for scaling giant neural networks. Together, our results help to elucidate a realistic and practical way forward for neural network scaling to achieve better model quality.

We have learned several lessons from our study. Our results suggest that progressive scaling of neural networks yields consistent quality gains, validating that the quality improvements have not yet plateaued as we scale up our models. While the results in this paper consolidate that model scaling is a must in deep learning practitioners' toolbox, we also urge practitioners to strive for training efficiency. To this end, we identified factors that affect the training efficiency and showed their implications on downstream task quality. We demonstrated how the neural networks built with conditional computation yield a favorable trade-off between scale and computational cost. In practice such critical design decisions allowed us to enjoy experimental cycles of not months or weeks, but only days to train models in the order of magnitude of trillion parameters.

Further, having a proper abstraction layer that separates model description from parallelization implementation, allows model developer to focus on network implementation, leaving GShard to partition the computation graphs automatically and generate programs that run on all devices in parallel. We found that generating a single program that is general enough to express computation on all underlying parallel devices is the key to compile scalably. The traditional way of generating multiple dedicated programs for different partitions results in explosive compilation time when scaling to thousands of partitions. To address this complexity, we introduced various compiler renovations based on SPMD sharding that allows any tensor dimension to be partitioned. As a takeaway, we emphasize that model scaling and training efficiency should go hand-in-hand; and algorithmic improvements such as conditional computation when coupled with easy to use interfaces can effectively utilize large computational power.

Lastly, our experimental results empirically support that, mere parameter counting does not always correlate with the effective capacity of the models at scale [85, 86]. Comparison of the models should also account in the nature of the problem, i.e. massively multi-task setting with a heavy training data imbalance across tasks as in our case, and control the factors affecting different operation modes of the networks, i.e. capacity bottleneck vs positive transfer.

Acknowledgements

We would like to thank the Google Brain and Translate teams for their useful input and insightful discussions, entire XLA and Lingvo development teams for their foundational contributions to this project. In particular Youlong Cheng, Naveen Arivazhagan, Ankur Bapna, Ruoming Pang, Yonghui Wu, Yuan Cao, David Majnemer, James Molloy, Peter Hawkins, Blake Hechtman, Mark Heffernan,

Dimitris Vardoulakis, Tamas Berghammer, Marco Cornero, Cong Liu, Tong Shen, Hongjun Choi, Jianwei Xie, Sneha Kudugunta, and Macduff Hughes.

References

[1] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. *arXiv preprint arXiv:1802.06509*, 2018.

[2] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[3] 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.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 181–196, 2018.

[6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[9] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7036–7045, 2019.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[11] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.

[12] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[13] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale, 2019.

[14] Naveen Arivazhagan, Ankur Bapna, Orhan Firat, Dmitry Lepikhin, Melvin Johnson, Maxim Krikun, Mia Xu Chen, Yuan Cao, George Foster, Colin Cherry, Wolfgang Macherey, Zhifeng Chen, and Yonghui Wu. Massively multilingual neural machine translation in the wild: Findings and challenges, 2019.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems 32*, pages 103–112, 2019.

[16] 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. *arXiv preprint arXiv:1701.06538*, 2017.

[17] Madhu S. Advani and Andrew M. Saxe. High-dimensional dynamics of generalization error in neural networks, 2017.

[18] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically, 2017.

[19] Joel Hestness, Newsha Ardalani, and Gregory Diamos. Beyond human-level accuracy. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, Feb 2019.

[20] Mario Geiger, Arthur Jacot, Stefano Spigler, Franck Gabriel, Levent Sagun, Stéphane d' Ascoli, Giulio Biroli, Clément Hongler, and Matthieu Wyart. Scaling description of generalization with number of parameters in deep learning. *Journal of Statistical Mechanics: Theory and Experiment*, 2020(2):023401, Feb 2020.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[22] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[23] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[24] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[25] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models, 2015.

[26] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive transformer. *ArXiv*, abs/1910.10073, 2020.

[27] Ankur Bapna, Naveen Arivazhagan, and Orhan Firat. Controlling computation versus quality for neural sequence models, 2020.

[28] XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla, 2019. Online; accessed 1 June 2020.

[29] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[30] Albert Einstein. Die grundlage der allgemeinen relativitätstheorie. In *Das Relativitätsprinzip*, pages 81–124. Springer, 1923.

[31] Jonathan Shen, Patrick Nguyen, Yonghui Wu, Zhifeng Chen, Mia Xu Chen, Ye Jia, Anjuli Kannan, Tara Sainath, Yuan Cao, Chung-Cheng Chiu, et al. Lingvo: a modular and scalable framework for sequence-to-sequence modeling. *arXiv preprint arXiv:1902.08295*, 2019.

[32] Youlong Cheng, HyoukJoong Lee, and Tamas Berghammer. Train ML models on large images and 3D volumes with spatial partitioning on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/train-ml-models-on-large-images-and-3d-volumes-with-spatial-partitioning-on-cloud-tpus, 2019. Online; accessed 12 June 2020.

[33] ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2019. Online; accessed 1 June 2020.

[34] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new ir for machine learning frameworks. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages - MAPL 2018*, 2018.

[35] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph lowering compiler techniques for neural networks, 2018.

[36] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: http://www.mpi-forum.org (Dec. 2009).

[37] Minsik Cho, Ulrich Finkler, and David Kung. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, Palo Alto, CA, 2019.

[38] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, USA, 1969. AAI7010025.

[39] Orhan Firat, Kyunghyun Cho, and Yoshua Bengio. Multi-way, multilingual neural machine translation with a shared attention mechanism. *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016.

[40] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, and et al. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics*, 5:339–351, Dec 2017.

[41] Roee Aharoni, Melvin Johnson, and Orhan Firat. Massively multilingual neural machine translation. *CoRR*, abs/1903.00089, 2019.

[42] Exploring massively multilingual, massive neural machine translation. https://ai.googleblog.com/2019/10/exploring-massively-multilingual.html. Accessed: 2020-06-05.

[43] Recent advances in google translate. https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html. Accessed: 2020-06-05.

[44] Timothy T Baldwin and J Kevin Ford. Transfer of training: A review and directions for future research. *Personnel psychology*, 41(1):63–105, 1988.

[45] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[46] Andrew Davis and Itamar Arel. Low-rank approximations for conditional feedforward computation in deep neural networks, 2013.

[47] Jakob Uszkoreit, Jay M. Ponte, Ashok C. Popat, and Moshe Dubiner. Large scale parallel document mining for machine translation. In *Proceedings of the 23rd International Conference on Computational Linguistics*, COLING '10, page 1101–1109, USA, 2010. Association for Computational Linguistics.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[49] Ankur Bapna, Mia Chen, Orhan Firat, Yuan Cao, and Yonghui Wu. Training deeper neural machine translation models with transparent attention. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

[50] Kazuki Irie, Albert Zeyer, Ralf Schlüter, and Hermann Ney. Language modeling with deep transformers. *Interspeech 2019*, Sep 2019.

[51] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[52] David R. So, Chen Liang, and Quoc V. Le. The evolved transformer, 2019.

[53] Using bfloat16 with TensorFlow models. https://cloud.google.com/tpu/docs/bfloat16, 2020. Online; accessed 12 June 2020.

[54] Heng-Tze Cheng, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, Hemal Shah, Levent Koc, Jeremiah Harmsen, and et al. Wide and deep learning for recommender systems. *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems - DLRS 2016*, 2016.

[55] Andrew K. Lampinen and Surya Ganguli. An analytic theory of generalization dynamics and transfer learning in deep linear networks, 2018.

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

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[58] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[59] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[60] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[61] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[62] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[63] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964. IEEE, 2016.

[64] Chung-Cheng Chiu, Tara N Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J Weiss, Kanishka Rao, Ekaterina Gonina, et al. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778. IEEE, 2018.

[65] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[66] Jonathan Shen, Ruoming Pang, Ron J Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, Rj Skerrv-Ryan, et al. Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4779–4783. IEEE, 2018.

[67] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. 2017.

[68] Behnam Neyshabur, Srinadh Bhojanapalli, David McAllester, and Nathan Srebro. Exploring generalization in deep learning, 2017.

[69] Paolo Ienne, Thierry Cornu, and Gary Kuhn. Special-purpose digital hardware for neural networks: An architectural survey. *Journal of VLSI signal processing systems for signal, image and video technology*, 13(1):5–25, 1996.

[70] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[71] Dan Claudiu Cires¸an, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[72] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[73] 2019 recent trends in GPU price per FLOPS. https://aiimpacts.org/2019-recent-trends-in-gpu-price-per-flops/. Accessed: 2020-06-05.

[74] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Summarizing cpu and gpu design trends with product data. *arXiv preprint arXiv:1911.11313*, 2019.

[75] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[76] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[77] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[78] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[79] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs. 2018.

[80] Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *Machine Learning and Systems (MLSys)*, 2018.

[81] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, Palo Alto, CA, 2019.

[82] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training, 2020.

[83] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

[84] Loren Lugosch, Derek Nowrouzezahrai, and Brett H. Meyer. Surprisal-triggered conditional computation with neural networks, 2020.

[85] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes, 2018.

[86] Wesley J. Maddox, Gregory Benton, and Andrew Gordon Wilson. Rethinking parameter counting in deep models: Effective dimensionality revisited, 2020.

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

[88] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. *ArXiv*, abs/1804.04235, 2018.

[89] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP*, 2018.

A Appendix

A.1 Decoding with Flat Beam Search

During decoding, we use beam search with length normalization similar to [61]. Decoding is auto-regressive and generates the target sequence one token at a time, so for an output of length m the decoder layer stack is executed m times, sequentially. In particular for each decoder MoE layer there are dispatch/combine operations, which require cross-device communication. Inference utilizes same cluster with same number of devices as training.

During beam search we flatten the beam hypotheses into a single sequence which contains all underlying tokens interleaved, and we modify decoder self-attention mask so that each hypothesis only has attention to appropriate positions in the joint flat sequence. We apply the same transformation to key/value tensors maintained by each decoder self-attention layer. This allows us to avoid reordering previously computed attention key/values after each beam expansion. Instead, we only reorder the $0/1$ mask representing the current active hypotheses. However, attention becomes k times longer.

This trade-off can be positive or negative depending on implementation details. As explained in [87], memory bandwidth limits are important for incremental decoding with Transformer models. From this point of view, by flattening the beam we replace two operations with low compute/memory ratio (attention dot product and key/value reordering) with a single operation with a slightly higher compute/memory ratio (attention dot product over a longer sequence with more keys), but with the same total amount of memory it has to access.

A.2 Machine Translation Experiments Details

In our Machine Translation experiments MoE Transformer models shared *a)* 1024 Transformer model dimension *b)* 8192 Feed Forward and MoE hidden dimension; *c)* 16 heads in multi-head attention; *d)* 128 attention key and value dimension; and *e)* 0.1 input, residual and attention dropout rate.

We used the Adafactor [88] optimizer with *a)* factored second-moment estimation; *b)* first moment decay $\beta_1 = 0.0$; *c)* second moment decay $\beta_2 = 0.99$ with $1 - t^{-0.8}$ schedule; *d)* update clipping threshold of 1.0; and *e)* 1.0 learning rate with square root decay after 10k training steps.

We used SentencePiece [89] subword tokenizer with a single multilingual vocabulary for source-side spanning 102 languages of size 64000, and English-only target-side vocabulary of size 32000.

A.3 General Sharding API

In addition to the two common APIs (replicate() and split()) for sharding listed in Section 3.2, users or the compiler may use a more advanced sharding strategy to minimize data transfers.

shard(tensor, device_assignment) annotates tensor to be partitioned with the provided device assignment, and returns the annotated tensor. We use *device assignment*, a multi-dimensional integer array, to represent how the split is done. device_assignment has the same rank as the data tensor; its element count is the total number of partitions, and each element is the ID of the device that occupies the corresponding data slice. For example, a 3D tensor with shape [3, 16, 64] with device assignment shape [1, 2, 4] will have partition shape [3, 8, 16], and the order of elements in *device assignment* determines which slice each partition occupies.

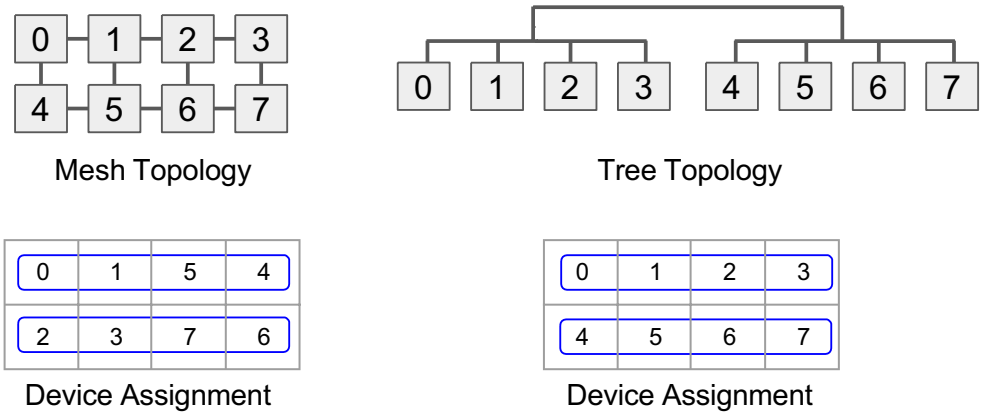


Figure 10: An example of two different *device assignments* based on the device topology. A 2D tensor is split by 2x4 partitions and the communication pattern is between partitions along the rows of the tensor. The numbers represent device ids.

Since data movement across devices critically affects the parallel execution performance, it is important to consider the target device topology as well as the communication between partitions of the tensor when assigning device ids in the *device assignment* for maximum performance. Figure 10 shows two different *device assignments* based on the device topology and the row-wise communication pattern on the tensor.

A.4 SPMD Partitioning for Convolution and Window-Based Operators

GShard is able to partition spatial dimensions in convolutions, and general enough to support use cases like giant images [32]. To spatially shard a convolutional layer, we can use the sharding API in the following way.

```
  # Partition input images [N,C,H,W] along W spatial dimension
inputs = split(inputs, 3, D)
  # Replicate the kernel
kernel = replicate(kernel)
  conv = conv2d(inputs, kernel)
  ...
```

GShard will then propagate the sharding on the spatial dimension to other layers and the backward pass. The rest of section discusses the specific complexity to partition Convolution and similar operators. There are several window-based operations (e.g., Convolution, ReduceWindow), and they all require some type of halo exchange since data may be shared between windows. We use the CollectivePermute operator to exchange halo data between partitions, but one complication is that the halo size may be different across partitions whereas CollectivePermute needs to be statically shaped.

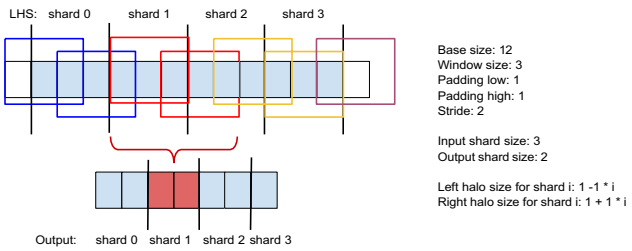


Figure 11: Convolution with non-constant halo size.

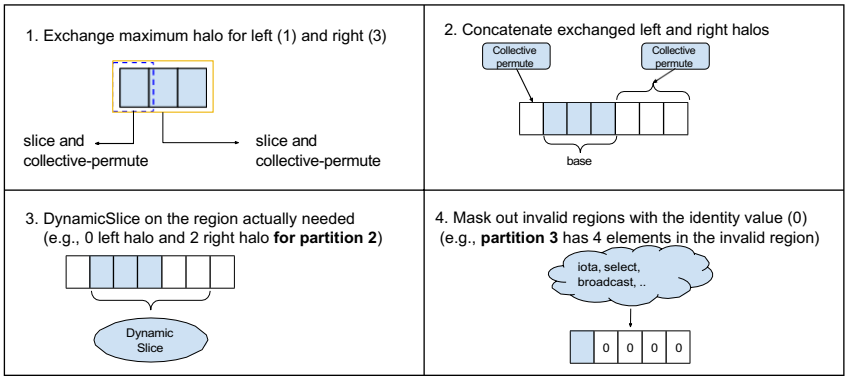


Figure 12: Sequence of operations for a general halo exchange.

We first introduce the window configurations that the SPMD partitioner has to consider. Each spatial dimension in the convolution has the following set of configurations.

- **Stride** is the distance (in number of elements) that the window moves to produce the next output element.
- **Low/high padding** is the number of elements padded to the low/high end of the dimension in LHS (base).
- **Base dilation** is the dilation factor of the LHS, i.e., one plus the number of elements padded between every element (excluding low/high padding). No base dilation means the value is set to 1.
- **Window dilation** is one plus the number of elements padded between every element in the RHS (window).

Non-constant halo size. We demonstrate that non-constant halo size is common using a simple example, which does not have dilation. Figure 11 shows a 4-way partitioned convolution, where the right halo sizes for the partitions are (1, 2, 3, 4) and can be expressed as a linear function of the partition ID: *partition_id* + 1. Partition 1 is in charge of generating 2 output elements (red cells), which means that the partition needs to get 0 elements from Partition 0, and 2 elements from Partition 2 (area covered by two dotted red windows).

Figure 12 describes the sequence of operations for a general halo exchange. First, we calculate the maximum size of left and right halo across partitions and perform the halo exchange of the maximum size (Steps 1 and 2). Since some partitions may have excessive halos than needed, we use DynamicSlice (based on the partition ID) to slice off the valid region for the current partition (Step 3). Finally, some partitions may include garbage values (e.g., halos from out-of-range input data), so we apply masking as described in Section 3.3.3.

Base dilation. Base dilation adds additional complexities to halo exchange, since the offset of each partition may be positioned at the dilation holes, and also low/high padding is applied after dilation, which makes the edges have different behavior than the interior elements. We handle base dilation in 3 cases (Figure 13).

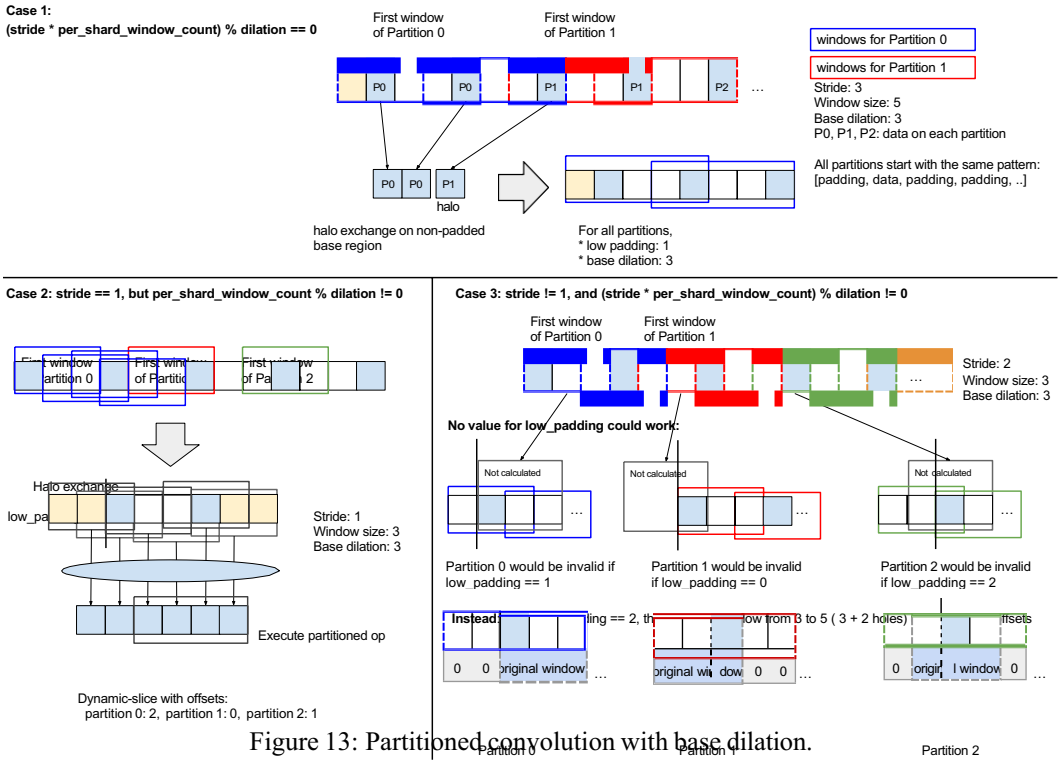


Figure 13: Partitioned convolution with base dilation.

. *stride* $\times$ *per_shard_window_count* is divisible by *dilation*, where *per_shard_window_count* is the number of windows to be processed by each partition (i.e., the number of output elements for each partition). This condition guarantees that all partitions start with the same number of (interior or low) padding elements before the first data element in the LHS, so that we can use the same low padding. Halo exchange occurs on the non-dilated/non-padded base region, and the limit index of required data for Partition *i* can be calculated as below.

$$\frac{stride \times per\_shard\_window\_count \times i + window\_size - low\_pad + dilation - 1}{dilation},$$

which determines the right halo size. Because $stride \times per\_shard\_window\_count$ is divisible by *dilation*, it can be simplified as $a \times i + b$, where a and b are both constants.

. *stride* == 1 but *per_shard_window_count* is not divisible by *dilation*. In this case, the low padding on different partitions are different, but it is a static configuration in windowed operations, which can't be specialized for each partition for SPMD execution. Using Pad and DynamicSlice on the operand also would not work, because those operators would be applied before dilation, so everything would be multiplied by the dilation factor. Fortunately, with *stride* == 1, all positions on the padded and dilated base region are valid window starts, and we can use the maximum low padding on all partitions to ensure that each partition calculates all required windows, then do a DynamicSlice on the output of the partitioned windowed operator to remove unnecessary data. The limit index of required data on the non-padded base region for Partition *i* is same as before,

$$\frac{per\_shard\_window\_count \times i + window\_size - low\_pad + dilation - 1}{dilation},$$

but cannot be simplified to $a \times i + b$.

. *stride* $\neq$ 1 and *stride* $\times$ *per_shard_window_count* is not divisible by *dilation*. If neither of the above conditions are true, different partitions could start with different number of padding elements, and not all offsets are valid window starts. Consider the last example in

Figure 13. Whatever low padding we chose, some partition will be invalid, because the valid windows could be skipped since $stride = 1$. A solution to this problem is to pad the window in addition to padding the base area. We can use the maximum low padding required by the partitions on the base area, then increase the window size by that low padding amount. However, the low and high padding amounts on the window vary on different partitions, which can be implemented by a Pad followed by a DynamicSlice. The window padding is used to mask off the unaligned elements in the base area, so that the start of the non-padding window element will be aligned with the desired start in the base area for each partition.

Window dilation. If the RHS is replicated, window dilation only affects the effective window size when partitioning the operator based on its LHS. If the dilated RHS is also partitioned, which typically occurs in the gradient computation of strided convolutions, handling window dilation is still simpler than handling base dilation, because there is no low/high padding on the RHS. We skip the details of the implementation.