GShard：通过条件计算和自动分片扩展巨型模型

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

马克西姆
·Krikunkrikun@google

Noam Shazeer
noam@google.com

Zhifeng Chen
zhifengc@google.com

Abstract

在许多具有大量训练数据和计算的现实机器学习应用中，神经网络扩展对于提高模型质量至关重要。尽管这种扩展趋势被认为是提高模型质量的可靠方法，但该方法仍面临一些挑战，例如计算成本、编程的简易性以及并行设备上的高效实现。 GShardis 是一个由一组轻量级注释 API 和 XLA 编译器扩展组成的模块。它提供了一种优雅的方式来表达各种并行计算模式，同时对现有模型代码进行最小的更改。 GShard 使我们能够使用自动分片将具有稀疏门控混合专家的多语言神经机器翻译 Transformer 模型扩展到超过 6000 亿个参数。我们证明，这样一个巨大的模型可以在 4 天内在 2048 个 TPU v3 加速器上进行有效训练，从而实现从 100 种语言到英语的翻译质量，与现有技术相比要高得多。

1 简介

扩展神经网络可以为各种机器学习问题带来显着的质量提升[1,2,3,4,5,6]。对于计算机视觉，增加模型容量可以提高各种计算机视觉架构的图像分类和检测精度[7,8,9]。类似地，在自然语言处理中，缩放 Transformers [10] 在语言理解任务 [4, 11, 12]、跨语言下游传输 [4, 13] 和（大规模）多语言神经机器翻译 [14, 15, 16] 方面取得了一致的成果]。这种总体趋势促使最近的研究仔细审视在扩展成功中发挥关键作用的因素[17,18,19,20,3]，包括过去研究发现的训练数据量、模型大小和所使用的计算。虽然最终的模型质量被发现与数据量、计算量和模型大小存在幂律关系 [18, 3]，但较大模型带来的显着质量增益也带来了各种实际挑战。最重要的训练效率（我们将其定义为相对于现有最佳系统实现卓越模型质量所使用的计算量和训练时间）常常被忽略。

预印本。正在审核中。

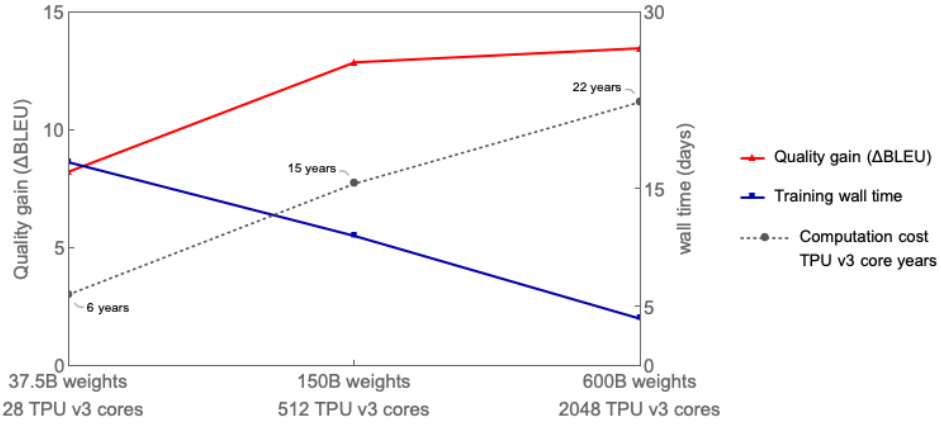


图 1：随着 MoE 模型大小增长到 600B，多语言翻译质量（与双语基线相比的平均 Delta BLEU）得到改善，而端到端训练成本（以 TPU v3core-year 计算）仅呈次线性增加。将模型大小从 37.5B 增加到 600B (16x)，导致计算成本从 6 年增加到 22 年 (3.6x)。达到最佳翻译质量的 600B 参数模型使用 2048 个 TPU v3 核心训练了 4 天，总成本为 22 个 TPU v3 核心年。相比之下，训练所有 100 个双语基线模型将需要 29 个 TPU v3core 年。我们的最佳质量密集单 Transformer 模型（2.3B 参数）达到 Delta BLEUof 6.1，使用 GPipe [15] 在 2048 个 TPU v3 核心上进行了 6 周或总共 235.5 个 TPU v3 核心年的训练。

1.1 扩展的实际挑战

在这里，我们列举了所面临的主要实际挑战，特别是在训练比单个加速器内存（例如 GPU 或 TPU）的容量限制大几个数量级的大规模模型时。

特定于架构的模型并行支持 在常用的深度学习框架（例如 TensorFlow [21] 和 PyTorch [22]）下，缺乏对高效模型并行算法的支持。支持具有图分区的朴素模型并行性，但由于网络的顺序依赖性和基于梯度的优化，这会导致严重的利用率不足。为了有效地扩展现有模型，用户通常需要投入大量的工程工作，例如例如，将模型代码迁移到特殊框架[23, 15]。

计算成本与模型大小的超线性缩放 通过增加深度或宽度 [6, 15] 直接缩放模式大小通常会导致训练步骤时间至少线性增加。通过在多个设备上分割层权重和计算来进行模型并行通常变得必要，从而导致网络通信开销和设备利用率不足。设备利用率不足源于底层神经网络的不平衡分配和顺序依赖性。计算成本和模型大小之间的这种超线性关系无法通过简单地使用更多设备来解决，从而使得训练大规模模型变得不切实际。

巨型模型表示的基础设施可扩展性分布在数千个设备上的大规模模型的简单图形表示可能会成为深度学习框架及其优化编译器的瓶颈。例如，在操作间分区中添加 D 倍的层或通过跨 D 个设备的操作内分区增加模型维度可能会产生具有 O(D) 个节点的图。设备之间的通信通道可以进一步将图大小增加最多 O(D 2)（例如，分区收集或转置）。图大小的这种增加将导致大规模模型的图构建和编译时间变得不可行。

实现分区策略的艰巨工作对模型进行分区以在许多设备上有效运行具有挑战性，因为它需要协调跨设备的通信。对于图级分区，需要复杂的算法 [15, 24] 来减少开销

由分配在不同设备上的图的不同分区之间的顺序依赖关系引入。对于算子级并行性，不同分区的算子有不同的通信模式，具体取决于语义，例如是否需要累积部分结果，或者重新排列数据分片。根据我们的经验，考虑到像 TensorFlow 这样的框架拥有大量具有临时语义的运算符，手动处理模型中的这些问题需要付出大量的努力。在所有情况下，实现模型分区对于从业者来说尤其是一种负担，因为改变模型架构将需要改变底层设备通信，从而导致连锁反应。

1.2 大规模高效培训的设计原则

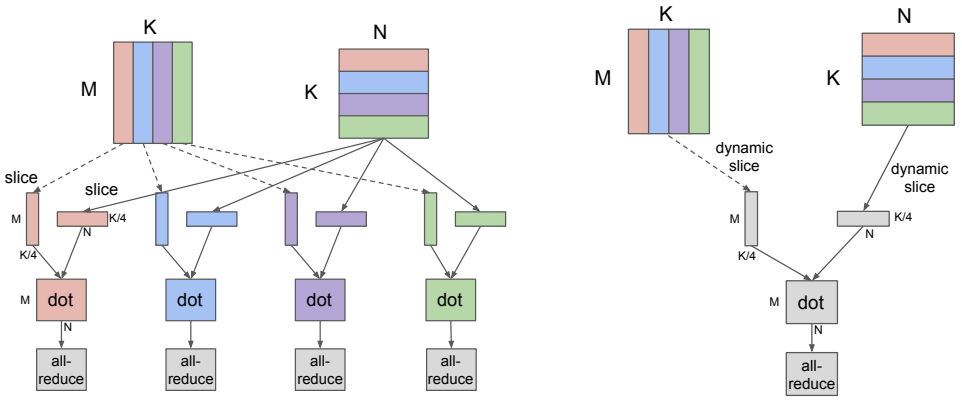
在本文中，我们演示了如何通过构建具有稀疏门控混合专家层的 6000 亿个参数的序列到序列 Transformer 模型来克服这些挑战，该模型具有次线性计算成本和 O(1) 编译时间。我们在多语言机器翻译任务中使用 2048 个 TPUv3 设备对该模型进行了 4 天的训练，在使用单个非集成模型将 100 种语言翻译成英语时，与现有技术相比，取得了远远优于现有技术的翻译质量。我们对各种模型大小进行了实验，发现翻译质量随着模型变大而提高，但训练的总墙时间仅相对于模型大小呈亚线性增加，如图 1 所示。模型中，我们做出了以下关键设计选择。

次线性缩放首先，模型架构的设计应使计算和通信需求在模型容量中保持次线性。条件计算[25,16,26,27]使我们能够通过在每个输入的基础上激活子网络来满足训练和推理效率。通过添加位置稀疏门控混合专家 (MoE) 层 [16] 来扩展基于 RNN 的机器翻译和语言模型的容量，从而能够以次线性计算成本实现最先进的结果。因此，我们在第 2 节中介绍了使用 MoE 层扩展 Transformer 架构的方法。

抽象的力量 其次，模型描述应该与分区实现和优化分开。这种关注点分离让模型开发人员能够专注于网络架构并灵活地更改分区策略，而底层系统则应用语义保留转换并实现高效的并行执行。为此，我们提出了一个模块 GShard，它只需要用户用分区策略注释模型中的几个关键张量。它由一组用于注释的简单 API 和 XLA [28] 中用于自动并行化的编译器扩展组成。模型开发人员编写模型就好像有一个具有巨大内存和计算能力的设备一样，编译器根据注释和自己的启发式自动划分目标的计算。我们在 3.2 节中提供了更多注释示例。

可扩展编译器 第三，系统基础设施（包括计算表示和编译）必须可扩展以支持数千个设备的并行执行。例如，图 2 说明了跨 4 个设备（颜色编码）划分点积运算的两种不同方法。请注意，使用图 2a 中常见的 MPMD（多程序多数据）方法，扩展变得更具挑战性，因为网络中的节点数量较多。图表随着设备数量线性增加。相反，我们开发了一种用于 SPMD（单程序多数据）转换的编译器技术，该技术可生成在所有设备上运行的单个程序，从而保持编译时间常数与设备数量无关，如图 2b 所示。我们将在 3.3 节中更详细地讨论我们的 SPMD 框架。

本文的其余部分组织如下。第 2 节更详细地描述了带有稀疏门控 MoE 层的 Transformer 架构。第 3 节介绍了我们的开发模块 GShard。第 4 节演示了我们的专家模型混合在超过 100 种语言对的多语言机器翻译任务上的应用。第 5 节对我们的实现进行了性能和内存测量。第 6 节讨论相关工作。



(a) MPMD Partition (b) SPMD Partition

图 2：MPMD 与我们提出的跨 4 个设备的点运算符 ([M, K] × [K, N] = [M, N]) 的 SPMD 分区之间的比较。在此示例中，两个操作数都沿收缩维度 K 进行分区，其中每个设备计算本地结果并与 AllReduce 全局组合。 MPMD 分区为每个设备生成单独的运算符，限制了其可扩展性，而 SPMD 分区生成一个程序在所有设备上运行。请注意，我们的 SPMD 分区的编译时间与所使用的设备数量无关。

2 型号

2.1 Transformer架构的稀疏扩展

Transformer [10]架构已广泛用于自然语言处理。它已成为许多序列到序列任务（例如机器翻译）的事实上的标准。Transformer 使用两个计算块：编码器和解码器，两者都是通过堆叠多个 Transformer 层来实现的。 Transformer 编码器层由两个连续层组成，即自注意力层和位置前馈层。解码器添加了第三个交叉注意力层，该层关注编码器的输出。我们通过条件计算稀疏地缩放 Transformer，用位置明智混合专家 (MoE) 层 [16] 替换所有其他前馈层，并在编码器和解码器中使用 top-2 门控变体（图 3）。 Transformer 层的数量和每个 MoE 层的专家数量，以扩展模型容量。

每个训练示例由一对子词标记序列组成。每个令牌在训练和推理期间都会激活 MoE Transformer 的子网络。子网络的大小大致独立于每个 MoE 层的专家数量，从而允许计算成本的次线性缩放，如上一节所述。计算复杂度在第 3.1 节中进一步分析，训练性能在第 5 节中进一步分析。

2.2 Position-wise Mixture-of-Experts Layer

我们模型中使用的专家混合（MoE）层基于[16]，但使用的稀疏函数和辅助损失有所不同。 Transformer 的 MoE 层由 Efeed-forward 网络 FFN 1 组成。 。 。 FFN：

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

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

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

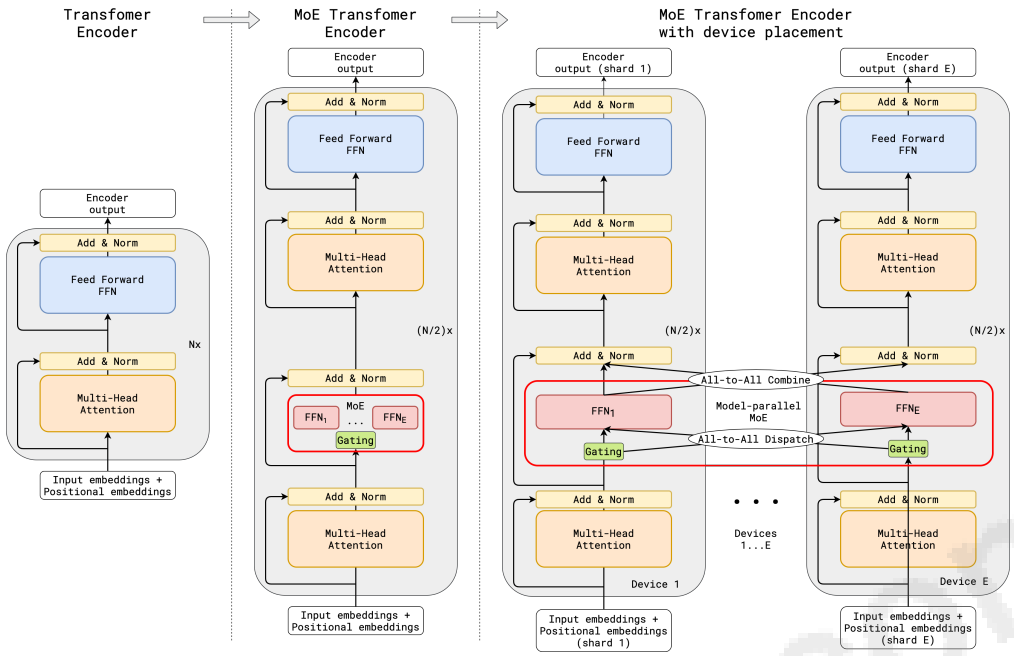


图 3：使用 MoE 层缩放 Transformer 编码器的图示。 MoE 层取代了所有其他 Transformer 前馈层。解码器修改类似。 (a) 标准 Transformer 模型的编码器是一堆自注意力和前馈层，与剩余连接和层归一化交错。 (b) 通过用 MoE 层替换所有其他前馈层，我们得到了 MoE Transformer Encoder 的模型结构。 (c) 当扩展到多个设备时，MoE 层在设备之间进行分片，而所有其他层都被复制。

其中 x s 是 MoE 层的输入标记，wi 和 wo 是前馈层（专家）的输入和输出投影矩阵。矢量 G s,E 由门控网络计算。 G s,E 对于每个专家都有一个非负数，其中大多数为零，这意味着令牌不会分发给该专家。代币被分发给极少数专家。我们选择让每个代币最多分发给两个专家。 G s,E 中的相应条目非零，表示专家对最终网络输出的贡献程度。每个专家 FFN e 都适用于使用 ReLU [29] 激活函数的 x s 全连接 2 层网络。 MoE 层的输出 y s 是所有选定专家输出的加权平均值。

门控函数 GATE(·) 对于 MoE 层至关重要，它由 softmax 激活函数建模，以指示每个专家在处理传入令牌时的权重。换句话说，表明专家处理传入令牌的能力如何。此外，门函数必须满足两个目标：

· 平衡负载 希望MoE 层能够稀疏地激活给定代币的专家。一个简单的解决方案是根据 softmax 概率分布选择前 k 个专家。然而，众所周知，这种方法会导致训练负载不平衡问题[16]：训练期间看到的大多数令牌将被分派给少数专家，为少数（繁忙）专家积累非常大的输入缓冲区，而其他专家则未经训练，减慢训练速度。与此同时，许多其他专家根本没有接受过足够的培训。门函数的更好设计将在所有专家之间更均匀地分配处理负担。

· 大规模效率 如果门控功能按顺序完成，那么实现平衡负载将是相当简单的事情。对于给定 E 个专家的输入批次中的所有 N 个令牌，仅门函数的计算成本至少为 O(NE)。然而，在我们的研究中，N 的数量级为数百万，E 的数量级为数千，门函数的顺序实现将使大部分计算资源在大部分时间处于闲置状态。因此，我们需要门函数的高效并行实现来利用许多设备。

我们在门函数 GATE(·) 中设计了以下机制来满足上述要求（详细信息如算法 1 所示）：

· 专家容量 为了确保负载平衡，我们强制一位专家处理的令牌数量低于某个统一阈值，我们将其定义为专家容量。假设训练批次中的令牌总数为 N ，并且每个令牌最多分派给两个专家，则专家容量设置为 O(N/E)。GATE(·) 保留一个运行计数器 c e 来计算如何许多代币被发送给专家。当令牌选择的两个专家都已经超出其容量时，该令牌被视为溢出令牌，其中 G s，E 退化为零向量。这些令牌的表示 x s 通过剩余连接传递到下一层。本地组调度 GATE(·) 将训练批次中的所有令牌均匀划分为 G 组，即每个组包含 S = N/G 令牌。所有组均独立并行处理。每个组都被赋予每个专家的分数能力，2N/(G·E)。每个组确保最多将这么多代币发送给专家。通过这种方式，我们可以确保专家容量仍然得到执行，并且总体负载是平衡的。 · 辅助损失 重要的是，门函数并不总是选择相同的少数专家，因为这会导致只有少数专家的容量溢出其余的则利用率不足。按照 [16]，我们定义了一个辅助损失项 aux 来强制执行这个约束。它通过常数乘数 k 添加到模型 L = nll + k ∗ aux 的整体损失函数中。算法 1 的第 (13) 行中辅助损失项 aux 的特定形式是出于以下考虑：项 c e /S 表示路由到每个专家的输入的分数，并且我们希望最小化 c e /S 的均方。但由于 c e 是从 top-2 运算导出的并且不可微分，因此我们使用平均门专家 m e 作为可微近似，并将 (c e /S) 2 替换为 m e (ce /S)，现在可以通过梯度下降进行优化。 · 随机路由 直观上，因为 y s 是所选专家返回的加权平均值，如果第二个专家的权重非常小，我们可以简单地忽略第二个专家以节省整体专家容量。因此，除了尊重专家容量约束之外，GATE(·) 还以与其权重 g 2 成比例的概率分派给第二最佳专家。

3 使用GShard的高度并行实现

本节介绍第 2 节中模型在 TPU 设备集群上高效运行的实现。

第一步是用线性代数运算来表达模型，其中我们的软件堆栈（TensorFlow [21]）和硬件平台（TPU）经过高度定制和优化。按照与原始 Transformer 相同的方式，很容易用线性代数来编写大部分模型的代码。然而，它需要一些努力来表达 MoE 层，特别是算法 1 中提出的 GATE(·) 函数，由于其顺序性质，我们在 3.1 节中描述了细节。

接下来，我们注释线性代数计算以表达并行性。计算中的每个张量都可以使用 3.2 节中的 shardingAPI 进行注释，以便在设备集群上复制或分发。使用分片注释可以分离模型描述和高效并行实现之间的关注点，并允许用户灵活地表达不同的并行化策略。例如，（1）注意力层通过沿着批量维度分割并将其权重复制到所有设备来并行化。另一方面，(2) MoElayer 中的专家由于其庞大的规模而无法在所有设备中复制，唯一可行的策略是将专家分片到许多设备中。此外，整个模型在这两种模式（1）-（2）之间交替。使用注释可以将模型开发人员从系统优化工作中解放出来，并避免将并行实现和低级细节烘焙到模型代码中。

最后，编译器基础设施采用（部分）带注释的线性代数计算，并生成可扩展到数千台设备的高效并行程序。正如3.3节中将描述的，编译器应用SPMD（单程序多数据）分区变换来表达每设备计算，插入必要的跨设备通信，处理不规则的情况

算法1：具有辅助损失的组级top-2门控数据：x S，一组大小为S的令牌数据：C，分配给该组的专家容量结果：G S，E，组组合权重结果：aux，组辅助损失

（1） c E ← 0 每个专家的门控决策(2) g S,E ← softmax(wg · x S)每个专家每个令牌的门，wg 是可训练的权重(3) m E ← 1 ▷ gating decisions per expert

en per expert, wg are trainable weights

s=1 g s,Emean 门每个专家(4) ▷ mean gates per expert

for s ← 1 到 S do

(5) g1, e1, g2, e2 = top_2(gs,E)top-2 门和专家索引 ▷ top-2 gates and expert indices

(6) $g1 \quad g1/(g1 + g2)$ ▷ normalized $g1$

(7) c ← c e1 专家缓冲区中的 e1 位置 ▷ position in $e1$ expert buffer

(8) **if** $c_{e1} < C$ **then**

(9) G s,e1 ← g1e1 x s 的专家组合重量 ▷ $e1$ expert combine weight for x_s

(10) **en**

c e1 ⇐ c + 1递增 e1 专家决策 ▷ incrementing $e1$ expert decisions count

count(12) end(13) aux = 1

(14) **for** $s \quad 1$ **to** S **do**

(15) g1, e1, g2, e2 = top_2(gs,E)top-2 门和专家索引 ▷ top-2 gates and expert indices

(16) $g2 \leftarrow g2/(g1 + g2)$ ▷ normalized $g2$

(17) $rnd \quad uniform(0, 1)$ ▷ dispatch to second-best expert with probability $\propto 2 \cdot g2$

(18) c ← c e2 专家缓冲区中的 e2 位置 ▷ position in $e2$ expert buffer

(19) **if** $c < C \quad 2 \quad g2 > rnd$ **then**

(20) G s,e2 ← g2e2 x s 的专家组合权重 ▷ $e2$ expert combine weight for x_s

(21) **end**

(22) $c_{e2} \leftarrow c + 1$

(23) **end**

诸如不均匀分区之类的模式，并最终生成一个可以在所有设备上启动以并行执行的程序。

3.1 以线性代数表示的位置明智混合专家层

我们的模型实现（算法 2）将整个加速器集群视为单个设备，并通过与集群的具体设置无关的几个张量运算来表达其核心数学算法。爱因斯坦求和符号 [30]（即 tf.einsum）是一个强大的构造，可以简洁地表达模型，我们在实现中广泛使用它。 softmax 门计算简单地由一个 einsum 和后面的 softmax 函数来表示。将输入分派给选定的专家由分派掩码和输入之间的单个 einsum 表示。所有 FFN e 权重被组合成单个 3-D 张量 wi amd wo 并由 FFN 1 进行计算。 。 。 FFN E 使用 3 个运算符表示（两个 einsum 和一个 relu）。最后，将所有专家输出的加权平均值代入最终输出，并用另一个 einsum 表示。

算法 2 中的 Top2Gating 计算算法 1 中描述的所有组局部 G S,E 的并集。combine_weights 是形状为 [G, S, E, C] 的 4-D 张量。当组 g 中的输入标记 s 被发送到缓冲区位置 c 处的专家 e 的输入缓冲区时，combine_weights[g,s, e, c] 值非零。对于特定的g和s，切片 combine_weight[g, s, :, :] 最多包含两个非零值。二进制dispatch_mask是通过将所有非零值简单地设置为1而从combine_weights生成的。

我们需要适当选择组数 G 和专家数 E，以便算法可以扩展到具有 D 台设备的集群。在给定 N 个标记的训练批次的情况下，分析训练步骤的整体计算复杂性（浮点运算的总数）是值得的。

算法 2：Positions-wise MoE 层的前向传递。带下划线的字母（例如 G 和 E）表示张量划分的维度。

1 门 = softmax (einsum ("GSM , ME ->GSE " , 输入 , wg))2combine_weights ,dispatch_mask = Top2Gating(gates)3 d i sp a t ch e d _ex per t _ in put ts = einsum (4"GSEC ,GSM->EGCM",dispatch_mask,reshape_inputs)5 h=einsum("EGCM,EMH->EGCH",dispatched_expert_inputs,wi)6h=relu(h)7expert_outputs=ei >GECM", h , wo)8 个输出 = einsum (9"GSEC ,GECM - >GSM " , 组合权重 , 专家输出)

我们分析算法 2 计算复杂度随设备数量 D 的变化，并做出以下假设： a) 每个设备的令牌数量 ND = O(1) 为常数 1； b) G = O(D),S = O(1) 且 N = O(GS) = O(D)；c) M = O(1)，H = O(1)； d) E = O(D)； e)C = O(2SE) = O(1D), D < S 且为正整数 2 。算法 2 中浮点运算总数 F LOP S：

F LOP S Softmax +F LOP S Top2Gating +F LOP S 调度|组合 +F LOP S FFN =O(GSM E)+O(GSEC)+O(GSM EC)+O(EGCHM) =

$$O(D \cdot 1 \cdot 1 \cdot D) + O(D \cdot 1 \cdot D \cdot \frac{1}{D}) + O(D \cdot 1 \cdot 1 \cdot D \cdot \frac{1}{D}) + O(D \cdot D \cdot \frac{1}{D} \cdot 1 \cdot 1) =$$

$$O(D^2) \qquad + O(D) \qquad + O(D) \qquad + O(D)$$

因此每设备 F LOP S/D = O(D) + O(1) + O(1) + O(1)。每个设备的 softmax 复杂度 F LOP S softmax /D = O(D) 与设备数量呈线性关系，但实际上受其他项支配，因为 D << H 且 D < S。因此，可以考虑 F LOP S/D O(1)，满足次线性缩放设计要求。第五节通过实证验证了这一分析。

除了计算成本之外，我们还有非恒定的跨设备通信成本，但它以适度的速度增长 O(D)（当我们增加 D 时（第 5.4 节）。

3.2 并行执行的GShard注解API

由于算法 1 中张量的巨大尺寸和计算需求，我们必须在许多设备上并行化该算法。算法 2 中的下划线字母说明了如何对算法中的每个张量进行分片的直接解决方案。GShard 中的分片 API 允许我们在程序中注释张量，以有选择地指定应如何对它们进行分区。该信息被传播到编译器，以便编译器可以自动应用并行执行的转换。我们在工作中使用 TensorFlow/Lingvo [31] 中的以下 API。

· replicate(tensor) 注释要跨分区复制的张量，并返回带注释的张量。这通常用于我们模型中的非 MoE 层来复制权重。

· split(tensor, split_dimension, num_partitions) 注释要沿split_dimension 分区的张量，并返回注释后的张量。分区 i 放置在第 i 个设备上，num_partitions 不得超过系统上的设备数量。

· shard(tensor, device_assignment) 概括了 split()，以允许对多个维度进行分区并指定每个分区的位置。附录 A.3 更详细地描述了该 API。

1 在实践中，这通常是必要的，以避免设备内存溢出。2 缩放 D > S 将需要对部分专家容量进行不同的使用。

请注意，对 split 或 shard 的调用仅添加注释，不会更改用户程序中的逻辑形状。用户仍然可以使用完整的形状，无需担心分区不均匀等问题。

GShard 是通用的，因为简单的 API 以相同的方式适用于所有维度。分片维度可以包括批处理（数据并行性）、特征、专家，甚至图像模型中的空间维度，具体取决于用例。此外，由于分片注释是相关张量，因此模型的不同部分可以以不同的方式进行划分。这种灵活性使我们能够划分巨大的 MoE 权重并在 MoE 和非 MoE 层之间切换划分模式，以及本文之外的用例，例如大图像的空间划分 [32]（附录 A.4）。

通过上述分片API，我们可以将算法2所示的分片策略表达如下。输入张量沿第一维度进行分割，并复制门控权重张量。在计算调度的专家输入后，我们应用 split 将分片从组 (G) 维度更改为专家 (E) 维度。 D 是设备数量。

1# 沿组 (G) 进行分区输入。

2 + 输入 = split (输入 , 0 , D)3# 复制门控权重

4 + wg = 复制 (wg)5gates = softmax (einsum (" GSM , ME -> GSE " , 输入 , wg))6combine_weights ,dispatch_mask = Top2Gating (gating_logits)7d 是 p a tch e d _ex p er t _ in p u ts = einsum (8" GSEC , GSM -> EGCM " ,dispatch_mask , reshape_inputs)9# 沿 Expert (E) dim 分区调度输入。

+ 调度 _ex p er t _ 输入 = split (dispatched_expert_inputs, 0, D)h = einsum(" EGCM, EMH -> EGCH",dispatched_expert_inputs, wi)...

每个张量的分片分配如上例所示，用户不需要注释程序中的每个张量。通常只需要在模型中的几个重要运算符（例如 Einsum）上进行注释，并且编译器使用自己的启发式方法来推断其余张量的分片 3 。例如，由于输入张量沿 G 进行分区并且权重张量进行复制，因此编译器选择沿相同的 G 维度对 einsum 输出进行分区（第 5 行）。类似地，由于输入调度 einsum（第 7 行）的两个输入都沿 G 维度进行分区，因此推断输出分片将沿 G 维度进行拆分，然后我们在输出上添加 splitannotation 以沿 E 维度重新分片。上面示例中的一些注释也可以由编译器确定（例如，replicate(wg)），但建议注释计算的初始输入和最终输出张量。

编译器当前使用迭代数据流分析，从用户注释的运算符开始，将分片信息从运算符传播到其邻居（操作数和用户）。该分析试图通过调整相邻运营商的分片决策来最大程度地减少重新分片的机会。可能还有其他方法，例如整数编程或机器学习方法，但改进自动分片分配不是本文的重点，我们将其留作未来的工作。

混合手动和自动分片 对于常见情况，带有分片注释的自动分区通常就足够了，但 GShard 还可以灵活地允许将手动分区运算符与自动分区运算符混合。这为用户提供了对运算符如何分区的更多控制，一个例子是用户拥有更多超出运算符语义的运行时知识。例如，XLA 和 TensorFlow 的 Gather 运算符定义都不会传递有关输入中不同范围的索引边界的信息，但用户可能知道特定的 Gather 运算符仅在每个分区内对数据进行混洗。在这种情况下，用户可以通过简单地缩小维度大小并执行局部收集来简单地划分算子；否则，编译器需要对索引范围保持保守，并增加不必要的通信开销。例如，算法 2 中的调度 Einsum（第 3 行）

3 对于编译器来说，推断缺失的分片也很重要，因为反向传播计算通常由前端框架自动生成，而用户无法访问这些张量。

算法 2 中使用单热矩阵来分派输入，也可以使用 Gather 运算符使用简单的手动分区来实现，而模型的其余部分则自动分区。下面是说明此用例的伪代码。

1 # 输入的形状为 [G , S , M]。 split () 不会改变逻辑形状 .2 input = split (input , 0 , num_devices)3 # s_indices 具有形状 [E , G , C , 1] 。值：输入中 S 的索引 .4 s_indices = split (s_indices , 1 , num_devices)56 # 开始手动分区 .7 #partitioned_input 的形状为 [G / num_devices , S , M]

```
partitioned_input = auto_to_manual_spmd_partition(input)
# partitioned_s_indices has shape [E, G/num_devices, C, 1]
```

partition ed_s_indice s = a ut o _ to _ man u al _ sp m d _ par t i t i o n (s_indices)# 与partitioned_input中的G索引相连接：G维度上的Iota .part iti one d_ gs_ in dic es = concat (iota ([E , G / num_devices , C , 1] , 1) ,partitioned_s_indices , 3)#partitioned_data 的形状为 [E , G / num_devices , C , M]partitioned_data = Gather (partitioned_input , par tit io ned _gs_indices)# 切换回 auto分区.# 数据的形状为 [E , G , C , M]

```
data = manual_to_auto_spmd_partition(partitioned_data)
...
```

3.3 The XLA SPMD Partitioner for GShard

本节介绍基于分片注释自动分区计算图的编译器基础设施。分片注释告知编译器每个张量应如何跨设备分布。 SPMD（单程序多数据）分区器（为了简单起见，称为“分区器”）是一个编译器组件，它将计算图转换为单个程序，以便在所有设备上并行执行。这使得无论分区数量如何，编译时间都接近恒定，这使我们能够扩展到数千个分区。 4

我们在 XLA 编译器中实现了分区器 [28]。包括 TensorFlow、JAX、PyTorch 和 Julia 在内的多个前端框架已经具有将其图表示转换为 XLA HLO 图的降低逻辑。与 TensorFlow 等流行的前端框架相比，XLA 的运算符集也少得多，这减少了实现分区器的负担，而不会损害通用性，因为前端现有的降低执行了繁重的工作以使其具有表现力。尽管我们在 XLA 中开发了基础设施，但我们在此描述的技术可以应用于其他机器学习框架中的中间表示（例如 ONNX [33]、TVM Relay [34]、Glow IR [35]）。

XLA 将计算建模为数据流图，其中节点是运算符，边是运算符之间流动的张量。分区器的核心是每个操作的处理，根据输入和输出上指定的分片将全尺寸的操作符转换为分区大小的操作符。当计算被分区时，引入了跨设备数据传输的各种模式。为了最大限度地提高大规模性能，必须定义一组核心通信原语并针对目标平台对其进行优化。

3.3.1 通信原语

由于分区器强制所有设备运行相同的程序，因此通信模式也是规则的，并且 XLA 定义了一组执行 MPI 样式通信的集体运算符 [36]。我们在下面列出了在 SPMD 分区器中使用的常见通信原语。

4 另一种替代方案是 MPMD（多程序多数据），它无法按图 2 所示进行扩展。

CollectivePermute 该运算符指定源-目标对的列表，并将源的输入数据发送到相应的目标。它用于两个地方：更改分区之间分片张量的设备顺序，以及本节后面讨论的光环交换。

AllGather 该运算符按照指定顺序连接所有参与者的张量。它用于将分片张量更改为复制张量。

AllReduce 该运算符对所有参与者的输入执行元素归约（例如求和）。它用于组合来自不同分区的部分减少的中间张量。在TPU设备网络中，当分区数量增长时，AllReduce具有恒定的成本（第5.2节）。它也是一种常用的原语，可以在其他类型的网络拓扑中有效实现[37]。

AllToAll 该运算符在逻辑上沿一个维度分割每个参与者的输入，然后将每个部分发送给不同的参与者。在从其他人接收到数据片段后，每个参与者将这些片段连接起来以产生其结果。它用于将分片张量从一个维度重新分片到另一个维度。 AllToAll 是在 TPU 设备网络中进行此类重新分片的有效方法，当分区数量增加时，其成本呈次线性增加（第 5.2 节）。

3.3.2 Per-Operator SPMD Partitioning

分区器的核心是根据指定的分片将每个算子从全尺寸算子转换为分区大小算子。虽然某些运算符（例如，elementwise）支持起来很简单，但我们讨论了需要跨分区通信的几种常见情况。

一般情况下存在一些重要的技术挑战，我们将在第 3.3.3 节中介绍这些挑战。为了使讨论与 MoE 模型更加相关，本节重点关注 Einsum 分区以说明一些通信模式。为了简单起见，我们假设所有张量都是均匀分区的，这意味着分区维度的大小是分区计数的倍数。

Einsum 案例研究 Einsum 是实施 MoE 模型中最关键的运营商。它们在 XLA HLO 中表示为点运算，其中每个操作数（LHS 或 RHS）由三种类型的维度组成：

· 批量维度是令人尴尬的平行维度。同一组批次维度必须存在于所有 LHS、RHS 和输出中，并且输出中的每个元素仅取决于 LHS 和 RHS 中相应的批次。

· 收缩尺寸仅存在于操作数中。 LHS 和 RHS 必须具有相同的一组收缩维度，并且它们在输出中被总结和折叠。

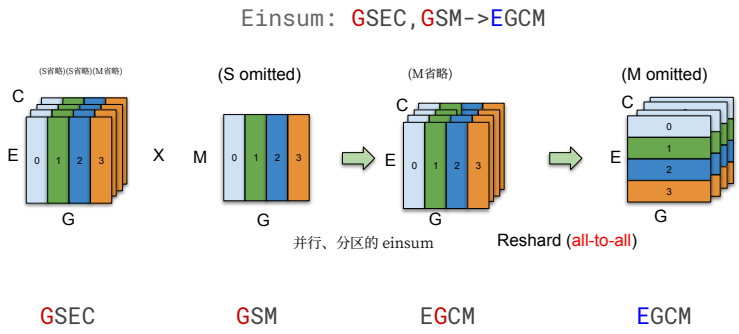
· 非收缩维度也是存在于操作数和输出之一中的平行维度。 LHS 和 RHS 中的每一个都有自己的一组非收缩维度，这些维度由输出继承。

分片传播优先考虑在 LHS、RHS 和输出的批量维度上选择相同的分片，因为这将避免任何跨分区通信。然而，这并不总是可能的，在以下三种情况下我们需要跨分区通信。

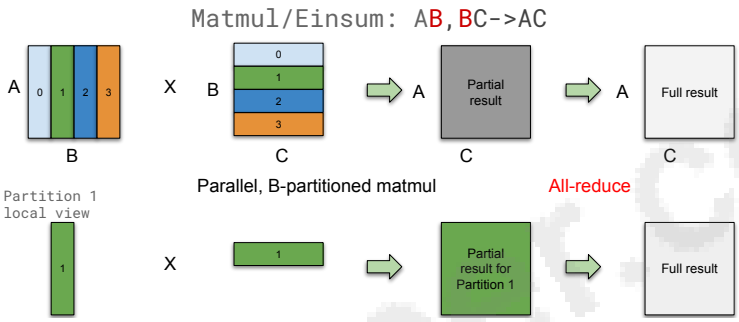
· 重新分片。在我们构建的 MoE 模型中，专家调度逻辑（算法 2 中的第 3 行）需要在 Einsum 之后切换分区维度。由于使用 AllToAll 进行重新分片非常有效（第 5.2 节），因此我们首先在本地执行 Einsum，然后将其重新分片到所需的维度，如图 4a 所示。

· 积累部分结果。如果输入沿收缩维度进行分区，则局部结果是部分的，我们需要使用 AllReduce 来组合它们并生成最终结果，如图 4b 所示。

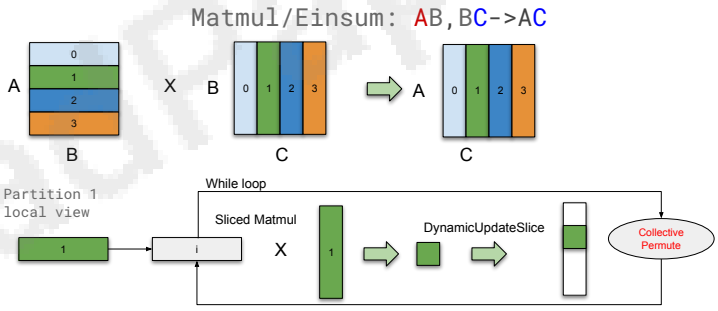
· 循环切片。对于某些场景，我们还实现了类似于Cannon算法[38]的算法，以限制每个分区上张量的大小。例如，



(a) 分区 Einsum 算子。彩色字母（G 和 E）代表每个张量的划分维度。分区器决定首先沿 G 维度执行批量并行 Einsum，然后将结果重新分片到 E 维度。



(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.

图 4：具有跨设备通信的 Einsum 分区示例。

如果两个操作数都在非收缩维度上进行划分，则我们无法直接计算局部 Einsum，因为操作数具有不同的非收缩维度。复制其中一个操作数不会导致冗余计算，但需要复制的操作数适合设备内存。因此，如果操作数的大小太大，我们就会对两个操作数进行分区，并使用循环迭代结果的每个切片，并使用 CollectivePermute 来传达输入切片（图 4c）。

3.3.3 支持全套算子

我们解决了几个额外的挑战，使 SPMD 分区器能够支持一整套算子，而无需张量形状或算子配置的额外约束。这些挑战通常涉及分区之间的不对称计算或通信模式，尤其是

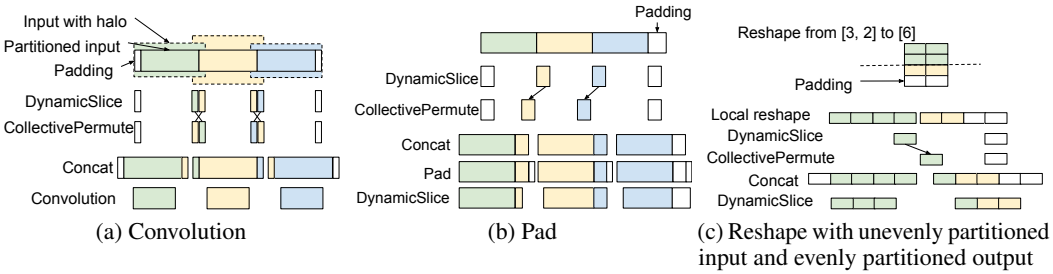


图 5：Halo 交换示例。

很难用 SPMD 来表达，因为单个程序需要对所有分区足够通用。我们不能简单地根据运行时设备ID在单个程序中创建许多分支，因为这会导致程序大小爆炸。

静态形状和不均匀分区 XLA 要求张量形状是静态的。 5 然而，当计算被分区时，并不总是所有分区都具有相同的输入/输出形状，因为维度可能无法被分区数量整除。在这些情况下，形状的大小将向上舍入到分区计数的下一个倍数，并且该填充区域中的数据可以是任意的。

在计算运算符时，我们可能需要将已知值填充到填充区域以确保正确性。例如，如果我们需要对Reduce-Add运算符进行分区，则需要使用恒等值零。考虑一个示例，其中分区维度 (15) 无法分为 2 (partitioncount)，因此分区 1 多了一列所需的列。我们创建一个范围为 [0, 8) 的 Iota 算子，添加分区偏移量（根据 PartitionId × 8 计算），并与完整形状偏移量 (15) 进行比较。根据谓词值，我们从操作数或从零中选择，结果是屏蔽操作数。

静态运算符配置 XLA 运算符具有静态配置，如卷积中定义的填充、步幅和膨胀。然而，不同的分区可能无法使用相同的操作器配置来执行。例如，对于卷积，最左侧的分区在其左侧应用填充，而最右侧的分区在其右侧应用填充。在这种情况下，分区器可能会选择一些配置，使某些分区产生比所需数据稍多的数据，然后切掉不相关的部分。附录 A.4 讨论了卷积和类似运算符的示例。

Halo 交换 某些操作员有一种通信模式，涉及与相邻分区进行部分数据交换，我们称之为 halo 交换。我们使用 CollectivePermuteoperator 在分区之间交换 halo 数据。

Halo 交换最典型的用例是用于基于窗口的运算符的分区（例如，卷积、ReduceWindow），因为相邻分区可能需要重叠的输入数据（图 5a）。在实践中，由于窗口配置（膨胀、跨步和填充）的高级使用以及不均匀的光环大小，这些运算符的光环交换通常需要与适当的填充、切片和遮罩相结合。我们在附录 A.4 中描述了各种场景。

光环交换的另一个用途是用于更改形状大小的数据格式化运算符。例如，在 Slice 或 Pad 运算符之后，张量的形状发生变化，分区之间的边界也发生变化。这需要我们重新排列不同分区上的数据，这可以作为光环交换的一种形式来处理（图 5b）。

其他数据格式化操作符虽然逻辑上不改变形状的大小，但也可能需要halo交换，特别是由于静态形状约束和不均匀分区。例如，Reverse 运算符反转张量中元素的顺序，但如果它的分区不均匀，我们需要跨分区移动数据，以使填充在逻辑上保持在结果张量的右侧。另一个例子是 Reshape。考虑将张量从 [3, 2] 重塑为 [6]，其中输入是

5 中间表示的有限活力通常是有效瞄准加速器所必需的。

在第一维上以两种方式不均匀地划分（划分形状 [2, 2]），并且输出也以两种方式划分（划分形状 [3]）。由于分区不均匀，输入上有padding，但是Reshape之后，输出张量不再有padding；因此，需要以与切片类似的方式进行光环交换（图 5c）。

编译器优化 SPMD 分区器创建各种数据格式化运算符，以执行切片、填充、串联、屏蔽和光环交换。为了解决这个问题，我们利用 XLA 在 TPU 上的融合功能，以及切片和填充的代码运动优化，以在很大程度上隐藏数据格式化的开销。因此，即使对于大量使用掩码和填充的卷积网络，运行时开销通常可以忽略不计。

4 大规模多语言、大规模机器翻译 (M4)

4.1 多语言翻译

我们选择多语言神经机器翻译 (MT) [39,40,41] 来验证我们使用 GShard 进行高效训练的设计。多语言 MT 本质上是一个多任务学习问题，旨在构建单个神经网络，以实现同时翻译多个语言对的目标。这将我们的工作范围 [15,14,16] 扩展到通用机器翻译模型 [42]，即可以在所有领域的一百多种语言之间进行翻译的单一模型。这种大规模多语言翻译模型不仅方便大规模压力测试模型，而且在现实生产系统中也被证明具有实际影响力[43]。

在大规模多语言 MT 中，有两个标准来定义模型质量的成功：1）具有大量训练数据（高资源）的语言所取得的改进，以及 2）具有有限数据（低资源）的语言的改进。随着单个翻译模型中要建模的语言对（任务）数量的增加，积极的语言迁移 [44]开始为低资源语言带来巨大的收益。考虑到考虑的语言数量，M4 在改善低资源任务方面具有明显优势。相反，对于高资源语言，任务数量的增加限制了模型内每个任务的容量，导致与在单一语言对上训练的模型相比翻译质量较低。高资源语言的容量瓶颈可以通过大规模增加模型大小来缓解，以满足额外容量的需求 [14, 15]。

因此，大规模多语言、大规模机器翻译的目标是在通过大规模多语言增加正迁移和通过大规模扩展缓解容量瓶颈之间取得平衡。同时，扩展模型大小和考虑的语言数量必须与方便的神经网络架构相结合。为了放大正迁移并减少负迁移 6 ，人们可以自然地设计一种模型架构，该架构包含跨语言的共享组件（共享子网络）以及一些特定于语言的组件（非共享、特定于语言的子网络）。然而，模型设计中的搜索空间（决定共享内容）随着语言数量的增加而迅速增长，使得基于启发式的搜索合适的架构变得不切实际。因此，对基于从数据中学习神经网络接线模式的方法的需求成为可扩展且实用的前进方向。

在本节中，我们提倡使用稀疏门控专家混合体 [16] 的条件计算 [45, 46] 如何适应上述详细的需求，并通过将神经机器翻译模型扩展到超过 1 万亿个参数来展示其有效性，同时保持如此大规模网络的训练时间实际的。例如。 M4 的 600B GShard 模型可以在 4 天内通过 250k 训练步骤处理 1T 令牌 7。我们尝试通过在模型中添加越来越多的专家来提高模型容量，并研究在收敛性、模型质量和训练效率中发挥作用的因素。此外，我们演示了条件计算如何加速训练[25]，以及如何在没有任何任务或语言相关性先验知识的情况下有效地学习通过网络的稀疏门控/路由每个令牌，举例说明了直接从数据学习路由决策的能力。

---

6 负迁移是通过不相关的任务共享模型容量的概念，这反过来会损害此类干扰任务的质量。7 子词分割后的源侧标记。

4.2 数据集和基线

逐渐扩大模型以获得更高质量的前提需要大量的训练数据[3]。继之前关于多语言机器翻译密集扩展的工作[15, 14]之后，我们致力于在野外进行机器翻译的实际测试床，并使用网络规模的内部数据集。从网络 [47] 中挖掘的训练语料库包含 100 种语言的并行文档，包括英语和英语，总共有 250 亿个训练示例。训练集的一些特征值得一提。从网络上挖掘后，联合语料库相当嘈杂，同时涵盖了不同的领域和语言。如此大的覆盖范围伴随着语言之间在每个语言对的示例数量方面的严重不平衡。这种不平衡遵循尖锐的幂律，范围从资源丰富的语言的数十亿个示例到资源匮乏的语言的数万个示例。虽然上述特征对我们的研究构成了挑战，但也使整体尝试尽可能切合实际。我们请读者参阅 [15, 14] 以获取所使用数据集的其他详细信息。

我们专注于提高从所有 100 种语言到英语的翻译质量（以 BLEU 评分 [48] 衡量）。这导致大约 130 亿个训练示例用于模型训练 8 。为了形成我们的基线，我们为每种语言对训练了单独的双语神经机器翻译模型（例如德语到英语的单个模型），并根据每种语言的可用训练数据进行调整 9 。我们没有显示每个语言对的单独 BLEU 分数，而是遵循将 x 轴上的基线置于零的惯例，并报告使用 GShard 训练的每个大规模多语言模型的 Delta BLEU 趋势线（见图 6）。图 6 按可用训练数据量的降序从左到右排序，其中最左侧分别对应于高资源语言，最右侧对应于低资源语言。重申一下，我们在通用机器翻译方面的最终目标是将单个多语言模型的 Delta BLEU 趋势线积累到高于所有考虑的语言的基线之上。我们还包括在与另一个基线相同的数据集上使用 GPipe 管道并行性进行训练的密集 96 层 Transformer Encoder-Decodernetwork T(96L) 的变体（图 6 中的虚线趋势线）。在 2048 个 TPU v3 核心 10 上进行训练到收敛需要超过 6 周的时间，其性能优于原始 GPipe T(128L) 11 [15]，并且是我们在比较中使用的最强的单一密集模型基线。

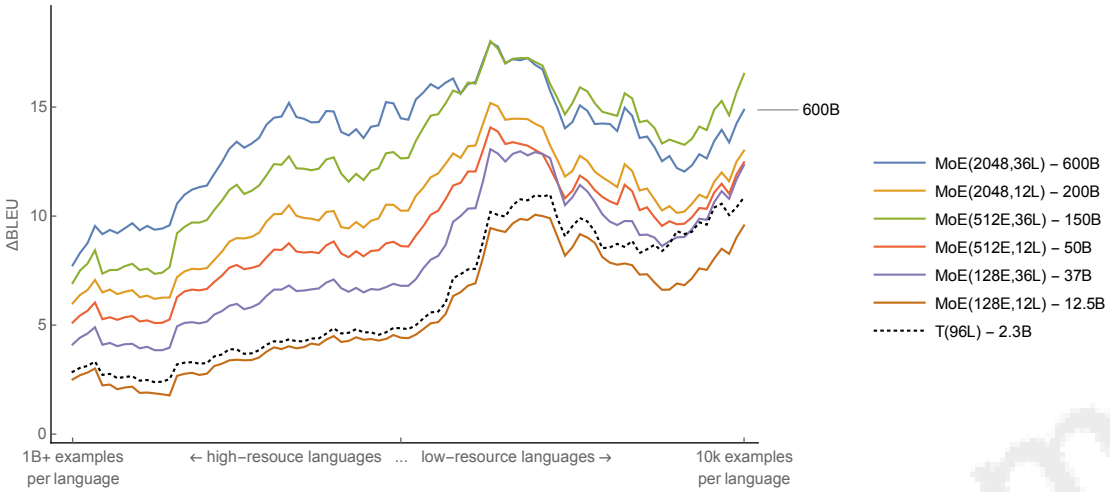
4.3 稀疏门控 MoE 变压器：模型和训练

扩展 Transformer 架构最近一直是一个探索性研究方向 [49,50,51]。不失一般性，新兴方法通过堆叠越来越多的层来扩展 Transformer [49, 15]，扩大网络的控制维度（即模型维度、隐藏维度或注意力头的数量）[4, 11]，并且最近学习了布线结构与 Architectureresearch [52] 12 。对于大规模多语言机器翻译，[15] 展示了使用 GPipe 管道并行性进行扩展的最佳实践；其中具有 60 亿个参数的 128 层 Transformer 模型被证明可以有效地改进高资源语言，同时表现出向低资源语言的最高正迁移。尽管非常有前途，并且满足了我们对通用翻译的期望，但 Transformer 架构的密集扩展具有实际限制，我们在第 1 节的训练效率中提到了这一点。

我们的目标是实际训练时间，并寻求保证训练效率的架构。我们的战略有三大支柱：通过堆叠更多类似于GPipe [15]的层来增加网络的深度，通过引入前馈网络（专家）的多个副本（如第2.2节中所述）来增加网络的宽度，并利用学习的路由模块（稀疏地）按照第 2.1 节所述将代币分配给专家。有了这三个成分，我们就得到了

---

8 与使用相同数据集的先前工作相比，哈萨克语和拉丁语到英语语言对被排除在评估之外。9 我们在 Transformer-Big 或 Transformer-Base 布局中调整了批量大小和正则化方法的不同值（例如 dropout），以实现高或低10 T(96L) 测量以 300k 步处理 1+ 万亿个 token，每步处理大约 4M 个 token，总预算为 235.5 TPU v3 核心年11 64 个编码器 + 64 个解码器层，16384 个隐藏暗淡，32 个注意力 head12 由于利用架构搜索的方法是计算密集型的，因此它们不属于本工作的范围。



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

图 6：使用 GShard 和单语言基线训练的多语言 MoE Transformer 模型的翻译质量比较。 x 轴上的位置代表语言，从资源丰富到资源匮乏。 Delta BLEU 表示单个多语言模型与针对特定语言训练和调整的单语言 Transformer 模型相比的质量增益。使用 GShard 训练的 MoE Transformer 模型具有可靠的趋势线。虚线趋势线表示在同一数据集上使用 GPipe 训练的单个 96 层多语言 Transformer 模型 T(96L)。为了清晰起见，每条趋势线均通过 10 的滑动窗口进行平滑。 （最好看彩色）

易于扩展、训练高效且具有高度表现力的架构，我们将其简称为“稀疏门控混合专家变压器”或“MoE 变压器”。

模型细节 为了详细说明模型的细节，每个专家都被设计为具有相同形状的规则 Transformer 前馈网络，并且专家（MoE 层）在每个其他 Transformer 层中分布一次。为了简单起见，我们将用于培训的设备数量与每个 MoE 层的专家数量联系起来，尽管这不是必需的。在训练过程中，我们使用 float32 来表示模型权重和激活，以确保训练稳定性。我们使用 MoE（2048E，60L）进行了额外的可扩展性实验，其中 bfloat16 [53] 激活总共有 1 万亿个模型权重。尽管可以通过仔细和手动诊断进行训练，但对于 1 万亿深度模型，我们遇到了几个数值稳定性的可训练性问题，因此为了再现性而未包括结果。更多模型和训练细节请参见附录A.2。

4.4 结果

在详细讨论训练效率之前，我们首先研究各种设计选择对构建 MoE Transformer 的影响。为了修剪搜索空间，我们探索了两种不同的方法

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

表 1：MoE 变压器模型系列。为了达到所需的容量，我们 i) 通过堆叠更多层来增加深度，ii) 通过扩展每个 MoE 层的专家数量以及用于训练的核心数量来增加网络的宽度。

变量、Transformer 编码器-解码器堆栈中的层数 (L) 以及用于每个其他 MoE 层的专家总数 (E)。对于深度，我们测试了三种不同的选项，12（原始Transformer深度，由6个编码器和6个解码器层组成）、36和60层。对于替换每隔一个前馈层的专家数量，我们还测试了三个选项，即128、512和2048个专家。请注意，用于训练的设备数量固定为等于每层专家的数量，分别使用 128、512 和 2048 个核心，与实验的深度无关。型号配置请参见表 1 的详细说明。

对于每个实验（表 1 的行），我们训练相应的 MoE Transformer 模型，直到它看到 1 万亿 (10 12) 个令牌。此时的模型检查点用于模型评估。到目前为止，我们在任何实验中都没有观察到任何过度拟合模式。相反，我们观察到，如果我们继续训练更长时间，训练损失会继续改善。我们评估了模型在保留的测试集上针对所有语言对获得的 BLEU 分数。图 6 报告了我们的所有结果。

在这里，我们分享每个实验的定性分析，并讨论每个设置对高资源和低资源语言的影响，以跟踪我们在通用翻译方面的进展。根据即将进行的分析，有必要重申潜在质量增益的预期行为。为了在单个模型中同时提高高资源和低资源语言的质量，缩放模型必须通过为高资源任务分配足够的容量来缓解容量瓶颈问题，同时通过促进充分的参数共享来放大向低资源任务的正向迁移。我们将此类系统的预期学习动态与长期存在的记忆和泛化困境松散地联系起来，最近沿着宽度与深度缩放工作进行了研究[54]。我们不仅期望我们的模型能够更好地泛化到保留的测试集，我们还期望它们能够表现出跨语言的高传输能力，作为泛化性能的另一种表现[55]。

更深的模型带来全面一致的质量增益我们首先研究高资源语言和低资源语言的模型深度和模型质量之间的关系。进行了三个不同的实验来测试泛化性能，同时保持数量每层专家数量固定。随着每个实验的每层专家数量不断增加（128、512 和 2048），我们将每个专家大小的网络深度增加了三倍，从 12 增加到 36。这导致了三个组，其中每层专家固定，但三倍于每组内的深度：

对于图 6 中所示的每个配置，我们观察到增加深度 (L)，同时保持每层专家 (E) 固定，为低资源语言和高资源语言带来一致的增益（沿 y 轴向上 Delta 移动），每次我们将深度从 12L 缩放到 36L 时，几乎都会使用恒定的附加因子（平均 2 到 3 个 BLEU 点，如表 3 最后一列所示）。

放宽容量瓶颈可显着提高质量，在第 4.1 节的前面，我们强调了容量瓶颈对任务干扰的影响，导致质量下降，尤其是对于资源丰富的语言。后来，我们通过增加每层专家的数量来缓解这种复杂性，这反过来又导致所研究模型的参数数量（权重）急剧增加。这里我们考察一下这个所谓的容量是否

瓶颈是明显可观察到的，并探讨一旦瓶颈放松对模型质量和效率的影响。为此，我们首先考虑三个具有相同深度（12L）的模型，每层专家数量不断增加：128、512 和 2048。当我们将每层专家数量从 128 增加到 512 的四倍时，我们请注意模型质量的大幅提升，100 种语言的 BLEU 平均得分为 +3.3。然而，每层专家数量再次扩大四倍，从 512 到 2048，仅产生 +1.3 平均 BLEU 分数。尽管质量显着提高，但收益的下降暗示着收益递减的出现。

据推测，对于我们实验设置中使用的特定参数化、语言数量和训练数据量，容量瓶颈预计存在于 128 到 512 名专家之间。一旦瓶颈得到缓解，模型的深度就会连续扩展，这一点可以通过比较 12 层和 36 层模型以及 128 位专家来看出。有趣的是，如果容量瓶颈没有得到缓解，增加深度并没有多大帮助。

拥有更多专家可以提高质量，尤其是高资源任务的质量另一个可以揭示多任务模型中扩展质量增益的维度是高资源语言改进和低资源语言改进之间的对比。如前所述，低资源语言从迁移中受益，而高资源语言则寻求增加容量。接下来我们检查在固定深度的同时增加每层专家的效果。

从图 6 中可以看出，对于 12 层模型，专家数量的增加会给高资源语言带来更大的收益，而不是早先揭示的低资源语言的收益递减。 36 层模型也观察到类似的模式。增加更多专家在缓解容量瓶颈的同时，由于共享子网络的减少而减少了传输量。

深度密集模型更擅长向低资源任务进行正迁移最后，我们研究深度对低资源任务的影响，作为我们之前实验的松散推论。为此，我们在分析中加入了使用 GPipeon 训练的 96 层 T(96L) 密集模型，并使用相同的数据。我们将 T(96L) 与浅层 MoE(128E, 12L) 模型进行比较。虽然对于大多数高中资源语言来说，两个模型之间的差距测量几乎恒定，但差距的增长有利于密集语言当我们进入资源匮乏状态时，深度 T(96L) 模型。按照我们之前的说法，随着任务之间共享子网络比例的增加（密集 T(96L) 为 100%），传输带宽得到最大化，并导致与浅层对应物相比具有更好的质量。另请注意，使用包含 370 亿个参数的 MoE（36E，128L）可以实现对资源匮乏语言的相同传输质量。

我们推测，增加深度可能会增加向低资源任务转移的程度，从而沿着该轴更好地概括。但我们还想强调的是，相比之下，这些模型对训练资源的要求不成比例。我们想再次强调培训效率的重要性，这正是我们接下来研究的主题。

4.5 Training Efficiency

在本节中，我们重点关注 MoE Transformer 模型的训练效率。到目前为止，我们已经看到了经验证据，证明沿各个轴扩展模型如何带来显着的质量提升，并研究了影响改进程度的因素。为了衡量训练效率，我们首先跟踪为达到特定训练损失而处理的令牌数量，然后跟踪模型处理一定数量令牌的挂钟时间。请注意，我们关注的是训练时间和训练损失 13 同时改变其他因素，而不是我们在上一节中分析的测试误差。

更深的模型样本效率更高，用更少的例子收敛得更快。事实证明，更深的模型样本效率更好，在给定相同数量的训练样本的情况下达到更好的训练/测试误差[15, 56]，通常归因于加速效应过度参数化[1]。我们使用 GShard 和 MoE Transformers 再次根据经验测试了该假设，并为不仅深度而且激活稀疏的模型进行权衡。

13 本节中报告的训练损失对应于交叉熵损失，并且不包括第 2.2 节中引入的辅助损失术语

为此，我们比较每个模型正在处理的令牌数量，以达到预设的训练损失。我们从表 2 中观察到的一个总体趋势是，深度为 3 倍的 MoE Transformer 模型需要减少 2 到 3 倍的 token 才能达到预设的训练损失阈值。例如，与 MoE(128E, 36L) 相比，MoE(128E, 12L) 需要 3 倍的标记数量才能达到 0.7 的训练交叉熵，(6) 与 (5)。我们观察到具有 512 名和 2048 名专家的模型也有类似的趋势，(4) vs (3) 和 (2) vs (1)。

十亿代币交叉熵为0.70.60.5(1) MoE(2048E, 36L)204882175542(2) MoE(2048E, 12L)2048 176484 1780(3) MoE(~~512E, 36L)512661~~70567(4) MoE(512E, 12) ~~L)512 141486-(5) MoE(128E, 36L)128 321 1074-~~(6) MoE(128E, 12L)128 995--表 2：模型在训练过程中看到的令牌数量达到三个不同的交叉点熵损失。总的趋势是，更深的模型比类似的浅层模型样本效率更高并且收敛速度更快。

表 2 中另一个有趣的观察结果再次与容量瓶颈的存在有关。比较具有相同深度的模型（5）、（3）和（1），我们注意到达到训练损失所需的令牌数量显着下降当我们的专家数量从 128 人过渡到 512 人时，该数字为 0.7。实际上，这就是我们观察到的容量瓶颈所在，与第 4.4 节中的假设一致。在此相移之后，具有充足容量的模型往往表现出类似的样本效率特征，如模型（3）和（1）中所示。

最大的模型 (600B) 可以在 4 天内完成训练，达到最佳质量 接下来我们深入研究模型大小和训练所花费的挂钟时间之间的相互作用。我们监控正在使用的 TPU 核心数量、每秒训练步骤、每批令牌总数、TPU 核心年数 14 以及训练所花费的实际挂钟时间（分别参见表 3 列）。

我们首先研究我们训练的最大模型之一，MoE(2048E, 36L)，有 6000 亿个参数，模型的 ID 为 (1)。使用 2048 个 TPU 核心 4 天后，该模型在平均 BLEU 方面实现了最佳翻译质量，但总共需要 22.4 个 TPU 年进行训练。虽然我们没有看到任何迹象表明随着我们扩大规模，质量改进会趋于稳定。模型，我们努力寻找具有成本效益的扩展解决方案。

表 3 中的结果再次验证了条件计算缩放比密集缩放更实用。考虑到 (1) 使用相同数量的 TPU 核心，密集缩放变体 T(96L) 似乎需要十次以上的时间来训练（235 个 TPU 核心年），而与经过训练的模型相比，模型质量落后与 GShard 一起。

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.

在本节中，我们使用 MoE Transformers 应用程序对 GShard 进行多语言机器翻译（特别是 M4）的基准测试。我们确定了影响最终结果的变量，例如

14 个 TPU 核心年数简单地通过核心数和挂钟时间（以年为单位）的乘积来衡量。

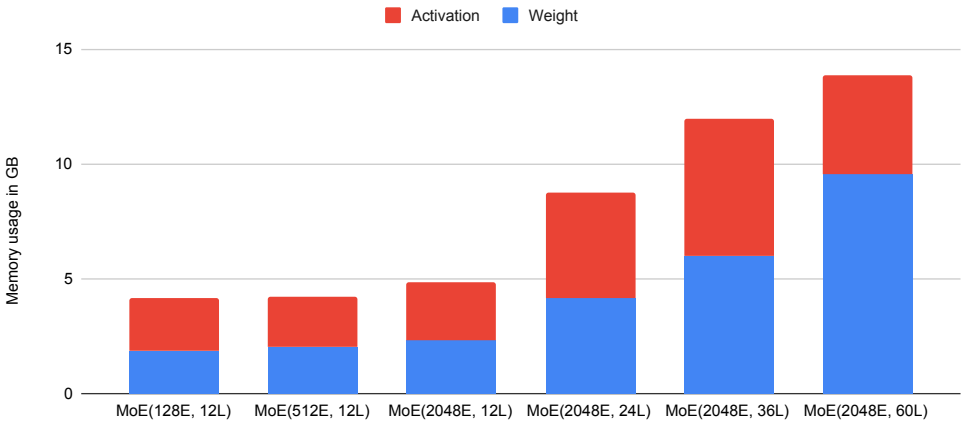


图 7：每个设备的内存消耗（以 GB 为单位）。

容量瓶颈、正迁移和训练效率，并提供实验结果以揭示它们之间的相互作用。接下来我们将深入研究 GShard 的性能相关主题，例如内存和运行时效率以及通信基准。

5 性能和内存消耗

本节讨论 GShard 如何在 TPU 平台上实现计算和内存效率。我们的测量和分析表明，当我们增加设备和专家的数量时，设备内存消耗大致恒定，并且步骤时间呈次线性增长，即当我们将模型从 128 台设备扩大到 2048 台设备 16 倍时，执行时间增加 1.7 倍。我们还为各种分区运算符提供微基准测试和分析，这可以指导本文之外的用例。

5.1 内存效率和可扩展性

在GShard模型中，内存使用主要有三种类型，当专家数量增加时，在SPMD分区后，每台设备的内存使用量都是恒定的。

- 复制权重（例如变压器前馈层）。
- 分布式权重（MoE 前馈层15）。
- 激活（在前向和后向传递中使用的每层的输出）。

图 7 演示了 O(1) 内存扩展，其中显示了不同模型的每设备内存使用分布。在层数固定的情况下，当专家数量增加时，权重记忆和激活记忆都保持不变。

另一方面，权重记忆和激活记忆都随着层数线性缩放。当内存需求超过每个设备上的可用内存时，基于编译器的重新物化将自动重新计算向后传递中的部分激活，以减少峰值激活内存。这就是为什么 MoE(2048E, 60L) 的激活大小小于 MoE(2048E, 36L) 的原因。重新物化的开销也得到了优化，例如对于 36L 和 60L 模型，仅总周期的 28% 和 34% 分别用于重新计算，而 12L 和 24L 则为 0%，因为它们无需重新物化即可装入设备内存。

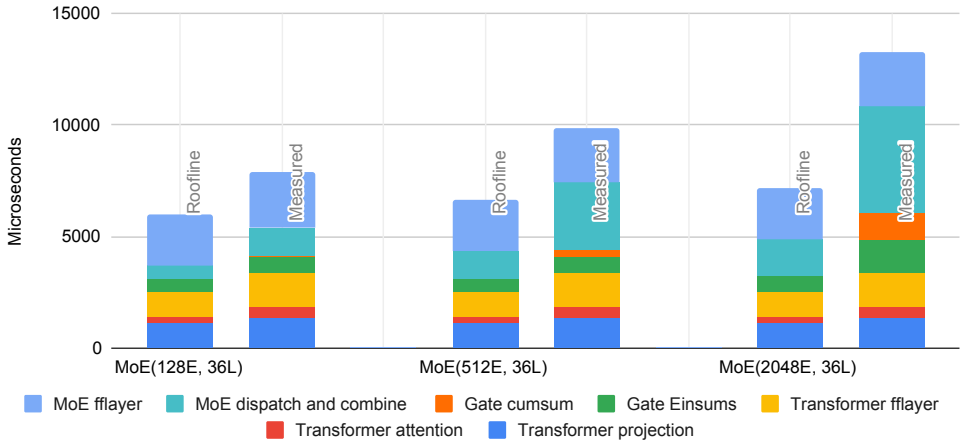


图 8：测量与 roofline 执行时间细分。只显示了前向传递，后向传递也有类似的故障。 “MoE调度合并”代表与AllToAll的跨分区通信。

5.2 运行时效率和可扩展性

图 8 显示了 MoE 层及其相邻 Transformer 层的执行时间细分。 它还将所实现的性能与 roofline 进行了比较，这是通过假设计算、内存或通信限制的操作可以实现 100% 的性能来估计的。峰值 FLOPS、内存带宽或互连带宽。这是一个非常乐观的估计，因为许多运营商都受到一组混合资源的限制。在较小的规模（128 名专家）下，我们的模型可以实现 > 70% 的屋顶线性能。当我们将模型扩大到 16 倍（2048 名专家）时，设备时间增加了 1.7 倍，并且仍然可以实现 48% 的 roofline 性能。

在分析性能可扩展性之前，我们回顾一下 3.1 节中讨论的相关张量维度的大小缩放。对于D设备，专家数量E和组计数G都设置为O(D)。每组专家能力的分数 C 设置为 O(1/D)。此设置无法无限扩展，因为 C 需要至少为 1，但它足以扩展到数千名专家。

Transformer 层和 MoE 前馈层这些是模型的密集部分，旨在实现峰值 TPU 利用率。在每台设备上，当我们扩展到更多专家时，这些计算也会产生恒定的成本。前馈层和 Transformer 投影主要是大型矩阵乘法，很好地利用了 TPU 的矩阵单元。在我们的实验中，这些操作已实现 > 85% 的峰值 FLOPS。注意力操作主要由批量 matmul 组成，当序列长度很小时，它们会受到内存带宽的限制。因此，在我们的实验中，注意力操作仅实现了 > 30% 的峰值 FLOPS。

门计算 在图 8 中，“Gate Einsum”代表算法 2 中的前两个和最后一个 Einsum。第一个 Einsum 是计算每个专家输入到 softmax 的投影。它具有 O(D) 成本，但它只占该层的很小一部分。另外两个 Einsum 正在调度代币并结合专家结果。他们使用 one-hot 矩阵有效地实现了 Gather，这种矩阵更昂贵，但成本恒定 O(GC) = O(1)，与专家的数量无关。当我们将专家数量从 128 名扩展到 2048 名（16 倍）时，这些 Einsum 的执行时间增加了约 2 倍。

剩余的每设备门控计算涉及许多通用计算，例如 ArgMax 和 Cumsum，它们本质上要么是内存限制的，要么是顺序的，因此不能很好地利用 TPU。大部分时间花在对代表每个 token 的选定专家的逆热矩阵的顺序 Cumsum 操作到代表的单热矩阵

15 门投影权重的大小为 O(E)，可以进行分区，但实际上它们足够小，可以复制，并且对峰值内存使用的影响可以忽略不计。

为每位专家选择的代币。 Cumsum的线性复杂度如图8所示。这部分门控计算也有O(D)成本，但幸运的是，与softmax之前的Einsum类似，它有一个非常小的常数因子。它在 128 名专家的情况下执行时间可以忽略不计，并且只占 MoE 层和 Transformer 层 2048 名专家花费的总时间的 10% 以下。

门控最重要的部分是通信，如图 8 中的“MoE 调度和组合”所示。这些是 AllToAll 运算符，正如我们将在第 3.3 节中讨论的，它们的成本是 O(

√D)。 专家数量从128增长16倍到2048时，执行时间增加约3.75倍，其在MoE和Transformer中的执行时间比例从16%增加到36%。

5.3 通信微基准和每个操作员的可扩展性

在本节中，我们测量和分析基本算子的 SPMD 分区器的性能可扩展性，这可用于指导本文提出的 MoE 模型之外的用例。

通信原语的性能扩展 MoE 模型中的两个关键集体通信运算符是 AllReduce 和 AllToAll。 AllReduce 用于累积部分结果，AllToAll 用于重新分片（第 3.3.2 节）。图 9 显示了从 16 个分区到 2048 个分区的性能可扩展性。 TPU 上的 AllReduce 的执行时间与设备数量无关。图 9 中的差异是由于每种拓扑的具体情况造成的，例如，它是正方形还是矩形，以及它是圆环还是网格。

另一方面，随着分区数量的增加，AllToAll 会变得更加昂贵，但是以次线性的方式。在我们的 2D TPU 集群上，AllToAll 成本大致为 O(

√D)，其中D是分区数。这是因为每个分区发送的数据量固定（图 9 中的 8MB 或 32MB），所有分区发送的数据总量为 d = O(D)。同时，每个数据块需要传输 h = O(

√D) 跳，网络中总共有 l = O(D) 条设备到设备链路。因此，如果它平均有受带宽限制，则 AllToAll 的执行时间为

$$— = O(\frac{D\sqrt{D}}{D}) = O(\sqrt{D}).$$

√D）。比较2048个分区和16个分区，当D增长128倍时，AllToAll的执行时间只增加了9倍。这使我们能够使用重新分片来有效地实现跨分区调度（图 4a）。

AllGather 和 CollectivePermute 更容易分析。 AllGather的输出比输入大D，如果我们固定输入大小，那么它的通信成本是O(D)。 CollectivePermute 具有一对一的通信模式，并且通过合理的设备布置（源-目标对靠近），对于固定的输入大小，其成本为 O(1)。

	$O(D)$ 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

表 4：分区运算符的可扩展性。通信原语的缩写：AR：AllReduce、AG：AllGather、CP：CollectivePermute、AA：AllToAll。 *这是我们模型中的dispatchEinsum，我们将C 设置为O(1/D)。 **I/O 是输入/输出特征维度，Bi 是批量维度，X/Y 是输入空间维度，x/y 是内核空间维度。

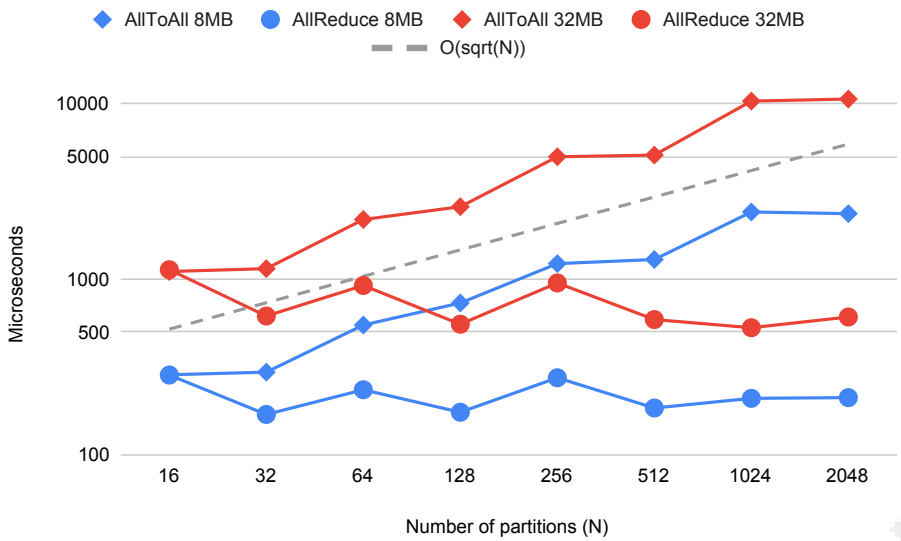


图 9：通信、AllReduce 和 AllToAll 的性能扩展，双轴上的对数刻度。AllReduce 成本大致为 O(1)，AllToAll 成本大致为 O(

√D)，其中D是分区数。我们使用 8MB 和 32MB 数据来衡量它们的性能。对于AllToAll，这意味着每个分区最初有8MB（或32MB）数据，然后将其分为D块，并将每块发送到不同的接收分区。

分区算子可扩展性 我们在表 4 中总结了使用 GShard 的常见算子的性能可扩展性。它包含第 3.3.2 节中的 Einsum/Matmul 示例，以及其他常见算子，如卷积和化简。该表包括每个分区上的本地计算，以及基于我们上面的分析所需的通信。

表4中的大多数算子在计算和通信方面都具有亚线性可扩展性，这与我们对MoE模型的性能测量是一致的。空间分区卷积的 O(1) 缩放也证明了 GShard 在图像分区方面的效率（附录 A.4）。

然而，表 4 中的最后两个 Matmul 运算符对每分区计算和通信具有 O(D) 缩放，它们在操作数中具有无与伦比的分片。这并不是由于分区算法效率低下，而是因为完整运算符中的总计算量非常大 (O(D 2))。针对这些情况可以使用不同的分区策略，从而产生不同的通信原语：复制一个操作数将导致AllGather（需要复制的操作数以适应设备内存），而循环切片（图 4c）将导致 CollectivePermute。

6 相关工作

神经网络深度学习模型在推进人工智能子领域方面非常成功。多年来，该领域一直在不断报告新的最先进的结果，使用各种模型架构进行计算机视觉任务[57,58,7]，自然语言理解任务[59,60,61]，语音识别和合成任务[62、63、64、65、66]。最近，基于注意力的 Transformer 模型进一步推进了这些领域的最新技术水平 [10, 4]。

模型缩放学术研究和行业应用都观察到，较大的神经网络往往在足够大的数据集和复杂任务上表现更好。在单个模型系列中，根据经验，简单地使网络更宽或更深通常可以提高模型质量。例如，更深的 ResNet 表现更好 [8]，更大的 Transformer 模型实现了更好的翻译质量 [10]，具有更大词汇量、嵌入或特征交叉的模型也效果更好 [14, 13]。穿过

不同的模型系列中，还观察到具有较大模型容量的较大模型不仅可以更好地拟合训练数据，而且在测试时间上也可以更好地泛化[67,68,15]。这一观察结果促使许多研究努力构建比深度学习研究模型或生产模型中通常使用的神经网络更大的神经网络。沙泽尔等人。结果表明，使用混合专家层的具有 690 亿个参数的循环语言模型在 10 亿个单词 (LM1B) 基准测试中实现了低得多的测试困惑度 [16]。布朗等人。结果表明，非稀疏 1750 亿个参数模型能够在多个下游 NLP 任务上表现出高度准确的小样本性能。

硬件神经网络需要不可忽略的计算能力。为了满足这种需求，为神经网络训练和推理而构建的特殊硬件（芯片和网络机器）可以追溯到 25 年前[69]。自 2000 年代末以来，研究人员开始利用 GPU 来加速神经网络 [70,57,71]。最近，该行业还投入巨资构建更多专用硬件系统，以追求更具成本效益的神经网络硬件[72]。由于神经网络的核心计算（各种形式的乘法求和：卷积、矩阵乘法、einsum）是高度并行化的数值计算，因此这些芯片配备了大量的浮动处理单元（FPU）。因此，这些专门设计的硬件的计算能力急剧增长。据报道，GPU 每浮点运算的价格在过去 4 年里下降了 10 倍 [73]，而每瓦浮点运算的价格在过去 12 年里增加了 2 个数量级 [74]。广泛可用的低成本计算能力是神经网络成功的主要推动力。

软件 支持神经网络的软件系统随着底层硬件的进步而发展[75,76,21,77]。虽然加速器是高度并行的计算机器，但直接编程要困难得多。这些框架使构建神经网络变得更加容易，并从实践者那里抽象出了许多硬件特定的细节。他们反过来依赖较低级别的库来有效地驱动特殊硬件（加速器）。例如，用于 Nvidia 的 GPU 的 CUDA [78]，或用于 Google 的 TPU 的 XLA [28]。这些较低层的库对于使用这些特殊硬件实现高效率至关重要。

模型训练和推理中的并行性现代神经网络广泛使用机器集群进行训练和推理，每个机器都配备了多个加速器。数据并行性[57]是最常用的方法，并受到主要框架的支持（TensorFlow [21]]、PyTorch [22]、JAX [79, 80]），其中设备使用不同的输入数据运行相同的程序，并在权重更新之前组合其局部梯度。另一方面，模型并行性将计算划分到输入批次之外，这是构建非常大的模型所必需的。例如，流水线 [15, 24] 将大型模型的层拆分为多个阶段，而运算符级别分区 [23, 81] 将单个运算符拆分为较小的并行运算符。 GShard 使用一种操作员级分区来将我们的模型扩展到大量并行专家。

自动并行性由于在分布式异构环境中编程具有挑战性，特别是对于高级从业者而言，深度学习框架试图减轻用户指定如何完成分布式计算的负担。例如，Tensor-Flow [21] 支持数据并行性，以及通过按节点设备分配进行图形分区的基本模型并行性。 Mesh TensorFlow [23] 通过在 TensorFlow 之上的 Python 库中重写计算，帮助用户使用 SPMD 风格的每运算符分区构建大型模型；相比之下，我们的方法基于轻量级在编译器中对图进行分区注释而不需要用户重写模型。 FlexFlow [81] 使用自动搜索来发现图中运算符的最佳划分，以获得更好的性能；虽然它侧重于确定分区策略，但我们的 SPMD 分区器侧重于转换注释图的机制。权重更新分片[82]是另一种基于XLA的自动并行化转换，主要关注TPU集群的性能优化，概念上可以视为GShard的特例。 Zero [83] 提出了一组优化，通过单独划分权重、激活和优化器状态来减少并行训练设备中的内存冗余，并且能够将模型扩展到 1700 亿个参数；相比之下，GShard 更通用，因为它不区分这些张量，并且所有这些特定的分区技术都可以通过简单地注释相应的张量来支持，从而使我们能够扩展超过 1 万亿个参数并探索更多的设计选择。

条件计算和机器翻译条件计算[25,16,26,27]前提是示例应该通过激活依赖于输入的子节点在网络内路由。

网络。路由取决于（或条件）特定的标准并且不失一般性，可以是以下任何一个：示例的估计难度[84]，可用的计算预算[26, 27]，或者更一般地具有稀疏性的学习标准诱导专家的混合物[16]。由于稀疏门控专家混合模型 [16] 的灵活性和易于扩展性，我们将其扩展到最先进的神经序列模型 Transformers [10]，以满足训练效率。

7 结论

在本文中，我们介绍了 GShard，这是一个可以自动大规模划分计算的深度学习模块。GShard 仅使用用户模型代码中所需的轻量级分片注释进行操作，并提供易于使用且灵活的 API，用于扩展巨型神经网络。我们应用 GShard 来扩展带有稀疏门控混合专家层 (MoETransformer) 的 Transformer 架构，并展示了 600B 参数的多语言神经机器翻译模型可以在 4 天内有效地进行训练，在将 100 种语言翻译成英语时，与现有技术相比，实现了卓越的性能和质量与单一模型。除了更好的翻译质量之外，使用 GShard 训练的 MoE Transformer 模型在训练效率方面也表现出色，训练成本为 22 个 TPU v3 核心年，而训练所有 100 个双语 Transformer 基线模型则需要 29 个 TPU 年。本文提出的实证结果证实，利用条件计算扩展模型不仅可以提高现实世界机器学习应用的质量，而且在训练过程中保持实用性和样本效率。我们提出的方法提出了有利的可扩展性/成本权衡，并减轻了对用于扩展巨型神经网络的特定于模型的框架或工具的需求。总之，我们的结果有助于阐明神经网络扩展的现实和实用方法，以实现更好的模型质量。

我们从研究中吸取了一些教训。我们的结果表明，神经网络的逐步扩展可以产生一致的质量增益，验证了当我们扩展模型时，质量改进尚未达到稳定水平。虽然本文的结果巩固了模型扩展是深度学习从业者工具箱中的必备条件，但我们也敦促从业者努力提高训练效率。为此，我们确定了影响训练效率的因素，并展示了它们对下游任务质量的影响。我们演示了通过条件计算构建的神经网络如何在规模和计算成本之间产生有利的权衡。在实践中，此类关键的设计决策使我们能够享受不是数月或数周，而是数天的实验周期，以数万亿个参数的数量级训练模型。

此外，拥有一个适当的抽象层将模型描述与并行化实现分开，可以让模型开发人员专注于网络实现，让 GShard 自动划分计算图并生成在所有设备上并行运行的程序。我们发现，生成一个足够通用的单个程序来表达所有底层并行设备上的计算是可扩展编译的关键。为不同分区生成多个专用程序的传统方法会在扩展到数千个分区时导致编译时间爆炸。为了解决这种复杂性，我们引入了基于 SPMD 分片的各种编译器改进，允许对任何张量维度进行分区。作为要点，我们强调模型扩展和训练效率应该齐头并进；条件计算等算法改进与易于使用的界面相结合，可以有效地利用强大的计算能力。

最后，我们的实验结果凭经验证明，单纯的参数计数并不总是与大规模模型的有效容量相关[85, 86]。模型的比较还应该考虑问题的本质，即大规模多任务设置，任务之间训练数据严重不平衡，就像我们的例子一样，并控制影响网络不同运行模式的因素，即容量瓶颈与正向转移。

Acknowledgements

我们要感谢 Google Brain 和 Translate 团队提供的有用意见和富有洞察力的讨论，感谢整个 XLA 和 Lingvo 开发团队为本项目做出的基础性贡献。特别是 Youlong Cheng、Naveen Arivazhagan、Ankur Bapna、庞若明、YonghuiWu、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 和 Laurens van der Maaten。探索弱监督预训练的局限性。欧洲计算机视觉会议 (ECCV) 会议记录，第 181-196 页，2018 年。

[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、MaximKrikun、Mia Xu Chen、Yuan Cao、George Foster、Colin Cherry、Wolfgang Macherey、ZhifengChen 和 Yonghui Wu。大规模多语言神经机器翻译：发现和挑战，2019 年。

[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] 马里奥·盖格、阿瑟·贾科特、斯特凡诺·斯皮格勒、弗兰克·加布里埃尔、莱文特·萨贡、斯特凡·德·阿斯科利、朱利奥·比罗利、克莱门特·洪勒和马蒂厄·怀亚特。用深度学习中的参数数量来缩放泛化描述。统计力学杂志：理论与实验，2020（2）：023401，2020年2月。

[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 Koanan-takool、Peter Hawkins、HyoukJoong Lee、Mingsheng Hong、Cliff Young 等。Mesh-tensorflow：超级计算机的深度学习。《神经信息处理系统进展》，第 10414-10423 页，2018 年。

[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] 程友龙，李赫仲，塔马斯·伯格哈默。在 Cloud TPU 上通过空间分区在大图像和 3D 体积上训练 ML 模型。https://cloud.google.com/blog/products/ai-machine-learning/train-ml-models-on-large-images-and-3d -云 tpus 上的空间分区卷，2019 年。在线的;访问日期：2020 年 6 月 12 日。

[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 和 Zachary Tatlock。 Relay：机器学习框架的新 ir。第二届 ACM SIGPLAN 机器学习和编程语言国际研讨会论文集 -MAPL 2018，2018。

[35] 纳达夫·罗特姆、乔丹·菲克斯、萨利姆·阿卜杜勒拉索尔、加勒特·卡特伦、萨默·邓、罗曼·扎巴罗夫、尼克·吉布森、詹姆斯·赫格曼、梅根·勒勒、罗曼·莱文斯坦、杰克·蒙哥马利、伯特·马赫、萨蒂什·纳达图尔、雅各布·奥尔森、Jongsoo Park、Artem 拉霍夫、米沙·斯梅尔扬斯基和王曼。 Glow：神经网络的图降低编译器技术，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 和 Yoshua Bengio。具有共享注意力机制的多路、多语言神经机器翻译。计算语言学协会北美分会 2016 年会议论文集：人类语言技术，2016 年。

[40] Melvin Johnson、Mike Schuster、Quoc V. Le、Maxim Krikun、吴永辉、Zhifeng Chen、Nikhil Thorat、Fernanda Viégas、Martin Wattenberg、Greg Corrado 等。谷歌的多语言神经机器翻译系统：实现零样本翻译。计算语言学协会汇刊，5:339–351，2017 年 12 月。

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

[42] 探索大规模多语言、大规模神经机器翻译。 https://ai.googleblog.com/2019/10/exploring-massively-multilingual.html。访问时间：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 和 Moshe Dubiner。用于机器翻译的大规模并行文档挖掘。第 23 届国际计算语言学会议论文集，COLING '10，第 1101-1109 页，美国，2010 年。计算语言学协会。

[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、XiaobingLiu、Hemal Shah、Levent Koc、Jeremiah Harmsen 等。推荐系统的广泛和深度学习。第一届推荐系统深度学习研讨会论文集 - 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 和 Andrew Rabinovich。深入研究卷积。IEEE 计算机视觉和模式识别会议论文集，第 1-9 页，2015 年。

[59] Ilya Sutskever、Oriol Vinyals 和 Quoc V Le。使用神经网络进行序列到序列学习。神经信息处理系统进展，第 3104-3112 页，2014 年。

[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] 吴永辉，Mike Schuster，陈志峰，Quoc V Le，Mohammad Norouzi，WolfgangMacherey，Maxim Krikun，曹元，高勤，Klaus Macherey，等。谷歌的神经机器翻译系统：弥合人类和机器翻译之间的差距。 arXiv 预印本arXiv：1609.08144，2016。

[62] Geoffrey Hinton、邓力、于东、George E Dahl、Abdel-rahman Mohamed、NavdeepJaitly、Andrew Senior、Vincent Vanhoucke、Patrick Nguyen、Tara N Sainath 等。用于语音识别中声学建模的深度神经网络：四个研究小组的共同观点。IEEE 信号处理杂志，29(6):82–97, 2012。

[63] William Chan、Navdeep Jaitly、Quoc Le 和 Oriol Vinyals。听、听和拼写：用于大词汇量会话语音识别的神经网络。 2016 年 IEEE 国际声学、语音和信号处理会议 (ICASSP)，第 4960-4964 页。 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，庞若明，Ron J Weiss，Mike Schuster，Navdeep Jaitly，杨宗衡，陈志峰，张宇，王宇轩，Rj Skerrv-Ryan，等。通过对梅尔频谱图预测调节波网进行自然 tts 合成。2018 年 IEEE 国际声学、语音和信号处理会议 (ICASSP)，第 4779-4783 页。 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 Cireşan, 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、RaminderBajwa、Sarah Bates、Suresh Bhatia、Nan Boden、Al Borchers 等。张量处理单元的数据中心内性能分析。第 44 届计算机架构国际研讨会论文集，第 1-12 页，2017 年。

[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] 诺姆·沙泽尔。快速变压器解码：您只需要一个写入头。 arXiv preprintarXiv: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.

附录

A.1 使用平波束搜索进行解码

在解码过程中，我们使用类似于[61]的带有长度归一化的波束搜索。解码是自回归的，并且每次生成一个令牌的目标序列，因此对于长度为 m 的输出，解码器层堆栈会顺序执行 m 次。特别是对于每个解码器 MoE 层，都有调度/组合操作，这需要跨设备通信。推理使用与训练相同的具有相同数量设备的集群。

在波束搜索期间，我们将波束假设展平为包含所有交错的底层标记的单个序列，并且我们修改解码器自注意力掩码，以便每个假设仅关注联合平坦序列中的适当位置。我们对每个解码器自注意力层维护的键/值张量应用相同的转换。这使我们能够避免在每次波束扩展后重新排序先前计算的注意键/值。相反，我们只对代表当前有效假设的 0/1 掩码进行重新排序。然而，注意力的时间会延长 k 倍。

这种权衡可以是积极的也可以是消极的，具体取决于实施细节。正如[87]中所解释的，内存带宽限制对于 Transformer 模型的增量解码非常重要。从这个角度来看，通过展平波束，我们用低计算/内存比（注意力点积和键/值重新排序）替换了两个操作计算/内存比率稍高的单个操作（注意具有更多键的较长序列的点积），但必须访问的内存总量相同。

A.2 机器翻译实验细节

在我们的机器翻译实验中，MoE Transformer 模型共享 a) 1024 Transformer 模型维度 b) 8192 前馈和 MoE 隐藏维度； c) 多头注意力中的 16 个头； d)128个注意力键和值维度； e) 0.1 输入、残差和注意力丢失率。

我们使用 Adafactor [88] 优化器和 a) 因子二阶矩估计； b) 一阶矩衰减 β 1 = 0.0; c) 二阶矩衰减 β 2 = 0.99，时间表为 1 − t −0.8； d) 将剪裁阈值更新为 1.0； e) 10k 训练步骤后具有平方根衰减的 1.0 学习率。

我们使用 SentencePiece [89] 子词分词器和单个多语言词汇，源端跨 102 种语言，大小为 64000，目标端词汇为英语，大小为 32000。

A.3 General Sharding API

除了 3.2 节中列出的用于分片的两个常见 API（replicate() 和 split()）之外，用户或编译器还可以使用更高级的分片策略来最大程度地减少数据传输。

shard(tensor, device_assignment) 使用提供的 deviceassignment 注释要分区的张量，并返回带注释的张量。我们使用设备分配（一个多维整数数组）来表示分割是如何完成的。 device_assignment 与数据张量具有相同的秩；其元素计数为分区总数，每个元素为占用相应数据片的设备的 ID。例如，形状为 [3, 16, 64] 且设备分配形状为 [1, 2, 4] 的 3D 张量将具有分区形状 [3, 8, 16]，设备分配中元素的顺序决定每个分区占用哪个切片。

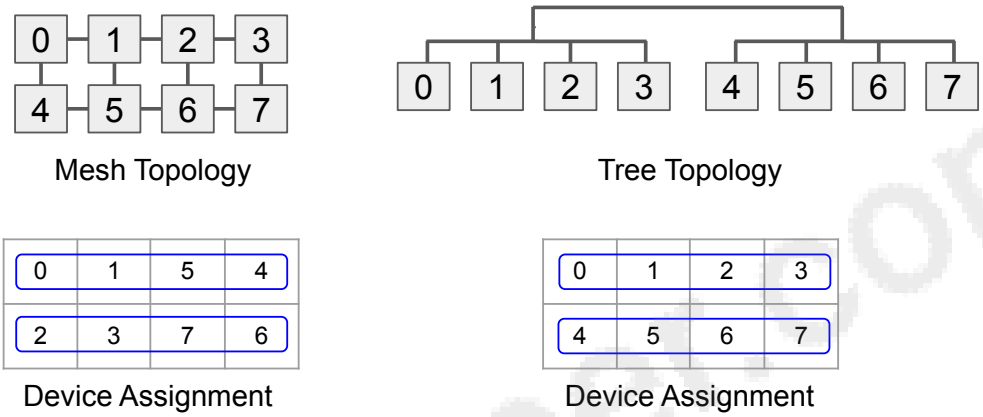


图 10：基于设备拓扑的两种不同设备分配示例。 2D 张量被 2x4 分区分割，并且通信模式是沿着张量的行的分区之间的。这些数字代表设备 ID。

由于跨设备的数据移动严重影响并行执行性能，因此在设备分配中分配设备 ID 以获得最大性能时，考虑目标设备拓扑以及张量分区之间的通信非常重要。图 10 显示了基于设备拓扑和张量上的行式通信模式的两种不同的设备分配。

A.4 卷积和基于窗口算子的 SPMD 划分

GShard 能够在卷积中划分空间维度，并且足够通用以支持巨型图像等用例 [32]。为了对卷积层进行空间分片，我们可以按以下方式使用分片 API。

```
# 沿 W 空间维度分割输入图像 [N ,C ,H , W ]
input = split(inputs, 3, D)#复制内核

内核 = 复制 ( 内核 ) conv =
conv2d ( 输入 , 内核 )...
```

然后，GShard 将空间维度上的分片传播到其他层和反向传递。本节的其余部分讨论分区卷积和类似算子的具体复杂性。有几种基于窗口的操作（例如，卷积、ReduceWindow），并且它们都需要某种类型的光环交换，因为数据可以在窗口之间共享。我们使用 CollectivePermute 运算符在分区之间交换 halo 数据，但一个复杂的情况是，不同分区之间的 halo 大小可能不同，而 CollectivePermute 需要静态整形。

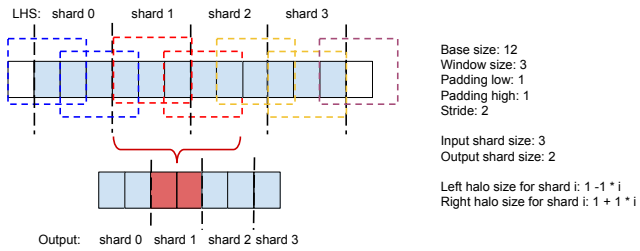


图 11：具有非恒定光晕大小的卷积。

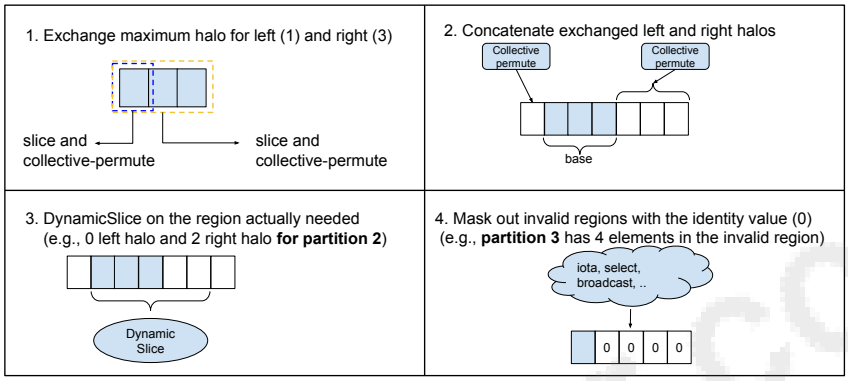


图 12：一般 Halo 交换的操作序列。

我们首先介绍 SPMD 分区器必须考虑的窗口配置。卷积中的每个空间维度都有以下一组配置。

· 步长是窗口移动以产生下一个输出元素的距离（以元素数量为单位）。 · 低/高填充是填充到 LHS（基）维度低/高端的元素数量。 · 基膨胀是LHS 的膨胀因子，即每个元素之间填充的元素数量加一（不包括低/高填充）。无基础膨胀意味着值设置为 1。 · 窗口膨胀是 RHS（窗口）中每个元素之间填充的元素数量加一。

光晕大小非恒定。我们使用一个没有膨胀的简单示例来证明非恒定光环大小是常见的。图 11 显示了 4 路分区卷积，其中分区的右晕大小为 (1, 2, 3, 4)，可以表示为分区 ID 的线性函数：partition_id + 1。分区 1 负责生成2个输出元素（红色单元格），这意味着该分区需要从分区0获取0个元素，从分区2获取2个元素（两个红色虚线窗口覆盖的区域）。

图 12 描述了一般 halo 交换的操作顺序。首先，我们计算跨分区的左右光环的最大尺寸，并执行最大尺寸的光环交换（步骤1和2）。由于某些分区可能有过多的光环，因此我们使用DynamicSlice（基于分区ID）来切掉当前分区的有效区域（步骤3）。最后，某些分区可能包含垃圾值（例如，来自超出范围的输入数据的光环），因此我们应用第 3.3.3 节中所述的屏蔽。

基底膨胀。基础膨胀为光环交换增加了额外的复杂性，因为每个分区的偏移可能位于膨胀孔处，并且膨胀后还应用了低/高填充，这使得边缘具有与内部元素不同的行为。我们在 3 种情况下处理基底扩张（图 13）。

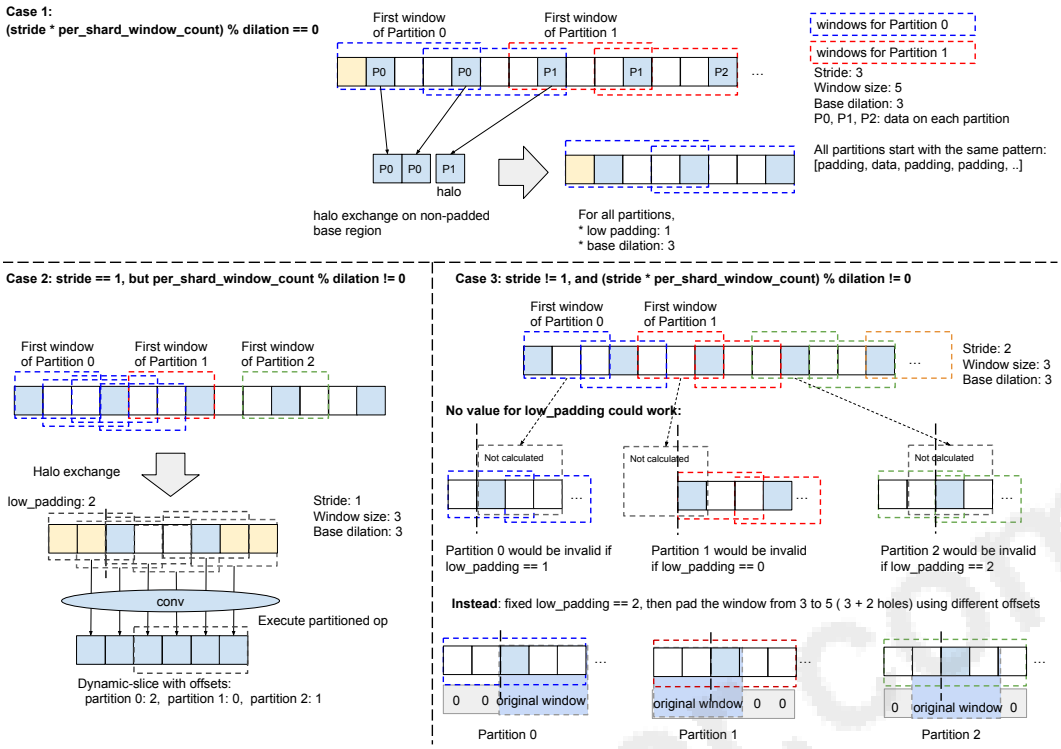


图 13：带有基础扩张的分区卷积。

· stride × per_shard_window_count 可被膨胀整除，其中 per_shard_window_count 是每个分区要处理的窗口数（即每个分区的输出元素数）。此条件保证所有分区在 LHS 中的第一个数据元素之前以相同数量的（内部或低）填充元素开始，以便我们可以使用相同的低填充。Halo 交换发生在 non-dilated/non-padding 基区上，分区 i 所需数据的限制索引可以计算如下：stride × per_shard_window_count × i + window_size − low_pad + dilation − 1dilation,

这决定了正确的光环大小。因为stride × per_shard_window_count 可以被膨胀整除，所以它可以简化为a × i + b，其中a 和b 都是常量。 · stride == 1 但per_shard_window_count 不能被膨胀整除。在这种情况下，不同分区上的低填充是不同的，但它是窗口操作中的静态配置，不能专门针对每个分区来执行 SPMD。在操作数上使用 Pad 和 DynamicSlice 也不起作用，因为这些运算符将在膨胀之前应用，因此所有内容都将乘以膨胀因子。幸运的是，在 stride == 1 的情况下，填充和扩张的基本区域上的所有位置都是有效的窗口开始，我们可以在所有分区上使用最大低填充，以确保每个分区计算所有所需的窗口，然后对分区的输出执行 DynamicSlice窗口操作符删除不必要的数据。分区 i 的非填充基本区域上所需数据的限制索引与以前相同，

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

但不能简化为a × i + b。 · stride = 1 并且stride×per_shard_window_count 不能被膨胀整除。如果上述条件都不成立，则不同的分区可能以不同数量的填充元素开始，并且并非所有偏移都是有效的窗口开始。考虑最后一个例子

图 13. 无论我们选择何种低填充，某些分区都将无效，因为由于步长 = 1，有效窗口可能会被跳过。解决此问题的方法是除了填充基本区域之外还填充窗口。我们可以使用基础区域上分区所需的最大低填充量，然后将窗口大小增加该低填充量。但是，不同分区上窗口的低填充量和高填充量有所不同，这可以通过 Pad 来实现通过 DynamicSlice。窗口填充用于屏蔽基本区域中未对齐的元素，以便非填充窗口元素的开头将与每个分区的基本区域中所需的开头对齐。

窗口膨胀。如果复制 RHS，窗口膨胀仅影响基于 LHS 划分算子时的有效窗口大小。如果扩张的 RHS 也被分区（这通常发生在跨步卷积的梯度计算中），则处理窗口扩张仍然比处理基础扩张更简单，因为 RHS 上没有低/高填充。我们跳过实现的细节。