PIT：通过置换不变变换优化动态稀疏深度学习模型

Ningxin Zheng*、Huiqiang Jiang*、Quanlu Zhang、Zhenhua Han、Lingxiao Ma、
Yuqing Yang Fan Yang、Chengruidong Zhang、LiliQiu、Mao Yang、Lidong Zhou
微软研究院

arXiv:2301.10936v2 [cs.LG] 8 Oct 2023

摘要

动态稀疏性是指直到运行时才确定稀疏模式，这给深度学习带来了重大挑战。由于预处理会产生显著开销，最先进的稀疏感知深度学习解决方案仅适用于预定义的静态稀疏模式。动态稀疏计算要实现高效执行，往往面临两者不匹配的问题：一方面是适合 GPU 高效执行的分块配置，另一方面是能够最大限度减少覆盖浪费（张量中的非零值）的稀疏感知分块形状。

本文提出了 PIT，一种用于动态稀疏性的深度学习编译器。PIT 提出了一种新颖的分块机制，利用经数学证明的置换不变变换（PIT）这一性质，将多个分散分布的微块转换为适合 GPU 高效处理的稠密分块，同时不改变计算结果，从而兼顾高 GPU 利用率与低覆盖浪费。给定一个模型，PIT 首先为其所有算子寻找可行的 PIT 规则，并据此生成高效的 GPU 内核。在运行时，借助新颖的 SRead 和 SWrite 原语，PIT 规则可以极快地执行，从而以在线方式支持动态稀疏性。对多种模型进行的大量评估表明，与最先进的编译器相比，PIT 可将动态稀疏计算加速最高达 5.9 倍（平均 2.43 倍）。

CCS 概念：• **软件及其工程领域** → **动态编译器**；• **计算机系统组织领域** →**神经网络**。

关键词：深度学习编译器、动态稀疏性、动态编译器、变换

* 贡献相同。

允许出于个人或课堂用途免费制作本作品全部或部分内容的数字或纸质副本，前提是这些副本不得用于营利或商业利益，且副本须附有本声明和首页的完整引文。本作品中由作者以外的其他人拥有版权的组件，必须尊重其版权。注明出处后可以摘录。以其他方式复制、再版、发布到服务器或重新分发至列表，均需事先获得明确许可和/或支付费用。请通过 permissions@acm.org 申请许可。*SOSP '23，2023*年*10*月*23*日至*26*日，德国科布伦茨© 2023年版权归所有者/作者所有。出版权已授权给 ACM。ACM ISBN 979-8-4007-0229-7/23/10...$15.00 https://doi.org/10.1145/3600006.3613139

1 引言

张量是深度学习中的关键数据抽象。它提供了一种强大而灵活的方式，用于表示图像、音频和语言句子等内容。深度学习中的计算主要涉及对张量执行操作（例如矩阵乘法）。为了高效执行深度学习模型，人们设计了 GPU 等加速器，以并行执行这些张量操作。GPU 通常包含多个流式多处理器（SM），每个流式多处理器拥有数百个 CUDA 核心，可以同时对张量的不同部分执行算术运算。为了充分利用这些并行处理能力，深度学习编译器使用分块技术，将张量拆分成更小且规则的张量切片，即分块，以便由多个 SM 并行处理。事实证明，分块是张量计算的一项关键优化技术，能够有效利用数据局部性和并行性。

近期研究进展表明，深度学习计算日益呈现稀疏特性，即在含有大量零值的张量上执行操作。除了稀疏模型权重（通常是静态的，且事先已知）之外，越来越多的稀疏模式取决于输入，只有在运行时才能确定，这就是动态稀疏性。例如，大语言模型（如 GPT [16, 50], OPT [66]）呈现出多种动态稀疏性。首先，Transformer模型（以及其他类型的模型）的输入、激活值和梯度中存在固有的动态稀疏性 [46]。激活图中只有极少数元素非零 [44]（例如，T5-Base 为 3.0% [55] , ViT-B/16 [23] 为 6.3%）。其次，人们正利用动态稀疏性进一步扩大深度神经网络模型的规模。现有的大多数参数量超过一万亿的模型都采用混合专家（MoE）结构，根据输入以稀疏且动态的方式激活专家 [29]。此外，动态稀疏训练会在训练过程中动态剪除模型中不太重要的连接，凭借出色的计算效率而受到越来越多的关注 [26, 46]。

在存在动态稀疏性的情况下，高效计算深度学习模型是一项艰巨的挑战。现代深度学习编译器 [9, 20, 71],（包括稀疏性感知编译器 [10, 68], ）会利用编译时已知的深度学习模型中的稀疏模式，来确定合适的内核配置。由于编译时间开销很高，这些编译器无法用于动态稀疏性。一些细粒度稀疏性解决方案可以在运行时执行 [3, 31],

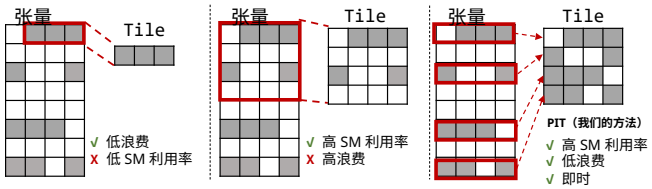


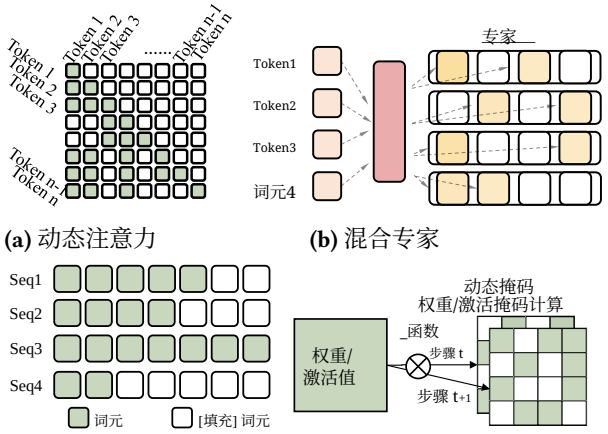
图 1. 动态稀疏性的分块（阴影块表示非零值）。稀疏粒度与高效 GPU 内核的分块形状不匹配。

但它们使用特殊的稀疏格式（例如 CSR：压缩稀疏行），导致格式转换开销很高。

高效的 GPU 计算需要将分块数据加载到共享内存中，以实现数据复用。然而，硬件高效分块的形状通常与多样且动态的稀疏模式不匹配。如图 1 左图和中图所示，尽管与稀疏模式对齐的分块覆盖浪费较低，但由于 SM 利用率和数据复用率低，它们在 GPU 上运行缓慢。但由于稀疏值通常不连续分布，高效 GPU 内核所用的分块形状无法避免覆盖大量零值，导致稀疏计算中的浪费。现有的稀疏性解决方案试图在高效分块与稀疏形状对齐之间找到更好的权衡。它们要么需要编译时信息，要么由于格式转换而在运行时产生显著开销。

本文提出 PIT，这是一种用于高效执行具有动态稀疏性的深度学习模型的编译器。PIT 通过构造由多个稀疏分布的"微块"组成、适用于 GPU 的块（例如图 1 右图），解决分块形状不匹配的问题。该构造在运行时执行，其正确性由"置换不变变换"这一数学性质保证。该性质广泛存在于深度学习算子中，但尚未被充分用于动态稀疏性。大多数深度学习算子有一个或多个维度（我们称之为 PIT 轴），其计算顺序可以任意重排而不影响结果。例如，对于张量表达式为 $C[m,n]$+=$A[m,k] \times B[k,n]$ 的矩阵乘法（Matmul），A 的列与 B 的行（即 k 维度）可以按任意顺序排列，而不影响计算结果。A 的行与 C 的行（即 m 维度）也可以重新排列而不影响计算。通过沿某个 PIT 轴将微块合并成稠密分块，PIT 兼得两方面优势：既能利用 GPU SM 上稠密分块的高效执行，又能通过微块对非零元素进行细粒度覆盖，从而减少无效计算。

利用排列不变变换的一项关键挑战
换的运行时转换开销。PIT 通过引入*SRead* 和*SWrite*来解决这一挑战。这两个数据重排原语用于在 GPU 全局内存中稀疏分布的数据与共享



(a) 动态注意力 **(b)** 混合专家

(c) Dy态稀 q列长度 gth **(d)** Sp疏训练

练**图2.** 深度学习中动态稀疏性的示例。

内存之间加载和存储数据。通过将数据重排搭载在标准张量计算过程中跨存储层次结构的数据移动上，*SRead* 和*SWrite* 几乎无需额外开销即可实现排列不变转换。即使张量中稀疏值的坐标未知，PIT 也能以在线方式快速检测它们。有趣的是，通过将其视为动态稀疏性的一种特殊情况，PIT 也能有效处理具有静态稀疏模式的深度学习模型，因为分块形状未对齐同样会造成效率低下。

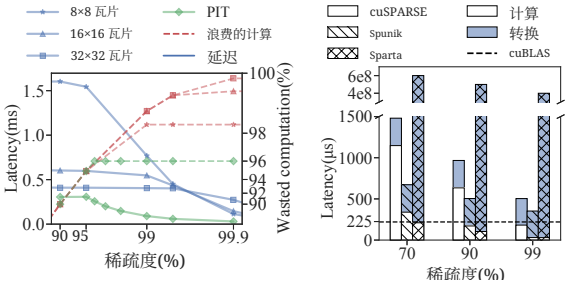
我们在五种代表性模型（即 Switch Transformer [63],OPT [66],BERT [22], Long- Former [14], Museformer [65]）上对 PIT 进行了全面评估，发现它在减少内存占用的同时，显著加快了推理和训练速度。在推理方面，与最先进的解决方案相比，PIT 的加速最高可达 5.9 倍，且内存消耗更少（§5.1）。此外，我们还将 PIT 用于加速大语言模型训练和稀疏训练。与先前的解决方案相比，PIT 在 OPT训练中最高可加速 1.8 倍，在稀疏训练中最高可加速 2.4 倍。动态稀疏性在深度学习中的重要性日益提高，我们计划将 PIT 作为开源软件发布，以推动动态稀疏性优化和算法方面的进一步研究。

2 背景与动机

2.1 动态稀疏性

随着深度学习模型日益加深、规模不断扩大，参数、激活值和梯度趋于接近零或变为零，从而产生模型稀疏性。随着大语言模型（LLMs）的快速发展，越来越多类型的稀疏性呈现出动态特征，只有在运行时才能确定，这已成为一种趋势。

大语言模型中的动态稀疏性。动态注意力使模型只需对数据中信息量最大的部分进行计算 [29, 33, 34, 37, 38, 45, 57, 60, 69, 70, 72]。如图所示



(a) 不同分块尺寸的延迟和计算浪费。**(b)** 转换开销。**图3**。 由于计算浪费和稀疏格式转换开销高，动态稀疏性的分块效率低下。

图2a中，模型可以屏蔽无关词元，以更少的计算实现更高的准确率。掩码会即时生成。我们的评估还显示，三种热门大语言模型（OPT、Switch Transformer 和 T5）的激活值稀疏率为 95%–99.9%（即零值所占的百分比），跳过这些零值不会影响模型准确率（参见 §5.1）。

稀疏训练。 稀疏训练至关重要，并广泛应用于深度学习的许多领域，例如动态稀疏训练（DST） [26], MAE [35], 剪枝 [40], 和超网络训练[18, 19]。图2d展示了训练过程中动态剪枝的一个示例。该算法会根据当前模型状态屏蔽部分权重。训练期间，每一步的掩码都在不断变化。

混合专家（MoE）。 MoE是一种具有动态稀疏性的神经网络架构，已广泛应用于流行的大型模型。大多数参数超过一万亿的大语言模型都采用MoE来扩展模型 [24, 29, 42]。如图2b所示，词元序列会经过一个门控函数，由该函数动态地将每个词元分配给一个或多个专家。每个专家只处理其专长范围内的一部分词元，并屏蔽未被路由至该专家的其他词元。因此，每个专家的计算都是稀疏的，并取决于输入，而输入只有在运行时才可知。
动态序列长度。 在自然语言处理 [22, 51, 55] 和多模态模型 [53, 56], 中，词元序列的输入和输出长度自然各不相同。要批量处理这类序列，需要将它们填充到相同的序列长度（通常是批次中的最大长度），如图2c所示。这种填充会造成计算浪费，可视为动态稀疏性。

2.2 动态稀疏性导致的低效

深度学习编译器中的分块。TVM、 [20],Triton [9],Roller [71], 等深度学习编译器通常采用分块技术，将张量切分成更小的块。通过重复利用缓存中的块，分块可以减少需要从动态随机存取存储器等较慢内存传输的数据量。通过调整分块形状（例如 32×32 或 16×64），编译器可以针对特定硬件架构上的模型计算优化数据复用，从而提升内核性能。**动态稀疏性下低效的分块**。尽管分块对传统稠密模型很有效，但在动态稀疏场景下可能效率低下。如图 1 所示，与稀疏模式对齐的分块形状能最大限度地减少非零值的覆盖范围，但在图形处理器上执行时效率不高。相反，适合 GPU 高效执行的块会造成浪费（覆盖过多零值）。图 3a 使用从 OPT [66] （一种大语言模型）提取的实际稀疏激活，展示了在不同稀疏率下执行稀疏矩阵乘法时，采用各种分块形状调优的 GPU 内核性能。当稀疏率低于 99.6% 时，尽管 32×32 瓦片覆盖了最多的零值、浪费最大，但效率更高。只有在稀疏度非常高时（本例中为 >99.9%），8×8 瓦片才更快，因为与其他分块形状相比，它节省的计算量更多。不同的分块形状都面临着两难选择：既要让块的覆盖方式适应稀疏性，又要保证 GPU 高效执行。
稀疏性感知编译器或库。 一些稀疏性感知编译器或库尝试通过编译专用 GPU 内核（例如 SparTA [68]），或将数据转换为特殊格式（例如 cuSPARSE [3],Spunik [31]）来打破这一困境。然而，这些方法会带来显著开销，降低运行时性能。图3b比较了 SparTA、cuSPARSE 和 Spunik 在处理动态稀疏性时的转换开销（编译或格式转换）。SparTA 编译专用内核需要 400–600 秒，因而不适合处理运行时发生变化的稀疏模式。虽然 cuSPARSE 和 Spunik 可用于动态稀疏性，但它们都面临较大的转换开销。因此，当稀疏率较高时，整体性能甚至不如直接使用稠密计算，处理动态稀疏性的效率也就很低。

2.3 机遇

图 1 右侧展示了，可以并行加载分散在不同位置的数据，并将其合并为适合计算的形状的稠密分块。这种稀疏到稠密转换不会影响结果的正确性。图 4 展示了矩阵乘法的两个示例。第一个示例将稀疏张量 A 与稠密张量 B相乘。阴影区域包含非零值，其余部分均为零。通过将 A 中的非零行重新排列到一个新张量 A'中，就可以在 A' 与 B'（ B的非零稠密分块）之间执行稠密矩阵乘法，得到结果张量$C' = A' \times B'$。将 C' 中的行写回 C 中原来的行后，我们得到相同的结果 $C = A \times B$。请注意， A的非零行可以按任意排列方式重新排列，而不会影响 C'中各行的正确性。 C' 中的行会按照逆排列写回 C，

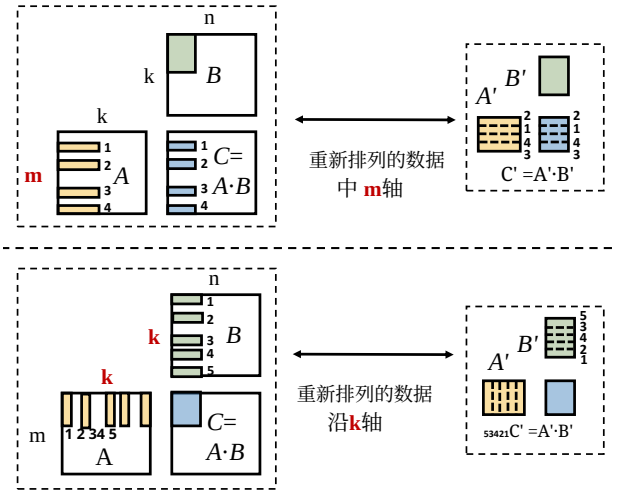


图4。稀疏矩阵乘法示例。通过沿某一轴重新排列稀疏数据，稀疏计算可以等价地通过稠密矩阵乘法完成。

$A \rightarrow A'$ 以恢复正确的行索引。第二个示例展示了两个稀疏矩阵A 和 B的矩阵乘法。通过重新排列 A和 B 在维度 k 上的数据（即 A 的列和 B的行），可以类似地使用两个稠密矩阵（即 A' 和 B'）之间的矩阵乘法来完成计算。如图 3a 所示，稠密瓦片的 GPU 效率更高。如果图 4 中的数据重排在运行时开销可以忽略不计，那么动态稀疏张量计算就可以使用低浪费且 GPU 效率高的稠密瓦片来执行，从而实现卓越的性能。这促使我们设计 PIT，以系统地利用深度学习算子的排列不变转换，高效支持动态稀疏性。

3 PIT设计

PIT 是一种旨在应对动态稀疏性所带来挑战的编译器框架。PIT 的核心是其置换不变变换（简称 PIT变换），它能够以在线方式将稀疏张量转换为计算效率高的稠密格式。图 5 展示了 PIT架构。PIT 引入了微瓦片这一数据单元，其尺寸为可用的最小值，用于组合成更大的硬件友好型瓦片，以实现高效计算。对于给定的稀疏张量，PIT变换策略会从所有可行的微瓦片中选出效率最高的一个；这些微瓦片由 PIT 规则推导而来，且对应于在数学上等价的深度学习算子计算变换。随后，稀疏内核生成器基于选定的微瓦片生成稀疏内核。为处理动态稀疏性，内核会接收稀疏数据以及包含非零值的微瓦片索引，以便进行正确计算。稀疏性检测器会在线构建索引。PIT 的特性使稀疏性检测和索引构建能够充分并行，从而最大限度地减少线上执行开销。在稀疏内核中，SRead 和 SWrite 会根据构建的索引将稀疏数据重新排列为稠密格式，随后通过高效的基于稠密瓦片的计算进行处理。

3.1 PIT变换机制

微块。微块是一种小型数据单元，其形状与加速器（例如 GPU）低级内存的读写事务粒度相匹配。微块使对稀疏数据的访问与对稠密数据的访问同样高效。例如，CUDA图形处理器中全局内存的读写事务为32字节，因此此类加速器上最小的微块大小为 1x8 float32（或 1x4 float64），其粒度足够细，能够适用于许多稀疏模式。例如，在图5中，输入张量的非零值粒度为 1x1、1x2、1x3 和 1x4。假设读写事务的大小为 1x4，则可使用 1x4 微块进行覆盖。这样，微块在计算效率和覆盖浪费之间实现了良好的权衡。

高效的稠密分块已与 GPU 共享内存的数据访问模式（例如，最大限度减少银行冲突 [71]）对齐，并通过经过充分优化的线程束调度 [48] 使计算核心达到饱和。通过将位置稀疏的微瓦片转换为稠密计算瓦片所需的数据格式，稀疏算子的计算便能与 GPU 的各个组成部分——包括全局内存、共享内存和计算核心——充分对齐，从而实现高效率。稀疏计算瓦片与稠密计算瓦片之间的转换利用了置换不变变换这一特性，使稠密分块能够正确处理重新排列后的微瓦片。深度学习算子中普遍存在这一特性，将在 §3.2 中详细阐述。

图 6 展示了稀疏算子中微瓦片的定义。它包含算子的输入/输出以及稠密计算瓦片的微块尺寸，微瓦片将映射到该稠密计算瓦片上。属性 TileInputFormats 和 TileOutputFormat 分别表示 DenseTileImpl 所需的输入和输出数据格式（即稠密块形状）。变换策略（§3.2）决定这些信息，并在后续用于稀疏内核生成。

SRead 和 SWrite。与稠密张量计算不同，稠密张量计算中的块会被连续加载、处理和存储，而 PIT 生成的稀疏内核则使用 SRead 和 SWrite 在微块级别处理稀疏数据。如图 5 右侧所示，SRead 和 SWrite 这两个基本操作会在线重排输入张量中的微块，以分别将数据整理为瓦片输入格式，并将数据以瓦片输出格式写入输出微块。数据重排与跨不同内存层级的数据移动同步进行，因此几乎不会产生额外开销，并且无需在稀疏内核之外进行传统的数据重排

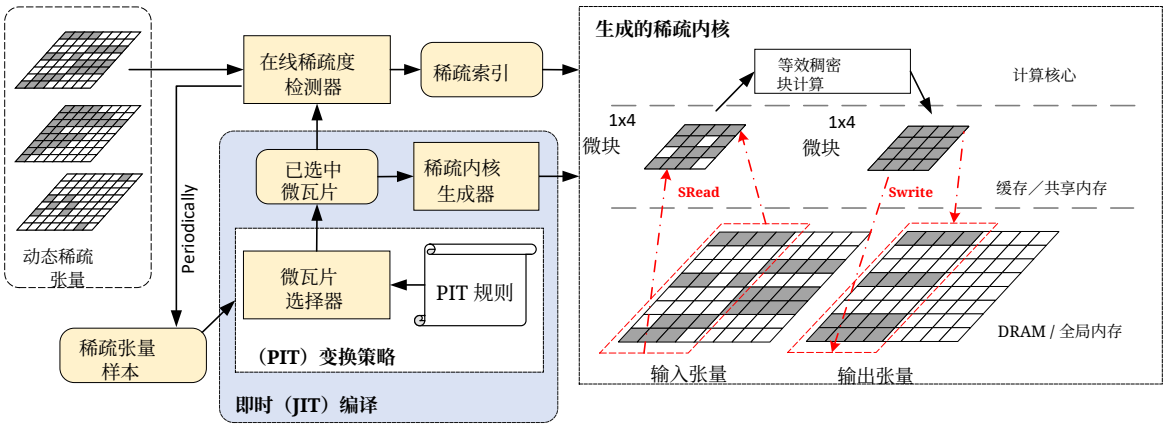


图 5. 架构概览：PIT 使用即时编译，为动态稀疏模式生成稀疏 GPU 内核。稀疏 GPU 内核使用 SRead，将输入张量中的微瓦片（在运行时检测）加载到共享内存中的稠密计算瓦片。完成等效稠密块计算后，它使用 SWrite 将输出微块写入输出张量。SRead 和 SWrite 仅在 PIT 轴上执行，PIT 保证其数据重排的正确性。

```
类 MicroTiledOp:
    # 数据    格式位于全局内存中,              A 是稀疏的
    输入微块大小                    # 1x4（针对 A）, None 用于 B
    输出微瓦片大小                  # C 不适用
    # 数据    格式位于共享内存
    瓦片输入格式                    # 4×4 密集格式   用于 A 和 B
    瓦片输出格式                    # 4×4 密集格式   用于 C
    DenseTileImpl                   # C[4x4]=A[4x4]*B[4x4]
    def 生成内核 ()
```

图 6. 稀疏算子的微块定义。注释标出了图 5 右侧示例微块的各项属性值。

```
/* 生成的稀疏内核*/
__global__ void 稀疏内核模板 (
    struct 张量 输入          , struct 稀疏索引 输入索引   ,
    struct 张量 输出          , struct 稀疏索引          输出索引 ,
){
    /* 首先分配共享内存 */
    输入瓦片块   =AllocSharedM ( 瓦片输入格式 );
    输出瓦片块   =AllocSharedM ( 瓦片输出格式 );
    SRead (输入 ,      输入瓦片块 ,     输入索引 );
    DenseTileImpl ( 输入瓦片块 , 输出瓦片块 );
    SWrite ( 输出瓦片块 ,     输出 ,      输出索引 );
}
```

图 7. 含有 SRead 和 SWrite 的稀疏内核模板。

（例如，为稀疏内核构造 CSR 格式 [17] ）。

图 7 展示了 PIT 稀疏内核的模板。该内核包含两个阶段：使用 SRead 和 SWrite 进行数据排列，以及在稠密块上进行计算。SRead 和 SWrite 的数据重排都需要快速在线构建微块索引（即输入索引、输出索引）（§3.3）。索引按照图 6 中定义的指定微瓦片形状构建。这种设计

算子	张量表达式	PIT 轴
归约求和	$C[p] \mathrel{+}= A[p,l]$	p, l
向量加法	$C[p] = A[p]+B[p]$	p
矩阵乘法	$C[m,n] \mathrel{+}= A[m,k]*B[k,n]$	~~m, n, k~~
批量矩阵乘法	$C[b,m,n] \mathrel{+}= A[b,m,k]*B[b,k,n]$	b, m, n, k
卷积	$C[n,f,x,y] \mathrel{+}= A[n,m,x+i,y+j]*B[f,m,i,j]$	n, m, f

表1. 常用算子的张量表达式，以及支持打乱其索引而不影响正确性的 PIT 轴。

有效地将稀疏内核中的数据编码/解码与计算分离，开创了一种新颖的稀疏计算范式，将数据重排与（稠密）计算瓦片相结合。

3.2 PIT策略

PIT 定义了一系列作用于特定瓦片轴的规则，可将沿该轴的微瓦片正确转换为适合 GPU 的稠密瓦片。具体而言，一条 PIT 规则由 PIT 轴、微瓦片形状和稠密计算瓦片组合而成。系统遵循 PIT 规则，在 PIT 轴上执行 SRead/SWrite，将该轴上多个稀疏分布的微瓦片加载到稠密计算瓦片中，或从中写出。一条 PIT 规则确保在 PIX 轴上的计算必须满足置换不变性 [41]。

为便于说明，我们使用爱因斯坦求和（ein- sum）表示法 [25] 来表示沿张量轴执行的操作。表1列出了一些深度学习中的常见算子及其对应的 einsum 表示法。当且仅当对某个轴上的索引进行任意打乱都不会影响算子的正确性时，einsum 表示法中的该轴才是 PIT 轴。以下定理给出了算子的所有 PIT 轴。

定理1。 当且仅当某个轴上的所有计算均可交换且可结合时，该轴才称为 *PIT* 轴。

该定理基本显而易见，受篇幅所限，我们省略证明。可交换性和可结合性保证了在 PIT 轴上随机打乱微块的正确性，并允许按任意顺序并行处理微块。首先，生成新轴的轴不是 PIT 轴。例如，卷积算子（表1）中的 x、 i、 y 和 j 轴不是 PIT 轴，因为它们会生成新轴（"$x+i$"和"$y+j$"），因此不满足可交换性。在其余轴中，有两类轴属于 PIT 轴。输出张量中的轴称为 空间轴。所有空间轴都是 PIT 轴，因为它们只会改变数据布局。不在输出张量中的轴称为 归约轴，其计算大多可交换且可结合（例如，求和、乘法、最大值、最小值），因此也属于 PIT 轴。表1还总结了所列算子的 PIT 轴。对于模型使用的每个算子，PIT 都会根据轴的类型，以及其是否涉及不可交换或不可结合的计算，通过该算子的张量表达式找出所有 PIT 轴。我们发现，仅使用一个 PIT 轴就足以涵盖深度学习中的大多数动态稀疏性。尽管我们确实识别出了更复杂的 PIT 规则，例如，对 BatchMatMul 中的 (b, m) 轴或 (b, n) 轴进行跨多个轴的排列，但受篇幅所限，我们将其留待未来工作。

微瓦片与内核选择。 PIT 会创建一个稀疏内核数据库，其中每个稀疏内核都会对某个算子的一个 PIT 轴应用 PIT变换。每个稀疏内核都定义了一种微瓦片形状，该形状由 PIT 轴以及其用于计算的稠密核的分块形状决定。当稀疏张量的内存布局在 PIT 轴上不连续时，我们将微块在 PIT 轴上的形状设为 1，同时保持其他轴上的形状与稠密核的分块形状相同。这样，GPU 就能并行加载/写入 PIT 轴上的值，从而充分利用内存事务。例如，考虑一个分块大小为 $[M, K] \times [K, N]$ 的稠密矩阵乘法内核，假设第一个输入张量是稀疏的，并以行主序内存布局存储（即在内存中沿 K 轴连续）。如果 M 是 PIT 轴，则微瓦片大小为 $[1, K]$。如果稀疏张量的内存布局在 PIT 轴上连续，我们就需要先更改其格式，使数据在 PIT 轴上变为非连续，例如从行主序改为列主序，以充分利用内存事务。这可以搭载在生成该稀疏张量的前序算子的输出过程中完成，因此开销可以忽略不计。

算法 1： 动态稀疏性的内核选择算子。

数据： Op：一个动态稀疏算子，
D_{sparse}：Op的 n 稀疏性样本列表。
结果： $Best$：Op的最佳计算瓦片。

```
1  函数 内核选择(D_sparse, Op):
2      Best = 空值; Cost_optimal = 无穷大;
3      遍历 T ∈ GetTilesFromTileDB(Op) 执行
4          遍历 A ∈ GetPITAxis(Op) 执行
5              Cost = 0;
6              micro_tile = GetMicroTile(T.SparseTensor, A);
7              遍历 D ∈ D_sparse 执行
8                  Num_tiles = CoverAlgo(D, micro_tile, A);
9                  Cost += Num_tiles * T.tile_cost;
10             如果 Cost < Cost_optimal 则
11                 Best = S;
12                 Cost_optimal = Cost;
13     返回 Best;
```

算法 1 showshowPIT 以即时编译的方式选择合适的 PIT轴、微瓦片形状和稠密计算瓦片，以生成稀疏内核。PIT 会遍历所有稠密计算瓦片以及算子的 PIT轴（第 3–4 行）。GetTilesFromTileDB 返回所有可用的稠密计算瓦片形状，这些形状对应着高效的 GPU 内核，通常由现有的深度学习编译器或实现提供，例如 TVM [20], OpenAI Block Sparse [6]。GetPITAxis 返回我们在定理1中定义的算子的所有可行 PIT轴。对于每种稠密计算瓦片和 PIX轴，GetMicroTile 会按照上文所述，找到有效的微瓦片形状（第 6 行）。CoverAlgo（第 8 行）会计算覆盖稀疏张量中所有非零值所需的微块数量（记为 Num_{tiles}）。生成的稀疏内核的时间成本估算为 $Num_{tiles} \times T.tile_cost$，其中 $T.tile_cost$ 是通过离线分析获得的对应稀疏内核运行时间。PIT 会选择时间成本最低的稀疏内核及其对应的微瓦片来执行稀疏计算。对于稀疏率相对较低的输入，该内核选择算法使 PIT 能够无缝回退到稠密计算。

瓦片开销的离线分析十分轻量。由于 PIT 会即时沿某个 PIT 轴将微块合并为稠密分块，因此离线分析可以采用不依赖模型、张量形状和稀疏模式的方式进行。PIT 只需记录稠密计算中不同分块形状（例如 32x32 和 64x64）的执行时间。因此，每种算子和 GPU 类型只需进行一次离线分析，与长时间运行的推理服务相比，开销非常轻量。

3.3 在线稀疏性检测

高效的稀疏计算需要线上检测不断变化的稀疏模式，并且只计算稀疏张量中的非零值。这意味着必须即时构建非零值的索引。然而，即时创建稀疏索引可能颇具挑战。

如图3b所示，cuSPARSE 的转换开销可能远高于计算时间，尤其是在稀疏度较高时。

我们提出了一种有效的线上索引构建机制，即以微瓦片 为粒度，以无序 方式构建稀疏索引。首先，借助选定的微瓦片，PIT 在该微瓦片粒度上检测非零值，从而大幅缩小稀疏索引的规模。其次，PIT 变换使稀疏度检测和索引构建能够以乱序方式高度并发地进行：当某个轴经过置换时，PIT 轴允许 PIT 执行计算（§3.2）。经过这种变换，PIT 不再需要沿特定轴构建有序索引。这大幅减少了基于微瓦片的索引构建过程中的限制，并将加速器线程间的同步开销降至最低。

具体而言，索引构建任务在加速器（例如 GPU）上运行，并被划分为多个块。对于每个块，任务都会遍历稀疏张量的一个区域，并检查其中是否包含非零值微块。检测到非零值微块后，其索引（即在张量中的偏移量）会写入预先分配的索引数组。由于多个块会并发地将非零值微块的索引写入索引数组，因此它们会使用原子加法来确定数组中的唯一位置，以记录这些索引，从而确保更新安全。因此，由于线程块的调度顺序不可预测，最终微块索引的排列是无序的。

此外，与现有的稀疏解决方案（如 cuSPARSE、MegaBlocks）不同，PIT 构建稀疏索引时不会改变稀疏张量的存储格式。在运行时，稀疏内核中的 SRead 和 SWrite 使用该索引，以微瓦片为粒度，直接从原始稀疏张量中加载非零值并将其重新排列回原始稀疏张量。这显著降低了数据格式转换（例如，将稠密张量中的稀疏数据转换为 CSR 格式）带来的内存访问开销，并实现了零拷贝数据重排。

4 实现

我们实现了 PIT，并将其集成到 PyTorch [52]中。它包含约 13,000 行 C++ 和 CUDA 代码，以及 5,600 行 Python 代码。PyTorch 是一种流行的开源深度神经网络框架，支持多种动态稀疏算法。通过将这些 PIT 变换规则应用于 500 多个稠密计算内核，PIT 生成了约 1,500 个稀疏内核。这些稠密计算内核包括手动优化的内核（例如 OpenAI Block Sparse [6]）、由硬件指令加速的内核（即 wmma [4]），以及由 TVM [20] 等编译器生成的稠密内核。这些稀疏内核存储在数据库中，并预先创建了性能查找表。随后，利用这些经过性能分析的数据来指导线上微瓦片选择。尽管离线

模型	数据集	模型结构	精密器件	
Switch Transformer[29]	MNLI [59]	编码器 解码器 MoE	半精度浮点数，单精度浮点数	A100
Swin-MoE [37]	ImageNet	编码器 MoE	fp16	A100
OPT [66]	Alpaca [58]	解码器	fp32	V100
BERT [22]	GLUE [59], 新闻 [27] 等。	编码器	fp32	V100
Longformer [14]	arXiv [21]	编码器	fp32	V100
Museformer [65]	LMD [54]	解码器	fp32	V100

表2. 评估中的模型和数据集 n.

性能分析需要数小时，但只需进行一次，并且可以通过在多个设备上并行进行性能分析来加速。

凭借动态稀疏优化的可扩展性，PIT 已支持 31 种自然语言处理（NLP）模型。其中，13 种是大语言模型，包括 OPT、T5 和 Switch Transformer。此外，PIT 还支持 8 种动态稀疏注意力模型、8 种混合专家模型和 13 种稀疏训练算法。PIT 易于集成，可将代码修改控制在极少范围内——在所有评估场景中，修改的代码均少于 10 行。这一特性使现有模型能够迅速适配动态稀疏优化，从而提升用户效率。

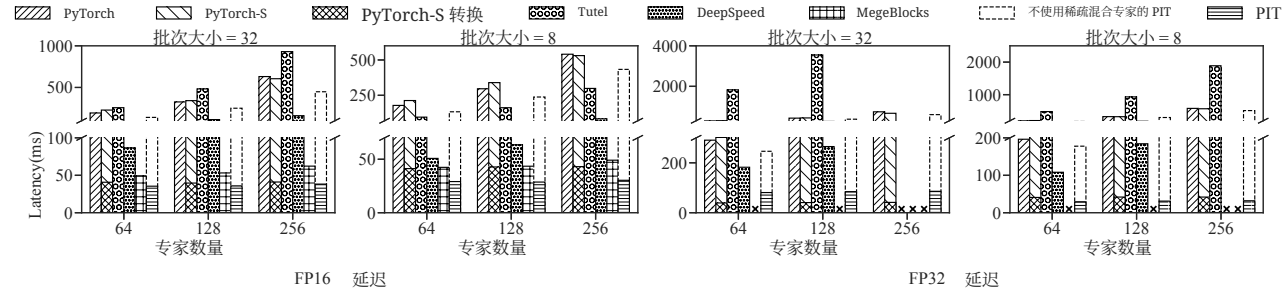
5 评估

本节将通过全面的实验，从多个角度展示 PIT 的有效性。具体而言，我们首先在 A100 和 V100 上使用六种有代表性的模型评估 PIT 的端到端推理性能（见 §5.1）。接着展示 PIT 在两种不同训练场景下的端到端性能（详见 §5.2）。此外，我们还进行了微基准测试，以突出 PIT 变换的有效性（§5.3）和转换开销（§5.4）。最后，我们展示微瓦片在线搜索（§5.5）以及检测稀疏模式变化（§5.6）的有效性。总之，我们的结果表明：

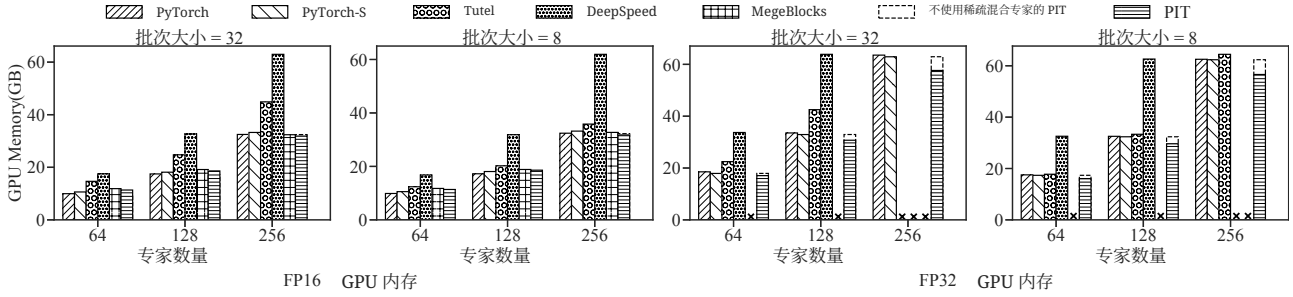
• PIT 在五个具有代表性的模型上显著降低了推理延迟并减少了内存占用，其性能分别超越 PyTorch、PyTorch稀疏[1]、Tutel、DeepSpeed、MegaBlocks 和 TurboTransformer，最高分别达到 18.1 倍、17.8 倍、59.1 倍、5.9 倍、1.6 倍和 1.9 倍（§5.1）。

• PIT 还能显著提升训练效率。与最先进的解决方案相比，PIT 在 OPT 训练中最高可实现 1.8 倍加速，在稀疏训练中最高可实现 2.4 倍加速（§5.2）。

[1]PyTorch 中封装的最先进稀疏内核。



(a) 延迟



(b) GPU 内存

图 8. Switch Transformer 的每批次端到端延迟和内存占用。

• 通过 PIT 变换，PIT 的性能优于最先进的稀疏性优化方案。具体而言，与 cuSPRARSE、Sputnik、OpenAI Block Sparse 和 SparTA 相比，PIT 分别最高可实现 88.7 倍、5.8 倍、17.5 倍和 5.7 倍加速（§5.3）。

• PIT 可以在线检测动态稀疏性，且开销可以忽略不计；在在线构建稀疏索引时，相较于此前最先进的方法，速度最高可提升 4.7 倍（§5.4）。

5.1 端到端推理

如表2所示，我们针对六种具有代表性的模型，将 PIT 与最先进的稠密和稀疏基线方法进行比较，评估其推理延迟和内存使用量。基线方法包括最流行的深度学习框架（PyTorch v1.11.0）、两个针对大规模模型优化的推理框架（DeepSpeed [11] 和 TurboTransformers [28]），以及针对特定模型的优化技术（用于混合专家模型的 Tutel 和 MegaBlocks [30]，以及用于 Longformer 模型的 Longformer-S）。我们还创建了 PyTorch-S，这是 PyTorch 的一个变体，采用来自 cuSPARSE (v11.6) [3]、Sputnik [31], 和 Triton [9] 的性能最佳稀疏内核。我们为每个模型从这些稀疏内核中选取最佳结果，作为 PyTorch-S 的最终性能。由于缺少算子，TurboTransformers 仅支持 BERT 模型，无法运行其他模型。**Switch Transformer**。我们评估了 PIT 在 Switch Transformer 上的性能。Switch Transformer 是一种同时包含编码器、解码器和混合专家结构的大语言模型。我们在 1x A100-80GB GPU 上使用不同精度（即 float32 和 float16）测量了 PIT 的延迟和内存使用量。Switch Transformer 的混合专家层会将每个词元分配给一位专家，这可能导致专家之间的词元分布不均。我们将 PIT 与几种基线方法进行比较：PyTorch 按顺序执行各位专家；Tutel 和 DeepSpeed 使用批量矩阵乘法融合所有专家，以并行执行；MegaBlocks 则利用稀疏内核，在将词元重新组织为稀疏格式后同时执行所有专家。MegaBlocks 仅提供 float16 精度的 GPU 内核，因此未在 float32 下进行评估。

在非MoE层中，PIT优化了同一批次内因序列长度变化而产生的动态稀疏性。在MoE层中，PIT使用 SRead 加载每个专家所需的词元，对分配给这些专家的词元进行稀疏计算，并使用 SWrite 将结果直接写入对应位置。我们使用 GLUE [59] 中的 MNLI数据集评估了 PIT。图 8 展示了不同批次大小和专家数量下 Switch Transformer 的端到端推理延迟和内存开销。如图 8a 所示，PIT 允许所有专家进行稀疏计算，避免了计算浪费，因此优于其他方法。与 PyTorch、PyTorch-S、Tutel 和 DeepSpeed 相比，PIT 在精度为 float32 时分别实现了 3.6x~18.1x、3.7x~17.8x、16.6x~59.1x 和 2.3x~5.9x 的加速。精度为 float16 时，与 PyTorch、PyTorch-S、Tutel、DeepSpeed 和 MegaBlocks 相比，加速比分别为 5.5x~17.8x、6.4x~17.5x、3.3x~24.3x、1.8x~4.2x 和 1.4x~1.7x。与 Tutel 和 DeepSpeed 相比，PIT 避免了 BatchMatMul 带来的计算浪费，因为 BatchMatMul 要求将所有专家的输入填充到相同长度。此外，与 MegaBlocks 相比，PIT 使用 SRead 和 SWrite 消除了

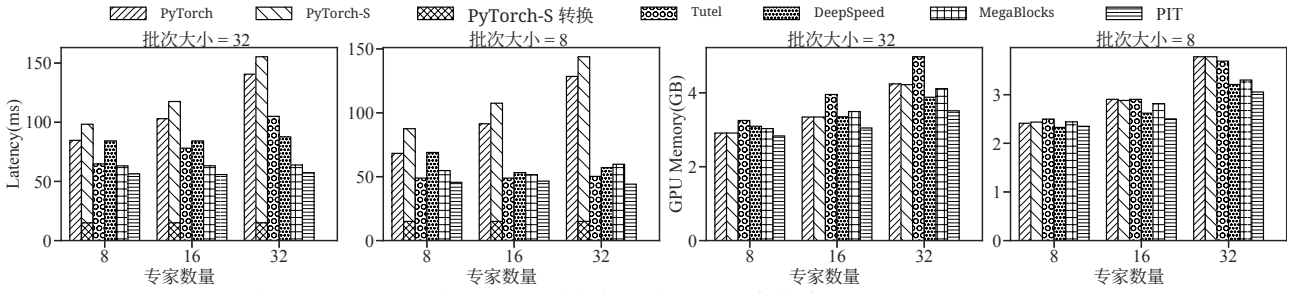


图 9. SwinMoE 在 A100 上的端到端延迟和内存占用。

输入准备过程中高昂的数据重组开销。“ PyTorch-S Convert”突显了 PyTorch-S 构建稀疏索引的开销。尽管 PyTorch-S 中的计算速度有所提升，但构建稀疏索引的成本抵消了速度收益。为了剖析稀疏 MoE 和序列长度变化情况下的收益，我们还评估了未应用动态稀疏 MoE 优化的 PIT（即图 8 中的“不使用稀疏混合专家的 PIT”）。我们发现，PIT 在 Switch Transformer 上的性能提升主要来自对 MoE结构中动态稀疏性的优化。我们还评估了图 8b 所示的 GPU 内存使用量。与基线方法相比，PIT 的内存使用量最低。由于批次大小和专家数量增加时填充过多，Tutel 和 DeepSpeed 会出现内存溢出（OOM）。

Swin-MoE 我们评估了 PIT 在 Swin-MoE 上的性能。Swin-MoE 是一种同时具备编码器和混合专家结构的大型视觉模型。我们在 float16 精度下测量 PIT 在 A100 上的延迟和内存使用量。对于视觉 Transformer，同一批次中的输入图像会被重新缩放至相同分辨率，以确保序列长度一致。在我们的实验中（如图 9 所示），我们比较了不同批次大小和专家数量下 Swin-MoE 的端到端推理延迟和内存占用。MegaBlocks 优于其他基线方法，这是因为它能同时执行所有专家，并通过高效利用稀疏内核避免计算浪费。与 MegaBlocks 相比，PIT 通过在内存层次结构间的数据移动过程中搭载数据重组，进一步提升了性能。与 PyTorch、PyTorch-S、Tutel、DeepSpeed 和 MegaBlocks 相比，PIT 分别实现了 1.5x~6.3x、1.5x~2.9x、1.1x~1.8x、1.2x~1.6x 和 1.1x~1.4x 的加速。PIT 在 Swin-MoE 上的性能提升低于在 Switch Transformer 上的提升，这是因为 Swin-MoE 的专家数量明显少于 Switch Transformer。因此，当专家数量从 8 增至 32 时，混合专家层仅占端到端延迟的 23.6% 至 61.2%。单独比较混合专家层的延迟时，PIT 比 MegaBlocks 快约 1.2x~1.7x。出于类似原因，如图所示，PIT 的 GPU 内存使用量与基线方法相近

图 9 右侧。

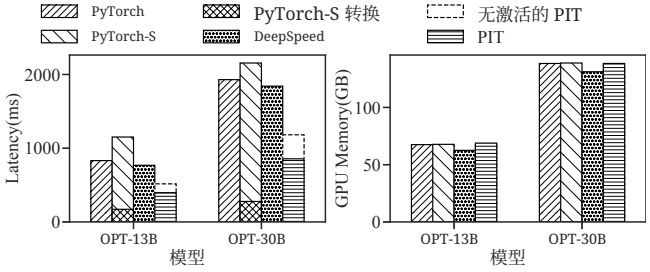


图10. OPT 的每批次端到端延迟和内存占用。

OPT 是一种仅含解码器的大语言模型 [66]。我们使用 Alpaca [58] 数据集，在八块 V100-32GB GPU 上评估了两个版本，分别包含 130亿和 300亿个参数。PIT 对 OPT 应用了两种动态稀疏优化：（1）消除同一批次中长度不一的句子所产生的填充开销；（2）利用 FFN 层中 ReLU 激活产生的细粒度稀疏性（最高可达 99%）。批次大小设为 32。PyTorch-S 使用 Triton 作为后端。

图10比较了 OPT 的端到端延迟和内存占用。PIT 相比 PyTorch、PyTorch- S 和 DeepSpeed，分别实现了 2.1x~2.3x、2.5x~3.0x 和 2.0x~2.2x 的性能提升。避免动态序列中的填充这一优势，使 PIT 相比基线方法（即图10中的 PITw/o activation）实现了 1.6x~1.7x 的加速。通过进一步利用 FFN 层中 ReLU 激活的动态稀疏性，PIT 又将性能提升了 1.3x~1.4x。与使用块大小为 32x32 的 Triton 分块稀疏内核的 PyTorch-S 相比，PIT 使用较小的微瓦片（即 1x32）并配合 SRead 和 SWrite 执行高效计算，从而避免计算浪费。此外，PyTorch-S 还受到稀疏格式转换开销的影响，因此延迟最高。在内存消耗方面，DeepSpeed 的内存使用量最低，因为它将整个编码器层融合为一个算子，并节省了激活内存。PIT 的内存占用与其他基线方法相近。

BERT。 我们还在 BERT-base [22], 这一仅含编码器的语言模型上测试了 PIT，使用 Float32，并在单块英伟达 V100- 32GB 上运行。在 BERT 中，PIT 仅优化由同一批次中不同序列长度导致的动态稀疏性

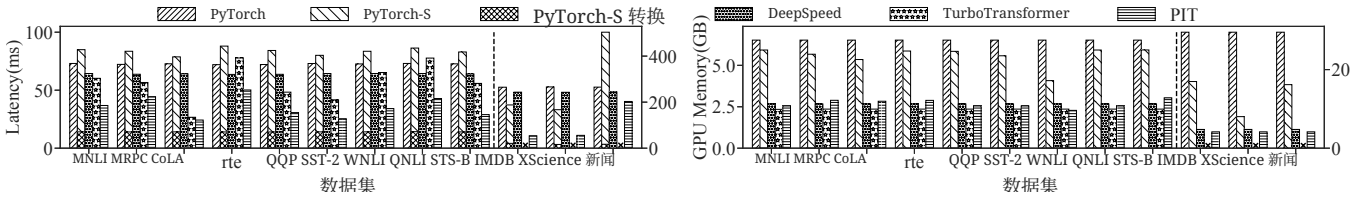


图 11。BERT 在 V100 上的端到端延迟和内存占用。

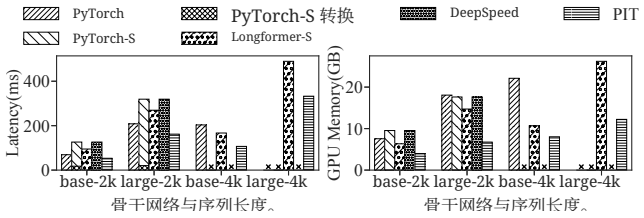


图 12。 Longformer 在 V100 上的端到端推理延迟和内存占用。

。为更全面地评估其性能，我们在多个数据集上进行了实验，包括 GLUE [59]（含 MNLI、MRPC、CoLA、RTC、QQP、SST-2、Wnli、QNLI、STS-B）、IMDB [49]、Multi-XScience [47], 和 Multi-News [27]。我们将 Turbo-Transformer [28] 作为基线，该推理框架通过智能动态批处理专门针对可变输入序列长度进行了优化。我们使用的批次大小为 32。PyTorch-S 使用 Triton 作为稀疏后端库。

如图 11 所示，PIT 的性能优于 PyTorch、PyTorch-S、DeepSpeed 和 TurboTransformer，分别快 1.3 至 4.9 倍、1.8~3.5 倍、1.2~4.5 倍和 1.1~1.9 倍。当序列长度较短时（例如，GLUE 中不超过 128 个词元），PyTorch-S 的表现较差。由于其后端（Triton）要求采用粗粒度稀疏度（即 Triton 内核中的 32 个词元），PyTorch-S 需要将输入序列填充至 32 的倍数。当序列长度较短时，这会造成高浪费（例如，长度为 16 个词元的序列必须填充至 32 个词元，导致 50% 的浪费）。TurboTransformer 根据句子长度将输入拆分为多个较小的批次并依次处理，从而避免浪费，因此优于其他基线方法。PIT 则通过并行处理整个批次且不产生浪费，进一步优于 TurboTransformer。在内存使用量方面，PIT 的内存消耗低于 PyTorch 和 PyTorch-S，与 DeepSpeed 和 TurboTransformer 相近。DeepSpeed 和 TurboTransformer 通过将整个层融合为单个算子来减少激活值内存，从而优化内存使用量；这与 PIT 的稀疏性优化兼容。由于内核实现问题，当输入序列长度增加时，TurboTransformer 会崩溃。

Longformer 是一种仅编码器语言模型，采用动态稀疏注意力 [14]。Longformer 会自适应地关注输入中的若干重要词语（例如分类标记）。不同输入的动态注意力位置各不相同，这正是动态稀疏性的来源。图 12 展示了输入序列长度为 2048 和 4096 时 Longformer 的推理延迟和内存开销。PIT 对动态稀疏注意力中的动态稀疏性进行了优化。为了进行全面评估，我们还加入了专为 Longformer 优化的稀疏实现（记为 Longformer-S [1]）。PyTorch-S 选择 Triton 作为后端。图 12 展示了单个 V100 上的延迟和 GPU 内存使用量。与 PyTorch、Longformer-S、PyTorch-S 和 DeepSpeed 相比，PIT 的速度最高分别提升 1.9 倍、1.8 倍、2.4 倍和 2.4 倍。Longformer-S 的表现优于 PyTorch-S，这是因为它针对自身设计的稀疏模式，通过稀疏模式分解专门优化了 GPU 内核。然而，这种设计难以应用于其他模型。DeepSpeed 使用 Triton 实现稀疏注意力，因此性能与 PyTorch-S 相近。PyTorch-S 的索引构建开销显示在“PyTorch-S 转换”中，占端到端延迟的 6.3% ~13.9%。此外，当 PyTorch-S 选择细粒度稀疏库作为后端时，表现甚至更差，因为稀疏率不够高。当 PyTorch-S 选择 Triton（块稀疏）作为后端时，由于动态全局注意力[14]造成计算浪费，其速度比 PIT 慢。Longformer-S 通过重排输入张量避免了计算浪费，因此效率更高，但会引入较大的数据重排开销。相比之下，PIT 按需组织微块，开销可以忽略不计，并使用 SRead 和 SWrite，直接在高效的稠密计算块中执行计算，避免计算浪费。在内存使用方面，PIT 占用的内存最少。当输入序列长度达到 4096 时，PyTorch-S 和 DeepSpeed 均因内存溢出而崩溃。PyTorch-S 和 DeepSpeed 都必须使用块稀疏（32×32 在 Triton 中）来覆盖所有剩余值，这会导致计算浪费和更高的稀疏率。Longformer-S 的数据重排会产生许多临时中间张量，因此增加了额外的内存开销。

Museformer 是一种仅解码器语言模型，它还会根据输入数据生成动态稀疏模式，以提升模型性能 [65]。图13展示了不同输入序列长度下的推理延迟和内存占用。PIT的速度分别比PyTorch、PyTorch-S和DeepSpeed快2.5倍、2.0倍和2.0倍

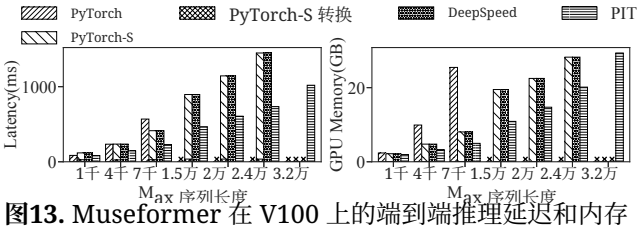


图13. Museformer 在 V100 上的端到端推理延迟和内存占用。

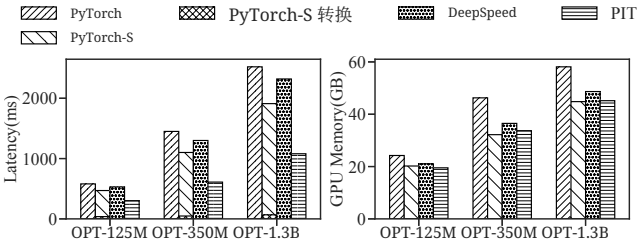


图14. OPT训练的每批次端到端延迟和内存占用。

，之后它们均因内存溢出而崩溃。对于较短序列，PyTorch-S中稀疏索引构建时间占端到端延迟的比例最高可达23.2%。随着输入长度增加，计算量增大，索引构建时间所占比例会逐渐降低。至于内存使用量，PIT的内存占用最低。PyTorch由于无法理解和优化动态稀疏性，因此消耗的内存多得多。与PyTorch-S和DeepSpeed相比，PIT通过PIT变换减少了计算浪费，从而降低了内存消耗。

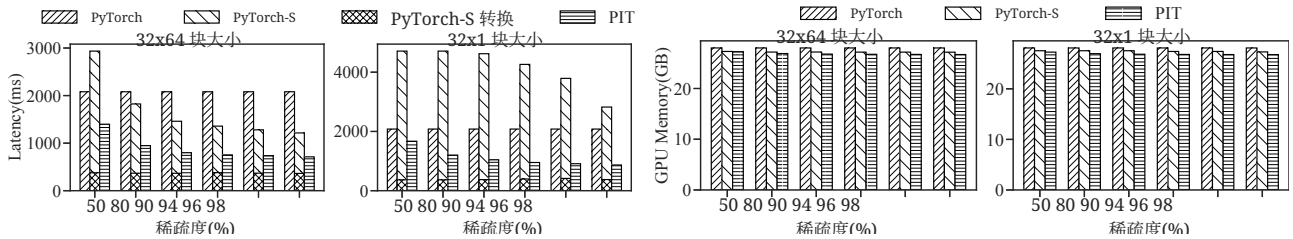
5.2 端到端训练

本节中，我们在 NVIDIA A100 和 V100 图形处理器上评估 PIT，以展示其利用动态稀疏性加速训练时的卓越性能。

OPT 训练。 我们使用 Alpaca 数据集，在 NVIDIA A100-80GB GPU 上对 125M、350M 和 1.3B 的 OPT 模型进行了微调。由于同一批次中的句子长度各不相同，我们使用 PIT 优化动态稀疏性。受内存限制，我们将训练批次大小设为 8。在实验中，我们比较了 PIT 与 PyTorch、PyTorch-S 和 DeepSpeed 的性能。图14所示结果展示了前向传播和反向传播的耗时及内存占用。与 Py- Torch、PyTorch-S 和 DeepSpeed 相比，PIT 分别实现了 1.9x~2.4x、1.6x~1.8x 和 1.8x~2.2x 的加速。与针对不同句子长度进行的推理优化类似，PIT 避免了 PyTorch 和 DeepSpeed 中填充所需的计算。与 PyTorch-S 相比，PIT 支持比 Triton 的块稀疏粒度（32 个词元）更细的稀疏粒度（1 个词元），从而提高计算效率。此外，PIT 避免了 PyTorch-S 将数据从稠密格式重新格式化为稀疏格式所产生的内存访问开销。在内存占用方面，PIT 和 PyTorch-S 在动态稀疏训练期间的内存占用最小。与推理不同，在训练中，DeepSpeed 无法通过将整个层融合到一个算子中来节省激活内存，因此会消耗更多内存。**稀疏训练**。 我们采用模型压缩的一种常见方法——迭代剪枝，来展示 PIT 如何利用动态稀疏性加速稀疏训练。在每一步中，剪枝算法都会根据权重幅值生成掩码，而掩码会因输入不同而变化。我们在 GLUE 数据集 [59] 上使用 BERT 模型评估了该方法，并以两种粒度进行分块稀疏剪枝： 32×64 和 32×1。图15a展示了每种稀疏粒度设置下处理一个数据批次所需的时间（包括前向传播和反向传播），批次大小为 32。PyTorch-S 使用 Triton 作为后端。当稀疏粒度为 32×64时，与 PyTorch 和 PyTorch-S 相比，PIT 分别实现了 1.5x~3.0x 和 1.7x~2.2x 的加速。虽然稀疏粒度为 32×64，大于 PyTorch-S 块稀疏内核的计算瓦片（即 32×32），但剪枝过程中每一层的稀疏模式都在不断变化。因此，PyTorch-S 的每一层都必须在每个批次重新构建稀疏索引，导致其承受高昂的索引构建开销。在粒度为32×64 时，PIT 主要凭借快速索引构建优于 PyTorch-S。

已有研究表明，较小的稀疏粒度可以提高准确率，但也给执行性能优化带来了挑战 [40]。在我们的评估中，采用该粒度 32×1 时，剪枝模型在 MNLI数据集上的准确率提升了 0.26%~ 0.72%，在 SST-2 数据集上的准确率提升了 0.11%~ 1.26%。在延迟方面，PIT 的表现优于 PyTorch 和 PyTorch-S，分别快 2.4 倍和 4.8 倍。由于数据稀疏粒度未对齐（32×1），且使用了块稀疏 GPU 内核（32×32 或 16×16），PyTorch-S 的计算时间显著增加。值得注意的是，当稀疏粒度设为 32×1 时，PIT 达到了几乎与 32×64 相同的速度，因为 PIT 可以使用细粒度的 32×1 微瓦片来覆盖稀疏数据，同时使用 32×64 内核实现最高效的计算，兼得两者之长。

图 15b 展示了剪枝过程中的内存使用量。与 PyTorch 和 PyTorch-S 相比，PIT 使用的内存最少。随着稀疏率提高，内存占用仅略有下降，因为迭代剪枝算法只在剪枝过程中对模型权重进行剪枝，而权重张量仅占内存的一小部分。相比之下，PyTorch 和 PyTorch-S 会存储稠密权重和梯度，因此其内存占用几乎不会随稀疏率变化。



(a) 延迟　　**(b)** 内存

图 15. 32×64 和 32×1 块下幅值迭代剪枝的端到端每批次延迟和内存占用。

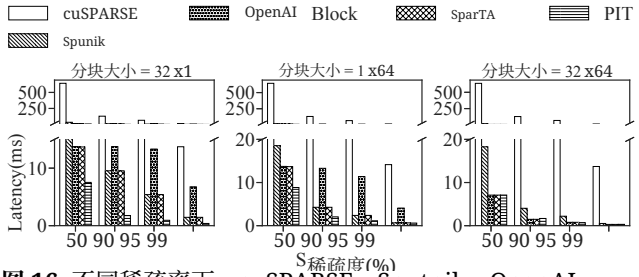


图 16. 不同稀疏率下，cuSPARSE、Sputnik、OpenAI Block Sparse、SparTA 和 PIT 在矩阵乘法（$4096\times4096\times4096$）上的比较。

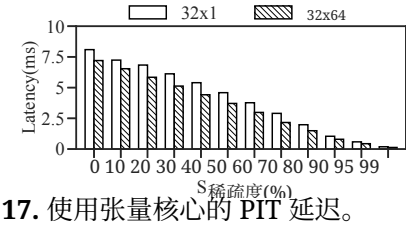


图 17. 使用张量核心的 PIT 延迟。

5.3 PIT变换的有效性

对稠密内核进行 PIT 变换。在本研究中，我们评估了 PIT 稀疏矩阵乘法内核在不同稀疏粒度和形状下的性能。为展示 PIT 变换的有效性，我们将 PIT 与其他几个稀疏库进行比较，包括 cuSPARSE、Sputnik 和 OpenAI Block Sparse（Triton）。我们还介绍了 SparTA [68], ，这是一款面向静态稀疏优化的先进稀疏深度学习编译器。在本实验中，我们使用静态稀疏模式评估计算效率，因此图 16 中展示的延迟结果不包括转换或编译开销。当稀疏粒度为 32×64 时，PIT、SparTA 和 OpenAI Block Sparse 的延迟相近，因为它们使用相同的稠密计算瓦片。由于计算粒度过细，cuSPARSE 和 Sputnik 的表现不佳。当稀疏粒度为 32×1 和 1×64 时，Sputnik 和 SparTA 凭借更好的粒度对齐，性能优于 cuSPARSE 和 OpenAI Block Sparse。对于稀疏粒度 32×1，PIT 的速度比 Sputnik 快 4.3x~5.8x，比 SparTA 快 1.5x~5.7x。对于稀疏粒度 64×1，PIT 的速度比 Sputnik 快 1.1x~2.3x，比 SparTA 快 1.1x~2.2x。PIT 变换使 PIT 即使在稀疏粒度较小（即 32×1）时也能执行高效计算，从而带来了相较于 Sputnik 和 SparTA 的性能提升。此外，在不同稀疏粒度下，SRead 和 SWrite 的开销可以忽略不计，其运行速度接近原始稠密计算瓦片。

硬件指令上的 PIT 变换。在本实验中，我们展示了 PIT变换 如何放宽硬件指令（如 wmma [4]）的限制。我们针对两种不同的稀疏粒度（32×1 和 32×64）对 $[4096,4096]\times[4096,4096]$ 矩阵乘法执行稀疏矩阵乘法。第一个输入张量是稀疏的，并以列主序格式存储。wmma 指令仅支持半精度下的三种形状（$[16,16]\times[16,16]$、$[32,8]\times[8,16]$、$[8,32]\times[32,16]$），因此不适用于 32×1 稀疏粒度。我们使用 PIT变换 分别生成微块为 32×1和 32×64的两个稀疏内核。我们使用微块为 32×1 的稀疏内核，以 32×1 稀疏粒度执行稀疏矩阵乘法；而微块为 32×64 的内核则用于 32×64 稀疏粒度。图 16 展示了 PIT 生成的两个稀疏内核在不同稀疏率下具有相近的延迟。这证明 PIT变换 引入的开销很小。

5.4 转换开销

在本实验中，我们更详细地展示 PIT 的转换开销。具体而言，我们在 V100-32GB GPU[2] 上评估不同稀疏粒度和稀疏率下的转换开销。图18展示了 PIT 和 PyTorch-S 的稀疏索引构建时间。当粒度为 1×1 时，PyTorch-S 选择 cuSPARSE 提供的索引构建函数；当粒度为 16×16 和 32×32 时，则选择 Triton 提供的函数。当粒度为 1×1 时，PIT 比 cuSPARSE 快 3.6x~4.7x；当粒度为

[2]与 V100-32GB 不同，在 V100-16GB 上，当分块大小为 1x1 时，PIT 的延迟与 PyTorch-S 相近。PIT 在 V100-16GB 上相较于其他分块大小的性能优势与图18所示相同。

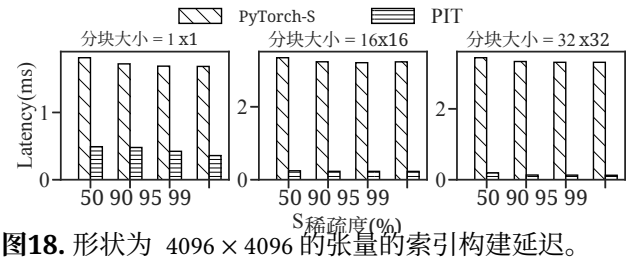


图18. 形状为 4096×4096 的张量的索引构建延迟。

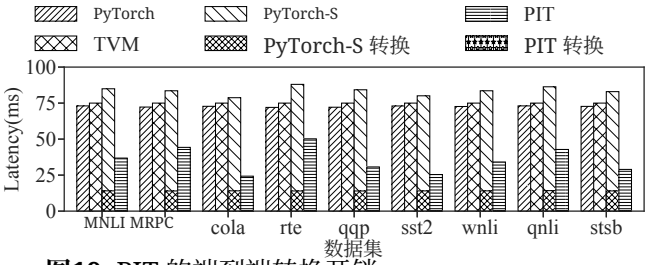


图19. PIT 的端到端转换开销。

16×16 时，PIT 比 Triton 快 11.2x.2x；当粒度为 32×32 时，PIT 的索引构建速度比 Triton 快 13.3x~26.5x。据我们所知，PIT 是首个支持所有类型稀疏粒度快速在线索引构建的方法。

为了进一步评估转换开销，我们还测量了转换开销占 PIT 端到端延迟的比例。具体而言，PIT 优化了 BERT 模型中由不同序列长度导致的动态稀疏性。除了 PyTorch，我们还引入了 TVM，这是一种流行的稠密张量编译器。TVM 中的每项任务都由自动调度器（Ansor [67]）微调 2000 步。图19展示了 PIT 和基线方法的端到端延迟，其中包括 PIT（"PIT 转换"）和 PyTorch-S（"PyTorch-S 转换"）的索引构建开销。PIT 的转换开销占端到端延迟的 0.7% 至 1.1%，在图中几乎不可见。

5.5 微瓦片线上搜索

不同的稀疏模式和稀疏率可能对应不同的最优微瓦片。PIT 会同时考虑计算内核的效率和计算浪费，以找到最佳的微瓦片配置。表3展示了在不同稀疏粒度和稀疏率下，$4096 \times 4096 \times 4096$矩阵乘法搜索得到的微块。以表3第一行为例，算法发现，当稀疏率为 95% 时，使用大小为 16×1 的微瓦片覆盖粒度为 2×1 的区域最为高效。针对 2×1 数据使用大小为 16×1 的微瓦片，会使 PIT 计算中的稀疏率达到 66.39%。微瓦片 16×1 是通过对第一个输入张量的第二个轴应用 PIT变换，从稠密计算瓦片 $16 \times 32 \times 128$ 推导而来的。PIT 会动态权衡内核效率与计算浪费。它需要

稀疏度粒度	原始稀疏度比率(%)	微瓦片	稀疏度后比率 覆盖率（%）	原始稠密核	延迟 (ms)
(2,1)	95	(16, 1)	66.39	[16, 32] × [32, 128]	8.04
(2,1)	99	(8, 1)	96.06	[8, 32] × [32, 128]	2.34
(4,1)	95	(16, 1)	81.45	[16, 32] × [32, 128]	4.29
(4,1)	99	(16, 1)	96.05	[16, 32] × [32, 128]	1.37
(8,1)	95	(8, 1)	95	[8, 32] × [32, 128]	2.34
(8,1)	99	(32, 1)	96.02	[32, 64] × [64, 32]	0.90
(32, 1)	95	(32, 1)	95	[32, 64] × [64, 32]	0.94
(32, 1)	99	(32, 1)	99	[32, 64] × [64, 32]	0.39

表3. 不同稀疏粒度和稀疏率下的微瓦片线上搜索结果。

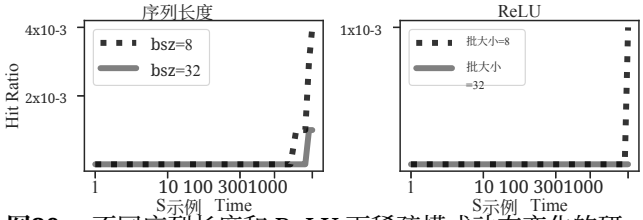


图20。 不同序列长度和 ReLU 下稀疏模式动态变化的研究。

PIT 搜索最佳微瓦片及其对应稠密计算瓦片需要 30us ~100us，这足以支持线上搜索。

5.6 动态稀疏模式研究

动态稀疏性的一种潜在替代方案是记忆常见的稀疏模式，为这些模式编译优化后的内核，并在模式再次出现时重用这些内核。在这项微基准测试中，我们通过研究动态稀疏模式重复出现的频率，证明了这种方案并不适用。我们采用两种普遍存在的动态稀疏模式，即输入序列长度变化，以及 ReLU 运算符导致的动态稀疏性。具体而言，我们以不同的批次大小（8 和 32）遍历 MNLI数据集，并检查当前输入批次的稀疏模式是否曾在之前的批次中出现。图20展示了先前批次中出现过的动态稀疏模式的累计命中率。我们发现，这两种稀疏模式的重复率都明显偏低。对于序列长度变化，只有 0.4% 的请求命中了先前批次中出现过的稀疏模式。ReLU 的这一比例更低，仅为 0.1%。鉴于这类稀疏性的动态特征，许多先前针对静态稀疏优化的研究已不再适用于动态稀疏性。针对特定稀疏模式优化的稀疏内核几乎无法重用。

6 相关工作

动态稀疏性已成为提升深度学习模型效率的关键领域。我们讨论了

现有解决方案的不同类别，包括针对静态稀疏模式和动态稀疏模式的软件解决方案，以及硬件解决方案。将 PIT 与这些解决方案进行比较，有助于更好地理解 PIT 的技术贡献。

静态稀疏模式的优化。 SparTA [68]、TACO [7]、SparseTIR [64]、Tiramisu [12] 和 SparseRT [61] 等基于提前编译的技术，需要为特定稀疏模式搜索合适的内核配置。这些技术能够针对给定的稀疏模式实现高性能，但无法处理仅在运行时才知晓的动态稀疏模式。PIT 的核心贡献是通过低开销的在线稀疏性检测和稀疏-稠密数据转换（即 SRead 和 SWrite），即时支持动态稀疏模式。此外，对于大多数静态稀疏模式，尽管 PIT 只能在运行时检测稀疏模式，其性能仍可媲美提前编译器，从而大幅降低编译开销。

通用稀疏模式的优化。尽管通用稀疏库不需要提前编译，例如 OpenAI 的块稀疏内核 [9]、cuSPARSE [3]、cuSPARSELT [5]、HipSparse [8]、Sputnik [31] 和 nmSparse [15]，但这些库仅支持或只能在特定数据粒度和计算粒度下高效运行，因此难以支持更复杂的稀疏模式。例如，OpenAI 的块稀疏内核 [9] 仅支持大小为 32×32 的稀疏数据块，当稀疏模式粒度更细（例如 1×32）时，会造成计算浪费。PIT 通过微块解决了这一问题。PIT 支持通过稀疏地读写多个微块，在线构建更大的块，从而同时实现高 GPU 效率和低稀疏性覆盖成本。尽管 ASpT [36] 提出了一种适用于细粒度稀疏矩阵乘法的自适应分块机制，但会带来显著的离线数据重排开销。只要微块能够充分利用 GPU 内存事务，PIT 就能以可忽略的开销构建分块。

稀疏计算的硬件优化。 除了上述软件解决方案之外，研究人员还提出了许多针对深度学习模型稀疏计算的硬件优化方法 [2, 13, 17, 32, 62]。这些硬件优化技术通常针对特定的稀疏模式。例如，英伟达 A100 GPU 中的稀疏张量核心 [2]仅支持严格的 2:4模式，即每个 1×4 分块都必须恰好包含 2 个零，这极大地限制了其对更多样化稀疏模式的适用性。PIT 不仅不假设稀疏模式，还能够扩展现有硬件解决方案所支持的有限稀疏模式。例如，当一个矩阵包含多个混合的 1x4 块，且这些块具有两种稀疏模式：两个零和全零时，PIT 可以构建微块，只将双零情形输入稀疏张量核心，从而避免不必要的全零块计算。这种 GPU 内核可以通过结合 PIT 的 SRead/SWrite 与稀疏张量核心的“mma.sp” PTX 指令来实现，我们将其留待未来工作。

大语言模型优化。在近期针对大语言模型推理优化的进展中，vLLM [39] 与 PIT 的关系最为密切。vLLM 提出了分页注意力，可从 GPU 内存的不同物理地址“稀疏地”加载和保存词元，打破了词元必须连续存储的限制。它避免了因序列长度不同而产生的大量填充，以及束搜索过程中的冗余复制。分页注意力可视为针对生成式语言模型中特殊动态稀疏模式的领域专用解决方案。PIT 可通过定制的 PIT变换策略实现 vLLM。PIT 在设计上是一种通用的动态稀疏解决方案，有助于支持不同的序列长度以及更具挑战性的稀疏模式（例如混合专家、稀疏注意力和稀疏训练）。

7 结论

PIT 基于置换不变变换这一深度学习计算中普遍存在的性质，以系统化的方法支持在商用加速器上高效执行动态稀疏模型。借助这一性质，PIT 能够以在线方式从适合硬件的微块构建计算高效的稠密分块。PIT 展示了一种处理动态稀疏性的新颖且有效的方法，而动态稀疏性正日益成为深度学习的发展趋势。大量评估结果表明，与最先进的解决方案相比，PIT 可将推理和训练中的动态稀疏计算加速至多 5.9 倍。

致谢

我们感谢指导人 Thomas Pasquier 提出了宝贵建议，提升了论文的质量。我们也感谢匿名评审者提供的建设性反馈。Quanlu Zhang、Zhenhua Han 和 Yuqing Yang 为通讯作者。

参考文献

[1] Longformer。https://github.com/allenai/longformer，2020年。[2] 使用 NVIDIA Ampere 架构和 NVIDIA TensorRT 借助稀疏度加速推理。
h
t
t
ps://
developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/，2021年。[3] cuSPARSE（CUDA稀疏矩阵库）API 参考指南。
https://docs.nvidia.com/cuda/cusparse/index.html，2021年。[4] CUDA C++ 编程指南。
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma，2021年。[5] cuSPARSELT：用于稀疏矩阵乘法的高性能 CUDA 库。
https://docs.nvidia.com/cuda/cusparselt/index.html，2021年。[6] OpenAI 块稀疏。
https://github.com/openai/blocksparse.git，2021年。[7] 复现 OOPSLA 2020 的结果。
https://github.com/tensor-compiler/taco/tree/oopsla2020，2021年。

[8] ROCm 稀疏编组库。https://github.com/ROCmSoftwarePlatform/hipSPARSE，2021年。[9] Triton。https://github.com/openai/triton.git，2021年。[10] TVM 稀疏性代码。
https://github.com/apache/tvm/blob/254563a3140cf63fe77a46058688209de3aa213c/python/tvm/topi/cuda/sparse.py#L96，2021年。[11] DeepSpeed。https://github.com/microsoft/DeepSpeed，2022年。[12] Riyadh Baghdadi、Abdelkader Nadir Debbagh、Kamel Abdous、Fatima Zohra Benhamida、Alex Renda、Jonathan Elliott Frankle、Michael Carbin 和 Saman Amarasinghe。Tiramisu：一种用于稠密与稀疏深度学习的多面体编译器。*arXiv* 预印本，abs/2005 04091，2020年。。[13] Nathan Bell 和 Michael Garland。基于 CUDA 的高效稀疏矩阵-向量乘法。技术报告，Citeseer，2008年。[14] Iz Beltagy、Matthew E Peters 和 Arman Cohan。Longformer：长文档 Transformer。*arXiv* 预印本，abs/2004.05150，2020年。[15] Lin Bin、Zheng Ningxin、Wang Yang、Cao Shijie、Ma Lingxiao、Zhang Quanlu、Zhu Yi、Cao Ting、Xue Jilong、Yang Yuqing 和 Yang Fan。用于深度学习中 N:M 稀疏权重的高效 GPU 内核。*MLSys2023*，2023年。[16] Tom B. Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel M. Ziegler、Jeffrey Wu、Clemens Winter、Christopher Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。语言模型是少样本学习者。载于 Hugo Larochelle、Marc'Aurelio Ranzato、Raia Hadsell、Maria-Florina Balcan 和 Hsuan-Tien Lin 编，《神经信息处理系统进展*33*：神经信息处理系统年会 *2020*，*NeurIPS 2020*，*2020*年*12*月*6*日至*12*日，线上》，2020年。[17] Aydin Buluç、Jeremy T Fineman、Matteo Frigo、John R Gilbert 和 Charles E Leiserson。使用压缩稀疏块进行并行稀疏矩阵-向量及矩阵转置-向量乘法。载于第二十一届算法与体系结构并行化年度研讨会论文集，第233–244页，2009年。[18] Han Cai、Chuang Gan、Tianzhe Wang、Zhekai Zhang 和 Song Han。Once-for-all：训练一个网络并对其进行专门化，以实现高效部署。载于第八届国际学习表征会议，*ICLR 2020*，亚的斯亚贝巴，埃塞俄比亚，*2020*年*4*月*26*日至*30*日。OpenReview.net，2020年。[19] Minghao Chen、Houwen Peng、Jianlong Fu 和 Haibin Ling。Autoformer：为视觉识别搜索 Transformer。载于*IEEE/CVF* 国际计算机视觉会议论文集，第12270–12280页，2021年。[20] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze 等。{TVM}：一种用于深度学习的自动化{端到端}优化编译器。载于第*13*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 18*），第578–594页，2018年。[21] Arman Cohan、Franck Dernoncourt、Doo Soon Kim、Trung Bui、Seokhwan Kim、Walter Chang 和 Nazli Goharian。一种面向语篇的注意力模型，用于长文档的抽象式摘要。载于*2018*年计算语言学协会北美分会会议论文集：人类语言技术，第*2*卷（短论文），第615–621页，新奥尔良，路易斯安那州，2018年。计算语言学协会。[22] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。BERT：用于语言理解的深度双向 Transformer 预训练。载于*2019*年计算语言学协会北美分会会议论文集：人类语言技术，第*1*卷（长篇与短篇论文），第4171–4186页，明尼阿波利斯，明尼苏达州，2019年。计算语言学协会。

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weis- senborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit 和 Neil Houlsby。《一张图像胜过 16×16 个词：面向大规模图像识别的 Transformer》。载于第*9*届国际学习表征会议，*ICLR 2021*，线上会议，奥地利，*2021*年*5*月*3*日至*7*日。OpenReview.net，2021。[24] Nan Du、Yanping Huang、Andrew M Dai、Simon Tong、Dmitry Lep- ikhin、Yuanzhong Xu、Maxim Krikun、Yanqi Zhou、Adams Wei Yu、Orhan Firat 等。《Glam：利用混合专家高效扩展语言模型》。载于国际机器学习会议，第5547–5569页。PMLR，2022。[25] 阿尔伯特·爱因斯坦。《运动物体的电动力学》。（德语）[《运动物体的电动力学》]。《物理学年鉴》，322(10):891–921，1905年。[26] Utku Evci、Trevor Gale、Jacob Menick、Pablo Samuel Castro 和 Erich Elsen。《操纵彩票：让所有彩票都成为赢家》。载于第*37*届国际机器学习会议论文集，*ICML 2020*，*2020*年*7*月*13*日至*18*日，线上会议，机器学习研究论文集第119卷，第2943–2952页。PMLR，2020。[27] Alexander Fabbri、Irene Li、Tianwei She、Suyi Li 和 Dragomir Radev。《Multi-News：大规模多文档摘要数据集与抽象式层次模型》。载于第*57*届计算语言学协会年会论文集，第1074–1084页，意大利佛罗伦萨，2019年。计算语言学协会。[28] Jiarui Fang、Yang Yu、Chengduo Zhao 和 Jie Zhou。《TurboTransformers：面向Transformer模型的高效GPU服务系统》。载于第*26*届*ACM SIGPLAN*并行编程原理与实践研讨会论文集，第389–402页，2021年。[29] William Fedus、Barret Zoph 和 Noam Shazeer。《Switch Transformers：以简单高效的稀疏性扩展至万亿参数模型》。《机器学习研究期刊》，23(120):1–39，2022年。[30] Trevor Gale、Deepak Narayanan、Cliff Young 和 Matei Zaharia。《MegaBlocks：利用混合专家实现高效稀疏训练》。*ML- Sys2023*，2023年。[31] Trevor Gale、Matei Zaharia、Cliff Young 和 Erich Elsen。《用于深度学习的稀疏 GPU 内核》。载于高性能计算、网络、存储与分析国际会议论文集，*SC 2020*，2020年。[32] Trevor Gale、Matei Zaharia、Cliff Young 和 Erich Elsen。《用于深度学习的稀疏 GPU 内核》。载于高性能计算、网络、存储与分析国际会议论文集，*SC 2020*，2020年。[33] Amirhossein Habibian、Davide Abati、Taco S Cohen 和 Babak Ehte- shami Bejnordi。《用于高效视频处理的跳跃卷积》。载于*IEEE/CVF*计算机视觉与模式识别会议论文集，第2695–2704页，2021年。[34] Yizeng Han、Gao Huang、Shiji Song、Le Yang、Honghui Wang 和 Yulin Wang。《动态神经网络：综述》。《*IEEE*模式分析与机器智能汇刊》，2021年。[35] 何恺明、Xinlei Chen、Saining Xie、Yanghao Li、Piotr Dollár 和 Ross Girshick。《掩码自编码器是可扩展的视觉学习器》。载于*IEEE/CVF*计算机视觉与模式识别会议论文集，第16000–16009页，2022年。[36] Changwan Hong、Aravind Sukumaran-Rajam、Israt Nisa、Kunal Singh 和 P Sadayappan。《稀疏矩阵乘法的自适应稀疏分块》。载于第*24*届并行编程原理与实践研讨会论文集，第300–314页，2019年。[37] Changho Hwang、Wei Cui、Yifan Xiong、Ziyue Yang、Ze Liu、Han Hu、Zilong Wang、Rafael Salas、Jithin Jose、Prabhat Ram 等。《Tutel：大规模自适应混合专家》。*arXiv*预印本，abs/2206 03382，2022 年。。[38] Nikita Kitaev、Lukasz Kaiser 和 Anselm Levskaya。《Reformer：高效的Transformer》。载于第8届国际学习

表征会议，*ICLR 2020*，埃塞俄比亚亚的斯亚贝巴，*2020*年*4*月*26*日至*30*日。OpenReview.net，2020。[39] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Shen、Lianmin Zheng、Cody Yu、Joey Gonzalez、Hao Zhang 和 Ion Stoica。《利用分页注意力为大语言模型服务实现高效内存管理》。载于*ACM SIGOPS*第*29*届操作系统原理研讨会论文集（*SOSP 23*），2023年。[40] François Lagunas、Ella Charlaix、Victor Sanh 和 Alexander Rush。《用于加速Transformer的块剪枝》。载于*2021*年自然语言处理实证方法会议论文集，第10619–10629页，线上会议及多米尼加共和国蓬塔卡纳，2021年。计算语言学协会。[41] Juho Lee、Yoonho Lee、Jungtaek Kim、Adam R. Kosiorek、Seungjin Choi 和 Yee Whye Teh。《集合Transformer：基于注意力的置换不变神经网络框架》。载于Kamalika Chaudhuri和Ruslan Salakhutdinov主编的第*36*届国际机器学习会议论文集，*ICML 2019*，*2019*年*6*月*9*日至*15*日，美国加利福尼亚州长滩，机器学习研究论文集第97卷，第3744–3753页。PMLR，2019。[42] Dmitry Lepikhin、HyoukJoong Lee、Yuanzhong Xu、Dehao Chen、Orhan Firat、Yanping Huang、Maxim Krikun、Noam Shazeer 和 Zhifeng Chen。《GShard：通过条件计算和自动分片扩展巨型模型》。载于第*9*届国际学习表征会议，*ICLR 2021*，*7*，*2021 3*2021线上会议，奥地利，*5*月-。OpenReview.net，。[43] Zhuohan Li、Eric Wallace、Sheng Shen、Kevin Lin、Kurt Keutzer、Dan Klein 和 Joey Gonzalez。《先训练大模型，再压缩：重新思考Transformer高效训练与推理的模型规模》。载于第*37*届国际机器学习会议论文集，*ICML 2020*，*2020*年*7*月*13*日至*18*日，线上会议，机器学习研究论文集第119卷，第5958–5968页。PMLR，2020。[44] Zonglin Li、Chong You、Srinadh Bhojanapalli、Daliang Li、Ankit Singh Rawat、Sashank J Reddi、Ke Ye、Felix Chern、Felix Yu、Ruiqi Guo 等。《大模型是简约型学习者：训练后的Transformer中的激活稀疏性》。*arXiv*预印本，abs/2210.06313，2022年。[45] Liu Liu、Zheng Qu、Zhaodong Chen、Yufei Ding 和 Yuan Xie。《利用动态稀疏注意力加速Transformer》。*arXiv*预印本，abs/2110.11299，2021年。[46] Shiwei Liu 和 Zhangyang Wang。《我们在新"稀疏领域"中学到的十条经验：稀疏神经网络研究人员简明手册》。*arXiv*，abs/2302.02596，2023年。[47] Yao Lu、Yue Dong 和 Laurent Charlin。《Multi-XScience：用于科学文章极端多文档摘要的大规模数据集》。载于*2020*年自然语言处理实证方法会议（*EMNLP*）论文集，第8068–8074页，线上会议，2020年。计算语言学协会。[48] Lingxiao Ma、Zhiqiang Xie、Zhi Yang、Jilong Xue、Youshan Miao、Wei Cui、Wenxiang Hu、Fan Yang、Lintao Zhang 和 Lidong Zhou。《Rammer：利用rtasks实现全面的深度学习编译器优化》。载于第*14*届*USENIX*操作系统设计与实现会议论文集，第881–897页，2020年。[49] Andrew L. Maas、Raymond E. Daly、Peter T. Pham、Dan Huang、Andrew Y. Ng 和 Christopher Potts。《学习用于情感分析的词向量》。载于第*49*届计算语言学协会年会：人类语言技术论文集，第142–150页，美国俄勒冈州波特兰，2011年。计算语言学协会。[50] OpenAI。《GPT-4技术报告》。*arXiv*，abs/2303.08774，2023年。[51] Long Ouyang、Jeff Wu、Xu Jiang、Diogo Almeida、Carroll L Wainwright、Pamela Mishkin、Chong Zhang、Sandhini Agar wal、Katarina Slama、Alex Ray 等。《通过人类反馈训练语言模型遵循指令》。*arXiv*预印本，abs/2203.02155，2022年。[52] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Brad- bury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Köpf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。PyTorch：一种命令式风格的高性能深度学习库。载于Hanna M. Wallach、Hugo Larochelle、Alina Beygelzimer、Florence d'Alché-Buc、Emily B. Fox 和 Roman Garnett 编，神经信息处理系统进展 *32*：*2019*年神经信息处理系统年度会议，*NeurIPS 2019*，*2019*年*12*月*8*日至*14*日，加拿大不列颠哥伦比亚省温哥华，第8024–8035 页，2019年。[53] Alec Radford、Jong Wook Kim、Chris Hallacy、Aditya Ramesh、Gabriel Goh、Sandhini Agarwal、Girish Sastry、Amanda Askell、Pamela Mishkin、Jack Clark、Gretchen Krueger 和 Ilya Sutskever。通过自然语言监督学习可迁移视觉模型。载于 Marina Meila 和 Tong Zhang 编，第*38*届国际机器学习大会，*ICML 2021*，*2021*年*7*月*18*日至*24*日，线上会议，第 139 卷《机器学习研究论文集》，第 8748–8763 页。PMLR，2021年。[54] Colin Raffel。基于学习的方法用于比较序列，并应用于音频到MIDI对齐和匹配。2016年。[55] Colin Raffel、Noam Shazeer、Adam Roberts、Katherine Lee、Sharan Narang、Michael Matena、Yanqi Zhou、Wei Li 和 Peter J Liu。通过统一的文本到文本转换器探索迁移学习的边界。《机器学习研究期刊》，21(1)：5485–5551，2020年。[56] Aditya Ramesh、Prafulla Dhariwal、Alex Nichol、Casey Chu 和 Mark Chen。利用 CLIP 潜变量进行层次化文本条件图像生成。*arXiv* 预印本，abs/2204.06125，2022年。[57] Yongming Rao、Wenliang Zhao、Benlin Liu、Jiwen Lu、Jie Zhou 和 Cho-Jui Hsieh。DynamicViT：采用动态令牌稀疏化的高效视觉 Transformer。神经信息处理系统进展，34：13937–13949，2021年。[58] Rohan Taori、Ishaan Gulrajani、Tianyi Zhang、Yann Dubois、Xuechen Li、Carlos Guestrin、Percy Liang 和 Tatsunori B. Hashimoto。斯坦福Alpaca：一个指令跟随 Llama 模型。https://github.com/tat su-lab/stanford_alpaca，2023年。[59] Alex Wang、Amanpreet Singh、Julian Michael、Felix Hill、Omer Levy 和 Samuel R. Bowman。GLUE：用于自然语言理解的多任务基准与分析平台。载于第*7*届国际学习表征会议，*ICLR 2019*，美国路易斯安那州新奥尔良，*2019*年*5*月*6*日至*9*日。OpenReview.net，2019年。[60] Hanrui Wang、Zhekai Zhang 和 Song Han。SpAtten：采用级联令牌和注意力头剪枝的高效稀疏注意力架构。载于*2021*年 *IEEE* 高性能计算机体系结构国际研讨会（*HPCA*），第 97–110 页。IEEE，2021年。[61] Ziheng Wang。SparseRT：在图形处理器上加速非结构化稀疏性，以实现深度学习推理。载于*ACM*并行体系结构与编译技术国际会议论文集，第 31–42 页，2020年。[62] Samuel Williams、Leonid Oliker、Richard Vuduc、John Shalf、Katherine Yelick 和 James Demmel。在新兴多核平台上优化稀疏矩阵-向量乘法。载于*SC'07*：*2007*年*ACM/IEEE*超级计算大会论文集，第 1–12 页。IEEE，2007年。[63]Thomas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、Pierric Cistac、Tim Rault、Remi Louf、Morgan Funtowicz、Joe Davison、Sam Shleifer、Patrick von Platen、Clara Ma、Yacine Jernite、Julien Plu、Canwen Xu、Teven Le Scao、Sylvain Gugger、Mariama Drame、Quentin Lhoest 和 Alexander Rush。Transformer：自然语言处理领域的最新进展。载于*2020*年自然语言处理经验方法会议论文集：系统演示，第38–45 页，线上，2020年。计算语言学协会。[64] Zihao Ye、Ruihang Lai、Junru Shao、Tianqi Chen 和 Luis Ceze。Sparse-TIR：用于深度学习稀疏编译的可组合抽象。*arXiv* 预印本，abs/2207.04606，2022年。

[65] Botao Yu、Peiling Lu、Rui Wang、Wei Hu、Xu Tan、Wei Ye、Shikun Zhang、Tao Qin 和 Tie-Yan Liu。Museformer：采用细粒度和粗粒度注意力的音乐生成 Transformer。载于 Alice H. Oh、Alekh Agarwal、Danielle Belgrave 和 Kyunghyun Cho 编，神经信息处理系统进展，2022年。[66] Susan Zhang、Stephen Roller、Naman Goyal、Mikel Artetxe、Moya Chen、Shuohui Chen、Christopher Dewan、Mona Diab、Xian Li、Xi Victoria Lin 等。OPT：开放式预训练 Transformer 语言模型。*arXiv* 预印本，abs/2205.01068，2022年。[67] Lianmin Zheng、Chengfan Jia、Minmin Sun、Zhao Wu、Cody Hao Yu、Ameer Haj-Ali、Yida Wang、Jun Yang、Danyang Zhuo、Koushik Sen 等。Ansor：为深度学习生成{高性能}张量程序。载于第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第 863–879 页，2020年。[68] Ningxin Zheng、Bin Lin、Quanlu Zhang、Lingxiao Ma、Yuqing Yang、Fan Yang、Yang Wang、Mao Yang 和 Lidong Zhou。{SparTA}：通过{带稀疏属性的张量}实现{深度学习}模型稀疏性。载于第*16*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 213–232 页，2022年。

[69] Haoyi Zhou、Shanghang Zhang、Jieqi Peng、Shuai Zhang、Jianxin Li、Hui Xiong 和 Wancai Zhang。Informer：超越高效 Transformer，实现长序列时间序列预测。发表于*AAAI*人工智能会议论文集，第 35 卷，11115，2021 11106页 – 。[70] Zhe Zhou、Junlin Liu、Zhenyu Gu 和 Guangyu Sun。Energon：利用动态稀疏注意力实现 Transformer 的高效加速。*IEEE*集成电路与系统计算机辅助设计汇刊，2022年。[71] Hongyu Zhu、Ruofan Wu、Yijia Diao、Shanbin Ke、Haoyu Li、Chen Zhang、Jilong Xue、Lingxiao Ma、Yuqing Xia、Wei Cui、Fan Yang、Mao Yang、Lidong Zhou、Asaf Cidon 和 Gennady Pekhimenko。ROLLER：面向深度学习的快速高效张量编译。发表于第*16*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 233–248 页，加利福尼亚州卡尔斯巴德，2022年。USENIX协会。[72] Barret Zoph、Irwan Bello、Sameer Kumar、Nan Du、Yanping Huang、Jeff Dean、Noam Shazeer 和 William Fedus。ST-MoE：设计稳定且可迁移的稀疏专家模型。*arXiv* 预印本，abs/2202 08906，2022年。