面向大规模稀疏张量计算的自动张量-关系分解

Yuxin Tang、Zhiyuan Xin、Zhimin Ding、Xinyu Yao、Daniel Bourgeois、Tirthak Patel、Chris Jermaine 莱斯大学，美国德克萨斯州休斯顿 {yt33,zx58,zd21,xy38,dcb10,tp53,cmj4}@rice.edu

摘要

张量关系计算是一种关系计算，其中每个元组都携带向量、矩阵或高维数组。张量-关系计算的优势在于，整个计算可以在关系系统之上执行，从而继承该系统自动处理超大规模、高稀疏性输入的能力，同时可以使用高性能内核（例如经过优化的矩阵-矩阵乘法代码）执行大部分底层数学运算。本文介绍大写-小写-大写 EinSum，它是经典爱因斯坦求和记号的张量关系版本。我们研究如何将爱因斯坦记号中的计算自动重写为大写-小写-大写 EinSum，使计算密集型部分能够通过高效的数值内核执行，同时以关系方式管理稀疏性。

PVLDB 引用格式：
Yuxin Tang、Zhiyuan Xin、Zhimin Ding、Xinyu Yao、Daniel Bourgeois、Tirthak Patel、Chris Jermaine。《面向大规模稀疏张量计算的自动张量-关系分解》。PVLDB，19(6)：1101–1114，2026年。doi:10.14778/3797919.3797921

PVLDB 工件可用性：
源代码、数据和/或其他工件已在 https://github.com/yuxineverforever/upper-case-lower-case-einstein-notation 发布。

1 引言

在机器学习中，张量是一种特殊的关系，它将指定多维数组中某个位置的键映射到一个标量。鉴于张量与关系之间的密切联系，几乎所有针对张量的计算都可以基于关系数据库系统实现。

例如，考虑矩阵乘法链（$\mathbf{X}\times\mathbf{Y})^T\times\mathbf{Z}$。这种乘法用于在图神经网络 [41, 50] 中实现消息传递方案。也就是说，**X** 的每一列都存储一个顶点的嵌入，**Y** 编码顶点之间的（有向）边，目标是将沿边传递到给定顶点的所有嵌入求和，然后对每个顶点处的和进行线性变换（由**Z**表示）。如果所有输入矩阵都以（行、列、值）三元组的形式存储，则与此计算对应的 SQL 如下：

选择 XtimesY .行 , Z.col ,**求和** (XtimesY . value * Z. value) **作为** value **从** (**选择** X.row **作为** col , Y.col **作为** 行 , **求和** (X. value * Y.value) **作为** value**从** X, Y **其中** X.col = Y.row**分组依据** X.row , Y.col) **作为** XtimesY , Z**其中** XtimesY .col = Z.row**分组依据** XtimesY .行 , Z.col

关系实现是否有效？ 遗憾的是，这种计算不太可能在关系系统上高效运行。设想一个由**Y** 表示的大型稀疏图，包含 1 亿个顶点，平均每个顶点有 1000 个邻居。每个顶点的嵌入（存储在**X**中）大小为 8192，而变换**Z** 会生成一个大小为 8192 的向量。在这种情况下，**X** 与**Y**的第一次连接将产生约 820 万亿个中间元组，随后聚合为 8200 亿个元组。然后将该结果与**Z**连接，产生*6000* 万亿个元组，这是一个令人难以承受的数字。

也可以直接在传统深度学习系统（例如 PyTorch [30]）中使用张量执行这项计算。这有助于使用 GPU 等加速器，但结果可能并不理想。首先，存储**X** 需要 3.2TB 的 GPU 内存（采用半精度）。这需要 40 块 GPU（假设每块有 80GB 存储空间）。其次还有计算方面的问题。GPU 并非为稀疏计算而设计，对于非常稀疏的矩阵，即使是最先进的手工调优稀疏-稠密 GPU 内核，在 A100 GPU [42] 上的计算利用率也只有约 0.1%。在 0.1% 的计算利用率下，完成第一次矩阵乘法所需的 1.6×10^{15} 浮点运算将需要超过一小时的 A100 GPU 时间。

一种稀疏张量关系实现。 相反，可以以张量关系方式 [46], 执行此计算，将问题分解，使关系系统能够利用计算中固有的稀疏性，同时在适当之处使用高效的稠密 CPU（或 GPU）内核：

X（列 **整数**，值 **向量** [8192]）Y（行 **整数** ，列 **整数** ，值 **双精度浮点数**）Z（值 **矩阵** [8192， 8192]）

选择 XtimesY .行 , vec_mat_mult (XtimesY .值 , Z.值) **作为** 值**来自** (**选择** Y.col **作为** 行 ,**求和** (vec_sca_mult (X.value , Y.value)) **作为** 值**来自** X, Y **其中** X.col =Y.row**按……分组** Y.col) **作为** XtimesY , Z

本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议授权。请访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看该许可协议副本。对于超出本许可协议所涵盖范围的任何使用，请发送电子邮件至 info@vldb.org 获取许可。版权所有者为所有者/作者。出版权已授权给 VLDB基金会。VLDB基金会会刊，第 19 卷，第 6 期，ISSN 2150-8097。doi:10.14778/3797919.3797921

内核 (vec_mat_mult() 和 vec_sca_mult()) 的效率会非常高，而且由于元组被分组到向量和矩阵结构中，中间元组的数量会大幅减少。**X** 和 **Y** 的连接现在只会产生 1000 亿个元组，而第二次连接也只会产生 1000 亿个元组。

此外，通过在关系系统之上运行此计算，内存以及并行化/分布将自动得到管理。分布式数据库系统可以自动在多台机器之间并行化该计算，并管理内存，从而避免现代机器学习编程中普遍存在的“内存不足”错误。

通过大写—小写实现的稀疏张量—关系分解EinSum。在上述示例中，我们选择将 **X** 分解为行向量，将 **Y** 分解为标量，并使 **Z** 保持不分解。本文要解决的问题是：如何将任意张量计算分解为张量关系计算，其中计算密集型部分使用高效内核（例如 vec_mat_mult()）处理，而稀疏性则以关系方式处理？

为应对这一挑战，我们提出爱因斯坦求和记号的一种新的张量关系变体 [4], , 称为大写—小写 EinSum。经典 EinSum 是一种用于表达机器学习和数值计算的标准张量演算。在大写—小写 EinSum 中，表达式明确指定计算的哪些部分以关系方式处理，哪些部分使用高效的数值内核执行。

给定一个 EinSum 计算的有向无环图，将其转换为等价且经过优化的张量-关系计算这一任务，可以归结为自动将表达式重写为等价的大写-小写-大写 EinSum 表达式，以最大化性能。本文提出了一种执行这种重写的算法，称为 SparseEinSum。SparseEinSum 算法将一种简单的成本模型与动态规划方法相结合：前者评估稀疏性条件下大写-小写-大写 EinSum 表达式的质量，后者用于优化重写过程。

我们通过实验展示，SparseEinSum 能生成高性能的张量-关系计算。我们的实验测试平台涵盖了多种稀疏张量工作负载，例如大规模图神经网络、基于图的注意力计算以及量子电路模拟。

2 大写-小写-大写 EINSUM 方法

假设给定一个（可能较为复杂的）数值计算，该计算由一组 EinSum 表达式构成的有向无环图（DAG）指定；我们的目标是生成一种经过优化、可在关系系统上执行的计算。

EinSum 在第 3 节中有详细介绍。概括而言，它要求程序员指定一个简单的声明式表达式，用以描述如何计算输出张量的各个元素。例如，考虑矩阵乘法 $\mathbf{W} \leftarrow \mathbf{X} \times \mathbf{Y}$。在 EinSum 中，这一计算可以表示为：

$$\forall i,k \;\; \mathbf{W}_{i,k} \leftarrow \sum_{j} \mathbf{U}_{i,j} \times \mathbf{V}_{j,k} \tag{1}$$

通过允许使用任意聚合函数和标量函数，并允许对输入张量、输出张量和聚合域进行任意索引，EinSum 能够表达极为广泛的一类张量计算。因此，EinSum 现已广泛应用于机器学习。NumPy [20], 、JAX [18], 、PyTorch [30], 和 Dask [13] 等流行框架都提供了 EinSum 的实现。如果要求程序员直接编写 EinSum 表达式并不现实，他们也可以改用类 PyTorch 语法来指定计算，其中高级运算符（例如矩阵乘法）是底层 EinSum 表示的轻量封装。

EinSum 与 SQL/关系型数据库之间的联系更为紧密，因为每个 EinSum 表达式都可以转换为 SQL，并通过一系列连接后接聚合来实现 [5]。然而，标准的 EinSum 到 SQL 转换隐含地假设所有数据都以完全关系形式存储，即单个标量值存储在元组中。如果处理不当，这种分解可能导致性能不佳。

为此，我们引入大写–小写 EinSum 记法的概念，作为 SparseEinSum 重写的目标表示。在这种记法中，大写字母表示的索引以关系方式处理（即“提升”），而小写字母表示的索引则通过张量索引处理（即“降级”）。例如，考虑两个 8192×8192 矩阵相乘的任务。若将方程 1 写作：

$$\forall i,k \;\; \mathbf{W}_{i,k} \leftarrow \sum_{J} \mathbf{U}_{i,J} \times \mathbf{V}_{J,k}$$

它对应以下张量关系实现：

U (J **整数**, valU **向量** [8192]) V (J **整数**,
valV **向量** [8192])

SELECT SUM(外层_乘积 (U.valU , V.valV)) **AS** valW**FROM** U, V

其中 U .J = V .J

请注意，由于 J 是大写记法中唯一的索引，因此它是唯一出现在 SQL 代码中的索引。索引 i 和 k 由 outer_prod() 核函数隐式处理。输出张量**W** 随后存储为单个元组，其中保存完整的输出 8192 × 8192 矩阵。作为替代方案，我们可以写成：

$$\forall I,K \;\; \mathbf{W}_{I,K} \leftarrow \sum_{j} \mathbf{U}_{I,j} \times \mathbf{V}_{j,K}$$

这对应于以下内容：

U（I **整数**，valU **向量** [8192]）V（
K **整数**，valV **向量** [8192]）

SELECT U.I、V.K、inner_prod（U.valU、V.valV）**AS** valW**FROM** U、V

同样，由于 I 和 K 是以关系方式处理的，而 j 不是，因此 j 不会出现在相应的 SQL 中；它由核函数 inner_prod() 处理，该函数沿着 j 维度遍历向量 valU 和 valV。输出矩阵 **W** 以一组标量的形式存储，并通过 I 和 K 建立索引。

给定大写-小写表示法，核心问题便转变为：如何将 EinSum 表达式的有向无环图（DAG）重写为大写-小写EinSum表达式的有向无环图，以便利用

关系引擎来适当处理稀疏性，并利用高性能内核来适当处理稠密性?

为此，本文其余部分组织如下。下一节更正式地描述EinSum。然后第4节描述大写-小写表示法以及它如何利用稀疏性。第6节描述如何将大写-小写-大写 EinSum 表达式转换为 SQL，从而在现有关系系统上执行。第7节和第8节描述 SparseEinSum 重写算法的两个主要组成部分：用于预测重写为大写-小写-大写 EinSum 之效用的成本模型，以及用于优化重写成本的简单动态规划过程。第9节中的实验评估该方法的效用。

3 EINSUM 背景

本节的前几段介绍 EinSum 以及界和标签投影等概念，取自先前的工作 [8]。

我们首先定义张量的概念。我们使用粗体大写字母（例如，$\mathbf{U}$）来表示张量。将$\mathbf{U}$的界 向量（记为$\mathbf{b}_\mathbf{U}$）定义为长度为 r 的整数向量。 r表示“秩”。矩阵是秩为2的张量。接下来，将 $\mathrm{I}(\mathbf{b}_\mathbf{U})$ 定义为集合 $\{0 \ldots \mathbf{b}_\mathbf{U}[0]-1\} \times \{0 \ldots \mathbf{b}_\mathbf{U}[1]-1\} \times \ldots \times \{0 \ldots \mathbf{b}_\mathbf{U}[r-1]-1\}$。这是所有遵守该界的索引或键的集合。于是，张量$\mathbf{U}$是从 $\mathrm{I}(\mathbf{b}_\mathbf{U})$ 到实数集合的函数。

简单的 `EinSum` 示例。我们从经典示例——矩阵乘法开始。设 $\mathbf{U}$ 和$\mathbf{V}$ 为矩阵，其界分别为 $[100, 200]$ 和 $[200, 50]$,。那么矩阵乘法可表示为： $\forall\, i,\ k \in \mathrm{I}([100, 50])$:

$$\mathbf{W}_{i,k} \leftarrow \sum_{j \in \mathcal{I}([200])} \mathbf{U}_{i,j} \times \mathbf{V}_{j,k} \tag{2}$$

我们通常会省略聚合上的下标，因为它没有必要：根据定义，输入张量中未用于对输出张量进行索引的索引都会被聚合掉。输出张量的界向量同样可由输入张量的界向量隐含确定，因此也可以省略。

这是一种扩展的 EinSum 表示法，因为我们不一定要将输入张量中的元素相乘，也不一定要使用求和。例如，我们可以计算每一行中任意两个元素的平方差的最大值：$\mathbf{W}_i \leftarrow \max(\mathbf{U}_{i,\ j} - \mathbf{V}_{i,\ j})^2$.

EinSum 的一般形式`EinSum`。为了描述 EinSum 的最一般形式，我们定义标签这一概念，它是一个可以绑定到某个值的符号。我们使用 $\ell_\mathbf{U}$ 表示用于对张量$\mathbf{U}$进行索引的标签列表（向量）。

我们还需要定义界向量的投影 和置换 。给定两个标签列表 ℓ_1 和 ℓ_2，以及一个界向量 $\mathbf{b}$，定义 $\mathbf{b}[\ell_1; \ell_2]$为一个长度为 $|\ell_1|$的向量，其中第 i个元素为$\mathbf{b}[j]$当且仅当 $\ell_1[i] = \ell_2[j]$。例如，设$\mathbf{b} = [2, 3, 4]$，且 $\ell_1 = [k, i]$和 $\ell_2 = [i, j, k]$。那么$\mathbf{b}[\ell_1; \ell_2] = [4, 2]$。

据此，二元 EinSum 表达式的一般形式如下：

$$\forall\, \ell_\mathbf{W} \in \mathcal{I}(\mathbf{b}_\mathbf{W}) : \mathbf{W}_{\ell_\mathbf{W}} \leftarrow \bigoplus_{\ell_{\text{agg}} \in \mathcal{I}(\mathbf{b}_{\mathbf{UV}}[\ell_{\text{agg}}; \ell_{\mathbf{UV}}])} \bigotimes (\mathbf{U}_{\ell_\mathbf{U}}, \mathbf{V}_{\ell_\mathbf{V}}) \tag{3}$$

其中， $\bigoplus$ 是聚合算子， $\bigotimes$ 是应用于连接值的标量函数（EinSum 已扩展 ，因为它支持任意的 $\bigoplus$ 和 $\bigotimes$ 操作）。在上述表达式中，我们用 $\ell_{\mathbf{UV}}$表示两个标签列表 $\ell_\mathbf{U}$ 和 $\ell_\mathbf{V}$的拼接。$\mathbf{b}_{\mathbf{UV}}$ 同样表示两个界向量的拼接。

考虑一个更复杂的 EinSum 表达式。假设我们有两个张量 $\mathbf{U}$ 和$\mathbf{V}$，其界向量分别为$\mathbf{b}_\mathbf{U} = [10, 100, 20]$ 和$\mathbf{b}_\mathbf{V} = [100, 20, 2000]$。我们希望对$\mathbf{U}$ 进行转置，得到一个界为 $[20, 10, 100]$, 的张量；然后对$\mathbf{V}$ 进行转置，得到一个界为 $[20, 100, 2000]$, 的新张量；再对所得的两个张量进行批量矩阵乘法，最后对批次维度求和。

在 EinSum 中，这用单个表达式表示为：

$$\forall i, k \in \mathcal{I}([10, 2000]), \mathbf{W}_{i,k} \leftarrow \sum_{b,j \in \mathcal{I}([20,100])} \mathbf{U}_{i,j,b} \times \mathbf{V}_{j,b,k}$$

考虑一般形式，我们有 $\ell_\mathbf{U} = [i, j, b]$、 $\ell_\mathbf{V} = [j, b, k]$、$\ell_{\text{agg}} = [b, j]$ $\mathbf{b}_{\mathbf{UV}} = [10, 100, 20, 100, 20, 2000]$ 以及 $\mathbf{b}$。聚合的界向量计算为 $\mathbf{b}_{\mathbf{UV}}[\ell_{\text{agg}}; \ell_{\mathbf{UV}}]$。如何得出？ ℓ_{agg}有两个标签：b和 j。由于 b占据$\ell_{\mathbf{UV}}$中的第三（及第五）个位置，而 j 占据 $\ell_{\mathbf{UV}}$中的第二（及第四）个位置，我们选取 $\mathbf{b}_{\mathbf{UV}}$ 中的第三（或第五）项，第二（或第四）项则得到 $\mathbf{b}_{\mathbf{UV}}$。由此得到界向量 $[20, 100]$。

多层感知机在`EinSum`中。 借助 EinSum，我们能够通过将各个 EinSum算子组合为算子的有向无环图，来指定非常复杂的计算，包括现代 Transformer。作为此类有向无环图的一个简单示例，下面指定了一个具有两个权重矩阵 $\mathbf{W}^{(1)}_{i,\ j}$ 和 $\mathbf{W}^{(2)}_{i,\ j}$的全连接神经网络，用于将输入连接到隐藏层，然后是 Softmax：

$$\mathbf{A}_i \leftarrow \sum_j \mathbf{W}^{(1)}_{i,j} \times \mathbf{X}_j; \quad \mathbf{B}_i \leftarrow \sum_\emptyset \mathrm{ReLU}(\mathbf{A}_i) \quad \mathbf{C}_i \leftarrow \sum_j \mathbf{W}^{(2)}_{i,j} \times \mathbf{B}_j;$$

$$\mathbf{D}_\emptyset \leftarrow \sum_i \exp(\mathbf{C}_i); \quad \mathbf{E}_i \leftarrow \sum_\emptyset \frac{\exp(\mathbf{C}_i)}{\mathbf{D}_\emptyset}$$

4 稀疏 EinSum

4.1 为什么要关注稀疏性?

EinSum 不限定底层张量的实现方式，但这种实现方式会对实际计算的效率产生显著影响，尤其是在底层张量为稀疏张量时。请看以下两个矩阵：

$$\mathbf{U} \equiv \begin{bmatrix} 1.4 & 2.2 & 0 & 2.1 \\ 0 & 0 & 0 & 0 \\ 1.4 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{V} \equiv \begin{bmatrix} 3.2 & 0 & 1.3 & 0 \\ 0 & 0 & 0.6 & 0 \\ 0 & 0 & 1.2 & 0 \\ 1.2 & 0 & 2.1 & 0 \end{bmatrix}$$

如果我们以经典的稠密格式（例如行主序或列主序）存储这些矩阵，那么大多数经典实现都会按照 EinSum $\mathbf{W}_{i,k} \leftarrow \sum \mathbf{U}_{i,\ j} \times \mathbf{V}_{j,k}$ 所指定的方式执行矩阵乘法，并进行 64 次成对标量乘法。不过，我们可以以关系方式存储这两个矩阵，从而更高效地实现矩阵乘法，至少可以减少所需的乘法次数。考虑将 $\mathbf{U}$ 表示为向量集合，其中只包含零值的向量不显式表示：

$$\tilde{\mathbf{U}} \equiv \left\{ \left(0, \begin{bmatrix} 1.4 & 2.2 & 0 & 2.1 \end{bmatrix}\right), \left(2, \begin{bmatrix} 1.4 & 0 & 1.1 & 0 \end{bmatrix}\right) \right\}$$

$\bar{\mathbf{U}}$ 实现为一个模式为 (i, valU) 的关系，其中 valU 的类型为向量[4]。$\mathbf{V}$ 实现为向量集合：

$$\bar{\mathbf{V}} \equiv \left\{ \left(0, \begin{bmatrix} 3.2 \\ 0 \\ 0 \\ 1.2 \end{bmatrix} \right), \left(2, \begin{bmatrix} 1.3 \\ 0.6 \\ 1.2 \\ 2.1 \end{bmatrix} \right) \right\} \tag{4}$$

模式为 (k, valV)。现在，.可以将相同的矩阵乘法实现为一个简单的SQL查询：该查询在两个关系的笛卡尔积上执行连接，并计算点积：

```
SELECT U.i, V.k,
       点积_ (U.valU , V. valV) AS valW
FROM Ū AS U, V̄ 作为 v
```

请注意，结果是所得矩阵的完全关系型（完全分解）实现，其模式为 (i, k, valW)：

$$\mathbf{W} \equiv \{(0,0,7), (0,2,7.55), (2,0,4.48), (2,2,3.14)\}$$

关键的是，实现这种方式只需要 16 次成对标量乘法，其中 4 次会通过调用 dot_product() 操作执行。因此，所需的标量乘法次数仅为传统方式的 25%。

4.2 提升和降级标签

对于任何关系实现，将每个元组传入系统都会产生开销，而这一开销可能使原始矩阵乘法成为最佳选择。不过，显然存在某种稀疏性程度，在此程度下，由于计算负载较低，这种分解表示更具优势。

对于任何张量，都可以将其以这种形式分解为多种张量关系。考虑一个关于张量 $\mathbf{U}$ 的 EinSum 表达式，其界限为 $\mathbf{b}_\mathbf{U}$，标签列表为 $\ell_\mathbf{U}$。我们通过将 $\ell_\mathbf{U}$ 划分为两个列表 $\uparrow\ell_\mathbf{U}$ 和 $\downarrow\ell_\mathbf{U}$，来描述 $\mathbf{U}$ 的一种分解：

(1) 将 $\uparrow\ell_\mathbf{U}$ 中的标签提升，使其成为结果关系的索引。(2) 将 $\downarrow\ell_\mathbf{U}$ 中的标签降级，使其成为关系内张量的索引。

由 $\uparrow\ell_\mathbf{U}$ 和 $\downarrow\ell_\mathbf{U}$ 诱导的$\mathbf{U}$ 的分解是一个具有模式的数据库关系

$$\bar{\mathbf{U}}(\texttt{INT}\ \uparrow\ell_\mathbf{U}[1],\ \texttt{INT}\ \uparrow\ell_\mathbf{U}[2],\ \ldots,\ \texttt{TENSOR valU})$$

在每个元组中，valU 是一个以$\mathbf{b}_\mathbf{U}[\downarrow\ell_\mathbf{U};\ \ell_\mathbf{U}]$为界的张量，并且该关系中的第$i$个属性 $\uparrow\ell_\mathbf{U}[i]$取从 0 到$\mathbf{b}_\mathbf{U}[\uparrow\ell_\mathbf{U};\ \ell_\mathbf{U}][i] - 1$的整数值。关键在于，假设 $\oplus$和 $\otimes$构成一个半环，如果 valU 仅填充该半环的特殊零值，则可以从所得张量关系中消除相应元组，从而促进稀疏性。

张量$\mathbf{U}$ 在 EinSum 表达式中的所有分解构成的集合，定义了该表达式中$\mathbf{U}$ 的所有张量-关系 实现的一个等价类，因为所有这些张量-关系实现都描述了从 $\mathbf{I}(\mathbf{b}_\mathbf{U})$ 到实数的同一映射。设 $\bar{\mathbf{U}}$ 是由 $\uparrow\ell_\mathbf{U}$ 和 $\downarrow\ell_\mathbf{U}$定义的$\mathbf{U}$的一个分解。那么，对于任意索引$\mathbf{i} \in \mathbf{I}(\mathbf{b}_\mathbf{U})$，以下关系查询返回一个包含标量的单一元组，该标量的值为$\mathbf{U}_\mathbf{i}$处的值：

```
SELECT (ISNULL(valU^i[↓ℓU:ℓU], 零)) FROM (SELECT
valU FROM Ū WHERE ↑ℓU = i[↑ℓU; ℓU]) 左外连接 ZERO
```

这里，我们假设存在一个关系 ZERO(zero)，其中包含一个元组，其值为该半环的零值。内层查询的 WHERE 子句使用提升到表示的关系部分中的标签集合，选择与$\mathbf{i}$对应的元组；随后，外层 SELECT 子句使用剩余的标签索引张量 valU，以获取所需的标量。如果内层查询没有返回结果，则 valU 将为 NULL，查询结果将是相应半环的零值。

以此为具体示例，重新考虑 $\mathbf{V}$、本节中的其（可能的）张量关系表示 $\bar{\mathbf{V}}$ 以及 EinSum $\mathbf{W}_{i,k} \leftarrow \sum_{j} \mathbf{U}_{i,j} \times \mathbf{V}_{j,k}$。在此情况下，$\bar{\mathbf{V}}$ 对应于分解 $\uparrow\ell_\mathbf{V} = k$、$\downarrow\ell_\mathbf{V} = j$。考虑索引向量 $\mathbf{i} = [3,2]$。在此情况下，$\mathbf{i}[\downarrow\ell_\mathbf{V};\ \ell_\mathbf{V}] = \mathbf{i}[j;\ jk] = 3$ 以及 $\mathbf{i}[\uparrow\ell_\mathbf{V};\ \ell_\mathbf{V}] = \mathbf{i}[k;\ jk] = 2$。因此，要在 $\bar{\mathbf{V}}$ 中获取 $\mathbf{V}[3,2]$，我们将使用以下查询：

```
SELECT (ISNULL(valV3, zero)) FROM (SELECT
valV FROM V̄ WHERE k = 2) 左外连接 ZERO
```

4.3 “大写-小写”记法

我们不建议显式列出$\uparrow\ell_\mathbf{U}$和 $\downarrow\ell_\mathbf{U}$，而是约定：在张量关系上的 EinSum 表达式中，大写 标签属于 $\uparrow\ell_\mathbf{U}$，小写 标签属于 $\downarrow\ell_\mathbf{U}$。这定义了大写-小写 记法。因此，考虑以下用于张量关系矩阵乘法的 EinSum 表达式：

$$\mathbf{W}_{i,K} \leftarrow \sum \mathbf{U}_{i,J} \times \mathbf{V}_{J,K}$$

在这种情况下，标签 J，K用于索引关系，而标签 i用于索引张量。因此，这对应于如下实现：$\mathbf{U}$ 表示为集合：

$$\left\{ \left(0, \begin{bmatrix} 1.4 \\ 0 \\ 1.4 \\ 0 \end{bmatrix} \right), \left(1, \begin{bmatrix} 2.2 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right), \left(2, \begin{bmatrix} 0 \\ 0 \\ 1.1 \\ 0 \end{bmatrix} \right), \left(3, \begin{bmatrix} 2.1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) \right\}$$

而 $\mathbf{V}$ 表示为集合：

$$\{(0,0,3.2), (0,2,1.3), (1,2,0.6), (2,2,1.2), (3,0,1.2), (3,2,2.1)\}$$

而输出 $\mathbf{W}$ 则仍表示为一组列向量，如公式 4 所示。

再举一个例子，考虑引言中的分解（用于实现图神经网络中消息传递的两个乘法构成的链）。按照将提升为关系表示的那些索引以大写字母表示这一惯例，可以通过 EinSum 指定如下：

$$\mathbf{T}_{K,i} \leftarrow \sum \mathbf{X}_{i,J} \times \mathbf{Y}_{J,K};\ \mathbf{U}_{I,k} \leftarrow \sum \mathbf{T}_{I,j} \times \mathbf{Z}_{j,k}$$

5 分解 EINSUM

我们可以将任意二元 EinSum 表达式分解为在张量关系上的等价计算。这种等价性为以关系方式在稀疏输入上实现张量关系 EinSum 提供了路线图。按照本文上一节的约定，$\bar{\mathbf{U}}$ 和 $\bar{\mathbf{V}}$ 分别用于表示 $\mathbf{U}$ 和 $\mathbf{V}$ 的张量关系分解。现在，假设我们有一个关联的 核函数 K。这是一个实现该 EinSum 的函数：

$$\mathbf{W}_{\downarrow\ell_\mathbf{W}} \leftarrow \bigoplus \bigotimes \left(\mathbf{U}_{\downarrow\ell_\mathbf{U}}, \mathbf{V}_{\downarrow\ell_\mathbf{V}}\right)$$

将方程 3 重写为：

$$\mathbf{W}_{\ell_W} \leftarrow \bigoplus_{\downarrow\ell_{agg},\uparrow\ell_{agg}} \bigotimes \left(\left(\sigma_{\uparrow\ell_U}(\bar{\mathbf{U}})\right)_{\downarrow\ell_U}, \left(\sigma_{\uparrow\ell_V}(\bar{\mathbf{V}})\right)_{\downarrow\ell_V} \right) \quad (5)$$

这里再引入一点额外记号。在 EinSum 表达式的执行过程中，给定将 $\uparrow\ell_U$ 中的标签绑定到向量 $\mathbf{i}$ 中的值，$\sigma_{\uparrow\ell_U}(\bar{\mathbf{U}})$ 表示关系选择的应用，等价于运行如下查询：

SELECT ISNULL（valU，zero）**FROM**（**SELECT** valU **FROM** $\bar{\mathbf{U}}$ **WHERE** $\uparrow\ell_U = \mathbf{i}$）**左外连接** ZERO

请注意，zero 是一个由零填充、边界为 $\mathbf{b}[\downarrow\ell_U; \ell_U]$ 的张量。此外，$\uparrow\ell_{agg}$ 和 $\downarrow\ell_{agg}$ 表示原始表达式中按两个列表划分的聚合标签，具体取决于它们出现在 $\uparrow\ell_U$、$\uparrow\ell_V$ 中，还是出现在 $\downarrow\ell_U$、$\downarrow\ell_V$ 中。

据此，方程 5 可重写为：

$$\mathbf{W}_{\ell_W} \leftarrow \bigoplus_{\uparrow\ell_{agg}} \left(\left(\sigma_{\uparrow\ell_W,\uparrow\ell_{agg}} \left(\bowtie \left(\bar{\mathbf{U}}, \bar{\mathbf{V}}, \text{valW} \leftarrow \mathcal{K}(\text{valU}, \text{valV}) \right) \right) \right)_{\downarrow\ell_W} \right)$$

这里，$\bowtie(\bar{\mathbf{U}}\ \bar{\mathbf{V}}$，valW $\leftarrow$ K(。)) 对关系 $\bar{\mathbf{U}}$ 和 $\bar{\mathbf{V}}$ 执行自然连接，并使用核函数 K 计算附加属性 valW。以关系方式定义 $\bigoplus_{\uparrow\ell_{agg}}$，使其通过对一对张量中的每个元素逐一应用 $\oplus$ 运算符来聚合存储在 valW 中的张量，则该表达式可进一步重写为 $\mathbf{W}_{\ell_W} \leftarrow$

$$\left(\sigma_{\uparrow\ell_W} \left(\bigoplus_{\uparrow\ell_{agg}} \left(\bowtie \left(\bar{\mathbf{U}}, \bar{\mathbf{V}}, \text{valW} \leftarrow \mathcal{K}(\text{valU}, \text{valV}) \right) \right) \right) \right)_{\downarrow\ell_W} \quad (6)$$

这表明，对于张量关系上的任意二元 EinSum 表达式，都可以采用如下张量关系实现：

选择 $\uparrow\ell_W$，**求和**（$\mathcal{K}$(valU, valV)）**作为** valW **从** $\bar{\mathbf{U}}$ **自然连接** $\bar{\mathbf{V}}$ **分组依据** $\uparrow\ell_W$

6 将 EINSUM 编译为张量关系 SQL

这表明，可以构建一个框架，将 EinSum 表达式组成的有向无环图编译为张量关系代数或张量关系 SQL。

(1) 对于有向无环图中的每个顶点（EinSum 表达式），将 EinSum 重写为大写-小写-大写 EinSum。

(2) 根据公式 6，为每个生成的表达式生成 SQL。也就是说，生成一个 SELECT-FROM-WHERE-GROUP BY 查询，其中 (a) SELECT 中选取的属性是 $\uparrow\ell_W$ 中的那些标签，以及“SUM(K (valU，valV)) as valW”；(b) WHERE 子句对两个输入关系执行自然连接；(c) 按照对应于 $\uparrow\ell_W$ 中标签的属性进行分组。

（3）如果不同分解后的 EinSum 表达式之间的张量关系分解存在不匹配，则生成张量关系代数/张量关系 SQL，以改变表示形式。

最后一点值得讨论。例如，假设我们生成了如下分解：

$$\mathbf{T}_{K,i} \leftarrow \sum \mathbf{X}_{i,J} \times \mathbf{Y}_{J,K} \qquad \mathbf{U}_{i,k} \leftarrow \sum \mathbf{T}_{i,J} \times \mathbf{Z}_{J,k}$$

第一个表达式对应的 SQL 如下：

SELECT K, **SUM**（$\mathcal{K}_1$(valX, valY)）**as** valT
FROM $\bar{\mathbf{X}}$ **自然连接** $\bar{\mathbf{Y}}$ **GROUP BY** K

第二个表达式对应的 SQL 如下：

SELECT SUM(K_2(valT, valZ)) **as** valU
FROM $\bar{\mathbf{T}}$ **NATURAL JOIN** $\bar{\mathbf{Z}}$

第一个 EinSum 的 $\mathbf{T}$ 输出分解（由行向量组成，因为第一个 [row] 索引是大写）与第二个 EinSum 的 $\mathbf{T}$ 输入分解（由列向量组成，因为第二个 [column] 索引是大写）不匹配。因此，必须生成代码，在两项操作之间重新分区 $\mathbf{T}$。假设第一个 EinSum 表达式的输出按 $\mathbf{T}_{K,i}$进行分区，并命名为 $\bar{\mathbf{T}}_{\text{in}}$，而我们希望按 $\mathbf{T}_{i,J}$ 重新分区，以形成 $\bar{\mathbf{T}}$。我们首先沿第一个维度进行分解：

创建视图 $\bar{\mathbf{T}}_{\text{int}}$(i, j, 张量值) **作为 SELECT** $\bar{\mathbf{T}}_{\text{in}}$.k, A.**索引**，$\bar{\mathbf{T}}_{\text{in}}$.张量值[A.**索引**] **FROM** ALLINTS(0, $\mathbf{b}_T$[1]) **作为** A，$\bar{\mathbf{T}}_{\text{in}}$

在这段代码中，$\bar{\mathbf{T}}_{\text{in}}$.valT[A.index] 会从 $\bar{\mathbf{T}}_{\text{in}}$.valT 中提取 A.index 位置处的子张量（此处为一个标量）。ALLINTS 是一个返回指定范围内所有值的表函数。然后，我们沿第二个维度重新组合：

创建视图 $\bar{\mathbf{T}}$(j, 张量值) **作为 SELECT** $\bar{\mathbf{T}}_{\text{int}}$.j, STACK($\bar{\mathbf{T}}_{\text{int}}$.张量值，$\bar{\mathbf{T}}_{\text{int}}$.i, 0, $\mathbf{b}_T$[0]) **FROM** $\bar{\mathbf{T}}_{\text{int}}$ **GROUP BY** $\bar{\mathbf{T}}_{\text{int}}$.j

这里，STACK 是一个聚合函数，可用于将标量（或张量）堆叠起来，从而创建更高维的张量。它接受：(1) 要堆叠的值，(2) 该值在更高维张量中的位置，(3) 要创建的新维度，以及 (4) 该维度的最大值。

一般情况下，从张量关系 $\bar{\mathbf{T}}_{\text{in}}$ 到 $\bar{\mathbf{T}}$ 的重新分区分两步进行。首先，对第一个 EinSum 表达式的结果进行分解，使其分区到 $\uparrow\ell_{\mathbf{T}_{\text{in}}}$ 或 $\uparrow\ell_{\mathbf{T}}$中的所有标签上。然后进行“取消分区”或聚合，以得到 $\bar{\mathbf{T}}$。请注意，如果 $\uparrow\ell_{\mathbf{T}_{\text{in}}}$ 是 $\uparrow\ell_{\mathbf{T}}$的子集，则可以跳过聚合；如果 $\uparrow\ell_{\mathbf{T}}$ 是 $\uparrow\ell_{\mathbf{T}_{\text{in}}}$ 的子集，则可以跳过分解。如果 $\uparrow\ell_{\mathbf{T}_{\text{in}}} = \uparrow\ell_{\mathbf{T}}$，则可以跳过整个重新分区，因为分区方式相同。

7 稀疏 EinSum 成本模型

这就留下了两个尚待解决的问题，因为我们的目标是找到最佳分解。(1) 如何以考虑稀疏性的方式评估给定分解的成本？(2) 如何高效地搜索所有分解构成的空间？本节将讨论第一个问题。

7.1 稀疏性条件下的规模估算

在开发成本模型时，我们首先必须解决的问题是：给定一个（可能是稀疏的）张量 $\mathbf{U}$，其标签集为 ℓ_U，以及由 $\uparrow\ell_U$ 和 $\downarrow\ell_U$定义的张量关系分解 $\bar{\mathbf{U}}$，其中

包含多少个元组？也就是说，如果我们将一个张量分解为张量关系，能否估算该关系中包含多少个元组？我们用记号 $T(\bar{\mathbf{U}})$来表示这个量。

我们首先定义两个统计量，它们足以准确估计 $T(\bar{\mathbf{U}})$。首先，对于 $\ell_{\mathbf{U}}$中的标签 l，令 $V(l, \mathbf{U})$表示由 l诱导且至少包含一个非零项的子张量数量，前提是我们使用 $\uparrow\ell_{\mathbf{U}} = \{l\}$。换句话说，$V(l, \mathbf{U})$回答了这样一个问题：如果通过以关系方式表示标签来分解$\mathbf{U}$，我们会得到多少个元组？

确切地说，令 $\ell'_{\mathbf{U}}$ 表示移除了 l的 $\ell_{\mathbf{U}}$。那么，可以使用以下 EinSum 表达式计算 $V(l, \mathbf{U})$：

$$\mathbf{V}_l \leftarrow \sum_{\ell'_{\mathbf{U}}} \texttt{if}(\mathbf{U}_{\ell_{\mathbf{U}}} == 0, 0, 1)$$

$$V(l, \mathbf{U}) \leftarrow \sum_{l} \texttt{if}(\mathbf{V}_l == 0, 0, 1)$$

这里，函数“if (bool, val1, val2)”在 bool 为 true 时返回 val1，否则返回 val2。

请注意，$V(l, \mathbf{U})$在某种意义上等价于关系查询优化中常用的属性不同值数量这一概念。为什么呢？设想$\mathbf{U}$ 完全以关系方式表示——也就是说，我们对$\mathbf{U}$ 进行张量关系分解，将其分解为 $\bar{\mathbf{U}}$，使得 $\uparrow\ell_{\mathbf{U}} = \ell_{\mathbf{U}}$ 和 $\downarrow\ell_{\mathbf{U}} = \{\}$，并移除存储标量值零的所有元组。现在考虑属性 l。如果 l在 $\bar{\mathbf{U}}$中有 d 个不同值，这意味着，若将具有相同 l 值的所有元组分组到$\mathbf{U}$的一个子张量中，则恰好会有 d 个子张量，且这些子张量都不会完全由零组成（纯零子张量不会为 $\bar{\mathbf{U}}$贡献任何元组，因此也不会计入 d）。因此，$V(\mathbf{U}\ l)$将为 d.

此外，假设我们有 $T(\mathbf{U})$，即张量$\mathbf{U}$中的非零条目数。同样，这类似于关系中的元组数。如果我们对$\mathbf{U}$执行纯关系分解并移除存储零值的元组，则张量$\mathbf{U}$ 中的非零条目数将与此类纯关系分解中的元组数相同。

令 $n(\uparrow\ell_{\mathbf{U}})$表示由提升标签集 $\uparrow\ell_{\mathbf{U}}$ 诱导的、针对$\mathbf{U}$的某个张量关系分解 $\bar{\mathbf{U}}$ 中可能的元组数。这可以计算为：

$$n(\uparrow\ell_{\mathbf{U}}) = \prod_{l \in \uparrow\ell_{\mathbf{U}}} V(l, \mathbf{U})$$

则由提升标签集 $\uparrow\ell_{\mathbf{U}}$ 诱导的张量关系分解$\bar{\mathbf{U}}$ 中的元组数可估计为：

$$T(\uparrow\ell_{\mathbf{U}}) = n(\uparrow\ell_{\mathbf{U}}) \left(1 - \exp\left(\frac{-T(\mathbf{U})}{n(\uparrow\ell_{\mathbf{U}})}\right)\right) \tag{7}$$

其直观含义是，张量关系 $\bar{\mathbf{U}}$中有 $n(\uparrow\ell_{\mathbf{U}})$个可能的元组。若 $T(\mathbf{U})$个非零值在张量关系中均匀随机分布，则 $T(\uparrow\ell_{\mathbf{U}})$估计其中张量至少包含一个非零值的元组数。

7.2 成本模型

以张量关系式地实现方程 3 形式的通用二元 EinSum 表达式时，需要执行两项操作，即连接和聚合。还有第三项操作——重新分区——可能有必要在一个操作的输出与下一个操作的输入之间转换张量关系分解。我们将探讨如何估算每项操作的成本。

连接成本估算。参与关系连接 ℓ_{join} 的标签集合，是同时出现在 $\uparrow\ell_{\mathbf{U}}$ 和 $\uparrow\ell_{\mathbf{V}}$中的标签列表。张量关系连接所产生的元组数量可估算为：

$$T_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) = \frac{T(\uparrow\ell_{\mathbf{U}})T(\uparrow\ell_{\mathbf{V}})}{\prod_{l \in \ell_{\text{join}}} \max(V(\mathbf{U}, l), V(\mathbf{V}, l))}$$

这是对连接所产生元组数量进行估算时采用的标准教科书式估算器。请注意，如果 $\uparrow\ell_{\mathbf{U}}$ 和 $\uparrow\ell_{\mathbf{V}}$ 没有共同标签，因此 ℓ_{join} 为空，则假定分母取值为 1。然后，张量关系连接的成本估算为：

$$C_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) = T_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) \times \left[(\texttt{sizeof}(\bar{\mathbf{U}}) + \texttt{sizeof}(\bar{\mathbf{V}}))C_{\text{xfer}} + C_{\text{krnel}} + C_{\text{fixed}}\right]$$

这里，sizeof 用于计算张量关系中一个元组的大小，单位为字节；C_{xfer} 是将一个元组从一台机器传输到另一台机器的每字节成本（假设处于分布式环境中）；C_{krnel} 是调用内核 K 的成本（通常为 K 所需的浮点运算次数乘以一个常数）；C_{fixed} 是固定的单个元组处理成本。总体而言，成本为：(a) 将两个输入中的元组传输到连接操作中的成本；加上 (b) 运行内核的成本；再加上 (b) 每个元组的固定成本。

一种经验法则是：通过检查在机器之间传输一个字节所需的时间（或在单机环境中检查从磁盘向机器传输一个字节所需的时间）来选择比率 C_{xfer}；通过检查在执行平台上运行一次浮点计算所需的时间来选择 C_{krnel}；再通过运行单机计算来选择 C_{fixed}。该计算会生成并聚合大量元组（例如，计算两个关系的笛卡尔积并聚合结果），然后将运行时间除以生成的元组数，以得到每元组开销。

聚合成本估算。后续聚合产生的元组数可估算为

$$T_{\text{agg}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) = \min\left(\frac{1}{2}T_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}), \prod_{l \in \uparrow\ell_{\text{agg}}} \begin{cases} \text{if } l \in \uparrow\ell_{\mathbf{U}} \wedge l \in \uparrow\ell_{\mathbf{V}} : \min(V(\mathbf{U}, l), V(\mathbf{V}, l)) \\ \text{else if } l \in \uparrow\ell_{\mathbf{U}} : V(\mathbf{U}, l) \\ \text{else} : V(\mathbf{V}, l) \end{cases}\right)$$

这是对标准教科书式估算器的一种变体，用于估算聚合产生的元组数量。请注意“min”第二个选项中的三种情况。每个提升后的分组标签的不同值数量为以下之一：(a) 如果该标签是连接属性，则取$\mathbf{U}$ 和$\mathbf{V}$ 中不同值数量的最小值；(b) 如果该标签仅来自$\mathbf{U}$，则取$\mathbf{U}$ 中的不同值数量；(c) 否则，取$\mathbf{V}$中的不同值数量。然后，聚合的成本估算如下：

$$C_{\text{agg}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) = (T_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) - T_{\text{agg}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}})) \times \left[\texttt{sizeof}(\bar{\mathbf{W}})C_{\text{xfer}} + C_{\text{add}} + C_{\text{fixed}}\right]$$

此公式与 C_{join}的公式相似，唯一的区别是聚合的输出为 $T_{\text{agg}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}})$"聚合组"；要将一个聚合组中的$\tau$连接（$\Upsilon_{\mathbf{U}}\ \Upsilon_{\mathbf{V}}$）$\tau$ agg($\Upsilon_{\mathbf{U}}\ \Upsilon_{\mathbf{V}}$)个元组缩减为单个元组，需要移动并相加$\tau$连接（$\Upsilon_{\mathbf{U}}\ \Upsilon_{\mathbf{V}}$）$\tau$agg($\Upsilon_{\mathbf{U}}\ \Upsilon_{\mathbf{V}}$) -1个元组，即总共进行 $T_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) - T_{\text{agg}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}})$次移动和相加。

重新分区的成本。最后，当生成张量关系 $\bar{\mathbf{W}}$ 的表达式的输出分区与 $\bar{\mathbf{W}}$的使用方输入分区不匹配时，还需要计算对张量关系 $\bar{\mathbf{W}}$ 重新分区的成本。

令 $\uparrow\ell_{\mathbf{W}_{\text{in}}}$ 为重新分区操作的输入中提升标签的集合，令 $\uparrow\ell_{\mathbf{W}}$ 为目标消费 EinSum 表达式中的提升标签集合。

如上一节给出的 SQL 所示，重新分区的典型实现方式是先对第一个 EinSum 表达式的结果进行分解，使其按 $\uparrow\ell_{\mathbf{W}_{\text{in}}}$ 或 $\uparrow\ell_{\mathbf{W}}$ 中的所有标签进行分区（虽然 $\uparrow\ell_{\mathbf{W}_{\text{in}}}$ 和 $\uparrow\ell_{\mathbf{W}}$ 都是列表而非集合，我们仍滥用记号，将同时包含两者全部标签的列表记为 $\uparrow\ell_{\mathbf{W}_{\text{in}}} \cup \uparrow\ell_{\mathbf{W}}$）。可以使用 $T(\uparrow\ell_{\mathbf{W}_{\text{in}}} \cup \uparrow\ell_{\mathbf{W}})$ 估算所得张量关系中的元组数量。请注意，该数量不能小于 $T(\uparrow\ell_{\mathbf{W}})$ 或 $T(\uparrow\ell_{\mathbf{W}_{\text{in}}})$，因为中间分解使用的提升标签集合是输入和输出提升标签集合的超集。

然后便可以计算出成本的表达式 C_{repart}：

$$
\begin{aligned}
&C_{\text{repart}}(\uparrow\ell_{\mathbf{W}_{\text{in}}}, \uparrow\ell_{\mathbf{W}}) = \\
&\quad \big(T(\uparrow\ell_{\mathbf{W}_{\text{in}}} \cup \uparrow\ell_{\mathbf{W}}) - T(\uparrow\ell_{\mathbf{W}_{\text{in}}})\big)\big(\texttt{sizeof}(\bar{\mathbf{W}}_{\text{int}})C_{\text{xfer}} + C_{\text{fixed}}\big) + \\
&\quad \big(T(\uparrow\ell_{\mathbf{W}_{\text{in}}} \cup \uparrow\ell_{\mathbf{W}}) - T(\uparrow\ell_{\mathbf{W}})\big)\big(\texttt{sizeof}(\bar{\mathbf{W}}_{\text{int}})C_{\text{xfer}} + C_{\text{fixed}}\big) \\
&\quad = \big(2T(\uparrow\ell_{\mathbf{W}_{\text{in}}} \cup \uparrow\ell_{\mathbf{W}}) - T(\uparrow\ell_{\mathbf{W}_{\text{in}}}) - T(\uparrow\ell_{\mathbf{W}})\big)\times \\
&\quad \big(\texttt{sizeof}(\bar{\mathbf{W}}_{\text{int}})C_{\text{xfer}} + C_{\text{fixed}}\big)
\end{aligned}
$$

C_{repart} 分为两部分。第一部分是第一步分解过程中创建和处理$T(\uparrow\ell_{\mathbf{W}_{\text{in}}}\cup\uparrow\ell_{\mathbf{W}})-T(\uparrow\ell_{\mathbf{W}_{\text{in}}})$个新元组的成本。第二部分则是将这些元组合并为下一项操作的 $T(\uparrow\ell_{\mathbf{W}})$个输入；这需要移动 $T(\uparrow\ell_{\mathbf{W}_{\text{in}}} \cup \uparrow\ell_{\mathbf{W}})$ 个 $T(\uparrow\ell_{\mathbf{W}})$中间元组。

算法1 ComputeMinCostForTree

按拓扑顺序 对输入 EinSum 图 G的顶点排序
初始化 查找成本表 C，使得对于任意一个没有输入的顶点，$C[v, \uparrow\ell_{\mathbf{W}}] = 0$ 对于所有 $\uparrow\ell_{\mathbf{W}} \in \ell_{\mathbf{W}}$，其中 $\ell_{\mathbf{W}}$ 是由$\mathbf{W}$ 张量生成的标签列表 v
对于 每个顶点 $v \in V$, 按排序顺序**执行**
  % 输入张量的代价始终为 0
  如果 v 没有输入**则**
    继续
  结束如果
  求解 $v_{\mathbf{U}}$，使得 $(v_{\mathbf{U}}, v, \text{左}) \in E$
  求解 $v_{\mathbf{V}}$，使得 $(v_{\mathbf{V}}, v, \text{右}) \in E$
  $cost_{\min} \leftarrow \infty$
  提取 $\ell_{\mathbf{U}}, \ell_{\mathbf{V}}, \ell_{\mathbf{W}}$ 自顶点 v
  % 此处将考虑 $\mathbf{W}$ 的每一种输出分解
  对于 每个子集 $\uparrow\ell_{\mathbf{W}} \in \ell_{\mathbf{W}}$ **执行**
    % $\uparrow\ell_{\mathbf{U}}$ 与 $\uparrow\ell_{\mathbf{W}}$ 一致，若对于 $(\ell_{\mathbf{U}} \cap \ell_{\mathbf{W}})$ 中的每个 l，
    % l属于 $\uparrow\ell_{\mathbf{U}}$ 当且仅当它也属于 $\uparrow\ell_{\mathbf{W}}$
    对于每个与 $\uparrow\ell_{\mathbf{W}}$ 一致的子集 $\uparrow\ell_{\mathbf{U}} \in \ell_{\mathbf{U}}$ **执行**
      $cost_{\mathbf{U}} \leftarrow$最小值$_{\uparrow\ell_{\mathbf{U}}\text{在}}$ $C_{\text{repart}}(\uparrow\ell_{\mathbf{U}_{\text{in}}}, \uparrow\ell_{\mathbf{U}}) + C[v_{\mathbf{U}}, \uparrow\ell_{\mathbf{U}_{\text{in}}}]$
      对于每个与 $\uparrow\ell_{\mathbf{W}}$ 一致的子集 $\uparrow\ell_{\mathbf{V}} \in \ell_{\mathbf{V}}$ **执行**
        $cost_{\mathbf{V}} \leftarrow$最小值$_{\uparrow\ell_{\mathbf{V}}\text{在}}$ $C_{\text{repart}}(\uparrow\ell_{\mathbf{V}_{\text{in}}}, \uparrow\ell_{\mathbf{V}}) + C[v_{\mathbf{V}}, \uparrow\ell_{\mathbf{V}_{\text{in}}}]$
        $cost_{\mathbf{W}} \leftarrow cost_{\mathbf{U}} + cost_{\mathbf{V}} + C_{\text{join}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}}) + C_{\text{agg}}(\uparrow\ell_{\mathbf{U}}, \uparrow\ell_{\mathbf{V}})$
        如果 $cost_{\mathbf{W}} < cost_{\min}$ **那么**
          $cost_{\min} \leftarrow cost_{\mathbf{W}}$
        结束如果
      结束循环
    结束循环
    % 记住，通过分解计算 $\mathbf{W}$ 的低成本 $\uparrow\ell_{\mathbf{W}}$
    $C[v, \uparrow\ell_{\mathbf{W}}] \leftarrow cost_{\min}$
  结束循环
结束循环
令 v 为 V中的最后一个顶点
返回最小值$_{\uparrow\ell_{\mathbf{W}}}$ $C[v, \uparrow\ell_{\mathbf{W}}]$

8 优化分解

现在，我们定义一种动态规划算法，用于计算有向无环图中 EinSum 表达式的最低成本、稀疏感知分解。该算法将有向无环图中的每个表达式重写为大写-小写表示法。结合第5节的结果，我们便可以将此类 EinSum 程序编译为高性能的张量-关系计算。

8.1 动态规划

考虑一个有向图 $G = V, E$，其中顶点集 V中的每个顶点都表示一个 EinSum 运算。V有两种类型的顶点：（1）没有输入的顶点（对应输入张量）；（2）有两个输入的顶点（对应二元 EinSum 表达式）。对于我们的优化算法，每个 EinSum 运算的重要特征是表达式中出现的标签集 $\ell_{\mathbf{U}}$、$\ell_{\mathbf{V}}$、$\ell_{\mathbf{W}}$，以及输入张量和输出张量的界向量。这些信息将使我们能够利用上一节中的成本模型。

E 中的每条有向边都是一个三元组，由以下部分组成：(1) 生成张量的顶点/EinSum 表达式；(2) 消耗该张量的顶点/EinSum 表达式；以及 (3) 一个常量，取值为 left 或 right，用于指明该张量是消耗顶点的左输入还是右输入。

动态规划算法会遍历图，并按拓扑排序顺序依次处理各个顶点。在此过程中，它会填充查找表 C。对于生成标签列表为 $\ell_{\mathbf{W}}$ 的张量的顶点 v，每一种可能的提升标签集合 $\uparrow\ell_{\mathbf{W}}$ 都会计算出 $C[v, \uparrow\ell_{\mathbf{W}}]$——也就是 $\ell_{\mathbf{W}}$ 的所有可能子集。$C[v, \uparrow\ell_{\mathbf{W}}]$存储生成张量 $\mathbf{W}$ 的最小可能成本——其中考虑了计算 $\mathbf{W}$ 所需的所有 EinSum 表达式——且须满足以下约束：输出张量的张量关系分解由 $\uparrow\ell_{\mathbf{W}}$描述。

由于每个张量都只被一个 EinSum 表达式消耗，因此可以通过动态规划算法一次遍历有向无环图来填充 C：我们不关心与顶点 v 关联的输出是如何计算的具体细节；只关心 (a) 由 v产生的输出分解，以及 (b) 计算 v的成本。只要 C 存储最低成本以

算法2获取统计量

1：将输入 EinSum 有向无环图编译为基于仅存储标量的关系的 SQL——也就是说，对于每个张量**U**，令 $\uparrow\ell_{\mathbf{U}} = \ell_{\mathbf{U}}$ 和 $\downarrow\ell_{\mathbf{U}} = \{\}$，并在假设存储零值标量的所有元组都从所得关系中删除的情况下计算统计信息 2：使用经典方法将纯关系 SQL 编译为关系代数 3：使用经典方法对关系代数进行**成本估算**，以计算/估计每个输入关系和中间关系中的元组数量；对于对应于张量 **U** 的关系 $\tilde{\mathbf{U}}$，令 $T(\mathbf{U})$ 表示该元组计数 4：使用经典方法对关系代数进行**成本估算**，以计算/估计每个属性的不同值数量；对于属性 l 和关系 $\tilde{\mathbf{U}}$，令 $V(\mathbf{U}\ l)$ 表示该值计数

数据集	类别	特征	节点	边
Cora	7	1,433	2,485	5,069
Planetoid.Cora [43]	7	1,433	2,708	10,556
Planetoid .CiteSeer [43]	6	3,703	3,327	9,104
CitationFull .CiteSeer [7]	6	602	4,230	10,674
Amazon.Photo [34]	8	745	7,650	0.2M
亚马逊 .计算机 [34]	10	767	13,752	0.4M
ogbn-arxiv* [21]	40	128	169,343	1.1M
ogbn-products* [21]	47	100	2.4M	61.8M
ogbn-papers100M* [21]	172	128	111M	1.6B
Friendster* [9]	100	128	65M	1.8B

表 1：图标准化后数据集的统计信息。标有 * 的数据集用于分布式执行。

要计算与顶点 v 对应的 EinSum 表达式的所有分解，为了优化 v 的使用方的分解，我们只需查看所有形如 $C[v, .]$ 的条目。

动态规划算法如算法1所示。对于每个顶点 v，都有一个三重嵌套循环。在最外层，我们为 v 的输出计算每种可能的张量关系分解的最优代价，这些分解由提升标签集合 $\uparrow\ell_{\mathbf{W}}$ 描述。目标是计算为 v 生成 $\uparrow\ell_{\mathbf{W}}$ 的最低代价。两个内层循环考虑所有与 $\uparrow\ell_{\mathbf{W}}$ 一致的可能输入分解。如果 $\uparrow\ell_{\mathbf{W}}$ 将标签 l 提升为以关系方式处理，那么两个输入都必须提升该标签。通过考虑所有这些一致的输入组合，即可计算出 $C[v, \uparrow\ell_{\mathbf{W}}]$ 的最优值。

8.2 补充考虑

预先计算所需统计量。 在实际应用中，运行算法1需要为每个张量 **U** 提供 T（**U** 中非零条目的数量），并为每个标签 l 提供 $V(\mathbf{U}\ l)$（由标签 l 导出的非零张量数量）。这些统计量可以通过运行 GetStats 算法（算法2）获得。在最高层面上，该算法依赖于为输入 EinSum 程序的完全关系化版本生成 SQL，然后使用经典的关系代价估算方法计算所需统计量。

当结果被多次使用时。在这种情况下，之前的动态规划算法无法奏效，因为该算法依赖于条目集合 $C[v, .]$ 能够完整描述所有必要信息，从而最优地使用与顶点 v 对应的 EinSum 表达式。然而，机器学习计算中的数据（包括输入数据和中间数据）往往会被多次使用。例如，在反向传播过程中，中间数据在前向和反向传播过程中都会被使用。在我们的实验中，无论是图卷积神经网络还是注意力计算，输入数据都会用于两个 EinSum 表达式。如果 v 被使用多次，最优算法就需要同时考虑 v 的两种用法，并对它们进行协同优化。

我们所实现的方法是将一个 EinSum 程序分解为一组“树”。也就是说，我们首先选择程序 G 的一个子图，其中没有任何 EinSum 结果被使用超过一次（一般而言，该子图应当是“极大的”，即如果我们还能再添加另一个顶点，并且仍然没有任何顶点被使用超过一次，我们就应当这样做）。然后，我们使用算法1对该子图进行优化。此时，每个已被优化的张量的张量关系分解被视为固定不变。随后重复这一过程，再选择并优化程序 G 中另一个没有任何 EinSum 结果被使用超过一次的子图，直到所有分解均已选定。

这不会得出全局最优解，而且某些被使用两次的张量关系分解可能不会被选为全局最优。不过，由于这仅会导致次优结果，额外的重新分区所带来的这种非最优性的预期代价应该无关紧要。

内存约束。 通常，从性能角度来看，最优的模式设计是只使用大型张量，而完全不对张量进行分解。实际应用中，这会导致内存问题。在我们的实现中，我们通过限制搜索范围来解决这一问题，使其反映内存限制——例如，对内核调用的输入和输出所需的总字节数设置上限。

非二元`EinSum`表达式。 本文迄今描述的所有算法和成本模型都可以轻松扩展到一元EinSum表达式，因此我们不再进一步讨论这一问题。

高阶EinSum表达式更具挑战性，因为收缩路径对性能至关重要。虽然这类表达式在机器学习计算中并不常见，但确实会出现，尤其是在其他应用领域中。我们建议使用优化库（如 opt_einsum [12]）来计算收缩路径，然后将这类表达式拆分为一系列二元 EinSum 表达式。更复杂的方法将留待未来研究。

9 实验

9.1 实验概述

通过实验，我们希望回答以下问题：*SparseEinSum* 生成的大小写混合爱因斯坦记号，与传统的张量方法或纯关系方法相比，是否具有更好的可扩展性？

数据集和机器数量		基于张量的		SparseEinSum		
		DGL	Ali	**Opt**	**2nd**	**3rd**
ogbn-arxiv	1	**4.3**	12.9	8.3	13.4	21.8
	2	**1.6**	6.9	5.2	8.8	12.6
	4	**0.9**	3.4	2.0	4.1	6.3
	8	**0.6**	2.0	1.2	3.6	5.7
ogbn-products	1	**20.3**	96.7	31.4	35.7	37.2
	2	17.9	51.1	**16.5**	22.6	28.1
	4	9.6	31.7	**8.4**	11.8	15.9
	8	6.3	18.8	**5.9**	8.6	11.2
ogbn-papers100M	1	OOM	OOM	**289.5**	306.1	351.9
	2	OOM	OOM	171.4	**168.4**	207.4
	4	92.5	OOM	**82.1**	93.5	95.1
	8	59.3	OOM	**42.4**	52.0	61.1
Friendster	1	OOM	OOM	**473.9**	686.0	721.1
	2	OOM	OOM	**324.0**	413.1	556.1
	4	OOM	OOM	**156.9**	277.6	381.3
	8	103.1	OOM	**94.5**	186.4	206.1

表2：不同集群规模下，GCN 训练单次迭代的运行时间（秒）对比。将 DGL（PyTorch）和阿里Graph与排名前三的 SparseEinSum 分解进行比较。

9.2 SparseEinSum 分解的评估

分布式图卷积神经网络（GCN）[26]。我们将 SparseEinSum 生成的张量-关系计算与运行在 PyTorch 后端上的 DGL v2.4.0 [1][39]（这是一个完全基于张量的系统）以及 AliGraph [51]进行比较。之所以选择这些系统，是因为它们是最知名的图神经网络计算系统。用于运行 SparseEinSum 生成的张量-关系计算的分布式引擎是 PlinyCompute [2] [52]，，这是一种分布式关系计算引擎。我们使用 TACO [3] 编译器生成的内核 [11, 27, 28]，以便在 SparseEinSum 生成的张量-关系计算中高效支持任意形状的张量运算。

计算问题。图卷积神经网络中的核心计算可表示为以下 EinSum：

$$\mathbf{T}^0_{i,k} \leftarrow \sum_j \hat{\mathbf{D}}_{i,j} \times \hat{\mathbf{A}}_{j,k}; \qquad \mathbf{T}^1_{i,l} \leftarrow \sum_k \mathbf{T}^0_{i,k} \times \hat{\mathbf{D}}_{k,l}$$

$$\mathbf{T}^2_{i,m} \leftarrow \sum_l \mathbf{T}^1_{i,l} \times \mathbf{H}^{(l)}_{l,m}; \qquad \mathbf{T}^3_{i,n} \leftarrow \sum_m \mathbf{T}^2_{i,m} \times \mathbf{W}^{(l)}_{m,n}$$

$$\mathbf{H}^{(l+1)}_{i,n} \leftarrow \sum_{\emptyset} \sigma(\mathbf{T}^3_{i,n})$$

这里，$\hat{}\mathbf{A}$ 是添加了自环的邻接矩阵，$\hat{}\mathbf{D}$ 是度矩阵，$\mathbf{H}^{(l)}$ 是第 l 层的节点嵌入矩阵，$\mathbf{W}^{(l)}$ 是第 l 层的权重矩阵。此表达式通过以下方式计算下一层的节点嵌入：(1) 对……进行归一化

使用度矩阵对邻接矩阵进行归一化（$\hat{}\mathbf{D}^{-\frac{1}{2}}\hat{}\mathbf{A}\hat{}\mathbf{D}^{-\frac{1}{2}}$），（2）通过该归一化邻接矩阵聚合邻居特征，以及（3）通过$\mathbf{W}^{(l)}$ 执行学习到的变换，随后应用非线性激活函数 σ。

[1] 2.4.x 分支截至提交 c6c874b[2] https://github.com/riceplinygroup/plinycompute [3] https://github.com/tensor-compiler/taco

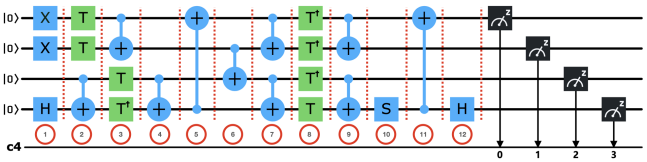


图1：量子电路从第 1 层到第 12 层划分的示例。

数据集	量子比特	深度	宽度	门计数
seca n11_	11	37	11	1275
multiplier_n13	13	7	13	496

表 3：量子基准测试的数据集统计信息。

电路	Num 机器	SparseEinSum		
		Opt	**第二佳**	**第三佳**
seca n11_	1	36.124	**26.7**	32.013
	2	**21.214**	27.512	27.752
	4	**13.817**	14.347	15.424
	8	**10.094**	12.531	12.309
乘数_n13	1	**102.632**	105.316	116.229
	2	**55.247**	64.731	71.315
	4	**36.651**	42.325	47.426
	8	**21.922**	25.127	30.98

表4：量子电路在不同机器数量下的运行时间（秒）。

我们在传导式设置下实现了一个两层图卷积网络，使用以下超参数：Adam 优化器，学习率为 $\eta = 0.1$，丢弃率为 $\gamma = 0.5$，隐藏层维度为 $D = 256$。

所有分布式实验均在配备 Intel® Xeon® Silver 4214 CPU、128GB DDR4 内存和 1TB 固态硬盘存储的服务器上进行。各节点通过 10 Gbps 以太网线互连。实验使用了由 1 至 8 台机器组成的集群。数据集的统计信息见表1。

结果与讨论。对于无法装入单台机器内存的大型图，在 DGL 和 AliGraph 上运行时，我们使用 ParMETIS [1]辅助进行分区。执行时间汇总于表2。为了评估 SparseEinSum 重写算法所用成本模型的准确性，我们执行了最佳分解（**优化**），以及排名第二（**第二**）和第三（**第三**）的分解。

对于最小的图（ogbn-arxiv），DGL（运行于 PyTorch 上）速度最快。这并不意外，因为 PlinyCompute 的设计目标是可扩展性，而非低延迟。对于更大的图（ogbn-products），两种方法的表现大致相当。对于更大的两张图（10亿+ 条边），SparseEinSum 生成的计算具有明显优势。它是在所有情况下都能运行的方法——尽管经过手动调优，DGL 和 AliGraph 仍都受到内存溢出问题的困扰。在 DGL 能够运行的情况下，其速度比 SparseEinSum 生成的计算慢 8.6% 至近 40%。

请注意，张量关系实现能够在不同数量的机器上良好扩展。SparseEinSum-on-PlinyCompute在从一台机器扩展到八台机器时，在ogbn-products、ogbn-papers100M和Friendster上分别实现了5.3倍、6.8倍和5.0倍的加速。DGL-on-Pytorch在两个较大的图上失败，但在ogbn-products上仅实现了3.2倍的加速。

方法	数据集					
	Cora	P. Cora	P.Cite Seer	C.Cite Seer	A. Photo	A. Comp
	稀疏注意力计算					
Hyper	866	901	1497	40	2416	7801
PstGSQL	8017	7990	12156	619	18429	OOD
SQLite	22052	22084	TLE	34875	TLE	OOD
SEinSum	**4.5**	**4.7**	**8.0**	**7.7**	**192**	**265**
	稠密注意力计算					
Hyper	2194	2185	3457	75	**2486**	**8123**
PstGSQL	8130	8186	12432	636	18464	OOD
SQLite	16424	15180	TLE	39470	TLE	TLE
SEinSum	**609**	**602**	**950**	**54**	OOD	OOD

表 5：注意力计算的运行时间（秒）。P 表示 Planetoid，C：CitationFull，A：亚马逊。OOD 表示磁盘空间耗尽，因为数据库写入了过多数据。

方法	数据集					
	Cora	P. Cora	P.Cite Seer	C.Cite Seer	A. Photo	A. Comp
	图卷积神经网络					
Hyper	10.7	10.8	**21.021**	4.4	197.5	390.4
PstGSQL	16.0	16.0	30.9	7.7	436.1	830.3
SQLite	TLE	TLE	TLE	TLE	TLE	TLE
SEinSum	**7.3**	**8.4**	22.9	**3.3**	**19.3**	**37.6**

表 6：图卷积神经网络的运行时间（秒）。TLE 表示超出时间限制。

单机卷积神经网络。 单机实验重点比较SparseEinSum生成的张量关系实现与等效的纯关系式实现。这些实现使用PortableSQL EinSum编译器 [5], 生成，选择该编译器是因为它是数据库文献中将EinSum转换为SQL的标准方法。SparseEinSum生成的张量关系计算在PostgreSQL之上运行，同样使用TACO生成的内核。纯关系实现在SQLite [3], PostgreSQL [15], 和Hyper [24]上运行。

结果与讨论。 运行时间如表 6 所示。对于最小的四个数据集，各选项之间没有明显差异：张量关系实现通常最快，但相较于“纯关系 + Hyper”，运行时间最多仅缩短 25%。对于较大的两个数据集，张量关系实现比“纯关系 + Hyper”快十倍，比“纯关系 + Postgres”快 40 多倍。

表 8 展示了 SparseEinSum 编译器针对两个输入图生成的模式。有趣的是，这些计算都以相当关系化的方式实现。然而，正如预期的那样，例外在于特征和嵌入（它们是稠密的）按每个顶点表示为向量，而线性变换则实现为稠密矩阵。

分布式量子电路模拟 [47]。 我们采用广泛使用的 QASMbench [29] 套件作为量子电路模拟基准。使用 Qiskit 1.2.4 [40] 和 cuda-quantum 24.03.0 [25] 来辅助量子电路生成与执行。我们

UCLC EinSum	Planetoid.CiteSeer
$\mathbf{T}^0_{I,k} \leftarrow \sum_M \mathbf{X}_{I,M} \times \mathbf{W}^Q_{M,k}$	X(整数 i, 整数 m, 双精度浮点数 val) $\mathbf{W}^Q$(整数 m, 张量[1024] val)
$\mathbf{T}^1_{J,k} \leftarrow \sum_N \mathbf{X}_{J,N} \times \mathbf{W}^K_{N,k}$	X(整数 j, 整数 n, 双精度浮点数 val) $\mathbf{W}^K$(整数 n, 张量[1024] val)
$\mathbf{T}^2_{I,J,k} \leftarrow \sum_\emptyset \mathbf{T}^0_{I,k} \times \mathbf{A}_{I,J}$	$\mathbf{T}^0$(整数 i, 张量[1024] val) A(整数 i, 整数 j, 双精度浮点数 val)
$\mathbf{T}^3_{I,J,k} \leftarrow \sum_k \mathbf{T}^2_{I,J,k} \times \mathbf{T}^1_{J,k}$ 注意力$_{I,J} \leftarrow \sum_\emptyset \frac{1}{\sqrt{d_k}}$(转置$^3_{I,J}$)	$\mathbf{T}^2$(整数 i, 整数 j, 张量[1024] val) $\mathbf{T}^1$(整数 j, 张量[1024] 值) 张量^3(整数 i, 整数 j, 双精度浮点数 val)

表 7：SparseEinSum 为 Planetoid.CiteSeer 生成的大小写爱因斯坦记号。

选择 seca_n11 和 multiplier_n13 作为大规模基准。这些基准的特征见表 3。

计算问题。量子门操作的张量收缩可以写为：

$$\psi'_{i_n} = \sum_{i_{n-1}} \mathbf{U}^{(n)}_{i_n,i_{n-1}} \sum_{i_{n-2}} \mathbf{U}^{(n-1)}_{i_{n-1},i_{n-2}} \cdots \sum_{i_1} \mathbf{U}^{(2)}_{i_2,i_1} \sum_{i_0} \mathbf{U}^{(1)}_{i_1,i_0} \psi_{i_0} \quad (8)$$

其中，$\mathbf{U}^{(k)}$ 表示第 k层的酉门操作矩阵，ψ是初始量子态向量，ψ' 是应用所有量子门操作后的最终量子态向量。

每个量子电路被划分为多个层，同一层内的多个门可以并行执行。对于每一层，我们通过对该层内所有门做张量积来生成一个酉矩阵。整个模拟过程将量子比特向量态逐层传入每一层对应的酉矩阵，如图1所示。

结果与讨论。 在分布式环境中运行大规模基准测试的结果如表4所示。我们根据 SparseEinSum 编译器所使用的成本模型，评估其生成的最佳、次佳和第三佳分解。在几乎所有情况下，成本模型都正确地对各种分解进行了排序。

扩展效率尚可：在 seca_n11 上，八台机器比一台快 3.6 倍；在 multiplier_n13 上快 4.6 倍。扩展效率不如图卷积网络的情况，因为数据移动开销较高，而计算成本相对较小。

单机注意力计算 [45, 48]. 稀疏注意力类似于经典的稠密注意力，但它允许用任意图来指示哪些项应当关注哪些项。

计算问题。 图变换器中的核心计算是稀疏注意力计算：

$$\mathbf{T}^0_{i,k} \leftarrow \sum_M \mathbf{X}_{i,m} \times \mathbf{W}^Q_{m,k}; \qquad \mathbf{T}^1_{j,k} \leftarrow \sum_n \mathbf{X}_{j,n} \times \mathbf{W}^K_{n,k}$$
$$\mathbf{T}^2_{i,j,k} \leftarrow \sum_\emptyset \mathbf{T}^0_{i,k} \times \mathbf{A}_{i,j}; \qquad \mathbf{T}^3_{i,j} \leftarrow \sum_k \mathbf{T}^2_{i,j,k} \times \mathbf{T}^1_{j,k}$$
$$\mathbf{Attn}_{i,j} \leftarrow \sum_\emptyset \frac{1}{\sqrt{d_k}} (\mathbf{T}^3_{i,j})$$

其中 $\mathbf{X}$ 存储节点特征，$\mathbf{W}^Q$ 和 $\mathbf{W}_K$ 分别是可学习的查询和键投影矩阵，$\mathbf{A}$ 是邻接矩阵

大写-小写-大写 EinSum	CitationFull.CiteSeer	Amazon.Computers
$\mathbf{T}^0_{I,K} \leftarrow \sum_J \hat{\mathbf{D}}_{I,J} \times \hat{\mathbf{A}}_{J,K}$	$\hat{\mathbf{D}}$(整数 i, 整数 j, 双精度浮点数 val) $\hat{\mathbf{A}}$(整数 j, 整数 k, 双精度浮点数 val)	$\hat{\mathbf{D}}$(整数 i, 整数 j, 双精度浮点数 val) $\hat{\mathbf{A}}$(整数 j, 整数 k, 双精度浮点数 val)
$\mathbf{T}^1_{I,L} \leftarrow \sum_K \mathbf{T}^0_{I,K} \times \hat{\mathbf{D}}_{K,L}$	$\mathbf{T}^0$(整数 i, 整数 k, 双精度浮点数 val) $\hat{\mathbf{D}}$(整数 k, 整数 l, 双精度浮点数 val)	$\mathbf{T}^0$(整数 i, 整数 k, 双精度浮点数 val) $\hat{\mathbf{D}}$(整数 k, 整数 l, 双精度浮点数 val)
$\mathbf{T}^2_{I,m} \leftarrow \sum_L \mathbf{T}^1_{I,L} \times \mathbf{H}^{(l)}_{L,m}$	$\mathbf{T}^1$(整数 i, 整数 l, 双精度浮点数 val) $\mathbf{H}^{(l)}$(整数 l, 张量[602] val)	$\mathbf{T}^1$(整数 i, 整数 l, 双精度浮点数 val) $\mathbf{H}^{(l)}$(整数 l, 张量[767] val)
$\mathbf{T}^3_{I,n} \leftarrow \sum_m \mathbf{T}^2_{I,m} \times \mathbf{W}^{(l)}_{m,n}$	$\mathbf{T}^2$(整数 i, 张量[602] val) $\mathbf{W}^{(l)}$ (张量[602][256]val)	$\mathbf{T}^2$(整数 i, 张量[767]值) $\mathbf{W}^{(l)}$ (张量[767][256]值)
$\mathbf{H}^{(l+1)}_{I,n} \leftarrow \sum_\emptyset \sigma(\mathbf{T}^3_{I,n})$	$\mathbf{T}^3$(整数 i, 张量[256] 值)	$\mathbf{T}^3$(整数 i, 张量[256] 值)

表 8：SparseEinSum 编译器为图卷积网络计算生成的大小写爱因斯坦记号，以及对应的关系模式，适用于两个测试数据集。

方法	数据集					
	Cora	P. Cora	P.Cite Seer	C.Cite Seer	A. Photo	A. Comp
	稀疏注意力计算					
SEinSum	4.5	4.7	8.0	7.7	192	265
贪心	21	23	52	17	769	884
	稠密注意力计算					
SEinSum	609	602	950	54	OOD	OOD
贪心	2198	2124	4107	315	OOD	OOD

表 9：比较纯贪心运行时间与用于注意力计算的完整动态规划方案。

表示图结构，d_k 是键向量的维度，而 $\mathbf{Attn}_{i,j}$ 表示归一化前的注意力分数。遵循先前的工作 [45, 48]，，我们评估了稠密注意力（每个节点关注所有其他节点）和稀疏注意力（节点只关注图中的局部邻域）。

结果与讨论。运行时间结果见表 5。这里令人惊讶的发现是，尽管关系计算本质上是稀疏的，但与稠密注意力相比，使用稀疏注意力时，纯关系实现并没有快多少。“纯关系 + Hyper”最能利用稀疏性，在稀疏情况下速度最高可达稠密情况下的 2 倍。相比之下，计算稀疏注意力时，SparseEinSum 的速度可比计算稠密注意力快 100 倍以上；在稀疏情况下，其速度是“纯关系 + Hyper”的 30 至 100 倍。

表 7 展示了其中一个数据集生成的模式。节点嵌入以向量形式存储，但由于内存限制，权重矩阵存储在向量中。输入数据本身是稀疏的，并已分解为纯关系元组。

9.3 消融实验和其他实验

动态规划的有效性。可以合理地提出这样的问题：在 SparseEinSum 分解过程中，是否有必要运行代价高昂且复杂的动态规划搜索？我们评估了一种更简单的贪心搜索：沿着 EinSum 表达式图中的一条路径，贪心地选择使代价最小的分解；不记忆具有不同输出张量格式且并非最优的分解。我们在表 9 中，以单台机器上的注意力计算为例，将贪心搜索与 SparseEinSum 进行了比较。

Comp-onent	数据集					
	Cora	P. Cora	P.Cite Seer	C.Cite Seer	A. Photo	A. Comp
	稀疏注意力计算					
执行	4.5	4.7	8.0	7.7	192	265
模式	253	96	428	86	344	239
SQL	46	11	32	2.0	5.4	3.4
内核	0.04	0.04	0.04	0.08	0.04	0.04
	稠密注意力计算					
Exec	609	602	950	54	OOD	OOD
模式	146	127	533	221	862	1690
SQL	51	24	58	44	130	401
内核	0.06	0.06	0.06	0.06	0.06	0.06

表 10：各系统组件、注意力计算的运行时间，单位为秒。“执行”是实际执行时间，“模式”是预处理数据并生成模式所需的时间，“SQL”是生成 SQL（包括数据加载代码）所需的时间，“内核”是使用 TACO 生成内核所需的时间。

	扰动					
None（原样）	C_{xfer} $\times\frac{1}{10}$	C_{xfer} $\times 10$	C_{krnel} $\times\frac{1}{10}$	C_{krnel} $\times 10$	C_{fixed} $\times\frac{1}{10}$	C_{fixed} $\times 10$
192	130	253	294	1027	1027	305

表 11：扰动各个成本参数（C_{xfer}，C_{krnel}，C_{fixed}）对亚马逊 Amazon.Photo 数据集上稀疏注意力计算运行时间的影响。

对成本模型误差的稳健性。准确的成本模型是否必要？我们通过将每个成本估计值乘以一个 Gamma（α，θ）随机变量来扰动成本估计值，其中 $\alpha \times \theta = 1$，从而使成本值的期望保持不变，但允许存在误差。随着我们减小 α 并增大 θ，估计值的方差（以及出错的可能性）会增加。表 12 给出了单机图神经网络计算的结果。运行时间取十次不同扰动运行的平均值。这里可以看到，当 $\alpha = \theta = 1$ 时，运行时间没有差异——鉴于这是一个相当显著的噪声水平，这表明系统具有一定的稳健性。更高的扰动水平会导致运行速度变慢。有趣的是，$\alpha = \frac{1}{3}$，$\theta = 3$ 的表现比 $\alpha = \frac{1}{10}$，$\theta = 10$更差。

对成本参数的鲁棒性。我们固定三个成本参数（C_{xfer}、C_{krnel}、C_{fixed}）中的两个，并对第三个进行调整，

错误	数据集					
	Cora	P. Cora	P. Cite Seer	C.Cite Seer	A. Photo	A. Comp
	图卷积神经网络					
None	7.3	8.4	22.9	3.3	19.3	37.6
$\Gamma(1,\ 1)$	7.2	8.3	21.5	3.4	19.5	37.1
$\Gamma(\frac{1}{3},\ 3)$	16.1	16.0	31.9	8.0	435.9	839.9
$\Gamma(1\text{-}10,\ 10)$	12.0	11.2	25.1	7.4	68.0	47.4

表 12：基数估计错误或含噪时，图神经网络的运行时间（单位：秒）。此处，$\Gamma(\alpha,\ \theta)$表示每个估计值都乘以一个使用 Gamma（$\alpha,\ \theta$）分布生成的随机数。

要么将其缩小十倍，要么将其扩大十倍。单机结果见表 12。成本参数选择不当可能造成相当大的影响。有趣的是，将 C_{xfer} 缩小十倍实际上缩短了所得分解的运行时间，这表明我们选择的 C_{xfer} 并非最优。

运行时间分解。 表 10 比较了实际注意力计算的执行时间与分解本身所需的时间。除执行时间外，还给出了另外三个时间：（1）模式：处理输入数据、计算必要统计信息以及运行动态规划所需的时间。（2）SQL：生成 SQL 所需的时间，其中主要是生成用于将所有数据加载到 PostgreSQL 中的 INSERT INTO语句所需的时间。（3）内核：生成 TACO内核所需的时间。我们可以看到，这些时间可能相当可观，尽管我们的代码使用 Python 编写，并且对于单机注意力实验，所有数据都存储在文本文件中。此外，为确保跨引擎的可移植性，所有数据加载均通过 SQL 处理，而这十分冗长。在部署或训练期间，注意力计算可能会运行数百次甚至更多，因此时间开销将在多次运行之间摊销。

10 相关工作

EinSum 由爱因斯坦提出 [16], ，如今已受到 NumPy [20]、JAX [18], 、PyTorch [30] 和 Dask [13]等机器学习/人工智能框架的支持。然而，大写-小写-大写 EinSum（通过修改 EinSum 来便于指定分片）则独具特色。这一领域的一个普遍问题是如何指定分片——例如，https://jax-ml.github.io/scaling-book/sharding/ 展示了这方面的一项尝试；但大写-小写-大写 EinSum 的独特之处在于，它能够将计算规范与计算分解方式结合起来。

我们针对将 EinSum 表达式的有向无环图（任意相连顶点之间的路径均唯一）重写为大写-小写-大写 EinSum 的有向无环图这一问题，提出了一种动态规划算法。这类动态规划算法早已存在；我们的算法与计算生物学中的费尔森斯坦算法 [17] 颇为相似，而其他针对机器学习计算图分片的研究也采用了动态规划 [8, 49]。同样，也已有用于稀疏张量计算的成本模型 [2, 35]。

目前已有多种用于张量计算的内核编译器，例如 TVM [10], Triton [38], 和 TACO [11, 27, 28]。TACO 和 TVM（通过 SparseTIR [44]）都明确支持稀疏性。在各种支持稀疏性的张量编译器中，Galley [14] 与我们提出的方法最为接近，借鉴了许多关系系统设计理念（例如，应用关系方法进行稀疏度估计；参见本文第 7 节）。这项已有工作旨在生成 CPU 或 GPU 内核或程序，而非可扩展的关系计算；后者旨在运行于关系系统之上。SparseEinSum 重写方法的核心思想是确定计算的哪些部分应以关系方式运行，哪些部分应利用由其他系统生成的高性能 CPU 或 GPU 内核。从这个意义上说，SparseEinSum 位于这些工具之上，并使用它们生成的内核。事实上，我们的 SparseEinSum 原型使用 TACO 来生成稠密内核。

近期还有其他研究探索张量与关系之间的密切联系，以优化张量计算。Schleich等人提出了 STOREL [31],，该系统接受用户为每个输入张量指定的存储格式，然后对张量计算进行优化。相比之下，SparseEinSum 接受以声明方式指定的计算，然后优化表示格式，选择如何将张量分解为关系，以优化性能。SparseEinSum 方法依赖底层关系系统来优化计算。

在 Duck's Brain [32], 中，Schüle等人是最近主张将由张量增强的关系系统用于现代机器学习计算的人——使用的正是本文所描述的这类张量-关系表示。鸭脑是一个 关系平台 示例，在该平台上可以运行由 SparseEinSum 生成的张量-关系计算。张量与关系相结合的处理方式也在其他工作中得到探索 [22, 23, 37, 46], , 包括 SystemML [19] 和 SystemDS [6], ；后者使用矩阵原语扩展了 SQL风格操作。然而，这些先前的工作并未考虑如何自动生成经过优化的张量-关系表示。与 Duck's Brain 的情况类似，SparseEinSum 可以用于生成这些系统所采用的分解。

文献中探索的另一种方法是使用纯关系实现来执行机器学习或人工智能计算（例如，参见近期的两项工作 Pytond [33] 和 Portable SQL [5] ）。另一项近期工作提出了一种算法，可在运行时动态地从稠密表示切换为稀疏表示 [36]。

11 结论

我们介绍了一种算法，可将张量计算分解为等价的张量-关系计算；此外还介绍了 SparseEinSum，这是一个以 EinSum 计算为输入，并将其编译为大小写爱因斯坦记号的系统。该计算几乎可以在任何支持多维数组的数据库系统上实现。编译过程融入了稀疏度估计技术和基于成本的自动模式优化，使生成的张量-关系计算能够显著提升性能。

致谢

本研究获得了美国国家科学基金会（NSF）项目资助，项目编号为 1918651、2008240、2131294 和 2212557；美国国立卫生研究院（NIH）临床与转化科学奖（CTSA）# UM1TR004906；以及美国交通部（US DOT）一级大学交通中心 CYBER-CARE 项目资助，编号为 #69A3552348332。

参考文献

[1] [n.d.]. ParMETIS。https://github.com/KarypisLab/ParMETIS。[2] Willow Ahrens、Fredrik Kjolstad 和 Saman Amarasinghe。2022。采用渐近成本模型的稀疏张量代数自动调度。载于《第43届 ACM SIGPLAN 编程语言设计与实现国际会议论文集》。269–285。[3] Grant Allen、Mike Owens、Grant Allen 和 Mike Owens。2010。SQLite简介。《SQLite权威指南》（2010），1–16。[4] Alan H Barr。1991。爱因斯坦求和记号。《基于物理的建模导论》（课程讲义19），第E 1页（1991），57。[5] Mark Blacher、Julien Klaus、Christoph Staudt、Sören Laue、Viktor Leis 和 Joachim Giesen。2023。SQL中的高效且可移植的爱因斯坦求和。《ACM数据管理会议录》1，2，文章121（2023年6月），19页。https://doi.org/10.1145/ 3589266[6] Matthias Boehm、Iulian Antonov、Sebastian Baunsgaard、Mark Dokter、Robert Ginthör、Kevin Innerebner、Florijan Klezin、Stefanie Lindstaedt、Arnab Phani、Benjamin Rath 等。2019。SystemDS：面向端到端数据科学生命周期的声明式机器学习系统。arXiv预印本 arXiv:1909.02976（2019）。[7] Aleksandar Bojchevski 和 Stephan Günnemann。2017。图的深度高斯嵌入：通过排序实现无监督归纳学习。arXiv预印本 arXiv:1707.03815（2017）。[8] Daniel Bourgeois、Zhimin Ding、Dimitrije Jankov、Jiehui Li、Mahmoud Sleem、Yuxin Tang、Jiawen Yao、Xinyu Yao 和 Chris Jermaine。2024。EinDecomp：对声明式指定的机器学习与数值计算进行分解，以实现并行执行。arXiv预印本 arXiv:2410.02682（2024）。[9] Danah Michele Boyd。2004。Friendster与公开呈现的社交网络。载于《CHI'04计算机系统中的人为因素扩展摘要》。1279–1282。[10] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze 等。2018。{TVM}：面向深度学习的端到端自动优化编译器。载于《第13届 USENIX 操作系统设计与实现研讨会（OSDI 18）》。578–594。[11] Stephen Chou、Fredrik Kjolstad 和 Saman Amarasinghe。2018。稀疏张量代数编译器的格式抽象。《ACM编程语言会议录》2，OOPSLA，文章 123 2018 30 10 1145 3276493（10月），页码。https://doi.org/ 。[12] 2018 / G Daniel、Johnnie Gray 等。Opt\ einsum——一个用于优化 einsum类表达式收缩顺序的 Python 软件包。《开源软件杂志》3，26（2018），753。[13] Jesse Daniel。2019。使用 Python 和 Dask 进行数据科学。Simon and Schuster。[14] Kyle Deeds、Willow Ahrens、Magdalena Balazinska 和 Dan Suciu。2025。Galley：面向稀疏张量程序的现代查询优化。《 ACM 数据管理会议录》3，3（2025），1–24。[15] Korry Douglas 和 Susan Douglas。2003。PostgreSQL：构建、编程和管理 PostgreSQL 数据库的综合指南。SAMS出版社。[16] Albert Einstein、Leopold Infeld 和 Banesh Hoffmann。1938。引力方程与运动问题。《数学年刊》（1938），65–100。[17] Joseph Felsenstein。1981。由 DNA序列推导进化树：最大似然方法。《分子进化杂志》17，6（1981），368–376。[18] Roy Frostig、Matthew James Johnson 和 Chris Leary。2018。通过高级跟踪编译机器学习程序。《机器学习系统》4，9（2018）。[19] Amol Ghoting、Rajasekar Krishnamurthy、Edwin Pednault、Berthold Reinwald、Vikas Sindhwani、Shirish Tatikonda、Yuanyuan Tian 和 Shivakumar Vaithyanathan。2011。SystemML：基于 MapReduce 的声明式机器学习。载于《2011年 IEEE 第27届数据工程国际会议》。IEEE，231–242。[20] Charles R Harris、K Jarrod Millman、Stéfan J Van Der Walt、Ralf Gommers、Pauli Virtanen、David Cournapeau、Eric Wieser、Julian Taylor、Sebastian Berg、Nathaniel J Smith 等。2020 585，7825 2020 357 362。使用 NumPy 进行数组编程。《自然》（），–。[21] Weihua Hu、Matthias Fey、Marinka Zitnik、Yuxiao Dong、Hongyu Ren、Bowen Liu、Michele Catasta 和 Jure Leskovec。2020。开放图基准：用于图上机器学习的数据集。《神经信息处理系统进展》33（2020），22118–22133。[22] Dimitrije Jankov、Shangyu Luo、Binhang Yuan、Zhuhua Cai、Jia Zou、Chris Jermaine 和 Zekai J Gao。2019。在关系数据库管理系统上进行声明式递归计算，或者说，为什么分布式机器学习应该使用数据库。arXiv预印本 arXiv:1904.11121（2019）。[23] Dimitrije Jankov、Binhang Yuan、Shangyu Luo 和 Chris Jermaine。2021。通过聚合连接树的两阶段执行进行分布式数值计算与机器学习计算。《VLDB基金会会刊》14，7（2021）。[24] Alfons Kemper 和 Thomas Neumann。2011。HyPer：一种基于虚拟内存快照的混合型联机事务处理与联机分析处理主存数据库系统。载于《2011年 IEEE 第27届数据工程国际会议》。IEEE，195–206。[25] Jin-Sung Kim、Alex McCaskey、Bettina Heim、Manish Modani、Sam Stanwyck 和 Timothy Costa。2023。Cuda quantum：集成量子经典计算平台。载于《2023年第60届 ACM/IEEE 设计自动化会议（DAC）》。IEEE，1–4。[26] Thomas N Kipf 和 Max Welling。2016。使用图卷积网络进行半监督分类。arXiv 预印本 arXiv:1609.02907（2016）。[27] Fredrik Kjolstad、Willow Ahrens、Shoaib Kamil 和 Saman Amarasinghe。2019。使用工作区进行张量代数编译。载于2019年IEEE/ACM国际代码生成与优化研讨会（CGO）。IEEE，第180–192页。[28] Fredrik Kjolstad、Shoaib Kamil、Stephen Chou、David Lugato 和 Saman Amarasinghe。2017。张量代数编译器。Proc. ACM Program. Lang. 1，OOPSLA，文章 77 2017 29 10 1145 3133901（10月），页码。https://doi.org/ 。[29] / Ang Li、Samuel Stein、Sriram Krishnamoorthy 和 James Ang。2023。Qasmbench：用于NISQ评估与模拟的低层级量子基准测试套件。ACM量子计算汇刊 4，2（2023），1–26。[30] A Paszke。2019。PyTorch：一种命令式风格的高性能深度学习库。arXiv 预印本 arXiv:1912.01703（2019）。[31] Maximilian Schleich、Amir Shaikhha 和 Dan Suciu。2023。在灵活存储上优化张量程序。ACM数据管理会议录 1，1 2023 1 27( )，–。[32] Maximilian Schüle、Thomas Neumann 和 Alfons Kemper。2024。鸭脑：在数据库引擎中训练和推理神经网络。Datenbank-Spektrum 24，3（2024），209–221。[33] Hesam Shahrokhi、Amirali Kaboli、Mahdi Ghorbani 和 Amir Shaikhha。2024。Pytond：依托数据库实现高效的Python数据科学。载于2024年IEEE第40届国际数据工程会议（ICDE）。IEEE，第423–435页。[34] Oleksandr Shchur、Maximilian Mumme、Aleksandar Bojchevski 和 Stephan Günnemann。2018。图神经网络评估的陷阱。arXiv 预印本 arXiv:1811.05868（2018）。[35] Johanna Sommer、Matthias Boehm、Alexandre V Evfimievski、Berthold Reinwald 和 Peter J Haas。2019。MNC：利用结构特征估计矩阵表达式的稀疏性。载于2019年国际数据管理会议论文集。1607–1623。[36] Christoph Staudt、Mark Blacher、Tim Hoffmann、Kaspar Kasche、Olaf Beyersdorff 和 Joachim Giesen。2025。在EinSum中利用动态稀疏性。载于第三十九届神经信息处理系统年会。[37] Yuxin Tang、Zhimin Ding、Dimitrije Jankov、Binhang Yuan、Daniel Bourgeois 和 Chris Jermaine。2023。面向超大规模机器学习的关系计算自动微分。载于国际机器学习会议。PMLR，第33581–33598页。[38] Philippe Tillet、Hsiang-Tsung Kung 和 David Cox。2019。Triton：用于分块神经网络计算的中间语言和编译器。载于第三届ACM SIGPLAN机器学习与编程语言国际研讨会论文集。10–19。[39] Minjie Wang、Da Zheng、Zihao Ye、Quan Gan、Mufei Li、Xiang Song、Jinjing Zhou、Chao Ma、Lingfan Yu、Yu Gai、Tianjun Xiao、Tong He、George Karypis、Jinyang Li 和 Zheng Zhang。2020。深度图学习库：一个以图为中心、高性能的图神经网络软件包。arXiv:1909。01315 [cs.LG] https://arxiv.org/abs/ 1909 01315 。[40] Robert Wille、Rod Van Meter 和 Yehuda Naveh。2019。IBM的Qiskit工具链：使用和开发面向真实量子计算机的应用。载于2019年欧洲设计、自动化与测试会议暨展览会（DATE）。IEEE，第1234–1240页。[41] Zonghan Wu、Shirui Pan、Fengwen Chen、Guodong Long、Chengqi Zhang 和 Philip S Yu。2020。图神经网络综合综述。IEEE神经网络与学习系统汇刊 32，1（2020），4–24。[42] Lizhi Xiang、Omid Asudeh、Gerald Sabin、Aravind Sukumaran-Rajam 和 P Sadayappan。2025。cuTeSpMM：利用GPU张量核心加速稀疏-稠密矩阵乘法。arXiv 预印本 arXiv:2504.06443（2025）。[43] Zhilin Yang、William Cohen 和 Ruslan Salakhudinov。2016。重新审视基于图嵌入的半监督学习。载于国际机器学习会议。PMLR，第40–48页。[44] Zihao Ye、Ruihang Lai、Junru Shao、Tianqi Chen 和 Luis Ceze。2023。SparseTIR：深度学习中稀疏编译的可组合抽象。载于第28届ACM编程语言与操作系统架构支持国际会议论文集，第3卷。660–678。[45] Chengxuan Ying、Tianle Cai、Shengjie Luo、Shuxin Zheng、Guolin Ke、Di He、Yanming Shen 和 Tie-Yan Liu。2021。Transformer在图表示任务上的表现真的很差吗？神经信息处理系统进展 34（2021），28877–28888。[46] Binhang Yuan、Dimitrije Jankov、Jia Zou、Yuxin Tang、Daniel Bourgeois 和 Chris Jermaine。2021。面向分布式机器学习系统设计的张量关系代数。Proc. VLDB Endow. 14，8（2021年4月），1338–1350。https://doi.org/ 10.14778/3457390.3457399[47] Chen Zhang、Zeyu Song、Haojie Wang、Kaiyuan Rong 和 Jidong Zhai。2021。HyQuas：基于GPU的量子电路模拟混合分区器系统。载于第35届ACM超级计算国际会议论文集。443–454。[48] Meng Zhang、Jie Sun、Qinghao Hu、Peng Sun、Zeke Wang、Yonggang Wen 和 Tianwei Zhang。2024。TorchGT：用于大规模图Transformer训练的整体系统。载于SC24：高性能计算、网络、存储与分析国际会议。IEEE，第1–17页。

[49] Lianmin Zheng、Zhuohan Li、Hao Zhang、Yonghao Zhuang、Zhifeng Chen、Yanping Huang、Yida Wang、Yuanzhong Xu、Danyang Zhuo、Eric P Xing等。2022。Alpa：自动化分布式深度学习中的跨{算子内} 并行化。载于第*16*届*USENIX*操作系统设计与实现研讨会（*OSDI 22*）。559–578。[50] Jie Zhou、Ganqu Cui、Shengding Hu、Zhengyan Zhang、Cheng Yang、Zhiyuan Liu、Lifeng Wang、Changcheng Li 和 Maosong Sun。2020。图神经网络：方法与应用综述。*AI Open* 1（2020），57–81。

[51] Rong Zhu、Kun Zhao、Hongxia Yang、Wei Lin、Chang Zhou、Baole Ai、Yong Li 和 Jingren Zhou。2019。AliGraph：全面的图神经网络平台。arXiv: 1902 08730 [ ] 1902 08730 。cs.DC

[52] https://arxiv.org/abs/。Jia Zou、R Matthew Barnett、Tania Lorido-Botran、Shangyu Luo、Carlos Mon- roy、Sourav Sikdar、Kia Teymourian、Binhang Yuan 和 Chris Jermaine。2018。PlinyCompute：用于高性能、分布式、数据密集型工具开发的平台。载于 *2018*年国际数据管理会议论文集。1189–1204。