arXiv:2601.12220v1 [cs.MS] 18 Jan 2026

用于调优检索的批量爱因斯坦求和规范化 val

KAUSHIK KULKARNI，伊利诺伊大学厄巴纳-香槟分校 Siebel计算与数据科学学院，美国

ANDREAS KLÖCKNER，伊利诺伊大学厄巴纳-香槟分校 Siebel计算与数据科学学院，美国

我们提出一种算法，通过将数学上等价的表达式映射为唯一范式，对批量爱因斯坦求和表达式进行规范化。具有相同爱因斯坦记号且存在大量数据重用的爱因斯坦求和批次，常见于有限元方法（FEM）、数值线性代数和计算化学中。为有效利用这种时间局部性以实现高性能，我们研究以批量形式组织的爱因斯坦求和组。

等价的批量 einsum 表示可能因索引重命名、批次内的排列，以及乘法运算的交换律和结合律而有所不同。缺乏规范表示会妨碍软件系统重用优化和调优知识。为此，我们提出一种将批量 einsum 编码为彩色图的新方法，并应用图规范化来推导范式。

除规范化算法外，我们还提出一种使用函数式数组操作数表示爱因斯坦求和的方法，并提供一种策略，将针对范式的变换迁移到具有相同范式的函数式批量爱因斯坦求和 上；这对于融合受内存带宽限制的爱因斯坦求和周边计算至关重要。我们将该方法与 JAX 进行评估，发现对于 TCCG 基准测试套件中的爱因斯坦求和以及一个有限元求解器，几何平均加速比为 4.7× 。

CCS 概念： • **应用计算** → 物理学； • **计算数学** → **数学软件性能**； •**计算方法** →数学对象的表示。

附加关键词和短语：张量收缩、优化编译器、图同构、领域专用语言

ACM 引用格式：
Kaushik Kulkarni 和 Andreas Klöckner。2026。用于调优检索的批量爱因斯坦求和规范化。1,1（2026 年 1 月），37 页。
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

线性代数子程序在性能工程师和计算科学家之间起到抽象层的作用。性能工程师负责提供针对特定硬件架构优化的内核，而计算科学家则通过组合这些内核来实现应用。此类抽象的知名示例包括 BLAS [36]（用于稠密线性算子应用中的基本运算）、LAPACK [6]（用于求解线性

作者联系方式：Kaushik Kulkarni，kgk2@illinois.edu，Siebel计算与数据科学学院，伊利诺伊大学厄巴纳-香槟分校，厄巴纳，伊利诺伊州，美国；Andreas Klöckner，andreask@illinois.edu，Siebel计算与数据科学学院，伊利诺伊大学厄巴纳-香槟分校，厄巴纳，伊利诺伊州，美国。

允许为个人或课堂使用而免费制作本作品的全部或部分数字或纸质副本，但须满足以下条件：不得制作或分发用于营利或商业利益的副本，且副本首页须附有本声明及完整引文。对于本作品中由作者以外的其他人拥有版权的组成部分，必须尊重其版权。注明出处后进行摘要使用是允许的。若要以其他方式复制、再版、发布到服务器或重新分发至列表，则须事先获得明确许可和/或支付费用。请通过 permissions@acm.org 申请许可。© 2026 版权所有归所有者/作者所有。出版权授权给 ACM。已提交至 ACM 的手稿

系统）、PETSc [9]（用于迭代线性／非线性求解器）等。借助此类抽象，计算科学家能够专注于构建应用，同时利用这些系统实现所提供的优化例程，在底层硬件系统上实现高性能执行。

在许多科学领域中都能观察到的一项关键操作是多重线性代数算子应用。这类算子将内积和外积推广到多维数组，并广泛应用于各种场景。例如，形式为 $\sum_{xj} J[x,r,e]D[x,i,j]u[e,j]$ 的表达式会出现在有限元方法（FEM）中的无矩阵组装中 [24],形式为 $\sum_{cd} g[i,j,c,d]t[c,d,a,b]$ 的表达式会出现在计算化学的簇算子中 [48],形式为 $\sum_{j_1 j_2} G_1[i_1,j_1]G_2[i_2,j_2]X[j_1,j_2]$ 的表达式会出现在神经网络的张量列车层中 [41], 等。Åhlander 等人 [4] 提出了一种领域专用语言（DSL），采用爱因斯坦求和（“einsum”）记号作为抽象，为表达这类多维缩并提供简洁的语法。此后，许多支持数组运算的系统都采用了这一记号，包括 NumPy [17],JAX [16], Cyclops [47],和 Deinsum [58]。

在大规模应用中，爱因斯坦求和很少孤立出现；相反，它们通常以组为单位进行计算，并反复访问相同的数据。例如，在有限元方法应用中，每次导数计算都对应于将同一个 einsum表达式应用于不同的场向量，而某些操作数（如几何因子）则保持不变。为了表示这种数据重用，我们认为，引入一种计算原语至关重要：它对应于一组共享相同爱因斯坦记号、但应用于不同数组操作数的爱因斯坦求和（无序集合）。我们将这一概念称为“批量 einsum”。

批量 einsum 在科学计算工作负载中无处不在，因此必须针对可用的目标硬件优化其执行速度。利用针对爱因斯坦求和的最先进编译技术来降低批量 einsum，往往会面临两个主要缺点。首先，这些方法通常会先获取一条最优收缩路径，再依靠张量收缩代码生成器 [20, 23, 38, 50]将爱因斯坦求和降低为程序。这会因分配和初始化中间数组而引入开销，即所谓的物化开销。其次，这些方法不会融合多次爱因斯坦求和调用中的数据访问，导致从DRAM到寄存器的读取产生额外开销。因此，我们采取不同的思路，设计了一种软件系统，用于维护高性能批量 einsum 的目录，以便在实际程序中重复使用。

我们将为计算原语的某个实例提供定制实现的软件系统称为采用基于表格的方法。由于批量爱因斯坦求和的表示不具备规范性，将其编码在此类表格中以供检索是一项关键挑战。这可能导致效率低下，因为多个表项可能对应数学上等价的计算。我们在图1中对此进行了说明：经过适当的重命名后，两个语法形式不同的批量爱因斯坦求和可以共用相同的计算实现。在第5节中，我们通过考察索引重命名、批次内置换以及爱因斯坦求和中操作数置换可能产生的重复项，对这些批量爱因斯坦求和的等价类进行了形式化。此外，我们还开发了一种规范化算法，可将批量爱因斯坦求和表达式映射为规范形式。该算法确保等价的批量爱因斯坦求和表达式都映射为唯一表示。

采用基于制表的方法来提供优化子程序（如 BLAS）的系统，其一个关键缺点是，计算原语的操作数必须先完成物化。这些物化开销会限制应用的计算吞吐量，尤其是在该原语本身受内存带宽限制时。算法 1对此进行了说明：矩阵-向量乘积的低算术强度使得融合通常更为有利

$$R_1[i,j,k,l] \leftarrow \sum_{m,n} A[m,n,j]B[m,n,i]C[k,l,i]$$
$$R_2[i,j,k,l] \leftarrow \sum_{m,n} D[m,n,j]E[m,n,i]A[k,l,i]$$
$$R_3[i,j,k,l] \leftarrow \sum_{m,n} B[m,n,j]D[m,n,i]F[k,l,i]$$

(a) 批量 einsum e_1。

$$R'_1[n',m',k',j'] \leftarrow \sum_{i',l'} A'[l',i',n']D'[k',j',n']E'[l',i',m']$$
$$R'_2[n',m',k',j'] \leftarrow \sum_{i',l'} E'[l',i',n']B'[k',j',n']C'[l',i',m']$$
$$R'_3[n',m',k',j'] \leftarrow \sum_{i',l'} C'[l',i',n']F\cdot[k',j',n']D'[l',i',m']$$

(b) 批量 einsum e_2。

图 1. 尽管 e_1 和 e_2 看起来是不同的一组表达式，但只要对 e_1 应用以下替换：$R_2 \to R'_1$、$R_3 \to R'_2$、$R_1 \to R'_3$、$i \to n'$、$j \to m'$、$k \to k'$、$l \to j'$、$m \to l'$、$n \to i'$、$A \to D'$、$B \to C'$ $C \to F'$ $D \to E'$ $E \to A'$　　$F \to B'$　　、、、、以及，就可以利用其实现来计算 e_2。

计算 q_1 与 BLAS 的 通用矩阵-向量乘法[1] 调用。此类未物化的数组表达式在许多科学应用中会作为爱因斯坦求和的操作数出现。例如，在计算一维库仑相互作用矩阵时，就会将距离矩阵的逐元素倒数作为操作数传入。依赖优化子程序库的高级编译器中也会出现此类物化开销，因为为了调用这些子程序，它们通常需要积极地物化中间值。在 JAX 和 PyTorch [42], 等惰性数组框架中可以观察到这一点；这些框架依赖 cuBLAS提供的稠密线性代数内核。在某些情况下，这会导致生成的二进制文件效率低下，因为内核边界会阻碍融合。

我们通过两个关键思想来应对这些由物化引起的性能损失。首先，计算科学家以批量 einsum表达式的函数式形式进行交互，其中 einsum 的操作数被指定为 λ- 表达式。算法2 展示了计算科学家如何将计算表示为函数式批量爱因斯坦求和。其次，我们维护一个数据库，记录优化函数式批量爱因斯坦求和所需的代码变换，其中每个 λ 操作数对应一个 C 风格多维数组。这种设计使计算科学家能够复用针对理想化批量爱因斯坦求和开发的优化，从而变换在结构上类似于批量 einsum 的程序。随着 NVIDIA H100、NVIDIA V100 和 AMD MI250X 等现代加速器展现出极高的饱和算术强度，这种代码优化迁移正变得愈发重要。因此，对于内存受限的工作负载，在处理单元上已驻留的数据上执行额外的浮点运算，几乎不会影响执行时间。

算法 1 计算 $\nabla\cdot$(正弦函数(q)) 的伪代码
使用有限元方法。

```
过程 FEMOperator(D, q)
    对于 i ∈ {0, ..., N} 执行
        q1[i] ← 正弦函数(q[i])
    y ← BLAS 矩阵向量乘法(D, q1)
    返回 y
```

算法 2函数式批量爱因斯坦求和表示
算法中的有限元方法算子1。

```
过程 FEMOperator(D, q)
    λA := λi j. D[i, j]
    λx := λj. 正弦函数(q[j])
    y ← 爱因斯坦求和(ij,  j → i,  λA,  λx)
    返回 y
```

我们在一个名为[1]GEMV 对应于稠密矩阵向量乘法操作的开源Python 库中，实现了范式计算、将函数式批量 einsum 与理想化批量 einsum 进行匹配，以及针对优化变体的原型数据库检索。

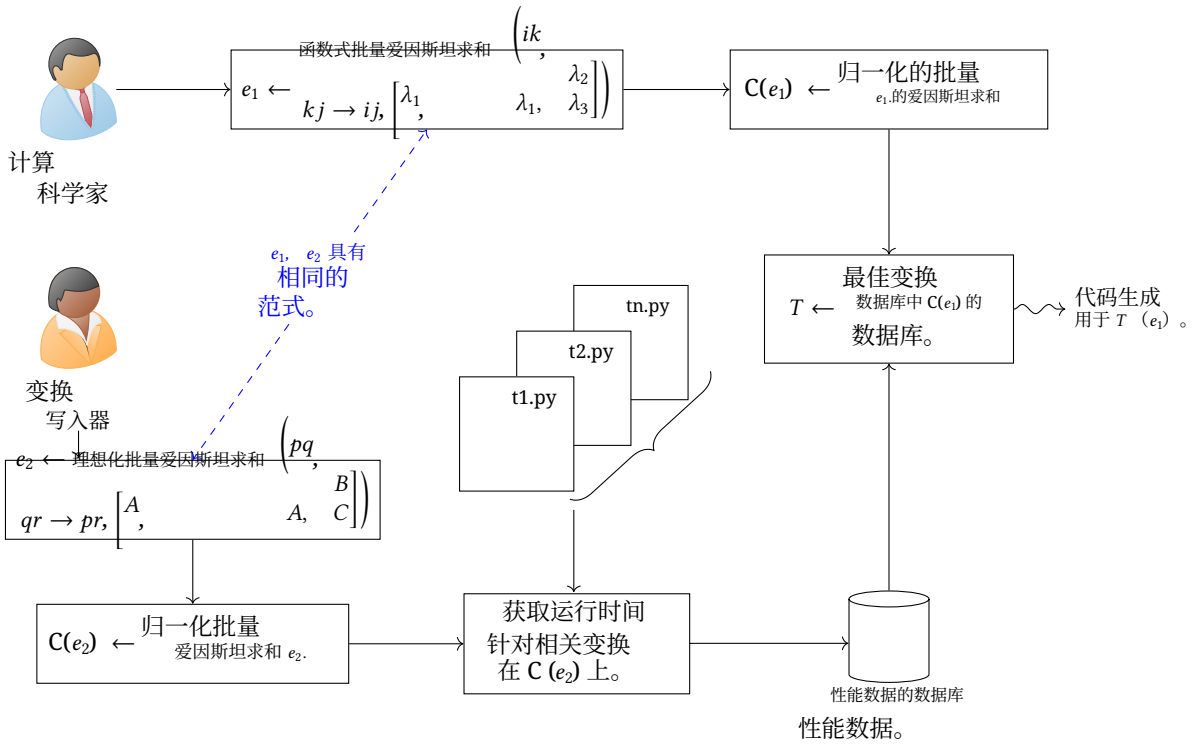


图2。Feinsum 实现关注点分离的方法。计算科学家将其工作负载表示为函数式批量爱因斯坦求和，并从数据库中获取优化方案。变换编写者为理想化批量爱因斯坦求和实现变换，并将其性能记录在数据库中。范式对于将性能工程师的变换知识复用于高级程序员至关重要。

Feinsum[2]。在 Feinsum 中，我们引入了一个 SQL 数据库，将规范批量 einsum 映射到目标硬件上的高性能代码变换。该数据库使计算科学家能够直接将学习到的变换知识融入其程序。此外，它还为变换写入器评估先进的自动调优方法提供了有价值的数据点。在图 2 中，我们可以看到，范式计算是 Feinsum 的一项必要组成部分。

在这项工作中，我们提出以下贡献：

- 我们定义了一种计算原语，用于表示爱因斯坦求和操作的无序集合，并将其称为“批量 einsum”。执行批量 einsum 的子程序能够对各个 einsum 之间的操作数复用进行建模，从而使性能工程师能够开发利用这种复用的定制实现。

- 我们提出了一种批量 einsum 表达式的规范化算法。该算法解决了爱因斯坦记号中由乘法交换律和索引重命名的可能性所导致的固有歧义。
- 一种称为 函数式批量 *einsum* 的批量 einsum 扩展，它将 λ-表达式视为操作数。这有助于将针对理想化批量 einsum 实例已知的代码优化迁移到实际程序中，对于算术强度较高的设备尤为重要。

[2] https://pypi.org/project/feinsum
已提交至 ACM 的手稿

- 一个开源软件系统，用于将函数式批量 einsum 与其理想形式进行匹配，并将批量 einsum 映射为规范形式，以创建和管理数据库。该数据库将部分批量 einsum 实例的规范形式映射到相应的近最优代码变换。

2 预备知识

2.1 多维数组

我们将采用以下多维数组定义。

定义 1. 多维数组是一种具有以下特征的数据类型：

维数 数组的维数。如果 a 是一个多维数组，那么 Dim(a) 的值为一个非负整数。**形状** 数组的形状。如果 a 是一个满足 Dim(a) $= N$的多维数组，那么 Shape(a) 的值为由N 个非负整数组成的序列。

数据类型 数组元素的数据类型。如果 a 是一个数组，那么 数据类型(a) 的计算结果为一种数值数据类型。关于完整的数值数据类型集合，请参阅NumPy [17] 。

它还支持索引访问 操作：如果 a是一个形状为（d_1，d_2，.....，d_n）的多维数组，并且（i_1，i_2，.....，i_n）是一个由非负整数组成的序列，使得对于所有 $k \in \{1, \ldots n\}$都有 $i_k < d_k$，那么 IndexInto(a，（i_1，i_2，.....，i_n）) 会产生一个类型为 Dtype(a) 的标量值。我们通常使用 $a[i_1, i_2, \ldots, i_n]$作为同一索引操作的简写。

现在，我们定义与数组相关联的定义域和陪域如下。

定义 2（数组定义域）。 设 A是一个 n维数组，其形状为（s_1，.....，s_n）。 A 的定义域 记为 dom(A)，是笛卡尔积

$$\mathrm{dom}(A) = \prod_{i=1}^{n} \{0, 1, \ldots, s_i - 1\},$$

也就是说，用于访问 A单元的所有有效索引元组的集合 .

定义 3（数组陪域）。 设 A 为一个多维数组。 A的陪域 记为 codomain(A)，是所有 Dtype(A)类型标量值的集合。

2.2 爱因斯坦记号

广为人知的爱因斯坦求和约定 [15] 最初是作为张量上的缩并运算记号而提出的。在科学软件中，该定义的一个略微修改版本被用于多维数组上的缩并运算。现在，我们引入一种语法，以便对爱因斯坦求和形式的表达式进行计算表示。

定义 4（索引）。 一个索引， i 是一个符合清单 1所示语法的表达式。

```
<索引> = “a” | “b” | “c” | .. | “z”
```

清单1. 索引的语法

我们使用索引表达式来定义索引列表。

定义 5 (索引列表**)**。 索引列表 $\mathcal{I}$ 是由一系列索引表达式构成的表达式，如列表 2 中的语法所示。

我们使用下标记法 I_k 来指代索引列表中的第 k 个索引。

```
<索引列表> := <索引 >*
```

清单2。索引列表的语法，使用索引清单1

定义 6 （索引列表长度）。若 $\mathrm{I} = (i_1, i_2, \ldots\ldots, i_n)$ 是一个索引列表，其中 $i_1, \ldots\ldots, i_n$ 为索引表达式，则 I 的长度记作 $|\mathrm{I}|$，其值等于 n。

定义 7 （爱因斯坦求和（“爱因斯坦求和”））。爱因斯坦求和是一个四元组 $(n, \mathrm{I}^{\text{out}}, (\mathrm{A}^k)_{k=1}^n, (\mathrm{I}^{\text{in},\ k})_{k=1}^n)$，它表示一个对应于多维数组 R的表达式，使得

- n 是一个非负整数，对应于该表达式的输入数量。
- I_{out} 是一个索引列表，对应于 R中的索引模式。
- $(\mathrm{A}_k)_{k=1}^n$ 是多维数组的符号序列，这些数组构成表达式的输入。
- $(\mathrm{I}_{\text{in}}{}^k)_{k=1}^n$ 是索引列表的序列，用于表示输入多维数组中的索引行为。
- R 计算如下：

$$R[\mathcal{I}_1^{\text{out}}, \ldots, \mathcal{I}_{|\mathcal{I}^{\text{out}}|}^{\text{out}}] = \sum_{\bigcup_{j=1}^{n} (\mathcal{I}^{\text{in},j} \setminus \mathcal{I}^{\text{out}})} \left(\prod_{k=1}^{n} \mathcal{A}^k [\mathcal{I}_1^{\text{in},k}, \ldots, \mathcal{I}_{\text{Dim}(\mathcal{A}^k)}^{\text{in},k}] \right). \tag{1}$$

可以看出，要使（1）成立，必须满足以下性质：

- 对于所有 $k \in \{1, \ldots, n\}$，$|\mathrm{I}_{\text{in}}{}^k|$ 必须等于 $\mathrm{Dim}\ (\mathrm{A}^k)$。
- 维数(R) 必须等于 $|\mathrm{I}_{\text{输出}}|$。
- 如果 (k_1, i_1, k_2, i_2) 满足 $\mathrm{I}_{\text{输入}}{}_{i_1}{}^{k_1} = \mathrm{I}_{\text{输入}}{}_{i_2}{}^{k_2}$，那么 $(\text{形状}_k(\mathrm{A}^{k_1}))_{i_1}$ 必须等于 $(\text{形状}(\mathrm{A}^{k_2}))_{i_2}$。
- 如果 $i \in \{1, \ldots\ldots, |\mathrm{I}_{\text{输出}}|\}$，那么 $\mathrm{I}_{\text{输出}i}$必须是 $\bigcup_{k=1}^{n} \{\mathrm{I}_{\text{输入}_1}{}^k, \ldots\ldots, \mathrm{I}_{\text{输入}\ \mathcal{I}_{\text{输入}\,k}}\}$ 的子集。

示例 1 （爱因斯坦求和）。由元组 $(2, (i, j), ((i, k), (k, j)) (A^{10\times4}, B^{4\times10}))$ 定义的爱因斯坦求和对应于一个矩形矩阵乘法运算，其输出 $R^{10\times10}$ 计算如下：

$$R[i, j] = \sum_{k=0}^{3} A[i, k] \cdot B[k, j].$$

定义 8 （索引列表的相等性）。我们称两个索引列表 I 和 I′ 相等，即 $\mathrm{I} = \mathrm{I}'$，当且仅当：

- $|\mathrm{I}| = |\mathrm{I}'|$。
- 对于所有 $k \in \{1, \ldots, |\mathrm{I}|\}$，$\mathrm{I}_k = \mathrm{I}'_k$。

定义 9 （爱因斯坦求和的相等性）。我们称由元组 $(n, \mathrm{I}^{\text{out}}, (\mathrm{I}^{\text{in},k})_{k=1}^n, (\mathrm{A}^k)_{k=1}^n)$ 定义的两个爱因斯坦求和 e_1和由元组 $(n, \mathrm{I}'^{\text{out}}, (\mathrm{I}'^{\text{in},k})_{k=1}^n, (\mathrm{A}'^k)_{k=1}^n)$ 定义的 e_2相等，即 $e_1 = e_2$，当且仅当：

- $\mathcal{I}_{\text{输出}} = \mathcal{I}'_{\text{输出}}$。
- 对于所有 $k \in \{1, \ldots, n\}$，$\mathrm{I}_{\text{in}}{}^k = \mathrm{I}'_{\text{in}}{}^k$。
- 对于所有的 $k \in \{1, \ldots\ldots, n\}$，$\mathrm{A}^k = \mathrm{A}'^k$。

定义 10 （爱因斯坦求和中的所有索引）。 如果 e是由元组 $(n, \mathrm{I}^{\mathrm{out}}, (\mathrm{I}^{\mathrm{in},k})_{k=1}^{n}, (\mathrm{A}^{k})_{k=1}^{n})$ 定义的爱因斯坦求和，则 $\mathrm{I}^{\mathrm{all}}(e)$ 表示该爱因斯坦求和指标记号中的所有索引，即：

$$\mathcal{I}^{\mathrm{all}}(e) = \bigcup_{k=1}^{n} \mathcal{I}^{\mathrm{in},k}.$$

3 相关工作

爱因斯坦求和是 n维数组操作，在科学计算的诸多定义域中都很常见。爱因斯坦求和如此普遍，因此科学计算框架提供相应的基本操作至关重要。早期框架（如 BLAS [36], LAPACK [6], 提供了实现矩阵乘法和转置的抽象，这些操作可以组合起来执行爱因斯坦求和。Åhlander 等人 [4, 5] 进一步采纳了这一思路，开发了一个 C++ 语言库，提供爱因斯坦求和基本操作，并将其用于高层偏微分方程（ PDE ）和迭代线性求解器。Hirata 等人 [20] 提出了张量收缩引擎（TCE），这是一个高层 n维数组操作框架，采用爱因斯坦记号描述量子化学计算。NWChem 等计算化学框架使用了 TCE 的指标记法 [52]。 NumPy [17]也提供爱因斯坦求和基本操作，此后被众多基于 Python的机器学习框架广泛采用。因此，TensorFlow [2], JAX [16], PyTorch [42], 等都提供“einsum”基本操作。Vasilache 等人 [54] 提出了张量推导，这是一种用于通过爱因斯坦记号表达机器学习操作的高层领域专用语言，随后通过多面体模型将其降低为机器码。Rogozhnikov [46] 开发了一个 n维数组库，依靠指标记法指定数组操作基本操作。此外，在张量代数编译器中，Kjolstad 等人 [25] 扩展了爱因斯坦记号，使其可用于稀疏操作数，以映射有限元方法（FEM）、稀疏神经网络等领域中的操作。

einsum 原语在各种计算框架中无处不在，因此出现了大量后续工作，致力于对其进行优化以实现高性能。优化的 BLAS 实现（例如 OpenBLAS [57], Intel MKL [55], BLIS [53], LibXSMM [18], ）提供了通用矩阵乘法（GEMM）的优化实现。这些高性能 GEMM 和转置操作是早期张量收缩算法（例如转置-转置-通用矩阵乘法-转置 [20] （TTGT））不可或缺的组成部分。张量收缩将矩阵乘法推广到多维数组，是爱因斯坦求和的一项基本子操作。通常会将其与收缩路径优化算法结合，以浮点运算次数最优的方式计算爱因斯坦求和。La m 等人 [13]指出，该优化问题是 NP难问题。Pfeifer 等人 [43] 提出了一种针对收缩路径搜索空间的剪枝策略。Smith 等人 [1] 在 Opt-Einsum 中实现了这一搜索启发式算法及其他算法；该工具后来也被 NumPy、JAX 等采用。此外，还有大量工作提出了适用于张量收缩程序的领域专用循环变换。我们请读者参阅 Springer 等人 [50] 的研究，其中综述了多种张量收缩方法，例如 TTGT、GEMM循环和嵌套循环方法。此外，作者还提出了类GEMM张量-张量乘法（GETT）：该方法避免显式生成转置，而是在适合放入缓存的分块操作数上执行转置；这些操作数随后作为 GEMM微内核的输入。Kim 等人 [23]开发了一种面向图形处理器的张量收缩算法。该算法将分块操作数预取到暂存存储器中，并通过外积构造分块输出。预取和外积计算由工作组中的工作项并行执行。在 TVM 中，Chen 等人 [12] 提出了一种基于自动调优的降级遍，可探索各种循环变换。研究表明，其基于模拟退火的自动调优器，性能可媲美通过 cuBLAS 实现的较简单手动调优张量收缩。

高性能张量收缩/爱因斯坦求和的可用性，也催生了依赖这些高级例程的关键应用。Rocktäschel [45] 在机器学习工作负载的背景下展示了这一点。在一项先前研究中 [34],我们证明，爱因斯坦求和是端到端有限元时间步进器的核心操作。Julien 等人 [26]展示了将数组表达式分解为爱因斯坦求和可能带来的收益。Barthels 等人 [10] 提出了一种编译器，可接受数组表达式，并将其改写为经过优化的 BLAS 和 LAPACK 例程。

为利用多次调用线性代数原语时的数据重用，并提高宽 SIMD 架构的利用率，人们对批量线性代数运算符进行了研究。Abdelfattah 等人 [3] 开发了一种基于自动调优的策略，用于在 GPU 架构上生成优化的批量 GEMM 例程。Dongarra 等人 [14] 采用跨矩阵向量化，在宽 SIMD 架构上实现高效的批量 GEMM 和三角求解器。Kim 等人 [22] 提出了一种用于张量收缩批次的降级策略。在这项工作中，我们为一组 einsum 操作引入了一种计算原语，称为批量 einsum。现有的批处理爱因斯坦求和方法通常会引入一个额外的索引来表示批次维度；然而，这类表述无法捕捉细粒度的操作数复用，并会施加任意的物化约束，从而导致性能下降。

总之，爱因斯坦求和是科学计算代码中的一种关键计算结构。人们提出了许多编译爱因斯坦求和的算法，但如 [50], 表 3、4 所示，没有任何算法能在所有情况下严格达到 Roofline 性能。因此，许多科学计算框架选择使用手写内核。在先进的有限元求解器 [30, 31]中也能看到这一点。为了解决这些问题，我们在这项工作中提出了一种软件框架，用于为批量 einsum 实例整理近屋顶线内核，同时允许这些整理好的知识在用户驱动的计算场景中重复使用。将批量 einsum 实例编码到用于检索的表中的一个关键要求，是能够将其映射到范式。据我们所知，此前尚无工作探讨如何为 einsum 表达式计算范式，更不用说批量 einsum 了。

4 批量爱因斯坦求和

在计算科学应用中，反复调用与某个 einsum 实例相对应的内核十分常见，并且不同调用所访问的数据存在大量重叠。例如，在有限元求解器的热点循环中，就会频繁出现以下形式的表达式

$$\sum_{j,x} J[x,r,e]D[x,i,j]u_k[e,j], \qquad k \in \{1,\ldots,N_{\text{field}}\},$$

其中， J 是雅可比矩阵， D 是与参考单元相关联的散度矩阵， u_k 是通过仿射几何映射在共同离散化上定义的场向量。当这些 N_{field} 爱因斯坦求和被独立计算时，共享操作数 J 和 D 会被反复加载，从而增加动态随机存取存储器到处理单元的数据传输，并导致性能下降。电子结构计算以及线性和非线性求解器中也会出现类似模式。

为此，我们引入一种涵盖多个 einsum 操作的运算，称为批量爱因斯坦求和 运算符。

定义 11 （批量爱因斯坦求和）。 批量爱因斯坦求和是一个五元组（b、 n、I^{out}、 $(\mathrm{I}^{\text{in},\ j})_{j=1}^{n}$、 $((\mathrm{A}^{i,\ j})_{j=1}^{n})_{i=1}^{b}$），用于计算由 b 多维数组 $(R_i)_{i=1}^{b}$组成的序列。每个 R_i 都是一个 einsum 操作的结果，该操作由四元组（n、I^{out}、 $(\mathrm{I}^{\text{in},\ j})_{j=1}^{n}$、 $(\mathrm{A}^{i,\ j})_{j=1}^{n}$）定义，其中 $i \in${1、 2、.....、 b}。

现在我们将给出一个批量 *einsum*的示例。提交给ACM的稿件

示例 2（批量 einsum）。由以下元组定义的批量 einsum：

$$(2, 3, (i, j), ((i, k), (k, l), (l, j)), ((A^{10\times10}, B^{10\times10}, C^{10\times10}), (B^{10\times10}, C^{10\times10}, D^{10\times10})))$$

对应于数组 $R_1^{10\times10}$和 $R_2^{10\times10}$的计算，其计算结果为：

$$\begin{aligned} R_1[i,j] &= \sum_{k=0}^{9}\sum_{l=0}^{9} A[i,k]\,B[k,l]\,C[l,j], \\ R_2[i,j] &= \sum_{k=0}^{9}\sum_{l=0}^{9} B[i,k]\,C[k,l]\,D[l,j]. \end{aligned} \tag{2}$$

现在，我们将定义批量 einsum 上的一些操作，这些操作将在后续章节中发挥作用 s.

定义 12（批量 einsum 的相等性）。两个批量 einsum，$e_1 = \left(b, n, \mathrm{I}^{\mathrm{out}}, (\mathrm{I}^{\mathrm{in},j})_{j=1}^{n}, \left((\mathrm{A}^{i,j})_{j=1}^{n}\right)_{i=1}^{b}\right)$和$e_2 = \left(b, n, \mathrm{I}'^{\mathrm{out}}, (\mathrm{I}'^{\mathrm{in},j})_{j=1}^{n}, \left((\mathrm{A}'^{i,j})_{j=1}^{n}\right)_{i=1}^{b}\right)$相等，即 $e_1 = e_2$，当且仅当对于所有 $i \in \{1, \ldots, b\}$，由 $\left(n, \mathrm{I}^{\mathrm{out}}, (\mathrm{I}^{\mathrm{in},j})_{j=1}^{n}, (\mathrm{A}^{i,j})_{j=1}^{n}\right)$ 定义的爱因斯坦求和等于由 $\left(n, \mathrm{I}'^{\mathrm{out}}, (\mathrm{I}'^{\mathrm{in},j})_{j=1}^{n}, (\mathrm{A}'^{i,j})_{j=1}^{n}\right)$定义的爱因斯坦求和。

定义 13（批量 einsum 的所有参数）。设 $e = \left(b, n, \mathrm{I}^{\mathrm{out}}, (\mathrm{I}^{\mathrm{in},j})_{j=1}^{n}, \left((\mathrm{A}^{i,j})_{j=1}^{n}\right)_{i=1}^{b}\right)$ 是一个批量 einsum，则可得到该爱因斯坦求和的所有参数，即 $\mathrm{A}^{\mathrm{all}}(e)$，如下所示：

$$\mathcal{A}^{\mathrm{all}}(e) = \bigcup_{i=1}^{b} \{\mathcal{A}^{i,1}, \ldots, \mathcal{A}^{i,n}\}.$$

5 同构批量爱因斯坦求和

在图 1 中，我们强调了批量 einsum 表达式缺乏规范性的特点：语法形式不同的批量 einsum 表达式可能对应同一运算。本节将正式定义批量 einsum 的等价类，并将其称为同构批量 *einsum*。

首先，我们介绍同构爱因斯坦求和的概念，然后用该概念定义同构批量 einsum。

定义 14（Einsum 同构）。若两个爱因斯坦求和 $e_1 = \left(n, \mathrm{I}^{\mathrm{out}}, (\mathrm{I}^{\mathrm{in},j})_{j=1}^{n}, (\mathrm{A}^{j})_{j=1}^{n}\right)$ 和 $e_2 = \left(n, \mathrm{I}'^{\mathrm{out}}, (\mathrm{I}'^{\mathrm{in},j})_{j=1}^{n}, (\mathrm{A}'^{j})_{j=1}^{n}\right)$ 同构，即 $e_1 \simeq e_2$，则有 $\left|\mathrm{I}^{\mathrm{all}}(e_1)\right| = \left|\mathrm{I}^{\mathrm{all}}(e_2)\right|$，且存在三个替换映射 σ^j、$\sigma^{\mathcal{I}}$ 和 $\sigma_{\mathcal{A}}$，使得：

- σ^j：{1、2、.....、 n} →{1、2、.....、 n} 是两个爱因斯坦求和之间对应操作数顺序的双射。
- $\sigma_{\mathcal{I}}$ ： $\mathrm{I}_{\mathrm{all}}(e_2)$ → $\mathrm{I}_{\mathrm{all}}(e_1)$ 是两个爱因斯坦求和之间对应索引名称映射的双射。
- $\sigma_{\mathcal{A}}$：{A'^1、.....、 A'^n} →{A^1、.....、 A^n} 是两个爱因斯坦求和之间对应参数名称映射的双射。
- 对于所有 $j \in${1、2、.....、 n}，形状 (A^{j}) =形状 $\left(\sigma_{\mathcal{A}}\left(\mathrm{A}'^{\sigma^j(j)}\right)\right)$以及数据类型 (A^{j}) =数据类型 $\left(\sigma_{\mathcal{A}}\left(\mathrm{A}'^{\sigma^j(j)}\right)\right)$。
- 对于所有 $j \in${1， 2， ....., n} 和 $d \in \{1, 2, \ldots, |\mathrm{I}^{d,\ in}|\}$, $|\mathrm{I}_{\text{属于}^j}| = |\mathrm{I}'_{\text{属于}\ \sigma^j(j)}|$ 以及 $\mathrm{I}_{\text{属于}\ d^{\ j}} = \sigma^{\mathcal{I}}\left(\mathrm{I}'_{\text{属于}\ d^{\ \sigma^j\ j}}\right)$。
- $\left|\mathcal{I}_{\text{输出}}\right| = \left|\mathcal{I}'_{\text{输出}}\right|$。
- 对于所有 $d \in \{1, 2, \ldots, |\mathrm{I}_{\mathrm{out}}|\}$， $\mathrm{I}_{\mathrm{out}\,d} = \sigma_{\mathcal{I}}\left(\mathrm{I}'_{\mathrm{out}\ d}\right)$。

示例 3 （同构的爱因斯坦求和）。 由元组 $\left(2, (i,j), ((i,k),(k,j)), \left(A^{10\times 4}, B^{4\times 10}\right)\right)$定义的爱因斯坦求和 e_1， 以及由元组 $\left(2, (p,q), ((r,q),(p,r)), \left(X^{4\times 10}, Y^{10\times 4}\right)\right)$定义的爱因斯坦求和 e_2，通过以下替换映射同构：

- $\sigma^j = \{1 \mapsto 2;\ 2 \mapsto 1\}$。
- σ我 $= \{p \mapsto i; q \mapsto j;\ r \mapsto k\}$.
- σA $= \{Y \mapsto A;\ X \mapsto B\}$。

定义 15 （批量 einsum 同构）。 我们称两个批量 einsum， $e_1 = \left(b, n, \mathrm{I}^{\mathrm{out}}, \left(\mathrm{I}^{\mathrm{in},\ j}\right)_{j=1}^{n}, \left(\left(\mathrm{A}^{i,\ j}\right)_{j=1}^{n}\right)_{i=1}^{b}\right)$和 $e_2 = \left(b, n, \mathrm{I}'^{\mathrm{out}}, \left(\mathrm{I}'^{\mathrm{in},\ j}\right)_{j=1}^{n}, \left(\left(\mathrm{A}'^{i,\ j}\right)_{j=1}^{n}\right)_{i=1}^{b}\right)$是同构的，即 $e_1 \simeq e_2$，如果存在四个替换映射σ^i, σ^j, $\sigma^{\mathcal{I}}$ 和 $\sigma_{\mathcal{A}}$，使得：

- $\sigma^i : \{1, 2, \ldots, b\} \to \{1, 2, \ldots, b\}$ is a bijective mapping.
- $\sigma^{\mathcal{A}} : \mathcal{A}^{\mathrm{all}}(e_2) \to \mathcal{A}^{\mathrm{all}}(e_1)$ is a bijective mapping.
- For all $i \in \{1, 2, \ldots, b\}$, the einsums defined by $\left(n, \mathcal{I}^{\mathrm{out}}, (\mathcal{I}^{\mathrm{in},j})_{j=1}^{n}, (\mathcal{A}^{i,j})_{j=1}^{n}\right)$ is isomorphic to $\left(n, \mathcal{I}'^{\mathrm{out}}, (\mathcal{I}'^{\mathrm{in},j})_{j=1}^{n}, \left(\mathcal{A}'^{\sigma^i(i),j}\right)_{j=1}^{n}\right)$ using the substitution mappings σ^j, $\sigma^{\mathcal{I}}$ and $\sigma_i^{\mathcal{A}}$, where $\sigma_i^{\mathcal{A}} = \left\{\mathcal{A}^{i,j} \mapsto \sigma^{\mathcal{A}}\left(\mathcal{A}'^{\sigma^i(i),\sigma^j(j)}\right) : j \in \{1, \ldots, n\}\right\}$.

例 4 （同构批量 einsum）。 考虑由以下定义的批量 einsum e_1

$$\left(2, 4, (i), ((i,j,k), (i,k), (i,j), (i,j)), \left(\left(A^{5\times 10\times 10}, B^{5\times 10}, C^{5\times 10}, D^{5\times 10}\right), \left(A^{5\times 10\times 10}, B^{5\times 10}, C^{5\times 10}, B^{5\times 10}\right)\right)\right),$$

以及由以下定义的 e_2

$$\left(2, 4, (i), ((i,k,j), (i,k), (i,k), (i,j)), \left(\left(P^{5\times 10\times 10}, S^{5\times 10}, R^{5\times 10}, S^{5\times 10}\right), \left(P^{5\times 10\times 10}, Q^{5\times 10}, R^{5\times 10}, S^{5\times 10}\right)\right)\right),$$

那么 $e_1 \simeq e_2$，使用替换映射：

- $\sigma^i = \{1 \mapsto 2; 2 \mapsto 1\}$。
- $\sigma^j = \{1 \mapsto 1;\ 2 \mapsto 4;\ 3 \mapsto 3;\ 4 \mapsto 2\}$.
- σ我 $= \{i \mapsto i;\ j \mapsto k;\ k \mapsto j\}$。
- σ一个 $= \{P \mapsto A;\ S \mapsto B;\ R \mapsto C; Q \mapsto D\}$.

6 批量 einsum 的规范形式

两个爱因斯坦求和（记为 e_1 and e_2）同构的一个直接推论是：为计算 e_1 而开发的算法过程，同样可以用于计算 e_2，前提是对相应变量进行适当的重新标记。判断两个爱因斯坦求和是否同构的一种方法，是为批量 einsum 定义一种规范形式，使得同构批量 einsum 具有相等的规范化表示。

定义 16 （批量 einsum 的规范化）。 令 B 表示所有批量 einsum 的集合。如果对于任意批量 einsum e，其规范形式 C(e) 满足以下条件，则称函数 C：$B \to B$ 为规范化函数：

$$e \simeq e' \ \Leftrightarrow\ C(e) = C(e'),$$

对于所有批量 einsum $e' \in B$。

在本节中，我们介绍这样一种规范化函数。该过程包含三个步骤：(i) 将批量 einsum 映射为一个彩色有向图，称为诱导图；(ii) 对诱导图进行规范化；(iii) 已提交至 ACM 的手稿

将规范化后的图映射回批量 einsum。批量 einsum 与诱导图之间的映射详见附录 B.1和附录 B.2。现在，我们将利用这些映射，开发一种基于彩色图规范化的规范化算法 [39]。

我们在过程 CanonicalizeBatchedEinsum 中描述了我们的批量 einsum 规范化算法，该过程见算法 3。该过程计算 $e' = C(e)$，并生成替换映射 $\sigma_{\mathcal{A}}$和 σ_I。映射$\sigma_{\mathcal{A}}$和 σ_I将 e'中的数组和索引符号映射到 e中对应的符号，如定义 15所述。

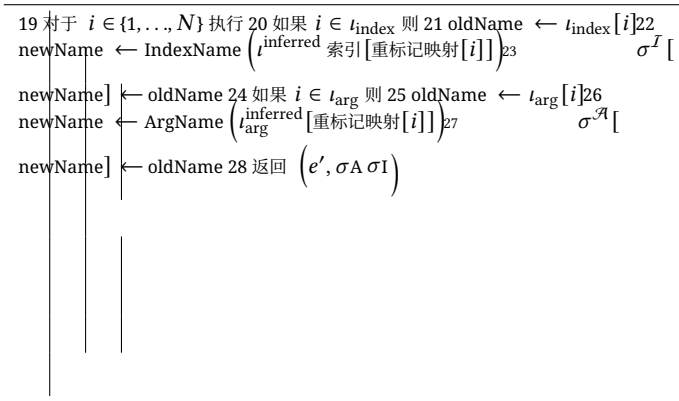
算法 3 首先构造一个表示批量 einsum 的诱导图 e(第 2行）。组件 A、c、 ι_{dtype}、 ι_{length}、 ι_{index}和 ι_{arg} 分别对应定义 24中所定义的诱导图组件。 A 是对应有向图中各边的邻接矩阵， c 是对应图中节点颜色的列向量，ι_{dtype} 是将诱导图中对应数据类型的节点标签映射到其数值数据类型的映射，ι_{length} 是将诱导图中对应轴长度的节点标签映射到其轴长度的映射，而 ι_{index} 是将 e 的索引对应的节点标签映射到其索引的映射。

然后，我们调用图规范化器生成重标记映射，将构造出的图中的节点与规范图中的对应节点关联起来。第 8–15 行应用此重标记映射，得到诱导图的规范形式。在第 16 行，我们使用附录 B.2 中的批量爱因斯坦求和重构过程，得到 e'，即重构后的批量爱因斯坦求和。在此重构过程中， $\iota^{\text{inferred}}_{\text{索引}}$ 和 $\iota^{\text{inferred}}_{\text{arg}}$分别表示过程中选取的索引和参数的枚举，如附录**步骤 4.** 和**步骤 5.** 中所述B.2。最后，在第 19–27 行，我们构造参数和索引替换映射，分别记为 $\sigma_{\mathcal{A}}$和 σ_I。

算法 3 批量爱因斯坦求和的规范化

要求： 过程 IndexName、ArgName。这些过程接受一个非负整数，并以确定的方式分别返回唯一的索引名称和批量爱因斯坦求和中唯一的数组操作数名称。参见附录 B.2 了解正式定义。

1 **过程**CanonicalizeBatchedEinsum(e)
2 $A, c, \iota_{\text{dtype}}, \iota_{\text{length}}, \iota_{\text{index}}, \iota_{\text{arg}} \leftarrow$ ToInducedGraph(e) ▷ 参见附录B.1。
3 重标记映射 $\leftarrow$ 获取规范标签(A, c) ▷ 任何图规范化 [21, 40]实现。
4 $N \leftarrow$ 图中的节点数对应于 A。
5 $A' \leftarrow$ 大小为 $N \times N$ 的零矩阵。
6 $c' \leftarrow$ 大小为 N 的零向量。
7 $\iota'^{\text{数据类型}}_{\text{长度}} \leftarrow$空映射。
8 **对于** $i \in \{1, \ldots, N\}$ **执行**
9 **对于** $j \in \{1, \ldots, N\}$ **执行**
10 A'[重标记映射[i]，重标记映射[j]] $\leftarrow A[i, j]$
11 c'[重标记映射[i]] $\leftarrow c[i]$
12 **如果** $i \in \iota_{dtype}$ **则**
13 ι' 数据类型[重标记映射[i]] $\leftarrow \iota_{\text{dtype}}[i]$
14 **如果** $i \in \iota_{length}$ **则**
15 ι' 长度[重标记映射[i]] $\leftarrow \iota_{\text{length}}[i]$
16 e' $\iota'^{\text{推断得出}}_{\text{索引}}$ $\iota^{\text{推断得出}}_{\text{arg}} \leftarrow$ ToBatchedEinsum(A', c', ι' 数据类型，ι' 长度, IndexName, ArgName) ▷ 参见附录B.2。
17
18 $\sigma_{\mathcal{A}}\, \sigma_I \leftarrow$ 空映射。



备注 1. 由于我们的规范化算法包含图规范化，因此它满足幂等性判据，即对于所有批量 *einsum* e，$C(e) = C(C(e))$。

示例 5. 对于由 $\left(1, 2, (i)、((ij), (ik))、((A, B))\right)$ 定义的爱因斯坦求和 e_1，其形状为 Shape(A) = Shape(B) = (72, 18)，数据类型为 Dtype(A) = Dtype(B) = F64，我们得到 C(e_1)，它是由 $\left(1, 2, (a)、((ab), (ac))、((A_0, A_1))\right)$ 定义的批量 einsum，其形状为 Shape(A_0) = Shape(A_1) = $(72, 18)$，数据类型为 Dtype(A_0) = Dtype(A_1) = F64。

对于由 $\left(1, 2, (i)、((ik), (ij))、((X, Y))\right)$ 定义的爱因斯坦求和 e_2，其形状为 = Shape(Y) = $(72, 18)$，数据类型为 =Dtype(Y) = F64，我们得到 C(e_2)，它是由 $\left(1, 2, (a)、((ab), (ac))、((A_0, A_1))\right)$ 定义的批量 einsum，其形状为 = Shape(A_1) = $(72, 18)$，数据类型为 = Dtype(A_1) = F64。由于 C(e_1) = C(e_2)，e_1 必须与 e_2 同构。

值得注意的是，批量 einsum 存在多个可能的规范化函数。我们提出的方法依赖于图规范化过程。此外，在我们的方法中，将底层图规范化器替换为不同的实现，会产生不同的规范化函数。

这种规范化对于对第 7 节所述软件系统中的批量 einsum 进行制表至关重要。

7 优化批量爱因斯坦求和在实际程序中的集成

本节介绍我们在高级程序中集成和实际复用批量 einsum 优化实现的策略。我们实现了一个软件系统，用于维护一个作用于函数式批量 einsum 的变换计划库。这些变换计划由性能工程师或自动调优器添加到该库中。此外，我们的系统还维护一个数据库，其中存储了针对函数式批量 einsum 的特定实例、通过实测获得的库中变换的性能数据；对于这些实例，每个操作数对应的函数都具有以下形式 $(i_1, \ldots, i_n) \mapsto A[i_1, \ldots, i_n]$，其中 A 是一个按行优先顺序排列的多维数组。我们将这类函数式批量 einsum 实例称为 理想化函数式批量 *einsum*。计算科学家利用这些已知的理想化函数式批量 einsum 性能数据，优化以函数式批量 einsum 表达的程序。我们将其实现为一个名为 Python 库的 Feinsum[3]。我们在图 Feinsum 中总结了其核心组件，以及计算科学家和变换写入器与该系统交互的方式，见图 2。

[3]Feinsum 是开源软件，可通过以下网址公开获取：
https://pypi.org/project/feinsum。已提交至 ACM 的手稿

对于 Feinsum 这类采用基于制表的方法来提供优化子程序的软件系统而言，一个关键的设计选择是计算原语的结构。诸如 BLAS、oneDNN [37],和 LAPACK 之类的软件系统，会为计算科学家提供包含计算原语优化子程序的二进制文件。这些方法将操作数作为内存中的缓冲区接收。然而，以缓冲区形式接收操作数会引入与缓冲区分配和初始化相关的额外开销。我们此前曾在算法 1 中观察到一个由抽象引起的此类开销实例。为避免与物化相关的开销，我们扩展了批量 einsum 的概念，使其允许将 Lambda 表达式而非已物化的缓冲区作为输入。我们将其称为 函数式批量 *einsum* 表达式。随 Feinsum 提供的变换库作用于此类函数式批量 einsum。性能数据数据库包含函数式批量 einsum 理想化版本的测量结果。现在，我们将正式定义函数式批量 einsum 及其相关的理想化函数式批量 einsum。

定义 17 （函数式批量爱因斯坦求和）。 设 $e = (b, n, \mathrm{I}^{\mathrm{out}}, (\mathrm{I}^{\mathrm{in},j})_{j=1}^{n}, ((\mathrm{A}^{i,j})_{j=1}^{n})_{i=1}^{b})$ 为一个批量 einsum。设 M 为一个映射，它为每个 $A \in \mathrm{A}^{\mathrm{all}}(e)$ 分配一个函数

$$\mathcal{M}(A) : \mathrm{dom}(A) \to \mathrm{codomain}(A).$$

由这对 (e,M) 定义的函数式批量 *einsum* 会计算一个由 b 多维数组组成的序列 $(R_i)_{i=1}^{b}$，满足对于每个 $i \in \{1, \ldots, b\}$,

$$R_i[\mathcal{I}_1^{\mathrm{out}}, \ldots, \mathcal{I}_n^{\mathrm{out}}] = \sum_{\cup_{j=1}^{n}(\mathcal{I}^{\mathrm{in},j} \setminus \mathcal{I}^{\mathrm{out}})} \left(\prod_{k=1}^{n} \mathcal{M}(\mathcal{A}^{i,k})(\mathcal{I}_1^{\mathrm{in},k}, \ldots, \mathcal{I}_{\mathrm{Dim}(\mathcal{A}^k)}^{\mathrm{in},k}) \right).$$

定义 18 （理想化函数式批量 einsum）。 设 $f = (e,\mathrm{M})$ 为一个函数式批量 einsum，且满足

$$\mathcal{M} = \left\{ A \mapsto \lambda i_1 \ \ldots \ i_n \ . \ P_A[i_1, \ldots, i_n] \ : \ A \in \mathcal{A}^{\mathrm{all}}(e), \ n = \mathrm{Dim}(A) \right\},$$

其中 P_A 表示与 A 关联的行优先多维数组的底层缓冲区，且程序中的其他数组均不与其别名。此时， f 称为理想化函数式批量爱因斯坦求和。

从本质上说，Feinsum 使计算科学家（或高级编译器）能够将针对理想化函数式批量爱因斯坦求和已知的高性能变换复用于类似批量 einsum 的程序。一般而言，函数式批量爱因斯坦求和的收益分析取决于$\mathcal{M}$中的表达式。尽管如此，我们认为，$\mathcal{M}$ 存在一大类选择，在真实场景中经常出现，并且可以沿用此类收益分析。为具体说明这一点，我们将若干微架构的饱和算术强度与 TCCG 基准测试套件中 einsum 的算术强度进行比较 [49]。比较结果如图 3所示。我们观察到，在 NVIDIA P100、NVIDIA Titan V、NVIDIA H100 和 AMD MI250X 等图形处理器上，平均有 60.4% 的基准测试条目受内存带宽限制。理想化函数式批量 einsum 受内存带宽限制，这意味着任何在不改变内存访问模式的情况下对操作数执行额外计算的函数式批量 einsum，都表现出相似的性能特征。因此，复用为理想化函数式批量爱因斯坦求和开发的变换通常是一种合适且有效的策略。

在我们的Feinsum实现中，我们使用Loopy 工具包 [27]。 Loopy 既是一种用于指定语法的语言，也是一种用于定义子程序变换的框架。我们在附录A中概述了 Loopy 中间表示（中间表示）。需要强调的是，本文介绍的所有概念都不依赖于Loopy。所提出的方法可以轻松地在任何高级中间表示中实现，例如

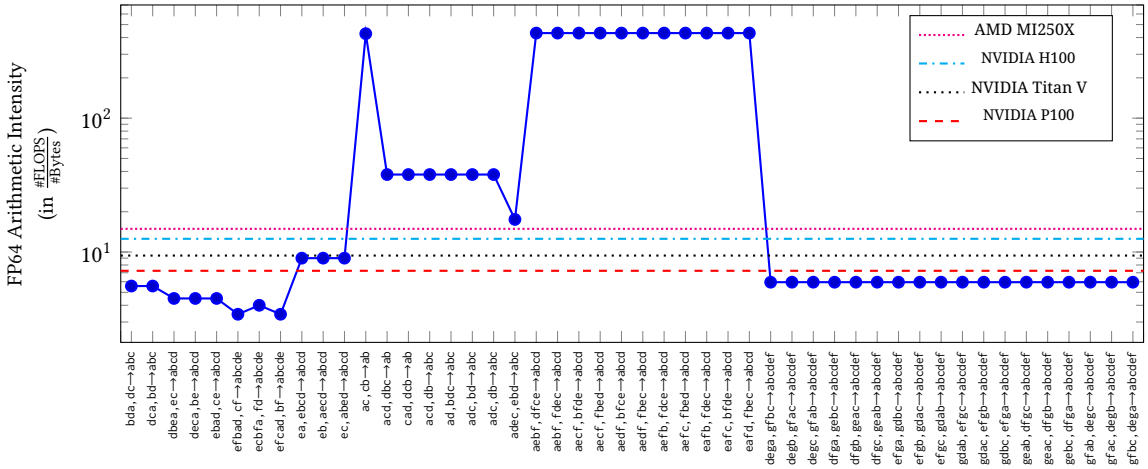


图 3. 使用双精度操作数的 TCCG 基准测试套件的算术强度（AI）。水平虚线表示现代加速器的饱和算术强度，超过该值后，工作负载将变为计算受限。

MLIR [35], Triton [51] 和Halide [44], 只要其表示支持用于表达多维数组、循环嵌套和函数的抽象语法树（AST）节点即可。

现在，我们通过一系列示例详细说明图 2 所示的流程。清单 3给出了一个来自变换库的变换示例。清单 4 展示了从 Feinsum 数据库中检索的内容，该检索会变换一个函数式批量爱因斯坦求和实例。最后，清单 5 展示了为清单 4中的检索生成的代码，假设清单 3 中的变换是数据库中已知的最佳变换。生成的代码会先计算所需的替换映射，然后应用库指定的变换。此外，与操作数对应的函数求值会融合到内核内部循环中。

```
1 import feinsum as fs 2 import loopy as lp 3 4 A = fs.array("A", (96, 4)) 5 B, C =
fs.array("B", (4,)), fs.array("C", (4,)) 6 ref_ensm = fs.batched_einsum("ij,j->i", [[
A, B], [A, C]]) 7 8 def transform(K): 9 sigma =fs.identify_as_einsum(K, ref_ens
m) 10 # 将 i- 循环按 32 个元素分块。 11 K = lp.split_iname(K, sigma ["i"], 32) 12 #
预计算 "A"，因为它会被使用两次。 13 K = lp.precompute(K, sigma ["A"],
sweep_inames=()) 14 return K 15 16 if __name__ =="__main__": 已提交至 ACM 的手稿
```

```
17  fs.record_facts(ref_ensm , 设备, __file__)
```

清单 3. 变换编写者眼中的 Feinsum。变换编写者将实现一个针对内核 K 的变换计划。此处展示的是针对批量爱因斯坦求和所实现的变换，该批量爱因斯坦求和对应于 ref_ensm 所表示的两个矩阵-向量乘法。

现在我们详细考察清单 3。代码变换在变换例程中指定，该例程以某个内核的中间表示（中间表示）为输入，并返回变换后的内核。此变换针对批量 einsum (2, 2, (*i*), ((*i j*), (*j*)), ((*A*, *B*), (*A*, *C*))) 定义。它首先调用 identify_as_einsum，将输入内核中的语句解释为函数式批量爱因斯坦求和，并计算一个重命名映射，将参考爱因斯坦求和中的变量映射到内核中间表示中的变量。将较低层级的中间表示解释为较高层级的操作（例如函数式批量爱因斯坦求和），通常称为提升 [11]。规范化是 identify_as_einsum 的关键组成部分；算法 4 给出了该过程的伪代码。随后，使用得到的重命名映射应用分块变换（lp.split_iname），并在私有内存中预计算该 A 操作数，从而避免对全局内存的冗余访问。脚本的最后一行调用 record_facts，记录应用变换例程后与 ref_ensm 对应的理想化函数式批量爱因斯坦求和的运行时间。在记录墙钟时间之前，我们会在设备上执行几次预热运行，以减轻瞬态效应。ref_ensm 的规范形式用作标识符，性能数据以此为键存储。

算法 4 用于将内核 K 识别为参考爱因斯坦求和 e_{ref} 的算法。

过程 IdentityAsEinsum(K, e_{ref})
  e_{raised}, $\sigma^{\mathcal{A}}$ σ^{I} ← 将爱因斯坦求和提升为批量爱因斯坦求和(K)
  e'_{ref} $\sigma^{\mathcal{A}}$ 提升后的 σ^{I} 提升后的 ← CanonicalizeBatchedEinsum(e_{ref})
  e' 提升 $\sigma^{\mathcal{A}}$ 规范化–到–参考 σ^{I} 规范化–到–参考 ← CanonicalizeBatchedEinsum(e_{raised})
  如果 e' ref $\neq e'$ **then**
    引发异常。 引发
  $\sigma^{\mathcal{A}} \leftarrow \{\}$
  $\sigma^{I} \leftarrow \{\}$
  对于 a_{raised}, $a_K \in \sigma^{\mathcal{A}}$ 引发 **do**
    σ一个 $[\sigma^{\mathcal{A}}$ $[(\sigma$A $)^{-1}[a_{\text{raised}}]]] \leftarrow a_K$
  对于 i_{raised}, $i_K \in \sigma^{I}$ 引发的 **do**
    σI $[\sigma^{I}$ 将–规范化为–ref $[(\sigma$I 将–规范化为–raised $)^{-1}[i_{\text{raised}}]]] \leftarrow i_K$

```
导入 loopy 为lp
导入 feinsum 为fs

内核 =lp.make_kernel(
  "{ [i0,i1] : 0<=i0<96 and 0<=i1<4 }" ,
  """
  u(i, j) := P[i, j]*P[i, j]
  v(i) := 3*cos(Q[i])+5
  w(i) := 正弦函数(R[i])
  for i0
    y1[i0] = 求和([i1], u(i0, i1)*v(i1))
    y2[i0] = sum([i1], u(i0, i1)*w(i1))
  end
  """)

变换 = fs.retrieve(knl , 设备 =...)
transformed_knl =变换(knl)
```

清单 4. 计算科学家视角下的 Feinsum。在 Feinsum 数据库中查询性能最佳的变换。

```
对于 (i0_out = 0; i0_out < 3; ++i0_out)
  对于 (i0_in = 0; i0_in < 32; ++i0_in) {
    双精度 acc_i1 = acc_i1_0 = 0.0;
    for (int i1 = 0; i1 < 4; ++i1) {
      双精度 p =P[i1+4*(32*i0_out+i0_in)]
              *P[i1+4*(32*i0_out+i0_in)];
      acc_i1 += p*(3*cos(Q[i1])+5);
      acc_i1_0 += p*( 正弦函数(R[i1]));
    }
    y1[32*i0_out +i0_in] =acc_i1;
    y2[32*i0_out +i0_in] =acc_i1_0;
  }
```

清单 5. OpenCL清单 4按清单 3 变换而成的子程序。

现在我们来考察清单 4所展示的计算科学家的视角。计算科学家将函数式批量爱因斯坦求和表示为一个Loopy内核，并调用 检索 例程，以获取存储在 Feinsum 数据库中、对应理想化函数式批量爱因斯坦求和的性能最佳变换。在调用 检索时，会遍历内核的中间表示，并将其提升为函数式批量爱因斯坦求和，随后对其进行规范化并用于查询数据库。检索后生成的代码如清单5所示，这里假设检索到的变换正是清单3中的变换。我们重点展示变换的迁移，尤其是经过分块的循环和经过预计算的操作数。

8 实验结果

在本节中，我们将介绍一系列实验，旨在回答三个问题。首先，与使用一系列经过优化的 einsum 子程序相比，在哪些真实场景中，使用批量 einsum 的优化子程序更有优势？我们将在第 8.1 节通过一个案例研究来探讨这一问题，该案例研究涉及间断伽辽金有限元法（DG-FEM）求解器中的批量 einsum 表达式。其次，与传统的存储对应于优化子程序的可执行文件的方法相比，我们选择复用针对函数式批量 einsum 的优化是否更有优势？我们将在第 [49] 8.2 节通过评估 TCCG 套件中的爱因斯坦求和来考察这一问题。最后，我们的 识别并变换策略所引入的编译时开销是否可控？我们将在第 8.3 节研究 TCCG 基准测试套件的这些代码生成成本。

8.1 DG-FEM 算子中的批量 einsum

间断伽辽金有限元法 [19] （DG-FEM）是一种用于求解偏微分方程的常用数值方法。DG-FEM 之所以具有吸引力，是因为它能在一般网格上实现高阶精度，同时保持局部守恒。DG-FEM 非常适用于通用图形处理器，因为它具有大量的单元局部计算，且与其他有限元方法相比，所需的跨单元通信相对较少。

Klöckner 等人 [30] 指出，三类批量 einsum 主要负责间断伽辽金有限元求解器中的大部分计算。这些批量 einsum 分别是：

面质量应用 这对应一类内核，可将其提升为由元组 $(b, 3, (e, i), ((f, e), (i, f, j), (f, e, j)), ((M, J, F^k))_{k=1}^n)$ 定义的批量爱因斯坦求和。这里，F^k 是数值通量场，M 是参考面质量矩阵，J 包含几何项。e 遍历网格中的单元，f 遍历一个单元的各个面，j 遍历输入场中位于单元各个面上的自由度 (DOF)，而 i 遍历输出场中位于单元体积内部的自由度。我们在清单 7 中展示了一个实现此类批量爱因斯坦求和函数式形式的内核。

局部散度计算 这些对应一类内核，可将其提升为由元组 $(b, 3, (e, i), ((x, r, e), (r, i, j), (e, j)), ((J, D, u^k))_{k=1}^n)$ 定义的批量 einsum。这里，u^k 是输入场，D 是参考导数矩阵，J 包含几何项。e 遍历网格中的单元，x 遍历拓扑维数，r 遍历环境函数空间的空间维数，而 j、i 分别遍历输入场和输出场的体积自由度。我们在列表 8 中展示了一个实现此类批量 einsum 函数式形式的内核。

局部梯度计算 这些对应一类内核，可识别为由元组 $(b, 3, (x, e, i), ((x, r, e), (r, i, j), (x, e, j)), ((J, D, u^k))_{k=1}^n)$ 定义的批量 einsum。这里，u^k 是要计算其局部梯度的场，D 是参考导数矩阵，J 包含几何项。e 遍历网格中的单元，x 遍历拓扑维数，r 遍历环境函数空间的空间维数，而 i、j 分别遍历输入场和输出场的体积自由度。我们在列表 9 中展示了一个实现此类批量 einsum 函数式形式的内核。

我们将上述批量 einsum 表示为 Loopy 内核，并使用集成在 Feinsum 库默认数据库中的变换。这些批量 einsum 操作的变换实现结合了循环置换与循环外提，以减少浮点运算总量（FLOPs）。此外，我们对矩阵 J和 D采用参数化分块策略，并使用参数化工作组网格。我们使用 OpenTuner [7]进行参数调优。为简洁起见，我们不再详述实现细节；有兴趣的读者可参阅我们先前研究的附录 C [34]，其中对此进行了更全面的讨论。

作为基线，我们还使用 JAX 实现了等效操作。JAX 是一种先进的数组库，利用惰性求值并支持算子融合。JAX 中的编译启发式依赖 cuBLAS 和 cuDNN，为线性代数原语提供经过优化的实现。在基于 JAX 的实现中，我们定义了一个对应于上述操作的 Python 可调用对象，并用 jax.jit 装饰它，以跟踪函数调用中的数组操作。

我们的所有实验均在搭载 CUDA 驱动程序 550.163 版本的 Nvidia Titan V GPU 上进行。变换后的内核使用 CUDA 工具包 12.4 版本中提供的 Nvidia OpenCL 内核编译器进行编译。

鉴于我们的评估旨在衡量所采用的变换策略相较于最先进方法的相对收益，而非进行绝对性能评估，因此我们报告变换后的内核相较于 JAX 版本所取得的加速比。加速比记为 η，计算公式如下：

$$\eta = \frac{\text{Wall-clock time for the JAX kernels}}{\text{Wall-clock time for the transformed kernels}}.$$

为避免瞬态效应，我们对每个内核多次执行所测得的墙钟时间取算术平均值。具体而言，我们持续启动内核，直至累计墙钟时间至少达到 2 秒，且内核调用次数不少于 20 次。在对某个内核进行实验之前，我们会先进行 5 轮预热，以进一步减少实验中的噪声。

我们在图 4 中展示了测得的加速比。图表显示，我们的宏内核测试套件获得了 1.6 至 100 倍的加速，证实了与当前最先进的方法相比，我们的变换抽象所带来的优势。我们预计，随着批量 einsum 中爱因斯坦求和的数量增加，加速比会更高，因为相对而言，这能大幅降低将数据加载到图形处理器动态随机存取存储器中的成本。不过，需要指出的是，数据并未始终呈现这一趋势。这主要是因为我们的数据库并未包含能够使每个 einsum 操作都达到 Roofline 性能的所有变换，因此实际结果偏离了理想趋势。

这项实验支持了我们的主张：将批量 einsum 用作一种基本操作，可以让计算实现通过隐藏不同 einsum 之间共享操作数的内存访问延迟来提升性能。

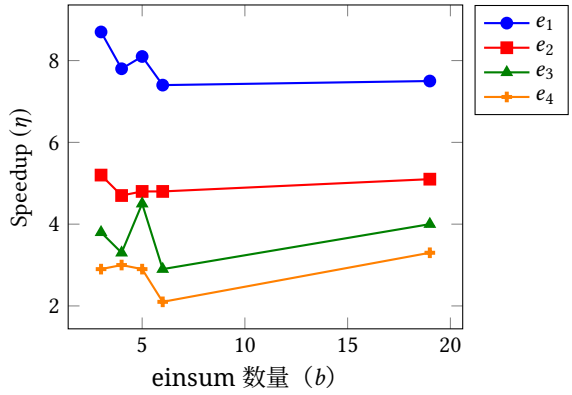
8.2 TCCG：未物化操作数基准测试套件

Springer 等人 [49] 编译了一系列 48 个张量收缩，这些收缩见于多种应用场景，例如电子结构计算、耦合簇理论和量子化学。为了模拟真实场景，我们对基准测试套件稍作扩展：不再将张量收缩的操作数视为已物化的操作数，而是将其视为内存中数组的线性多项式。具体而言，我们不再评估使用数组 A 和 B进行张量收缩的性能，而是评估使用操作数 $(\alpha_1 A + \beta_1)$ 和 $(\alpha_2 B + \beta_2)$进行张量收缩的性能，其中 α_1、α_2、β_1、β_2 是只有在运行时才知晓的标量。在所有基准测试中，α_1、α_2、β_1和 β_2 均为双精度标量，而 A 和 B 则是元素为双精度数的多维数组。

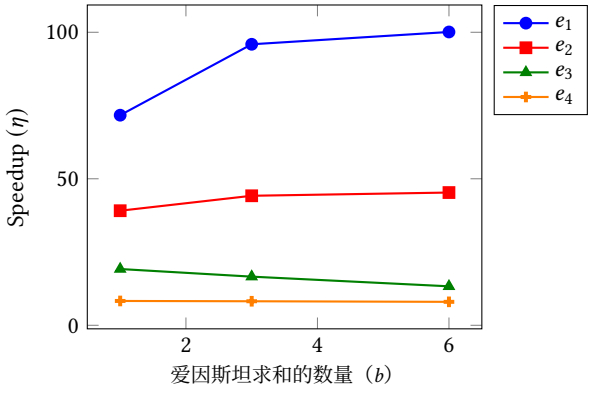
对于测试套件中的每个测试用例，我们将浮点运算吞吐量报告为

$$\frac{\text{来自 Opt-Einsum 的收缩中的每秒浮点运算次数} + \text{用于计算 } \alpha_1 A + \beta_1 \text{ 和 } \alpha_2 B + \beta_2 \text{ 的每秒浮点运算次数}}{t_{\text{wall}}},$$

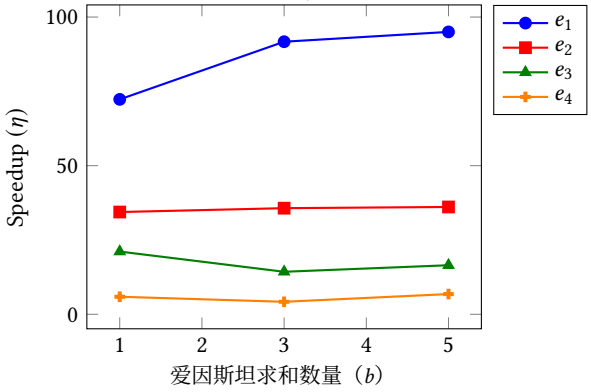
其中 t_{wall} 是在内核中花费的平均挂钟时间。为了获得 t_{wall}，我们执行 5 轮预热，并使用接下来 N 轮执行的挂钟时间的算术平均值，其中 N 是大于 20 的最小数，使得在内核内花费的累计挂钟时间至少为 2 秒。我们在 Nvidia Titan V GPU 上执行这些测试，运行 CUDA 驱动程序版本 550.163。变换后的内核使用 CUDA 工具包版本 12.4 中提供的 Nvidia OpenCL 内核编译器进行编译。



（a）面质量应用的实验结果。e_k 对应于使用 P^k 函数空间时，在 DG-FEM 求解器中观察到的面质量算子中的批量 einsum 表达式。对于 e_1，索引到轴长度的映射为 $\{i \mapsto 4;\ \ e \mapsto 10^5;\ \ f \mapsto 4;\ \ j \mapsto 3\}$。对于 e_2，索引到轴长度的映射为 $\{i \mapsto 10;\ \ e \mapsto 10^5;\ \ f \mapsto 4;\ j \mapsto 6\}$。对于 e_3，索引到轴长度的映射为 $\{i \mapsto 20;\ e \mapsto 10^5;\ f \mapsto 4;\ \ j \mapsto 10\}$。对于 e_4，索引到轴长度的映射为 $\{i \mapsto 35;\ \ e \mapsto 10^5;\ \ f \mapsto 4;\ \ j \mapsto 15\}$。



（b）局部散度计算的实验结果。e_k 对应于采用 P^k 函数空间时，在 DG-FEM 求解器中观察到的局部散度算子中的批量 einsum 表达式。对于 e_1，索引到轴长度的映射为 $\{i \mapsto 4;\ \ j \mapsto 4;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ r \mapsto 3\}$。对于 e_2，索引到轴长度的映射为 $\{i \mapsto 10;\ j \mapsto 10;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ \ r \mapsto 3\}$。对于 e_3，索引到轴长度的映射为 $\{i \mapsto 20;\ \ j \mapsto 20;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ r \mapsto 3\}$。对于 e_4，索引到轴长度的映射为 $\{i \mapsto 35;\ j \mapsto 35;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ \ r \mapsto 3\}$。



（c）局部梯度计算的实验结果。e_k 对应于采用 P^k 函数空间时，在 DG-FEM 求解器中观察到的局部散度算子里的批量 einsum 表达式。对于 e_1，索引到轴长度的映射为 $\{i \mapsto 4;\ \ j \mapsto 4;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ r \mapsto 3\}$。对于 e_2，索引到轴长度的映射为 $\{i \mapsto 10;\ j \mapsto 10;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ \ r \mapsto 3\}$。对于 e_3，索引到轴长度的映射为 $\{i \mapsto 20;\ \ j \mapsto 20;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ r \mapsto 3\}$。对于 e_4，索引到轴长度的映射为 $\{i \mapsto 35;\ j \mapsto 35;\ \ e \mapsto 10^5;\ \ x \mapsto 3;\ \ r \mapsto 3\}$。

图4. 使用 Feinsum 执行批量 einsum 相较于使用 JAX 实现等效操作所观察到的加速比。

我们还通过一个保守模型报告屋顶线性能 [56] 该模型假设图形处理器是一种能够隐藏延迟的机器，其性能可能受执行单元或存储单元的限制。因此，我们得到：

$$\text{Roofline FLOPS} = \min(\text{Peak FLOPS}, \frac{\text{Total FLOPS}}{\text{\# Footprint Bytes}} \cdot \text{Peak Bandwidth}).$$

我们使用制造商报告的峰值每秒浮点运算次数和带宽值。

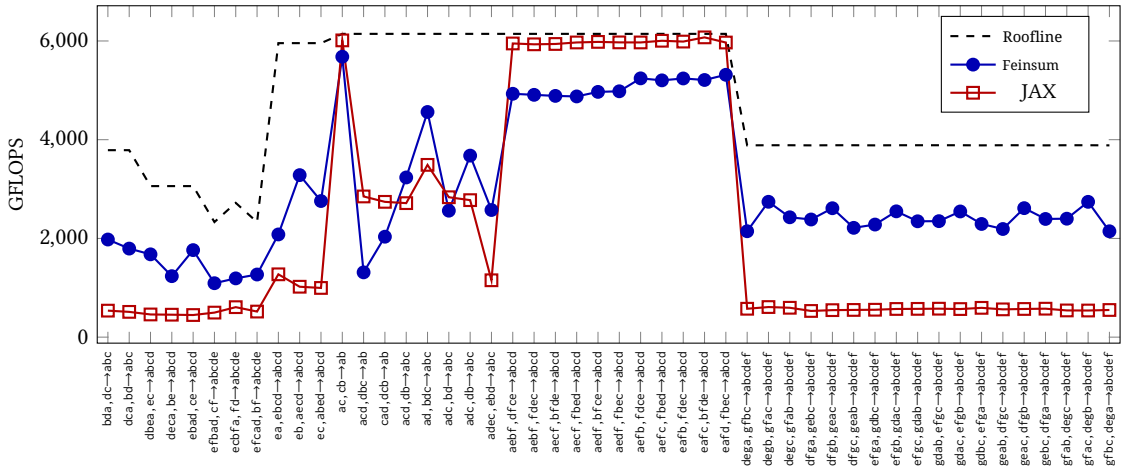


图 5. FP64 浮点运算吞吐量 基于 TCCG 套件的基准测试吞吐量，如 第 8.2 节所述。

我们在图 5 中展示了这些实验的数据。可以看出，我们的变换数据库包含能够实现 2.1× 相较于 JAX 的几何平均加速比（针对 TCCG 套件）。我们将这一加速归因于两个主要原因。首先，JAX 使用了与 TTGT 类似的变换策略 [20]，并依赖cuBLAS提供的矩阵乘法运算，而我们的变换数据库包含基于 TTGT 和 C OGENT [23]变换空间的条目。此前已有研究表明，从几何平均意义上看，COGENT 优于 TTGT。其次，我们的识别并变换策略还会将其与操作数周围计算的融合结合起来。对于受内存带宽限制的测试用例，实际浮点运算的成本不应体现在实际运行时间中。这一点可以从图 3.6× 中受内存限制用例里，我们的工作相较于 JAX 的相对加速比 3。

这项实验验证了我们的观点：广泛使用的数组编译器依赖 BLAS 原语，显著限制了它们在实际程序中处理张量收缩表达式的性能。与此同时，我们为理想化批量爱因斯坦求和整理优化变换的工作，表明在整个基准测试套件中，成本均大幅降低。

8.3 将批量 einsum 规范化所需的时间

我们还旨在分析与变换相关的成本，这些成本未计入内核执行时间。尽管这些开销是在编译一个可执行文件会被重复使用多次的较大程序时产生的，但较高的编译时间开销可能会影响所提出系统的实用性。在我们的方法中，这些开销包括批量 einsum 规范化过程，以及用于匹配 einsum 的 AST 遍历。在这些操作中，匹配表达式涉及成本较低的 AST 遍历。然而，批量 einsum 规范化依赖于图规范化，其复杂度相对于已提交至 ACM 的手稿中的

图中节点数量呈亚指数级。这促使我们开展另一系列研究，以量化规范化过程中产生的开销。

在图 6中，我们展示了规范化 TCCG 基准测试中每个爱因斯坦求和实例所需的墙钟时间。这些实验是在 AMD Ryzen 7 5800HS 处理器上进行的。在整个基准测试套件中，我们一致观察到，规范化过程耗时不到 1 毫秒。

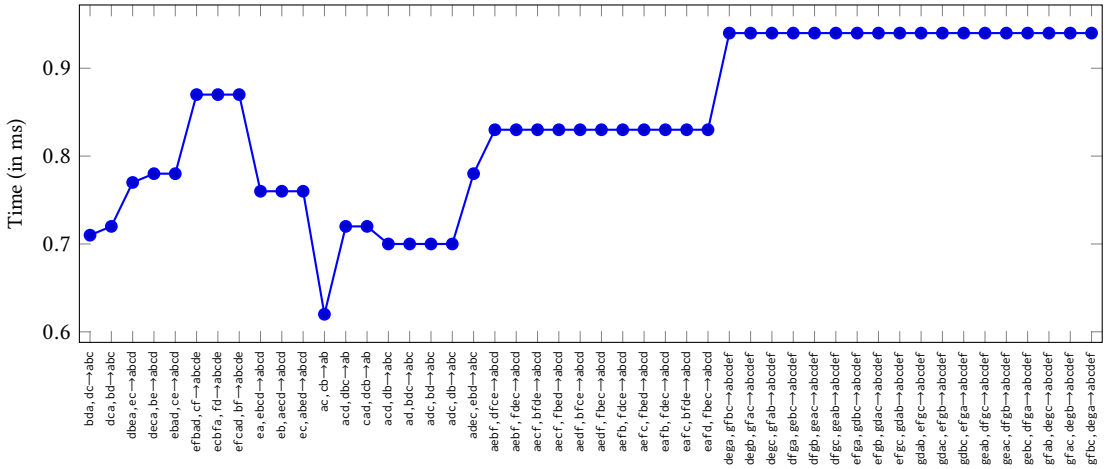


图 6. TCCG 基准测试套件的规范化成本。

8.4 代码可用性

为确保结果可复现，我们提供了本文所呈现结果所用软件归档版本的数字对象标识符。包含第 6 节中的规范化算法以及第 7 节中抽象实现的软件工具包已作为Python 库提供，并已归档至 Zenodo [32]，对应于本文研究的版本。同样，第8 节中用于获取实证数据的软件也已归档至 Zenodo [33]。归档记录中包含详细的安装说明，详见 README 文件。归档中的所有内容均根据 MIT 许可证发布。

9 结论

在这项工作中，我们提出了三项贡献。首先，我们扩展了 einsum 计算原语，使其支持批次。批处理是为稠密线性代数程序合成高效代码时一个重要却常被忽视的方面。我们通过一系列真实应用场景中的实验来支持这一观点。其次，我们将批量 einsum 的概念推广为一种接受函数作为操作数的形式，从而能够针对其理想化版本中已知的优化实施算子融合。针对经过修改的 TCCG 基准测试套件所开展的案例研究，凸显了此类技术对于高效处理内存受限的批量 einsum 的必要性。最后，我们实现了一个开源软件系统，利用上述抽象为批量 einsum 表达式这一类别维护经过优化的代码变换数据库。

这项工作引出了未来研究中的一些有趣问题：

是否存在具有多项式复杂度的 einsum 规范化算法? 在这项工作中，我们使用图规范化作为爱因斯坦求和规范化的中间步骤。然而，目前尚无最坏情况下渐近复杂度具有多项式上界的图规范化算法。一种可能的方法是采用 Babai等人提出的 [8] 图规范化算法，从而实现具有准多项式复杂度的算法。不过，是否存在一种不依赖图规范化作为中间步骤的 einsum 规范化算法，仍是一个未解的问题。

einsum表达式性能的严格上界是什么? 目前尚无专门针对批量 einsum 的屋顶线模型。类 einsum 表达式的性能取决于设备的存储层次结构，因为同一内存位置会被多次访问。因此，要开发一个能够涵盖所有可能瓶颈的模型颇具挑战，其中包括 L1 缓存和本地内存。虽然基于全局内存带宽建立模型相对简单，但它无法反映这一场景下所有影响性能的因素。此类模型或许有助于变换写入器，让其了解调优过程何时会出现收益递减。

致谢

作者的研究获得了美国国家科学基金会的资助，资助项目编号为 SHF-1911019 和 OAC-1931577；还获得了美国能源部的资助，资助编号为 DE-NA0003963；此外，伊利诺伊大学厄巴纳-香槟分校 Siebel计算与数据科学学院也为本研究提供了支持。本文所表达的任何观点、发现、结论或建议均仅代表作者，不一定反映资助方的观点；资助方并未批准或认可本文内容。

参考文献

[1] Daniel G. a. Smith 和 Johnnie Gray。2018。opt_einsum——用于优化类似 einsum 表达式收缩顺序的 Python 软件包。《开放源代码软件期刊》3，26（2018），753。doi:10.21105/joss.00753[2] Martín Abadi、Ashish Agarwal、Paul Barham、Eugene Brevdo、Zhifeng Chen、Craig Citro、Greg S Corrado、Andy Davis、Jeffrey Dean、Matthieu Devin 等。2015。TensorFlow：异构系统上的大规模机器学习。[3] Ahmad Abdelfattah、Azzam Haidar、Stanimire Tomov 和 Jack Dongarra。2016。图形处理器上批量 GEMM 的性能、设计与自动调优。载于《高性能计算》。施普林格国际出版，Cham，21–38。doi:10.1007/978-3-319-41321-1_2[4] Krister Åhlander。2002。多维数组的爱因斯坦求和。《计算机与应用数学》44，8-9（2002），1007–1017。[5] Krister Åhlander 和 Kurt Otto。2002。关于基于指标记法的有限差分格式的软件支持。载于《国际计算科学会议》。施普林格，711–718。[6] Edward Anderson、Zhaojun Bai、Christian Bischof、L Susan Blackford、James Demmel、Jack Dongarra、Jeremy Du Croz、Anne Greenbaum、Sven Hammarling、Alan McKenney 等。1999。《LAPACK 用户指南》。工业与应用数学学会。[7] Jason Ansel、Shoaib Kamil、Kalyan Veeramachaneni、Jonathan Ragan-Kelley、Jeffrey Bosboom、Una-May O'Reilly 和 Saman Amarasinghe。2014。OpenTuner：用于程序自动调优的可扩展框架。载于《第 23 届国际并行体系结构与编译会议论文集》（加拿大艾德蒙顿）（PACT '14）。计算机协会，纽约，纽约州，美国，303–316。doi:10.1145/2628071.2628092[8] László Babai。2019。准多项式时间内的图的规范形式：初步报告。载于《第 51 届 ACM SIGACT计算理论研讨会论文集》。1237–1246。[9] Satish Balay、Kris Buschelman、William D Gropp、Dinesh Kaushik、Matthew G Knepley、L Curfman McInnes、Barry F Smith 和 Hong Zhang。2001。PETSc。见 http://www. mcs. anl. gov/petsc（2001）。[10] Henrik Barthels、Christos Psarras 和 Paolo Bientinesi。2021。Linnea：高效线性代数程序的自动生成。《ACM 数学软件汇刊》47，3，文章 22（2021 年 6 月），26 页。doi:10.1145/3446632[11] Lorenzo Chelini、Andi Drebes、Oleksandr Zinenko、Albert Cohen、Nicolas Vasilache、Tobias Grosser 和 Henk Corporaal。2021。多级中间表示中的渐进式提升。载于《2021 IEEE/ACM国际代码生成与优化研讨会》（CGO）。15–26。doi:10.1109/CGO51591.2021.9370332[12] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。2018。TVM：用于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）。USENIX 协会，美国加利福尼亚州卡尔斯巴德，578–594。https://www.usenix.org/conference/osdi18/presentation/chen

[13] Lam Chi-Chung、P. Sadayappan 和 Rephael Wenger。1997。通过归约实现并行执行时一类多维循环的优化。《并行处理快报》07，02（1997），157–168。doi:10.1142/S0129626497000176 arXiv:https://doi.org/10.1142/S0129626497000176[14] Jack Dongarra、Sven Hammarling、Nicholas J. Higham、Samuel D. Relton 和 Mawussi Zounon。2017。面向现代体系结构的优化批量线性代数。载于《Euro-Par 2017：并行处理》。施普林格国际出版，Cham，511–522。doi:10.1007/978-3-319-64203-1_37[15] Albert Einstein。1922。广义相对论的基础。第 49 卷。JA Barth。[16] Roy Frostig、Matthew James Johnson 和 Chris Leary。2018。通过高级追踪编译机器学习程序。《机器学习系统》4，9（2018）。[17] 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。使用 NumPy 进行数组编程。《自然》585，7825（2020），357–362。[18] Alexander Heinecke、Greg Henry、Maxwell Hutchinson 和 Hans Pabst。2016。LIBXSMM：通过运行时代码生成加速小型矩阵乘法。载于《SC '16：国际高性能计算、网络、存储与分析会议论文集》。981–991。doi:10.1109/SC.2016.83[19] Jan Hesthaven 和 Tim Warburton。2007。《节点间断伽辽金方法》（第 1 版）。施普林格，纽约，纽约州，美国。doi:10.1007/978-0-387-72067-8[20] So Hirata。2003。张量收缩引擎：组态相互作用、耦合簇和多体微扰理论的抽象与自动并行实现。《物理化学杂志 A》107，46（2003），9887–9897。[21] Tommi Junttila 和 Petteri Kaski。2011。用于规范标记的冲突传播与分量递归。载于《（计算机）系统中的算法理论与实践》，Alberto Marchetti-Spaccamela 和 Michael Segal（编）。施普林格柏林海德堡，柏林，海德堡，151–162。doi:10.1007/978-3-642-19754-3_16[22] Jinsung Kim、Aravind Sukumaran-Rajam、Changwan Hong、Ajay Panyala、Rohit Kumar Srivastava、Sriram Krishnamoorthy 和 P. Sadayappan。2018。优化 CCSD(T) 中的张量收缩以实现图形处理器上的高效执行。载于《2018 年国际超级计算会议论文集》（中国北京）（ICS '18）。计算机协会，纽约，纽约州，美国，96–106。doi:10.1145/3205289.3205296[23] Jinsung Kim、Aravind Sukumaran-Rajam、Vineeth Thumma、Sriram Krishnamoorthy、Ajay Panyala、Louis-Noël Pouchet、Atanas Rountev 和 Ponnuswamy Sadayappan。2019。面向图形处理器上高性能张量收缩的代码生成器。载于《2019 IEEE/ACM国际代码生成与优化研讨会》（CGO）。IEEE，85–95。[24] Robert C Kirby。2006。使用三级 BLAS 优化 FIAT。《ACM 数学软件汇刊》（TOMS）32，2（2006），223–235。[25] Fredrik Kjolstad、Shoaib Kamil、Stephen Chou、David Lugato 和 Saman Amarasinghe。2017。张量代数编译器。《ACM 程序设计语言会议论文集》1，OOPSLA，文章 77（2017 年 10 月），29 页。doi:10.1145/3133901[26] Julien Klaus、Mark Blacher 和 Joachim Giesen。2023。将张量表达式编译为爱因斯坦求和。载于《计算科学——ICCS 2023》，Jiří Mikyška、Clélia de Mulatier、Maciej Paszynski、Valeria V. Krzhizhanovskaya、Jack J. Dongarra 和 Peter M.A. Sloot（编）。施普林格自然瑞士，Cham，129–136。[27] Andreas Klöckner。2014。Loo. py：面向图形处理器和 CPU 的基于变换的代码生成。载于《ACM SIGPLAN 数组编程库、语言与编译器国际研讨会论文集》。82–87。[28] Andreas Klöckner。2015。Loo.Py：通过变换与替换规则从 Fortran 迈向高性能。载于《第 2 届 ACM SIGPLAN 数组编程库、语言与编译器国际研讨会论文集》（美国俄勒冈州波特兰）（ARRAY 2015）。计算机协会，纽约，纽约州，美国，1–6。doi:10.1145/2774959.2774969[29] Andreas Klöckner、Lucas C. Wilcox 和 T. Warburton。2016。通过示例了解使用 Loo.Py 进行数组程序变换：高阶有限元。载于《第 3 届 ACM SIGPLAN 数组编程库、语言与编译器国际研讨会论文集》（美国加利福尼亚州圣巴巴拉）（ARRAY 2016）。计算机协会，纽约，纽约州，美国，9–16。doi:10.1145/2935323.2935325[30] A. Klöckner、T. Warburton、J. Bridge 和 J.S. Hesthaven。2009。图形处理器上的节点间断伽辽金方法。《计算物理学杂志》228，21（2009），7863–7882。doi:10.1016/j.jcp.2009.06.041[31] Martin Kronbichler 和 Katharina Kormann。2019。间断伽辽金有限元算子的快速无矩阵求值。《ACM 数学软件汇刊》45，3，文章 29（2019 年 8 月），40 页。doi:10.1145/3325864[32] Kaushik Kulkarni。2026。Feinsum。doi:10.5281/zenodo.18285088[33] Kaushik Kulkarni。2026。Feinsum 评估制品。doi:10.5281/zenodo.18285105[34] Kaushik G Kulkarni。2023。基于多面体模型的面向科学计算的领域专用代码变换。博士学位论文。伊利诺伊大学厄巴纳-香槟分校。[35] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。2021。MLIR：扩展面向领域专用计算的编译器基础设施。载于《2021 IEEE/ACM国际代码生成与优化研讨会》（CGO）。2–14。doi:10.1109/CGO51591.2021.9370308[36] Chuck L Lawson、Richard J. Hanson、David R Kincaid 和 Fred T. Krogh。1979。用于 Fortran 的基本线性代数子程序。《ACM 数学软件汇刊》（TOMS）5，3（1979），308–323。[37] Jianhui Li、Zhennan Qin、Yijie Mei、Jingze Cui、Yunfei Song、Ciyong Chen、Yifei Zhang、Longsheng Du、Xianhang Cheng、Baihui Jin、Yan Zhang、Jason Ye、Eric Lin 和 Dan Lavery。2024。oneDNN 图编译器：面向高性能深度学习编译的混合方法。载于《2024 IEEE/ACM国际代码生成与优化研讨会》（CGO）。460–470。doi:10.1109/CGO57630.2024.10444871[38] Devin A Matthews。2018。无需转置的高性能张量收缩。《SIAM 科学计算杂志》40，1（2018），C1–C24。[39] Brendan D McKay 等。1981。实用图同构。（1981）。

[40] Brendan D McKay 和 Adolfo Piperno。2014。实用图同构 II。《符号计算杂志》60（2014），94–112。[41] Alexander Novikov、Dmitrii Podoprikhin、Anton Osokin 和 Dmitry P Vetrov。2015。张量化神经网络。载于《神经信息处理系统进展》，C. Cortes、N. Lawrence、D. Lee、M. Sugiyama 和 R. Garnett（编），第 28 卷。Curran Associates, Inc. doi:10.48550/arXiv.1509.06569[42] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Kopf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。2019。PyTorch：一种命令式风格的高性能深度学习库。载于《神经信息处理系统进展》，H. Wallach、H. Larochelle、A. Beygelzimer、F. d'Alché-Buc、E. Fox 和 R. Garnett（编），第 32 卷。Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf[43] Robert N. C. Pfeifer、Jutho Haegeman 和 Frank Verstraete。2014。更快地识别张量网络的最优收缩序列。《物理学评论 E》90（2014 年 9 月），033315。第 3 期。doi:10.1103/PhysRevE.90.033315[44] Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman Amarasinghe。2013。Halide：一种用于优化图像处理流水线中并行性、局部性和重新计算的语言与编译器。载于第 34 届 ACM SIGPLAN 编程语言设计与实现大会论文集（美国华盛顿州西雅图）（PLDI '13）。计算机协会，美国纽约州纽约，519–530。doi:10.1145/2491956.2462176[45] Tim Rocktäschel。2018。你所需的一切都是 einsum——深度学习中的爱因斯坦求和。[46] Alex Rogozhnikov。2022。Einops：使用类爱因斯坦记号进行清晰可靠的张量操作。载于《国际学习表征会议》。https://openreview.net/forum?id=oapKSVM2bcj[47] Edgar Solomonik、Devin Matthews、Jeff Hammond 和 James Demmel。2013。Cyclops 张量框架：减少通信并消除大规模并行收缩中的负载不均衡。载于 2013 年 IEEE 第 27 届并行与分布式处理国际研讨会。IEEE，813–824。[48] Edgar Solomonik、Devin Matthews、Jeff R Hammond、John F Stanton 和 James Demmel。2014。用于耦合簇计算的大规模并行张量收缩框架。《并行与分布式计算杂志》74，12（2014），3176–3190。[49]Paul Springer 和 Paolo Bientinesi。2016。高性能类 GEMM 张量-张量乘法的设计。CoRR（2016）。arXiv:1607.00145 [quant-ph] http://arxiv.org/abs/1607.00145[50] Paul Springer 和 Paolo Bientinesi。2018。高性能类 GEMM 张量–张量乘法的设计。《ACM 数学软件汇刊》44，3，文章 28（2018 年 1 月），29 页。doi:10.1145/3157733[51] Philippe Tillet、H. T. Kung 和 David Cox。2019。Triton：一种用于分块神经网络计算的中间语言和编译器。载于第 3 届 ACM SIGPLAN 机器学习与编程语言国际研讨会论文集（美国亚利桑那州凤凰城）（MAPL 2019）。计算机协会，美国纽约州纽约，10–19。doi:10.1145/3315508.3329973[52] M. Valiev、E.J. Bylaska、N. Govind、K. Kowalski、T.P. Straatsma、H.J.J. Van Dam、D. Wang、J. Nieplocha、E. Apra、T.L. Windus 和 W.A. de Jong。2010。NWChem：用于大规模分子模拟的全面且可扩展的开源解。《计算物理学通信》181，9（2010），1477–1489。doi:10.1016/j.cpc.2010.04.018[53] Field G. Van Zee 和 Robert A. van de Geijn。2015。BLIS：快速实例化 BLAS 功能的框架。《ACM 数学软件汇刊》41，3，文章 14（2015 年 6 月），33 页。doi:10.1145/2764454[54] Nicolas Vasilache、Oleksandr Zinenko、Theodoros Theodoridis、Priya Goyal、Zachary DeVito、William S. Moses、Sven Verdoolaege、Andrew Adams 和 Albert Cohen。2018。张量推导：与框架无关的高性能机器学习抽象。arXiv 电子预印本，文章 arXiv:1802.04730（2018 年 2 月），arXiv:1802.04730 页。doi:10.48550/arXiv.1802.04730 arXiv:1802.04730 [cs.PL][55] Endong Wang、Qing Zhang、Bo Shen、Guangyong Zhang、Xiaowei Lu、Qing Wu 和 Yajuan Wang。2014。英特尔数学核心函数库。施普林格国际出版，Cham，167–188。doi:10.1007/978-3-319-06486-4_7[56] Samuel Williams、Andrew Waterman 和 David Patterson。2009。Roofline：一种富有洞察力的多核架构可视化性能模型。《ACM 通讯》52，4（2009 年 4 月），65–76。doi:10.1145/1498765.1498785[57] Zhang Xianyi、Wang Qian 和 Zhang Yunquan。2012。龙芯 3A 处理器上三级 BLAS 性能的模型驱动优化。载于 2012 年 IEEE 第 18 届并行与分布式系统国际会议。684–691。doi:10.1109/ICPADS.2012.97[58] Alexandros Nikolaos Ziogas、Grzegorz Kwasniewski、Tal Ben-Nun、Timo Schneider 和 Torsten Hoefler。2022。Deinsum：实用的 I/O 最优多重线性代数。arXiv 预印本 arXiv:2206.08301（2022）。

A Loopy 中间表示概览

Loopy 中间表示（中间表示），由 Klöckner 等人提出。 [27], 它是一种用于建模和转换基于数组的程序的中级中间表示。在Loopy中间表示中，关键数据结构是内核，它由多个语句组成，这些语句在多面体定义的迭代域内执行。这些语句对具有特定内存布局、数据类型和地址空间的数组进行操作。此外，Loopy 内核编码了语句之间的依赖图，从而对依赖关系进行了部分规范。我们在清单 6 中展示了一个Loopy 内核示例。

```
参数:
A: 数据类型: 64位浮点数, 形状: (n, n)
n: 值参数, 数据类型: int32
x: 数据类型: 64位浮点数, 形状: (n)
y1: 数据类型: 64位浮点数, 形状: (n)
y2: 数据类型: 64位浮点数, 形状: (n)
---------------------------------
定义域:
[n] -> { [i, j] : 0 <= i,j < n }
---------------------------------
替换规则:
u1(i) := 2*cos(x[i]) + 3
u2(i) := 3*正弦函数(x[i]) + 5
---------------------------------
指令:
for i
  y1[i] = 求和([j], A[i, j]*u1(j))
  y2[i] = 求和([j], A[i, j]*u2(j))
结束 i
```

列表 6。计算数组 y1 和 y2 的 Loopy 内核。

此外，Loopy IR 提供了一种称为替换规则的节点。这些规则在代码清单的第 12–13 行声明，并在第 17–18 行调用。这些替换规则 [28]会在代码生成过程中内联；不过，它们也可用于实现有针对性的变换，例如，将对替换规则的访问预取到局部/私有临时变量中。Klöckner 等人 [29] 在高阶有限元方法的背景下展示了这一点。

在第 7 节中，我们提出了一种规范，用于存储执行 einsum 操作的程序的变换知识。在此规范中，我们使用 Loopy ，既将其作为描述性语法，也将其作为指定变换的语言。值得注意的是，本文的贡献独立于 Loopy；它仅作为中间表示和变换描述语言，用于配套实现。

B 诱导图

本节中，我们给出从批量 einsum 到彩色图的映射，以及从彩色图到批量 einsum 的映射。在第 6 节中，我们看到，这些映射是批量规范化算法的基础。在描述与批量 einsum 相对应的图构造之前，我们先给出以下术语的定义，这些定义将有助于后续过程。

Definition 19 （爱因斯坦求和中的所有维度）。 我们定义批量 einsum 中所有维度的集合 e as

$$\mathrm{AllDims}(e) = \{\mathrm{Dim}(a) : \ a \in \mathcal{A}^{\mathrm{all}}(e)\} \cup \{|\mathcal{I}^{\mathrm{out}}(e)|\}.$$

定义 n 20（爱因斯坦求和中的输入访问）。 我们定义批量 einsum 的所有输入访问集合 e a s

$$\text{InputAccesses}(e) = \bigcup_{j=1}^{n} \Big\{ \quad (i, j, \mathcal{I}_d^{\text{in},j}, d) : \\ d \in \left\{1, \ldots, |\mathcal{I}^{\text{in},j}|\right\}, i \in \{1, 2, \ldots, b\}, j \in \{1, 2, \ldots, n\} \Big\} \tag{3}$$

定义 21（爱因斯坦求和中的输出访问）。 我们将批量 einsum e 中的所有输出访问构成的集合定义为

$$\text{OutputAccesses}(e) = \left\{ \left(\mathcal{I}_d^{\text{out}}, d\right) : d \in \{1, \ldots, |\mathcal{I}^{\text{out}}|\} \right\}.$$

定义 22（批量 einsum 中的所有数据类型）。 批量 einsum e中输入的数据类型集合为

$$\text{Dtypes}(e) = \{\text{Dtype}(a) : a \in \mathcal{A}^{\text{all}}(e)\}.$$

定 义 23（批量 einsum 中的轴长度）。 批量 einsum e中的轴长度集合为

$$\text{AxisLengths}(e) = \left\{ (\text{Shape}(a))_i : i \in \{1, \ldots, \text{Dim}(a)\}, a \in \mathcal{A}^{\text{all}}(e) \right\}.$$

定义24（诱导图）。 由批量 einsum $\left(N_{\text{IDG}}、\iota_{\text{dtype}}、\ \iota_{\text{lengths}}、\ A^{N^{\text{IDG}} \times N_{\text{IDG}}}\right.$ 和 $\left.c_{N^{\text{IDG}}}\right)$ 诱导的图，其中：

- N_{IDG} 是图中的节点数
- A 是图的邻接矩阵。
- c 是图中节点的着色向量。
- $\iota_{\text{数据类型}}$ 是从 {1、2、.....、 N_{IDG}} 的子集到数值数据类型的映射。
- $\iota_{\text{长度}}$ 是从 {1、2、.....、 N_{IDG}} 的子集到非负整数的映射。

B.1 将批量爱因斯坦求和映射到诱导图

接下来，我们概述为由元组 $\left(b, n, \mathrm{I}^{\text{out}}, (\mathrm{I}^{\text{in},\ k})_{k=1}^{n}, \left((\mathrm{A}^{\ j,\ k})_{k=1}^{n}\right)_{j=1}^{b}\right)$ 定义的批量爱因斯坦求和 e 构造诱导图的构造过程。我们将使用以下记号来简化构造过程：

- $N_{\text{arg}} = |\mathrm{A}_{\text{all}}(e)|$
- $N_{\text{索引}} = |\mathrm{I}_{\text{all}}(e)|$
- $N_{\text{长度}} = |\text{AxisLengths}(e)|$
- $N_{\text{维数}} = \max(\text{Dims}(e))$
- $N_{\text{数据类型}} = |\text{维数}(e)|$
- $N_{\text{输入访问集合，输入}} = |\text{InputAccesses}(e)|$
- $N_{\text{输出访问集合，输出}} = |\text{OutputAccesses}(e)|$

在此构造中，我们为批量爱因斯坦求和的各个组成部分分配节点，具体如下：每个参数一个节点、每个索引一个节点、数组中每种唯一数值数据类型一个节点、每种唯一轴长度一个节点、每种索引访问一个节点、每个维度索引一个节点，以及每个输出一个节点。因此，我们得到图中的节点数如下：

$$N^{\text{IDG}} = \left(N_{\text{arg}} + N_{\text{index}} + N_{\text{dtype}} + N_{\text{length}} + N_{\text{dim}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n\right) \tag{4}$$

现在，我们定义批量爱因斯坦求和的不同组成部分到图中节点标签的映射。

- 为输入访问选择一种编号方式，用双射 $\iota_{\text{access,in}} : \{N_{\text{arg}} + N_{\text{index}} + 1, \ldots, N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}}\} \mapsto$ 输入访问集合(e) 表示。- 为输出访问选择一种编号方式，用双射 $\iota_{\text{access,out}} : \{N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + 1, \ldots, N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}}\} \mapsto$ 输出访问集合(e) 表示。- 为输出选择一种编号方式，用双射 $\iota_{\text{output}} : \{N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + 1, \ldots, N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b\} \mapsto \{1, 2, \ldots, b\}$ 表示。- 为操作数位置选择一种编号方式，用双射 $\iota_{\text{arg-pos}} : \{N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + 1, \ldots, N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n\} \mapsto \{1, 2, \ldots, n\}$ 表示。- 为数据类型选择一种编号方式，用双射

$$\begin{aligned}\iota_{\text{dtype}} : \{ \quad & N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n + 1, \\ & \ldots, \\ & N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n + N_{\text{dtype}}\} \mapsto \text{Dtypes}(e).\end{aligned} \tag{5}$$

- 为轴长度选择一种编号方式，记为双射

$$\begin{aligned}\iota_{\text{length}} : \{ & \\ & N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n + N_{\text{dtype}} + 1, \\ & \ldots, \\ & N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n + N_{\text{dtype}} + N_{\text{length}}\} \mapsto \text{AxisLengths}(e).\end{aligned} \tag{6}$$

- 为数组的维度选择一种编号方式，记为双射，

$$\begin{aligned}\iota_{\text{dim}} = \{ \quad & \left(N_{\text{arg}} + N_{\text{index}} + N_{\text{access,in}} + N_{\text{access,out}} + b + n + N_{\text{dtype}} + N_{\text{length}} + i\right) \mapsto i : \\ & i \in \{1, 2, \ldots, N_{\text{dim}}\}\}\end{aligned} \tag{7}$$

接下来，我们定义一些特定集合，这些集合构成图中的边，用于表示 e 中各个组件之间的相互作用。

– 我们定义一组边，用于描述输入数组对输出的索引贡献。将其记为 $E_{\text{access, in-to-arg}}$。

$$\begin{aligned}E_{\text{access,in-to-arg}} = \Big\{ \quad & \left(\iota^{-1}_{\text{access,in}}\left(\left(\left(i, j, \mathcal{I}^{\text{in},j}\right)_d, d\right)\right), \iota^{-1}_{\text{arg}}\left(\mathcal{A}^{i,j}\right)\right) : \\ & d \in \{1, \ldots, \text{Dim}(\mathcal{A}^{i,j})\}, j \in \{1, \ldots, n\}, i \in \{1, \ldots, b\}\Big\}\end{aligned}$$

– 现在，我们定义用于描述访问节点的边，即该节点的索引、所索引的维度以及它所贡献的输出。我们将它们记为$E_{\text{arg-pos-to-access, in}}$、 $E_{\text{output-to-access, in}}$、 $E_{\text{index-to-access, in}}$和$E_{\text{dim-to-access, in}}$。

$$E_{\text{arg-pos-to-access,in}} = \{(\iota^{-1}_{\text{arg-pos}}(a), \iota^{-1}_{\text{access,in}}((j, a, i, k))) : (j, a, i, k) \in \text{InputAccesses}(e)\} \tag{8}$$

$$E_{\text{output-to-access,in}} = \{(\iota^{-1}_{\text{output}}(i), \iota^{-1}_{\text{access,in}}((j, a, i, k))) : (j, a, i, k) \in \text{InputAccesses}(e)\} \tag{9}$$

$$E_{\text{index-to-access,in}} = \{(\iota^{-1}_{\text{index}}(j), \iota^{-1}_{\text{access,in}}((j, a, i, k))) : (j, a, i, k) \in \text{InputAccesses}(e)\} \tag{10}$$

$$E_{\text{dim-to-access,in}} = \{(\iota^{-1}_{\text{dim}}(k), \iota^{-1}_{\text{access,in}}((j, a, i, k))) : (j, a, i, k) \in \text{InputAccesses}(e)\} \tag{11}$$

– 现在，我们定义用于描述输出索引节点的边。我们将它们记为$E_{\text{index−to−access，out}}$和$E_{\text{dim−to−access，out}}$。

$$E_{\text{index−to−access,out}} = \{(\iota^{-1}_{\text{index}}(i), \iota^{-1}_{\text{access,out}}((i,d))) : (i,d) \in \text{OutputAccesses}(e)\} \tag{12}$$

$$E_{\text{dim−to−access,out}} = \{(\iota^{-1}_{\text{dim}}(d), \iota^{-1}_{\text{access,out}}((i,d))) : (i,d) \in \text{OutputAccesses}(e)\} \tag{13}$$

– 我们引 入边以表征与每个索引对应的迭代次数。我们将其记为 E_{length}:

$$E_{\text{length}} = \Big\{ \quad \Big(\iota^{-1}_{\text{length}}\left(\left(\text{Shape}(\mathcal{A}^{i,k})\right)_d\right), \iota^{-1}_{\text{index}}\left((\mathcal{I}^{\text{in},k})_d\right)\Big) : \\ d \in \{1,\ldots,\text{Dim}(\mathcal{A}^{i,k})\}, k \in \{1,\ldots,n\}, i \in \{1,\ldots,b\}\Big\} \tag{14}$$

– 我们引 入边以表征与每个数组对应的数值数据类型。我们将其记为 E_{dtype} .

$$E_{\text{dtype}} = \left\{\left(\iota^{-1}_{\text{dtype}}(\text{Dtype}(a)), \iota^{-1}_{\text{arg}}(a)\right) : \quad a \in \mathcal{A}^{\text{all}}(e)\right\}$$

– 我们引入边以记录轴长度节点之间的差异。我们将其记为 $E_{\text{lengthranks}}$。

$$E_{\text{lengthranks}} = \left\{(\iota^{-1}_{\text{length}}(l_i), \iota^{-1}_{\text{length}}(l_j)) : l_j > l_i, l_i \in \text{AxisLengths}(e), l_j \in \text{AxisLengths}(e)\right\} \tag{15}$$

– 我们引入边来记录数据类型节点之间的差异。我们将其表示为 $E_{\text{dtyperanks}}$。

$$E_{\text{dtyperanks}} = \Big\{ \quad (\iota^{-1}_{\text{dtype}}(t_i), \iota^{-1}_{\text{dtype}}(t_j)) : \\ \text{DtypeRank}(t_j) > \text{DtypeRank}(t_i), \\ t_i \in \text{Dtypes}(e), \\ t_j \in \text{Dtypes}(e)\Big\} \tag{16}$$

其中，DtypeRank 是映射 {int8 ↦ 1; int32 ↦ 2; int64 ↦ 3; float16 ↦ 4; float32 ↦ 5; float64 ↦6; complex64 ↦ 7; complex128 ↦ 8}。

– 我们引入边来记录维度节点之间的差异。我们将其表示为 E_{dimranks}。

$$E_{\text{dimranks}} = \left\{(\iota^{-1}_{\text{dim}}(d_i), \iota^{-1}_{\text{dim}}(d_j)) : d_j > d_i, d_i \in \text{Dims}(e), d_j \in \text{Dims}(e)\right\} \tag{17}$$

基于这些边，我们将邻接矩阵初始化为：

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E_{\text{access,in-to-arg}} \\ 1 & \text{if } (i,j) \in E_{\text{output-to-access,in}} \\ 1 & \text{if } (i,j) \in E_{\text{arg-pos-to-access,in}} \\ 1 & \text{if } (i,j) \in E_{\text{index-to-access,in}} \\ 1 & \text{if } (i,j) \in E_{\text{dim-to-access,in}} \\ 1 & \text{if } (i,j) \in E_{\text{index-to-access,out}} \\ 1 & \text{if } (i,j) \in E_{\text{dim-to-access,out}} \\ 1 & \text{if } (i,j) \in E_{\text{length}} \\ 1 & \text{if } (i,j) \in E_{\text{dtype}} \\ 1 & \text{if } (i,j) \in E_{\text{lengthranks}} \\ 1 & \text{if } (i,j) \in E_{\text{dtyperanks}} \\ 1 & \text{if } (i,j) \in E_{\text{dimranks}} \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

我们根据与节点关联的批量 einsum 实体的类型定义节点着色。

$$c_i = \begin{cases} 1 & \text{if } i \le N_{\text{arg}} \\ 2 & \text{if } 0 < \left(i - N_{\text{arg}}\right) \le N_{\text{index}} \\ 3 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}}\right) \le N_{\text{access,in}} \\ 4 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}} - N_{\text{access,in}}\right) \le N_{\text{access,out}} \\ 5 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}} - N_{\text{access,in}} - N_{\text{access,out}}\right) \le b \\ 6 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}} - N_{\text{access,in}} - N_{\text{access,out}} - b\right) \le n \\ 7 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}} - N_{\text{access,in}} - N_{\text{access,out}} - b - n\right) \le N_{\text{dtype}} \\ 8 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}} - N_{\text{access,in}} - N_{\text{access,out}} - b - n - N_{\text{dtype}}\right) \le N_{\text{length}} \\ 9 & \text{if } 0 < \left(i - N_{\text{arg}} - N_{\text{index}} - N_{\text{access,in}} - N_{\text{access,out}} - b - n - N_{\text{dtype}} - N_{\text{length}}\right) \le N_{\text{dim}} \end{cases} \tag{19}$$

利用上述构造，我们得到与批量 einsum e 关联的诱导图 G。该图由元组 $\left(N^{IDG}、\ \iota_{\text{dtype}}、\ \iota_{\text{lengths}}、\ A、\ c\right)$ 定义，其中各项分别在方程（4）、（5）、（6）、（18）和（19）中定义。

示例 6（诱导图）。对于爱因斯坦求和 $\left(1, 2, (i)、((ik),\ (ij))、((A,\ \ B))\right)$，其中 $\text{Shape}(A) = \text{Shape}(B) = (72,\ 18)$，且 $\text{Dtype}(A) = \text{Dtype}(B) = \text{F64}$，我们得到如图 7 所示的诱导图。值得注意的是，与 j 和 k 对应的节点在图中构成自同构的一部分。

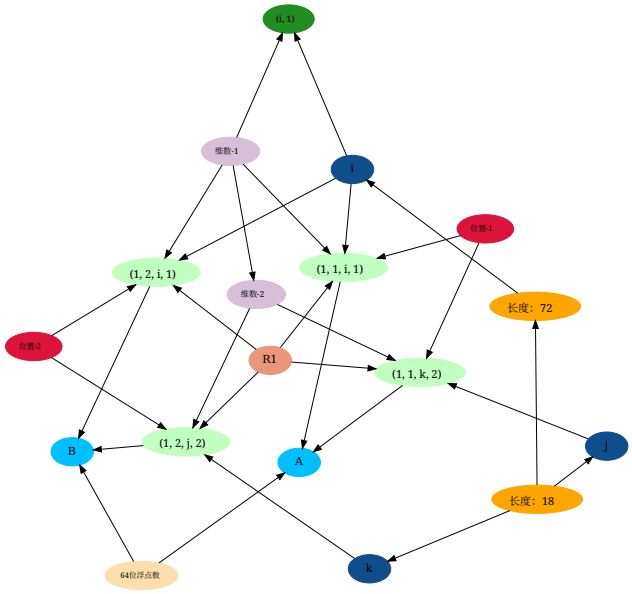


图7. 示例中批量 einsum 的诱导图6。

B.2 从诱导图重构批量爱因斯坦求和

在本节中，我们将 G（一个诱导图（定义24））映射为批量爱因斯坦求和。我们将考虑 G 由元组 $(N_{\text{IDG}}、\iota_{\text{dtype}}、\iota_{\text{lengths}}、A^{N^{\text{IDG}}\times N_{\text{IDG}}}、c_{N^{\text{IDG}}})$ 定义，并构造一个批量爱因斯坦求和。本节的目标是对第B.1节中的构造求逆。

在讨论构造过程之前，必须注意，并非所有诱导图都能映射为有效的批量爱因斯坦求和。一个不合规图的例子是包含颜色值为 12 的节点的图，因为它违反了公式（19）中规定的条件。

现在，我们介绍一些术语，这些术语将对我们提出的构造方法起到重要作用。

– 我们对公式（19）中的着色映射取逆，得到以下集合：

$$\begin{aligned}
V_{\text{arg}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 1\} \\
V_{\text{index}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 2\} \\
V_{\text{access,in}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 3\} \\
V_{\text{access,out}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 4\} \\
V_{\text{output}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 5\} \\
V_{\text{arg-pos}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 6\} \\
V_{\text{dtype}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 7\} \\
V_{\text{length}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 8\} \\
V_{\text{dims}} &= \{i : i \in \{1,\dots,N^{\text{IDG}}\}, c_i = 9\}
\end{aligned} \tag{20}$$

– 我们将图中的邻接关系定义为 G:

$$\begin{aligned} \mathrm{Succs}(i) &= \left\{ j : j \in \{1, \ldots, N^{\mathrm{IDG}}\}, A_{ij} = 1 \right\} \\ \mathrm{Preds}(i) &= \left\{ j : j \in \{1, \ldots, N^{\mathrm{IDG}}\}, A_{ji} = 1 \right\} \end{aligned} \tag{21}$$

要反转第B.1 节中的构造，必须满足以下条件:

(C1) 对所有 $i \in${1、.....、 n}， c_i必须属于 {1、...
...、9}。(C2) 参数节点不得有任何后继节点

$$i \in V_{\mathrm{arg}} \Rightarrow \mathrm{Succs}(i) = \emptyset.$$

(C3) 参数节点的前驱节点必须是输入访问节点或数据类型节点

$$i \in V_{\mathrm{arg}} \Rightarrow \mathrm{Preds}(i) \subseteq (V_{\mathrm{access,in}} \cup V_{\mathrm{dtype}}).$$

(C4) 参数节点必须恰好有一个数据类型节点作为前驱节点

$$n \in V_{\mathrm{arg}} \Rightarrow |\mathrm{Preds}(n) \cap V_{\mathrm{dtype}}| = 1.$$

(C5) 数据类型节点的前驱节点只能是其他数据类型节点

$$i \in V_{\mathrm{dtype}} \Rightarrow \mathrm{Preds}(i) \subset V_{\mathrm{dtype}}.$$

（C6）数据类型节点的后继节点必须是参数节点或数据类型节点

$$i \in V_{\mathrm{dtype}} \Rightarrow \mathrm{Succs}(i) \subseteq V_{\mathrm{arg}} \cup V_{\mathrm{dtype}}.$$

（C7）输入访问节点的后继节点必须是参数节点

$$i \in V_{\mathrm{access,in}} \Rightarrow \mathrm{Succs}(i) \subseteq V_{\mathrm{arg}}.$$

（C8）输入访问节点必须恰好有四个前驱节点，即一个索引节点、一个维度节点、一个参数位置节点和一个输出节点，即

$$\begin{aligned} i \in V_{\mathrm{access,in}} &\Rightarrow |\mathrm{Preds}(i)| = 4 \\ i \in V_{\mathrm{access,in}} &\Rightarrow |\mathrm{Preds}(i) \cap V_{\mathrm{arg-pos}}| = 1 \\ i \in V_{\mathrm{access,in}} &\Rightarrow |\mathrm{Preds}(i) \cap V_{\mathrm{index}}| = 1 \\ i \in V_{\mathrm{access,in}} &\Rightarrow |\mathrm{Preds}(i) \cap V_{\mathrm{output}}| = 1 \\ i \in V_{\mathrm{access,in}} &\Rightarrow |\mathrm{Preds}(i) \cap V_{\mathrm{dim}}| = 1 \end{aligned} \tag{22}$$

（C9）输出节点的后继节点必须是输入访问节点

$$i \in V_{\mathrm{output}} \Rightarrow \mathrm{Succs}(i) \subseteq V_{\mathrm{access,in}}.$$

（C10）输出节点不得有任何前驱节点

$$i \in V_{\mathrm{output}} \Rightarrow \mathrm{Preds}(i) = \emptyset.$$

(C11) 输出访问节点不得有任何后继节点。

$$i \in V_{\mathrm{access,out}} \Rightarrow \mathrm{Succs}(i) = \emptyset$$

(C12) 输出访问节点必须恰好有两个前驱节点：一个索引节点和一个维度节点。

$$\begin{aligned} i \in V_{\text{access,out}} &\Rightarrow |\operatorname{Preds}(i)| = 2 \\ i \in V_{\text{access,out}} &\Rightarrow |\operatorname{Preds}(i) \cap V_{\text{index}}| = 1 \\ i \in V_{\text{access,in}} &\Rightarrow |\operatorname{Preds}(i) \cap V_{\text{dim}}| = 1 \end{aligned} \tag{23}$$

(C13) 索引节点的后继节点必须是输入访问节点或输出访问节点

$$i \in V_{\text{index}} \Rightarrow \operatorname{Succs}(i) \subseteq \left(V_{\text{access,in}} \cup V_{\text{access,out}}\right).$$

(C14) 索引节点必须恰好有一个节点作为前驱节点。此外，该前驱节点必须是索引长度节点。

$$i \in V_{\text{index}} \Rightarrow |\operatorname{Preds}(i)| = 1 \text{ and } |\operatorname{Preds}(i) \cap V_{\text{length}}| = 1.$$

(C15) 轴长度节点的前驱节点必须是轴长度节点

$$i \in V_{\text{length}} \Rightarrow \operatorname{Preds}(i) \subseteq V_{\text{length}}$$

(C16) 轴长度节点的后继节点必须是索引节点或轴长度节点

$$i \in V_{\text{length}} \Rightarrow \operatorname{Succs}(i) \subseteq \left(V_{\text{index}} \cup V_{\text{length}}\right).$$

(C17) 维度节点的前驱节点必须是维度节点

$$i \in V_{\text{dim}} \Rightarrow \operatorname{Preds}(i) \subseteq V_{\text{dim}}.$$

(C18) 维度节点的后继节点必须是访问节点或维度节点

$$i \in V_{\text{dim}} \Rightarrow \operatorname{Succs}(i) \subseteq \left(V_{\text{access,in}} \cup V_{\text{access,out}} \cup V_{\text{dim}}\right).$$

(C19) 参数位置节点不能有任何前驱节点

$$i \in V_{\text{arg-pos}} \Rightarrow \operatorname{Preds}(i) = \emptyset.$$

(C20) 参数位置节点的后继节点必须是输入访问节点

$$i \in V_{\text{arg-pos}} \Rightarrow \operatorname{Succs}(i) \subseteq V_{\text{access,in}}.$$

(C21) 通过仅保留维度节点及其对应边而从 G 得到的子图必须是传递锦标赛。等价地，存在 V_{dim} 中顶点的一个排序 $v_1, \ldots, v_n$，使得

$$v_j \in \operatorname{Succs}(v_i) \text{ if and only if } i < j.$$

这些条件反映了前一节所述的边，这些边对应于式（17）中指定的不同维度上的节点排序。

(C22) 通过仅保留长度节点及其对应边而从 G 得到的子图必须是传递锦标赛。等价地，存在 V_{length} 中顶点的一个排序 $v_1, \ldots, v_n$，使得

$$v_j \in \operatorname{Succs}(v_i) \text{ if and only if } i < j.$$

这些条件涵盖了上一节概述的边，这些边对应于式（15）中指定的不同轴长度下的节点排序。

(C23) 仅保留数据类型节点及其对应边后得到的 G 子图必须是传递锦标赛。等价地，存在一种对 V_{dtype} 中顶点的排序 v_1、 ..... 、 v_n，使得

$$v_j \in \text{Succs}(v_i) \text{ 当且仅当 } i < j.$$

这些条件涵盖了上一节所述的边，这些边对应于公式（16）中指定的不同数值数据类型之间的节点排序。

（C24）如果 $n_1 \in V_{\text{access, in}}$, $n_2 \in V_{\text{access, in}}$, $i_1 \in \text{Preds}(n_1) \cap V_{\text{index}}$,$i_2 \in \text{Preds}(n_2) \cap V_{\text{index}}$, $(\text{Succs}(n_1) \cap V_{\text{arg}} = \text{Succs}(n_2) \cap V_{\text{arg}}) \wedge (\text{Preds}(n_1) \cap V_{\text{dim}} = \text{Preds}(n_2) \cap V_{\text{dim}})$，则 $\text{Preds}(i_1)$ 必须等于 $\text{Preds}(i_2)$。这通过以下条件确保数组具有一致的形状。

（C25）如果 $n \in V_{\text{access, out}}$, $i \in \text{Preds}(n) \cap V_{\text{index}}$，则 $\text{Succs}(i) \cap V_{\text{access, in}}$ 必须非空。这确保了如果索引节点正在索引一个输出节点，那么该索引节点也会在某个输入节点中出现。

(C26) 若 $n \in V_{\text{access,in}}$, $i \in \text{Preds}(n) \cap V_{\text{index}}$, $d \in \text{Preds}(n) \cap V_{\text{dim}}$, $k \in \text{Preds}(n)$ 且 $k \in V_{\text{arg-pos}}$，那么对于所有 $o \in V_{\text{output}}$，集合 $\{n' \in V_{\text{access,in}} : \{i, d, k, o\} \subseteq \text{Preds}(n')\}$ 必须是单元素集合。该条件确保所有输出的输入索引一致。

(C27) 集合 $\{\text{Preds}(n) \cap V_{\text{index}} : n \in V_{\text{access,out}}\}$ 的维数必须为 $|V_{\text{access,out}}|$

该条件确保所有输出索引互不相同。

(C28) 对于所有 $k \in V_{\text{arg-pos}}$, $o \in V_{\text{output}}$，$\text{Acc}_{k,o} = \{n : k \in \text{Preds}(n), o \in \text{Preds}(n), n \in V_{\text{access,in}}\}$。那么，集合 $\left(\bigcup_{a \in \text{Acc}_{k,o}} \text{Succs}(a)\right)$ 必须是单元素集合。该条件规定，每个输出只能对单个参数进行索引。

(C29) 对于所有 $k \in V_{\text{arg-pos}}$, $o \in V_{\text{output}}$，如果 $\text{Acc}_{k,o} = \{n : n \in V_{\text{access,in}}, k \in \text{Preds}(n), o \in \text{Preds}(n)\}$，且 $\text{Dims}_{k,o} \supseteq \bigcup_{n \in \text{Acc}_{k,o}} \text{Preds}(n) \cap V_{\text{dim}}$，那么必须满足以下条件：$|\{d : d \in \text{Dims}_{k,o}, |\text{Preds}(d) \cap \text{Dims}_{k,o} \neq \emptyset\}| = |\text{Dims}_{k,o}| - 1$该条件确保参数的所有连续维度都被索引。(C30) 我们定义 $\text{Dims}_{\text{out}} = \bigcup_{n \in V_{\text{access,out}}} \text{Preds}(n) \cap V_{\text{dim}}$。那么，$|\{d : d \in \text{Dims}_{\text{out}}, \text{Preds}(d) \cap V_{\text{dim}} = \emptyset\}| = 1|\{d : d \in \text{Dims}_{\text{out}}, |\{d : d \in \text{Dims}_{k,o}, \text{Preds}(d) \cap V_{\text{dim}} = \emptyset\}| = 1$ $\text{Preds}(d) \cap \text{Dims}_{\text{out}} \neq \emptyset\}| = |\text{Dims}_{\text{out}}| - 1$该条件确保输出的所有连续维度都被索引。

备注 2. 第 *B.1* 节中的诱导图构造满足属性 （*C1*）–（*C30*）。

对于满足上述合规性属性的诱导图，我们可以按照以下步骤得到相应的批量爱因斯坦求和：

步骤 1。选择两个函数 $\text{IndexName} : \mathbb{Z}^+ \to \text{identifier}$，用于命名重构后的批量爱因斯坦求和中的索引；以及 $\text{ArgName} : \mathbb{Z}^+ \to \text{identifier}$，用于命名重构后的批量爱因斯坦求和中的参数。**步骤 2**。为重构后的批量爱因斯坦求和定义参数排序的映射，记为 $\iota^{\text{inferred}}_{\text{arg-pos}} : V_{\text{arg-pos}} \mapsto \{1, \ldots, |V_{\text{arg-pos}}|\}$，使得如果 $n_1, n_2 \in V_{\text{dim}}$ 且 $n_1 < n_2$，那么 $\iota^{\text{inferred}}_{\text{arg-pos}}(n_1) < \iota^{\text{inferred}}_{\text{arg-pos}}(n_2)$。

步骤 3。为重构后的批量 einsum 中的输出顺序定义一个映射，记为 $\iota^{\text{inferred}}_{\text{output}}: V_{\text{output}} \mapsto \{1、\ldots\ldots、|V_{\text{output}}|\}$，使得如果 $n_1,\ n_2 \in V_{\text{output}}$ 和 $n_1 < n_2$，则 $\iota^{\text{inferred}}_{\text{output}}(n_1) < \iota^{\text{inferred}}_{\text{output}}(n_2)$。

步骤 4。为重构的批量爱因斯坦求和中的索引符号顺序定义一个映射，记作 $\iota^{\text{inferred}}{}_{\text{索引}}: V_{\text{index}} \mapsto \{1, \ldots, |V_{\text{index}}|\}$，使得如果 $n_1, n_2 \in V_{\text{index}}$ 和 $n_1 < n_2$，则 $\iota^{\text{inferred}}{}_{\text{索引}}(n_1) < \iota^{\text{inferred}}{}_{\text{索引}}(n_2)$。

步骤 5。为重构的批量爱因斯坦求和中的参数排序定义一个映射，记作 $\iota^{\text{inferred}}_{\text{arg}}: V_{\text{arg}} \mapsto \{1, \ldots, |V_{\text{arg}}|\}$，使得如果 $n_1, n_2 \in V_{\text{arg}}$ 和 $n_1 < n_2$，则 $\iota^{\text{inferred}}_{\text{arg}}(n_1) < \iota^{\text{inferred}}_{\text{arg}}(n_2)$。

步骤 6。定义重构的批量爱因斯坦求和中维度的映射，$\iota^{\text{inferred}}{}_{\text{维数}}: V_{\text{dim}} \mapsto \{1, \ldots, |V_{\text{dim}}|\}$，如下：

$$\iota^{\text{inferred}}_{\text{dim}}(n) = 1 + |\,\text{Preds}(n) \cap V_{\text{dim}}|. \tag{24}$$

步骤 7。查找输出索引，记为 $\mathrm{I}^{\text{out},\ \text{inferred}}$，如下：

$$\mathcal{I}^{\text{out,inferred}} = \text{IndexName}\left(\iota^{\text{inferred}}_{\text{index}}\left(\text{OutIndex}(k)\right)\right) \tag{25}$$

其中，

$$\text{OutIndex}(k) \in \left\{i : \left(\iota^{\text{inferred}}_{\text{dim}}\right)^{-1}(k) \in \text{Preds}(n), n \in V_{\text{access,out}}\right\}. \tag{26}$$

根据方程（23）中的判据可知，对于上述方程（OutIndex(k) 只有一个解（26）。

步骤 8。查找参数维度，记为 ArgDim(k)，如下：

$$\begin{aligned}\text{ArgDim}(k) = \Big|\{d \in V_{\text{dim}} :\quad & \{d, \left(\iota^{\text{inferred}}_{\text{arg-pos}}\right)^{-1}(k)\} \subset \text{Preds}(a), \\ & a \in V_{\text{access,in}}\}\Big|\end{aligned} \tag{27}$$

步骤 9。查找输入索引列表，记为 $\mathrm{I}^{\text{in},\ \text{inferred}}$，如下：

$$\mathcal{I}^{\text{in,inferred}} = \left(\left(\text{IndexName}\left(\iota^{\text{inferred}}_{\text{index}}\left(\text{InputIndex}(d,j)\right)\right)\right)_{d=1}^{\text{ArgDim}(j)}\right)_{j=1}^{|V_{\text{arg-pos}}|} \tag{28}$$

其中，

$$\begin{aligned}\text{InputIndex}(d,j) \in \{i :\ & i \in \text{Preds}(n), \\ & i \in V_{\text{index}}, \\ & \{\left(\iota^{\text{inferred}}_{\text{arg-pos}}\right)^{-1}[j], \left(\iota^{\text{inferred}}_{\text{dim}}\right)^{-1}[d]\} \subset \text{Preds}(n), \\ & n \in V_{\text{access,in}}\}\end{aligned} \tag{29}$$

根据公式（22）中的判据，我们可以看到，上述公式（输入索引（d，j）中恰好存在一个解（29）。

步骤 10。求出参数序列 $\left(\left(\mathrm{A}^{i,}{}_{j,\text{inferred}}\right)_{j=1}^{|V_{\text{arg}}\ -\ |}\right)_{i=1}^{|V_{\text{output}}|_{\text{pos}}}$，其中 $\mathrm{A}^{i,}{}_{j,}{}^{\text{inferred}}$ 是一个数组，其符号为 $\text{ArgName}\left(\iota^{\text{inferred}}_{\text{array}}(\text{ArgLabel}(i,j))\right)$，形状为 $\left(\iota_{\text{length}}(\text{LengthLabel}(i,j))\right)_{j=1}^{\text{ArgDim(j)}}$，数据类型为 ι_{dtype} (DtypeLabel(i, j))，其中，

$$\begin{aligned}\text{ArgLabel}(i,j) \in \Big\{ a : \{ &\left(\iota_{\text{output}}^{\text{inferred}}\right)^{-1}(i), \\ &\left(\iota_{\text{arg-pos}}^{\text{inferred}}\right)^{-1}(j)\} \subset \text{Preds}(n), \\ &n \in \left(\text{Preds}(n) \cap V_{\text{access,in}}\right), \\ &a \in V_{\text{arg}} \Big\}\end{aligned} \tag{30}$$

$$\text{DtypeLabel}(i,j) \in \left\{ d : V_{\text{dtype}}, d \in \text{Preds}(\text{ArgLabel}(i,j)) \right\} \tag{31}$$

$$\text{LengthLabel}(i,j) \in \left\{ l : l \in V_{\text{length}}, l \in \text{Preds}(\text{InputIndex}(i,j)) \right\} \tag{32}$$

根据判据 (C28), (C4),以及 (C14)，我们可以看出，方程（30）、（31）和（32）恰有一个解。

步骤 11。根据前面的步骤，我们将批量 einsum 重构为由元组$\left(|V_{\text{output}}|, |V_{\text{arg-pos}}|, \mathrm{I}^{\text{out,inferred}}, \left(\mathrm{I}^{k}_{\text{,in,inferred}}\right)^{|V_{\text{arg}}|}_{i=1}, \left(\left(\mathrm{A}^{i}_{,j,\text{inferred}}\right)^{|V_{\text{arg}}|}_{j=1}\right)^{|V_{\text{output}}|}_{i=1}\right)_{\text{pos}}$定义的批量 einsum。

C DG-FEM 批量 Einsum 内核模式

```
jac(f, e)     := J[f, e]
mat(i, f, j) := D[i, f, j]
f1(f, e, j)  := F_1[f, e, j]
f2(f, e, j)  := F_2[f, e, j]
# ...
fb(f, e, j)  := F_b[f, e, j]

适用于iel , idof
 y1[iel,idof] = sum([iface,jdof], f1(iface,iel,jdof)
                                  * 雅可比(iface,jdof)
                                  * mat(idof,iface,jdof]))
 y2[iel,idof] = 求和([iface,jdof], f2(iface,iel,jdof)
                                  * jac(iface,jdof)
                                  * mat(idof,iface,jdof]))
 # ...
 yb[iel,idof] = 求和([iface,jdof], fb(iface,iel,jdof)
                                  * 雅可比矩阵(iface,jdof)
                                  * mat(idof,iface,jdof]))
末尾 雅可比 , idof
```

清单 7. 一个执行 *b*次面质量运算的内核。变量 F_1、.....、F_b 对应表面通量。jac 是对应雅可比项的替换规则，mat 是对应表列参考曲面积分的规则。

```
雅可比(x,r,e)   := J[x,r,e]
```

```
 2 mat(r,i,j) :=  D[r,i,j]3 f1(x,e,j) :=  u1[e, j]  if x  ==  0  else (v1[e, j]  if x  ==  1  else w1[e, j]) 4
f2(x,e,j) :=  u2[e, j]  if x  ==  0  else (v2[e, j]  if x  ==  1  else w2[e, j]) 5 # ... 6 fb(x,e,j) :=  ub[e, j]  if x
 ==  0  else (vb[e, j]  if x  ==  1  else wb[e, j]) 7 8 for iel, idof 9 y1[iel,idof]  =sum([itopo_dim ,
iambient_dim ,jdof], f1(itopo_dim ,iel ,jdof)
10                                                                  *jac(itopo_dim ,iambient_dim ,iel)
11                                                                  *mat(iambient_dim ,idof ,jdof])) 12 y2[
iel,idof]  =sum([itopo_dim ,iambient_dim ,jdof], f2(itopo_dim ,iel ,jdof)
13                                                                  *jac(itopo_dim ,iambient_dim ,iel)
14                                                                  *mat(iambient_dim ,idof ,jdof])) 15 # ...
16 yb[iel,idof]  =sum([itopo_dim ,iambient_dim ,jdof], fb(itopo_dim ,iel ,jdof)
17                                                                  *jac(itopo_dim ,iambient_dim ,iel)
18                                                                  *mat(iambient_dim ,idof ,jdof])) 19
end iel, idof
```

列表 8。执行 *b*次局部散度运算的内核。元组 {(u1, v1, w1), . . ., (ub, vb, wb)} 对应于输入向量场的各个分量。jac 是与雅可比项对应的替换规则，mat 是参考导数矩阵。

```
 1 jac(x,r,e) :=  J[x,r,e]2 mat(r,i,j) :=  D[r,i,j]3 f1(x,e,j) :=  u1[e, j]4 f2(x,e,j) :=  u2[e, j]5 # ... 6 fb(x,e,j) :=  ub[e, j]7 8
for iel, idof 9 y1[itopo_dim , iel,idof]  =sum([iambient_dim ,jdof], f1(iel ,jdof)
10                                                                  *jac(itopo_dim ,iambient_dim ,iel)
11                                                                  *mat(iambient_dim ,idof ,jdof])) 12 y2[
itopo_dim , iel ,idof]  =sum([iambient_dim ,jdof], f2(iel ,jdof)
13                                                                  *jac(itopo_dim ,iambient_dim ,iel)
14                                                                  *mat(iambient_dim ,idof ,jdof])) 15 # ... 16
yb[itopo_dim , iel ,idof]  =sum([iambient_dim ,jdof], fb(iel ,jdof)
17                                                                  *jac(itopo_dim ,iambient_dim ,iel)
18                                                                  *mat(iambient_dim ,idof ,jdof])) 已提交至
ACM 的手稿
```

```
19 末尾 iel，idof
```

清单 9. 执行 b 次局部梯度运算的内核。变量 {u1, . . ., ub} 对应输入标量场。jac 是对应于雅可比项的替换规则，mat 是参考导数矩阵。