

神经网络加速器中的数据布局变换： 原理、实现与优化

nn_dataflow 项目文档

2025 年 12 月 21 日

摘要

数据布局（Data Layout）是深度学习加速器设计中的关键问题。本文系统性地阐述了数据布局的基本概念、布局变换的实现机制以及优化策略。重点讨论了以下核心问题：（1）布局变换应作用于权重还是激活值；（2）布局变换应与哪个算子融合实现；（3）变换应在读取阶段还是写入阶段完成；（4）实现布局变换的硬件单元设计。本文结合具体代码实现，详细分析了各种设计权衡，为神经网络加速器的设计提供参考。

目录

1 什么是数据布局	5
2 为什么需要布局	5
2.1 问题：多维数据 vs 一维内存	5
2.2 不同的展平方式	5
3 布局的形式化定义	5
3.1 基本形式	5
3.2 步长的含义	6
3.3 紧凑布局 vs 非紧凑布局	6
4 常见布局类型	6
4.1 连续布局	6
4.2 分块布局（Tiled Layout）	6
4.3 交错布局（Interleaved Layout）	7
5 布局对性能的影响	7
5.1 访存局部性	7
5.2 硬件适配	7
5.3 布局转换的代价	7

目录	2
6 布局传播	7
6.1 定义	7
6.2 传播规则	7
6.3 传播的作用	8
7 工程实现	8
7.1 布局的表示	8
7.2 地址计算	8
8 Data Layout 的底层实现	9
8.1 张量的核心数据结构	9
8.2 步长如何实现布局	9
8.2.1 不同布局对应不同步长	9
8.3 零拷贝操作的实现	9
8.3.1 Transpose (转置)	10
8.3.2 Slice (切片)	10
8.3.3 Reshape (重塑)	10
8.4 连续性检查	10
8.5 完整的地址计算流程	11
8.6 广播的实现: 零步长	11
8.7 实现总结	12
9 布局变换融合	12
9.1 变换的本质: 索引映射	12
9.2 多次变换 = 映射的复合	12
9.3 例子: 两次 Transpose 的融合	13
9.4 例子: Reshape + Transpose 融合	13
9.5 融合的条件	13
9.6 工程实现	13
9.7 融合的收益	14
10 布局变换算子的分类	14
10.1 纯索引重映射 (Bijective)	14
10.2 数据重组 (Non-bijective, 数据增加)	14
10.2.1 im2col: 典型的数据膨胀算子	14
10.2.2 显式 vs 隐式 im2col	15
10.3 数据选择 (数据减少)	15
10.4 分类总结	16
11 Layout Transformation 的对象: 权重 vs 激活值	16
11.1 权重的布局变换	16

11.1.1 特点分析	16
11.1.2 优化策略	16
11.1.3 代价分析	17
11.2 激活值的布局变换	17
11.2.1 特点分析	17
11.2.2 挑战	17
11.3 权重 vs 激活值：决策总结	19
12 Layout Transformation 与算子融合	19
12.1 融合位置的选择	19
12.1.1 策略 A: 与 Producer (生产者) 融合	19
12.1.2 策略 B: 与 Consumer (消费者) 融合	20
12.1.3 策略 C: 与 Memory Copy 融合	21
12.2 融合策略的代价比较	22
12.3 代码实现：算子间 Layout 协商	22
12.4 Elementwise 算子的特殊优势	23
13 读取时变换 vs 写入时变换	24
13.1 写入时变换 (Write-Time Transform)	24
13.1.1 原理	24
13.1.2 代码实现	24
13.1.3 优缺点	25
13.2 读取时变换 (Read-Time Transform)	25
13.2.1 原理	25
13.2.2 代码实现	26
13.2.3 优缺点	27
13.3 选择决策树	27
13.4 决策总结	28
14 实现 Layout Transformation 的硬件单元	28
14.1 地址生成单元 (Address Generation Unit, AGU)	28
14.1.1 基本架构	29
14.1.2 支持布局变换的 AGU 设计	30
14.2 DMA 控制器 (Direct Memory Access Controller)	30
14.2.1 支持布局变换的 DMA 设计	30
14.3 Scatter-Gather Engine	31
14.4 片上网络 (Network-on-Chip, NoC) 支持	33
14.5 硬件设计总结	35

15 2 级 Tiling 的 Data Layout 与 Access 分析	35
15.1 存储层次与 Tiling 层次	35
15.2 不匹配发生的两个层次	35
15.2.1 L1 层次: Data Layout vs L1 Tile Order	35
15.2.2 L2 层次: Data Layout vs L2 Sub-tile Order	36
15.3 匹配规则	36
15.4 效率量化分析	36
15.4.1 实验数据	36
15.5 设计指导	37
16 完整代码示例: 端到端的布局变换	37
16.1 场景设定	37
16.2 完整实现	37
16.3 运行结果	43
17 总结	44
17.1 变换对象: 权重 vs 激活值	44
17.2 算子融合策略	44
17.3 读取 vs 写入时机	45
17.4 硬件实现	45

1 什么是数据布局

数据布局（Data Layout）的本质是一个映射：

$$\boxed{\text{逻辑索引 } (n, c, h, w) \longrightarrow \text{物理地址 } \text{addr}}$$

给定一个多维张量的逻辑坐标，布局决定了它在内存中的实际存储位置。

2 为什么需要布局

2.1 问题：多维数据 vs 一维内存

张量是多维的，例如一个特征图有 4 个维度：

- N — 批次 (batch)
- C — 通道 (channel)
- H — 高度 (height)
- W — 宽度 (width)

但物理内存是一维的线性地址空间。布局定义了如何把多维索引“展平”到一维地址。

2.2 不同的展平方式

同一个张量，不同的布局会产生完全不同的内存排列：

布局	地址计算	特点
NCHW	$n \cdot CHW + c \cdot HW + h \cdot W + w$	通道优先 (PyTorch 默认)
NHWC	$n \cdot HWC + h \cdot WC + w \cdot C + c$	空间优先 (TensorFlow 默认)

3 布局的形式化定义

3.1 基本形式

设张量的形状为 $(D_0, D_1, \dots, D_{k-1})$ ，布局可以表示为一个排列 π 和对应的步长 (stride) 向量 s :

$$\text{addr}(i_0, i_1, \dots, i_{k-1}) = \text{base} + \sum_{j=0}^{k-1} i_j \cdot s_j \quad (1)$$

其中 s_j 是第 j 维的步长，决定了该维索引增加 1 时地址的增量。

3.2 步长的含义

以 NCHW 布局为例，形状为 (N, C, H, W) 的张量：

$$s_W = 1 \quad (\text{W 维连续}) \quad (2)$$

$$s_H = W \quad (\text{H 维跨 W 个元素}) \quad (3)$$

$$s_C = H \times W \quad (\text{C 维跨一个空间平面}) \quad (4)$$

$$s_N = C \times H \times W \quad (\text{N 维跨一个完整样本}) \quad (5)$$

3.3 紧凑布局 vs 非紧凑布局

紧凑布局 步长严格按维度大小递推，没有空隙：

$$s_{\pi(j)} = \prod_{i < j} D_{\pi(i)}$$

非紧凑布局 步长可以任意指定，可能存在：

- **Padding:** 步长大于所需，元素之间有空隙
- **子视图:** 从更大张量中切出的子区域
- **广播:** 某维步长为 0，表示该维重复使用同一数据

4 常见布局类型

4.1 连续布局

最简单的布局，维度按固定顺序排列：

- **NCHW:** 先遍历 W，再 H，再 C，最后 N
- **NHWC:** 先遍历 C，再 W，再 H，最后 N
- **CHWN:** 先遍历 N，再 W，再 H，最后 C

4.2 分块布局 (Tiled Layout)

将张量按固定大小分块，块内和块间可以有不同的排列顺序。

例如 NCHW 按 8×8 分块：

```
for n in range(N):
    for c in range(C):
        for h_tile in range(H // 8):
            for w_tile in range(W // 8):
                for h_in_tile in range(8):      # 块内
                    for w_in_tile in range(8): # 块内
                        addr = ...
```

分块布局对硬件友好，因为：

- 块大小可以匹配缓存行或 SRAM 大小
- 块内访问连续，提高局部性

4.3 交错布局 (Interleaved Layout)

将某个维度的元素交错存放，常用于向量化访问。

例如通道交错 (channel interleaving): 每 k 个通道的同一空间位置连续存放。

5 布局对性能的影响

5.1 访存局部性

不同的布局决定了访问模式的局部性：

- **卷积**: NCHW 布局下，空间维度连续，利于滑动窗口访问
- **通道拼接**: NHWC 布局下，通道连续，拼接操作更高效
- **批处理**: 如果 N 在最内层，同一位置的不同样本连续

5.2 硬件适配

不同硬件偏好不同布局：

硬件	偏好布局
NVIDIA GPU (cuDNN)	NCHW 或 NHWC (取决于算法)
Intel CPU (MKL-DNN)	分块布局 (如 nChw16c)
TPU	NHWC
自定义加速器	取决于数据通路设计

5.3 布局转换的代价

当上下游算子需要不同布局时，需要进行布局转换 (layout transform)：

- 本质是一次内存重排 (memory transpose)
- 代价 = 读取全部数据 + 写入全部数据
- 应尽量避免，或与其他操作融合

6 布局传播

6.1 定义

布局传播是指：在神经网络的计算图中，根据算子语义和上游布局，推导下游数据的布局。

6.2 传播规则

恒等传播 逐元素算子 (ReLU、Add) 保持输入布局不变。

转置传播 Transpose/Permute 改变维度顺序，相应调整步长。

卷积传播 输入输出通常保持相同的空间布局，但通道维度可能变化。

重塑传播 Reshape 可能需要转换为连续布局才能正确重塑。

6.3 传播的作用

1. 确定每个中间张量的内存布局
2. 识别需要布局转换的位置
3. 为代价模型提供访存模式信息

7 工程实现

7.1 布局的表示

常见的表示方式：

```
# 方式1：步长表示
layout = {
    'shape': (N, C, H, W),
    'strides': (C*H*W, H*W, W, 1) # NCHW
}
```

```
# 方式2：维度顺序表示
layout = 'NCHW' # 或 'NHWC'
```

```
# 方式3：分块表示
layout = {
    'outer': 'NCHW',
    'tile_size': (1, 8, 8, 8),
    'inner': 'CHWN'
}
```

7.2 地址计算

给定布局和索引，计算地址：

```
def compute_addr(index, strides, base=0):
    addr = base
    for i, s in zip(index, strides):
        addr += i * s
    return addr
```

8 Data Layout 的底层实现

本节解释深度学习框架（如 PyTorch、NumPy）如何在底层实现 Data Layout。

8.1 张量的核心数据结构

一个张量在内存中由两部分组成：

1. **数据存储 (Storage)**: 一块连续的原始内存
2. **元数据 (Metadata)**: 描述如何解释这块内存

```
class Tensor:
    storage: ptr          # 指向原始数据的指针
    shape: tuple          # 形状, 如 (2, 3, 4)
    strides: tuple        # 步长, 如 (12, 4, 1)
    offset: int           # 起始偏移 (默认为 0)
    dtype: DataType        # 数据类型, 如 float32
```

关键点：多个张量可以共享同一块 `storage`，只是用不同的元数据来“解释”它。

8.2 步长如何实现布局

步长 (strides) 是实现布局的核心机制：

```
# PyTorch 示例
import torch

x = torch.randn(2, 3, 4)
print(x.shape)      # torch.Size([2, 3, 4])
print(x.stride()) # (12, 4, 1) -- 紧凑 C 顺序

# 地址计算: x[i,j,k] 的地址 = base + i*12 + j*4 + k*1
```

8.2.1 不同布局对应不同步长

布局	shape	strides
NCHW (PyTorch 默认)	(N, C, H, W)	(C*H*W, H*W, W, 1)
NHWC (TensorFlow 默认)	(N, H, W, C)	(H*W*C, W*C, C, 1)
Fortran 顺序	(M, N)	(1, M)

8.3 零拷贝操作的实现

许多操作不需要移动数据，只需要修改元数据：

8.3.1 Transpose (转置)

```
x = torch.randn(2, 3, 4)
print(x.stride())           # (12, 4, 1)

y = x.transpose(1, 2)        # 交换维度 1 和 2
print(y.shape)              # torch.Size([2, 4, 3])
print(y.stride())           # (12, 1, 4) -- 步长交换!

# x 和 y 共享同一块内存, 只是步长不同
print(x.data_ptr() == y.data_ptr()) # True
```

8.3.2 Slice (切片)

```
x = torch.randn(10, 20)
print(x.stride())           # (20, 1)

y = x[2:8, 5:15]           # 切出子区域
print(y.shape)              # torch.Size([6, 10])
print(y.stride())           # (20, 1) -- 步长不变
print(y.storage_offset())   # 45 = 2*20 + 5

# y 是 x 的视图, 指向同一块 storage, 但偏移不同
```

8.3.3 Reshape (重塑)

```
x = torch.randn(2, 3, 4)
print(x.stride())           # (12, 4, 1)

y = x.reshape(6, 4)          # 如果连续, 只改元数据
print(y.stride())           # (4, 1)

# 如果不连续, reshape 会先拷贝数据
x_t = x.transpose(1, 2)      # 非连续
print(x_t.is_contiguous())   # False
y_t = x_t.reshape(6, 4)       # 这会触发拷贝!
```

8.4 连续性检查

连续 (**contiguous**) 意味着步长严格递减且相邻元素在内存中相邻:

```
def is_contiguous(shape, strides):
    """检查是否为 C 顺序连续"""
    pass
```

```

expected_stride = 1
for dim, stride in reversed(list(zip(shape, strides))):
    if stride != expected_stride:
        return False
    expected_stride *= dim
return True

```

许多算子（如 GEMM）要求输入连续，非连续张量需要先调用 `.contiguous()` 复制为连续布局。

8.5 完整的地址计算流程

```

def tensor_element_address(tensor, *indices):
    """
    计算 tensor[i0, i1, ..., ik] 的内存地址
    """

    # 1. 检查索引有效性
    for idx, (i, dim) in enumerate(zip(indices, tensor.shape)):
        assert 0 <= i < dim, f"Index {idx} out of bounds"

    # 2. 计算相对偏移
    offset = tensor.storage_offset
    for i, stride in zip(indices, tensor.strides):
        offset += i * stride

    # 3. 计算绝对地址
    element_size = sizeof(tensor.dtype)  # 如 float32 = 4 bytes
    byte_address = tensor.storage.data_ptr + offset * element_size

    return byte_address

```

8.6 广播的实现：零步长

广播（broadcast）通过将某维度的步长设为 0 来实现：

```

# 原始向量 (3,)
a = torch.tensor([1, 2, 3])
print(a.stride())           # (1,)

# 扩展为 (4, 3)，但不复制数据
b = a.expand(4, 3)
print(b.shape)              # torch.Size([4, 3])

```

```
print(b.stride())           # (0, 1) -- 第一维步长为 0!
```

```
# b[0,:], b[1,:], b[2,:], b[3,:] 都指向同一行
```

步长为 0 意味着该维度的索引变化不影响地址，实现了“虚拟复制”。

8.7 实现总结

操作	实现方式	是否拷贝
创建张量	分配 storage + 设置 strides	是（初始分配）
Transpose	交换 strides 和 shape	否
Slice	调整 offset + shape	否
Reshape（连续时）	重算 strides	否
Reshape（非连续时）	先拷贝再重算	是
Expand/Broadcast	插入步长为 0 的维度	否
Contiguous	复制到新 storage	是

9 布局变换融合

当计算图中存在多个连续的布局变换时，可以将它们融合为一个变换，减少内存访问。

9.1 变换的本质：索引映射

每个布局变换本质上是一个索引映射函数：

$$T : \text{index}_{\text{old}} \longrightarrow \text{index}_{\text{new}}$$

完整的数据搬运涉及两步：

$$\text{addr}_{\text{src}} = \text{layout}_{\text{src}}(\text{index}) \quad (6)$$

$$\text{addr}_{\text{dst}} = \text{layout}_{\text{dst}}(T(\text{index})) \quad (7)$$

9.2 多次变换 = 映射的复合

假设有两个连续变换 T_1 和 T_2 ：

不融合 (3 次内存访问)：

原数据 --读取--> 中间结果1 --写入-->

中间结果1 --读取--> 中间结果2 --写入-->

融合 (1 次内存访问)：

$$\text{addr}_{\text{dst}} = \text{layout}_{\text{dst}}(T_2(T_1(\text{index}))) = \text{layout}_{\text{dst}}((T_2 \circ T_1)(\text{index}))$$

9.3 例子：两次 Transpose 的融合

考虑 NCHW \rightarrow NHWC \rightarrow NCHW:

变换	映射
T_1	$(n, c, h, w) \rightarrow (n, h, w, c)$
T_2	$(n, h, w, c) \rightarrow (n, c, h, w)$
$T_2 \circ T_1$	$(n, c, h, w) \rightarrow (n, c, h, w)$ = 恒等映射

融合结果：两次转置抵消，不需要任何数据搬运。

9.4 例子：Reshape + Transpose 融合

原始张量 $(1, 64, 56, 56)$ NCHW，目标布局 $(1, 3136, 64)$:

不融合：

1. Reshape: $(1, 64, 56, 56) \rightarrow (1, 64, 3136)$, 写入临时缓冲
2. Transpose: $(1, 64, 3136) \rightarrow (1, 3136, 64)$, 写入最终结果

融合：直接计算复合地址映射

```
def fused_transform(n, c, h, w):
    src_addr = n*64*56*56 + c*56*56 + h*56 + w      # NCHW
    dst_addr = n*3136*64 + (h*56+w)*64 + c           # (1, 3136, 64)
    return src_addr, dst_addr
```

一次读取源地址，一次写入目标地址，省掉中间缓冲。

9.5 融合的条件

1. 代数可消: $T_2 \circ T_1 = I$ (恒等映射)
2. 代数可简化: 复合映射比分开执行更简单
3. 硬件支持: 地址生成器能实现复合映射 (如支持任意步长)

9.6 工程实现

融合后的变换在 DMA 或地址生成器中实现:

```
# 不融合: 两次 memcpy
tmp = transpose(src, perm1)
dst = transpose(tmp, perm2)

# 融合: 一次 memcpy, 复合地址计算
for idx in all_indices:
```

```

src_addr = compute_src_addr(idx)
dst_addr = compute_fused_dst_addr(idx) # T2(T1(idx))
dst[dst_addr] = src[src_addr]

```

9.7 融合的收益

	不融合	融合
内存访问	多次读写	一次读写
中间缓冲	需要	不需要
地址计算	简单	复杂但一次完成

10 布局变换算子的分类

并非所有“布局变换”都是纯粹的索引重映射。根据映射的数学性质，可以将其分为三类。

10.1 纯索引重映射 (Bijective)

这类变换是双射：输入和输出元素一一对应，数据总量不变。

算子	映射	特点
Transpose	$(n, c, h, w) \rightarrow (n, h, w, c)$	维度置换
Reshape	$(n, c, h, w) \rightarrow (n, c \cdot h \cdot w)$	维度合并/拆分
Permute	任意维度重排	Transpose 的一般化

核心性质：

- 完全可逆
- 可以实现为“零拷贝”（只改变步长/元数据）
- 多个变换可融合为复合映射

10.2 数据重组 (Non-bijective, 数据增加)

这类变换会复制数据，输出元素多于输入元素。

10.2.1 im2col: 典型的数据膨胀算子

im2col 将卷积运算转换为矩阵乘法：

$$\text{im2col} : \mathbb{R}^{N \times C \times H \times W} \rightarrow \mathbb{R}^{(N \cdot H_{out} \cdot W_{out}) \times (C \cdot K_h \cdot K_w)}$$

为什么数据量增加：卷积滑窗有重叠，同一输入元素被复制多次。

输入 3×3 , 卷积核 2×2 , stride=1:

[a b c] im2col 矩阵（每行是一个展开的 patch）：

[d e f]	\rightarrow	[a b d e]	\leftarrow 位置(0,0)
[g h i]		[b c e f]	\leftarrow 位置(0,1), b和e被复制
		[d e g h]	\leftarrow 位置(1,0)
		[e f h i]	\leftarrow 位置(1,1)

输入 9 个元素 \rightarrow 输出 16 个元素，元素 e 出现了 4 次

数学表示：im2col 是一个 gather 操作：

$$M[i, j] = X[\text{gather_index}(i, j)]$$

其中 gather_index 不是单射——多个 (i, j) 可以指向同一个输入位置。

其他类似算子：

- unfold —im2col 的 PyTorch 版本
- repeat / tile —显式复制
- broadcast —隐式复制（不物化）

10.2.2 显式 vs 隐式 im2col

方式	内存开销	实现
显式 im2col	高（真正复制数据）	<code>torch.nn.functional.unfold</code>
隐式 im2col	零（不物化）	cuDNN implicit GEMM

隐式 im2col 的思路：不创建中间矩阵，而是在 GEMM 内部动态计算源地址。

```
# 隐式 im2col: 融合 gather 到计算中
for out_row, out_col in output_positions:
    for k in reduction_dim:
        # 根据 (out_row, out_col, k) 计算输入地址
        in_addr = compute_input_addr(out_row, out_col, k)
        result[out_row, out_col] += input[in_addr] * weight[k, out_col]
```

10.3 数据选择（数据减少）

这类变换选取部分数据，输出元素少于输入元素。

算子	操作	特点
Slice	连续子区域	可通过调整 base + shape 实现
Gather / Index_select	任意位置	需要索引数组
Squeeze	删除大小为 1 的维度	零拷贝

10.4 分类总结

类别	映射性质	数据量	可融合性
纯索引重映射	双射	不变	可融合为复合映射
数据重组	非单射（多对一的逆）	增加	需决定是否物化
数据选择	非满射	减少	可部分融合

硬件映射启示：

- 纯索引重映射：可完全通过地址生成器实现，零额外内存
- 数据重组（如 im2col）：需权衡“显式物化”（内存换规整计算）vs “隐式计算”（省内存但访问复杂）
- 数据选择：通常可通过调整地址范围实现

11 Layout Transformation 的对象：权重 vs 激活值

布局变换可以应用于神经网络中的两类数据：权重（Weight）和激活值（Activation）。两者在变换时机、代价和优化策略上有本质区别。

11.1 权重的布局变换

11.1.1 特点分析

权重数据具有以下特性：

- **静态性**：推理阶段权重固定不变
- **可预处理**：可在编译期/部署前完成布局变换
- **一次变换**：只需变换一次，可被所有输入复用
- **存储于外部内存**：通常存储在 DRAM/Flash 中

11.1.2 优化策略

由于权重的静态特性，最佳策略是离线预处理：

Listing 1: 权重布局预处理示例

```

1 class WeightLayoutOptimizer:
2     """
3         权重布局优化器
4
5         在模型部署前，将权重转换为硬件最优布局
6     """
7
8     @staticmethod
9     def preprocess_weight(weight, target_layout):
10        """
11            离线预处理权重布局

```

```

12
13     Args:
14         weight: 原始权重 (OIHW format)
15         target_layout: 目标布局, 如 (0, 2, 3, 1) for OHWI
16
17     Returns:
18         变换后的权重, 直接以目标布局存储
19         """
20         # 计算目标布局的步长
21         shape = weight.shape
22         target_strides = compute_permuted_strides(shape, target_layout)
23
24         # 创建新的存储并按目标布局写入
25         new_weight = allocate_storage(shape, target_strides)
26
27         # 一次性完成变换 (离线执行, 不影响推理延迟)
28         for idx in iterate_all_indices(shape):
29             src_addr = compute_addr(idx, weight.strides)
30             dst_addr = compute_addr(idx, target_strides)
31             new_weight[dst_addr] = weight[src_addr]
32
33     return new_weight

```

11.1.3 代价分析

阶段	代价	影响
离线预处理	$O(W)$ 内存访问	不影响推理延迟
模型部署	存储空间增加 (如果保留原始权重)	可接受
推理时	零额外开销	最优

11.2 激活值的布局变换

11.2.1 特点分析

激活值数据具有以下特性：

- **动态性：**每次推理都会产生新的激活值
- **无法预处理：**必须在运行时完成变换
- **数据量大：**通常比权重大 (尤其是大 batch 时)
- **生命周期短：**中间激活值可以及时释放

11.2.2 挑战

激活值的布局变换是推理延迟的关键因素：

Listing 2: 激活值布局变换的代价模型

```

1  class ActivationLayoutCost:
2      """
3          激活值布局变换的代价模型
4      """
5
6      @staticmethod
7      def estimate_transform_cost(shape, src_layout, dst_layout,
8                                     mem_config):
9          """
10             估算激活值布局变换的代价
11
12         Args:
13             shape: 激活值形状 (N, C, H, W)
14             src_layout: 源布局
15             dst_layout: 目标布局
16             mem_config: 内存配置
17
18         Returns:
19             cycles: 执行周期数
20             energy: 能耗 (pJ)
21
22         """
23
24         data_size = prod(shape) * 4 # float32
25
26
27         if src_layout == dst_layout:
28             return 0, 0 # 无需变换
29
30         # 单独的 layout transform 算子 (DRAM -> DRAM)
31         # 需要读取全部数据 + 写入全部数据
32         read_cycles = data_size / mem_config.dram_bandwidth
33         write_cycles = data_size / mem_config.dram_bandwidth
34         total_cycles = read_cycles + write_cycles + 2 *
35                         mem_config.dram_latency
36
37         energy = 2 * data_size * mem_config.dram_energy_per_byte
38
39         return total_cycles, energy

```

11.3 权重 vs 激活值：决策总结

特性	权重	激活值
变换时机	离线预处理	运行时
代价分摊	一次变换，多次推理复用	每次推理都需要
优化策略	编译期确定最优布局	融合到算子/内存拷贝
存储位置	DRAM/Flash（持久存储）	SRAM/DRAM（临时）

核心原则：

- 权重布局变换应在离线阶段完成，零运行时开销
- 激活值布局变换应融合到其他操作中，避免单独的变换算子

12 Layout Transformation 与算子融合

当激活值需要布局变换时，关键问题是：变换应该与哪个算子融合？

12.1 融合位置的选择

考虑以下计算图：

$$\text{Conv}_1 \xrightarrow{\text{NCHW}} \text{ReLU} \xrightarrow{\text{NCHW}} ? \text{ 变换? } \xrightarrow{\text{NHWC}} \text{Conv}_2$$

有三种可能的融合策略：

12.1.1 策略 A：与 Producer (生产者) 融合

变换融合到上游算子（如 Conv₁ 或 ReLU）的输出阶段：

Listing 3: Producer 融合策略

```

1 class ProducerFusedTransform:
2     """
3         将 Layout Transform 融合到 Producer 的写入阶段
4
5         Producer 计算完成后，按照 Consumer 期望的 layout 生成写地址
6     """
7
8     @staticmethod
9     def create_write_address_generator(shape, producer_compute_order,
10                                         target_layout, base_addr=0):
11         """
12             创建融合 layout 变换的写入地址生成器
13
14             Producer 在写回结果时，直接按 target_layout 写入，
15             避免后续的显式 layout transform
16         """

```

```

17     # 计算目标 layout 的步长
18     target_strides = compute_strides(shape, target_layout)
19
20     return AddressPattern(
21         base_addr=base_addr,
22         shape=shape,
23         strides=target_strides, # 按目标布局的步长写入
24         loop_order=producer_compute_order
25     )

```

适用场景:

- Producer 的输出被多个 Consumer 使用，且它们期望相同布局
- Producer 的计算循环顺序与目标布局兼容

12.1.2 策略 B: 与 Consumer (消费者) 融合

变换融合到下游算子（如 Conv₂）的输入读取阶段:

Listing 4: Consumer 融合策略

```

1 class ConsumerFusedTransform:
2     """
3         将 Layout Transform 融合到 Consumer 的读取阶段
4
5         Consumer 按照上游实际存储的 layout 生成读地址
6     """
7
8     @staticmethod
9     def create_read_address_generator(shape, source_layout,
10                                         consumer_compute_order,
11                                         base_addr=0):
12         """
13             创建融合 layout 变换的读取地址生成器
14
15             Consumer 读取时，按 source_layout 计算读地址，
16             数据在读取过程中被重新排列到计算单元期望的顺序
17
18             # 计算源 layout 的步长
19             source_strides = compute_strides(shape, source_layout)
20
21             return AddressPattern(
22                 base_addr=base_addr,
23                 shape=shape,
24                 strides=source_strides, # 按源布局的步长读取
25                 loop_order=consumer_compute_order
26             )

```

适用场景:

- 上游数据有多种可能的布局（如来自不同分支）
- Consumer 对读取顺序有灵活性

12.1.3 策略 C: 与 Memory Copy 融合

变换融合到内存层次之间的数据搬移（如 DRAM → SRAM）：

Listing 5: Memory Copy 融合策略

```

1 class MemoryCopyFusedTransform:
2     """
3         将 Layout Transform 融合到 Memory Copy 过程
4
5         在 DRAM -> SRAM 的必需搬移中完成布局变换
6     """
7
8     @staticmethod
9     def create_fused_copy_generator(shape, source_layout, target_layout):
10        """
11            创建融合 layout 变换的 memory copy 地址生成器对
12        """
13        source_strides = compute_strides(shape, source_layout)
14        target_strides = compute_strides(shape, target_layout)
15
16        # 确定最优遍历顺序（优先让读取连续）
17        read_order = optimal_traverse_order(source_strides)
18
19        read_pattern = AddressPattern(
20            shape=shape,
21            strides=source_strides,
22            loop_order=read_order
23        )
24
25        write_pattern = AddressPattern(
26            shape=shape,
27            strides=target_strides,
28            loop_order=read_order    # 使用相同遍历顺序
29        )
30
31        return read_pattern, write_pattern

```

适用场景:

- 数据需要从 DRAM 预取到 SRAM
- 变换可以隐藏在必需的数据搬移延迟中

12.2 融合策略的代价比较

策略	额外 DRAM 访问	硬件要求	灵活性
单独 Transform 算子	$2 \times$ (读 + 写)	简单	差
Producer 融合	0×	灵活地址生成	中
Consumer 融合	0×	灵活地址生成	高
Memory Copy 融合	0×	DMA 支持	高

12.3 代码实现：算子间 Layout 协商

Listing 6: Layout 协商器实现

```

1 class OperatorLayoutNegotiator:
2     """
3         算子间 Layout 协商器
4
5         在相邻算子之间协商 layout，决定最优的变换策略
6     """
7
8     def negotiate(self, producer, consumer, tensor_name):
9         """
10            协商两个算子之间的 layout 处理方式
11        """
12
13         producer_layout = producer.output_layouts[tensor_name]
14         consumer_layout = consumer.input_layouts[tensor_name]
15         shape = producer.output_shapes[tensor_name]
16
17         if producer_layout == consumer_layout:
18             return {"transform_needed": False, "strategy": "DIRECT"}
19
20         # 比较不同策略的代价
21         strategies = []
22
23         # 策略 1: Producer 写入时变换
24         cost_1 = self.estimate_producer_transform_cost(
25             shape, producer, consumer_layout)
26         strategies.append(("PRODUCER_WRITE_TRANSFORM", cost_1))
27
28         # 策略 2: Consumer 读取时变换
29         cost_2 = self.estimate_consumer_transform_cost(
30             shape, producer_layout, consumer)
31         strategies.append(("CONSUMER_READ_TRANSFORM", cost_2))
32
33         # 策略 3: Memory Copy 中变换
34         cost_3 = self.estimate_memcpy_transform_cost(

```

```

34     shape, producer_layout, consumer_layout)
35     strategies.append(("FUSED_COPY_TRANSFORM", cost_3))
36
37     # 选择代价最小的策略
38     best_strategy = min(strategies, key=lambda x: x[1])
39
40     return {
41         "transform_needed": True,
42         "best_strategy": best_strategy[0],
43         "estimated_cost": best_strategy[1]
44     }

```

12.4 Elementwise 算子的特殊优势

关键观察: Elementwise 算子（如 ReLU、Add）是布局变换融合的理想位置。

原因分析:

- **布局不敏感:** Elementwise 算子对输入布局没有要求
- **可以透传布局:** 输入什么布局，输出就什么布局
- **计算简单:** 不会增加地址生成的复杂度

Listing 7: Elementwise 算子透传布局

```

1 class ReductionAnalyzer:
2     """
3     规约分析器
4
5     分析每个算子是否包含规约操作，以此判断布局敏感性
6     """
7
8     REDUCTION_PATTERNS = {
9         # 有规约的算子 -> 布局敏感
10        'Conv': {'has_reduction': True, 'reduction_dims': ['C', 'R', 'S']},
11        'MatMul': {'has_reduction': True, 'reduction_dims': ['K']},
12        'Pool': {'has_reduction': True, 'reduction_dims': ['H', 'W']},
13
14         # 无规约的算子 -> 布局不敏感，可以透传
15        'ReLU': {'has_reduction': False, 'reduction_dims': []},
16        'Add': {'has_reduction': False, 'reduction_dims': []},
17        'Mul': {'has_reduction': False, 'reduction_dims': []},
18    }
19
20    @classmethod
21    def is_layout_sensitive(cls, op_type):
22        """判断算子是否对布局敏感"""
23        pattern = cls.REDUCTION_PATTERNS.get(op_type, {})

```

```
24     return pattern.get('has_reduction', True)
```

最佳实践: 将布局变换插入到布局敏感算子之间的 Elementwise 算子链中, 利用 Elementwise 的布局透传特性。

13 读取时变换 vs 写入时变换

布局变换可以在读取阶段或写入阶段完成。这两种方案在实现复杂度、性能和适用场景上有显著差异。

13.1 写入时变换 (Write-Time Transform)

13.1.1 原理

Producer 在将计算结果写回内存时, 直接按照下游期望的布局组织数据:

写入时变换流程:

Producer 计算单元 $\xrightarrow{\text{计算结果}}$ AGU (按目标布局计算地址) $\xrightarrow{\text{写入}}$ Memory (目标布局存储)

Producer 计算循环:

```
for n, c, h, w in compute_order:
    result = compute(n, c, h, w)

    # 写入时按 NHWC 布局计算地址
    addr = n * (H*W*C) + h * (W*C) + w * C + c
    memory[addr] = result
```

13.1.2 代码实现

Listing 8: 写入时变换的地址生成

```
1 class WriteTimeTransform:
2     """
3     写入时变换的实现
4
5     核心思想: Producer 写回时, 使用目标布局的步长计算写地址
6     """
7
8     def __init__(self, shape, target_layout):
9         self.shape = shape
10        # 计算目标布局的步长
11        self.target_strides = self.compute_strides(shape, target_layout)
12
13    def compute_strides(self, shape, layout_order):
```

```

14     """根据 layout 顺序计算步长"""
15     physical_shape = [shape[layout_order[i]] for i in
16                         range(len(shape))]
17
18     physical_strides = [1] * len(shape)
19     for i in range(len(shape) - 2, -1, -1):
20         physical_strides[i] = physical_strides[i + 1] *
21             physical_shape[i + 1]
22
23     # 映射回逻辑步长
24     logical_strides = [0] * len(shape)
25     for physical_dim, logical_dim in enumerate(layout_order):
26         logical_strides[logical_dim] = physical_strides[physical_dim]
27
28     return tuple(logical_strides)
29
30
31 def get_write_address(self, n, c, h, w, base_addr=0):
32     """计算写入地址"""
33     offset = (n * self.target_strides[0] +
34               c * self.target_strides[1] +
35               h * self.target_strides[2] +
36               w * self.target_strides[3])
37     return base_addr + offset * 4 # 4 bytes per float32

```

13.1.3 优缺点

优点:

- 数据一旦写入，后续所有 Consumer 都可以直接使用
- 适合一个 Producer 对应多个 Consumer 的情况
- 写入通常有更好的局部性（顺序写）

缺点:

- Producer 需要知道 Consumer 的布局需求
- 如果不同 Consumer 需要不同布局，无法同时满足
- 写地址计算可能变复杂

13.2 读取时变换 (Read-Time Transform)

13.2.1 原理

Consumer 在读取数据时，根据上游实际存储的布局计算读地址：

Consumer 计算循环（期望 NHWC 顺序）：

```

for n, h, w, c in compute_order: # NHWC 顺序
    # 读取时按源数据的 NCHW 布局计算地址

```

```

addr = n * (C*H*W) + c * (H*W) + h * W + w
data = memory[addr]

result = compute(data)

```

13.2.2 代码实现

Listing 9: 读取时变换的地址生成

```

1 class ReadTimeTransform:
2     """
3     读取时变换的实现
4
5     核心思想: Consumer 读取时, 使用源布局的步长计算读地址
6     """
7
8     def __init__(self, shape, source_layout):
9         self.shape = shape
10        # 计算源布局的步长
11        self.source_strides = self.compute_strides(shape, source_layout)
12
13    def compute_strides(self, shape, layout_order):
14        """根据 layout 顺序计算步长"""
15        # 与 WriteTimeTransform 相同
16        physical_shape = [shape[layout_order[i]] for i in
17                           range(len(shape))]
18
19        physical_strides = [1] * len(shape)
20        for i in range(len(shape) - 2, -1, -1):
21            physical_strides[i] = physical_strides[i + 1] *
22                                  physical_shape[i + 1]
23
24        logical_strides = [0] * len(shape)
25        for physical_dim, logical_dim in enumerate(layout_order):
26            logical_strides[logical_dim] = physical_strides[physical_dim]
27
28        return tuple(logical_strides)
29
30    def get_read_address(self, n, c, h, w, base_addr=0):
31        """计算读取地址"""
32        offset = (n * self.source_strides[0] +
33                  c * self.source_strides[1] +
34                  h * self.source_strides[2] +
35                  w * self.source_strides[3])
36        return base_addr + offset * 4

```

13.2.3 优缺点

优点：

- Consumer 独立决定如何读取，不影响 Producer
 - 灵活性高，可以适配多种上游布局
 - 适合数据来源不确定的场景（如多分支合并）

缺点:

- 读取可能不连续，降低内存带宽利用率
 - 每个 Consumer 都需要处理布局差异
 - 读地址计算复杂度增加

13.3 选择决策树

Listing 10: 读写时机选择的决策逻辑

```
1 def choose_transform_timing(producer, consumers, source_layout,
2                             target_layouts):
3     """
4         选择布局变换的时机：写入时 vs 读取时
5
6     Args:
7         producer: 生产者算子
8         consumers: 消费者算子列表
9         source_layout: 源布局
10        target_layouts: 各消费者期望的布局
11
12    Returns:
13        "WRITE_TIME" 或 "READ_TIME"
14
15    """
16
17    # 规则 1: 如果所有 consumer 期望相同布局, 写入时变换
18    unique_layouts = set(target_layouts)
19    if len(unique_layouts) == 1:
20        return "WRITE_TIME"
21
22    # 规则 2: 如果 consumer 数量少, 读取时变换更灵活
23    if len(consumers) <= 2:
24        return "READ_TIME"
25
26    # 规则 3: 分析写入连续性 vs 读取连续性
27    write_continuity = analyze_continuity(
28        producer.compute_order, list(unique_layouts)[0])
29
30    avg_read_continuity = sum(
31        analyze_continuity(c.compute_order, source_layout)
```

```

30         for c in consumers
31     ) / len(consumers)
32
33     if write_continuity > avg_read_continuity:
34         return "WRITE_TIME"
35     else:
36         return "READ_TIME"
37
38
39 def analyze_continuity(compute_order, layout):
40     """
41     分析计算顺序与布局的匹配度
42
43     返回 0-1 之间的分数，越高表示访问越连续
44     """
45
46     # 检查最内层循环是否对应最小步长的维度
47     innermost_dim = compute_order[-1]
48     layout_strides = compute_strides_for_layout(layout)
49
50     min_stride_dim = min(range(len(layout_strides)),
51                           key=lambda d: layout_strides[d])
52
53     if innermost_dim == min_stride_dim:
54         return 1.0  # 完美连续
55     else:
56         return 0.5  # 部分连续

```

13.4 决策总结

场景	推荐方案	原因
单 Producer 多相同 Consumer	写入时变换	一次变换多次使用
多 Producer 单 Consumer	读取时变换	Consumer 统一处理
计算顺序匹配目标布局	写入时变换	写入连续性好
计算顺序匹配源布局	读取时变换	读取连续性好
DRAM → SRAM 预取	Memory Copy 融合	隐藏延迟

14 实现 Layout Transformation 的硬件单元

布局变换的高效实现需要专用硬件支持。本节介绍实现布局变换的核心硬件单元。

14.1 地址生成单元（Address Generation Unit, AGU）

AGU 是实现布局变换的核心硬件。它根据逻辑索引计算物理内存地址。

14.1.1 基本架构

Listing 11: AGU 硬件模型

```

1 class AddressGenerationUnit:
2     """
3         地址生成单元的硬件模型
4
5         功能：根据多维索引和步长计算内存地址
6
7         硬件实现：
8             - 多个乘法器（index × stride）
9             - 加法树（累加各维度贡献）
10            - 基地址加法器
11
12        """
13
14    def __init__(self, num_dims=4):
15        self.num_dims = num_dims
16        # 配置寄存器
17        self.base_addr = 0
18        self.strides = [0] * num_dims
19        self.bounds = [0] * num_dims # 各维度的范围
20
21    def configure(self, base_addr, strides, bounds):
22        """
23            配置 AGU 参数
24
25            在算子执行前由控制器配置
26        """
27        self.base_addr = base_addr
28        self.strides = strides
29        self.bounds = bounds
30
31    def compute_address(self, indices):
32        """
33            计算地址（硬件并行执行）
34
35            addr = base + sum(index[d] * stride[d])
36
37            硬件延迟：1-2 个时钟周期
38        """
39        offset = 0
40        for d in range(self.num_dims):
41            # 这里的乘法在硬件中并行执行
42            offset += indices[d] * self.strides[d]

```

```
43     return self.base_addr + offset
```

14.1.2 支持布局变换的 AGU 设计

标准 AGU 只需通过配置不同的步长即可支持任意布局:

Listing 12: AGU 配置示例

```
1 # 示例: (1, 64, 56, 56) 张量
2
3 # NCHW 布局配置
4 agu.configure(
5     base_addr=0,
6     strides=[64*56*56, 56*56, 56, 1], # N, C, H, W
7     bounds=[1, 64, 56, 56]
8 )
9
10 # NHWC 布局配置 (相同硬件, 不同参数)
11 agu.configure(
12     base_addr=0,
13     strides=[56*56*64, 1, 56*64, 64], # N, C, H, W 的步长
14     bounds=[1, 64, 56, 56]
15 )
16
17 # 关键: 布局变换只是步长的重新配置, 无需额外硬件
```

14.2 DMA 控制器 (Direct Memory Access Controller)

DMA 控制器负责在内存层次之间搬移数据, 是实现融合布局变换的理想位置。

14.2.1 支持布局变换的 DMA 设计

Listing 13: 支持布局变换的 DMA 控制器

```
1 class LayoutAwareDMA:
2     """
3         支持布局变换的 DMA 控制器
4
5             在数据搬移过程中完成布局变换, 无需额外的 DRAM 访问
6     """
7
8     def __init__(self, read_agu, write_agu, buffer_size=256):
9         self.read_agu = read_agu      # 读地址生成单元
10        self.write_agu = write_agu   # 写地址生成单元
11        self.buffer_size = buffer_size # 内部缓冲区大小
12
```

```

13 def configure_fused_copy(self, shape, src_layout, dst_layout,
14                             src_base, dst_base):
15     """
16     配置融合布局变换的 DMA 传输
17     """
18     # 配置读 AGU (按源布局)
19     src_strides = compute_strides(shape, src_layout)
20     self.read_agu.configure(src_base, src_strides, shape)
21
22     # 配置写 AGU (按目标布局)
23     dst_strides = compute_strides(shape, dst_layout)
24     self.write_agu.configure(dst_base, dst_strides, shape)
25
26 def execute_transfer(self, total_elements):
27     """
28     执行数据传输
29
30     硬件实现：
31     1. 读 AGU 生成读地址，发起读请求
32     2. 数据通过内部缓冲区
33     3. 写 AGU 生成写地址，发起写请求
34
35     读写可以流水线执行
36     """
37     for i in range(0, total_elements, self.buffer_size):
38         batch_size = min(self.buffer_size, total_elements - i)
39
40         # 并行生成读写地址
41         read_addrs = [self.read_agu.compute_address(
42                         self.index_from_linear(i + j)) for j in range(batch_size)]
43         write_addrs = [self.write_agu.compute_address(
44                         self.index_from_linear(i + j)) for j in range(batch_size)]
45
46         # 执行读取
47         data = [memory_read(addr) for addr in read_addrs]
48
49         # 执行写入 (数据被重新排列)
50         for j, addr in enumerate(write_addrs):
51             memory_write(addr, data[j])

```

14.3 Scatter-Gather Engine

对于复杂的非连续访问模式，Scatter-Gather 引擎提供更高效的实现：

Listing 14: Scatter-Gather 引擎


```

46     def fused_layout_transform(self, shape, src_layout, dst_layout,
47                               src_base, dst_base):
48         """
49             使用 Gather-Process-Scatter 模式实现布局变换
50         """
51
52         src_strides = compute_strides(shape, src_layout)
53         dst_strides = compute_strides(shape, dst_layout)
54
55         total_elements = prod(shape)
56         buffer = [0] * self.vector_width
57
58         for batch_start in range(0, total_elements, self.vector_width):
59             batch_size = min(self.vector_width, total_elements -
60                               batch_start)
61
62             # 生成源地址 (Gather)
63             src_addrs = []
64             for i in range(batch_size):
65                 idx = self.linear_to_multi_index(batch_start + i, shape)
66                 src_addrs.append(src_base + sum(
67                     idx[d] * src_strides[d] for d in range(len(shape))))
68
69             # Gather
70             self.gather(src_addrs, buffer)
71
72             # 生成目标地址 (Scatter)
73             dst_addrs = []
74             for i in range(batch_size):
75                 idx = self.linear_to_multi_index(batch_start + i, shape)
76                 dst_addrs.append(dst_base + sum(
77                     idx[d] * dst_strides[d] for d in range(len(shape))))
78
79             # Scatter
80             self.scatter(buffer, dst_addrs)

```

14.4 片上网络 (Network-on-Chip, NoC) 支持

在多核加速器中，NoC 也可以参与布局变换：

Listing 15: NoC 支持的布局变换

```

1 class NoCLayoutTransform:
2     """
3         利用片上网络实现分布式布局变换
4
5         适用于数据分布在多个 PE 的场景

```

```
6      """
7
8      def __init__(self, num_pes, noc_topology):
9          self.num_pes = num_pes
10         self.noc_topology = noc_topology
11
12     def distributed_transpose(self, local_data, src_partition,
13                               dst_partition):
14         """
15         分布式转置
16
17         每个 PE 持有数据的一部分，通过 NoC 通信完成全局转置
18
19         Args:
20             local_data: 本地数据块
21             src_partition: 源分区方式
22             dst_partition: 目标分区方式
23
24         """
25         my_pe_id = get_current_pe_id()
26
27         # 计算需要发送给其他 PE 的数据
28         send_buffers = {}
29         for target_pe in range(self.num_pes):
30             if target_pe != my_pe_id:
31                 # 确定需要发送的数据子集
32                 data_to_send = self.compute_send_data(
33                     local_data, my_pe_id, target_pe,
34                     src_partition, dst_partition)
35                 send_buffers[target_pe] = data_to_send
36
37         # 全交换通信 (All-to-All)
38         recv_buffers = noc_all_to_all(send_buffers)
39
40         # 组装本地结果
41         result = self.assemble_local_result(
42             local_data, recv_buffers, dst_partition)
43
44         return result
```

14.5 硬件设计总结

硬件单元	功能	适用场景	设计复杂度
AGU	地址计算	所有场景	低
DMA + 双 AGU	融合搬移	DRAM \leftrightarrow SRAM	中
Scatter-Gather	非连续访问	复杂布局变换	中-高
NoC	分布式变换	多核加速器	高

设计原则:

1. AGU 需要支持可配置步长, 这是实现布局变换的基础
2. DMA 控制器应配备独立的读写 AGU, 支持融合布局变换
3. 对于高性能需求, 可以增加 Scatter-Gather 能力
4. 大规模并行系统需要考虑 NoC 层面的数据重分布

15 2 级 Tiling 的 Data Layout 与 Access 分析

现代加速器采用多级存储层次结构 (DRAM \rightarrow SRAM \rightarrow Register), 相应地采用 2 级 Tiling 策略。本节分析数据布局与 2 级访问顺序的匹配问题。

15.1 存储层次与 Tiling 层次

典型的加速器存储层次:

存储层	容量	带宽	能耗
DRAM	GB 级	100 GB/s	高
SRAM (片上)	KB-MB 级	500+ GB/s	中
Register File	KB 级	TB/s 级	低

对应的 2 级 Tiling:

- **L1 Tiling:** 将完整 tensor 划分为 tiles, 从 DRAM 搬到 SRAM
- **L2 Tiling:** 将 tile 划分为 sub-tiles, 从 SRAM 搬到 Register

15.2 不匹配发生的两个层次

15.2.1 L1 层次: Data Layout vs L1 Tile Order

L1 Tiling 决定了从 DRAM 读取 tiles 的顺序。如果遍历顺序与数据布局不匹配, 会导致 DRAM 访问非连续:

DRAM 中的 tensor (NCHW 布局):

```
stride_C = H × W = 3136
stride_H = W = 56
stride_W = 1 (W 连续)
```

L1 Tile 遍历：

W_H_C 顺序：先遍历 W 方向的 tiles → stride=1 → 连续访问

C_W_H 顺序：先遍历 C 方向的 tiles → stride=3136 → 非连续

15.2.2 L2 层次：Data Layout vs L2 Sub-tile Order

在 SRAM 中，tile 保持与 DRAM 相同的布局。L2 Tiling 决定了 tile 内部访问的顺序：

SRAM 中的 tile (NCHW 布局, tc=16, th=14, tw=14)：

```
stride_C = th × tw = 196
stride_H = tw = 14
stride_W = 1
```

L2 Sub-tile 遍历：

W_H_C 顺序：先遍历 W 方向 → stride=1 → 连续访问

C_W_H 顺序：先遍历 C 方向 → stride=196 → 非连续

15.3 匹配规则

核心匹配规则：

Layout 的连续维度 = Access Order 的最内层维度 → 匹配

具体对应关系：

Layout	连续维度	最优 Access Order
NCHW	W	W_H_C (W 在最内层)
NHWC	C	C_W_H (C 在最内层)

15.4 效率量化分析

访问效率与 stride 的关系：

$$\eta = \begin{cases} 0.9 & \text{stride} = 1 (\text{连续访问}) \\ \frac{\text{element_size}}{\text{stride} \times \text{element_size}} & \text{stride} > 1 (\text{非连续}) \end{cases} \quad (8)$$

15.4.1 实验数据

以 (1, 64, 56, 56) tensor, L1 Tile (16, 14, 14), L2 Sub-tile (4, 7, 7) 为例：

配置	L1 匹配	L1 效率	L2 匹配	L2 效率	总代价
NCHW + W_H_C + W_H_C	0.90		0.90		16.1 μ s
NCHW + C_W_H + C_W_H	0.06		0.25		154.1 μ s
NHWC + C_W_H + C_W_H	0.90		0.90		16.1 μ s
NHWC + W_H_C + W_H_C	0.06		0.25		154.1 μ s

关键结论：全匹配 vs 全不匹配的性能差异可达 **9.6 倍**。

15.5 设计指导

1. **两级都要匹配：** L1 和 L2 的 Access Order 都应与 Layout 一致
2. **L1 优先：** 如果必须不匹配，优先保证 L1 匹配（DRAM 带宽是瓶颈）
3. **与 Dataflow 配合：**
 - Output Stationary → 倾向于 C 方向遍历 → NHWC + C_W_H
 - Weight Stationary → 倾向于 spatial 遍历 → NCHW + W_H_C
4. **保持一致：** 两级 Tiling 的 Order 应保持一致，简化控制逻辑

16 完整代码示例：端到端的布局变换

本节给出一个完整的代码示例，展示从分析到执行的完整流程。

16.1 场景设定

考虑以下神经网络片段：

Conv1 (NCHW output) → ReLU → Conv2 (expects NHWC input)

需要在 Conv1 和 Conv2 之间完成 NCHW → NHWC 的布局变换。

16.2 完整实现

Listing 16: 端到端布局变换示例

```

1 """
2 完整的布局变换实现示例
3
4 演示从分析、决策到执行的完整流程
5 """
6
7 from dataclasses import dataclass
8 from typing import Tuple, Dict
9 from enum import Enum
10 import math
11
12
13 # =====#
14 # 第一步：定义数据结构
15 # =====#
16
17 @dataclass
18 class TensorInfo:
19     """张量信息"""

```

```
20     name: str
21     shape: Tuple[int, ...]
22     layout: Tuple[int, ...] # 维度存储顺序
23
24
25 @dataclass
26 class OperatorInfo:
27     """算子信息"""
28     name: str
29     op_type: str
30     input_layout_preference: Tuple[int, ...]
31     output_layout: Tuple[int, ...]
32     compute_order: Tuple[int, ...]
33
34
35 # =====
36 # 第二步：布局分析
37 # =====
38
39 def analyze_layout_mismatch(producer: OperatorInfo,
40                             consumer: OperatorInfo,
41                             tensor_shape: Tuple[int, ...]):
42     """
43         分析两个算子之间的布局不匹配情况
44     """
45     src_layout = producer.output_layout
46     dst_layout = consumer.input_layout_preference
47
48     if src_layout == dst_layout:
49         return {
50             "mismatch": False,
51             "message": "Layouts match, no transformation needed"
52         }
53
54     # 计算变换的数据量
55     data_size = math.prod(tensor_shape) * 4 # float32
56
57     return {
58         "mismatch": True,
59         "source_layout": src_layout,
60         "target_layout": dst_layout,
61         "tensor_shape": tensor_shape,
62         "data_size_bytes": data_size,
63         "transform_type": "TRANSPOSE" if len(src_layout) ==
64             len(dst_layout) else "RESHAPE"
```

```
64     }
65
66
67 # =====
68 # 第三步：选择变换策略
69 # =====
70
71 class TransformStrategy(Enum):
72     PRODUCER_WRITE = "producer_write"
73     CONSUMER_READ = "consumer_read"
74     MEMORY_COPY_FUSED = "memory_copy_fused"
75
76
77 def select_transform_strategy(analysis_result: Dict,
78                             producer: OperatorInfo,
79                             consumer: OperatorInfo) ->
80                     TransformStrategy:
81     """
82     选择最优的变换策略
83     """
84
85     if not analysis_result["mismatch"]:
86         return None
87
88     src_layout = analysis_result["source_layout"]
89     dst_layout = analysis_result["target_layout"]
90
91     # 计算各策略的代价
92
93     # 策略 1: Producer 写入时变换
94     # 检查 producer 的计算顺序是否与目标布局兼容
95     producer_write_cost = compute_address_generation_cost(
96         producer.compute_order, dst_layout)
97
98     # 策略 2: Consumer 读取时变换
99     consumer_read_cost = compute_address_generation_cost(
100        consumer.compute_order, src_layout)
101
102     # 策略 3: Memory Copy 融合
103     # 假设总是可行，代价取决于数据搬移本身
104     memcpy_cost = 1.0  # 归一化代价
105
106     # 选择最小代价的策略
107     costs = [
108         (TransformStrategy.PRODUCER_WRITE, producer_write_cost),
109         (TransformStrategy.CONSUMER_READ, consumer_read_cost),
```

```
108         (TransformStrategy.MEMORY_COPY_FUSED, memcopy_cost),
109     ]
110
111     best = min(costs, key=lambda x: x[1])
112     return best[0]
113
114
115 def compute_address_generation_cost(compute_order: Tuple[int, ...],
116                                       layout: Tuple[int, ...]) -> float:
117     """
118     计算地址生成的代价
119
120     如果计算顺序与布局匹配（最内层循环对应最小步长），
121     则访问是连续的，代价低
122     """
123
124     # 检查最内层循环维度
125     innermost_compute_dim = compute_order[-1]
126
127     # 找到布局中最小步长的维度
128     layout_list = list(layout)
129     innermost_storage_dim = layout_list[-1] # 最后一个是最内层存储
130
131     if innermost_compute_dim == innermost_storage_dim:
132         return 0.5 # 连续访问，代价低
133     else:
134         return 1.5 # 非连续访问，代价高
135
136 # =====
137 # 第四步：生成地址配置
138 # =====
139
140 def generate_address_config(strategy: TransformStrategy,
141                             tensor_shape: Tuple[int, ...],
142                             src_layout: Tuple[int, ...],
143                             dst_layout: Tuple[int, ...],
144                             producer: OperatorInfo,
145                             consumer: OperatorInfo) -> Dict:
146     """
147     根据策略生成地址配置
148     """
149
150     def compute_strides(shape, layout_order):
151         """计算指定布局的步长"""
152         physical_shape = [shape[layout_order[i]] for i in
```

```
        range(len(shape))]

153    physical_strides = [1] * len(shape)
154    for i in range(len(shape) - 2, -1, -1):
155        physical_strides[i] = physical_strides[i + 1] *
156            physical_shape[i + 1]

157    logical_strides = [0] * len(shape)
158    for physical_dim, logical_dim in enumerate(layout_order):
159        logical_strides[logical_dim] = physical_strides[physical_dim]
160
161    return tuple(logical_strides)

162
163 config = {"strategy": strategy.value}

164
165 if strategy == TransformStrategy.PRODUCER_WRITE:
166     # Producer 使用目标布局的步长写入
167     config["producer_write_strides"] = compute_strides(tensor_shape,
168               dst_layout)
169     config["consumer_read_strides"] = compute_strides(tensor_shape,
170               dst_layout)
171     config["stored_layout"] = dst_layout

172 elif strategy == TransformStrategy.CONSUMER_READ:
173     # Consumer 使用源布局的步长读取
174     config["producer_write_strides"] = compute_strides(tensor_shape,
175               src_layout)
176     config["consumer_read_strides"] = compute_strides(tensor_shape,
177               src_layout)
178     config["stored_layout"] = src_layout

179 elif strategy == TransformStrategy.MEMORY_COPY_FUSED:
180     # DMA 读写使用不同步长
181     config["dma_read_strides"] = compute_strides(tensor_shape,
182               src_layout)
183     config["dma_write_strides"] = compute_strides(tensor_shape,
184               dst_layout)
185     config["producer_write_strides"] = compute_strides(tensor_shape,
186               src_layout)
187     config["consumer_read_strides"] = compute_strides(tensor_shape,
188               dst_layout)

189
190 return config

191
192
193 # =====
```

```
188 # 第五步：执行演示
189 # =====
190
191 def demo_layout_transform():
192     """
193     演示完整的布局变换流程
194     """
195     print("=" * 60)
196     print("Layout Transformation 端到端演示")
197     print("=" * 60)
198
199     # 定义算子
200     conv1 = OperatorInfo(
201         name="conv1",
202         op_type="Conv",
203         input_layout_preference=(0, 1, 2, 3),    # NCHW
204         output_layout=(0, 1, 2, 3),                # NCHW
205         compute_order=(0, 1, 2, 3)                 # 按 N, C, H, W 顺序计算
206     )
207
208     conv2 = OperatorInfo(
209         name="conv2",
210         op_type="Conv",
211         input_layout_preference=(0, 2, 3, 1),    # NHWC
212         output_layout=(0, 2, 3, 1),                # NHWC
213         compute_order=(0, 2, 3, 1)                 # 按 N, H, W, C 顺序计算
214     )
215
216     tensor_shape = (1, 64, 56, 56)    # 中间特征图
217
218     print(f"\n场景: {conv1.name} -> {conv2.name}")
219     print(f"张量形状: {tensor_shape}")
220     print(f"数据大小: {math.prod(tensor_shape) * 4 / 1024:.1f} KB")
221
222     # 步骤 1: 分析布局不匹配
223     print("\n--- 步骤 1: 布局分析 ---")
224     analysis = analyze_layout_mismatch(conv1, conv2, tensor_shape)
225     print(f"布局不匹配: {analysis['mismatch']}")
226     if analysis['mismatch']:
227         print(f"  源布局 (NCHW): {analysis['source_layout']}")
228         print(f"  目标布局 (NHWC): {analysis['target_layout']}")
229
230     # 步骤 2: 选择变换策略
231     print("\n--- 步骤 2: 策略选择 ---")
232     strategy = select_transform_strategy(analysis, conv1, conv2)
```

```
233     print(f"选择策略: {strategy.value}")
234
235     # 步骤 3: 生成地址配置
236     print("\n--- 步骤 3: 地址配置 ---")
237     config = generate_address_config(
238         strategy, tensor_shape,
239         analysis['source_layout'], analysis['target_layout'],
240         conv1, conv2
241     )
242
243     for key, value in config.items():
244         print(f"  {key}: {value}")
245
246     # 步骤 4: 显示具体的地址计算示例
247     print("\n--- 步骤 4: 地址计算示例 ---")
248
249     # 计算索引 (0, 32, 28, 28) 的地址
250     test_index = (0, 32, 28, 28)
251
252     nchw_strides = (64*56*56, 56*56, 56, 1)
253     nhwc_strides = (56*56*64, 64, 56*64, 1)
254
255     nchw_addr = sum(i * s for i, s in zip(test_index, nchw_strides))
256     nhwc_addr = sum(i * s for i, s in zip(test_index, nhwc_strides))
257
258     print(f"  测试索引 (n=0, c=32, h=28, w=28):")
259     print(f"    NCHW 地址: {nchw_addr} (offset)")
260     print(f"    NHWC 地址: {nhwc_addr} (offset)")
261     print(f"    地址差异: {abs(nhwc_addr - nchw_addr)}")
262
263     print("\n" + "=" * 60)
264     print("演示完成")
265     print("=" * 60)
266
267
268 if __name__ == "__main__":
269     demo_layout_transform()
```

16.3 运行结果

```
=====
Layout Transformation 端到端演示
=====
```

场景: conv1 -> conv2

张量形状: (1, 64, 56, 56)

数据大小: 784.0 KB

--- 步骤 1: 布局分析 ---

布局不匹配: True

源布局 (NCHW): (0, 1, 2, 3)

目标布局 (NHWC): (0, 2, 3, 1)

--- 步骤 2: 策略选择 ---

选择策略: memory_copy_fused

--- 步骤 3: 地址配置 ---

strategy: memory_copy_fused

dma_read_strides: (200704, 3136, 56, 1)

dma_write_strides: (200704, 1, 3584, 64)

producer_write_strides: (200704, 3136, 56, 1)

consumer_read_strides: (200704, 1, 3584, 64)

--- 步骤 4: 地址计算示例 ---

测试索引 (n=0, c=32, h=28, w=28):

NCHW 地址: 101948 (offset)

NHWC 地址: 101920 (offset)

地址差异: 28

=====

演示完成

=====

17 总结

本文详细讨论了神经网络加速器中数据布局变换的四个核心问题:

17.1 变换对象: 权重 vs 激活值

- 权重: 静态数据, 应在离线阶段完成布局变换, 零运行时开销
- 激活值: 动态数据, 需要在运行时处理, 应通过融合优化来降低开销

17.2 算子融合策略

- 与 Producer 融合: 写入时按目标布局组织, 适合单 Producer 多 Consumer

- 与 **Consumer** 融合：读取时适配源布局，适合多 Producer 单 Consumer
- 与 **Memory Copy** 融合：利用必需的数据搬移，隐藏变换开销
- **Elementwise 算子**：布局不敏感，是插入变换的理想位置

17.3 读取 vs 写入时机

- **写入时变换**：数据一次变换，多次使用；要求 Producer 知道下游需求
- **读取时变换**：灵活适配不同源布局；可能导致非连续读取
- 选择取决于计算顺序与布局的匹配度

17.4 硬件实现

- **AGU**（地址生成单元）：核心组件，通过可配置步长支持任意布局
- **DMA 控制器**：配备独立读写 AGU，实现融合布局变换
- **Scatter-Gather 引擎**：处理复杂非连续访问模式
- **NoC**：大规模并行系统中的分布式数据重排

核心设计原则：

1. 避免单独的布局变换算子 ($\text{DRAM} \rightarrow \text{DRAM}$)，这会带来 $2\times$ 的额外内存访问
2. 利用必需的数据搬移来隐藏布局变换开销
3. **AGU** 的步长可配置性是实现灵活布局支持的基础
4. 编译期优化可以消除大部分运行时布局变换需求