深度神经网络加速器的自动化数据布局传播框架

GitHub Copilot

2026 年 1 月 4 日

1 问题定义 (Problem Definition)

在深度神经网络加速器中，当生产者算子的输出数据组织形式与消费者算子的最佳输入形式不匹配时，必须进行 ** 布局转换（Layout Transformation）**。本文关注的核心问题并非简单的逻辑布局选择，而是如何从物理实现层面以最小的代价完成这一转换。

无论是何种实现方式，底层硬件都需要依赖 ** 地址生成单元（AGU）** 来计算非连续的物理地址。因此，AGU 的存在是前提，而非区分不同策略的特征。真正的决策在于转换发生的 ** 存储层级 ** 与 ** 时机 **：

1. ** 显式布局转换（Explicit Layout Transformation / DRAM-to-DRAM）**：这种方式将布局转换视为一个独立的、显式的 ** 内存拷贝（Memory Copy）** 任务。
   - ** 实现机制 **：系统插入一个专门的 ‘Reorder‘ 或 ‘Transpose‘ 算子，将数据从 DRAM 读取，经过重排后写回 DRAM 的另一块区域。
   - ** 代价 **：这会产生额外的内存流量（Memory Traffic），即增加了显式的读写带宽消耗。
   - ** 优势 **：解耦了生产者和消费者，使得后续的消费者可以获得完美的连续内存访问模式。

   （注：虽然部分加速器支持片上 SRAM-to-SRAM 的显式重排，但在处理大规模 DNN 时，受限于片上容量，DRAM-to-DRAM 的重排仍是主要瓶颈，因此本文主要关注后者。）

2. ** 隐式布局转换（Implicit Layout Transformation / Cross-Hierarchy）**：这种方式将布局转换“融合”在不同存储层级之间的数据传输过程中（例如 SRAM $\leftrightarrow$ DRAM）。
   - ** 实现机制 **：不增加额外的 DRAM 读写操作，而是利用 AGU 在数据离开或进入片上存储（SRAM）时即时进行地址重映射。
   - ** 场景 **：
     - **Transform-on-Write**：生产者在将 SRAM 中的计算结果写入 DRAM 时，按消费者需要的顺序离散写入。
     - **Transform-on-Read**：消费者在从 DRAM 读取数据到 SRAM 时，按自身需要的顺序离散读取。
   - ** 代价 **：虽然没有增加数据传输量，但会导致 DRAM 侧出现复杂的跨步（Strided）访问模式，引发严重的行缓冲区未命中（Row-buffer Miss），降低有效带宽利用率。

因此，本文旨在解决的问题是：** 在全网范围内，针对每一处布局不匹配，应选择支付“额外的带宽开销”（显式转换），还是支付“降低的访问效率”（隐式转换）？** 这是一个在数据传输量与 DRAM 访问效率之间寻找全局平衡点的优化问题。

2 系统建模 (System Modeling)

2.1 数据布局表示与搜索空间 (Data Layout Representation & Search Space)

在我们的编译框架中，数据布局并非仅限于简单的 NCHW 或 NHWC 排列，而是由底层的 ** 映射策略（Map Strategy）** 决定的。一个完整的布局描述包含以下三个维度的自由度，构成了一个巨大的搜索空间：

1. ** 维度切分与重排（Tiling & Permutation）**：逻辑维度（如 C）可以被切分为多级物理维度（如 $C \to C_{out}, C_{in}$），并与其他维度任意交织。例如，为了适配脉动阵列的脉动数据流，可能需要 $[N, C_{out}, H, W, C_{in}]$ 这样的复杂嵌套结构。
2. ** 循环分块因子（Blocking Factors）**：每一级切分的大小（如 $C_{in} = 16$ vs $C_{in} = 32$）直接决定了数据的粒度是否与硬件的向量宽度或 Burst Length 匹配。
3. ** 存储层级映射（Memory Hierarchy Mapping）**：数据在 Global Buffer、Register File 等不同层级上的驻留方式也会隐式地影响其在 DRAM 中的理想布局。

因此，本文所述的“布局 L”实际上是上述所有映射参数的一个封装。在生成候选集时，我们利用现有的单层优化器（Single-layer Solver）在上述空间中进行搜索，为每个算子找到其局部最优的复杂布局配置，而非仅仅从预设的几个模板中选择。

设逻辑维度集合为 $\mathcal{D}_{logical} = \{N, C, H, W\}$。一个物理布局 L 可以形式化表示为物理维度的有序列表：

$$L = [d_1, d_2, \ldots, d_k] \tag{1}$$

其中 d_i 是带有具体切分大小的物理维度。例如，$[N, C_{out}, H, W, C_{in}^{16}]$ 表示通道维度被切分，且最内层大小为 16。这种细粒度的表示能力使我们能够捕捉到微小的布局差异对 DRAM Row Hit Rate 的影响。

2.2 代价模型 (Cost Model)

为了量化不同布局策略的优劣，我们定义总开销 $Cost_{total}$ 为网络中所有节点的执行开销与所有边的转换开销之和：

$$Cost_{total} = \sum_{v \in V} Cost_{exec}(v, L_v) + \sum_{(u,v) \in E} Cost_{trans}(L_u, L_v) \tag{2}$$

2.2.1 执行开销 (Execution Cost)

执行开销衡量了算子 v 在采用布局 L_v 时访问内存的效率。我们将 DRAM 访问的总延迟建模为两部分之和：

- **突发传输开销 (Burst Cost)**：反映了数据传输的有效带宽占用。DRAM 通过突发模式（Burst Mode）传输数据（例如 Burst Length = 8）。此开销与访问的总数据量成正比，代表了在理想连续访问情况下的基准延迟。

- **行激活开销 (Row Activation Cost)**：反映了由于访问局部性差而引入的额外延迟。当连续的内存请求落在不同的 DRAM 行（Row）时，控制器必须发出 Precharge 和 Activate 命令来打开新行。

总执行开销公式为：

$$Cost_{exec} = C_{burst} \times N_{req} + C_{act} \times N_{row\_miss} \tag{3}$$

其中 N_{req} 是总请求次数（数据量/总线宽度），N_{row_miss} 是行缓冲区未命中次数。当布局与算子访问模式不匹配（如 Strided Access）时，N_{row_miss} 会显著增加，主导总开销。

2.2.2 转换开销 (Transition Cost)

当生产者节点 u 的输出布局 L_u 与消费者节点 v 的输入布局 L_v 不一致时，需要插入数据重排操作。我们考虑两种策略：

- **直接写入 (Direct Write)**：生产者直接按其自然顺序（例如计算顺序）写入内存，消费者按非优顺序读取。此时显式的转换开销为 0，但消费者承担高额的读取执行开销（因为读取模式是 Strided 的）。
- **写入时变换 (Transform-on-Write)**：生产者在写入内存前，利用片上 SRAM 缓冲区对数据进行重排，使其符合消费者的偏好。这增加了生产者的写入开销（需要复杂的地址计算和可能的 SRAM 冲突），但降低了消费者的读取开销（消费者可以顺序读取）。

我们的策略选择器（Strategy Selector）会自动比较这两种策略，选择总代价最小的方案。

3 布局传播算法 (Layout Propagation Algorithm)

我们的优化过程分为两个阶段：候选布局传播（Candidate Propagation）和最优布局选择（Optimal Selection）。

3.1 阶段一：候选布局传播 (Candidate Propagation)

此阶段的目标是生成每个节点的“候选布局集”。在此过程中，我们针对 ** 静态权重（Weights）** 与 ** 动态激活（Activations）** 采取了不同的处理策略：

- **权重（Weights）**：作为编译期已知的常量，权重可以被离线重排为任意最优布局，而无需支付运行时的转换代价。因此，权重节点总是作为布局约束的“强发起者”，将其对计算核心（如脉动阵列）最友好的布局无条件地传播给相连的算子。
- **激活（Activations）**：作为运行时生成的中间数据，其布局转换会产生显式的延迟和带宽消耗。因此，激活张量的布局传播是一个双向协商过程，需要在生产者的写入效率与消费者的读取效率之间寻找平衡。

基于上述区分，传播过程如下：

1. **识别源节点**：首先识别图中对布局“敏感”的节点（如卷积层）。这些节点根据其硬件特性生成初始的**首选布局**（Preferred Layouts）。例如，一个 3×3 卷积层可能首选 $NCHW$。

2. **波前传播：**将这些首选布局作为“波前”向邻居节点传播。如果节点 u 偏好布局 L，它会告诉其邻居 v：“如果你也能使用布局 L，我们之间就没有转换开销”。

3. **不敏感节点适配：**对于 ReLU、Add 等对布局“不敏感”的节点，它们不生成新布局，而是收集并通过所有邻居传来的布局。这使得布局约束可以穿过这些节点，影响更远的算子。例如，ResNet 中的残差连接（Add）会将主分支的布局约束传播到跳跃连接（Skip Connection）分支。

Algorithm 1 布局候选传播 (Layout Candidate Propagation)

```
procedure PROPAGATE(G)
    Queue ← G 中所有敏感节点
    while Queue 不为空 do
        u ← Queue.pop()
        for u 的每一个邻居 v do
            NewCandidates ← ∅
            for u 候选集中的每一个布局 L do
                if L 与 v 的维度兼容 then
                    NewCandidates.add(L)
                end if
            end for
            if NewCandidates 中包含 v 尚未拥有的布局 then
                v.candidates ← v.candidates ∪ NewCandidates
                Queue.push(v)                                  ▷ 继续传播
            end if
        end for
    end while
end procedure
```

3.2 阶段二：最优布局选择 (Optimal Selection)

在获得每个节点的候选集后，我们需要做出最终决策。这是一个动态规划问题，但在有环图（如 RNN）中是 NP-hard 的。我们采用基于拓扑排序的贪心策略（对于有环图，先打破回边）。

对于每个节点 v，我们遍历其所有候选布局 L，计算局部代价：

$$LocalCost(v, L) = Cost_{exec}(v, L) + \sum_{u \in Predecessors(v)} \min\_strategy(L_u, L) \tag{4}$$

其中 $\min\_strategy$ 会在“直接写入”和“变换写入”之间选择代价较小者。我们选择使 $LocalCost$ 最小的 L 作为节点 v 的最终布局。

4 实验评估 (Evaluation)

我们在多个经典神经网络模型上评估了该框架。对比基准为“Linear”策略（强制所有张量使用 NCHW 布局）。实验环境模拟了一个具有 16 个 Bank 的 DRAM 系统，行缓冲区大小为 2KB。

Algorithm 2 最优布局选择 (Layout Selection)

```
procedure SelectLayouts(G)
    for 按拓扑序遍历每个节点 v do
        BestLayout ← None
        MinTotalCost ← ∞
        for v 候选集中的每一个布局 L do
            CurrentCost ← EvaluateExecutionCost(v, L)
            for v 的每一个前驱 u do
                PrevLayout ← u.selected_layout
                TransCost ← EvaluateTransition(PrevLayout, L)
                CurrentCost ← CurrentCost + TransCost
            end for
            if CurrentCost < MinTotalCost then
                MinTotalCost ← CurrentCost
                BestLayout ← L
            end if
        end for
        v.selected_layout ← BestLayout
    end for
end procedure
```

4.1 实验结果

表 1 展示了不同网络在两种策略下的归一化总开销。

表 1: 不同网络的布局优化结果对比

Network	Linear Cost (NCHW)	Optimized Cost	Improvement
ResNet50	111.75	57.12	**48.9%**
ResNet152	315.75	163.12	**48.3%**
VGG19	22.81	22.25	2.5%
BERT-Base	10.00	10.00	0.0%
LSTM (Phoneme)	10.53	9.00	14.5%

4.2 结果分析

- **ResNet50 / ResNet152：**
  - **结果：**总开销降低约 **49%**。
  - **分析：**ResNet 包含大量的残差块（Residual Blocks）。在 Linear 策略中，分支的汇合点（Add 操作）往往因为输入布局不一致或非对齐导致高额开销。我们的算法成功识别出整个残差块应

统一使用分块布局（Blocked Layout），消除了块内部的所有转换开销。此外，对于 1×1 卷积，分块布局显著提高了通道维度的访问局部性。

- **BERT-Base：**
  - **结果：**开销持平。
  - **分析：**我们将 BERT 中的线性层建模为 1×1 卷积（序列长度 128）。在此特定维度下，默认的 NCHW 布局与分块布局在行缓冲区命中率上差异不大。这表明对于某些计算密集型且维度规则的算子，布局优化的边际效应递减。如果序列长度增加或 Batch Size 增大，优化效果可能会显现。
- **LSTM (Phoneme)：**
  - **结果：**开销降低约 **15%**。
  - **分析：**LSTM 包含复杂的门控机制和逐元素操作。布局传播算法有效地将矩阵乘法（FC）偏好的布局传播到了后续的 Sigmoid/Tanh 激活函数中，避免了中间的重排。

5 结论 (Conclusion)

本文提出的布局传播框架通过显式建模执行与转换开销，成功解决了深度学习编译器中的布局选择难题。实验证明，该方法能够自动发现并利用网络拓扑中的结构特征（如残差连接），在保证算子执行效率的同时，最小化全局数据移动开销。未来的工作将集中在支持更复杂的内存层次结构（如多级缓存）以及与多面体编译技术的结合。