PIM-DLS: 面向存内计算架构的自动化数据流优化框架

Co-Author

2026 年 1 月 2 日

1 问题形式化

我们致力于解决在基于 DRAM 的存内计算 (PIM) 架构上优化卷积神经网络 (CNN) 数据流映射的问题。优化目标是通过确定最佳的循环分块 (Loop Tiling)、循环排序 (Loop Ordering) 和数据放置策略，来最小化端到端推理延迟（或能耗）。

1.1 系统模型

架构模型 ($\mathcal{A}$): 硬件被建模为一个具有 $M+1$ 层的分层存储系统，索引从 $m=0$ 到 M。

- **计算层级 ($m=0$)**: 代表 PIM 处理单元 (PE) 阵列。我们的模型支持丰富的计算单元类型，包括标量 (Scalar)、SIMD、张量核心 (Tensor Core)、归约树 (Reduction Tree) 以及脉动阵列 (Systolic Array)。每个 PE 的特征由其内部并行度（如 MAC 数量）、归约深度和能耗特性定义。
- **存储层级 ($m>0$)**: 为了精确研究行缓冲区的复用问题，我们将存储层次细分为：
  - **L1 ($m=1$)**: 全局缓冲区 (Global Buffer)，用于片上数据共享。
  - **L2 ($m=2$)**: 行缓冲区 (Row Buffer)，作为 DRAM 的高速缓存行，是行激活优化的关键层级。
  - **L3 ($m=3$)**: DRAM 存储单元 (DRAM Cell)，代表片外大容量存储。

  每一层由其容量 C_m、读写带宽 BW_m 和旁路 (Bypass) 能力定义。

工作负载模型 ($\mathcal{W}$): 一个 CNN 层由 7 维循环嵌套定义：Batch (N)、输出通道 (K)、输入通道 (C)、输出高度 (P)、输出宽度 (Q)、卷积核高度 (R) 和卷积核宽度 (S)。工作负载参数还包括步长 (str) 和膨胀系数 (dil)。

1.2 优化问题

给定 $\mathcal{A}$ 和 $\mathcal{W}$，问题在于找到一个映射配置 $\mathcal{M}$，使得成本函数最小化：

$$\min_{\mathcal{M}} \text{Cost}(\mathcal{M}|\mathcal{A}, \mathcal{W}) = \max(L_{compute}, L_{memory}, L_{interconnect}) \tag{1}$$

约束条件包括硬件资源限制（容量、带宽、空间维度）。

> 代码凭证: `src/pim_optimizer/optimizer.py:PIMOptimizer`; `src/pim_optimizer/arch/arch.py:PIMArchitecture`; `src/pim_optimizer/workload/workload.py:ConvWorkload`.

2 方法概述

我们提出了一个基于整数线性规划 (ILP) 的自动化框架。核心创新在于其能够对基于 DRAM 的 PIM 的独特性能特征进行建模，特别是“行激活 (Row Activation)”开销。与内存访问成本均匀的传统加速器不同，基于 DRAM 的 PIM 性能对数据布局和访问模式（行命中与未命中）高度敏感。

我们的方法将映射空间分解为三个耦合的子问题：1. **循环分块 (Loop Tiling)**: 确定每个存储层级上每个循环维度的块大小。2. **空间映射 (Spatial Mapping)**: 将特定的循环维度分配给硬件空间资源（PE 行、PE 列、内部并行性）。3. **行激活建模 (Row Activation Modeling)**: 一种基于预计算的方法，用于估算 DRAM 行缓冲区冲突和激活周期。

> 代码凭证: `src/pim_optimizer/optimizer.py`; `src/pim_optimizer/model/row_activation.py`.

3 核心方法

3.1 决策变量

我们定义了一组二进制决策变量来表示映射配置。

3.1.1 分块变量 (xb)

令 $xb_{w,m,s,j,i} \in \{0, 1\}$ 为二进制变量，如果第 i 个候选因子被选中用于存储层级 m 和空间方向 s 上的第 j 个循环维度，则该变量为 1。

- w: 工作负载索引（用于多层优化）。
- m: 存储层级索引 $(0 \dots M)$。

- s: 空间方向索引。
  - 对于 PE 层级 ($m = 0$): $s \in \{0 : H, 1 : W, 2 : Internal, 3 : Temporal\}$。其中 $Internal$ 对应 PE 内部并行性（如 Tensor Core 或 SIMD）。
  - 对于其他层级 ($m > 0$): $s \in \{0 : Spatial(dummy), 1 : Temporal\}$。
- j: 循环维度索引 ($0 \dots 6$，对应 R, S, P, Q, C, K, N)。
- i: 除数列表中候选因子的索引。

3.1.2 排列变量 (xp)

令 $xp_{w,m,p,j} \in \{0, 1\}$ 为二进制变量，如果第 j 个循环维度被放置在存储层级 m 的时间循环顺序中的第 p 个优先级，则该变量为 1。其中 $p = 0$ 代表最内层循环。

3.1.3 旁路变量 (xd)

令 $xd_{w,m,t} \in \{0, 1\}$ 为二进制变量，如果数据类型 t（0: Input, 1: Weight, 2: Output）存储在存储层级 m（即未被旁路），则该变量为 1。

3.1.4 布局选择变量 (x_{layout})

令 $x_{layout,w,t,mode} \in \{0, 1\}$ 为二进制变量，用于选择 DRAM 中的数据布局模式。

- $mode \in \{sequential, row\_aligned\}$: 分别对应紧凑的顺序布局和对齐到 Row Buffer 的布局。

3.1.5 辅助变量 (xr, xj)

为了处理数据复用和行激活开销，我们引入了辅助变量：

- $xr_{w,t,m,m',p}$: 追踪在层级 m 和 m' 之间，优先级 p 处是否存在相关的内层循环。
- $xj_{w,t,m,m',j}$: 指示维度 j 在层级 m 和 m' 之间是否包含相关的内层循环（即维度 j 是否导致了数据的重复访问）。

3.1.6 RowBuffer 块选择变量

针对 Input 张量的特殊优化，我们定义了 $x_{rb\_block,w,i}$ 来选择 Row Buffer 层级的逻辑块大小（Block H/W），以优化滑动窗口的跨行行为。

代码凭证: `src/pim_optimizer/model/variables.py:VariableSet`; `src/pim_optimizer/model/variables.py:SpatialDim`.

3.2 约束条件

3.2.1 维度因式分解

所有层级和方向上的分块因子的乘积必须等于每个维度 j 的总问题规模 D_j。我们使用对数将其线性化：

$$\sum_{m=0}^{M} \sum_{s} \sum_{i} xb_{w,m,s,j,i} \cdot \log(\text{factor}_{j,i}) = \log(D_j), \quad \forall j \tag{2}$$

3.2.2 唯一性

对于任何特定的循环位置（由 m, s, j 定义），必须且只能选择一个分块因子：

$$\sum_{i} xb_{w,m,s,j,i} = 1, \quad \forall m, s, j \tag{3}$$

3.2.3 空间资源约束

在 PE 层级 ($m = 0$)，我们强制映射的空间并行性不超过硬件限制。

- **互斥性**: 一个循环维度 j 最多只能映射到一个空间方向（H、W 或 Internal），且因子 > 1。

$$\sum_{s \in \{H,W,Int\}} (1 - xb_{w,0,s,j,\text{idx}(1)}) \leq 1 \tag{4}$$

其中 idx(1) 是因子 1 的索引。

- **阵列大小**: 映射到方向 s 的因子的乘积必须适应硬件尺寸 $Size_s$：

$$\sum_{j} \sum_{i} xb_{w,0,s,j,i} \cdot \log(\text{factor}_{j,i}) \leq \log(Size_s), \quad \forall s \in \{H, W\} \tag{5}$$

3.2.4 缓冲区容量约束

对于每个存储层级 m 和数据类型 t，所需的存储大小不得超过容量 C_m。存储大小是相关维度的分块因子的乘积。

$$\sum_{j \in \text{Relevant}(t)} \sum_{s} \sum_{i} xb_{w,m,s,j,i} \cdot \log(\text{factor}_{j,i}) \leq \log(C_m) + M \cdot (1 - xd_{w,m,t}) \tag{6}$$

如果数据类型被旁路 ($xd = 0$)，项 $M \cdot (1 - xd_{w,m,t})$ 会放宽约束。

代码凭证: `src/pim_optimizer/model/constraints.py:add_basic_constraints`; `src/pim_optimizer/model/expressions.py:build_buffer_utilization`.

3.3 行激活建模 (Row Activation Modeling)

DRAM 行激活 (Row Activation) 是 PIM 性能的关键瓶颈。我们提出了一种混合模型，结合了高精度微观模拟 (Micro-Trace Simulation) 和解析公式 (Analytical Formula)。总开销模型如下：

$$TotalCost = (1 - z_{aligned}) \cdot Cost_{Seq} + z_{aligned} \cdot Cost_{Aligned} + Cost_{BlockCrossing} \tag{7}$$

其中 $z_{aligned} \in \{0, 1\}$ 是选择对齐布局的决策变量。

3.3.1 顺序布局开销 ($Cost_{Seq}$)

对于顺序布局 (Sequential Layout)，我们根据张量类型采用不同的建模策略：

- **输入张量 (Input)**: 由于滑动窗口导致的复杂访问模式，我们采用查表法。预先计算不同 Tile 尺寸下的平均激活次数 $AvgCost_i$，并将其代入 ILP 公式：

$$Cost_{Seq}^{Input} = \sum_{i \in Table} z_i \cdot [N_{tiles} + (AvgCost_i \cdot N_{tiles} - N_{tiles}) \cdot ReusePenalty] \tag{8}$$

  其中 z_i 是选择第 i 个查表项的二进制变量，$AvgCost_i \cdot N_{tiles}$ 是预计算的总激活次数。公式的物理含义是：基础开销为 N_{tiles}（每个 Tile 至少激活一次），额外的跨行开销 $(AvgCost - 1) \cdot N_{tiles}$ 会受到 $ReusePenalty$ 的放大。

- **权重/输出张量 (Weight/Output)**: 采用混合成本模型 (Hybrid Cost Model)。以理想的流式访问成本 ($TotalBytes/RowSize$) 为基准，根据 Tile 的对齐性（Alignment）和复用模式（Thrashing）施加惩罚系数，从而区分流式访问和抖动访问。

$$Cost_{Seq}^{W/O} = C_{stream} \cdot Reuse \cdot (1 + \mathbb{I}_{penalty}) \tag{9}$$

3.3.2 对齐布局开销 ($Cost_{Aligned}$)

在对齐布局中，每一行数据都被填充至 DRAM Row 边界，消除了行内冲突，但增加了带宽浪费。其开销由解析公式计算：

$$Cost_{Aligned} = \frac{\text{TotalBytes}_{padded}}{\text{RowSize}} \cdot ReusePenalty \tag{10}$$

3.3.3 输入块跨越开销 ($Cost_{BlockCrossing}$)

由于卷积的滑动窗口特性，输入 Tile 可能会跨越逻辑块 (Layout Block) 边界。我们使用基于 GCD 的数论方法计算跨越概率 P_{cross}：

$$P_{cross} = \frac{\text{Period} - \lceil (BlockH - TileH + 1)/g \rceil}{\text{Period}} \tag{11}$$

其中 $g = \gcd(Step, BlockH)$，$Period = BlockH/g$。最终开销为 $Cost_{BlockCrossing} = N_{tiles} \cdot P_{cross} \cdot C_{penalty}$。

代码凭证: `src/pim_optimizer/model/row_activation.py:precompute_tile_crossing_info`; `src/pim_optimizer/model/expressions.py:compute_unique_input_size`.

3.4 目标函数

目标是最小化计算和内存延迟的最大值：

$$\min \max(L_{compute}, L_{DRAM}) \tag{12}$$

DRAM 延迟 (L_{DRAM}): 对于每个数据类型 t，延迟是数据传输时间和行激活时间的总和：

$$L_{DRAM,t} = \frac{\text{DataVolume}_t}{BW_{RowBuffer}} + N_{act,t} \times t_{act} \tag{13}$$

总 DRAM 延迟由瓶颈数据类型主导：

$$L_{DRAM} = \max_{t \in \{In, Wgt, Out\}} L_{DRAM,t} \tag{14}$$

代码凭证: `src/pim_optimizer/model/objective.py:_build_dram_latency_cycles`.

4 实现细节

该框架使用 Python 实现，并采用 Gurobi 优化器。

4.1 软件架构

- **优化器入口 (`optimizer.py`)**: `PIMOptimizer` 类初始化架构和工作负载，然后编排优化过程。
- **模型构建 (`model/`)**:

  - `variables.py`: 定义包含所有 Gurobi 变量 (xb, xp, xd) 的 `VariableSet` 数据类。
  - `constraints.py`: 实现约束生成函数，如 `add_basic_constraints` 和 `add_buffer_constraints`。
  - `expressions.py`: 包含计算缓冲区利用率和唯一输入大小的逻辑。

- **预计算**: 为了处理行激活的非线性，系统预先计算成本表（通过 `row_activation.py`），并将其作为线性系数或查找表注入 ILP 模型。

4.2 执行流程

1. **解析配置**: 加载 `arch.yaml` 和 `workload.yaml`。2. **生成除数**: 将工作负载维度分解为候选质因数。3. **预计算表**: 运行 `precompute_tile_crossing_info` 为所有可能的分块大小生成激活成本。4. **构建 ILP**: 实例化 Gurobi 模型，添加变量和约束。5. **求解**: 运行 `model.optimize()`，并指定时间限制（默认 300 秒）。6. **提取结果**: 将二进制变量映射回可读的映射参数。

代码凭证: `src/pim_optimizer/cli.py`; `src/pim_optimizer/optimizer.py`.

5 行激活模块实现

本节详细介绍 `src/pim_optimizer/model/row_activation.py` 中的核心算法实现。该模块负责为 ILP 模型提供精确的 DRAM 行激活开销估算。针对不同类型的张量（Tensor），我们采用了差异化的建模策略。

5.1 输入张量 (Input Tensor) 建模

输入张量由于卷积操作的滑动窗口（Sliding Window）特性，其内存访问模式最为复杂。相邻的计算窗口在输入特征图上存在重叠，导致数据复用和行缓冲区冲突（Row Buffer Conflict）难以通过简单的线性公式计算。

5.1.1 基于 GCD 的跨越分析

为了解决这一问题，我们实现了 `compute_input_block_crossing_count` 函数。该函数利用数论中的最大公约数（GCD）性质，计算滑动窗口跨越逻辑块（Layout Block）边界的精确概率。

- **周期性分析**: 访问模式的周期由 $Period = BlockH / \gcd(Step, BlockH)$ 定义。

- **跨越计数**: 在一个周期内，跨越边界的次数可以通过解析公式直接计算，无需逐个模拟。
- **核分裂 (Kernel Splitting)**: 当卷积核尺寸较大导致需要分多次访问时，函数会分别计算每个子核（Sub-kernel）的跨越情况并累加。

5.1.2 查表法与 AvgCost

对于输入张量，我们使用 `precompute_input_block_crossing_table` 生成查找表。该表存储了不同分块策略下的平均激活开销 ($AvgCost$)。

- $AvgCost$ 反映了在特定的 $(P_{tile}, Q_{tile}, C_{tile})$ 组合下，平均每个 Tile 需要多少次额外的行激活。
- ILP 模型通过 `build_input_block_crossing_expr` 将这些预计算的成本注入到优化目标中。

5.2 权重与输出张量 (Weight/Output) 建模

与输入张量不同，权重（Weight）和输出（Output）张量的访问模式通常是线性的或分块线性的（Tiled-Linear）。对于这两种张量的顺序布局（Sequential Layout），我们不使用平均值，而是采用基于 GCD 的精确计数方法。

5.2.1 混合成本模型 (Hybrid Cost Model)

对于权重和输出张量的顺序布局，代码实现了一种混合成本模型，旨在区分“流式访问 (Streaming)”和“抖动访问 (Thrashing)”两种模式。该模型不直接累加每个 Tile 的跨行开销，而是以理想的流式访问成本为基准，根据 Tile 的对齐情况和复用模式施加惩罚。

- **基准流式成本 (C_{stream})**: 假设数据能够以理想的顺序流过 Row Buffer，其最小激活次数仅取决于数据总量和行大小：
$$C_{stream} = \frac{\text{TotalBytes}}{\text{RowSize}} \tag{15}$$
- **惩罚系数 (M_k)**: 针对每个候选 Tile 尺寸 k，根据其是否与 Row Size 对齐以及是否存在复用抖动，确定倍率系数：
  - **流式模式 ($M_k = 1$)**: 当 Tile 与 Row 对齐，或者虽然不对齐但没有复用（Reuse=1）时，开销接近基准值。
  - **抖动模式 ($M_k = 2$)**: 当 Tile 不对齐且存在频繁复用（Reuse>1），或者内部循环导致 Row Buffer 频繁切换（Thrashing）时，开销加倍。

- **复用惩罚 (*ReusePenalty*)**: 该变量由两部分组成，反映了不同层级的无关循环对行激活的影响。
  - **组成部分**:
    1. **L3 内层无关维度**: 位于 L3 (DRAM) 层级之下的无关循环。
    2. **L2 内层无关维度**: 位于 L2 (Global Buffer) 层级之下的无关循环。
  - **公式**:

$$ReusePenalty = \left(\prod_{j \in Irr} \text{Bound}_j^{(1-x_{j,L3})} \right) \times \left(\prod_{j \in Irr} \text{Bound}_j^{(1-x_{j,L2})} \right) \tag{16}$$

    其中 $x_j = 0$ 表示维度 j 为内层（Stationary），$1 - x_j = 1$ 表示该维度对复用有贡献。

- **最终公式**:

$$Cost_{Seq}^{W/O} = \sum_k x_k \cdot C_{stream} \cdot ReusePenalty \cdot M_k \cdot OuterIrrProduct \tag{17}$$

  其中 x_k 是选择第 k 个 Tile 尺寸的二进制变量。这种建模方式在保证计算效率的同时，能够捕捉到由于不对齐和抖动导致的带宽浪费。

5.3 ILP 集成

模块通过 `_build_layout_conditional_acts` 函数将上述两种成本模型整合。它引入了二进制变量 $z_{aligned}$ 来动态选择“顺序布局”或“对齐布局”，从而允许优化器在“节省空间（紧凑布局）”和“减少延迟（对齐布局）”之间进行权衡。

6 局限性与范围

- **工作负载支持**: 当前实现专门针对 `ConvWorkload`。支持其他算子（如矩阵乘法、深度卷积）需要扩展工作负载定义和输入大小公式。
- **静态建模**: 延迟模型假设确定性的 DRAM 行为，未考虑动态刷新周期或行缓冲区模型之外的复杂 Bank 冲突。
- **整数因子**: 分块优化仅限于循环维度的整数因子。

  代码凭证: `src/pim_optimizer/workload/workload.py`; `src/pim_optimizer/model/constraints.py`.