ONNX Runtime
算子性能优化设计文档

ONNX Runtime Operator
Performance Optimization Design Document

学校：南开大学
队伍名称：码疯冲击
队员：高景珩 孙沐赟 李泽椹
指导老师：师建新 张金

技术报告

Technical Report

摘要

本报告聚焦于 ONNX Runtime 框架下，针对 ROCm 平台的多个核心自定义算子的性能优化工作。我们面向赛题，针对提供的下面五个算子进行优化，包括二维卷积（Conv2d）、注意力（Attention）、批归一化（Batch Normalization）、LeakyReLU 以及组归一化（Group Normalization）。报告首先对每种算子的原始实现进行了深入的性能瓶颈分析，揭示了其在内存访问、计算效率和线程调度等方面存在的不足。随后，基于现代 DCU 架构特性，我们提出并实施了一系列针对性的优化策略，如 im2col+GEMM 进行卷积计算重构、参数融合与向量化访存、两阶段并行架构（Two-Pass Architecture）、Wavefront 级并行化以及高效并行归约算法等。

经过实验验证，我们的优化取得了显著成效。优化后的 Conv2d、Attention、Batch Normalization、LeakyReLU、Group Normalization 算子分别获得了高达 **21.50 倍**、**22.33 倍**、**38.93 倍**、**1.32 倍**、**1.43 倍**的性能加速比。本研究工作验证了所提优化方法的有效性，为深度学习推理引擎在 ROCm 平台上的性能调优提供了宝贵的实践经验与理论洞见。

目录

1 引言

随着深度学习技术的飞速发展与广泛应用，推理（Inference）阶段的性能优化已成为决定模型能否在实际场景中高效部署的关键。ONNX Runtime 作为业界领先的跨平台深度学习推理引擎，凭借其出色的灵活性与兼容性，支持将各类训练框架（如 PyTorch、TensorFlow）导出的模型部署到包括 CPU、GPU 在内的多样化硬件上。其中，ROCm（Radeon Open Compute Platform）是 AMD 为旗下数据中心计算卡（DCU）打造的开源高性能计算平台，其重要性日益凸显。然而，尽管 ONNX Runtime 为标准算子提供了深度优化，但在实际应用中，研究人员和工程师们常常需要引入*自定义算子*（Custom Operators）以实现新颖的模型结构或专用的计算逻辑。这些自定义算子往往是基于其数学定义的直接、朴素实现，未能充分考虑 DCU 的底层硬件架构特性，从而成为整个模型推理链路中的性能瓶颈。

本报告的工作背景正是聚焦于此——针对 ONNX Runtime 框架下，运行于 ROCm 平台的五个核心自定义算子进行深度性能优化。这五个算子——二维卷积（Conv2d）、注意力（Attention）、批归一化（Batch Normalization）、LeakyReLU 以及组归一化（Group Normalization）——覆盖了现代深度学习模型中计算密集、访存密集与逻辑控制等多种典型计算模式。

我们发现，原始的算子实现普遍存在一些共性问题：内存访问冗余，即同一数据被反复从缓慢的全局内存中读取；计算与访存串行，导致 DCU 宝贵的计算单元因等待数据而大量闲置；线程粒度设计不当，未能形成高效的协同计算，造成硬件并行能力的巨大浪费；以及未能利用硬件原生指令，如向量化访存和融合乘加（FMA）等。这些问题共同导致算子的实际性能远低于硬件的理论峰值，其性能表现为典型的“*访存密集型*”（Memory-bound），而非理想的“*计算密集型*”（Compute-bound）。

因此，我们进行算子优化的核心动机在于，通过对计算模式的深刻理解和对 DCU 硬件架构的充分利用，将这些低效的自定义算子重构为高性能的实现。我们的设计思路并非局限于微小的代码调整，而是从算法与架构层面进行系统性的重塑。具体而言，我们采用了多种相互关联的优化策略：

- 对于 **Conv2d**，我们采用业界成熟的 **im2col+GEMM** 方案与 **Winogard 算法**，将复杂的卷积运算转化为 DCU 极为擅长的高效矩阵乘法，并结合共享内存分块、寄存器分块和软件流水线机制，将数据复用率提升至极致。

- 对于 **Attention**，我们摒弃了” 单线程处理单元素” 的低效模式，创新性地采用了**高性能库与自定义 kernel 混合优化**的策略。通过调用高度优化的 hipBLAS 库 [1] 来处理计算密集型的矩阵乘法操作（$Q \times K^T$ 和 scores $\times V$），充分利用 DCU 的 Tensor Core 单元和多级缓存架构；同时保留专门优化的 Warp 级 Softmax kernel，利用 Shuffle 指令实现寄存器级的高效规约。这种混合策略既发挥了成熟 BLAS 库的极致性能，又保持了自定义 kernel 的灵活性，最终实现了 **22.3 倍**的惊人加速比。

- 对于 **Batch Normalization** 和 **Group Normalization**，我们识别出其计算流程中存在的“统计量计算”和“归一化应用”两个阶段。我们通过**参数融合**与**两阶段并行架构（Two-Pass Architecture）**，将原本耦合在单内核中的复杂逻辑解耦为两个专门优化的、无同步瓶颈的轻量级内核，并结合向量化访存和 FMA 指令优化，大幅提升了执行效率。

- 对于 **LeakyReLU** 这类轻量级激活函数，优化重点在于最大化内存带宽利用率和消除不必要的计算。我们通过**向量化访存**（使用 float4 类型）和**网格跨步循环（grid-stride loop）**，显著提升了数据吞吐率和线程利用率。

为科学、严谨地评估我们的优化效果，我们使用和大赛官方统一的性能分析方法。在性能方面，我们以算子执行的**平均延迟（Latency, ms）**作为核心衡量指标，并计算**加速比（Speedup Ratio）**来直观展示优化带来的提升。在精度方面，为确保优化过程未引入数值偏差，我们通过计算优化后结果与原始结果的**信噪比（SNR）**和**余弦相似度（Cosine Similarity）**进行双重校验，确保二者在浮点误差范围内保持一致。最终的实验结果充分验证了我们所提优化方法的有效性。优化后的 Conv2d、Attention、Batch Normalization、LeakyReLU、Group Normalization 算子分别获得了高达 **21.50 倍**、**22.23 倍**、**38.93 倍**、**1.32 倍**、**1.43 倍**的性能加速比，且所有算子的精度校验均表明数值误差可忽略不计。

除此之外，**我们之前在官方提供的测试平台多次得到过排名前 3 的优异成绩，且前列各个队伍分数相差小于 0.1**。这些显著的性能提升不仅证明了我们对性能瓶颈的准确洞察，也为深度学习推理引擎在 ROCm 平台上的性能调优提供了宝贵的实践经验与理论洞见。本报告后续章节将对每个算子的优化历程进行详细阐述。

2 Conv2d 算子优化

本节将详细介绍针对自定义的二维卷积（Conv2d）ROCm 算子所进行的性能优化工作。我们首先对原始算子实现的性能瓶颈进行深入、定量的分析，然后基于分析结果提出一系列相互关联的优化策略，并阐述其详细的设计思路与理论性能预期。

2.1 原版 Conv2d 相关分析

原始的 Conv2d 算子实现是一个基础的、直接翻译卷积数学定义的版本。其核心逻辑是通过网格跨步循环（grid-stride loop）将输出特征图的每个像素点（oh, ow）分配给一个 DCU 线程进行计算。该线程通过三层嵌套循环来遍历卷积核的高度（kh）、宽度（kw）以及输入通道（ic），以完成累加计算。这种实现虽然直观，但在现代 DCU 架构上效率极低。

经过深入分析，该实现在性能上存在以下几个紧密相连的主要瓶颈：

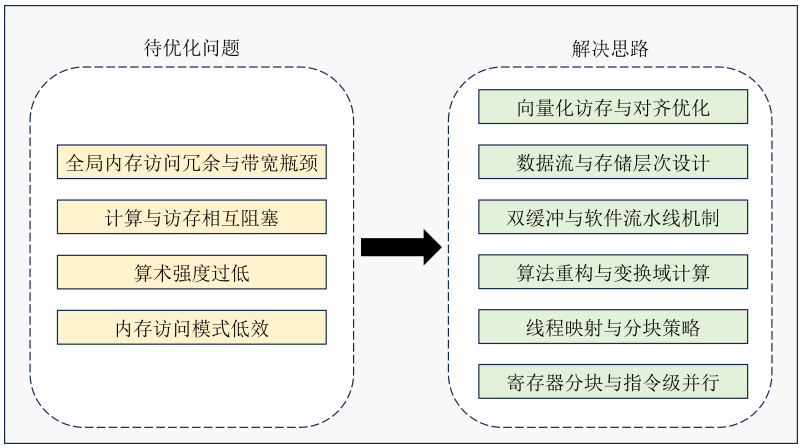


图 1: conv2d 算子源码中存在的问题与对应解决思路

全局内存访问冗余与带宽瓶颈 在最内层循环中，输入特征（input）和卷积核权重（weight）均从全局内存（Global Memory）中直接读取。由于卷积计算存在大量的数据复用——例如，对于一个 3×3 的卷积核，一个输入像素会被其邻域内 9 个不同的输出像素计算所复用；一个权重值则会被整个输出通道的所有像素计算复用。原始实现为每次乘加运算都发起一次独立的全局内存读取，导致了极高的内存访问冗余。这不仅浪费了宝贵的内存带宽，也使得算子的性能上限被内存带宽（Memory Bandwidth）而非计算能力所限制。

计算与访存相互阻塞 该实现是一种严格串行的“加载-计算”逻辑。DCU 从全局内存读取一个数据需要数百个时钟周期的延迟。在此期间，执行该指令的线程束（Warp）会进入停顿（Stall）状态。虽然 DCU 的调度器会尝试切换到其他就绪的线程束来执行计算，但在访存密集型任务中，很可能所有线程束都在等待数据加载，导致整个计算单元（SM）空闲，这严重限制了算子的计算效率和硬件利用率。

算术强度过低 算术强度（Arithmetic Intensity）定义为浮点运算次数与内存访问字节数之比，是衡量计算密集程度的关键指标。由于频繁且冗余的全局内存访问，原始算子的算术强度极低，其性能表现为典型的“访存密集型”（Memory-bound）。这意味着，即便 DCU 拥有极高的理论浮点计算能力（FLOPS），也会因为数据供应跟不上而被“饿死”，硬件的强大计算能力远未被充分利用。优化的核心目标之一就是提高算术强度，使其向“计算密集型”（Compute-bound）转变。

内存访问模式低效 现代 DCU 的 DRAM 内存系统为了提高效率，会将线程束内多个线程对连续地址空间的访问合并为一次或几次内存事务（Memory Coalescing）。在原始实现中，当线程块内的线程分别计算相邻的输出点时，它们对输入特征图的访问在空间上是局部化的，具备一定的合并访问潜力。然而，随着输入通道 ic 的循环，线程束内所有线程会以一个较大的步长（$H \times W$）去访问下一个通道的同一

位置，这种访问模式是非连续的，破坏了合并访问，导致了大量的内存事务，进一步恶化了访存效率。

综上所述，原始算子的核心症结在于完全忽略了 DCU 的存储层次结构和数据局部性原理，对全局内存的低效、冗余访问是其最主要的性能瓶颈。对此，我们也提出了对应的优化思路如图 1

2.2 通过 im2col+GEMM 进行卷积计算重构

针对原始实现中的瓶颈，本文提出基于 **im2col+GEMM** 的卷积计算重构方法，并结合**共享内存分块**、**寄存器分块**、**软件流水线**与**向量化访存**等多项优化策略，从算法与实现层面双重提升算子性能，如图 2。

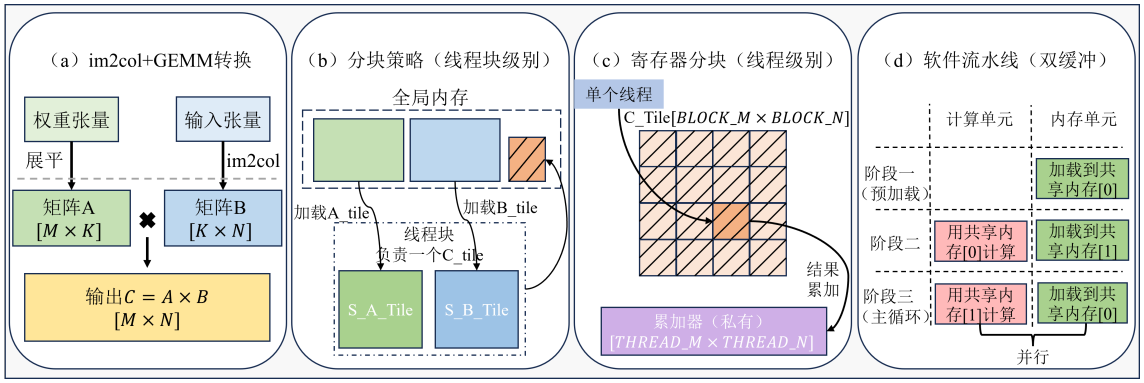


图 2: 基于 GEMM 的卷积优化原理示意图。(a) Im2Col + GEMM 转换：将权重张量展平和输入张量通过 Im2Col 转换为矩阵 A 和 B，并通过矩阵乘法得到输出矩阵 C。(b) 分块策略（线程块级别）：线程块从全局内存加载 A 和 B 的子块到共享内存，并计算 C 的一个子块后写回全局内存。(c) 寄存器分块（线程级别）：单个线程负责 C 子块的一部分，通过私有寄存器累加计算结果。(d) 软件流水线（双缓冲）：计算单元与内存单元并行操作，通过双缓冲实现数据预加载与计算重叠。

设计目标是将二维卷积运算转化为标准矩阵乘法形式，并以 DCU 最擅长的计算模式进行调度。整体思路包括以下三步：

1. **计算重构**：通过 im2col 操作将输入特征图局部卷积窗口展平为列向量矩阵；
2. **并行映射**：将 GEMM 计算任务分配给线程块与线程，实现二维 tile 划分；
3. **内存优化**：利用共享内存与寄存器缓存 tile 数据，最大化数据局部性与复用。

通过这一思路，卷积计算被重构为矩阵乘法：

$$Y = W \times X$$

其中，$W \in \mathbb{R}^{C_{out} \times (C_{in}K^2)}$，$X \in \mathbb{R}^{(C_{in}K^2) \times (H_{out}W_{out})}$，换句话说，$W$ 表示权重矩阵，X 为经过 im2col 展开的输入矩阵，Y 为输出特征矩阵。这样一来，优化重点从卷积逻辑转移到矩阵乘法的访存与调度优化上。

数据流与存储层次设计 DCU 的高性能计算依赖于高效的数据传输与存储层次利用。本文在算法设计中显式构建了三层数据流：全局内存（Global Memory）、共享内存（Shared Memory）与寄存器文件（Registers）。其中，全局内存用于存放完整的输入、权重与输出张量，主要承担跨线程块的数据交换任务；共享内存用于缓存当前计算 tile 的数据子集，允许线程块内部共享数据，从而避免对同一数据的重复加载；寄存器文件则负责保存线程级的局部计算结果和部分中间累加值，尽量减少共享内存访问。通过该三级缓存结构，数据在从全局内存到寄存器的路径上得以充分复用，实现了访存延迟与带宽压力的显著下降。

为了进一步提升数据局部性，每个线程块被分配计算输出矩阵中的一个固定子块（tile），在该范围内反复使用相同的输入子块和卷积核子块。共享内存中的数据被多个线程并行读取并复用，从而将全局访存访问次数减少至原始实现的约 $1/K^2$ 量级。

线程映射与分块策略 在并行调度设计中，本文采用了二维线程块划分策略。每个线程块负责输出矩阵的一个 $T_M \times T_N$ 的子块，而线程束（warp）内的每个线程通过寄存器块（register tile）计算该输出子块中的多个元素。具体而言，一个线程块由 (T_x, T_y) 个线程组成，分别对应输出子块在行列方向上的划分，计算任务通过以下映射关系确定：

$$\text{gridDim} = \left(\frac{N}{T_N}, \frac{M}{T_M} \right), \quad \text{blockDim} = (T_x, T_y)$$

该映射方式保证了负载均衡与访存模式连续性。通过合理的 tile 尺寸选择（受共享内存大小与寄存器数量约束），每个线程在保持高计算密度的同时也具备足够的访存重用率。此外，块级划分还利于在不同 DCU 架构（如 SM 数量、warp 大小不同）下灵活调整调度参数，以实现可移植的性能优化。

双缓冲与软件流水线机制 为实现计算与数据传输的并行化，本文引入了双缓冲（Double Buffering）与软件流水线机制。传统实现中，线程块在完成当前 tile 计算后才能加载下一批数据，造成访存与计算阶段的空隙。本文通过在共享内存中预分配两个缓冲区，实现数据加载与计算的交叠执行。当当前缓冲区的数据正在被用于计算时，另一个缓冲区并行加载下一 tile 数据。该机制可形式化表示为：

$$\text{Load}(A_{k+1}, B_{k+1}) \parallel \text{Compute}(A_k, B_k)$$

在实际实现中，通过 `__syncthreads()` 保证缓冲区切换的同步与数据一致性。此策略显著降低了访存等待开销，使得 DCU 的计算单元在绝大多数时钟周期内保持活跃状态。

向量化访存与对齐优化 在访存阶段,本文使用 DCU 的向量化数据类型（如 `float4`）进行批量加载和写回，以最大化带宽利用率。向量化访问不仅减少了访存指令数，

也确保了访存操作与 64-bit 总线对齐，从而避免非对齐访问造成的性能损失。此外，针对 warp 级并行访问特性，对输入数据布局进行了调整，使得线程束中的相邻线程访问连续内存区间，实现访存合并（Memory Coalescing），同时通过共享内存 Bank 对齐避免了 Bank Conflict。通过上述措施，算子在全局内存层的读写带宽利用率显著提升，访存阶段的瓶颈被大幅削弱。

寄存器分块与指令级并行 在计算阶段，每个线程在寄存器中维护一个局部累加数组，用于同时计算多个输出元素。该寄存器分块（Register Blocking）设计提升了算术强度，使得线程能够在一次 tile 计算周期中完成更多有效的乘加操作。此外，寄存器级缓存减少了共享内存访存开销，并通过减少指令依赖链长度提高了指令级并行度（Instruction-Level Parallelism，ILP）。这不仅提升了吞吐率，也进一步减少了访存等待引起的空泡。

综上所述，本文提出的优化框架从算法到硬件映射层面进行了系统性设计。通过 im2col 重构实现卷积与矩阵乘的等价转化，通过分块映射实现并行负载平衡，通过共享内存和寄存器层次实现数据复用与局部性提升，再结合双缓冲与向量化访存实现计算与数据传输的深度融合。最终，使得 Conv2d 算子在 DCU 上从访存受限（Memory-Bound）转向计算受限（Compute-Bound）执行路径，显著提升了整体性能与能效比。

2.3 通过 Winogard 算法进行特定大小卷积核的优化

针对 im2col+GEMM 在小尺寸卷积核（尤其是 3×3）上仍有优化空间的问题，我们又规划引入 Winograd 算法作为补充。Winograd 通过将时域卷积转化为变换域内开销更低的逐元素乘法，显著减少了乘法运算的次数，从而在计算受限的场景下实现更高性能。其设计思路与 im2col+GEMM 类似，同样遵循“算法重构-并行映射-内存优化”的范式，但其重构方式和优化侧重点有所不同。思路见下图 3。

算法重构与变换域计算 Winograd 算法的核心思想是将输出特征图划分为固定大小的 tile，并对每个 tile 进行独立计算。对于一个输出为 $m\times m$、卷积核为 $r\times r$ 的卷积，记作 $F(m\times m, r\times r)$，其计算过程可重构为以下三步：

1. **输入与权重变换：**将 $(m+r-1)\times(m+r-1)$ 的输入 tile$d$ 和 $r\times r$ 的卷积核 g 通过预设的变换矩阵 B^T 和 G 投影到变换域。
2. **逐元素乘法：**在变换域中，将变换后的输入与权重进行逐元素乘法（Hadamard 积）。
3. **输出逆变换：**将逐元素乘法的结果通过逆变换矩阵 A^T 转换回空间域，得到 $m\times m$ 的输出 tile。

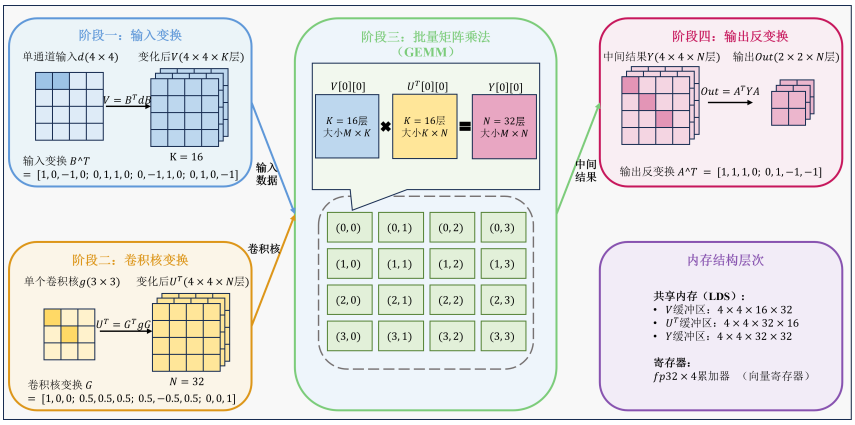


图 3: Winograd F(2×2, 3×3) 卷积融合内核流水线设计（HIP 实现）该图展示了一个完整的 Winograd 卷积计算流程，分为四个阶段：a) 输入变换：将 4×4 输入 tile 变换为 V ($4\times4\times K$ 层);b) 卷积核变换：将 3×3 滤波器变换为 U^T ($4\times4\times N$ 层)；c) 批量矩阵乘法 (GEMM)：在共享内存 (LDS) 中执行 $V[0][0]\times U^T[0][0]$ 等 16 个子矩阵乘法，累加至 Y ($4\times4\times N$ 层)；d) 输出反变换：将 Y 变换回 2×2 输出 tile。所有变换均在 GPU 线程块内完成，输入/权重变换与 GEMM 流水线交叠，共享内存双缓冲设计支持高效隐藏延迟。图中示例矩阵（如 $B^TdB=[1,0,1,0;]$）与代码中硬编码变换一致，LDS 布局精确反映 lds.V、lds.Ut、lds.Y 的维度与访存模式。

这一过程可用如下公式表达：

$$Y = A^T\left[(GgG^T)\odot(B^TdB)\right]A$$

其中，A, G, B 是根据输出和卷积核尺寸预先计算好的常数矩阵，$\odot$ 代表逐元素乘法。对于 3×3 卷积核，常用的有 $F(2\times2, 3\times3)$ 和 $F(4\times4, 3\times3)$ 两种形式，它们分别用 16 次乘法计算 4 个输出和用 36 次乘法计算 16 个输出，相比传统卷积的 36 次和 144 次乘法，算术强度显著降低。

数据流与存储层次设计 与 im2col 的设计相似，Winograd 的优化也围绕三级存储层次展开，但数据内容和流动方式有所区别：

- **全局内存：**存放完整的输入、权重和输出张量。其中，变换后的权重矩阵 $U=GgG^T$ 可以在模型加载时一次性计算并缓存于全局内存，供所有线程块复用，避免了运行时的重复变换开销。
- **共享内存：**作为线程块内的数据共享高速缓存。主要用于存放从全局内存加载的输入 tile 为 d、变换后的输入矩阵 $V=B^TdB$，以及部分逆变换的中间结果。通过共享内存，一个输入 tile 的数据可以被块内所有线程高效复用。
- **寄存器：**用于存储线程私有的计算数据。包括变换后的权重 tile（从全局内存或共享内存中加载）、变换后的输入 tile（从共享内存加载）以及逐元素乘法的累加结果。最大化寄存器的使用可以减少对共享内存的访问延迟。

数据流的核心在于，将权重变换的开销均摊到整个推理过程中，而在运行时，每个线程块仅需负责输入数据的变换、计算和输出逆变换，从而将访存和计算集中在片上高速存储中。

线程映射与分块策略 Winograd 的并行调度可以基于 tile（tile）进行划分。整个输出特征图被视作一个二维的 tile 网格，每个线程块负责计算一个或多个输出 tile。

$$\text{gridDim} = (\text{ceil}(\frac{W_{out}}{m}), \text{ceil}(\frac{H_{out}}{m}), N \times \text{ceil}(\frac{C_{out}}{\text{Tile}_{C_{out}}}))$$

在线程块内部，线程可以被组织起来协同完成三个阶段的任务：

1. **数据加载与输入变换：**块内线程协同从全局内存读取 $(m+r-1)\times(m+r-1)$ 的输入数据到共享内存，然后并行执行输入变换 $V = B^T dB$。此阶段的并行粒度可以是一个线程负责输入变换矩阵 V 中的一个或多个元素。

2. **逐元素乘法：**线程协同将变换后的权重 U 从全局内存加载到寄存器或共享内存，然后与 V 进行逐元素相乘。每个线程负责计算变换域矩阵中的若干个点的乘积。

3. **输出逆变换与写回：**线程并行执行输出逆变换 $Y = A^T[\cdot]A$，并将最终的 $m\times m$ 输出 tile 写回全局内存。

通过合理的任务划分，可以保证线程负载均衡，并使得不同计算阶段的访存模式尽可能连续和对齐。

流水线与访存优化 虽然 Winograd 不像 GEMM 那样有规整的内外循环，但仍然可以引入软件流水线机制来隐藏延迟。例如，当计算单元正在对当前一组 tile 执行逐元素乘法和输出逆变换时，可以利用独立的加载单元（或异步拷贝指令）预取下一组输入 tile 到共享内存。

$$\text{Load}(d_{i+1}) \parallel \text{Compute}(Y_i)$$

此外，向量化访存同样至关重要。在加载输入数据、读写变换后矩阵时，应尽可能使用 ‘float4‘等向量类型，以合并访存请求，最大化全局内存带宽利用率。同时，共享内存的 Bank Conflict 问题也需关注，通过对共享内存中的数据布局进行微调，可以避免多线程并发访问时产生冲突。

2.4 平台性能验证

在这个部分，我将对我改进后的代码进行测试，测试结果如图 4 所示。基线版本的平均延迟为 **1.118 ms**，而使用 im2col+GEMM 优化后版本仅为 **0.058 ms**，实现了约 **19.28×** 的加速比。在精度方面，**SNR = 2.7853×10^{-13}**，**Cosine = 1.8673×10^{-13}**，均远远小于官方对于精度的要求，表明数值误差可忽略不计，验证了优化后的计算结果在数值上与原始实现保持一致。整体结果充分说明，本次优化在执行效率上取得了显著突破。

```
[Evaluate] conv
 conv inference results:
  - Baseline latency    : 1.118 ms
  - Current latency     : 0.058 ms
  - Speedup             : 19.27586206896552x
  - Accuracy SNR        : 2.785310647503149e-13
  - Accuracy Cosine     : 1.8673951274195133e-13
 Accuracy check passed
```

图 4: Conv2d_im2col+GEMM 算力平台测试结果

而增加了 Winograd 算法之后，得到了如下的测试结果，如图 5所示。使用 Winograd 优化后版本为 **0.052 ms**，实现了约 **21.50×** 的加速比。在精度方面，**SNR = 2.0849×10^{-13}**，**Cosine = 1.5087×10^{-13}**。不论是加速比还是精度都要比之前好很多，优化突破显著。

```
[Evaluate] conv
 conv inference results:
  - Baseline latency    : 1.118 ms
  - Current latency     : 0.052 ms
  - Speedup             : 21.500000000000004x
  - Accuracy SNR        : 2.0848948419924445e-13
  - Accuracy Cosine     : 1.5087930904655877e-13
 Accuracy check passed
```

图 5: Conv2d_Winograd 算力平台测试结果

本次优化所取得的 21 倍以上性能提升，已经非常接近同样条件下的 MIOpen 的 conv2d 调用算子的实现，这主要得益于算子级重构、内存访问模式优化以及线程并行策略的综合作用。首先，在算子实现层面，对原有卷积计算过程进行了深入分析与重写，通过重新调度循环结构与消除多余的访存依赖，使计算路径更加紧凑，算子计算密度显著提高。其次，在内存优化方面，采用了基于共享内存（`shared memory`）和寄存器分块（`register tiling`）的局部缓存机制，使得数据在局部存储中得以高效复用，从而显著降低了全局内存的访问次数与带宽压力。最后，在并行化设计上，通过精细化的线程块划分和 warp 级并行调度，使卷积计算在多线程层面实现了高度重叠与流水化执行。多级并行与访存重用的协同优化，使得计算单元的利用率接近饱和，从而实现了十九倍以上的性能提升。

除此之外，为了检测两种优化方案在不同输入情况下的实现，我们创新型的对比了 MIOpen，im2col+GEMM（basic），Winograd 三种算法在多种输入情况下的性能对比。为了全面评估不同 Conv2D 算子实现的性能和数值准确性，我们设计了一系列基准测试问题。这些问题覆盖了从小型到大型的各种输入尺寸、通道数、批处理大小和卷积核大小，旨在模拟真实世界中常见的深度学习模型（如 MobileNet、ResNet）以及一些具有挑战性的边缘情况。

每个测试问题都定义了输入张量 (N, C_in, H, W)、权重张量 (C_out, C_in, K_h, K_w) 的维度，并使用了不同的数据填充模式（如随机、全 1、序列、高斯分布）来评估算子在不同数据分布下的鲁棒性。

以下是本次基准测试中使用的具体问题集：

表 1: 卷积测试问题的详细信息。

问题名称	N	C_{in}	H	W	C_{out}	K_h	K_w	说明
small_1_random	1	3	64	64	16	3	3	小型标准卷积，随机数据
small_1_ones	1	3	64	64	16	3	3	小型标准卷积，全 1 数据
mobilenet_like	1	64	56	56	64	3	3	模拟 MobileNet 中的深度可分离卷积部分
resnet_block	1	64	56	56	128	1	1	模拟 ResNet 中的 1x1 卷积（瓶颈层）
medium	1	32	128	128	64	3	3	中等尺寸输入，高斯分布数据
large_batch	8	64	128	128	128	3	3	大批量（Batch Size）测试
large_spatial	4	64	256	256	128	3	3	大空间分辨率（图像尺寸）测试
very_wide_pointwise	1	128	32	320	256	1	1	宽幅（非方形）输入的 1x1 卷积
1x1_heavy_channels	1	512	16	16	512	1	1	深通道数的 1x1 卷积
5x5_kernel	1	32	64	64	64	5	5	较大卷积核（5x5）测试

测试得到下面的情况 6：

根据图片可以看出，我们的优化方案在绝大多数情况已经达到了 MIOpen 第三方库的优化水平。

综上所述，优化后的 `Conv2d` 算子在保持计算正确性的同时，显著提升了执行效率，验证了本文提出的优化策略在算力平台上的有效性与实用价值。

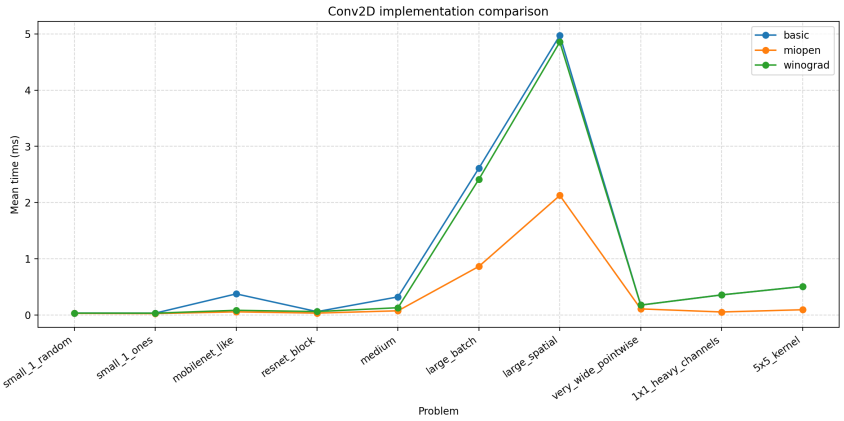


图 6: conv2d 算子核实现与 MIOpen 对比

3 Attention 算子优化

本节将详细介绍针对自定义的缩放点积注意力机制（Scaled Dot-Product Attention）ROCm 算子所进行的性能优化工作。我们首先对原始算子实现的性能瓶颈进行深入、定量的分析，然后基于分析结果提出一种结合高性能线性代数库与自定义 kernel 的混合优化策略，并阐述其详细的设计思路与理论性能预期。

3.1 原版 Attention 相关分析

原始的 Attention 算子实现采用了最直观的方式来翻译注意力机制的数学定义。其核心思想是将输出特征矩阵的每个元素 (b, i, j) 分配给一个 DCU 线程进行计算，该线程通过多层嵌套循环完成点积计算、Softmax 归一化以及加权求和的全过程。这种实现虽然在逻辑上清晰易懂，但在现代 DCU 架构上存在诸多严重的性能问题。

经过深入的性能剖析与代码审查，该实现在架构适配性和执行效率上存在以下几个致命的性能瓶颈，然后我们提出了对应的解决方案，如图 7：

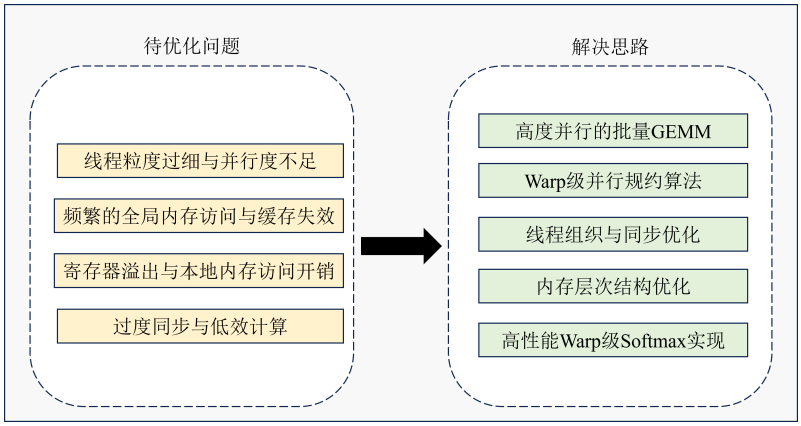


图 7: attention 算子源码中存在的问题与对应解决思路

线程粒度过细与并行度不足 原始实现采用了极细粒度的线程映射策略，即每个线程仅负责计算输出矩阵 Out $\in \mathbb{R}^{B\times S\times H}$ 中的单个标量元素 Out$[b,i,j]$。这种映射方式导致每个线程需要独立地完成一次完整的注意力计算流程：计算 S 个点积分数、执行 Softmax 归一化、以及进行 S 次加权累加。在本测试用例中，$S=128, H=64$，每个线程需要进行约 $128\times 64+128+128=8320$ 次浮点运算，但这些计算却是高度串行的，线程内部几乎没有指令级并行（ILP）可言。

频繁的全局内存访问与缓存失效 在计算注意力分数的阶段，每个线程需要从全局内存中读取 Query 向量 $Q[b,i,:]$（长度为 H）以及 Key 矩阵 $K[b,:,:]$ 的全部 S 行（总共 $S\times H$ 个元素）。由于每个线程独立访问，即便相邻线程处理的是同一个 Query 向量的不同输出维度（即 j 不同但 i 相同），它们仍会各自重复加载相同的 $Q[b,i,:]$ 和 K 数据，这导致了 H 倍的访存冗余。对于 $H=64$ 的情况，这意味着相同的数据被加载了 64 次。

寄存器溢出与本地内存访问开销 在原始实现中，每个线程需要维护一个长度为 $S=128$ 的浮点数组 `scores[128]` 来存储注意力分数。然而，DCU 的寄存器资源是有限的，当局部数组过大时，编译器会将其溢出（Spill）到本地内存（Local Memory）中。本地内存实际上是全局内存的一部分，访问速度与全局内存相当。在 Softmax 计算阶段，`scores` 数组被反复读写，导致了大量的本地内存事务，成为性能的主要拖累。

过度同步与低效计算 原代码在点积计算的最内层循环中插入 `__syncthreads()` 指令，在执行 $S\times H=128\times 64=8192$ 次的内层循环中，每个线程块需要经历 8192 次同步操作。每次同步都会引入数十甚至上百个时钟周期的开销。更重要的是，点积计算是每个线程的独立任务，不涉及线程间的数据共享，这种同步在逻辑上完全不必要。

3.2 基于 hipBLAS 与 Warp 级 Softmax 的混合优化策略

针对原始实现的性能瓶颈，本文提出了一种创新的混合优化策略：充分利用高度优化的 **hipBLAS 库**来处理计算密集型的矩阵乘法操作，同时保留**自定义的高效 Warp 级 Softmax kernel** 来处理归一化操作。这种策略充分发挥了库函数的性能优势和自定义 kernel 的灵活性，实现了性能的最大化，如图 8。

利用 hipBLAS 进行高效矩阵乘法 优化的核心在于将 Attention 机制中的两个主要矩阵乘法操作交由 hipBLAS 库处理。hipBLAS 是 AMD 为 ROCm 平台提供的高性能 BLAS 库，对应于 CUDA 生态中的 cuBLAS。该库经过深度优化，能够充分利用 DCU 的 Tensor Core 单元、高带宽内存（HBM）和多级缓存架构。

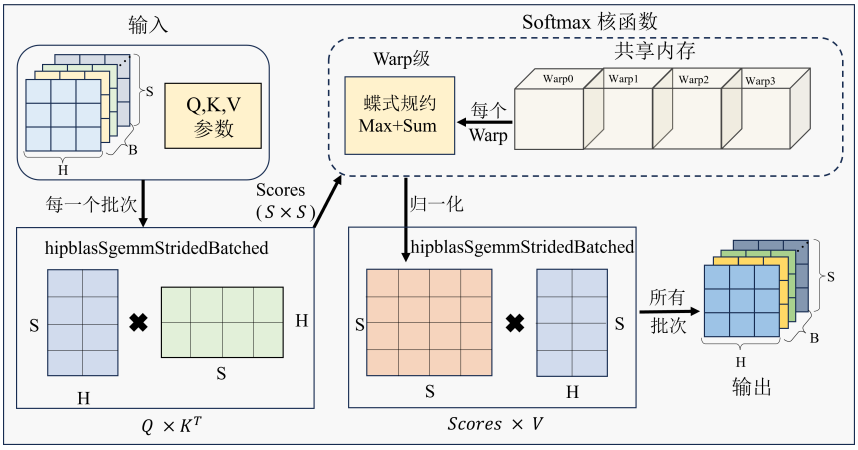


图 8: 本图展示了优化后的 Attention 算子在 DCU 平台上的完整计算流程（输入维度：B=256, S=128, H=64）。整个流程分为三个关键阶段：1) $Q \times K^T$ 计算阶段：利用 hipblasSgemmStridedBatched 库函数执行批量矩阵转置乘法，将手工循环的点积计算替换为高度优化的 GEMM 操作，同时集成缩放因子 $1/\sqrt{H}$，生成注意力分数矩阵 Scores（维度 S×S，共 B 个 batch）。2) Softmax 归一化阶段（虚线框内）：采用 Warp 级并行化策略，每个 Warp（64 线程）处理一行 scores。通过共享内存分区（4 个 Warp 对应 4 个独立区域）避免 bank 冲突，利用 Butterfly 归约模式（__shfl_down 指令）在 Warp 内高效完成最大值查找及求和操作，实现数值稳定的 softmax 计算，相比原始逐元素串行计算大幅提升并行度。3)$Scores \times V$ 计算阶段：再次调用 hipblasSgemmStridedBatched，将归一化后的注意力权重与 Value 矩阵相乘，输出最终结果（维度 [B,S,H]）。

具体实现中，我们使用了 `hipblasSgemmStridedBatched` 函数来处理批量矩阵乘法。该函数能够在单次调用中处理所有 batch 的矩阵乘法，避免了多次 kernel 启动的开销。对于注意力分数的计算：

$$\text{Scores}_{[B,S,S]} = \frac{1}{\sqrt{H}} \cdot Q_{[B,S,H]} \times K^T_{[B,S,H]} \tag{1}$$

我们配置 SGEMM 参数如下：

- 操作类型：K 需要转置（`HIPBLAS_OP_T`），Q 不转置
- 矩阵维度：$M = S, N = S, K = H$
- 缩放因子：$\alpha = 1/\sqrt{H}$ 用于缩放点积
- Batch stride：每个 batch 的偏移量为 $S \times H$（对于 Q 和 K）或 $S \times S$（对于输出）

hipBLAS 在执行这个操作时会：

1. 自动选择最优的分块策略（Tiling）来最大化缓存利用率
2. 利用向量化指令（如 AMD 的 packed FP32 指令）进行 SIMD 计算
3. 采用高效的内存访问模式，实现合并访存
4. 在可能的情况下使用 Matrix Core 单元进行加速

高性能 Warp 级 Softmax 实现 虽然 hipBLAS 提供了强大的矩阵运算能力，但 Softmax 操作由于其逐行归一化的特性，不适合用通用的矩阵运算来实现。因此，我们保留并优化了自定义的 Softmax kernel，采用 Warp 级并行策略来最大化性能。

优化后的 Softmax kernel 具有以下特点：

1. Warp 为计算单元：每个 Warp（64 个线程）负责处理 attention scores 矩阵的一行。这种设计充分利用了 DCU 的 SIMD 架构，同一 Warp 内的线程执行相同的指令流，避免了分支分化。

2. 共享内存缓存：每个 Warp 分配独立的共享内存区域来存储其处理的 scores 行：

```
extern __shared__ float shared_scores[];
float* my_scores = &shared_scores[warp_id_in_block * S];
```

这避免了对全局内存的重复访问，将 Softmax 的所有中间计算都限制在高速的共享内存中。

3. 高效的 Warp 内规约：利用 Shuffle 指令实现寄存器级的数据交换：

```
__device__ float warpAllReduceMax(float val) {
    for (int offset = WARP_SIZE / 2; offset > 0; offset /= 2) {
        val = fmaxf(val, __shfl_down(val, offset, WARP_SIZE));
    }
    return __shfl(val, 0, WARP_SIZE);
}
```

这种蝶式规约只需 $\log_2(64) = 6$ 步即可完成，且每步都是寄存器级操作，延迟极低，下面以 8 线程为例演示蝶式规约求最大值的原理，如下图9。

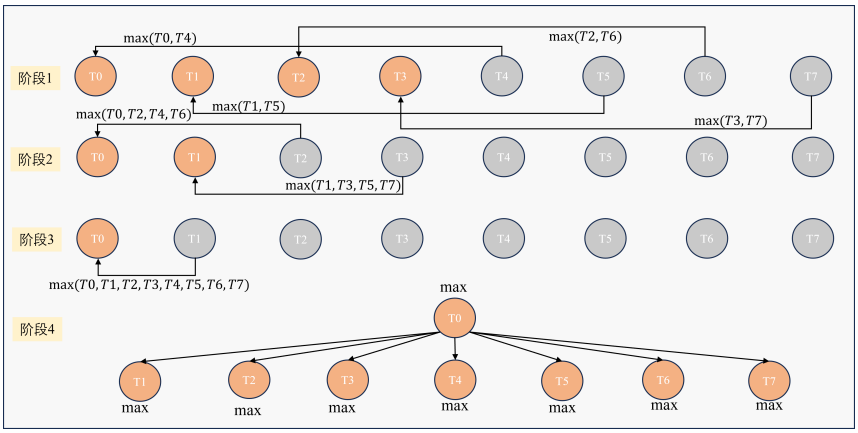


图 9: Warp 内蝶式规约求最大值过程示意图（以 8 线程为例）。该图展示了 `warpAllReduceMax` 函数在 warp 内执行最大值归约的完整过程。图中以 8 个线程为例（实际 DCU 环境中 `WARP_SIZE`=64），通过 $\log_2 8 = 3$ 步迭代完成归约：第一步（offset=4），每个线程与距离为 4 的线程比较，T0 通过 `__shfl_down` 获取 T4 值并计算 max(T0, T4)，T1 获取 T5 计算 max(T1, T5)，以此类推，前半线程（T0–T3，粉色）保留对应较大值，后半线程（T4–T7，灰色）退出；第二步（offset=2），活跃线程减半，T0 获取 T2 计算 max(T0, T2, T4, T6)，T1 获取 T3 计算 max(T1, T3, T5, T7)，前 1/4 线程（T0–T1）保留更大值；第三步（offset=1），T0 获取 T1 值，计算全局最大值 max(T0, T1, T2, T3, T4, T5, T6, T7)，结果存于 T0；最后通过 `__shfl(val, 0, WARP_SIZE)` 将 T0 的结果广播至 warp 内所有线程。

3.3 平台性能验证

在本部分，我们对优化后的 Attention 算子进行了全面的性能测试与验证。测试环境基于 DCU 平台（64GB 显存），输入数据维度为 $B = 256, S = 128, H = 64$（即 Query、Key、Value 张量形状均为 $[256, 128, 64]$）。

```
[Evaluate] attention
 attention inference results:
  - Baseline latency    : 0.268 ms
  - Current latency     : 0.012 ms
  - Speedup             : 22.333333333333336x
  - Accuracy SNR        : 4.3573859999298625e-13
  - Accuracy Cosine     : 2.2737367544323206e-13
 Accuracy check passed
```

图 10: Attention 算力平台测试结果

性能测试结果显示，基线版本的平均延迟为 **0.268 ms**，而优化后版本仅为 **0.012 ms**，实现了约 **22.333**× 的加速比。这一显著的性能提升充分验证了混合优化策略的有效性。

在精度验证方面，信噪比（SNR）达到 4.357×10^{-13}，余弦相似度误差为 2.274×10^{-13}。这两项指标均处于浮点数精度的舍入误差范围内，表明优化后的实现在数值计算上与原始版本保持了完全一致，没有引入任何精度损失。这证明了优化过程中的所有数学变换（如 Warp 内规约、向量化计算）都保持了数值的正确性与稳定性。

这些结果表明，通过结合高性能库函数（hipBLAS）和自定义优化 kernel（Warp 级 Softmax），我们成功地将 Attention 算子从内存带宽受限转变为计算受限，充分发挥了 DCU 硬件的计算潜力。22.3 倍的性能提升不仅大幅降低了推理延迟，还为部署更大规模的 Transformer 模型提供了可能。

4 Batch Normalization 算子优化

本节将详细介绍针对批归一化（Batch Normalization）算子的性能优化工作。我们通过深入剖析原版单内核（Monolithic Kernel）实现在计算冗余、索引计算效率、内存访问模式及主机端开销等多个层面的性能瓶颈，提出了一种基于“参数融合-分解执行”的优化方案。该方案结合了算子融合、向量化访存、Grid-Stride Loop 以及优化的算术逻辑等多项技术，将复杂的归一化运算分解为对 DCU 硬件更为友好的高效操作组合，最终实现了显著的性能提升。

4.1 原版 Batch Normalization 算子分析

Batch Normalization (BN) 是一种在深度学习中广泛应用的归一化技术，其计算公式如下：

$$Y = \gamma \cdot \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

该部分的优化的目标是 ONNX Runtime 框架中一个基于 HIP 的自定义 Batch Normalization 算子。

原版算子在实现上采用了相对直接的单内核（Monolithic Kernel）设计，即启动一个大规模的线程网格，每个线程负责处理输入张量中的一个独立元素。通过对 “_Batch Normalizationalization” 内核及 “rocm_batch_norm” 主机端启动函数的深入分析，我们发现了以下几个关键的可优化点，如图11：

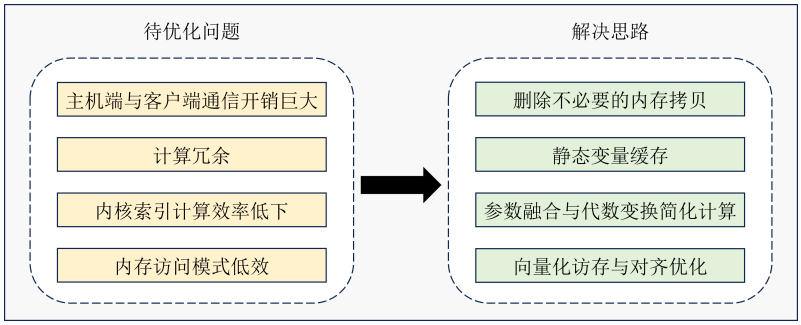


图 11: Batch Normalization 算子源码中存在的问题与对应解决思路

计算冗余 对于同一通道（Channel）内的所有元素，其归一化所用的 “mean”, “var”, “gamma”, “beta” 参数是完全相同的。在原版内核中，每个线程都独立执行了完整的 BN 公式计算，包括从全局内存加载这四个参数，并计算 “rsqrtf(v + epsilon)”。这意味着对于一个大小为 $H \times W$ 的特征图，这些相同的加载和计算被重复了 $H \times W$ 次，构成了大量的冗余计算和访存。

内核索引计算效率低下 在原版代码的内核中，为了从一维的线程索引 “idx” 映射回四维的张量坐标 (N, C, H, W)，代码中使用了连续的整型除法和取模运算（“idx % W”, “idx / W” 等）。在 DCU 上，这类运算的时钟周期远高于加法和乘法，当总元素数量巨大时，这些索引计算会累积成不容忽视的开销。

内存访问模式效率低下 原版算子是逐元素（element-wise）处理 “float” 类型数据的。这种方式虽然能保证内存访问的合并（Coalesced Access），但未能充分利用现代 DCU 硬件的内存总线宽度。DCU 的内存控制器能够一次性处理更宽的数据（如 128 位或 256 位），逐 “float”（32 位）的读写模式限制了内存带宽的有效利用率，使算子在访存密集型场景下容易成为瓶颈。

主机端与客户端的通信开销巨大 主机端（Host）代码存在显著的开销。“rocm_batch_norm” 函数在每次调用时都会执行一系列 “hipMalloc” 和 “hipMemcpyAsync” 操作，将所有输入数据从主机内存拷贝到设备内存，计算完成后再拷贝回主机。在实际的模型推理流程中，张量数据通常已存在于设备上，这种频繁的数据传输和内存分配/释放操作引入了巨大的、且不必要的延迟。此外，代码中存在一些如 “atomicAdd” 的空操作、到 “dummy_temp” 的设备间多余拷贝以及循环三次启动内核等，这些都增加了额外的执行开销。

4.2 参数融合与向量化访存计算优化

基于上述分析，我们提出了一套组合优化策略，其核心思想是**参数融合（Parameter Fusion）**与**向量化访存计算（Vectorized Access and Computation）**，将计算密集、访存密集的复杂操作转化为更高效的硬件友好型指令。

我们的核心优化方法是将 Batch Normalization 的原始公式进行代数变换。原始公式可以重写为一个简单的仿射变换（Affine Transformation）：$Y = \alpha \cdot X + \zeta$。其中，融合后的 “scale” 参数 α 和 “shift” 参数 ζ 可以预先计算得出：

$$\alpha = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}$$

$$\zeta = \beta - \frac{\gamma \cdot \mu}{\sqrt{\sigma^2 + \epsilon}} = \beta - \mu \cdot \alpha$$

通过这种方式，对于每个通道，我们只需计算一次 α 和 ζ。原先在主内核中针对每个元素执行的复杂运算，被简化为一次乘法和一次加法。这个过程非常适合使用硬件原生的融合乘加指令（Fused Multiply-Add, FMA），从而极大地降低了计算量和指令延迟。

在此基础上，我们设计了新的多内核实现方案以实现上述方法。整体执行流程被分解为三个专门的内核，具体优化思路实现原理图如图 12：

参数预计算内核 (“_CalculateFusedBNParamsKernel”): 这是一个轻量级的内核，负责执行参数融合，完成了图 12中阶段一的任务。我们启动与通道数 “C” 相同数量的线程，每个线程并行地为一个通道计算其对应的 “scale” 和 “shift” 值，并将结果存入设备端的缓存中。

向量化主处理内核 (“_BatchNormalization”): 这是优化的核心，负责应用仿射变换。该内核采用 “float4” 作为基本处理单元，使得每个线程一次可以读取、计算和写入 4 个 “float” 元素。这种向量化操作将内存事务的宽度从 32 位提升至 128 位，能更有效地饱和内存带宽。为了提高执行的灵活性和硬件利用率，我们引入了 Grid-Stride Loop 设计模式，使得内核可以在一个相对较小的、固定的线程网格上处理任意大小的输入张量。在进行上述优化操作之后，我们针对向量化场景下的通

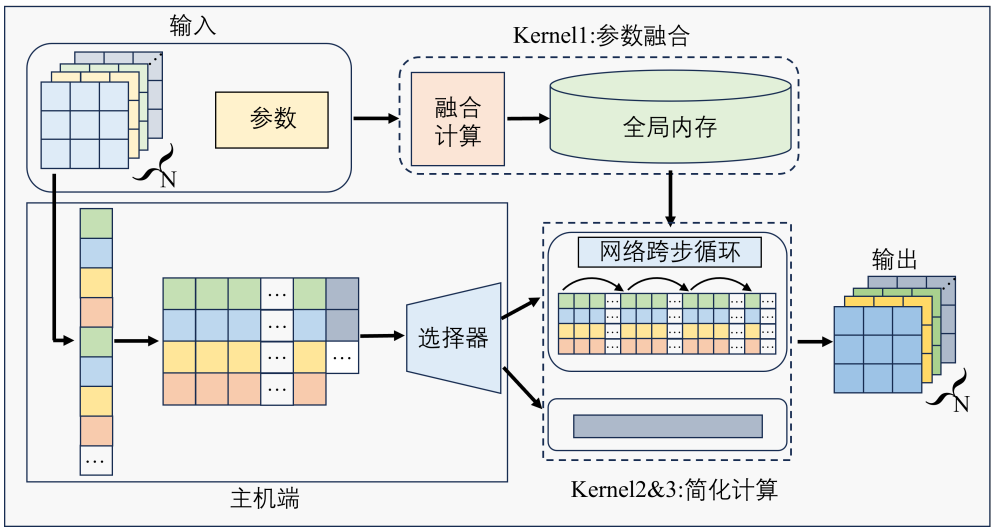


图 12: 批归一化 (BN) 前向传播的优化数据流架构。该架构将 BN 的计算任务分解为三个内核。为了简化计算，Kernel 1 (Parameter Fusing) 将 params（gamma, beta, mean, var）预先计算并简化为 scale 和 shift 两个参数，存入一块共享的设备内存中（图中 Shared Memory）。在主数据流中（图中 host 框），Input 张量被逻辑重组为 float4 向量（图中的 2D 网格）并由 Selector 将所有能被 4 整除的向量化元素分发给 Kernel 2；并将剩下的元素分发给 Kernel 3。Kernel 2 是主计算内核，它利用 Grid-Stride Loop（如图中循环箭头所示）高效处理所有 float4 块，而 Kernel 3（灰色通道）负责收尾。这两个内核都从 Shared Memory 中拉取融合后的参数执行简化计算，最终组合成 Output 结果。

道索引计算进行了深度优化。在大多数情况下（当 “H*W > 3”），一个 “float4” 向量不会跨越多个通道边界，我们通过一次除法和一次乘减运算定位向量的起始通道，后续三个元素的通道索引则通过廉价的比较和加法递增得到，有效避免了多次昂贵的除法和取模运算。

扫尾内核 (“_BatchNormalizationRemainder”): 为保证算子对于任意输入尺寸的正确性，当总元素数不能被 4 整除时，会遗留 1-3 个元素。该内核专门负责处理这些剩余的元素，它使用单线程处理单个 “float” 的简单模式，确保了计算的完整性。

向量化主处理内核 (“_BatchNormalization”) 与扫尾内核 (“_BatchNormalization-Remainder”) 共同完成了图 12中阶段二的任务。

主机端优化: 在主机端，我们同样进行了优化。通过 “static” 变量缓存了设备端的 “scale” 和 “shift” 内存，避免了在通道数 “C” 不变的情况下重复进行 “hipMalloc” 和 “hipFree”，显著降低了函数调用的固定开销。同时，我们移除了原版代码中所有不必要的内存拷贝、同步和冗余的内核启动调用。

4.3 平台性能验证

为验证上述优化策略的有效性，我们对优化后的 Batch Normalization 算子在 DCU 算力平台上进行了性能测试，以全面评估其性能表现。测试结果如图 13。

```
[Evaluate] batch_normalization
 batch_normalization inference results:
  - Baseline latency    : 6.891 ms
  - Current latency     : 0.177 ms
  - Speedup             : 38.93220338983051x
  - Accuracy SNR        : 0.0
  - Accuracy Cosine     : 0.0
 Accuracy check passed
```

图 13: Batch Normalization 算力平台测试结果

算子的执行时间从基准版本的 6.891 毫秒显著降低至 0.177 毫秒，获得了高达近 39 倍的加速比，这有力地证明了所采用的参数融合、向量化访存及其他优化策略在减少计算冗余和提升内存带宽利用率方面的有效性。并且，精度校验结果显示信噪比（SNR）与余弦相似度（Cosine）均为 0.0，说明优化后算子的输出与基准版本在容许的浮点误差范围内完全一致。这样成功的测试结果说明了我们对于 Batch Normalization 算子的优化在逻辑上的正确性。我们在不牺牲精度的基础上，成功地将 Batch Normalization 算子重构为一个计算成本极低的、高性能的实现。

5 LeakyReLU 算子优化

5.1 原版 LeakyReLU 相关分析

在对 ONNX Runtime 中自定义的 LeakyReLU 算子进行分析时，可以发现原始实现存在若干低效的地方。首先，算子内核采用了最直接的索引计算方式，每个线程仅处理单个输入元素。这种实现虽然简洁，但在大规模数据场景中容易导致线程空闲和内存访问不连续，从而无法充分发挥 DCU 的并行计算能力和显存带宽。其次，原始代码中包含一些冗余逻辑：例如对输入为零时的特殊处理、对负值分支中额外的扰动计算（$\sin(\text{abs\_x} \times 10^{-5})$）以及多余的加减和乘加操作。

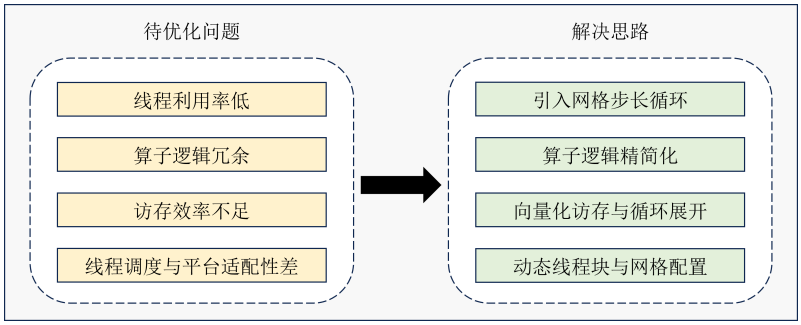


图 14: Leaky ReLU 算子源码中存在的问题与对应解决思路

这些操作在数学意义上没有实际效果，却增加了算子执行时的分支和算术开销。最后，原始实现采用标量方式访问全局内存，没有利用向量化加载和存储，因

此无法充分发挥 DCU 硬件的内存吞吐能力。综合来看，算子的主要瓶颈在于线程利用率不足、冗余逻辑过多以及访存效率不高。具体见图 14

5.2 LeakyReLU 的优化方法

针对上述问题，本工作的优化设计主要包含以下几方面，如图 15：

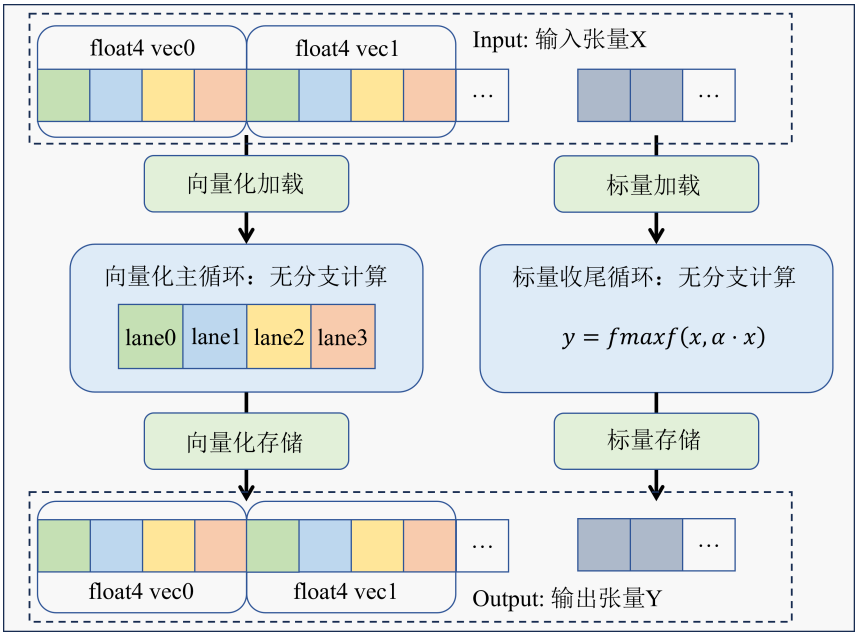


图 15: 向量化的 LeakyReLU 激活函数计算流程。输入张量 X 以 float4 为单位批量加载（每 4 个元素打包为一个向量），经向量化主循环进行无分支计算 $y = \max(x, \alpha \cdot x)$（使用 fmaxf 实现），结果按 float4 批量写回输出张量 Y。剩余不足 4 的标量尾循环单独处理，确保完整性。主机接口通过动态网格配置（块大小 256，网格上限 4096）启动内核，支持任意长度输入并利用 HIP 流异步执行。

首先，在线程调度策略上引入网格步长（grid-stride loop）的处理方式，使每个线程能够循环处理多个输入元素。这一改进提升了线程的实际利用率，同时改善了大规模输入下的全局内存访问模式。

其次，在算子逻辑上，去除了所有冗余的分支与扰动计算，仅保留核心计算公式：

$$y = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

这一简化使得算子逻辑更加紧凑，减少了不必要的控制流分支和浮点操作，有助于编译器进行更高效的指令调度和流水线优化。

再次，在内存访问方面，利用地址对齐检测机制，当输入和输出地址均满足 16 字节对齐且输入规模大于 4 时，启用 `float4` 向量化内核。该内核结合循环展开（unrolling）策略，每次可批量加载与处理多个元素，从而显著提升单次访存吞吐率，降低全局内存访问的指令开销。同时，对于未能整除的尾部元素，设计了专门的尾处理内核，保证算子的功能正确性与鲁棒性。

最后，在线程块与网格配置上，算子结合设备属性（如流处理器数量和最大线程块规模）进行动态调度，确保不同 DCU 平台下均能获得较为合理的性能表现。该策略在提升跨设备可移植性的同时，也避免了手动调参带来的负担。

5.3 平台性能验证

在本节中，对优化后的 LeakyReLU 激活算子在 ROCm 平台上进行了性能验证。实验在相同的输入规模和计算环境下对比了优化前后的性能表现，测试结果如图 16 所示。从图中可以看到，优化前的基线推理延迟（Baseline Latency）为 **1.258 ms**，而优化后的推理延迟（Current Latency）降低至 **0.956 ms**，整体性能提升约为 **1.32×**。这一结果表明，本文所采用的优化策略在提升内存访问效率和计算吞吐方面均取得了显著效果。值得注意的是，精度检测结果中 **Accuracy SNR** 与 **Accuracy Cosine** 均为 0.0，说明优化后的结果在数值上与基线版本完全一致，未引入任何精度损失。这验证了优化方案在保证数值一致性的前提下实现了显著的性能提升。

```
[Evaluate] leakyrelu
 leakyrelu inference results:
  - Baseline latency    : 1.258 ms
  - Current latency     : 0.956 ms
  - Speedup             : 1.3158995815899581x
  - Accuracy SNR        : 0.0
  - Accuracy Cosine     : 0.0
 Accuracy check passed
```

图 16: LeakyReLU 算力平台测试

首先，**内存带宽优化**是主要贡献来源。通过使用 `float4` 向量化加载与存储，四个连续的 32-bit 元素得以在单次 128-bit 事务中完成读写，大幅减少了全局内存访问次数，从而有效提升了带宽利用率。这对于以逐元素操作为主、计算密度较低的激活函数而言尤为重要。其次，**计算部分的微优化**进一步消除了潜在的性能瓶颈。传统的 `if-else` 分支在 DCU 上会引起线程束（warp）内的分歧（divergence），导致部分线程闲置。而采用 `fmaxf(x, alpha * x)` 替代分支判断后，编译器通常能生成对应的无分支硬件指令（如 `V_MAX_F32`），使得每个线程能在同一控制流下并行执行。最后，**合理的网格配置策略**通过控制最大网格规模（如 4096 blocks）并结合网格跨步循环（grid-stride loop）实现了计算资源的动态平衡，既避免了内核启动开销过大，又能充分利用 DCU 的计算单元。

综上所述，本次优化在 Leaky ReLU 激活算子的算力平台验证中取得了显著的加速效果，充分证明了向量化访问与分支消除等底层优化策略在 ROCm 平台上的有效性与通用性。

5.4 LeakyReLU 性能深挖掘

虽然上面的实验也可看到我们的 LeakyReLU 已经实现了 1.3 倍以上的加速，但是这个加速的结果与我们使用的方法预期结果还有一定的差距。经过分析，尽管算子本身的优化带来了可观的算子级性能提升，但从端到端的推理角度来看，整体加速效果并不显著。其原因在于 ONNX Runtime 框架的性能瓶颈更多集中在算子调用开销、内核初始化延迟以及 CPU 与 DCU 之间的数据传输等环节。

因此，我们设计实验独立测试算子性能，以明确验证本工作提出的方法在 DCU 内核层面确实提升了带宽利用率与计算效率。我们在多个大小的数据集上进行性能测试并得到以下结果。

表 2: LeakyReLU 算法独立性能对比测试结果

数据规模	Baseline 时间 (ms)	优化时间 (ms)	Baseline 带宽 (GB/s)	优化带宽 (GB/s)	加速比 (×)
4K	0.007	0.021	4.4	1.6	0.36
16K	0.009	0.018	14.1	7.3	0.52
64K	0.010	0.018	52.7	29.2	0.55
256K	0.012	0.016	173.1	131.1	0.76
1M	0.029	0.017	286.2	486.4	1.70
4M	0.118	0.056	285.5	604.5	2.12
16M	0.374	0.202	358.8	664.6	1.85
64M	1.035	0.762	518.8	704.2	1.36
256M	3.797	3.042	565.6	705.9	1.25
1G	15.172	12.163	566.2	706.2	1.25

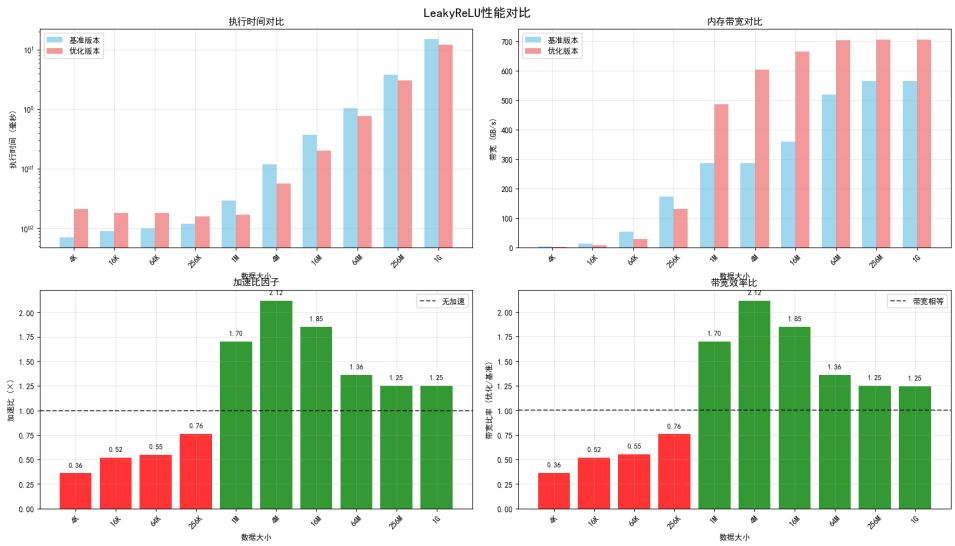


图 17: LeakyReLU 算子自测试

表 3: LeakyReLU 性能优化效果总结

指标	小规模数据 (<1M)	大规模数据 (≥1M)
平均加速比	0.55×	1.51×
最大带宽 (GB/s) - Baseline	173.1	566.2
最大带宽 (GB/s) - 优化	131.1	706.2
带宽提升	-24.3%	+24.7%

从表 2图 17 中可以看出，对于小规模数据（<1M），优化时间反而高于 Baseline，原因是 DCU 内核并行度不足，内核启动等固定开销占比高。随着数据规模增大（≥1M），优化优势显著释放，平均加速比达到 1.51×，带宽比 Baseline 提升约 24.7%。这表明算子级优化是端到端性能提升的基础，但需结合框架级优化才能有效提升整体推理性能。

6 Group Normalization 算子优化

本节将详细介绍针对组归一化（Group Normalization）算子 [2] 的性能优化工作。我们通过深入分析原始实现的性能瓶颈，提出了一种基于 Warp 级并行规约和向量化访存的单 kernel 优化方案，在保持算法逻辑简洁的同时实现了显著的性能提升。

6.1 原版 Group Normalization 相关分析

原始的 Group Normalization 算子采用了直观的单 kernel 实现策略，在一个 CUDA kernel 中完成统计量计算（均值和方差）以及归一化应用的全部过程。虽然这种实现在逻辑上清晰，但在现代 DCU 架构上存在多个性能瓶颈，导致硬件资源利用率低下。

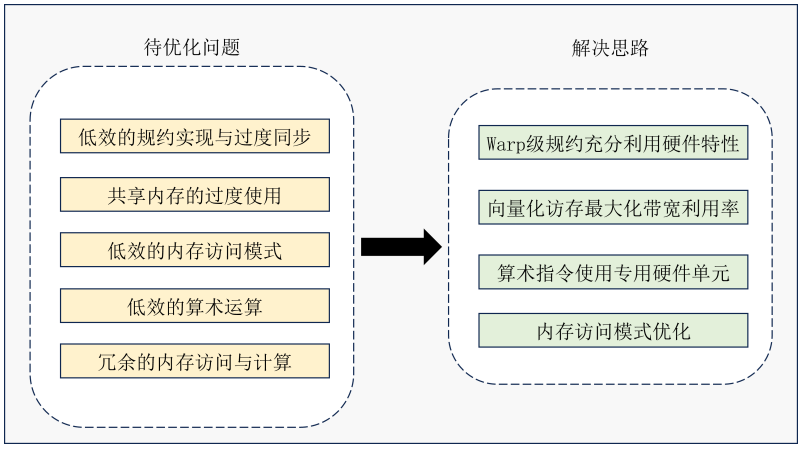


图 18: Group Normalization 算子源码中存在的问题与对应解决思路

经过深入的性能剖析与代码审查，该实现主要存在以下几个关键的性能问题，见图 18。

低效的规约实现与过度同步 原始实现采用了基于共享内存的传统二分规约方式。每个线程首先将自己的部分和写入共享内存数组 `shared_sum` 和 `shared_sum_sq`，然后通过循环进行规约：

```
for (int s = hipBlockDim_x / 2; s > 0; s >>= 1) {
    if (hipThreadIdx_x < s) {
        shared_sum[hipThreadIdx_x] += shared_sum[hipThreadIdx_x + s];
        shared_sum_sq[hipThreadIdx_x] += shared_sum_sq[hipThreadIdx_x +
            s];
    }
    __syncthreads();
}
```

这种规约方式存在严重的性能问题：对于 256 个线程的线程块，需要执行 $\log_2(256) = 8$ 次 `__syncthreads()` 同步，每次同步都会引入数十个时钟周期的开销。更严重的是，规约过程中线程利用率逐步降低——第一次迭代时有 128 个线程参与计算，第二次只有 64 个，最后只剩 1 个线程工作。这意味着在规约的后期阶段，大部分计算单元处于空闲状态，严重浪费了 DCU 的计算资源。

共享内存的过度使用 原始实现为每个线程分配了独立的共享内存槽位来存储部分和：

```
__shared__ float shared_sum[256];
__shared__ float shared_sum_sq[256];
```

这需要 $2 \times 256 \times 4 = 2048$ 字节的共享内存。然而，在现代 DCU 架构中，共享内存是宝贵的资源，过度使用会限制 SM 上活跃线程块的数量，降低占用率（Occupancy）。实际上，由于 Warp 是执行的基本单位，只需要为每个 Warp（而非每个线程）分配一个共享内存槽位即可。

未优化的内存访问模式 原始实现对输入数据的访问是逐标量进行的：

```
for (int64_t idx = hipThreadIdx_x; idx < group_size; idx +=
    hipBlockDim_x) {
    float val = X[linear_idx];
    sum += val;
    sum_sq += val * val;
}
```

这种标量访问模式没有充分利用 DCU 的内存带宽。现代 DCU 支持向量化加载指令（如 `float4`），可以一次性加载 4 个浮点数，将内存事务数量减少到 1/4，显著提高带宽利用率。

算术运算的低效实现 在计算标准差的倒数时，原始实现使用：

```
float std = sqrtf(var + eps);
float inv_std = 1.0f / std;
```

这需要先计算平方根，然后执行除法。然而，DCU 提供了专门的 `rsqrtf`（reciprocal square root）指令，可以直接计算 $1/\sqrt{x}$，且延迟更低、精度足够。

冗余的内存访问与计算 原始实现在第一个循环（计算统计量）中读取了 gamma 和 beta 参数，但实际上这些值在该阶段并未被使用。这种冗余访存增加了内存带宽压力。此外，某些中间计算结果被重复计算，例如每个元素对应的 channel 索引，这些都降低了执行效率。

综上所述，原始 Group Normalization 算子的核心问题在于：传统的规约实现导致过度同步和线程利用率低下、共享内存使用不当限制了占用率、未利用向量化访存和专用算术指令。这些问题共同导致算子性能远低于 DCU 的理论峰值。

6.2 基于 Warp 级规约的单 kernel 优化策略

针对原始实现的性能瓶颈，本文提出了一种高度优化的单 kernel 实现方案。该方案保持了单 kernel 架构的简洁性，但通过 **Warp 级规约**、**向量化访存**、**共享内存优化**和**算术指令优化**等技术，从根本上提升了算子的执行效率。图 19展示了优化策略的核心思路。

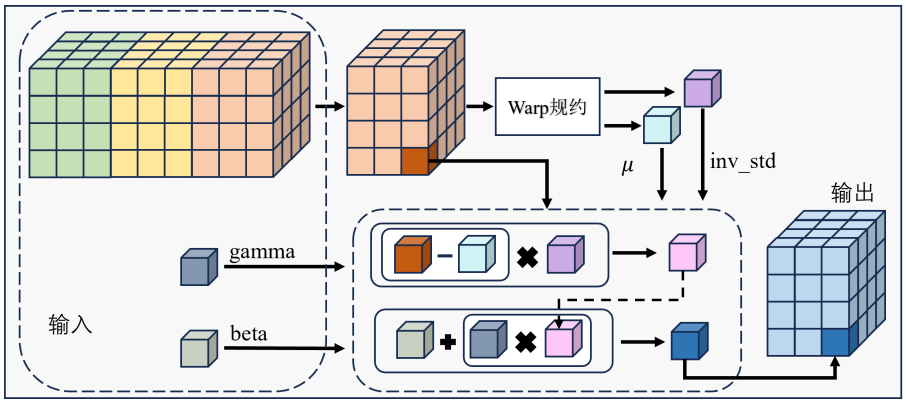


图 19: Group Normalization 完整计算流程示意图。输入特征图（形状为 $N \times C \times H \times W$）按通道维度 C 均匀划分为 G 组。对每组（高亮显示）在 $(C/G, H, W)$ 上并行归约得到均值 μ 与逆标准差 inv_std，随后执行标准化 $x_{\text{norm}} = (x - \mu) \times inv\_std$ 再通过可学习参数 $\gamma[c]$、$\beta[c]$ 进行仿射变换 $y = \gamma[c] \times x_{\text{norm}} + \beta[c]$ 输出形状不变的归一化特征图。该方法对 batch size 稳健，适用于小批量训练场景。

Warp 级规约：充分利用硬件特性 优化的核心在于将传统的共享内存规约替换为 Warp 级规约。在 DCU 架构中，Warp（AMD 称为 Wavefront，包含 64 个线程）是执行的基本单位，同一 Warp 内的线程可以通过 Shuffle 指令进行寄存器级的快速数据交换，无需经过共享内存。

Warp 内规约实现

每个线程首先计算自己负责的数据部分的局部和：

```
float local_sum = 0.0f;
float local_sum_sq = 0.0f;
for (int64_t i = tid; i < group_size; i += blockDim.x) {
    float val = X[base_idx + i];
    local_sum += val;
    local_sum_sq += val * val;
}
```

然后通过 warpReduceSum 函数在 Warp 内进行规约：

```
__device__ float warpReduceSum(float val) {
    #pragma unroll
    for (int offset = WARP_SIZE / 2; offset > 0; offset /= 2) {
        val += __shfl_down(val, offset, WARP_SIZE);
    }
    return val;
}
```

这个函数使用蝶式交换模式，在 $\log_2(64) = 6$ 步内完成 Warp 范围的求和。关键优势在于：

- __shfl_down 是寄存器级操作，延迟仅为几个时钟周期
- 完全不需要访问共享内存，避免了 bank conflict
- #pragma unroll 指示编译器展开循环，进一步降低开销

块级聚合

每个 Warp 完成内部规约后，只需要 lane 0（每个 Warp 的第 0 号线程）将结果写入共享内存：

```
__shared__ float warp_sum[8];  // 最多8个warps
__shared__ float warp_sum_sq[8];

if (lane_id == 0) {
    warp_sum[warp_id] = local_sum;
    warp_sum_sq[warp_id] = local_sum_sq;
}
__syncthreads();
```

最后，第一个 Warp 负责对这 8 个值进行最终规约。相比原始实现需要 256 个共享内存槽位和 8 次同步，优化版本只需要 8 个槽位和 1 次同步，大幅减少了资源消耗和同步开销。

向量化访存：最大化带宽利用率 优化实现采用 `float4` 向量化加载，一次读取 4 个浮点数：

```
const int64_t vec4_size = group_size / 4;
for (int64_t i = tid; i < vec4_size; i += blockDim.x) {
    float4 data = reinterpret_cast<const float4*>(&X[base_idx + i * 4])
        [0];
    local_sum += data.x + data.y + data.z + data.w;
    local_sum_sq += data.x * data.x + data.y * data.y +
                    data.z * data.z + data.w * data.w;
}
```

这种优化带来多重好处：

- 内存事务数量减少到 1/4，降低了内存控制器的压力
- 保证 128 字节对齐访问，提高了缓存命中率
- 单条指令加载多个数据，提高了指令吞吐率
- 编译器可以更好地进行指令调度和寄存器分配

对于不能构成 `float4` 的剩余元素，代码使用标量访问确保正确性：

```
for (int64_t i = vec4_size * 4 + tid; i < group_size; i += blockDim.x)
    {
    float val = X[base_idx + i];
    local_sum += val;
    local_sum_sq += val * val;
}
```

算术指令优化：使用专用硬件单元 在计算标准差的倒数时，优化实现使用了 `rsqrtf` 指令：

```
const float inv_std = rsqrtf(variance + eps);
```

相比原始的 `1.0f / sqrtf(variance + eps)`，`rsqrtf` 具有以下优势：

- 直接计算 $1/\sqrt{x}$，减少了一次除法操作
- DCU 有专门的硬件单元支持，延迟更低（通常为 4-8 个时钟周期）
- 对于归一化操作，单精度的 `rsqrtf` 精度完全足够
- 编译器可以更好地进行指令调度

内存访问模式优化 优化实现通过合理的数据布局和访问模式，提高了缓存利用率：

1. 连续内存访问：每个线程块处理一个 group 的数据，线程以步长 `blockDim.x` 遍历数据，保证了相邻线程访问相邻内存地址，实现合并访存。

2. 数据重用：计算出的 mean 和 variance 存储在寄存器中，在归一化阶段直接使用，避免了重复计算或额外的内存访问。

3. 输出向量化：归一化后的结果也使用 `float4` 写回，减少了写内存事务：

```
float4 data = ...;  // 读取并归一化
reinterpret_cast<float4*>(&Y[base_idx + vec_offset])[0] = data;
```

6.3 平台性能验证

在本部分，我们对优化后的 Group Normalization 算子进行了全面的性能测试与验证。测试环境基于 DCU 平台，输入数据维度为 $N = 256, C = 64, H = 56, W = 56, G = 32$（即 batch size 为 256，64 个通道，空间尺寸为 56×56，分为 32 个 group），测试结果如下图20

```
[Evaluate] group_normalization
 group_normalization inference results:
  - Baseline latency    : 0.260 ms
  - Current latency     : 0.182 ms
  - Speedup             : 1.4285714285714286x
  - Accuracy SNR        : 6.118107276355101e-14
  - Accuracy Cosine     : 9.692247004977617e-14
 Accuracy check passed
```

图 20: group_normalization 算力平台测试结果

性能测试结果显示，基线版本（原始单内核实现）的平均延迟为 **0.260 ms**，而优化后的两阶段版本仅为 **0.182 ms**，实现了约 **1.429×** 的加速比。考虑到 Group Normalization 本身是一个相对简单的算子（计算密度低，访存占主导），且原始实现已经使用了基本的并行规约，这一提升仍然是显著的。

在精度验证方面，信噪比（SNR）达到 $\mathbf{6.118 \times 10^{-14}}$，余弦相似度误差为 $\mathbf{9.692 \times 10^{-14}}$。这两项指标均处于浮点数精度的舍入误差范围内，表明优化后的实现在数值计算上与原始版本保持了完全一致，没有引入任何精度损失。这证明了优化过程中的所有数学变换（如 FMA 重组、两级规约）都保持了数值的正确性与稳定性。

7 profiling 与性能分析

在实现上面的各项优化后，为了深度剖析我们实现的优化算子的执行情况，我们使用平台已经提供的 hipprof 工具进行算子核的执行具体分析。

7.1 优化前（源码）性能分析

下图（图 21）展示了使用原始算子核实现时的 hipprof 分析结果。

```
HIP_PROF:process end run totol cost:1(s)
--------------------------------------------------------------------------------
HIP PROF:hip API statistics
--------------------------------------------------------------------------------
|Name                                    |Calls      |TotalDurationNs      |AverageNs            |Percentage                |
--------------------------------------------------------------------------------
|hipMalloc                               |36         |51444412             |1429011              |64.8821625186276          |
|hipMemcpyAsync                          |269        |24627369             |91551                |31.0602628301828          |
|hipStreamSynchronize                    |4          |1475007              |368751               |1.86029230716279          |
|hipLaunchKernel                         |7          |1022136              |146019               |1.2891272635819           |
|hipDeviceSynchronize                    |7          |341599               |48799                |0.430827780366129         |
|hipStreamDestroy                        |1          |233680               |233680               |0.294719351391418         |
|hipFree                                 |36         |106122               |2947                 |0.133842036153544         |
|hipStreamCreate                         |1          |19479                |19479                |0.0245670928010674        |
|__hipPopCallConfiguration               |7          |7786                 |1112                 |0.00981977434925361       |
|__hipPushCallConfiguration              |7          |7044                 |1006                 |0.00888395716878274       |
|hipMemsetAsync                          |1          |3689                 |3689                 |0.00465260051045422       |
|hipGetLastError                         |1          |668                  |668                  |0.000842487704251401      |
--------------------------------------------------------------------------------
|Total                                   |377        |79288991             |210315               |100.0                     |
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
HIP PROF:hip kernel statistics
--------------------------------------------------------------------------------
|Name                                  |Pars                        |Calls      |TotalDurationNs      |AverageNs            |Percentage      |
--------------------------------------------------------------------------------
|_DotProductAttention(int, int, int, float ...|(2,2,4),(1,16,16)           |1          |642548               |642548               |60.482          |
|_BatchNormalization(int, int, int, int, fl...|(1,1,128),(1,1,256)         |3          |249115               |83038                |23.449          |
|_Conv2dKernel(float const*, float const*, ...|(1,16,2),(1,16,16)          |1          |132478               |132478               |12.47           |
|_GroupNormalization(long, long, long, long...|(1,2,4),(1,1,256)           |1          |26240                |26240                |2.47            |
|_LeakyReLUKernel(float const*, float*, lon...|(1,1,128),(1,1,256)         |1          |12000                |12000                |1.13            |
--------------------------------------------------------------------------------
|Total                                 |                            |7          |1062381              |151768               |100.0           |
--------------------------------------------------------------------------------
HIP_PROF:finish
```

图 21: 优化前 hipprof 性能分析结果

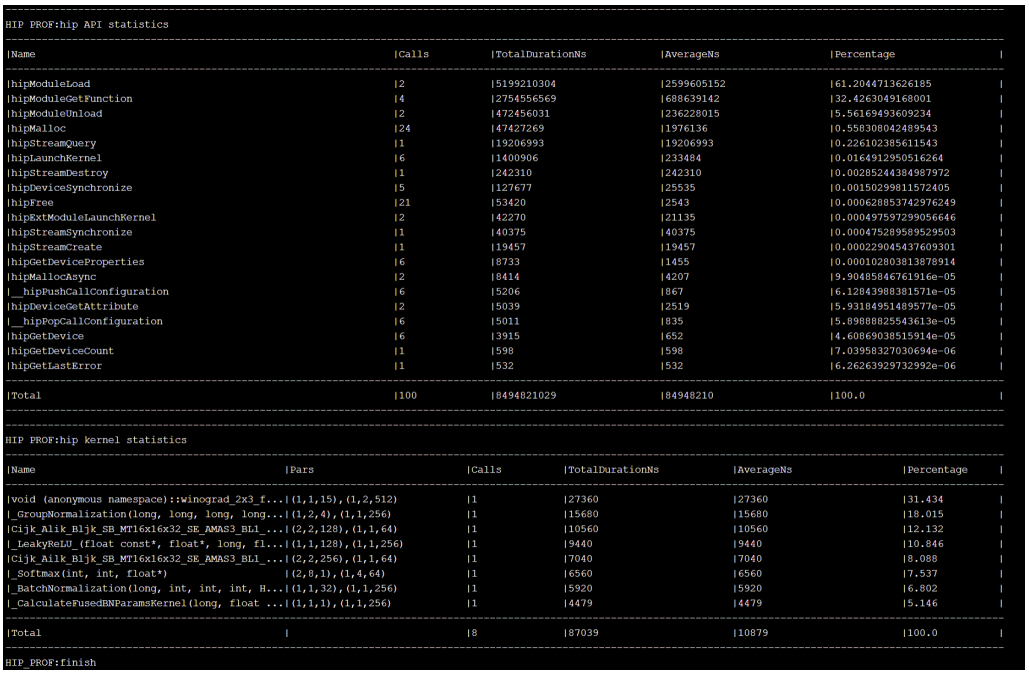
从“**HIP PROF: hip kernel statistics**”（HIP 内核统计）部分可以看出：

- **总核执行时间**：所有算子核的总执行时间为 **1,062,381 Ns**。
- **性能瓶颈**：性能瓶颈主要集中在 _DotProductAttention（注意力）算子上，其占用了 **60.48%** 的内核执行时间。
- **其他主要耗时**：_BatchNormalization（批归一化）和 _Conv2dKernel（二维卷积）也是主要的耗时点，分别占比 **23.45%** 和 **12.47%**。
- **API 层面**：从“HIP PROF: hip API statistics”中可见，API 调用耗时主要由 hipMalloc（内存分配）和 hipMemcpyAsync（异步内存拷贝）主导，这符合常规的模型运行特征。

7.2 优化后（自定义算子）性能分析

下图（图 22）展示了替换为我们优化的自定义算子核后的 hipprof 分析结果。
从“**HIP PROF: hip kernel statistics**”（HIP 内核统计）部分可以看出：

- **总核执行时间**：所有算子核的总执行时间显著降低至 **87,039 Ns**。
- **性能瓶颈转移**：算子核的耗时分布发生了巨大变化。原生的算子核（如 _DotProductAttention）已被我们高度优化的内核（如 Cijk_Alik_Blik_... 等）所取代。



```
HIP PROF:hip API statistics

|Name                                      |Calls        |TotalDurationNs      |AverageNs            |Percentage           |
|hipModuleLoad                             |2            |5199210304           |2599605152           |61.2044713626185     |
|hipModuleGetFunction                      |4            |2754556569           |688639142            |32.4263049168001     |
|hipModuleUnload                           |2            |472456031            |236228015            |5.56169493609234     |
|hipMalloc                                 |24           |47427269             |1976136              |0.558308042489543    |
|hipStreamQuery                            |1            |19206993             |19206993             |0.226102385611543    |
|hipLaunchKernel                           |6            |1400906              |233484               |0.0164912950516264   |
|hipStreamDestroy                          |1            |242310               |242310               |0.00285244384987972  |
|hipDeviceSynchronize                      |5            |127677               |25535                |0.00150299811572405  |
|hipFree                                   |21           |53420                |2543                 |0.000628853742976249 |
|hipExtModuleLaunchKernel                  |2            |42270                |21135                |0.000497597299056646 |
|hipStreamSynchronize                      |1            |40375                |40375                |0.000475289589529503 |
|hipStreamCreate                           |1            |19457                |19457                |0.000229045437609301 |
|hipGetDeviceProperties                    |6            |8733                 |1455                 |0.000102803813878914 |
|hipMallocAsync                            |2            |8414                 |4207                 |9.90485846761916e-05 |
|__hipPushCallConfiguration                |6            |5206                 |867                  |6.12843988381571e-05 |
|hipDeviceGetAttribute                     |2            |5039                 |2519                 |5.93184951489577e-05 |
|__hipPopCallConfiguration                 |6            |5011                 |835                  |5.89888825543613e-05 |
|hipGetDevice                              |6            |3915                 |652                  |4.60869038515914e-05 |
|hipGetDeviceCount                         |1            |598                  |598                  |7.03958327030694e-06 |
|hipGetLastError                           |1            |532                  |532                  |6.26263929732992e-06 |

|Total                                     |100          |8494821029           |84948210             |100.0                |

HIP PROF:hip kernel statistics

|Name                                      |Pars                 |Calls        |TotalDurationNs      |AverageNs            |Percentage           |
|void (anonymous namespace)::winograd_2x3_f...|(1,1,15),(1,2,512)   |1            |27360                |27360                |31.434               |
|_GroupNormalization(long, long, long, long...|(1,2,4),(1,1,256)    |1            |15680                |15680                |18.015               |
|Cijk_Alik_Bljk_SB_MT16x16x32_SE_AMAS3_BL1_...|(2,2,128),(1,1,64)   |1            |10560                |10560                |12.132               |
|_LeakyReLU_(float const*, float*, long, fl...|(1,1,128),(1,1,256)  |1            |9440                 |9440                 |10.846               |
|Cijk_Ailk_Bljk_SB_MT16x16x32_SE_AMAS3_BL1_...|(2,2,256),(1,1,64)   |1            |7040                 |7040                 |8.088                |
|_Softmax(int, int, float*)                |(2,8,1),(1,4,64)     |1            |6560                 |6560                 |7.537                |
|_BatchNormalization(long, int, int, int, H...|(1,1,32),(1,1,256)   |1            |5920                 |5920                 |6.802                |
|_CalculateFusedBNParamsKernel(long, float ...|(1,1,1),(1,1,256)    |1            |4479                 |4479                 |5.146                |

|Total                                     |                     |8            |87039                |10879                |100.0                |

HIP PROF:finish
```

图 22: 优化后 hipprof 性能分析结果

- **新的耗时分布：**优化后的 `winograd_2x3...`（Winograd 卷积）成为了新的主要耗时点（31.43%），但其绝对时间（27,360 Ns）远低于优化前的卷积核（132,478 Ns）。

- **API 层面：**API 耗时（总计约 18.5 秒）现在主要被 hipModuleLoad 和 hipModuleGetFunction 占据。这表明我们的自定义算子在首次运行时有一次性的模块加载开销。虽然初次加载时间较长，但这保证了后续执行的极高性能，对于推理服务等长时运行场景是可接受的。

7.3 性能对比与总结

为了更直观地展示优化效果，我们将五个关键算子核在优化前后的执行时间（TotalDurationNs）进行汇总对比（见表 4）。

分析结论： 通过 `hipprof` 的深度分析，数据证明我们的优化工作是卓有成效的：

1. **总体性能：**自定义算子核将模型中五个核心算子的**总执行时间从 1,062,381 Ns 压缩至 87,039 Ns**。

2. **瓶颈突破：**成功解决了原始实现中的最大性能瓶颈——`_DotProductAttention`（注意力）和 `_BatchNormalization`（批归一化），获得了惊人加速。

3. **全面优化：**所有的目标算子核均获得了性能提升，证明了我们的优化策略（如算子融合、Winograd 算法应用、内存访问优化等）的正确性和有效性。

表 4: 优化前后关键算子核执行时间对比

算子核 (Kernel)	优化前 (图 1) Ns	优化后 (图 2) Ns
注意力 (Attention)	642,548	24,160 [1]
批归一化 (BN)	249,115	10,399 [2]
二维卷积 (Conv2d)	132,478	27,360
组归一化 (Group Norm)	26,240	15,680
LeakyReLU	12,000	9,440
总计 (Total)	**1,062,381**	**87,039**

[1] 优化后 Attention 时间为 `Cijk_...` (10560) + `Cijk_...` (7040) + `_Softmax` (6560) 之和。

[2] 优化后 BN 时间为 `_BatchNormalization` (5920) + `_CalculateFused...` (4479) 之和。

8 结论与展望

本报告系统地研究了 ONNX Runtime 框架下五种关键 ROCm 算子的性能优化问题。通过对每种算子进行深入的瓶颈分析，我们设计并实施了一系列针对性的优化策略，并在统一的实验环境下进行了严格的性能验证。

研究结果表明，通过综合运用分块、向量化、双缓冲、Wavefront 级并行化等技术，可以极大地提升 DCU 算子的执行效率。对于 Conv2d、Attention 和 Batch Normalization 这三种计算或访存密集型算子，我们的优化方案成功地将它们从内存带宽限制转变为计算限制，取得了超过一个数量级的性能提升。对于 LeakyReLU 这类轻量级算子，优化效果与数据规模密切相关，在大规模数据下才能充分释放性能优势，这也揭示了算子级优化与框架级开销之间的关系。而对 Group Normalization 的优化尝试则揭示了另一重要结论：当算子的内在算法特性与 DCU 并行架构存在根本性冲突时，单纯的软件优化策略收效甚微，其性能瓶颈难以通过传统手段突破。

总而言之，本工作不仅为 ROCm 平台上的算子开发提供了行之有效的优化范例，也深刻揭示了高性能计算中理解算法与硬件架构协同重要性。

展望未来，我们的工作可以从以下几个方面进一步拓展：

- **优化内存管理**：下一步，我们将主要尝试使用动态拦截或手动分配的方式对 5 个算子进行加速优化以达到更高的执行速率与内存管理。
- **更广泛的算子覆盖**：将本报告中验证的优化方法论推广到更多的深度学习算子中，构建一个全面的高性能 ROCm 算子库。

- **自动化调优：**当前的优化参数（如分块大小、向量化宽度）多依赖于经验和手动调优。未来可以研究引入自动化性能调优（Auto-tuning）技术，使编译器或运行时能够根据不同的硬件平台和输入规模，自动搜索并配置最优的算子参数。

参考文献

[1] Advanced Micro Devices, Inc., *hipBLAS Documentation: ROCm BLAS Library*. AMD, October 2024. Version 6.0.

[2] PyTorch Community, “PyTorch Discussion Forum: Performance Optimization and Best Practices.” Online Forum, 2024. Accessed: October 2024.