第 16 章、triton 编程

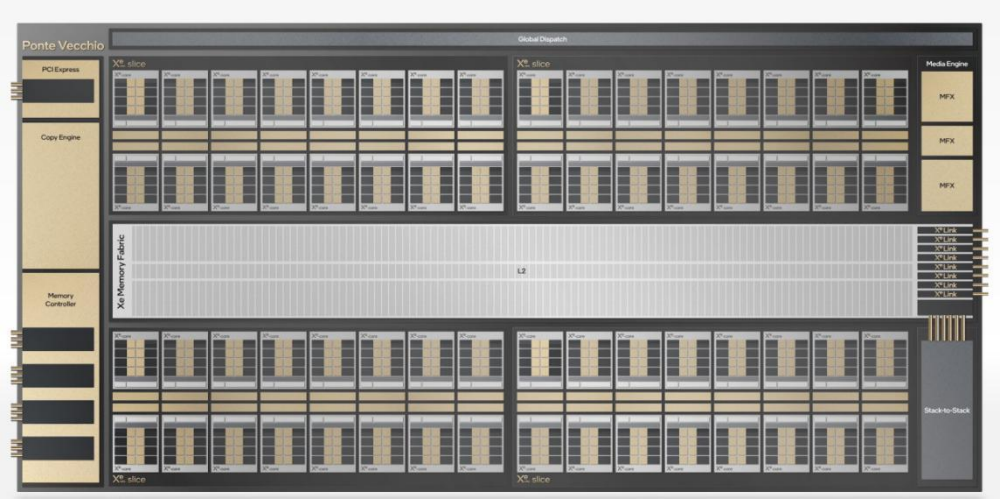
2.1.4 triton 硬件平台与编程范式

Triton 硬件平台

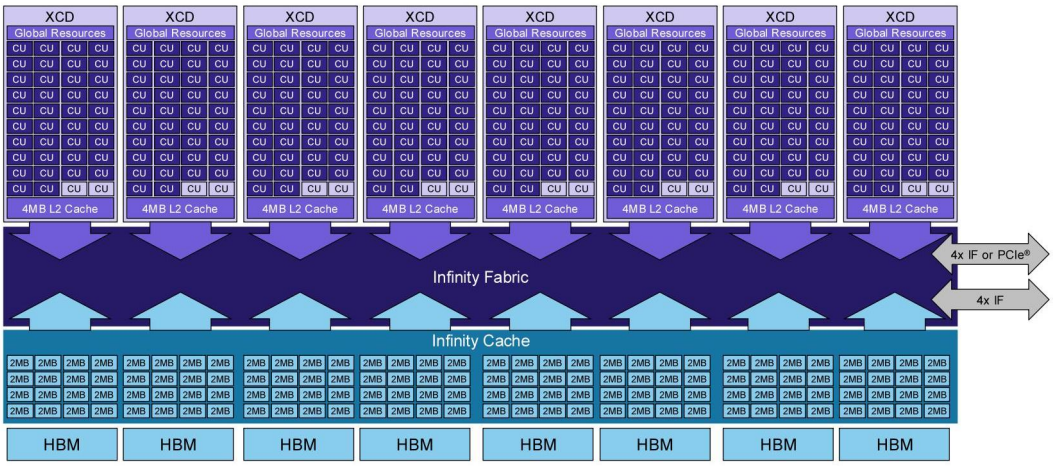
Triton 是 OpenAI 发布的一门开源 Python 风格编程语言及编译器，专门为深度学习中的自定义算子（如稀疏矩阵、Attention、卷积等）加速设计。因 GPU 在深度学习基础设施中的广泛使用，Triton 目前特定于 GPU 做了系列优化，并对不同厂商的 GPU 有着不同程度的兼容和支持，笔者截稿前 Triton 的最新版本为 v3.3.1，对不同厂商 GPU 的具体支持如下：

- NVIDIA GPU（计算能力 ⩾ 8.0 的 Ampere/Ada Lovelace/Hopper/Blackwell）：性能稳定成熟；
- AMD GPU（基于 ROCm ⩾ 6.2 平台），目前已经集成进 Triton 后端；
- Intel GPU（通过 Intel XPU 后端、SYCL 支持 Triton），目前在单独的仓库维护（https://github.com/intel/intel-xpu-backend-for-triton），可通过特定于 Intel GPU 支持的 Pytorch 来安装（https://docs.pytorch.org/docs/stable/notes/get_start_xpu.html）。

低版本的 Triton 支持某些老旧型号的 GPU，如计算能力为 7.0 的 Volta 架构的 NVIDIA GPU，但因不再更新且存在未修复的问题，目前已不建议使用。



图表 13 Intel Ponte Vecchio GPU Architecture



图表 14 AMD MI300X GPU Architecture

图表 15 NVIDIA GB202 GPU Architecture

图表 13 图表 14 图表 15 分别是 Intel、AMD 和 NVIDIA 三家 GPU 厂商的代

表性 GPU 架构图，表格 1 给出了这些厂商 GPU 子结构命名差异。即使它们对 GPU 架构内各子结构的命名存在一定的差异，但结构相似的基本元件具备近乎相同的功能。Triton 的开发者充分的考虑了 GPU 设计的这一特点，从软件层面进行了高度抽象，使得 Triton 的后端设计能够更好地对全架构提供支持。开发者不必关心不同厂商 GPU 架构差异的具体细节，只需掌握好其中一种平台的软硬件架构（笔者建议优先掌握 NVIDIA GPU 的相关内容）就可以使用 Triton 在其它平台实现媲美的峰值性能。

表格 1 不同厂商 GPU 子结构命名差异对比

	Intel 术语	AMD 术语	NVIDIA 术语	说明
最小执行单元	EU（Execution Unit）	SIMD（在 CU 内）	SM（Streaming Multiprocessor）	Intel 的 EU 是基本调度与执行单元；AMD 的 CU 包含多个 SIMD 引擎；NVIDIA 的 SM 是核心模块。
计算核心	ALU（在 EU 内）	SP（Stream Processor）	CUDA Core	三者均为基础计算单元，数量决定并行能力。
线程调度单位	线程（由 EU 调度）	Wavefront（64 线程）	Warp（32 线程）	每厂商使用不同线程分组与调度策略。
向量宽度模	SIMD-8（每	SIMD-16（每	SIMT（每 Warp	Intel 和 AMD

型	EU）	SIMD 引擎）	动态调度）	使用固定宽度 SIMD；NVIDIA 使用 SIMT 模型，调度更灵活。
显存控制器	Memory Controller	Memory Controller	Memory Partition	NVIDIA 的 Memory Partition 包含控制器和 ROP/L2 分区。
光栅化单元	Render Slice	ROP（Render Output Unit）	ROP（Render Output Unit）	Intel 的 Render Slice 包含 ROP 等渲染模块；AMD/NVIDIA 的 ROP 为独立单元。
多 GPU 互联	Xe Link	Infinity Fabric	NVLink	各自专有互联技术，拓扑与带宽差异显著。
架构层级	Tile → Slice → EU	CU → SIMD	GPC → TPC → SM	Intel 使用 Tile 分层设计；AMD 以 CU 为基本单元；NVIDIA

				层级清晰。

Triton 编程范式

我们在前述章节已经接触了并行计算和 CUDA 并行优化的基础知识，这一节主要介绍 Triton 与 CUDA 的异同，以便更好地掌握 Triton。

在并行计算的基本思路上，Triton 与 CUDA 具有许多共通之处。两者都遵循 GPU 编程中“单指令多数据（SIMD）”的执行模型，支持大量线程并行执行相同的程序逻辑，用于加速密集型计算任务。同时，它们都以线程块（CUDA 中为 thread block，Triton 中为 program）为调度单元，将全局计算任务划分为多个子任务，通过显式控制线程内的数据访问和计算流程，从而实现高效的并行计算。在性能优化方面，Triton 和 CUDA 都强调内存访问模式、线程间协作与寄存器利用率等关键因素，力求最大化 GPU 的吞吐率。正因如此，Triton 用户在迁移或借鉴 CUDA 编程经验时能迅速上手。

在差异化方面，Triton 的核心理念是基于分块的编程范式，这一范式有助于构建神经网络的高性能计算核心。与 CUDA 传统的 “单程序，多数据”GPU 执行模型在线程细粒度上的并行编程不同，Triton 是在分块的粗粒度上进行并行编程，而每一个块内部进行串行编程（这一部分的串行后续会自动被 Triton 给 offload 到 GPU 并转为并行执行）。这种方式产生了块结构的迭代空间，为程序员在实现稀疏操作时提供了更多的灵活性，同时也便于编译器为数据局部性和并行性进行积极的优化（也就是每一个块内部的串行编程的部分）。例如，在处理大规模矩阵运算时，Triton 可以将矩阵划分为多个小块，每个小块由一个独立的程序（线程块）进行处理，通过合理的分块和调度，充分利用 GPU 的并行计算资源，提高计算效率。

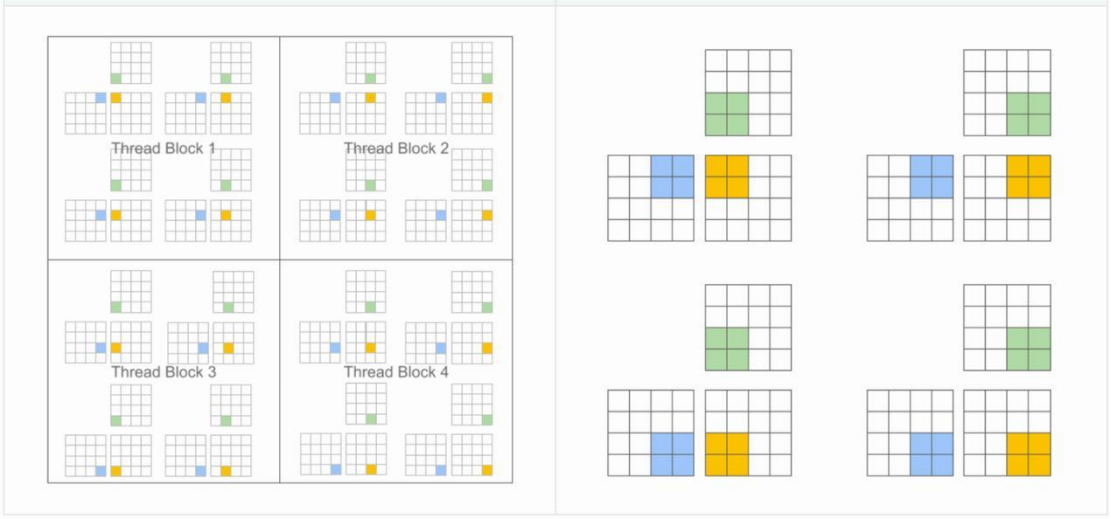
CUDA Programming Model
(Scalar Program, Blocked Threads)

```
#pragma parallel
for(int m = 0; m < M; m++)
#pragma parallel
for(int n = 0; n < N; n++){
  float acc = 0;
  for(int k = 0; k < K; k++)
    acc += A[m, k] * B[k, n];

  C[m, n] = acc;
}
```

Triton Programming Model
(Blocked Program, Scalar Threads)

```
#pragma parallel
for(int m = 0; m < M; m += MB)
#pragma parallel
for(int n = 0; n < N; n += NB){
  float acc[MB, NB] = 0;
  for(int k = 0; k < K; k += KB)
    acc +=  A[m:m+MB, k:k+KB]
          @ B[k:k+KB, n:n+NB];
  C[m:m+MB, n:n+NB] = acc;
}
```

图表 16 CUDA 和 Triton 的编程范式原语对比



图表 17 CUDA 与 Triton 的编程范式逻辑粒度对比

图表 16 直观地对比了 CUDA 与 Triton 编程模型在语法结构和逻辑组织上的差异。图中以伪代码形式分别展示了同一矩阵乘法任务在两种编程范式下的实现方式，并用 #pragma parallel 标注哪些代码部分是并行的，哪些是串行执行的（需要特别说明的是，这里的 #pragma parallel 仅用于示意，不代表 CUDA 或 Triton 原生支持该语法）。左侧是典型的 CUDA 编程模型，其核心特点是“标量程序、线程分块”（Scalar Program, Blocked Threads）：开发者编写的是一个标量级的指令流，即单个线程的逻辑，其并行性依赖于底层线程块（Thread Block）结构的调度。例如，最外层的两重 for 循环分别遍历矩阵的行和列，形成每个线程计算的单个输出元素 C[m, n]，而最内层的 for 循环负责完

成单个输出元素的点积累加。由于线程级别非常细粒度，CUDA 开发者通常需要显式地考虑线程的组织方式、共享内存使用、线程间同步等硬件层面的优化手段，这使得 CUDA 编程具有高度灵活性，但也增加了开发复杂度。右侧展示的 Triton 编程模型则采取了“分块程序、标量线程”（Blocked Program, Scalar Threads）的设计理念：开发者直接编写以子块（Tile）为单位的程序逻辑，而不是以单个线程为单位。在图中，可以看到最外层两个循环的步长是 MB 和 NB，表示以较大块状（例如 128×128）的形式在矩阵空间中移动；变量 acc 是一个大小为 [MB, NB] 的局部中间累加器，用于同时计算一个子块中的多个输出元素。在 for k 循环中，分别从矩阵 A 和 B 中提取对应的子块，完成子块级别的点积操作。最终，该子块的结果会被写入矩阵 C 的相应位置。相比 CUDA，Triton 的程序结构更加粗粒度，开发者不需要处理线程调度等底层细节，而是通过操作子块实现更高级别的并行性，Triton 编译器会自动将这些块划分为实际可并行执行的线程程序。

图表 17 进一步从逻辑粒度的角度揭示了 CUDA 和 Triton 编程范式的根本差异。左图展示的是 CUDA 编程模型中线程块的组织结构，其中每个小方格表示一个线程所处理的最小计算单元（例如矩阵的一个元素），深色区域表示开发者需要显式处理的串行逻辑范围。可以看到，在 CUDA 中，开发者需要在每个线程级别进行控制和优化，线程数庞大且分布密集，这种细粒度的串行控制使得程序的灵活性较强，但也要求开发者具备深入的硬件并行编程知识。在复杂的算法中，开发者通常需要针对每个线程块甚至每个线程编写专门的内存访问模式和同步机制，以最大程度利用硬件性能。相比之下，右图展示的是 Triton 编程模型中的逻辑粒度，深色区域表示一个程序处理的子块单位（例如一个 16×16 的 tile）。在 Triton 中，开发者所关注的最小串行部分是一个相对较大的计算块，而不是单个元素，因此 Triton 屏蔽了线程级别的复杂性，由编译器负责自动将这些块并行化调度执行。即使开发者编写的程序存在数据耦合（如块内元素间存在依赖关系），Triton 编译器也能够自动识别并在不影响正确性的前提下进行有效并行划分。这种模型大大降低了编程门槛，允许开发者更专注于算法本身，而非底层线程管理。

从另一个角度来说，Triton 的编程范式更像是 CPU 并行计算的编程范式。在 CPU 并行计算中，线程数量是没有 GPU 充裕的（CPU 通常有几十到几百个核，而 GPU 通常有几千到几万个核）。我们通常会把任务均匀分配到每一个 CPU 核上，并尽可能确保每一个核心的计算任务彼此是数据通信最低的。Triton 的编程范式也是在做同样的事情，它把 GPU 看成了一个核心数不那么多的并行处理器，不让开发者去考虑 GPU 众核的硬件细节，将分块内的串行计算任务到 GPU 上的并行计算任务的映射留给编译器处理。这种设计模式也使得将 CPU 并行的程序转化为 GPU 并行的程序这一过程更加自然。

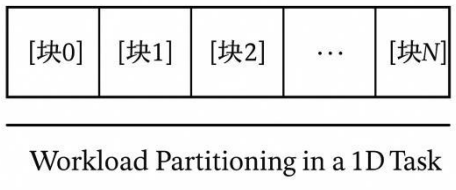
2.1.5 Triton 内核函数与线程划分

Triton 的核心概念包括：

- 网格(Grid): 类似于 CUDA 的网格，但更灵活
- 块(Program，也可以称作 Block，但注意与 CUDA 的 Block 做出区分): 执行的基本单位
- 程序 ID: 标识当前执行实例的位置
- 内存指针: 提供对数据的访问

一维任务划分

如图表 18 所示，在一维任务中，我们将工作负载划分为线性的一系列块。每个块处理数据的一个连续部分。



图表 18 一维任务分块示意图

```
import triton
```

```
import triton.language as tl


@triton.jit
def vector_add_kernel(
    x_ptr,  # 输入向量 x 的指针
    y_ptr,  # 输入向量 y 的指针
    output_ptr,  # 输出向量的指针
    n_elements,  # 向量的大小
    BLOCK_SIZE: tl.constexpr,  # 每个块处理的元素数量
):
    # 获取当前程序的 ID
    pid = tl.program_id(axis=0)

    # 创建范围掩码以防止越界访问
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements

    # 从全局内存加载数据
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)

    # 计算向量加法
    output = x + y

    # 将结果存储回全局内存
tl.store(output_ptr + offsets, output, mask=mask)
```

```
def vector_add(x, y, output, block_size=1024):
    n_elements = output.numel()
    # 根据总元素数和每个块处理的元素数，计算需要启动多少个线程块来覆盖所有元素
    # 定义一个计算 grid 大小的 lambda 函数
    # meta: 包含内核配置信息的字典
    grid = lambda meta: (
        # 计算需要的线程块数量:
        # 总元素数除以每个块处理的元素数，向上取整
        triton.cdiv(n_elements, meta['BLOCK_SIZE']),
        # 注意最后的逗号表示这是一个单元素元组(1D 网格)
    )
    # 启动内核
    vector_add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=block_size)
```

以上给出了一维任务的 Triton Kernel 和 Triton Kernel 的 Wrap 函数，使用以下方式调用上述函数：

```
import torch
import triton
import triton.language as tl


# 定义向量大小
n = 16
BLOCK_SIZE = 4


# 创建输入张量
x = torch.arange(n, dtype=torch.float32, device='cuda')
y = torch.arange(n, 0, -1, dtype=torch.float32, device='cuda')  # 16, 15, ..., 1
```

```
# 创建输出张量
output = torch.empty_like(x)


# 调用 Triton 内核函数执行加法
vector_add(x, y, output, block_size=BLOCK_SIZE)


# 输出结果
print("x:", x)
print("y:", y)
print("x + y:", output)
```

我们以 BLOCK_SIZE = 4、向量长度 n = 16 为例，输入数据为 x = [0, 1, 2, ..., 15]，y = [16, 15, 14, ..., 1]。Triton 会将总任务划分为若干个块，每个块负责 BLOCK_SIZE = 4 个元素的加法操作。在这个例子中，共需要 16 ÷ 4 = 4 个块，块的划分如下：

- 块 0 处理索引 [0, 1, 2, 3]，即 x[0:4] + y[0:4]
- 块 1 处理索引 [4, 5, 6, 7]
- 块 2 处理索引 [8, 9, 10, 11]
- 块 3 处理索引 [12, 13, 14, 15]

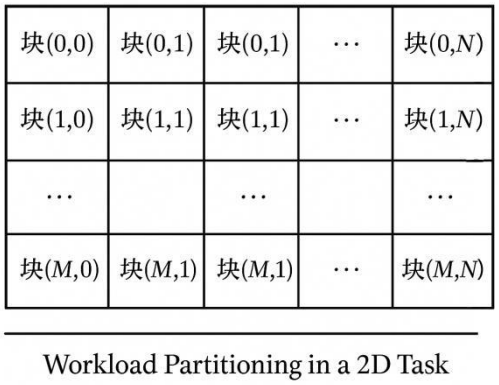
每个块在运行时由一个 Triton 的“程序实例”处理，它们通过 program_id(axis=0) 获取自己对应的块号 pid，然后通过 pid * BLOCK_SIZE + arange(0, BLOCK_SIZE) 计算需要处理的全局索引。通过掩码 mask 来避免越界访问（例如处理非对齐大小的情况）。每个块从全局内存中加载 x 和 y 的对应部分，逐元素相加，并将结果写入到 output 中相应位置。

最终，整个向量加法操作被并行化执行在 GPU 上，效率远高于串行处理。

二维任务划分

如图表 19 所示，在二维任务中，我们将工作负载划分为一个二维网格。这

在处理矩阵或图像时特别有用。



图表 19 二维任务分块示意图

```
@triton.jit
def matrix_add_kernel(
    x_ptr,  # 输入矩阵 x 的指针
    y_ptr,  # 输入矩阵 y 的指针
    output_ptr,  # 输出矩阵的指针
    n_rows,  # 矩阵行数
    n_cols,  # 矩阵列数
    BLOCK_SIZE_ROW: tl.constexpr,  # 每块处理的行数
    BLOCK_SIZE_COL: tl.constexpr,  # 每块处理的列数
    stride_x,  # x 的步长
    stride_y,  # y 的步长
    stride_out,  # 输出的步长
):
    # 获取当前程序在行和列维度的 ID
    row_pid = tl.program_id(axis=0)
    col_pid = tl.program_id(axis=1)

    # 计算行和列的偏移量
```

```
    row_offsets = row_pid * BLOCK_SIZE_ROW + tl.arange(0, BLOCK_SIZE_ROW)
    col_offsets = col_pid * BLOCK_SIZE_COL + tl.arange(0, BLOCK_SIZE_COL)

    # 创建二维掩码
    row_mask = row_offsets[:, None] < n_rows
    col_mask = col_offsets[None, :] < n_cols
    mask = row_mask & col_mask

    # 计算内存偏移量
    x_ptrs = x_ptr + row_offsets[:, None] * stride_x + col_offsets[None, :]
    y_ptrs = y_ptr + row_offsets[:, None] * stride_y + col_offsets[None, :]
    out_ptrs = output_ptr + row_offsets[:, None] * stride_out + col_offsets[None, :]

    # 加载数据
    x = tl.load(x_ptrs, mask=mask)
    y = tl.load(y_ptrs, mask=mask)

    # 计算矩阵加法
    output = x + y

    # 存储结果
      tl.store(out_ptrs, output, mask=mask)

def matrix_add(x, y, output, block_size_row=32, block_size_col=32):
    n_rows, n_cols = output.shape
    # 计算需要的块数
```

```
    grid = lambda meta: (
        triton.cdiv(n_rows, meta['BLOCK_SIZE_ROW']),
        triton.cdiv(n_cols, meta['BLOCK_SIZE_COL'])
    )
    # 启动内核
    matrix_add_kernel[grid](
        x, y, output, n_rows, n_cols,
        BLOCK_SIZE_ROW=block_size_row,
        BLOCK_SIZE_COL=block_size_col,
        stride_x=x.stride(0),
        stride_y=y.stride(0),
        stride_out=output.stride(0)
    )
```

调用方式为：

```
import torch



# 创建输入矩阵
x = torch.tensor([
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [8, 9, 10, 11],
    [12, 13, 14, 15]
], dtype=torch.float32, device='cuda')


y = torch.tensor([
    [15, 14, 13, 12],
```

```
    [11, 10, 9, 8],
    [7, 6, 5, 4],
    [3, 2, 1, 0]
], dtype=torch.float32, device='cuda')

# 创建输出矩阵（与 x 和 y 同形状）
output = torch.empty_like(x)

# 调用 Triton 内核函数执行加法
matrix_add(x, y, output, block_size_row=2, block_size_col=2)

# 输出结果
print("x:\n", x)
print("y:\n", y)
print("x + y:\n", output)
```

在这个例子中，我们处理的是两个 4×4 的矩阵，使用 Triton 进行矩阵加法运算。我们设定 BLOCK_SIZE_ROW = 2，BLOCK_SIZE_COL = 2，即每个 Triton 程序块（或线程块）处理一个 2×2 的子矩阵。整个矩阵被划分成 4 个计算块，每个块负责处理一部分数据，如下所示：

块(0,0)：处理 x[0:2, 0:2] 与 y[0:2, 0:2]

块(0,1)：处理 x[0:2, 2:4] 与 y[0:2, 2:4]

块(1,0)：处理 x[2:4, 0:2] 与 y[2:4, 0:2]

块(1,1)：处理 x[2:4, 2:4] 与 y[2:4, 2:4]

Triton 使用 program_id(axis=0) 和 program_id(axis=1) 分别获取当前线程块的行 ID 和列 ID。然后用 tl.arange 和这些 ID 计算出当前块需要访问的全局行列索引（即 row_offsets 和 col_offsets）。通过计算偏移地址和掩码 mask，每个块从全局内存中提取数据、执行加法，并将结果写回输出矩阵。

三维任务划分

对于更高维度的数据，如体积数据或批量处理，可以使用三维划分。

```
@triton.jit
def batched_matrix_multiply_kernel(
    a_ptr, b_ptr, c_ptr,              # 输入矩阵 A, B 和输出矩阵 C 的指针
    M, N, K,                          # A: [M, K]，B: [K, N]，C: [M, N]
    stride_am, stride_ak, stride_ab,  # A 的三维存储步长：行、列、批
    stride_bk, stride_bn, stride_bb,  # B 的三维存储步长：行、列、批
    stride_cm, stride_cn, stride_cb,  # C 的三维存储步长：行、列、批
    BLOCK_SIZE_M: tl.constexpr,       # 每个 Triton 程序块负责处理 M 维的大小
    BLOCK_SIZE_N: tl.constexpr,       # 每个 Triton 程序块负责处理 N 维的大小
    BLOCK_SIZE_K: tl.constexpr,       # 每次从 K 维加载的数据块大小
    GROUP_SIZE_M: tl.constexpr        # 程序块在 M 方向上的分组大小（用于线程调度优化）
):
    # 当前 Triton 程序块的三维 ID：batch 维、M 维、N 维
    batch_pid = tl.program_id(axis=0)
    pid_m = tl.program_id(axis=1)
    pid_n = tl.program_id(axis=2)

    # 当前程序块处理的行（rm）和列（rn）索引范围
    rm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    rn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)

    # 初始化累加器 acc，用于存储结果子矩阵，初始值为 0
    acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
```

```
    # K 维按照 BLOCK_SIZE_K 划分成多个子块，逐块处理
    for k in range(0, K, BLOCK_SIZE_K):
        k_offsets = k + tl.arange(0, BLOCK_SIZE_K)

        # 计算当前 A 子块的地址：batch + 行索引 + K 轴偏移
        A_k = a_ptr + batch_pid * stride_ab + rm[:, None] * stride_am + k_offsets[None, :] * stride_ak
        # 计算当前 B 子块的地址：batch + K 轴偏移 + 列索引
        B_k = b_ptr + batch_pid * stride_bb + k_offsets[:, None] * stride_bk + rn[None, :] * stride_bn

        # 根据实际范围创建掩码，防止越界访问
        A_mask_k = (rm[:, None] < M) & (k_offsets[None, :] < K)
        B_mask_k = (k_offsets[:, None] < K) & (rn[None, :] < N)

        # 从全局内存中加载 A 和 B 子块
        a = tl.load(A_k, mask=A_mask_k, other=0.0)
        b = tl.load(B_k, mask=B_mask_k, other=0.0)

        # 累加当前子块的矩阵乘法结果
        acc += tl.dot(a, b)

    # 计算输出矩阵 C 的存储地址
    C = c_ptr + batch_pid * stride_cb + rm[:, None] * stride_cm + rn[None, :] * stride_cn

    # 创建写入掩码，防止越界写入
    C_mask = (rm[:, None] < M) & (rn[None, :] < N)
```

```
    # 将计算结果写回输出矩阵
     tl.store(C, acc, mask=C_mask)


def batched_matmul(a, b, output, block_m=64, block_n=64, block_k=32, group_m=8):
    """
    启动 batched 矩阵乘法内核:
    计算多个批次的 A @ B 并将结果写入 output。

    参数:
        a: Tensor of shape [B, M, K]
        b: Tensor of shape [B, K, N]
        output: Tensor of shape [B, M, N]
        block_m, block_n, block_k: 每个程序块处理的 M、N、K 子块大小
        group_m: Triton 程序在 M 维方向的分组数，用于负载均衡优化
    """
    B, M, K = a.shape
    _, _, N = b.shape

    # 定义 Triton 网格大小：每个维度启动的程序块数
    grid = lambda META: (
        B,  # batch 维
        triton.cdiv(M, META['BLOCK_SIZE_M']),  # M 维被划分为多少个程序块
        triton.cdiv(N, META['BLOCK_SIZE_N'])   # N 维被划分为多少个程序块
    )

    # 启动 Triton 内核并传递所有必要参数
```

```
    batched_matrix_multiply_kernel[grid](
        a, b, output,
        M, N, K,
        a.stride(1), a.stride(2), a.stride(0),
        b.stride(1), b.stride(2), b.stride(0),
        output.stride(1), output.stride(2), output.stride(0),
        BLOCK_SIZE_M=block_m,
        BLOCK_SIZE_N=block_n,
        BLOCK_SIZE_K=block_k,
        GROUP_SIZE_M=group_m
    )
```

这段 Triton 程序实现的是一个批量矩阵乘法（Batched Matrix Multiplication）的三维并行任务。在深度学习等计算中，往往会一次性处理多个矩阵对的乘法操作，例如对形如 (B, M, K) 的张量 a 和形如 (B, K, N) 的张量 b，执行批量乘法以生成 (B, M, N) 的输出张量 c。整个计算任务被映射为一个三维网格，其维度分别对应 批次 (B)、输出矩阵的行块划分 (M) 和 列块划分 (N)，从而实现跨多个批次、多个矩阵子块的并行计算。

使用如下的方式调用上述程序:

```
import torch


# 定义一个 2 批次的矩阵乘法任务，每个矩阵大小为 (4, 4)
B, M, K, N = 2, 4, 4, 4


# 创建输入张量：形状为 (B, M, K) 和 (B, K, N)
a = torch.arange(B * M * K, dtype=torch.float32, device='cuda').reshape(B, M, K)
b = torch.arange(B * K * N, dtype=torch.float32, device='cuda').reshape(B, K, N)
```

```
# 创建输出张量
output = torch.empty((B, M, N), dtype=torch.float32, device='cuda')

# 调用 Triton 内核
batched_matmul(a, b, output, block_m=2, block_n=2, block_k=2, group_m=1)

# 输出结果
print("a:\n", a)
print("b:\n", b)
print("a @ b:\n", output)
```

在这个例子中，我们进行的是两个批次（B = 2）的矩阵乘法，每批乘法都涉及两个 4×4 的矩阵相乘。整个任务被分配给一个三维 Triton 网格，每个程序实例 program_id(axis=0, 1, 2) 分别对应于某个批次编号 batch_pid、输出矩阵的行块编号 pid_m 和列块编号 pid_n。例如，对于第 0 批（batch_pid = 0），输出矩阵的第一个 2×2 子块由程序块 (0, 0, 0) 负责计算，该程序实例会加载 a[0, 0:2, 0:2] 和 b[0, 0:2, 0:2] 执行乘法并将结果写入 c[0, 0:2, 0:2]。每次通过 BLOCK_SIZE_K 的步进进行累加，最终完成该子块的所有乘法。多个程序实例协同处理整个批次的矩阵乘法，从而实现批量并行计算。

与 CUDA 编程模型的区别

在 Triton 中，grid 是启动 kernel 时用于定义并行度的参数，决定了 kernel 的并发执行单元（称为 “program”）的组织方式。Triton 的每一个 kernel 实际上会被并发地调用很多次，每次调用就是一个独立的 program。开发者通过 grid=(n,) 或 grid=(n, m) 来指定启动多少个 program，从而实现数据的分片处理。这种方式与 CUDA 中的 thread block 类似，但 Triton 的 program 粒度更粗，组织方式也更灵活。

与 CUDA 编程模型不同，Triton 更加强调 program 之间的逻辑独立性，我们通常希望每个 program 处理一块互不依赖的数据，从而最大化并行性。而在

program 内部，Triton 编译器能够自动进行高效的数据访问优化、矢量化和内存调度。这意味着即使存在一定程度的数据相关性，Triton 也能自动处理这些细节，减轻开发者的负担。

相比之下，CUDA 编程通常要求开发者显式地管理 block 内 thread 之间的协作，如通过共享内存、内存访问对齐等技术来避免性能瓶颈。这种手动优化方式不仅复杂，而且容易出错，性能表现也高度依赖于实现细节。总的来说，Triton 的 program 不能简单地类比为 CUDA 的 block，它更像是一个逻辑上完整的数据处理单元。最适合 Triton 的编程方式，是将结构清晰、数据划分明确的任务交由每个 program 独立处理，充分发挥其在自动优化方面的优势。

2.1.6 triton 自动优化机制

Triton 的自动优化机制通过高层次抽象为不同硬件提供兼容性支持，它封装了底层细节并内置了多种优化策略。这些自动化的优化涵盖了大部分 CUDA 实用的优化技术，并保证了跨硬件平台的通用性。当然，这也在一定程度上限制了用户的深度优化空间——开发者无需（也难以）显式地指定底层优化，大部分优化决策都由框架自动完成，仅保留有限的调优接口供用户干预。

Triton 内存优化

虽然 Triton 和 CUDA 都面向 GPU 内核开发，但在内存访问控制方面，两者的抽象层次和开发者参与度差异显著。Triton 倾向于自动化与声明式，而 CUDA 更偏向显式控制与底层优化。

全局内存访存优化

表格 2 Triton 与 CUDA 的全局内存访问控制对比

对比维度	**Triton**	**CUDA**
编写方式	声明式：tl.load(ptr + offsets)	显式指针：data[i] = A[threadIdx.x + ...];
合并访存（coalescing）	编译器自动分析访问模式并生成合并指令	需手动保证 thread 按顺序访问连续地址

向量化	自动向量化（如生成 ld.global.v4.f32）	需使用 float4* 等类型明确 vector load
错误保护机制	使用 mask 避免越界（自动屏蔽）	需手动判断边界并 return，易出错

表格 2 给出了 Triton 与 CUDA 的全局内存访问控制对比。我们重点关注合并访存，它指将多个线程对全局内存的分散访问合并为更少的内存事务，以提升带宽利用率。Triton 编译器可自动分析张量操作的访问模式，在满足合并条件时生成合并的访存指令。而 CUDA 更加依赖开发者经验和手工调优。但 Triton 的自动访存合并仍然依赖于程序员编写规则化、结构化的访存逻辑。若算法的空间局部性很差，会显著降低 Triton 自动访存合并的效率，这个时候往往编程人员需要从算法设计的角度重新思考来解决性能问题。

Triton 通过以下步骤实现合并的全局内存访问：

1. 访问模式分析：分析 `tl.load()` / `tl.store()` 所访问的内存地址是否为线程连续递增。
2. 自动数据布局重排（编译期）：在张量视图和算子之间自动重排数据布局以提高访存合并度。
3. 高效 PTX 生成：针对 NVIDIA 平台生成 `ld.global.v4.f32` 等向量化指令。

其中，自动数据布局重排是一项关键优化，旨在提升全局内存访问的合并度和带宽效率。它主要支持维度交换、张量重塑和切片重组等重排策略。当 tl.load() / tl.store() 的访问模式可静态分析、线程访存呈现连续递增结构且不涉及对维度顺序敏感的计算语义时，Triton 编译器会自动调整张量视图的布局以匹配高效的访存模式。但对于数据依赖的动态访问、reduce 或 softmax 等依赖维度语义的操作，或涉及跨线程通信的场景，重排则会被禁止，以保证语义正确性与执行效率的兼顾。

Triton 自动数据布局重排支持如下几种方式：

- 维度交换（Dimension permutation）

将张量维度的物理顺序进行变换，例如 [B, M, N] → [M, N, B]，使访问更适合合并。

举例：将原本跨线程不连续的 innermost dimension 交换到最内层，使线程访问成为连续地址。

- 张量视图重塑（View reshape）

将张量重构为等价形状以配合访存模式。例如 [64, 16] reshape 为 [16, 64]。

- 张量切片重组（Tiling-based rematerialization）

基于 block size / tile size 的划分重建张量，按 tile 区间重排 memory layout。

- Transpose 优化识别与融合

将显式或隐式的转置操作（如 x.T）识别并融合到前后操作的数据访问中，以避免额外代价。

我们考虑一个场景：对一个二维矩阵的每一行进行 L2 归一化。内核如下：

```
import triton
import triton.language as tl
import torch


@triton.jit
def row_norm_kernel(x_ptr, y_ptr, M, N, stride_xm, stride_xn, stride_ym, stride_yn, BLOCK_SIZE: tl.constexpr):
    row_id = tl.program_id(axis=0)
    offsets = tl.arange(0, BLOCK_SIZE)
    row_offsets = row_id * stride_xm + offsets * stride_xn


    # 判断是否越界
```

```
    mask = offsets < N

    # Load row
    row = tl.load(x_ptr + row_offsets, mask=mask, other=0.0)

    # 计算范数
    norm = tl.sqrt(tl.sum(row * row, axis=0))

    # 归一化
    row = row / norm

    # Store
    tl.store(y_ptr + row_id * stride_ym + offsets * stride_yn, row, mask=mask)
```

Host 端调用的代码为：

```
import torch
import triton
import triton.language as tl

# 生成一个随机矩阵 [M, N]
M, N = 1024, 512
x = torch.randn((M, N), device='cuda')
y = torch.empty_like(x)

# 启动 Triton kernel
BLOCK_SIZE = 128
grid = lambda META: (M,)

# 调用 kernel
```

```
row_norm_kernel[grid](
    x, y,
    M, N,
    x.stride(0), x.stride(1),
    y.stride(0), y.stride(1),
    BLOCK_SIZE=BLOCK_SIZE
)

# 验证
torch.testing.assert_close(y.norm(dim=1), torch.ones(M, device='cuda'), rtol=1e-2, atol=1e-2)
```

在上述示例中，x 是一个 [M, N] 的二维矩阵，用户以 row-wise 方式访问 x[i, :]，也就是内核中 offsets 是沿着 最内层维度 N（即列） 增加的。

在某些情况下，如果张量在内存中实际存储的维度顺序并非 [M, N]（例如来自于某个 transpose() 操作后，其内存顺序为 [N, M]），那么直接以行优先的方式访问会导致 非连续内存访问，从而难以合并。

Triton 编译器能够自动分析内核中 tl.load() 的访问模式，识别出沿着某一维度连续递增的访问特征，并据此智能地重排数据布局（例如通过重建视图或调整内存映射），以确保最内层维度保持连续递增的访问顺序，最终自动选择生成高效的向量化指令（如 ld.global.v4.f32 等），从而优化内存访问性能。这背后的关键优化是：即使用户传入的张量已经经历了转置或 reshape 操作，只要计算语义允许，Triton 编译器就会通过 layout rematerialization 或视图调整，确保访存模式尽可能合并，从而提升带宽利用率。

Triton 的自动内存优化提供了很好的跨硬件架构兼容性，但若有特殊需求，开发者也可通过 block pointer 等接口显式地控制访存行为，适用于高级性能调优。这里有一个例子：

```
import triton
import triton.language as tl
import torch

@triton.jit
def load_with_block_ptr_kernel(a_ptr, out_ptr, M, K, stride_am, stride_ak,
                               BLOCK_SIZE_M: tl.constexpr,
                               BLOCK_SIZE_K: tl.constexpr):
    pid = tl.program_id(0)
    # 计算每个 program 的起始行索引
    row_start = pid * BLOCK_SIZE_M

    # 构造 block pointer
    a_block_ptr = tl.make_block_ptr(
        base=a_ptr,
        shape=(M, K),
        strides=(stride_am, stride_ak),
        offsets=(row_start, 0),
        block_shape=(BLOCK_SIZE_M, BLOCK_SIZE_K),
        order=(1, 0)  # K 在内层，M 在外层（行主序）
    )

    # 加载该块数据
    a_block = tl.load(a_block_ptr)  # BLOCK_SIZE_M × BLOCK_SIZE_K 张量

    # 写入输出张量
    out_ptrs = out_ptr + row_start * K + tl.arange(0, BLOCK_SIZE_K)[None, :] +
tl.arange(0, BLOCK_SIZE_M)[:, None] * K
```

```
    tl.store(out_ptrs, a_block)


# 模拟输入数据
M, K = 64, 64
BLOCK_SIZE_M = 16
BLOCK_SIZE_K = 16
a = torch.arange(M * K, dtype=torch.float32, device='cuda').reshape(M, K)
out = torch.empty_like(a)


# 启动 kernel
grid = lambda meta: (M // BLOCK_SIZE_M,)
load_with_block_ptr_kernel[grid](
    a_ptr=a, out_ptr=out,
    M=M, K=K,
    stride_am=a.stride(0), stride_ak=a.stride(1),
    BLOCK_SIZE_M=BLOCK_SIZE_M,
    BLOCK_SIZE_K=BLOCK_SIZE_K
)


# 验证结果
assert torch.allclose(out, a)
```

在该示例中，我们展示了如何使用 Triton 的 tl.make_block_ptr 显式构造块指针，从而精确控制张量子块的加载方式。首先通过 tl.make_block_ptr 创建指向输入张量 a 中某个子块的 block pointer，其中 shape 和 strides 描述全局张量布局，offsets 指定当前程序处理的块起始位置，block_shape 和 order 控制块大小和存储顺序。接着通过 tl.load 载入该块数据，并使用 tl.store 将其写入输出张量中。该方法绕过了 Triton 默认的访存模式，使得高级开发者可以更精细地调度访存顺序和内存对齐策略，在异构硬件架构（如 A100, H100 或 MI300）中

进行手工调优时非常有用。

共享内存访存优化

表格 3 Triton 与 CUDA 的共享内存访问控制对比

对比维度	Triton	CUDA
使用方式	编译器自动调度共享内存（隐式）	开发者显式声明 __shared__ 缓冲区
缓存范围控制	依赖于 tl.dot / make_block_ptr 等操作自动缓存	开发者需手动管理分块加载、索引和写入
同步机制	自动插入 barrier	需手动添加 __syncthreads()
使用限制	不支持手动控制共享内存大小或布局	灵活控制共享内存分配方式与线程访问策略

表格 3 给出了 Triton 与 CUDA 的共享内存访问控制对比。共享内存是 GPU 上的片上高速缓存，适合线程块内重复访问的数据。Triton 当前不提供用户声明共享内存的接口，而是通过对 `load` / `dot` 等指令模式分析，隐式安排数据缓存和同步。Triton 对常见模式（如矩阵乘法）提供共享内存自动优化以最大化软硬件兼容性，降低使用门槛，但灵活性不如 CUDA，难以实现手动调度和特殊布局。

Triton 通过以下步骤实现自动的共享内存的使用：

1. 自动数据预取与缓存：识别跨线程重复访问的数据块并缓存在共享内存
2. 同步自动插入：在 block 内部的 thread 组之间自动插入 barrier 操作
3. 临时变量融合：中间结果尽可能驻留在寄存器或共享内存中

Triton 通过静态地址分析、访问等价类识别与 tile-based reuse 分析，自动判断多个线程是否访问重复的数据块。一旦检测到跨线程存在对同一内存区域

的重叠访问，编译器会将这部分数据提前预取并缓存至共享内存，并在必要位置自动插入 barrier 保证同步。此外，Triton 编译器还会对 index 表达式进行模式匹配，从而在 tile/block 级别发现共享数据使用模式。这一机制显著减少了 DRAM 带宽压力，提高了访存效率。

Triton 通过以下步骤识别“跨线程重复访问的数据块”:

- 静态地址分析（Static address analysis）

Triton 的编译器会对 tl.load() / tl.store() 等内存访问操作进行静态分析，具体包括:

1. 提取访问地址表达式：将访问模式表示为线程索引的函数，如:

```
addr=base+stride×pid_m+offset
```

其中 pid_m 是 block 内线程的并行索引。

2. 分析访问重叠区域：当多个线程对相同的地址范围进行访问时，编译器判定存在重复访存:

例如两个线程访问:

线程 A: x[i, j]

线程 B: x[i+1, j]

如果 j 范围共享，即不同线程访问的数据在 j 这个维度上存在重叠，即多个线程访问的是同一列的数据。则可判断对 x[:, j] 的访问有交叉区域。

3. 判断重复访问是否可缓存（即是否具备 Temporal Locality）

当发现某个数据块被多个线程或不同 block 中的线程多次使用时，编译器认为其具有共享缓存价值。

- 内存访问等价类识别（Equivalence Class Analysis）

编译器会将访问地址划分为“等价类”（equivalence classes）:

如果多个线程访问某一共享内存区域的不同部分，但具有“重叠”或“邻接”特性（如相邻 tile），则合并为一个共享缓存访问单元。

- Tile-based Data Reuse 分析

Triton 的核心优化策略之一是 tile-based 编程模型。

如果在多个 tile 中访问相同的 global memory 数据块，编译器将其识别为可缓存模式，并将该数据 block 映射至 shared memory。

例如：

```
x = tl.load(input_ptr + tile_offsets + offset)
```

如果 tile_offsets 是不同线程的偏移，而 offset 是共享的，那么 offset 指向的数据被多线程重复访问，Triton 会将其放入 shared memory。在 tile-based 编程模型中，每个 tile 对应一组并行执行的线程（或称 program），这些线程通常会对数据块进行局部访问。如果 tile_offsets 是每个线程不同的偏移量，说明线程之间访问的起始位置不同；而 offset 是所有线程共享的常数，意味着每个线程在其自身偏移位置基础上，都额外访问相同的一段数据。因此，虽然每个线程访问的位置不同（由 tile_offsets 决定），但它们都会共同访问 offset 所指定的那部分数据。换句话说，这段数据在多个线程中被重复使用。Triton 编译器识别到这一重复访问模式后，会自动将这段共享数据提前加载到共享内存（shared memory）中，从而避免多次从全局内存读取，提升访存效率。

- 迭代器/Indexing 模式的 pattern matching（Pattern matching of indexing structure）

Triton 会对 tl.load() 的 indexing 模式做模式匹配（例如是否是 broadcast、tile、group 模式）来判定是否有跨线程重用：

例如：

```
# 假设 block 中每个线程处理一行
col_idx = tl.arange(0, BLOCK_COLS)
row_idx = tl.program_id(0)
x = tl.load(A + row_idx * stride + col_idx)
```

如果多个 row_idx 的线程访问的是相邻或重叠的 col_idx，则可识别为可重用。

然而，Triton 并不会在所有场景下启用共享内存缓存。当内存访问模式依

赖运行时动态索引、线程之间的访问区域无交集、共享内存空间不足或可能引发 bank conflict 时，编译器会主动放弃缓存策略。此外，对于生命周期极短的中间变量，若其无法带来显著的访存收益，Triton 也会选择保留在寄存器中而不占用共享内存。这样的策略在保持高性能的同时，也避免了资源浪费和潜在的同步开销。

为了演示 Triton 编译器的智能共享内存优化能力，下面我们实现一个极为朴素的矩阵乘法 kernel，它没有显式声明任何共享内存，也没有手动做任何数据重用：

```
@triton.jit
def matmul_kernel(A, B, C, M, N, K, BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    offs_k = tl.arange(0, BLOCK_K)

    a_ptrs = A + offs_m[:, None] * K + offs_k[None, :]
    b_ptrs = B + offs_k[:, None] * N + offs_n[None, :]

    acc = tl.zeros([BLOCK_M, BLOCK_N], dtype=tl.float32)
    for k in range(0, K, BLOCK_K):
        a = tl.load(a_ptrs)
        b = tl.load(b_ptrs)
        acc += tl.dot(a, b)
        a_ptrs += BLOCK_K
```

```
        b_ptrs += BLOCK_K * N

    c_ptrs = C + offs_m[:, None] * N + offs_n[None, :]
    tl.store(c_ptrs, acc)
```

它的调用方法为：

```
import torch
import triton
import triton.language as tl

# 假设我们要计算 A [M, K] @ B [K, N] = C [M, N]
M, N, K = 128, 256, 64

# 初始化输入矩阵，使用 float16/float32 都可以
A = torch.randn((M, K), device='cuda', dtype=torch.float16)
B = torch.randn((K, N), device='cuda', dtype=torch.float16)
C = torch.empty((M, N), device='cuda', dtype=torch.float32)  # 输出用 float32，匹配 acc 精度

# 定义 block size
BLOCK_M = 32
BLOCK_N = 32
BLOCK_K = 32

# 计算 grid size：每个 block 处理 BLOCK_M × BLOCK_N 的输出子块
grid = (
    (M + BLOCK_M - 1) // BLOCK_M,
    (N + BLOCK_N - 1) // BLOCK_N,
)
```

```
# 启动 Triton kernel
matmul_kernel[grid](
    A, B, C,
    M, N, K,
    BLOCK_M=BLOCK_M,
    BLOCK_N=BLOCK_N,
    BLOCK_K=BLOCK_K,
)
```

尽管这个 kernel 没有任何显式的共享内存使用，Triton 编译器依然可以自动识别并插入共享内存缓存机制。Triton 编译器通过跨线程重用分析（Cross-thread Reuse Analysis）机制，能够在无需程序员干预的情况下自动实现共享内存加速。具体而言，编译器会静态分析 a_ptrs 和 b_ptrs 的访问模式，发现它们在 block 内多个线程之间存在高度重合，尤其是同一个 BLOCK_K 范围内的数据会被 BLOCK_M × BLOCK_N 个线程重复使用。基于这一分析，Triton 自动在每次循环内将这些数据预取至共享内存，并在 tl.dot(a, b) 运算前插入必要的 barrier 以确保线程间同步。生成的 PTX 代码中，访存指令从原本的 ld.global 优化为 ld.shared，实现了显著的带宽节省和并发效率提升。

这个例子说明，哪怕开发者写的是“最傻的”全局内存访问模式，Triton 编译器依然能够通过跨线程重用分析和 tile-based 缓存插入机制，自动将部分数据缓存至共享内存，并插入必要的同步操作。开发者无需关心共享内存细节，也能享受到其带来的性能收益。这种“语义简单 → 编译器智能优化”的设计，是 Triton 编程模型的重要优势。

其它内存优化技术

除了上述全局内存和共享内存的优化外， Triton 为了能生成高效的 GPU 内核也引入了其它自动优化技术，即使开发者缺乏底层 CUDA 编程经验，也能获得接近手写 kernel 的性能。

表格 4 Triton 与 CUDA 的其它优化技术对比

对比维度	Triton	CUDA
寄存器分配	自动寄存器复用与常驻	编译器自动分配，但可通过 __restrict__ 指令调优
延迟隐藏	编译器尝试自动调度访存与计算重叠	开发者可手动使用双缓冲/流水线隐藏访存延迟
数据复用表达	倾向高层张量表达（块操作）	可精细控制单个寄存器的复用与转发
Tensor Core 指令	编译器自动生成	显式 API 调用
Kernel fusion	分析不同 kernel 之间的数据依赖关系,将没有冲突的 kernel 进行融合	需手工进行

表格 4 总结了 Triton 与 CUDA 的其它优化技术对比。尽管 Triton 相比于 CUDA 有自动内存优化方面的优势，但 Triton 在极端性能需求场景下（如 warp-level 数据复用、流水线调度）可调性不如 CUDA，当前尚无法实现所有手工 CUDA 技巧。开发者仍然需要结合 CUDA 优化的相关知识，如解决 warp divergence 等问题，避免复杂条件控制或不规则访问，来最大化提升 Trtion 优化的效率。

Triton 编译优化

基础编译优化

Triton 编译器高度自动化，帮助开发者绕过底层的线程和 memory 管理细节，直接专注于算法逻辑。借助自动调优机制与编译期优化（循环展开、指令调度等），Triton 可以在可读性与执行效率之间实现良好平衡。我们先介绍两个核心编译优化概念：循环展开（Loop Unrolling） 与 指令调度（Instruction Scheduling）。这有助于读者更深入理解 Triton 编译器在底层如何生成高性能的

GPU 代码。

循环展开（Loop Unrolling）

循环展开是一种编译器优化技术，它将小规模、固定次数的循环直接展开为一系列连续的指令，从而减少循环控制逻辑（如分支判断、索引递增）带来的开销。

原始代码：

```
for (int i = 0; i < 4; ++i) {
    y[i] = x[i] * 2;
}
```

展开后：

```
y[0] = x[0] * 2;
y[1] = x[1] * 2;
y[2] = x[2] * 2;
y[3] = x[3] * 2;
```

循环展开具有诸多优势，例如消除循环判断（如 i < 4）；避免分支跳转，提高流水线效率；有助于后续进行向量化（SIMD）；以及在 GPU 上可减少每线程的控制分支，提高并发执行效率等。手工编程循环展开的代码时会面临代码可读性差、调试难度加大等诸多问题，幸运的是 Triton 可以自动化处理这一优化过程，当使用 tl.constexpr 声明循环边界时，Triton 会在编译期自动展开循环体，生成等效但无分支的内联指令：

```
for i in range(BLOCK_SIZE):  # BLOCK_SIZE 是 constexpr
```

指令调度（Instruction Scheduling）

指令调度是指编译器根据指令间的数据依赖关系，重新排序指令执行顺序，以最大限度利用处理器资源、减少数据等待和资源冲突。

原始顺序（存在数据依赖）：

```
a = load(mem[0]);  // 需要等待内存
b = a + 1;         // 等待 a
c = b * 2;
```

优化后顺序（交叉调度，隐藏内存延迟）：

```
a = load(mem[0]);   // 延迟加载
d = load(mem[100]); // 并行加载其他数据
e = d + 3;          // 并行计算
b = a + 1;          // 回到主依赖链
c = b * 2;
```

指令调度技术可以隐藏延迟，交错执行独立指令，减少空闲周期；可以避免资源冲突，合理安排指令，避免寄存器、ALU 冲突；还可以提升并行性，充分利用硬件流水线和多功能单元。手工实现指令调度需要进行低级的指令级优化，这对大多数开发者来说都是不小的挑战，然后 Triton 也自动实现了这一优化过程，Triton 在生成 PTX 前会自动进行调度，将访存延迟指令如 load 提前；尽可能推迟依赖计算以充分利用并发；并避免同一周期多个线程访问同一寄存

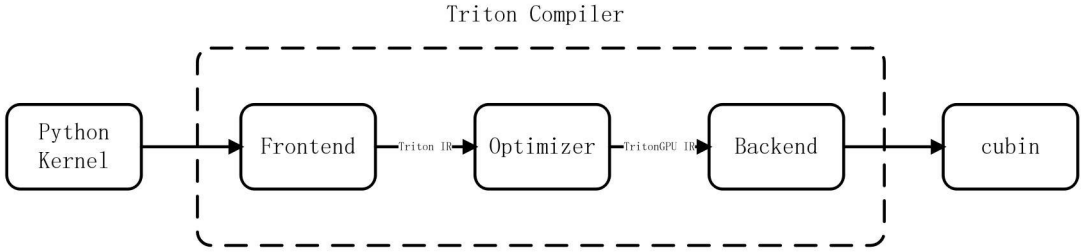
器或共享内存地址，提升效率。

编译流程概述

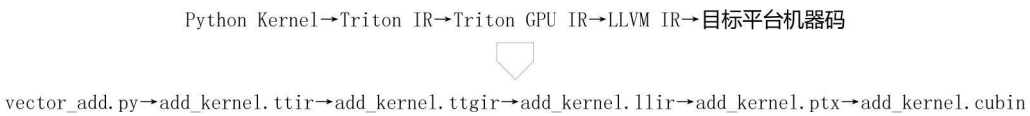
Triton 的编译流程大致如下：

- 前端解析：将 Python 中使用 @triton.jit 装饰的函数解析为 Triton IR（中间表示）。
- 中间优化：包括向量化、循环展开（Loop Unrolling）、内存访问模式分析（Memory Coalescing）等优化。
- 指令调度（Instruction Scheduling）：Triton 会自动分析数据依赖并调度指令顺序以减少寄存器压力与执行瓶颈。
- 后端代码生成：最终将 Triton IR 转换为 NVIDIA PTX 代码，通过 libcuda 动态加载并在 GPU 上执行。

图表 20 展示了整个流程：



图表 20 Triton 的编译流程



图表 21 Triton 编写的 GPU kernel 从 Python 源代码到目标硬件平台的机器代码的完整编译流程

图表 21 展示了一个 Triton 编写的 GPU kernel 从 Python 源代码到目标硬件平台的机器代码的完整编译流程。整个过程从 Python 编写的 kernel 函数（如图中的 01-vector-add.py）开始，首先被转换为 Triton IR（中间表示，.ttir），

它保留了 Triton 编程模型的高层语义。接着，Triton IR 被进一步转换为 Triton GPU IR（.ttgir），这一步引入了更多与目标 GPU 架构相关的调度与并行执行语义。随后，Triton GPU IR 会被转换为 LLVM IR（.llir），这是一个更加底层的中间语言，广泛用于构建各种平台上的编译器工具链。在这一阶段，Triton 会依赖 LLVM 基础设施将中间代码进一步编译为 PTX（.ptx），这是 NVIDIA GPU 的一种中间汇编语言，表示目标平台上的并行执行逻辑。最后，PTX 会被 NVIDIA 的 ptxas 工具进一步编译为目标机器代码（.cubin），即用于 GPU 硬件实际执行的二进制指令集合。整个流程体现了 Triton 编译器将高层 Python kernel 映射为低层 GPU 二进制代码的编译流水线，每一个阶段都在逐步细化和优化程序的表示，最终生成适合在目标硬件上高效执行的机器码。

我们在运行完 vectoradd 的 triton demo 之后，可以从以下目录获取所有生成的中间文件

```
/home/ubuntu/.triton/cache/MEAWZ14paZvH3RRVMIX4StGPuChuDqfJhBaAyJ3pQKY

add_kernel.cubin  add_kernel.json  add_kernel.llir  add_kernel.ptx  add_kernel.ttgir  add_kernel.ttir  __grp__add_kernel.json
```

其中/home/ubuntu/.triton/cache/中有三个目录，需要依次打开确认位置

Triton 手动优化

Triton 的自动优化已经涵盖了大部分优化内容，但 Triton 仍然提供了部分可选的手动优化 API，以供特需时使用。

流优化

Triton 的 CUDA stream 机制是依赖于 PyTorch 提供的 CUDA stream API 实现的，它本身不直接构造 CUstream。

```
import torch
import triton
import triton.language as tl
```

```
BLOCK_SIZE = 1024


@triton.jit
def scale_kernel(X, Y, scale, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    x = tl.load(X + offsets, mask=mask)
    y = x * scale
    tl.store(Y + offsets, y, mask=mask)


def run_kernel_in_stream(stream: torch.cuda.Stream, X, Y, scale: float):
    torch.cuda.set_stream(stream)


    n_elements = X.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)


    scale_kernel[grid](X, Y, scale, n_elements, BLOCK_SIZE=BLOCK_SIZE)


    stream.synchronize()


# 测试数据
N = 4096
X = torch.randn(N, device='cuda', dtype=torch.float32)
Y_default = torch.empty_like(X)
Y_streamed = torch.empty_like(X)


# 默认 Stream
```

```
run_kernel_in_stream(torch.cuda.default_stream(), X, Y_default, scale=2.0)


# 自定义 Stream
custom_stream = torch.cuda.Stream()
run_kernel_in_stream(custom_stream, X, Y_streamed, scale=2.0)


# 检查结果
assert torch.allclose(Y_default, Y_streamed), "结果不一致"
print("Results match! Streamed and default stream execution are consistent.")
```

在这个例子中，我们可以看到 Triton 的 kernel 本身只专注于执行特定的数据操作（如缩放向量），而并不涉及调度策略、流控制或跨 kernel 协调等更高层次的优化。这些任务仍然由 PyTorch 接管，Triton 只是专注于生成高效的 kernel 执行代码。而通过 torch.cuda.Stream 传入自定义 CUDA stream 并调用 Triton kernel 的方式可以正常运行并产生与默认 stream 一致的结果，这说明 Triton 能很好地与 PyTorch 的流机制兼容，开发者可以在不修改 Triton kernel 的情况下，将其灵活地集成进更复杂的异步执行框架中。这正体现了 Triton 是一个专注于 Kernel 内部优化的工具，同时又对 Kernel 外部的执行环境保持开放和兼容的设计理念。

参数自动调优（Autotune）

Triton 的 @triton.autotune 装饰器允许用户定义多个候选配置（如 block size、num warps 等），由编译器自动选择最优配置。

```
@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE': 64}, num_warps=2),
        triton.Config({'BLOCK_SIZE': 128}, num_warps=4),
        triton.Config({'BLOCK_SIZE': 256}, num_warps=8),
    ],
    key=['n']  # 按输入规模 n 自动选择最优配置
```

```
)
@triton.jit
def kernel_autotuned(x_ptr, y_ptr, n, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n
    x = tl.load(x_ptr + offsets, mask=mask)
    y = x * 2
    tl.store(y_ptr + offsets, y, mask=mask)
```

在上述代码中，configs 明确列出了多组候选配置（如不同的 BLOCK_SIZE 和 num_warps 组合），而 key=['n'] 指定了输入规模 n 作为调优的依据。Triton 会在第一次运行时，对每一组配置在当前输入条件下进行基准测试（benchmark），评估哪一组配置执行速度最快，并自动选定该配置用于当前和后续相同规模的输入。这一机制就是所谓的“自动调优”：开发者预先提供若干配置方案，实际运行时由 Triton 自动选择性能最优者。

2.1.7 triton 编程例子

由于 Triton 分块执行的特性，程序员编程的重点变了，以前需要瞻前顾后，既需要考虑 gird，block，thread 的划分，还需要解决“内部矛盾”，也就是每一个 kernel 内部的性能调优。有了 Triton 之后，程序员甩掉了半个包袱，kernel 升华为了 program 之后，程序员只需要关注如何将原本的计算任务划分为若干个数据不相关的 program，并且让 program 拥有尽可能合适的尺寸来为 Triton 对它的优化铺路即可。我们以 Triton 的两个经典案例来讲解如何将 Triton 编程应用到实战中。

矩阵乘法：分块策略与内存布局优化

这个例子主要在讲特定于 Triton 的分块的程序是什么样子的，以及分块后，如何在 program 外部尽可能优化以提高内存访问效率。

矩阵乘法是现代高性能计算系统的核心运算单元，同时也是极具挑战性的优化对象。正因如此，硬件厂商通常会提供专门的"内核库"（如 NVIDIA 的

cuBLAS）来实现高度优化的矩阵乘法。然而，这类专有库存在明显的局限性：一方面其源代码不可见且不可修改，另一方面难以针对现代深度学习任务的需求（如融合激活函数等特殊运算）进行灵活调整。而使用 Triton 语言自主实现矩阵乘法，则能兼顾计算效率和开发灵活性，满足深度学习场景下的定制化需求。

Triton 内核的基本实现思路是采用分块（block）算法来计算矩阵乘法。具体来说，给定一个大小为 (M, K) 的矩阵 A 和一个大小为 (K, N) 的矩阵 B，该算法会将计算分解为多个小块（block）操作，以提高计算效率和并行性。以下代码为待优化的 Python 实现：

```
# 外层循环：沿 M 维度分块并行计算
for m in range(0, M, BLOCK_SIZE_M):
  # 内层循环：沿 N 维度分块并行计算
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float32)
    # 累加循环：沿 K 维度分块计算矩阵乘积
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    # 将计算结果写回输出矩阵
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

在 Triton 的执行模型中，双重嵌套 for 循环的每个迭代都由一个独立的程序实例并行执行。虽然这个分块算法概念上很简单，但在实际实现时需要特别注意内存访问模式——关键在于如何高效计算内循环中矩阵块 A 和 B 的内存地址，这涉及到多维指针运算。

对于采用行主序存储的二维张量 X，元素 X[i,j]的内存地址可通过以下公式计算：

```
&X[i,j] = base_address + i * stride_xi + j * stride_xj
```

其中：

- base_address 是张量的起始地址
- stride_xi 表示在第 i 维（行方向）的步长
- stride_xj 表示在第 j 维（列方向）的步长

基于此，我们可以用以下伪代码表示矩阵块 A 和 B 的指针计算：

```
# 计算矩阵 A 的当前分块的内存地址：
# 将 [BLOCK_SIZE_M x BLOCK_SIZE_K] 的二维索引映射为线性地址，便于从 a_ptr 加载子块数据。

# 行偏移：生成一个列向量，表示 A 的第 m 行到第 m+BLOCK_SIZE_M 行的起始位置，
#       每行之间的地址间隔为 A.stride(0)，即“行步长”（对应第 0 维）。
# 列偏移：生成一个行向量，表示每列相对起始列 k 的偏移量，
#       每列之间的地址间隔为 A.stride(1)，即“列步长”（对应第 1 维）。
&A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K] =
  a_ptr + (m : m+BLOCK_SIZE_M)[:, None] * A.stride(0)  # 行方向地址计算（沿第 0 维）
        + (k : k+BLOCK_SIZE_K)[None, :] * A.stride(1)  # 列方向地址计算（沿第 1 维）

# 计算矩阵 B 的当前分块的内存地址：
# 同理，对 B 的 [BLOCK_SIZE_K x BLOCK_SIZE_N] 子矩阵计算线性地址。

# 行偏移：B 的“行”对应的是原始矩阵 B 的第 k 行到第 k+BLOCK_SIZE_K 行，
#       步长为 B.stride(0)（对应第 0 维，即 K 维度）。
# 列偏移：B 的“列”为第 n 列到第 n+BLOCK_SIZE_N 列，
#       步长为 B.stride(1)（对应第 1 维，即 N 维度）。
&B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N] =
```

```
    b_ptr + (k : k+BLOCK_SIZE_K)[:, None] * B.stride(0)  # 行方向地址计算（沿第 0 维）
          + (n : n+BLOCK_SIZE_N)[None, :] * B.stride(1)  # 列方向地址计算（沿第 1 维）
```

在 Triton 中初始化矩阵 A 和 B 的分块指针时，通常从 k = 0 开始计算，即先处理参与矩阵乘法的第一个“乘法维度”的子块。需要特别注意的是，当矩阵的行数 M 或列数 N 不是分块大小（BLOCK_SIZE_M 或 BLOCK_SIZE_N）的整数倍时，最后一个分块往往无法填满，这种边界情况如果处理不当，容易引发越界访问或计算错误。常见的应对策略包括：使用模运算或条件判断精确限制每个线程块的有效计算范围；对不满一整个分块的数据区域进行填充（padding），填充值需合理设置，以确保不影响最终的计算结果。此外，还需注意矩阵乘法中的“累加维度”——即 K 维度。在 A (M×K) 与 B (K×N) 的矩阵乘法中，K 表示两个矩阵共享的内部维度，对应的是 A 的列数和 B 的行数。在 K 维度上，分块过程也可能遇到边界不整齐的情况，此时通常采用掩码加载（masked load）机制：通过为超出有效范围的元素设置屏蔽掩码，既能防止非法访存，又能保持 GPU 并行执行的高吞吐率。以下代码给出了初始化矩阵 A 和 B 的分块指针的实现：

```
# 获取当前程序实例在 M×N 网格中的位置：
# pid_m 表示当前程序实例在“块行”方向（对应矩阵 A 的 M 维）上的编号；
# pid_n 表示当前程序实例在“块列”方向（对应矩阵 B 的 N 维）上的编号；
pid_m = tl.program_id(0)
pid_n = tl.program_id(1)


# 计算 A 矩阵当前块的“行索引范围”：
# 每个程序实例处理一个 BLOCK_SIZE_M × BLOCK_SIZE_N 的子块，
# 这里通过 pid_m 确定该子块在 A 中起始的行号（块行号），再加上线程内的行偏移；
# 若 M 不是 BLOCK_SIZE_M 的整数倍，通过模运算防止越界访问。
```

```
offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M

# 计算 B 矩阵当前块的“列索引范围”:
# 与 A 类似，pid_n 确定当前块在 B 中的起始列号（块列号），再加上线程内的列偏移;
# 同样用模运算解决 N 不能整除 BLOCK_SIZE_N 的边界问题。
offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N

# 生成 K 维度上的偏移量:
# K 是 A 的列数、B 的行数，即“累加维度”，按 BLOCK_SIZE_K 分块;
# 不需边界保护，由后续掩码机制控制越界读取。
offs_k = tl.arange(0, BLOCK_SIZE_K)

# 计算 A 矩阵每个线程要加载的元素地址:
# offs_am 是当前块的行索引（沿第 0 维展开）;
# offs_k 是当前累加块的列索引（沿第 1 维展开）;
# stride_am 和 stride_ak 分别是 A 在行和列方向上的内存步长。
a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)

# 计算 B 矩阵每个线程要加载的元素地址:
# offs_k 是当前累加块的行索引（沿第 0 维展开）;
# offs_bn 是当前块的列索引（沿第 1 维展开）;
# stride_bk 和 stride_bn 分别是 B 在行和列方向上的内存步长。
b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
```

在矩阵乘法的内循环中，每次迭代都需要处理一小段沿 **K 维度**（即 A 的列方向、B 的行方向）对齐的数据块，并将其贡献累加到输出结果中。由于整个乘法过程本质上是对 K 维度上的乘积求和，因此我们需要逐步推进 A 和 B 的指针，分别读取下一段待乘的列块（对 A 来说）和行块（对 B 来说）。具体来说，

A 的列偏移需要前进 BLOCK_SIZE_K * stride_ak，而 B 的行偏移则前进 BLOCK_SIZE_K * stride_bk，以确保两个指针始终指向 K 维度上相应位置的数据块。这种方式实现了按块顺序扫描 K 维度的全部数据，并与掩码加载机制配合，有效地支持任意 K 值长度的矩阵乘法，同时避免越界访问。典型的内循环指针更新实现如下：

```
# 沿 K 维度移动 A 矩阵指针：每次前进 BLOCK_SIZE_K 个元素
# stride_ak 表示 A 矩阵在 K 维度（列方向）的步长
a_ptrs += BLOCK_SIZE_K * stride_ak;


# 沿 K 维度移动 B 矩阵指针：每次前进 BLOCK_SIZE_K 个元素
# stride_bk 表示 B 矩阵在 K 维度（行方向）的步长
b_ptrs += BLOCK_SIZE_K * stride_bk;
```

在并行计算矩阵乘法时，Triton 通常会为每个程序实例（program）分配输出矩阵 CCC 的一个子块，大小为 [BLOCK_SIZE_M, BLOCK_SIZE_N]。最简单的调度方式是按照线性 ID（program_id）逐个分配程序实例，并将该线性 ID 映射回二维的块坐标（行索引 pid_m，列索引 pid_n）。这种方式虽然实现简单，但往往存在性能瓶颈。原因在于传统行主序（row-major）或列主序（column-major）下，线性递增的程序 ID 在物理内存中往往对应的计算块不具备良好的空间局部性，导致缓存行冲突、重复访存、以及 L2 缓存命中率低。下列代码展示了这一基础调度逻辑：先计算出 M 和 N 方向的分块数量 grid_m 和 grid_n，再将一维的 pid 映射为二维的 (pid_m, pid_n) 坐标，以定位每个程序实例要处理的输出子块。

```
# 获取当前程序实例的全局 ID (1D 启动网格中的线性索引)
pid = triton.program_id(0);


# 计算网格维度：
# 沿 M 方向的分块数（向上取整）
grid_m = (M + BLOCK_SIZE_M - 1) // BLOCK_SIZE_M;
```

```
# 沿 N 方向的分块数（向上取整）
grid_n = (N + BLOCK_SIZE_N - 1) // BLOCK_SIZE_N;


# 将线性 pid 转换为 2D 分块坐标:
# pid_m = 行分块索引 (0 <= pid_m < grid_m)
pid_m = pid / grid_n;
# pid_n = 列分块索引 (0 <= pid_n < grid_n)
pid_n = pid % grid_n;
```

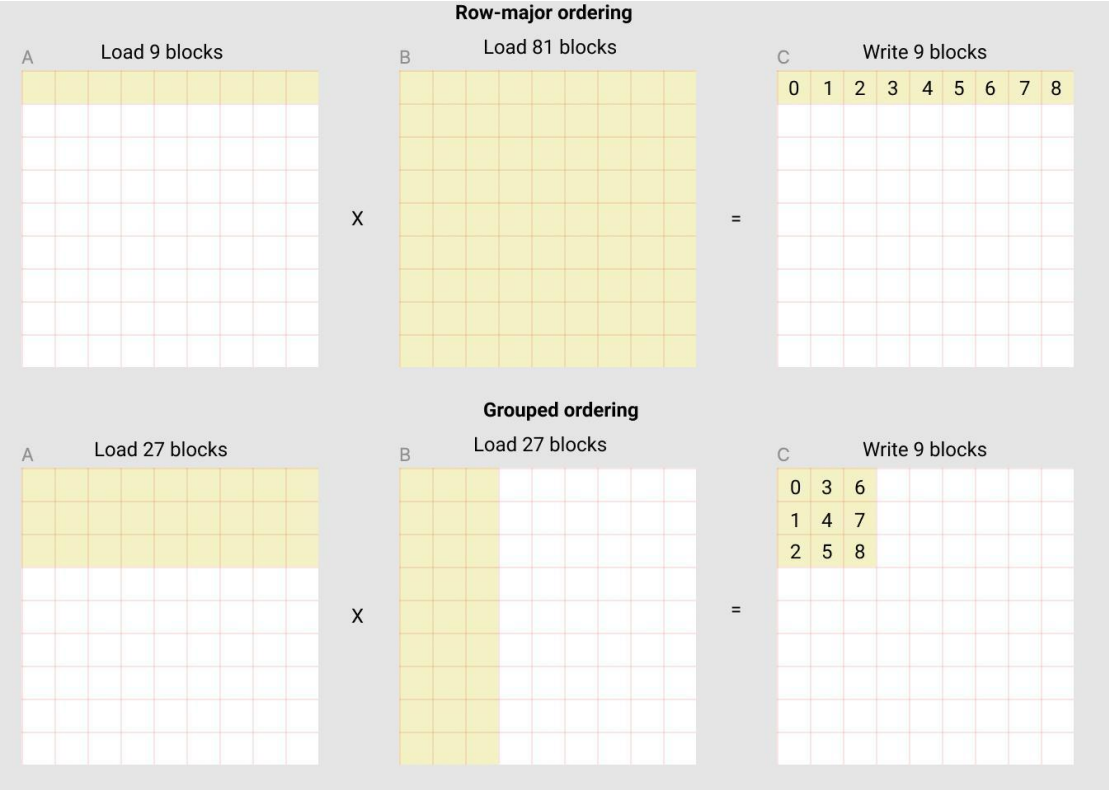
为了进一步提升缓存命中率与内存访问效率，Triton 引入了一种基于“超级分块（super-grouping）”的调度策略。该策略的核心思想是将输出矩阵 C 的多个相邻“行块”组织成一个 **超级分组（super group）**，每个分组包含 GROUP_SIZE_M 行。在每个分组内部，程序实例按列优先顺序（column-major）依次遍历所有列块，延迟在 M 维度上的切换，从而强化对输入矩阵 A（按行访问为主）的数据复用。这样的调度策略不仅改善了缓存局部性，还减少了对同一数据块的重复加载，提升了 L2 缓存的利用率。下列代码展示了该策略的实现过程：通过计算超级分组 ID、起始行号和组内偏移，将一维的 pid 映射为分组中的二维块坐标 (pid_m, pid_n)，从而形成按列优先排列的块调度顺序。

```
# 获取当前程序实例的线性 ID（从 0 开始，全局唯一）
# Triton 启动的一维 grid 中，每个并行线程对应一个 pid
pid = tl.program_id(axis=0)


# 计算分块总数:
# 输出矩阵 C 被分成 num_pid_m × num_pid_n 个块，每块大小为 BLOCK_SIZE_M × BLOCK_SIZE_N
num_pid_m = tl.cdiv(M, BLOCK_SIZE_M) # 沿 M 维（行方向）的分块数量
num_pid_n = tl.cdiv(N, BLOCK_SIZE_N) # 沿 N 维（列方向）的分块数量


# 计算超级分组的大小（以程序实例数计）:
```

```
# 每个“超级分组”负责 GROUP_SIZE_M 行的计算
# 每一行要遍历 N 方向上所有列块（共 num_pid_n 个）
# 所以每个超级分组总共包含 GROUP_SIZE_M × num_pid_n 个程序实例
num_pid_in_group = GROUP_SIZE_M * num_pid_n

# 计算当前程序实例所属的超级分组编号：
# 整个调度空间被划分成若干个超级分组，每个分组编号从 0 开始
# 一个超级分组内有 num_pid_in_group 个 pid
# 所以当前 pid 属于的组就是：pid // num_pid_in_group
group_id = pid // num_pid_in_group  # 超级分组的编号（0-based）

# 计算当前分组在 M 维上的起始块编号：
# 每个分组负责连续的 GROUP_SIZE_M 行块，从 group_id * GROUP_SIZE_M 开始
first_pid_m = group_id * GROUP_SIZE_M

# 考虑边界情况：当 M 维上的分块总数不是 GROUP_SIZE_M 的整数倍时，
# 最后一个分组可能不够 GROUP_SIZE_M 行，因此需要取最小值
group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)

# 将线性 pid 映射为当前超级分组内的二维坐标（行优先 → 列主序）
# 行索引 pid_m：当前组起始行 + pid 在组内的行偏移
pid_m = first_pid_m + (pid % group_size_m)
# 列索引 pid_n：组内 pid 相对于 group_size_m 的整除值，表示其对应的列块
pid_n = (pid % num_pid_in_group) // group_size_m
```

在 Triton 实现的矩阵乘法中，将矩阵划分为 9×9 的输出块网格后，不同的调度策略会显著影响访存效率。若采用传统的行主序调度，计算首行的 9 个输

出块时，每个输出块都需从全输入矩阵中独立加载相应数据，导致总共需加载约 90 个输入块到片上 SRAM（静态随机存储器，一种集成在处理器芯片内部的高速缓存），数据存在较大冗余。而采用分组调度策略（例如设置 `GROUP_SIZE_M=3`），则每次计算一个 3×3 的超级分组，在该区域内共享输入数据块，仅需加载约 54 个输入块，即可完成相同数量的输出块计算，从而显著减少冗余加载，提升约 40% 的数据复用率。这种策略优化了内存带宽利用，有助于充分发挥 GPU 的计算和缓存资源。



图表 22 Triton 矩阵乘法调度策略对比

在深度学习中，**激活函数（Activation Function）** 是一种在神经网络计算中常见的非线性操作，用于增加模型表达能力，使其能够学习复杂的输入-输出映射关系。在矩阵乘法内核中加入激活函数的目的是支持典型的网络层融合（如 Dense + ReLU 或 Dense + LeakyReLU），从而减少单独的后处理步骤，提升整体执行效率，这也与 Triton 为深度学习算子服务的主题相符，而不拘泥于单纯的矩阵乘法。在本例中，激活函数是通过传入 ACTIVATION 参数控制的，例如 "leaky_relu" 会启用带泄漏因子的 ReLU 变体；如果为空字符串，则不执行任何激活操作。我们给出完整的实现例子：

```
import torch
import triton
import triton.language as tl


# 自动调优装饰器：通过基准测试选择最优配置
# configs: 待测试的配置列表，包含分块大小/组大小等参数
# key: 触发重新调优的输入维度变化条件
@triton.autotune(
    configs=[
        # 配置 1：大块计算（128x256x64），适合计算密集型场景
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 64, 'GROUP_SIZE_M': 8}, num_stages=3, num_warps=8),
        # 配置 2：中等块大小，平衡计算和内存访问
        triton.Config({'BLOCK_SIZE_M': 64, 'BLOCK_SIZE_N': 256, 'BLOCK_SIZE_K': 32, 'GROUP_SIZE_M': 8}, num_stages=4, num_warps=4),
        # 其他配置：不同分块组合，适应不同硬件特性
        ...
    ],
    key=['M', 'N', 'K'], # 当这些输入维度变化时重新调优
)
@triton.jit
def matmul_kernel(
        a_ptr, b_ptr, c_ptr, # 输入输出矩阵指针
        M, N, K, # 矩阵维度
        stride_am, stride_ak, # A 矩阵步长（行/列）
        stride_bk, stride_bn, # B 矩阵步长（行/列）
        stride_cm, stride_cn, # C 矩阵步长（行/列）
        BLOCK_SIZE_M: tl.constexpr, # M 方向分块大小
```

```
    BLOCK_SIZE_N: tl.constexpr,  # N 方向分块大小
    BLOCK_SIZE_K: tl.constexpr,  # K 方向分块大小
    GROUP_SIZE_M: tl.constexpr,  # 分组大小（L2 优化）
    ACTIVATION: tl.constexpr  # 激活函数类型
):
    """矩阵乘法内核 (C = A @ B)

    参数:
        A: (M, K) 行主序矩阵
        B: (K, N) 行主序矩阵
        C: (M, N) 输出矩阵

    特性:
        - 支持自动分块大小选择
        - L2 缓存优化分组计算
        - 边界条件安全处理
        - 可选激活函数支持
    """

    # === 分块坐标计算 ===
    # 将线性 PID 转换为 2D 分块坐标，采用分组策略优化 L2 缓存
    pid = tl.program_id(axis=0)  # 当前计算单元 ID
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)  # M 方向分块数
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)  # N 方向分块数
    # 分组计算参数（提高数据局部性）
    num_pid_in_group = GROUP_SIZE_M * num_pid_n  # 每组包含的计算单元数
    group_id = pid // num_pid_in_group  # 当前组 ID
```

```
    first_pid_m = group_id * GROUP_SIZE_M  # 组起始行 ID
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)  # 实际组大小（处理边界）
    # 最终分块坐标
    pid_m = first_pid_m + (pid % group_size_m)  # 行分块坐标
    pid_n = (pid % num_pid_in_group) // group_size_m  # 列分块坐标

    # === 指针初始化 ===
    # 计算初始内存偏移（处理非对齐边界）
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M  # A 行偏移
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N  # B 列偏移
    offs_k = tl.arange(0, BLOCK_SIZE_K)  # K 维偏移

    # 构造分块指针（行主序内存访问）
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)  # A 分块指针
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)  # B 分块指针

    # === 分块矩阵乘法 ===
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)  # 累加器
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):  # K 维度分块循环
        # 带掩码的安全加载（处理边界块）
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
```

```
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator += tl.dot(a, b)  # 矩阵乘累加
        # 指针更新（移动到下一个 K 分块）
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk

    # === 后处理与存储 ===
    if ACTIVATION == "leaky_relu":
        accumulator = leaky_relu(accumulator)  # 应用激活函数
    c = accumulator.to(tl.float16)  # 类型转换

    # 计算输出位置（处理非对齐边界）
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)  # C 行偏移
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)  # C 列偏移
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]  # C 指针
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)  # 存储掩码
    tl.store(c_ptrs, c, mask=c_mask)  # 安全存储

# LeakyReLU 激活函数实现
@triton.jit
def leaky_relu(x):
    """LeakyReLU 激活函数 (alpha=0.01)"""
    x = x + 1  # 示例修改，实际应移除
    return tl.where(x >= 0, x, 0.01 * x)
```

```
def matmul(a: torch.Tensor, b: torch.Tensor, activation: str = ""):
    """矩阵乘法封装函数

    参数:
        a: (M, K) 输入矩阵
        b: (K, N) 输入矩阵
        activation: 激活函数类型（""或"leaky_relu"）

    返回:
        c: (M, N) 结果矩阵
    """
    # 输入验证
    assert a.shape[1] == b.shape[0], "矩阵维度不匹配"
    assert a.is_contiguous(), "输入 A 必须是内存连续的"
    assert b.is_contiguous(), "输入 B 必须是内存连续的"

    M, K = a.shape
    K, N = b.shape
    c = torch.empty((M, N), device=a.device, dtype=a.dtype)  # 预分配输出

    # 动态计算启动网格（基于自动调优参数）
    grid = lambda META: (
        triton.cdiv(M, META['BLOCK_SIZE_M']) * triton.cdiv(N,
META['BLOCK_SIZE_N']),
    )

    # 启动内核
    matmul_kernel[grid](
```

```
        a, b, c,  # 数据指针
        M, N, K,  # 维度信息
        # 传入矩阵的行/列步长信息:
        # 举例：a.stride(0) 表示在 A 中向下一行（M 方向）元素的内存间距，单位是元素数
        #       a.stride(1) 表示在 A 中向右一列（K 方向）元素的间距
        # Triton 使用显式步长来支持任意内存布局（如转置矩阵）
        a.stride(0), a.stride(1),  # A 矩阵的行/列步长
        b.stride(0), b.stride(1),  # B 矩阵的行/列步长
        c.stride(0), c.stride(1),  # C 矩阵的行/列步长

        # 激活函数类型参数（默认为空字符串，表示无激活）
        # 可选值："（无激活），"leaky_relu"（启用 LeakyReLU）
        ACTIVATION=activation    )
    return c
```

启动内核时的传参包括了矩阵的步长，由于 Triton 是低层 DSL（Domain-Specific Language，领域特定语言，专为特定计算任务设计的编程语言），需要明确指定输入输出矩阵在内存中的行/列步长（stride），即不同维度的访问间隔。这种设计允许 Triton 内核灵活支持各种内存布局（如行主序、列主序、转置等），同时确保对底层数据访问的一致性与高效性。

Softmax 融合内核：SRAM 数据复用与规范化流程

这个例子主要体现 program 内的 SRAM 的编程就是类似于 kernel 内的 shared memory 编程，但是不用考虑 bank conflict 以及 shared memory 的容量限制了，Triton 会自动处理这一切。

我们以 Pytorch 朴素实现的 softmax 运算为例进行分析。

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } z = (z_1, \ldots, z_K) \in R^K$$

公式 1 softmax **计算公式**

```
@torch.jit.script
def naive_softmax(x):
    """计算矩阵 x 的逐行 softmax（数值稳定实现）

    实现说明:
    1. 每行减去最大值，避免指数计算中的数值溢出（保持数学等价）
    2. 每一步显式分解，便于分析内存访问模式和优化计算

    输入:
        x: [M, N] 张量，表示 M 行 N 列的矩阵

    返回:
        [M, N] 张量，softmax 概率输出

    内存访问统计（以元素为单位，输入尺寸为 M×N）:
    - 步骤 1（max）:
        - 读取 MN 次（读入所有元素以找每行最大值）
        - 写入 M 次（每行最大值结果，大小为 [M]）
    - 步骤 2（减 max）:
        - 读取 MN 次（原始 x）
        - 读取 M 次（广播 x_max）
        - 写入 MN 次（结果 z）
    - 步骤 3（exp）:
        - 读取 MN 次（z）
        - 写入 MN 次（exp(z) 结果）
    - 步骤 4（sum）:
        - 读取 MN 次（exp(z)）
        - 写入 M 次（每行和）
```

- 步骤 5（归一化）：
  - 读取 MN 次（exp(z)）
  - 读取 M 次（广播分母）
  - 写入 MN 次（最终 softmax 输出）

总计（元素访问次数）：
  - 读取：5MN + 2M 次
  - 写入：3MN + 2M 次

注意：此处统计为近似估计，仅反映访存数量级，不考虑缓存优化或向量化等硬件细节。
"""
# 阶段 1：行最大值计算（避免指数溢出）
x_max = x.max(dim=1)[0] # [M]向量

# 阶段 2：数值稳定化处理
z = x - x_max[:, None] # 广播减 [M, N]

# 阶段 3：指数运算
numerator = torch.exp(z) # [M, N]

# 阶段 4：归一化分母计算
denominator = numerator.sum(dim=1) # [M]

# 阶段 5：概率归一化
ret = numerator / denominator[:, None] # [M, N]

```
    return ret
```

在 PyTorch 的原生实现中，朴素版本的 softmax 操作面临着显著的内存访问效率问题。这个实现需要从 DRAM 中读取多达 5MN+2M 个元素，同时还要写回 3MN+2M 个元素。这种频繁的内存访问模式造成了严重的带宽浪费，主要是因为中间结果需要反复读写。实际上，通过设计一个定制化的融合内核，我们可以将数据读取次数大幅降低到仅需 MN+M 次。这种优化理论上能带来约 4 倍的性能提升，这正是高性能计算中常说的"数据局部性"优势的体现。虽然 PyTorch 的 JIT 编译器尝试通过自动内核融合来优化这类操作，但在实际中这种自动优化往往难以达到理想效果。特别是在处理 softmax 这种需要复杂数值稳定化计算的场景时，编译器的优化能力显得力不从心。

在 Triton 的实现方案中采用了一种行级并行的计算策略。每个计算单元负责处理输入矩阵的一整行数据，在计算过程中巧妙地处理了数值稳定性问题。值得注意的是，Triton 对计算块的大小有着特殊要求：每个处理块必须包含 2 的幂次方个元素。为了适应任意形状的输入矩阵，Triton 实现中加入了智能的填充机制，并通过精心设计的内存访问掩码来确保计算的正确性。这种设计既遵守了硬件约束，又保持了算法的通用性。

```
@triton.jit
def softmax_kernel(output_ptr, input_ptr, input_row_stride, output_row_stride,
n_cols, BLOCK_SIZE: tl.constexpr):
    """
    Triton 实现的 Softmax 核函数，逐行计算矩阵的 Softmax

    参数:
        output_ptr: 输出矩阵指针
        input_ptr: 输入矩阵指针
        input_row_stride: 输入矩阵行步长（单位：元素个数）
```

```
    output_row_stride: 输出矩阵行步长
    n_cols: 每行的实际列数
    BLOCK_SIZE: 分块大小（2 的幂次）
    """
    # 计算当前处理的行索引，每行由一个 CUDA 线程块独立处理
    row_idx = tl.program_id(0)

    # 计算当前行的起始指针位置，考虑行步长参数
    row_start_ptr = input_ptr + row_idx * input_row_stride

    # 生成列偏移量，BLOCK_SIZE 需为大于 n_cols 的最小 2 的幂
    # 这样可以确保每行数据都能完整处理
    col_offsets = tl.arange(0, BLOCK_SIZE)

    # 构造当前行所有元素的访问指针
    input_ptrs = row_start_ptr + col_offsets

    # 安全加载行数据：超出实际列数的位置用负无穷填充
    # 这是为了处理非对齐的尾部数据
    row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-float('inf'))

    # 数值稳定化处理：减去行最大值避免指数溢出
    row_minus_max = row - tl.max(row, axis=0)

    # 计算指数部分，Triton 使用近似快速指数函数
    # 精度类似 CUDA 的__expf，牺牲少量精度换取性能
    numerator = tl.exp(row_minus_max)
```

```
    # 计算归一化分母（行内求和）
    denominator = tl.sum(numerator, axis=0)

    # 最终 softmax 结果计算
    softmax_output = numerator / denominator

    # 准备输出指针，考虑行步长参数
    output_row_start_ptr = output_ptr + row_idx * output_row_stride
    output_ptrs = output_row_start_ptr + col_offsets

    # 安全存储结果，只写入有效的列位置
      tl.store(output_ptrs, softmax_output, mask=col_offsets < n_cols)

def softmax(x):
    """计算矩阵 x 的 softmax 变换（逐行）

    参数:
        x: 输入二维张量，形状为(n_rows, n_cols)
    返回:
        同形状的输出张量，每行和为 1
    """
    n_rows, n_cols = x.shape

    # 确定计算块大小：取大于列数的最小 2 的幂
    # 这是为了满足 Triton 硬件的对齐要求
    BLOCK_SIZE = triton.next_power_of_2(n_cols)

    # 动态调整 warp 数量：随着处理行宽的增加
```

```
    # 使用更多 warp 来保持计算效率
    num_warps = 4  # 默认使用 4 个 warp
    if BLOCK_SIZE >= 2048:  # 宽行使用更多资源
        num_warps = 8
    if BLOCK_SIZE >= 4096:  # 超宽行配置
        num_warps = 16

    # 预分配输出张量（保持输入形状和类型）
    y = torch.empty_like(x)

    # 启动内核配置：
    # - 一维启动网格，每行一个计算实例
    # - 传递输入/输出指针和形状信息
    # - 指定计算资源参数（warp 数/块大小）
    softmax_kernel[(n_rows,)](
        y, x,
        x.stride(0),  # 输入矩阵 x 中，相邻两行元素之间的内存偏移量（以元素为单位）
        y.stride(0),  # 输出矩阵 y 中，相邻两行元素之间的内存偏移量
        n_cols,       # 每行实际有效的列数（非 BLOCK_SIZE）
        num_warps=num_warps,  # 为当前 BLOCK_SIZE 选择的并行 warp 数量
        BLOCK_SIZE=BLOCK_SIZE,  # 每次处理的列块大小（向上对齐到 2 的幂）
    )
    return y
```

这段程序虽然没有显式使用 @triton.autotune 装饰器，但它通过动态调整 BLOCK_SIZE 和 num_warps 来适配输入张量的宽度，本质上体现了“静态调优”的思想。例如：对于非常宽的矩阵行（如列数大于 2048 或 4096），程序自动选

择更大的计算块和更多的 warps（并行线程组）以充分利用硬件资源、提升吞吐率。这种根据输入规模自适应调整资源配置的做法，就是 Triton 编程中的典型优化实践之一，也为进一步使用 @autotune 做动态调优提供了基础结构。