7 算子开发

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 使用 Triton 编写自定义 GPU 算子；
- ⋆⋆⋆ 了解 Triton 的自动参数调优机制；
- ⋆⋆⋆ 可以将 Triton 算子集成到 PyTorch 模型。

7.1 算子开发

算子开发是指针对深度学习模型中的基本计算单元（例如矩阵乘法、卷积等），在特定硬件平台上进行高性能实现与优化的过程。尽管直接使用 CUDA C++ 编写算子具有极高的灵活性，但其实现过程往往较为复杂。开发者需要手动设计线程块（block）与线程（thread）的划分策略，显式管理全局内存与共享内存之间的数据搬运，同时还需处理同步与并发问题，并通过反复实验进行性能调优。这种底层优化方式对开发者的硬件理解能力和并行编程经验提出了较高要求。

为降低算子开发的门槛，近年来出现了一类基于领域特定语言（Domain-Specific Language, DSL）的 AI 编译器系统。这类系统允许开发者使用更高层次的抽象来描述计算逻辑，由编译器自动完成并行化映射、调度优化以及目标代码生成，从而显著提升开发效率与代码可移植性。典型代表包括 OpenAI 的 Triton[1]和 nVidia 公司新出的 cuTile[2]。这两款软件都是开源软件，可以与 PyTorch 无缝集成。

本章内容以 Triton 为例，详细讲解算子开发的过程。

7.2 Triton 算子开发

Triton 是一种嵌入在 Python 中的领域特定语言（DSL），专门用于高性能 GPU 算子的开发。开发流程上，用户通过 `@triton.jit` 装饰器定义 Kernel，随后由编译器自动生成优化的 CUDA/PTX 代码，从而避免手写底层 CUDA 的复杂性。在编程模型上，Triton 与 CUDA 存在一定的对应关系，但整体抽象更加简洁：

- **Program Instance 与并行映射**：Triton 以 program instance 作为基本执行单元（类似 CUDA 的 thread block），每个 instance 处理一块数据（tile）。通过 `tl.program_id(dim)` 获取其在 grid 中的位置（类似 `blockIdx`）并计算数据偏移；线程级并行由向量化操作隐式表达，无需显式管理 `threadIdx` 或线程布局。
- **掩码（Mask）机制**：通过掩码控制访存有效性，自动避免越界访问，替代 CUDA 中常见的边界判断，如 `if (idx < N)`。
- **高效内存访问与广播**：内建矢量化 load/store 和广播语义，便于表达连续访存和数据复用，从而提升内存带宽利用率。

总体而言，Triton 在保留 CUDA 核心执行模型的基础上，通过更高层的抽象，简化了并行映射与索引计算，同时仍能生成高性能代码。

[1] `https://github.com/openai/triton`

[2] `https://github.com/nvidia/cutile-python`

7.2.1 编写算子

本节先以向量逐元素相乘为例，演示 Triton 算子开发方法，然后将其扩展到矩阵乘法以及分块矩阵乘法。

逐元素相乘 代码 7.1 使用 Triton 实现了向量的逐元素相乘算子，其关键步骤如下：

- 使用 `@triton.jit` 装饰器将 `vecmul_kernel` 标记为 GPU Kernel，由编译器生成高效的底层实现。
- 参数 `BLOCK_SIZE: tl.constexpr` 表示该值在编译期确定，使编译器能够进行循环展开和访存优化。
- 在 Kernel 内部，`tl.program_id(0)` 获取当前 program instance 在第 0 维网格中的编号，其作用类似于 CUDA 中的 `blockIdx.x`。每个 program instance 负责处理一段连续的数据，其全局索引通过如下方式计算：

$$\texttt{offsets} = \texttt{pid} \times \texttt{BLOCK\_SIZE} + \texttt{tl.arange}(0, \texttt{BLOCK\_SIZE})$$

- 基于索引构造掩码 `mask = offsets < N`，用于避免越界访问。当掩码为 `False` 时，对应位置不会发生实际内存操作，从而避免显式分支判断。
- 基于 `offsets` 与 `mask`，通过 `tl.load` 和 `tl.store` 对数据进行批量读取与写回，实现向量化访存。

```python
import triton
import triton.language as tl

@triton.jit
def vecmul_kernel(
    x_ptr, y_ptr, c_ptr,
    N,                # 元素总数
    BLOCK_SIZE: tl.constexpr  # 每个 block 处理的元素数
):
    # 获取当前 block 的 ID (相当于 blockIdx.x)
    pid = tl.program_id(0)

    # 计算当前 block 负责的全局偏移
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)

    # 构建掩码，防止越界访问
    mask = offsets < N

    # 加载数据 (自动处理内存合并)
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)

    # 执行逐元素乘法
    output = x * y

    # 写回结果
    tl.store(c_ptr + offsets, output, mask=mask)
```

代码 7.1: Triton 实现逐元素相乘

矩阵乘法 设矩阵 $A \in \mathbb{R}^{M\times K}, B \in \mathbb{R}^{K\times N}, C = AB$。我们将输出矩阵 C 划分为大小为 BLOCK_SIZE_M× BLOCK_SIZE_N 的子块，每个 program instance 负责计算其中一个子块 (pid_m, pid_n)。

对于每个子块，其计算可以表示为对应子矩阵的矩阵乘法：

$$C^{(block)} = A^{(block)} \cdot B^{(block)}$$

其中：

- $A^{(block)} \in \mathbb{R}^{\text{BLOCK\_SIZE\_M}\times K}$ 表示 A 中对应行范围的子矩阵
- $B^{(block)} \in \mathbb{R}^{K\times \text{BLOCK\_SIZE\_N}}$ 表示 B 中对应列范围的子矩阵

该形式等价于在 K 维上的求和：

$$C_{m,n} = \sum_{k=0}^{K-1} A_{m,k} B_{k,n}$$

代码 7.2给出了对应实现。

```python
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr,
    M, N, K,
    BLOCK_SIZE_M: tl.constexpr = 64,
    BLOCK_SIZE_N: tl.constexpr = 64
):
    # 每个 program instance 对应输出矩阵 C 的一个子块
    pid_m = tl.program_id(0)  # 行方向 block index
    pid_n = tl.program_id(1)  # 列方向 block index

    # 当前 block 对应的行/列索引
    offs_m = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)  # (BM,)
    offs_n = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)  # (BN,)
    # K 维完整索引（一次性展开）
    offs_k = tl.arange(0, K)  # (K,)
    # 加载 A 的子矩阵：(BM, K)
    a = tl.load(a_ptr + offs_m[:, None] * K + offs_k[None, :],
        mask=(offs_m[:, None] < M) & (offs_k[None, :] < K), other=0.0
    )
    # 加载 B 的子矩阵：(K, BN)
    b = tl.load(b_ptr + offs_k[:, None] * N + offs_n[None, :],
        mask=(offs_k[:, None] < K) & (offs_n[None, :] < N),other=0.0
    )
    # 矩阵乘法（在 K 维上完成隐式求和）
    acc = tl.dot(a, b)  # (BM, BN)
    # 写回结果
    tl.store(c_ptr + offs_m[:, None] * N + offs_n[None, :], acc,
        mask=(offs_m[:, None] < M) & (offs_n[None, :] < N)
    )
```

代码 7.2: 2D Grid 版本 Triton 矩阵乘法

该表达形式上避免了循环，比较简洁。但需要一次性加载大小为 BLOCK_SIZE_M × K 和 K × BLOCK_SIZE_N 的数据，当 K 较大时会带来较高的寄存器和内存访问开销，成为性能瓶颈。

分块矩阵乘法 下面介绍 Triton 上的分块矩阵乘法实现。其核心思想是将 K 维划分为大小为 BLOCK_K 的子块，并在每次迭代中仅处理一个 K 子块对应的计算贡献。具体而言，每个 program instance 负责计算输出矩阵 C 的一个子块（大小为 BLOCK_SIZE_M × BLOCK_SIZE_N），并在 K 维循环中累加局部结果。每一步计算均对应于如下形式的子块矩阵乘法：

$$(\text{BLOCK\_SIZE\_M} \times \text{BLOCK\_SIZE\_K}) \times (\text{BLOCK\_SIZE\_K} \times \text{BLOCK\_SIZE\_N})$$

其结果被逐步累加至 C 的对应子块中。由于 K 维被划分为多个较小的子块，A 和 B 的局部数据可以在计算过程中被重复利用，从而提升缓存命中率并减少全局内存访问开销，最终在 GPU 上获得更高的计算性能。代码 7.3实现了上述分块矩阵乘法。

```python
@triton.jit
def matmul_kernel(
    a_ptr, b_ptr, c_ptr,
    M, N, K,
    BLOCK_SIZE_M: tl.constexpr = 64,
    BLOCK_SIZE_N: tl.constexpr = 64,
    BLOCK_SIZE_K: tl.constexpr = 32,
):

    # 每个 program instance 直接对应一个 2D tile (pid_m, pid_n)
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)

    # 当前 block 在全局矩阵中的索引范围
    offs_m = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_n = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    offs_k = tl.arange(0, BLOCK_SIZE_K)

    acc = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)

    num_k = tl.cdiv(K, BLOCK_SIZE_K) # 计算 K 维度上的子块个数，向上取整

    for k in range(num_k):
        k_offsets = k * BLOCK_SIZE_K + offs_k

        # A: (BLOCK_SIZE_M, BLOCK_SIZE_K)
        a = tl.load(a_ptr + offs_m[:, None] * K + k_offsets[None, :],
            mask=(offs_m[:, None] < M) & (k_offsets[None, :] < K), other=0.0
        )

        # B: (BLOCK_SIZE_K, BLOCK_SIZE_N)
        b = tl.load(b_ptr + k_offsets[:, None] * N + offs_n[None, :],
            mask=(k_offsets[:, None] < K) & (offs_n[None, :] < N), other=0.0
        )

        acc += tl.dot(a, b)

    tl.store( c_ptr + offs_m[:, None] * N + offs_n[None, :], acc,
        mask=(offs_m[:, None] < M) & (offs_n[None, :] < N)
    )
```

代码 7.3: 2D Grid Triton 分块矩阵乘法

7.2.2 调用算子

由于 Triton 使用 Python 编写，因此调用非常方便。代码 7.4 和 7.5 分别给出了向量逐元素相乘以及矩阵乘法内核主机端调用过程。

其中，需要着重说明的是 `grid` 用于定义 Triton kernel 的并行执行规模。在 Triton 中，每个 kernel 实例（program instance）负责处理一个数据块，而 `grid` 决定了需要启动多少个 instance。从形式上看，`grid` 支持一维、二维甚至三维结构，其本质与 CUDA 中的 `gridDim` 类似。

```python
# Host 端函数：封装 Triton Kernel 调用过程
def vecmul(x, y):
    assert x.shape == y.shape, "向量维度不匹配"
    out = torch.empty_like(x)  # 创建输出向量，与输入形状相同
    N = x.numel() # 计算待处理的元素总数

    # 定义 grid 规模：根据 BLOCK_SIZE 计算所需 program instance 数量
    grid = lambda meta: (triton.cdiv(N, meta['BLOCK_SIZE']))

    # 启动 Triton Kernel
    # 等价于：vecmul_kernel[triton.cdiv(N, BLOCK_SIZE)](x, y, out, N, BLOCK_SIZE=1024)
    vecmul_kernel[grid](x, y, out, N, BLOCK_SIZE=1024)

    return out
```

代码 7.4: Host 端调用封装：Triton 实现逐元素相乘

```python
# Host 端函数：封装 Triton 2D 矩阵乘法 kernel 调用
def matmul(a, b):
    assert a.shape[1] == b.shape[0], "矩阵维度不匹配"
    M, K = a.shape
    K_, N = b.shape
    out = torch.empty((M, N), device=a.device, dtype=a.dtype)

    # 2D grid：每个 program instance 对应一个 (tile_m, tile_n)
    grid = lambda META: (
        triton.cdiv(M, META['BLOCK_SIZE_M']),
        triton.cdiv(N, META['BLOCK_SIZE_N']),
    )

    # 启动 Triton kernel
    matmul_kernel[grid](a, b, out, M, N, K, BLOCK_SIZE_M=64, BLOCK_SIZE_N=64)

    return out
```

代码 7.5: Host 调用封装：Triton 2D Grid 矩阵乘法

在 Triton 的实际实现中，也可以将二维 grid 展平成一维进行调度，再在 kernel 内通过 `program_id` 手动恢复二维索引，以简化调度逻辑，提升编译器优化空间。

7.3 Triton 的 Autotune 机制

在前面的实现中，我们在调用 kernel 时通常需要手动指定超参数，例如 BLOCK_SIZE_M、BLOCK_SIZE_N、BLOCK_SIZE_K 等。这些参数对性能影响非常显著，但其最优组合往往依赖具体硬件（GPU 型号）、数据规模以及访存模式。传统手动调参方式需要对不同配置进行 benchmark 来选择最优组合。这种方式虽然有效，但参数迁移性差，在不同 GPU 上往往需要重新调优。

为了解决这一问题，Triton 提供了 autotune 机制，可以在运行时结合 JIT 编译自动搜索最优 kernel 配置。也就是说，kernel 并不是在编译期一次性生成唯一版本，而是会根据不同配置动态生成多个编译后的 kernel 版本（compiled variants），并在实际执行时进行选择。

以分块矩阵乘法为例，开发者可以定义多个候选配置：

```
@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE_M': 64,  'BLOCK_SIZE_N': 64,  'BLOCK_SIZE_K': 32}),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 64,  'BLOCK_SIZE_K': 32}),
        triton.Config({'BLOCK_SIZE_M': 64,  'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 32}),
        triton.Config({'BLOCK_SIZE_M': 128, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 32}),
    ],
    key=['M', 'N', 'K'],
)
@triton.jit
def matmul_kernel(
    a_ptr, b_ptr, c_ptr,
    M, N, K,
    BLOCK_SIZE_M: tl.constexpr = 64,
    BLOCK_SIZE_N: tl.constexpr = 64,
    BLOCK_SIZE_K: tl.constexpr = 32,
):
```

在这里，`@triton.jit` 表示该 kernel 会在运行时触发 JIT 编译。不同的 autotune 配置会生成不同的已编译 kernel 实例，这些实例在第一次被调用时才会被编译并缓存，从而避免重复编译开销。

除了上述参数，Triton 还提供用于控制线程并行度的 num_warps 和用于调节软件流水线深度的 num_stages 参数。在实际应用中，这些参数也可以加入搜索空间，从而进一步提升性能。

Autotune 的核心流程如下：

1) 根据 shape 输入配置生成候选 kernel 配置集合；

2) 通过 JIT 编译为多个不同的 kernel 编译版本；

3) 分别在 GPU 上执行并测量运行时间；

4) 选择最快的配置，并将对应的编译结果缓存；

5) 后续相同 shape 直接复用已编译的最优 kernel。

因此，autotune 本质上是一个“运行时 JIT + 编译期搜索”的混合优化过程。Triton autotune 将传统人工调参问题转化为自动搜索问题，并通过 JIT 编译机制动态生成和复用多个 kernel 编译版本，从而显著降低 kernel 优化门槛。但需要注意，autotune 会带来首次运行开销：因为需要触发多次 JIT 编译与 benchmark。当搜索空间过大时，这一过程会显著增加启动延迟。因此，在实际工程中，常见策略是先手动设计合理的搜索空间，再交由 autotune 自动选择最优解。

7.4 在 PyTorch 中使用自定义算子

在实际深度学习系统中，我们常需用自定义算子替代框架原生实现，以优化性能、支持新功能或进行算法实验。例如，在注意力机制（Attention）、融合算子（fused operator）以及特殊数据布局计算中，使用 Triton 可以显著减少内存访问开销并提升执行效率。

总体而言，将 Triton 算子集成到 PyTorch 中可以分为以下三个步骤：

1） **实现算子**：使用 Triton 编写底层 kernel，实现具体计算逻辑，如矩阵乘法或卷积；

2） **封装为 PyTorch 接口**：通过 `torch.autograd.Function` 对 kernel 进行封装，定义前向与反向传播，并进一步封装为 `nn.Module`，以便在模型中复用；

3） **在模型中使用算子**：将封装好的模块嵌入到具体网络结构中，从而参与端到端训练与推理。

需要注意的是，Triton 主要负责高性能前向计算，而反向传播通常可以直接使用 PyTorch 的矩阵运算实现，从而降低开发复杂度。

本节以经典的 LeNet-5 模型为例，展示如何将一个用 Triton 编写的自定义全连接层算子集成到 PyTorch 模型中，并参与端到端训练。

7.4.1 全连接层算子

全连接层本质上是矩阵乘法加偏置操作。设输入为 $X \in \mathbb{R}^{B\times \text{in\_features}}$，权重为 $W \in \mathbb{R}^{\text{out\_features}\times \text{in\_features}}$，偏置为 $b \in \mathbb{R}^{\text{out\_features}}$，则输出为：

$$Y = XW^T + b$$

该计算可以直接复用前文的分块矩阵乘法实现。具体而言，全连接层的核心操作 XW^T 等价于一个矩阵乘法，因此可以在代码 7.3的基础上，对输入矩阵与权重矩阵进行分块计算；在此基础上，只需在输出矩阵的每一行上加上偏置向量 b，即可得到最终结果。对应的 Triton 实现如代码 7.6 所示。

```python
@triton.jit
def linear_kernel(in_ptr, weight_ptr, out_ptr, bias_ptr,
                  B, in_features, out_features,
                  BLOCK_SIZE_B: tl.constexpr = 32,
                  BLOCK_SIZE_OUT: tl.constexpr = 64,
                  BLOCK_SIZE_K: tl.constexpr = 32):
    """
    in_ptr: [B, in_features]
    weight_ptr: [out_features, in_features]
    out_ptr: [B, out_features]
    bias_ptr: [out_features]
    """
    # 2D grid: 每个 program instance 负责一个Tile (Batch, Out Feature)
    pid_batch = tl.program_id(0)
    pid_out = tl.program_id(1)
    offs_batch = pid_batch * BLOCK_SIZE_B + tl.arange(0, BLOCK_SIZE_B)          # (BB,)
    offs_out = pid_out * BLOCK_SIZE_OUT + tl.arange(0, BLOCK_SIZE_OUT)  # (BO,)
    offs_k = tl.arange(0, BLOCK_SIZE_K)                                        # (BK,)
    acc = tl.zeros((BLOCK_SIZE_B, BLOCK_SIZE_OUT), dtype=tl.float32)
    num_k = tl.cdiv(in_features, BLOCK_SIZE_K)
    for k in range(num_k):
        k_offsets = k * BLOCK_SIZE_K + offs_k    # (BK,)

        # 加载 X 子块: (BB, BK)
        x_block = tl.load(in_ptr + offs_batch[:, None] * in_features + k_offsets[None, :],
            mask=(offs_batch[:, None] < B) &
            (k_offsets[None, :] < in_features), other=0.0
        )

        # 加载 W 子块: (BK, BO);
        w_block = tl.load(weight_ptr + offs_out[None, :] * in_features + k_offsets[:, None],
            mask=(offs_out[None, :] < out_features) & (k_offsets[:, None] < in_features),
            other=0.0
        )

        # 子块矩阵乘法累加
        acc += tl.dot(x_block, w_block)

    # 加 bias
    if bias_ptr is not None:
        bias = tl.load(bias_ptr + offs_out, mask=offs_out < out_features, other=0.0)
        acc += bias

    # 写回
    tl.store(out_ptr + offs_batch[:, None] * out_features + offs_out[None, :], acc,
        mask=(offs_batch[:, None] < B) & (offs_out[None, :] < out_features)
    )
```

代码 7.6: Triton 实现全连接层算子（分块矩阵乘法版本）

7.4.2 算子封装

为了在 PyTorch 中直接使用 Triton kernel，需要将其封装为 PyTorch 算子，主要包括以下两个步骤：

- **兼容自动微分**：通过继承 `torch.autograd.Function` 并实现 `forward` 和 `backward` 方法，将 Triton kernel 集成到 PyTorch 计算图中，从而支持端到端梯度计算。
- **模块化封装**：通过 `torch.nn.Module` 对算子进行封装，使其能够像原生 PyTorch 层一样被调用。

```python
class TritonLinearFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias=None):
        B, in_features = input.shape
        out_features = weight.shape[0]
        out = torch.empty((B, out_features), device=input.device, dtype=input.dtype)
        # 2D grid: batch 维 × 输出特征维
        grid = lambda meta: ((B + meta['BLOCK_SIZE_B'] - 1) // meta['BLOCK_SIZE_B'],
            (out_features + meta['BLOCK_SIZE_OUT'] - 1) // meta['BLOCK_SIZE_OUT'],
        )
        # 调用 Triton kernel
        linear_kernel[grid](input, weight, out, bias, B, in_features, out_features,
                            BLOCK_SIZE_B=32, BLOCK_SIZE_OUT=64)
        # 保存反向传播所需张量
        ctx.save_for_backward(input, weight, bias)
        return out

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors # 取出 forward 保存的张量
        grad_input = grad_weight = grad_bias = None # 初始化梯度
        if ctx.needs_input_grad[0]:
            # [B, out] @ [out, in] -> [B, in]
            grad_input = grad_output @ weight    # 输入梯度：dX = dY @ W（传递给前一层）
        if ctx.needs_input_grad[1]:
            # [out, B] @ [B, in] -> [out, in]
            grad_weight = grad_output.t() @ input    # 权重梯度：dW = dY^T @ X
        if bias is not None and ctx.needs_input_grad[2]:
            # [B, out] -> sum over batch -> [out]
            grad_bias = grad_output.sum(0)  # bias 梯度：对 batch 维求和
        return grad_input, grad_weight, grad_bias

class TritonLinear(torch.nn.Module):
    def __init__(self, in_features, out_features, bias=True):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = torch.nn.Parameter(torch.randn(out_features, in_features))
        self.bias = torch.nn.Parameter(torch.randn(out_features)) if bias else None

    def forward(self, x):
        return TritonLinearFunction.apply(x, self.weight, self.bias)
```

代码 7.7: PyTorch 自定义全连接层模块

7.4.3 集成到 LeNet-5

下面将自定义算子集成到经典 LeNet-5 网络中，如代码 7.8所示。

```
class LeNet5(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = torch.nn.Conv2d(1, 6, 5)
        self.pool = torch.nn.AvgPool2d(2)
        self.conv2 = torch.nn.Conv2d(6, 16, 5)

        # 使用 Triton 实现的全连接层
        self.fc1 = TritonLinear(16*5*5, 120)
        self.fc2 = TritonLinear(120, 84)
        self.fc3 = TritonLinear(84, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 16*5*5)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

代码 7.8: 集成 Triton 算子的 LeNet-5

练习

1) 用 Triton 实现 Average Pooling 算子，并集成到 LeNet-5 中。

2) 结合上节课的内容，对比分析 CUDA 和 Triton 的方案实现，包括：

- 抽象设计与开发复杂度
- 使用便捷性

3) 结合第三章的内容，分析 Triton 装饰器（@triton.jit）的实现原理。