8 算子融合

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 理解深度神经网络中的常见性能瓶颈及其分析方法；
- ⋆⋆⋆ 掌握算子融合的基本原理与优化动机；
- ⋆⋆ 了解 PyTorch 中算子融合的实现方式及其优化机制；

8.1 问题分析

8.1.1 访存与执行开销分析

以代码 8.1所示的 MLP 模型为例，我们从内存访问等方面分析其性能瓶颈。

```python
class MLP(nn.Module):
    def __init__(self, feature=1, hidden=64):
        super().__init__()
        # 第一层参数：1 → hidden
        self.weight1 = nn.Parameter(torch.randn(feature, hidden))   # (F, H)
        self.bias1   = nn.Parameter(torch.randn(hidden))      # (H,)

        # 第二层参数：hidden → 1
        self.weight2 = nn.Parameter(torch.randn(hidden, 1))   # (H, 1)
        self.bias2   = nn.Parameter(torch.randn(1))           # (1,)

    def forward(self, x):
        x = x @ self.weight1 + self.bias1   # (B, H)
        x = torch.relu(x)
        x = x @ self.weight2 + self.bias2   # (B, 1)
        return x
```

代码 8.1: MLP 模型

设输入为 $x \in \mathbb{R}^{B\times F}$，隐藏层维度为 H。该 MLP 前向传播主要包含两次矩阵乘法：

- 第一层：$(B \times F) \cdot (F \times H) \to (B \times H)$
- 第二层：$(B \times H) \cdot (H \times 1) \to (B \times 1)$

如果不考虑算子融合，可以将矩阵乘法、bias 加法和 ReLU 视为多个独立算子。在 GPU 上，每个算子通常对应一次 kernel 启动，并需要从全局内存读取输入数据，计算完成后再将结果写回全局内存，供后续算子使用。因此，算子之间的中间结果需要在全局内存中显式存储与传递。

在该模型中，$(B \times H)$ 的张量正是不同算子之间传递的中间激活，其访存次数如表 8.1 所示。从整体规模来看，访存主要由输入 $O(BF)$、权重 $O(FH)$ 以及中间激活 $O(BH)$ 构成。其中，中间激活在多个算子之间被反复读取与写回，是主要的数据搬运来源。当 B 和 H 较大时，这部分开销会占主导，使模型更容易受到内存带宽限制。该问题通常可以通过算子融合减少中间结果的读写来缓解。

表 8.1: MLP 前向传播访存分析

阶段	读取（Read）	写入（Write）	主要张量
第一层矩阵乘法 XW_1	$BF + FH$	$B \times H$	x, W_1
第一层 bias 加法	$B \times H + H$	$B \times H$	中间激活, b_1
ReLU 激活	$B \times H$	$B \times H$	中间激活
第二层矩阵乘法 XW_2	$B \times H + H$	B	激活, W_2
第二层 bias 加法	$B + 1$	B	输出, b_2
总计	$\approx BF + FH + 3BH + 2H + 2B$	$\approx 2BH + 2B$	–

在 GPU 上，这一问题还受到片上存储容量的影响。以 nVidia Ampere 架构为例，每个 SM 上共享内存约为 164 KB。例如，当 $B = 32, H = 64$ 时，中间激活大小约为 8 KB（FP32），从容量上可以放入共享内存。但在实际执行中，还需要同时存放输入、权重以及其它线程块的数据，因此能够用于缓存中间激活的空间是有限的。随着 B 或 H 增大，中间激活规模按 $O(BH)$ 增长，很快就无法完全驻留在共享内存中，只能频繁在全局内存与共享内存之间进行数据交换，从而增加访存开销。该问题通常通过分块（tiling）等方法提升局部数据复用（例如分块矩阵乘法）。

8.1.2 算子融合

算子融合（Operator Fusion）是深度学习框架中的一种重要优化技术，其核心思想是将多个连续的算子合并为一个计算单元，从而减少中间张量的读写次数、降低内存访问开销，并提高计算效率。

例如，代码 8.1中的全连接层由矩阵乘法与偏置加法两个独立操作组成：

$$y = xW + b$$

在 PyTorch 中，这两步已经被实现为一个融合算子 `nn.Linear`，从而减少一次内存读写操作，提高计算效率。代码 8.2给出基于 `nn.Linear` 的标准实现。

```python
class MLP(nn.Module):
    def __init__(self, feature=1, hidden=64):
        super().__init__()
        self.fc1 = nn.Linear(feature, hidden)
        self.fc2 = nn.Linear(hidden, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return x
```

代码 8.2: MLP 模型：使用 `nn.Linear` 替换矩阵乘法和加法

改写后，原本由 5 个独立算子产生的全局内存读写，被减少为 3 个融合算子的读写操作，从而显著降低了访存开销。在此基础上，还可以进一步构造更大的融合算子（如将全连接层与 ReLU 合并），以继续减少中间结果的读写次数。接下来，我们将进一步探讨自动化的算子融合技术。

8.2 计算图优化

在深度学习中，模型的计算过程（包括前向计算与反向传播）通常可表示为有向无环图（DAG），称为计算图（Computation Graph）。每个节点代表一个算子，边代表张量数据流。计算图优化是指在保持语义不变的前提下，对计算图进行变换，以通过算子融合减少算子数量、通过降低内存占用提升缓存利用率等。

8.2.1 在 PyTorch 中开启计算图优化

在 PyTorch 中，可以通过 `torch.compile` 启用计算图级别优化，从而实现算子融合与执行加速。如代码 8.3 所示，在模型实例化并迁移至 GPU 后，对模型进行编译优化，后续直接使用返回的编译模型。

```
model = MLP().to(device)
model = torch.compile(model, backend="inductor", mode="default")
```

代码 8.3: 编译 MLP 模型

`torch.compile` 是 PyTorch 2.x 提供的图编译接口，其函数签名支持多个关键参数，用于控制计算图捕获方式、优化策略及执行后端。主要参数如下：

- `model`：待编译的 PyTorch 模型或函数，是必选参数。
- `backend="inductor"`：指定编译后端，默认使用 `TorchInductor`。
- `mode=None`：用于控制编译优化策略的预设模式，在编译开销与运行性能之间进行权衡。常见选项包括：
  - `"default"`：默认模式，在编译时间与运行性能之间取得平衡；
  - `"reduce-overhead"`：减少 Python 调度与框架开销，适用于小模型或频繁调用场景；
  - `"max-autotune"`：通过更激进的 kernel 搜索获得最高运行性能，但编译时间显著增加。
- `fullgraph=False`：控制是否强制进行完整计算图捕获。当设为 `True` 时，若执行过程中发生 graph break（如 Python 控制流无法静态追踪），将直接报错，从而保证图优化的完整性，但会降低兼容性。
- `dynamic=False`：控制是否启用动态形状支持。若为 `True`，模型允许输入张量 shape 在运行时变化，但可能限制部分静态图优化（如算子融合与内存优化）。

8.2.2 PyTorch 计算图优化流程

`torch.compile` 的优化流程以 FX Graph 作为统一的中间表示（IR），整体由 TorchDynamo（图捕获）、AOTAutograd（自动微分图分解）与 TorchInductor（图优化与代码生成）等组件协同完成，如图 8.1 所示。

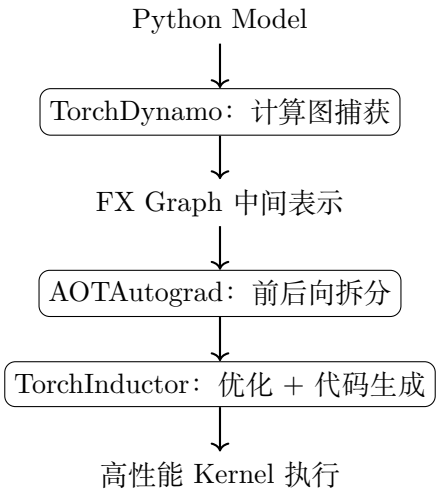


图 8.1: PyTorch 计算图优化组件的协作流程

- **TorchDynamo（捕获）**：负责在运行时捕获 Python 程序中的张量操作，将动态图转换为静态计算图（FX Graph）。其目标是在不改变用户代码的前提下，实现图级分析与优化的入口。
- **FX Graph（中间表示）**：作为统一的中间表示（IR），描述模型的计算结构。该阶段支持符号追踪（symbolic tracing）以及图变换（graph rewrite），为后续优化提供基础。
- **AOTAutograd（前后向拆分）**：将自动求导过程提前（ahead-of-time）展开，把前向计算与反向梯度计算分离成两个独立的计算图，从而提升优化空间与执行效率。
- **TorchInductor（优化 + 代码生成）**：作为 PyTorch 编译器后端，对计算图进行多层级优化，并基于中间表示进行调度，最终生成高性能后端代码（GPU 上生成 Triton kernel）。

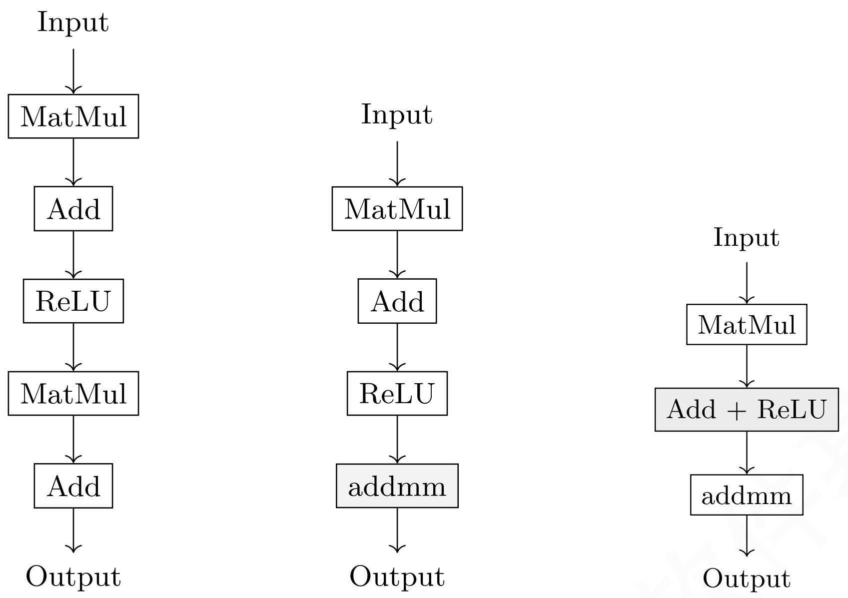
为了分析编译过程中的图结构与优化细节，可以通过开启 `TORCH_COMPILE_DEBUG` 环境变量来导出中间表示与生成代码。

```
TORCH_COMPILE_DEBUG=1 python mlp.py
```

一般会生成如下文件结构，其中 `forward` 与 `backward` 目录分别对应前向计算图与反向梯度计算图的编译结果，涵盖从高层计算图表示到低层执行代码的多个阶段。其中，`fx_graph_readable.py` 表示优化前的 FX Graph，`fx_graph_transformed.py` 表示经过图变换优化后的计算图结构，而 `output_code.py` 则对应算子融合后生成的高性能执行代码。

```
|-- aot_model___0_debug.log              # AOTAutograd 调试日志
|-- model__0_forward_1.0/                # 前向计算图编译结果
|   |-- fx_graph_readable.py             # FX Graph（前向）
|   |-- fx_graph_transformed.py          # 优化后的 FX Graph
|   |-- ir_pre_fusion.txt                # fusion 前 IR 表示
|   |-- ir_post_fusion.txt               # fusion 后 IR 表示
|   |-- output_code.py                   # 生成的 kernel 代码
`-- model__0_backward_3.1/               # 反向计算图编译结果
    |-- fx_graph_readable.py             # FX Graph（反向）
    |-- fx_graph_transformed.py          # 优化后的 FX Graph
    |-- ir_pre_fusion.txt                # fusion 前 IR 表示
    |-- ir_post_fusion.txt               # fusion 后 IR 表示
    `-- output_code.py                   # 生成的 kernel 代码
```

8.2.3 forward 优化分析



(a) 原始图　(b) 图优化　(c) Triton 算子融合

图 8.2: 前向计算图优化过程

优化前的 FX Graph 代码 8.4展示了 `fx_graph_readable.py` 文件的内容。该计算图的返回值列表 `[add_1, relu, permute_1, permute_2]` 为后续反向传播与编译优化提供了必要的计算上下文。各返回值的具体作用如下表所示：

表 8.2: MLP 前向图返回值在梯度计算中的作用

返回值	形状	梯度计算	作用说明
`add_1`	$[128, 1]$	$\frac{\partial \mathcal{L}}{\partial \texttt{add\_1}}$	模型最终输出张量，用于计算损失函数 $\mathcal{L}(\texttt{add\_1}, y)$。
`relu`	$[128, 64]$	$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2}$	隐藏层输出，用于计算第二层权重梯度：$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \texttt{relu}^\top \cdot \frac{\partial \mathcal{L}}{\partial \texttt{add\_1}}$
`permute_1`	$[1, 64]$	$\frac{\partial \mathcal{L}}{\partial \texttt{relu}}$	第二层权重转置 $\mathbf{W}_2^\top$，用于梯度回传：$\frac{\partial \mathcal{L}}{\partial \texttt{relu}} = \frac{\partial \mathcal{L}}{\partial \texttt{add\_1}} \cdot \texttt{permute\_1}$
`permute_2`	$[1, 128]$	$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}$	输入张量转置，用于计算第一层权重梯度：$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \texttt{permute\_2} \cdot \frac{\partial \mathcal{L}}{\partial \texttt{relu}}$

```python
class GraphModule(torch.nn.Module):
    def forward(self,
        primals_1: "f32[1, 64]",    # weight1: 第一层权重 (in_features=1 -> hidden=64)
        primals_2: "f32[64]",       # bias1:   第一层偏置
        primals_3: "f32[64, 1]",    # weight2: 第二层权重 (hidden=64 -> out_features=1)
        primals_4: "f32[1]",        # bias2:   第二层偏置
        primals_5: "f32[128, 1]"    # input: 输入数据 (batch_size=128, features=1)
    ):
        # ---------------- 第一层: Linear + ReLU ----------------
        # 对应 mlp.py:21: x = x @ self.weight1 + self.bias1   # (B, H)
        # 矩阵乘法: (128,1) @ (1,64) -> (128,64)
        mm: "f32[128, 64]" = torch.ops.aten.mm.default(primals_5, primals_1)
        primals_1 = None  # FX 内存优化标记: 显式置空, 提示编译器/运行时可提前释放该张量显存
        # 加法 (广播) : (128,64) + (64,) -> (128,64)
        add: "f32[128, 64]" = torch.ops.aten.add.Tensor(mm, primals_2)
        mm = primals_2 = None  # 释放中间变量, 降低峰值内存占用
        # 对应 mlp.py:22: x = torch.relu(x)
        relu: "f32[128, 64]" = torch.ops.aten.relu.default(add)
        add = None

        # ---------------- 第二层: Linear ----------------
        # 对应 mlp.py:23: x = x @ self.weight2 + self.bias2   # (B, 1)
        # 矩阵乘法: (128,64) @ (64,1) -> (128,1)
        mm_1: "f32[128, 1]" = torch.ops.aten.mm.default(relu, primals_3)
        # 加法 (广播) : (128,1) + (1,) -> (128,1)
        add_1: "f32[128, 1]" = torch.ops.aten.add.Tensor(mm_1, primals_4)
        mm_1 = primals_4 = None

        # ---------------- 用于梯度计算 ----------------
        permute_1: "f32[1, 64]" = torch.ops.aten.permute.default(primals_3, [1, 0])  # weight2.T
        primals_3 = None
        permute_2: "f32[1, 128]" = torch.ops.aten.permute.default(primals_5, [1, 0])  # input_x.T
        primals_5 = None

        return [add_1, relu, permute_1, permute_2]
```

代码 8.4: MLP 模型优化前的前向 FX Graph（fx_graph_readable.py 文件）

优化后的 FX Graph 代码 8.5展示了计算图优化后的结果。

```python
class GraphModule(torch.nn.Module):
    def forward(self, ... ):
        ...
        # ---------------- 第二层: bias + (relu @ weight2) ----------------
        addmm_default: "f32[32, 1]" = torch.ops.aten.addmm.default(
            primals_4,    # bias2
            relu,         # input
            primals_3     # weight2
        )
       ...
```

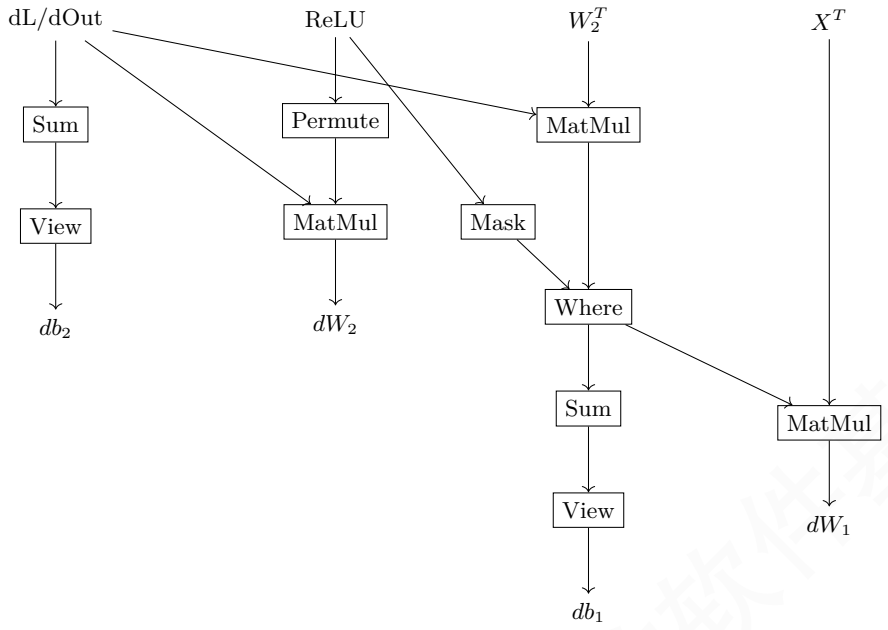
代码 8.5: MLP forward 的 FX Graph 优化结果（fx_graph_transformed.py 文件）

算子融合结果 代码 8.6展示了最终算子融合后的执行结果。TorchInductor 在该阶段完成了从 FX Graph 到 Triton Kernel 的 lowering，并通过 kernel fusion 将 `add + relu` 的融合。

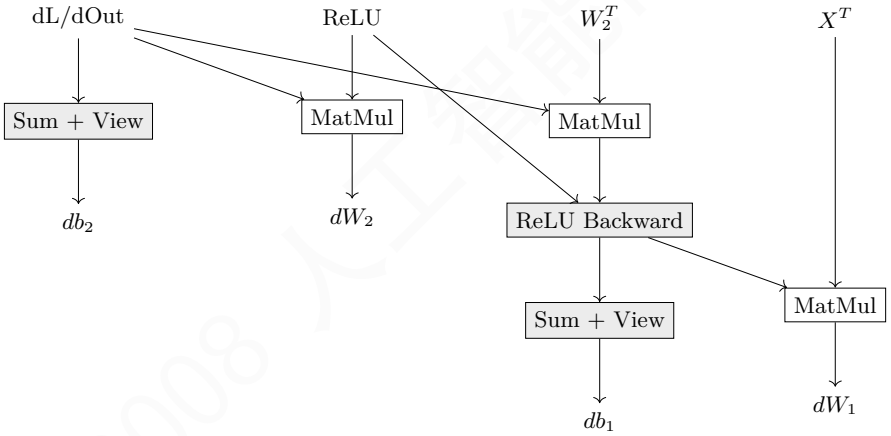
```
@triton_heuristics.pointwise(
    size_hints=[8192],
    triton_meta={'kernel_name': 'triton_poi_fused_add_relu_0', ...}
)
@triton.jit
def triton_(in_out_ptr0, in_ptr0, xnumel, XBLOCK: tl.constexpr):
    xnumel = 8192
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x2 = xindex
    x0 = xindex % 64
    tmp0 = tl.load(in_out_ptr0 + (x2), None)
    tmp1 = tl.load(in_ptr0 + (x0), None, eviction_policy='evict_last')
    # ------------------ 融合执行: add + relu ------------------
    tmp2 = tmp0 + tmp1 # add
    tmp3 = tl.full([1], 0, tl.int32)
    tmp4 = triton_helpers.maximum(tmp3, tmp2) # ReLU: max(0, x)

    tl.store(in_out_ptr0 + (x2), tmp4, None)

def call(args):
    primals_1, primals_2, primals_3, primals_4, primals_5 = args
    with torch.cuda._DeviceGuard(0):
        torch.cuda.set_device(0)
        buf0 = empty_strided_cuda((128, 64), (64, 1), torch.float32)
        extern_kernels.mm(primals_5, primals_1, out=buf0)
        del primals_1
        buf1 = buf0
        del buf0

        # ------------------ 使用融合后的Triton算子: add + relu ------------------
        stream0 = get_raw_stream(0)
        triton_poi_fused_add_relu_0.run(
            buf1,         # in-out tensor
            primals_2,    # bias
            8192,         # total elements
            grid=grid(8192),
            stream=stream0
        )
        del primals_2

        buf3 = empty_strided_cuda((128, 1), (1, 1), torch.float32)
        extern_kernels.addmm(primals_4, buf1, primals_3, alpha=1, beta=1, out=buf3)
        del primals_4

    return (buf3, buf1, ...)
```

代码 8.6: MLP forward 的最终优化结果（`output_code.py` 文件）

8.2.4 backward 优化分析



(a) 原始 Backward FX 图



(b) Triton 反向算子融合

图 8.3: 反向计算图优化过程

优化前的 FX Graph 代码 8.7是反向传播的 FX Graph，其结构直接对应梯度计算过程。

```
class GraphModule(torch.nn.Module):
    def forward(self,
        relu: "f32[128, 64]",          # forward中间激活值（ReLU前输入）
        permute_1: "f32[1, 64]",       # W2^T（第二层权重转置）
        permute_2: "f32[1, 128]",      # X^T（第一层输入转置）
        tangents_1: "f32[128, 1]"      # dL/dOutput（loss对输出梯度）
    ):
        # ------------------ dL/db2 = sum_batch ------------------
        # bias梯度：对batch维度求和（128 → 1）
        sum_1: "f32[1, 1]" = torch.ops.aten.sum.dim_IntList(tangents_1, [0], True)
        view: "f32[1]" = torch.ops.aten.view.default(sum_1, [1])
        sum_1 = None

        # ------------------ dL/dW2 = relu^T @ dL/dout ------------------
        # 第二层权重梯度（GEMM）
        permute: "f32[64, 128]" = torch.ops.aten.permute.default(relu, [1, 0])
        mm_2: "f32[64, 1]" = torch.ops.aten.mm.default(permute, tangents_1)
        permute = None

        # ------------------ dL/drelu = dL/dout @ W2^T ------------------
        mm_3: "f32[128, 64]" = torch.ops.aten.mm.default(tangents_1, permute_1)
        tangents_1 = permute_1 = None

        # ------------------ dL/dx = dL/drelu * 1(relu > 0) ------------------
        # ReLU backward（逐元素mask）
        le: "b8[128, 64]" = torch.ops.aten.le.Scalar(relu, 0)
        full_default: "f32[]" = torch.ops.aten.full.default([], 0.0,
            dtype=torch.float32,
            layout=torch.strided,
            device=torch.device('cuda', 0),
            pin_memory=False
        )
        where: "f32[128, 64]" = torch.ops.aten.where.self(le, full_default, mm_3)
        le = full_default = mm_3 = None

        # ------------------ dL/db1 = sum_batch ------------------
        # 第一层bias梯度（128 → 1）
        sum_2: "f32[1, 64]" = torch.ops.aten.sum.dim_IntList(where, [0], True)
        view_1: "f32[64]" = torch.ops.aten.view.default(sum_2, [64])
        sum_2 = None

        # ------------------ dL/dW1 = X^T @ dL/dx ------------------
        # 第一层权重梯度（GEMM）
        mm_4: "f32[1, 64]" = torch.ops.aten.mm.default(permute_2, where)
        permute_2 = where = None

        return [mm_4, view_1, mm_2, view, None]
```

代码 8.7: MLP backward 的 FX Graph

算子融合结果 代码 8.8展示了 TorchInductor 在反向传播阶段的算子融合与 kernel lowering 结果。系统将 FX Graph 中的算子组合映射为多个 Triton kernel，实现计算图级别优化。

```
@triton_heuristics.persistent_reduction(size_hints=[1, 32],
    triton_meta={'kernel_name': 'triton_per_fused_sum_0', ...})
...
@triton_heuristics.pointwise(size_hints=[2048],
    triton_meta={'kernel_name': 'triton_poi_fused_threshold_backward_1', ...})
...
@triton_heuristics.reduction(size_hints=[64, 32],
    triton_meta={'kernel_name': 'triton_red_fused_sum_2', ...})
...
def call(args):
    relu, permute_1, permute_2, tangents_1 = args
    ...
    with torch.cuda._DeviceGuard(0):
        torch.cuda.set_device(0)
        # ------------------ dL/db2 = sum_batch ------------------
        buf0 = empty_strided_cuda((1, 1), (1, 1), torch.float32)
        stream0 = get_raw_stream(0)
        # 算子融合1: sum_1 + view
        triton_per_fused_sum_0.run(tangents_1, buf0, 1, 128, grid=grid(1), stream=stream0)

        # ------------------ dL/dW2 = relu^T @ dL/dout ------------------
        buf1 = empty_strided_cuda((64, 1), (1, 1), torch.float32)
        extern_kernels.mm(reinterpret_tensor(relu, (64, 128), (1, 64), 0), tangents_1, out=buf1)

        # ------------------ dL/drelu = dL/dout @ W2^T ------------------
        buf2 = empty_strided_cuda((128, 64), (64, 1), torch.float32)
        extern_kernels.mm(tangents_1, permute_1, out=buf2)
        del permute_1; del tangents_1

        # reuse buffer for ReLU backward
        buf3 = buf2; del buf2

        # ------------------ dL/dx = dL/drelu * 1(relu > 0) ------------------
        # 算子融合2: where + mask
        triton_poi_fused_threshold_backward_1.run(buf3, relu, 8192, grid=grid(8192), stream=stream0)
        del relu

        # ------------------ dL/db1 = sum_batch ------------------
        buf4 = empty_strided_cuda((1, 64), (64, 1), torch.float32)
        # 算子融合23: sum_2 + view
        triton_red_fused_sum_2.run(buf3, buf4, 64, 128, grid=grid(64), stream=stream0)

        # ------------------ dL/dW1 = X^T @ dL/dx ------------------
        buf5 = empty_strided_cuda((1, 64), (64, 1), torch.float32)
        extern_kernels.mm(permute_2, buf3, out=buf5)
        del buf3; del permute_2

    return (buf5, ...)
```

代码 8.8: MLP backward 的优化结果

8.3 性能分析工具

理想情况下，能够把算力用满，不因算子调度和访存影响性能最理想。在 Linux 环境下，可以使用 nVidia 提供的 profiling 工具获取程序运行的统计信息，从而分析性能瓶颈。常用工具是 nsys，可以统计 CPU/GPU 时间开销占比、kernel 调度等信息。

安装方法 上述工具通常随 CUDA Toolkit 一同安装。

```
sudo apt install nvidia-cuda-toolkit
```

使用 nsys

```
nsys profile python mlp.py
```

执行后会生成 `.nsys-rep` 文件，可通过以下命令查看摘要：

```
nsys stats report.qdrep
```

```
 ** OS Runtime Summary (osrtsum):
Time(%)   Total(ns)     Calls   Avg(ns)     Name
-------------------------------------------------------
73.6      6.55e10       1035    6.32e7      pthread_cond_wait
 8.3      7.37e9          77    9.57e7      poll
 6.3      5.61e9        5527    1.02e6      read
 5.9      5.27e9          85    6.20e7      pthread_cond_timedwait
 3.0      2.66e9          13    2.05e8      sem_wait
...
 ** CUDA API Summary (cudaapisum):
Time(%)   Total(ns)    Calls   Avg(ns)    Name
-------------------------------------------------------
84.7      1.88e8       11010   1.71e4     cudaLaunchKernel
 6.3      1.40e7        2000   7.00e3     cuLaunchKernel
 5.2      1.16e7           3   3.86e6     cudaFree
 0.4      8.50e5          65   1.31e4     cudaMemcpyAsync
...
 ** CUDA GPU Kernel Summary (gpukernsum):
Time(%)   Total(ns)   Inst    Avg(ns)    Kernel
-------------------------------------------------------
 7.9      3.08e6      1000    3.07e3     gemvx
 7.0      2.72e6      1000    2.72e3     gemm (cublas)
 6.3      2.43e6       500    4.86e3     reduce_kernel
 6.1      2.35e6      1000    2.35e3     triton_
 6.0      2.31e6      1000    2.31e3     elementwise
 5.8      2.26e6       500    4.53e3     multi_tensor_apply
 5.6      2.17e6      1009    2.15e3     fill_kernel
...
```

代码 8.9: nsys 统计结果

从上述统计结果可以观察到，`cudaLaunchKernel` 调用占据了绝大部分 CUDA API 时间（84.7%），每次调用的平均开销约为 1.7×10^4 ns（约 17 μs）。从 GPU Kernel 统计可以看出，大量 kernel 的执行时间仅为 $2 \sim 5\mu$s。这类短时、高频的 kernel 反映出 GPU 计算粒度过细，单个 kernel 的计算时间甚至低于其调度开销，很可能导致 GPU 计算资源未能得到充分利用。

练习

1) 对下列 MLP 模型应用 `torch.compile` 进行编译优化，并分析其性能表现。

- 代码对比：分别给出优化前与优化后的模型实现，对比其结构与调用方式的差异；
- 性能测试：在不同 batch size（如 128、1024、4096）下，测量模型的运行时间或吞吐量，对比优化前后的性能变化，并分析其趋势；
- 性能分析：使用 `nsys` 对程序执行过程进行 profiling，尝试分析为什么在小 batch size 下优化效果可能不明显甚至下降。

```python
class MLP(nn.Module):
    def __init__(self, feature=1, hidden=1024):
        super().__init__()
        self.w1 = nn.Parameter(torch.randn(feature, hidden))
        self.b1 = nn.Parameter(torch.randn(hidden))
        self.w2 = nn.Parameter(torch.randn(hidden, hidden))
        self.b2 = nn.Parameter(torch.randn(hidden))
        self.w3 = nn.Parameter(torch.randn(hidden, 1))
        self.b3 = nn.Parameter(torch.randn(1))

    def forward(self, x):
        x = x @ self.w1 + self.b1
        x = torch.relu(x)
        x = x @ self.w2 + self.b2
        x = torch.relu(x)
        x = x @ self.w3 + self.b3
        return x
```

代码 8.10: MLP 模型：使用 `nn.Linear` 替换矩阵乘法和加法

2) 对 LeNet-5 模型应用 `torch.compile` 进行编译优化，对比优化前后的代码和运行性能。