HW3

何重阳　学号：2025211210

2026 年 1 月 6 日

目录

1 算子实现

1.1 Add

1.1.1 Add 算子算法设计

Add 算子是基于 Triton 框架开发的 GPU 并行化逐元素加法算子，核心功能为完成两个任意形状张量的逐元素加法运算（$output = input + other$）。

1.1.2 核心代码实现

Add 算子的核心代码如下：

```
import triton
import triton.language as tl

@triton.jit
def add_kernel(
input_ptr, other_ptr, output_ptr,
n_elements,
BLOCK_SIZE: tl.constexpr,  # 编译期常量：控制并行块大小
):
# 1. 解析程序ID，计算当前块的起始位置
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    # 2. 生成当前块内的元素偏移量
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # 3. 生成掩码：避免访问超出张量长度的元素
    mask = offsets < n_elements

    # 4. 加载输入数据（全局内存 → GPU共享内存）
    input = tl.load(input_ptr + offsets, mask=mask)
    other = tl.load(other_ptr + offsets, mask=mask)
    # 5. 核心计算：逐元素加法
    output = input + other
    # 6. 存储结果到输出张量
    tl.store(output_ptr + offsets, output, mask=mask)


```

Listing 1: 基于 Triton 的 Add 算子核心实现代码

1.2 Linear

1.2.1 Linear 算子算法设计

本文基于 Triton 框架实现了适配大/小矩阵场景的 Linear 算子，核心完成全连接层计算 $Y = X \cdot W^T + b$（$X \in \mathbb{R}^{M \times K}$，$W \in \mathbb{R}^{N \times K}$，$b \in \mathbb{R}^N$）。

算子核心执行流程如下：

1. 解析输入维度 M, N, K；
2. 计算网格维度 $\text{grid} = \lceil M/\text{BLOCK}_M \rceil \times \lceil N/\text{BLOCK}_N \rceil$，每个网格对应一个输出子块；
3. 核函数内解析程序 ID（pid），划分行/列块索引，计算块偏移量；
4. K 维度分块循环：加载输入/权重子块，按场景执行矩阵乘法并累加；
5.（可选）加载偏置并广播到累加结果，掩码存储最终结果到输出矩阵。

1.2.2 核心代码实现

关键逻辑如下：

```
        @triton.jit
        def linear_kernel_large(
        input_ptr, weight_ptr, bias_ptr, output_ptr,
        M, N, K,  # 维度参数
        # 步长/块大小等constexpr参数（省略）
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr,
    BLOCK_SIZE_K: tl.constexpr,
        ):
        # 1. 块索引与偏移计算（核心流程）
        pid = tl.program_id(axis=0)
        num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
        pid_m = pid // num_pid_n  # 行块索引
        pid_n = pid % num_pid_n   # 列块索引
        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)

        # 2. 累加器初始化
        accumulator = tl.zeros([BLOCK_SIZE_M, BLOCK_SIZE_N], dtype=tl.
    float32)

```

```
        # 3. K维度分块循环（核心：tl.dot矩阵乘法）
        for k in range(0, K, BLOCK_SIZE_K):
            k_offs = k + offs_k
            # 越界掩码（保证内存安全）
            a_mask = (offs_m[:, None] < M) & (k_offs[None, :] < K)
            b_mask = (offs_n[None, :] < N) & (k_offs[:, None] < K)

            # 加载子块并转换精度
            a = tl.load(input_ptr + offs_m[:, None]*stride_im + k_offs[None, :]*stride_in, mask=a_mask, other=0.0)
            b = tl.load(weight_ptr + offs_n[None, :]*stride_wn + k_offs[:, None]*stride_wk, mask=b_mask, other=0.0)
            a_f32, b_f32 = a.to(tl.float32), b.to(tl.float32)

            # ===== 核心：tl.dot实现子块矩阵乘法 =====
            accumulator += tl.dot(a_f32, b_f32, allow_tf32=False)

        # 偏置添加+结果存储（省略重复代码）
        if bias_ptr is not None:
            bias = tl.load(bias_ptr + offs_n, mask=offs_n < N, other=0.0)
            accumulator += bias[None, :]
        tl.store(output_ptr + offs_m[:, None]*stride_om + offs_n[None, :]*stride_on, accumulator, mask=(offs_m[:, None]<M) & (offs_n[None, :]<N))

```

Listing 2: Linear 算子核心逻辑

1.2.3 linear.py 运行结果

```
root@autodl-container-fcb946b0f6-6ca1dbb1:~/autodl-tmp/大模型计算_HW3/llaisys-thu# bash test_ops.sh
Testing Ops.linear on nvidia
   out (2, 3), x (2, 4), w (3, 4), bias True, dtype <f32>
   out (2, 3), x (2, 4), w (3, 4), bias True, dtype <f16>
   out (2, 3), x (2, 4), w (3, 4), bias True, dtype <bf16>
   out (512, 4096), x (512, 4096), w (4096, 4096), bias True, dtype <f32>
   out (512, 4096), x (512, 4096), w (4096, 4096), bias True, dtype <f16>
   out (512, 4096), x (512, 4096), w (4096, 4096), bias True, dtype <bf16>
Test passed!
```

图 1: Linear 算子运行结果

1.3 Argmax

1.3.1 Argmax 算子算法设计

该算子根据输入规模自动选择不同的执行路径：当张量长度较小时，使用单个 block 在一次 kernel 调用中完成最大值及其索引的计算；当张量规模较大时，则在 kernel 内部采用分块迭代的方式顺序扫描输入数据，从而避免多 kernel 启动和中间归约缓冲区带来的额外开销。

在实现层面，算子充分利用 Triton 提供的向量化加载与归约原语，通过 `tl.max` 计算局部最大值，并结合布尔掩码与 `tl.min` 操作获取最大值的首个出现位置。为保证数值稳定性，所有参与比较的中间结果统一提升至 `float32` 精度；同时，在 masked load 场景下使用负无穷作为填充值，以避免越界元素对归约结果产生干扰。该设计在保持实现简洁性的同时，实现了良好的性能与可扩展性，并能够与 LLAISys 张量表示无缝集成。

1.3.2 核心代码实现

```
        @triton.jit
        def argmax_kernel_single(
        max_idx_ptr,      # 输出：最大值索引
        max_val_ptr,      # 输出：最大值
        vals_ptr,         # 输入：数据指针
        n_elements,       # 输入：元素个数
        BLOCK_SIZE: tl.constexpr,
        ):
            # 生成 block 内偏移
            offsets = tl.arange(0, BLOCK_SIZE)
            mask = offsets < n_elements

            # 向量化加载数据，越界位置填充为 -inf
            vals = tl.load(vals_ptr + offsets, mask=mask, other=float('-
    inf'))

            # block 内归约求最大值（返回 float32）
            local_max = tl.max(vals, axis=0)

            # 查找最大值的首个出现位置
            is_max = (vals == local_max) & mask
            local_idx = tl.where(is_max, offsets, n_elements)
            local_idx = tl.min(local_idx, axis=0)

```

```
            # 写回结果
            tl.store(max_val_ptr, local_max)
            tl.store(max_idx_ptr, local_idx)

```

Listing 3: Triton Argmax Kernel 核心实现

1.3.3 argmax.py 运行结果

图 2: argmax 算子运行结果

1.4 Embedding

1.4.1 Embedding 算子算法设计

该 Triton embedding kernel 采用按序列分块的并行策略，将索引到向量的查表操作显式展开为内存加载与存储过程。每个 program 负责一个连续的 token block，并对词表索引进行合法性检查后完成对应嵌入向量的拷贝，从而实现对变长输入的安全支持。

1.4.2 核心代码实现

```
        @triton.jit
        def embedding_kernel(
        weight_ptr, index_ptr, output_ptr,
        vocab_size, hidden_size, seq_len,
        BLOCK_SIZE: tl.constexpr,
        ):
        # program id, 对应序列维度上的一个 block
        pid = tl.program_id(axis=0)
        block_start = pid * BLOCK_SIZE

        # block 内的 token 偏移
        offsets = block_start + tl.arange(0, BLOCK_SIZE)
        mask = offsets < seq_len

```

```
        # 加载索引（越界位置填充为 0）
        indices = tl.load(index_ptr + offsets, mask=mask, other=0)

        # 逐 token 处理 embedding 查表
        for i in range(BLOCK_SIZE):
            if i + block_start < seq_len:
            # 读取当前 token 的词表索引
            idx = tl.load(index_ptr + block_start + i)

            # 索引合法性检查
            if idx >= 0 and idx < vocab_size:
            weight_offset = idx * hidden_size
            output_offset = (block_start + i) * hidden_size

            # 将对应 embedding 向量逐元素拷贝到输出
            for j in range(hidden_size):
                val = tl.load(weight_ptr + weight_offset + j)
                tl.store(output_ptr + output_offset + j, val)

```

Listing 4: Triton Embedding Kernel 核心实现

1.4.3 embedding.py 运行结果

```
root@autodl-container-fcb946b0f6-6ca1dbb1:~/autodl-tmp/大模型计算_HW3/llaisys-thu# bash test_ops.sh
Testing Ops.embedding on nvidia
   idx_shape (1,) embd_shape (2, 3) dtype <f32>
   idx_shape (1,) embd_shape (2, 3) dtype <f16>
   idx_shape (1,) embd_shape (2, 3) dtype <bf16>
   idx_shape (50,) embd_shape (512, 4096) dtype <f32>
   idx_shape (50,) embd_shape (512, 4096) dtype <f16>
   idx_shape (50,) embd_shape (512, 4096) dtype <bf16>
Test passed!
```

图 3: embedding 算子运行结果

1.5 Rope

1.5.1 Rope 算子算法设计

RoPE 算子以 (head, position) 作为并行粒度，每个 Triton program 负责处理一个注意力头在某一序列位置上的旋转计算。特征维度 d 被等分为两部分，对应旋转变换中的实部与虚部。对于每一对特征分量 (a, b)，算子根据位置索引 p 和频率参数 θ 计算旋转角度，并

执行如下变换：

$$a' = a\cos(\omega_p) - b\sin(\omega_p),$$

$$b' = b\cos(\omega_p) + a\sin(\omega_p),$$

其中频率 ω_p 由位置索引与维度相关的幂函数共同确定。

1.5.2 核心代码实现

```
        @triton.jit
        def rope_kernel(
        input_ptr, pos_ids_ptr, output_ptr,
        seq_len, n_head, d, theta,
        stride_im, stride_in, stride_ih,
        stride_om, stride_on, stride_oh,
        BLOCK_SIZE: tl.constexpr,
        ):
        # 每个 program 对应一个 (head, position)
        pid = tl.program_id(axis=0)
        head_id = pid // seq_len
        pos = pid % seq_len

        # 越界检查
        if head_id >= n_head or pos >= seq_len:
        return

        # 读取位置索引
        pos_id = tl.load(pos_ids_ptr + pos)
        d_half = d // 2

        # 按 block 遍历特征维度
        for i in range(0, d_half, BLOCK_SIZE):
        idx = i + tl.arange(0, BLOCK_SIZE)
        mask = idx < d_half

        # 实部与虚部索引
        a_idx = idx
        b_idx = idx + d_half

        # 计算输入偏移
```

```
32          a_offset = pos * stride_im + head_id * stride_in + a_idx *
    stride_ih
33          b_offset = pos * stride_im + head_id * stride_in + b_idx *
    stride_ih
34
35          # 加载输入
36          a = tl.load(input_ptr + a_offset, mask=mask, other=0.0)
37          b = tl.load(input_ptr + b_offset, mask=mask, other=0.0)
38
39          # 转换为 float32 进行旋转计算
40          a = a.to(tl.float32)
41          b = b.to(tl.float32)
42
43          # 使用 float64 计算高精度频率
44          pos_id_f64 = pos_id.to(tl.float64)
45          idx_f64 = idx.to(tl.float64)
46          exponent = 2.0 * idx_f64 / d
47
48          # theta^exponent = 2^(exponent * log2(theta))
49          theta_f64 = theta + 0.0
50          log2_theta = tl.log2(theta_f64).to(tl.float64)
51          log2_power = exponent * log2_theta
52          theta_power = tl.exp2(log2_power)
53
54          freqs_f64 = pos_id_f64 / theta_power
55          angle = freqs_f64.to(tl.float32)
56
57          cos_val = tl.cos(angle)
58          sin_val = tl.sin(angle)
59
60          # 应用 RoPE 旋转变换
61          a_out = a * cos_val - b * sin_val
62          b_out = b * cos_val + a * sin_val
63
64          # 输出偏移
65          a_out_offset = pos * stride_om + head_id * stride_on + a_idx *
    stride_oh
66          b_out_offset = pos * stride_om + head_id * stride_on + b_idx *
    stride_oh
```

```
67
68          # 写回结果
69          tl.store(output_ptr + a_out_offset, a_out, mask=mask)
70          tl.store(output_ptr + b_out_offset, b_out, mask=mask)
71
```

Listing 5: Triton RoPE Kernel 核心实现

1.5.3 rope.py 运行结果

```
root@autodl-container-fcb946b0f6-6ca1dbb1:~/autodl-tmp/大模型计算_HW3/llaisys-thu# bash test_ops.sh
Testing Ops.rope on nvidia
   shape (2, 1, 4) range (0, 2) dtype <f32>
   shape (2, 1, 4) range (0, 2) dtype <f16>
   shape (2, 1, 4) range (0, 2) dtype <bf16>
   shape (512, 4, 4096) range (512, 1024) dtype <f32>
   shape (512, 4, 4096) range (512, 1024) dtype <f16>
   shape (512, 4, 4096) range (512, 1024) dtype <bf16>
Test passed!
```

图 4: rope 算子运行结果

1.6 self_attention

1.6.1 self_attention 算子算法设计

Key 和 Value 序列沿序列维度按 `BLOCK_SIZE_N` 分块，每个 block 依次参与与当前 Query block 的注意力计算。对于注意力分数的计算，当特征维度较大时优先使用 `tl.dot` 原语，并启用 IEEE 精度以提升数值稳定性；在特征维度较小时，则回退至显式逐元素乘加实现，以降低 kernel 启动与调度开销。

为支持自回归生成场景，算子在计算过程中同时引入边界掩码与因果（causal）掩码，确保每个 Query 位置只能关注其当前位置之前的 Key。掩码后的注意力分数被设置为负无穷，从而在 Softmax 计算中自然衰减为零权重。

在 Softmax 实现上，该算子采用 online softmax 技术，通过维护逐 block 更新的最大值 m_i 和归一化因子 d_i，避免显式存储完整注意力矩阵，有效降低显存占用并提升数值稳定性。所有中间统计量均在 `float32` 精度下进行更新，以平衡计算精度与性能。

1.6.2 核心代码实现

```
1           for kv_block_start in range(0, kv_len, BLOCK_SIZE_N):
2           # 加载 Key / Value block，并转换为 float32
3           k = tl.load(k_block_ptr, boundary_check=(0, 1)).to(tl.float32)
4           v = tl.load(v_block_ptr, boundary_check=(0, 1)).to(tl.float32)
5
6           # 计算注意力分数 QK^T
```

```
        # 当维度较大时，使用 tl.dot 以获得更好的数值精度
        if USE_DOT:
        attention_scores = tl.dot(q, k, input_precision="ieee") * scale
        else:
        q_expanded = q[:, :, None]
        k_expanded = k[None, :, :]
        product = q_expanded * k_expanded
        product = tl.where(q_mask[None, :, None], product, 0.0)
        attention_scores = tl.sum(product, axis=1) * scale

        # 构造边界掩码
        kv_idx = kv_block_start + tl.arange(0, BLOCK_SIZE_N)
        kv_mask = kv_idx < kv_len

        # 构造因果掩码（causal mask）
        causal_limits = (
        BLOCK_SIZE_M * pid_m
        + tl.arange(0, BLOCK_SIZE_M)
        + (kv_len - q_len)
        + 1
        )
        casual_mask = kv_idx[None, :] < causal_limits[:, None]

        mask_m = (BLOCK_SIZE_M * pid_m + tl.arange(0, BLOCK_SIZE_M)) <
    q_len

        # 合并所有掩码
        full_mask = mask_m[:, None] & kv_mask[None, :] & casual_mask

        # 被掩码位置设置为 -inf，使 Softmax 权重为 0
        masked_attention_scores = tl.where(
        full_mask, attention_scores, float("-inf")
        )

        # Online Softmax 更新（float32）
        m_i_new = tl.maximum(m_i, tl.max(masked_attention_scores, axis=1)
    )
        alpha = tl.exp2(log2_e * (m_i - m_i_new))
        exp_attention_scores = tl.exp2(
```

```
        log2_e * (masked_attention_scores - m_i_new[:, None])
        )
        d_i_k = tl.sum(exp_attention_scores, axis=1)

        # 计算加权 Value 聚合
        if USE_DOT and dv >= 16 and BLOCK_SIZE_M >= 16:
        o = tl.dot(exp_attention_scores, v, input_precision="ieee")
        else:
        exp_scores_expanded = exp_attention_scores[:, :, None]
        v_expanded = v[None, :, :]
        o = tl.sum(exp_scores_expanded * v_expanded, axis=1)

        # 更新累积输出
        acc = acc * alpha[:, None] + o

        # 更新 Softmax 统计量
        m_i = m_i_new
        d_i = d_i * alpha + d_i_k

        # 指针前移，处理下一个 KV block
        k_block_ptr = tl.advance(k_block_ptr, (0, BLOCK_SIZE_N))
        v_block_ptr = tl.advance(v_block_ptr, (BLOCK_SIZE_N, 0))

```

Listing 6: Triton Self-Attention Kernel 核心计算逻辑

1.6.3 self_attention.py 运行结果

```
root@autodl-container-fcb946b0f6-6ca1dbb1:~/autodl-tmp/大模型计算_HW3/llaisys-thu# bash test_ops.sh
Testing Ops.self_attention on nvidia
   qlen=2 kvlen=2 nh=1 nkvh=1 hd=4 dtype <f32>
   qlen=2 kvlen=2 nh=1 nkvh=1 hd=4 dtype <f16>
   qlen=2 kvlen=2 nh=1 nkvh=1 hd=4 dtype <bf16>
   qlen=5 kvlen=11 nh=4 nkvh=2 hd=8 dtype <f32>
   qlen=5 kvlen=11 nh=4 nkvh=2 hd=8 dtype <f16>
   qlen=5 kvlen=11 nh=4 nkvh=2 hd=8 dtype <bf16>
   qlen=1 kvlen=1 nh=4 nkvh=2 hd=8 dtype <f32>
   qlen=1 kvlen=1 nh=4 nkvh=2 hd=8 dtype <f16>
   qlen=1 kvlen=1 nh=4 nkvh=2 hd=8 dtype <bf16>
Test passed!
```

图 5: self_attention 算子运行结果

1.7 swiglu

1.7.1 swiglu 算子算法设计

SwiGLU 算子以序列维度为并行轴，每个 Triton program 负责处理一个 token 对应的一整行中间特征。特征维度（`intermediate_size`）在 kernel 内部按 `BLOCK_SIZE` 分块处理，从而在保证并行度的同时支持任意长度的中间特征维度。算子显式接收输入与输出张量的 stride 信息，能够正确处理非连续内存布局。

在数值计算方面，算子在 kernel 内部将输入提升至 `float32` 精度完成 Sigmoid 与乘法运算，以对齐 PyTorch 中 `gate.float()` 的计算语义并提升数值稳定性，最终再转换回原始数据类型进行存储。

1.7.2 核心代码实现

```
        @triton.jit
        def swiglu_kernel(
        gate_ptr, up_ptr, output_ptr,
        seq_len, intermediate_size,
        stride_gm, stride_gn,
        stride_um, stride_un,
        stride_om, stride_on,
        BLOCK_SIZE: tl.constexpr,
        ):
        # 每个 program 负责一个序列位置
        pid = tl.program_id(axis=0)
        row = pid

        # 当前 block 处理的列索引
        cols = tl.arange(0, BLOCK_SIZE)
        mask = cols < intermediate_size

        # 越界检查
        if row >= seq_len:
        return

        # 计算输入与输出的内存偏移
        gate_offset = row * stride_gm + cols * stride_gn
        up_offset = row * stride_um + cols * stride_un
        output_offset = row * stride_om + cols * stride_on

```

```
        # 加载 gate 与 up 分支数据
        gate = tl.load(gate_ptr + gate_offset, mask=mask, other=0.0)
        up = tl.load(up_ptr + up_offset, mask=mask, other=0.0)

        # 将输入转换为 float32 以提升数值精度
        gate_f32 = gate.to(tl.float32)
        up_f32 = up.to(tl.float32)

        # 计算 sigmoid(gate)
        sigmoid_gate = 1.0 / (1.0 + tl.exp(-gate_f32))

        # SwiGLU: up * gate * sigmoid(gate)
        result_f32 = up_f32 * gate_f32 * sigmoid_gate

        # 转回原始数据类型并写回输出
        output = result_f32.to(gate.dtype)
        tl.store(output_ptr + output_offset, output, mask=mask)

```

Listing 7: Triton SwiGLU Kernel 核心实现

1.7.3 swiglu.py 运行结果

图 6: swiglu 算子运行结果

1.8 rms_norm

1.8.1 rms_norm 算子算法设计

RMSNorm 算子以序列维度为并行轴，每个 Triton program 负责处理一个 token 对应的一整行特征向量。由于特征维度通常较大，算子在 kernel 内部采用两阶段（two-pass）计算策略：第一阶段以 block 为单位遍历特征维度，累积该行输入的平方和；第二阶段在得到归一化因子后，再次遍历特征维度完成归一化与缩放操作。该设计避免了额外的中间张量分配，同时能够支持任意长度的特征维度。

1.8.2 核心代码实现

```
        @triton.jit
        def rms_norm_kernel(
        input_ptr, weight_ptr, output_ptr,
        seq_len, hidden_size, eps,
        stride_im, stride_in,
        stride_om, stride_on,
        BLOCK_SIZE: tl.constexpr,
        ):
        # 每个 program 负责一个 token（一行特征）
        row = tl.program_id(axis=0)

        # -------- 第一阶段：累积平方和 --------
        sum_squared = 0.0
        for block_start in range(0, hidden_size, BLOCK_SIZE):
        cols = tl.arange(0, BLOCK_SIZE) + block_start
        mask = cols < hidden_size

        input_offsets = row * stride_im + cols * stride_in
        x = tl.load(input_ptr + input_offsets, mask=mask, other=0.0)

        x_squared = x * x
        sum_squared += tl.sum(x_squared, axis=0)

        # 计算均方根的倒数
        mean_squared = sum_squared / hidden_size
        mean_squared_eps = mean_squared + eps
        rsqrt_val = tl.rsqrt(mean_squared_eps)

        # -------- 第二阶段：归一化并缩放 --------
        for block_start in range(0, hidden_size, BLOCK_SIZE):
        cols = tl.arange(0, BLOCK_SIZE) + block_start
        mask = cols < hidden_size

        input_offsets = row * stride_im + cols * stride_in
        output_offsets = row * stride_om + cols * stride_on

        x = tl.load(input_ptr + input_offsets, mask=mask, other=0.0)
```

```
        weight = tl.load(weight_ptr + cols, mask=mask, other=0.0)

        # RMSNorm: x / sqrt(mean(x^2) + eps) * weight
        x_norm = x * rsqrt_val
        output = x_norm * weight

        tl.store(output_ptr + output_offsets, output, mask=mask)

```

Listing 8: Triton RMSNorm Kernel 核心实现

1.8.3 rms_norm.py 运行结果

图 7: rms_norm 算子运行结果

2 推理框架搭建

2.1 test_infer.py 运行结果展示

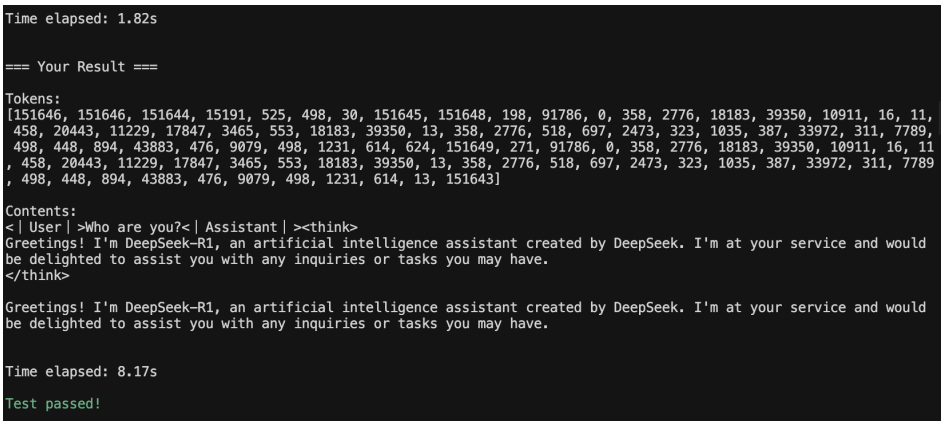
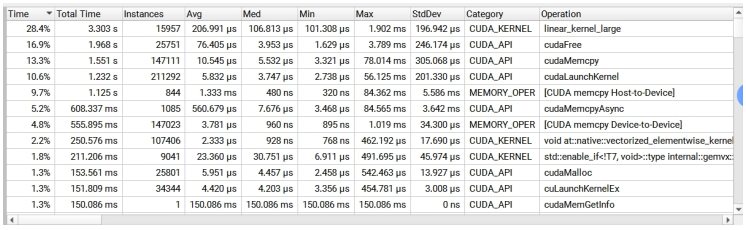


图 8: test_infer.py 运行结果

3 性能分析与算子优化

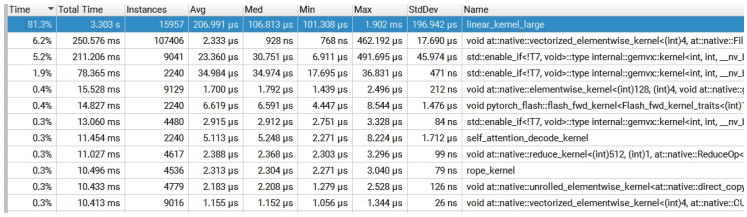
3.1 基础版分析

在未优化的版中，test_infer.py 的运行时间为 **8.17 s**。实验表明，linear_lernel_large、cudaFree、cudaMemcpy、以及 cudaLaunchKernel 等操作的耗时较长、时间占比较大。因此，可以针对这几个操作进行优化。



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
28.4%	3.303 s	15957	206.991 µs	106.813 µs	101.308 µs	1.902 ms	196.942 µs	CUDA_KERNEL	linear_kernel_large
16.9%	1.968 s	25751	76.405 µs	3.953 µs	1.629 µs	3.789 ms	246.174 µs	CUDA_API	cudaFree
13.3%	1.551 s	147111	10.545 µs	5.532 µs	3.321 µs	78.014 ms	305.068 µs	CUDA_API	cudaMemcpy
10.6%	1.232 s	211292	5.832 µs	3.747 µs	2.738 µs	56.125 ms	201.330 µs	CUDA_API	cudaLaunchKernel
9.7%	1.125 s	844	1.333 ms	480 ns	320 ns	84.362 ms	5.586 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device]
5.2%	608.337 ms	1085	560.679 µs	7.676 µs	3.468 µs	84.565 ms	3.642 ms	CUDA_API	cudaMemcpyAsync
4.8%	555.895 ms	147023	3.781 µs	960 ns	895 ns	1.019 ms	34.300 µs	MEMORY_OPER	[CUDA memcpy Device-to-Device]
2.2%	250.576 ms	107406	2.333 µs	928 ns	768 ns	462.192 µs	17.690 µs	CUDA_KERNEL	void at::native::vectorized_elementwise_kernel
1.8%	211.206 ms	9041	23.360 µs	30.751 µs	6.911 µs	491.695 µs	45.974 µs	CUDA_KERNEL	std::enable_if<!T7, void>::type internal::gemvx:
1.3%	153.561 ms	25801	5.951 µs	4.457 µs	2.458 µs	542.463 µs	13.927 µs	CUDA_API	cudaMalloc
1.3%	151.809 ms	34344	4.420 µs	4.203 µs	3.356 µs	454.781 µs	3.008 µs	CUDA_API	cuLaunchKernelEx
1.3%	150.086 ms	1	150.086 ms	150.086 ms	150.086 ms	150.086 ms	0 ns	CUDA_API	cudaMemGetInfo

(a) CUDA Summary(API_Kernels_MemOp)



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
81.3%	3.303 s	15957	206.991 µs	106.813 µs	101.308 µs	1.902 ms	196.942 µs	linear_kernel_large
6.2%	250.576 ms	107406	2.333 µs	928 ns	768 ns	462.192 µs	17.690 µs	void at::native::vectorized_elementwise_kernel<(int)4, at::native::Fil
5.2%	211.206 ms	9041	23.360 µs	30.751 µs	6.911 µs	491.695 µs	45.974 µs	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_l
1.9%	78.365 ms	2240	34.984 µs	34.974 µs	17.695 µs	36.831 µs	471 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_l
0.4%	15.528 ms	9129	1.700 µs	1.792 µs	1.439 µs	2.496 µs	212 ns	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::<
0.4%	14.827 ms	2240	6.619 µs	6.591 µs	4.447 µs	8.544 µs	1.476 µs	void pytorch_flash::flash_fwd_kernel<Flash_fwd_kernel_traits<(int)'
0.3%	13.060 ms	4480	2.915 µs	2.912 µs	2.751 µs	3.328 µs	84 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_l
0.3%	11.454 ms	2240	5.113 µs	5.248 µs	2.271 µs	8.224 µs	1.712 µs	self_attention_decode_kernel
0.3%	11.027 ms	4617	2.388 µs	2.368 µs	2.303 µs	3.296 µs	99 ns	void at::native::reduce_kernel<(int)512, (int)1, at::native::ReduceOp<
0.3%	10.496 ms	4536	2.313 µs	2.304 µs	2.271 µs	3.040 µs	79 ns	rope_kernel
0.3%	10.433 ms	4779	2.183 µs	2.208 µs	1.279 µs	2.528 µs	126 ns	void at::native::unrolled_elementwise_kernel<at::native::direct_copy
0.3%	10.413 ms	9016	1.155 µs	1.152 µs	1.056 µs	1.344 µs	26 ns	void at::native::vectorized_elementwise_kernel<(int)4, at::native::CU

(b) CUDA GPU Kernel Summary

图 9: test_infer.py 运行性能分析

3.2 改进 1

不足总结

仔细分析代码之后，我发现了以下不足：

- 设备内存频繁调用 `cudaMalloc/cudaFree`，无复用，导致 `cudaFree` 开销高。
- 大矩阵 kernel 配置一刀切：block 尺寸、TF32、swizzling、warp/stage 固定，可能与硬件或规模不匹配。
- L2 swizzling 在小规模时仍启用，增加额外开销。
- TF32 之前固定开关，无法按规模或需求切换。

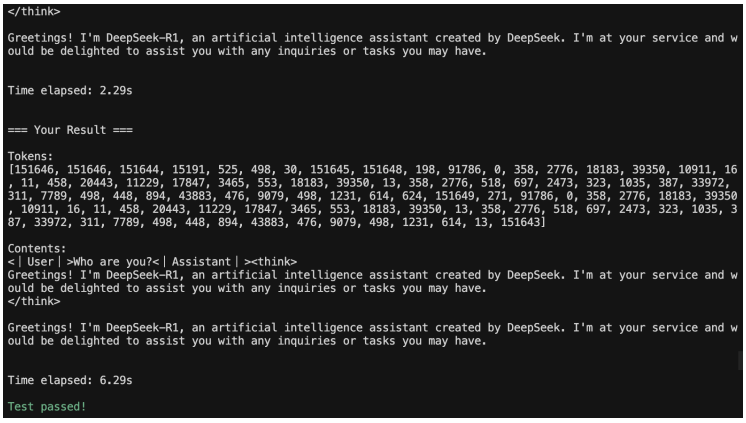
已完成改进

- 在 `src/device/nvidia/nvidia_runtime_api.cu` 增加设备内存缓存池（256B 对齐，池上限 512MB）：优先复用空闲块，未知指针或超限则直接 `cudaFree`，显著降低频繁分配释放的开销。

- 在 `python/llaisys/libllaisys/triton/kernels/linear.py` 为大 kernel 做自适应参数选择：根据规模与 K 选择 BM/BN/BK、group_size、num_warps、num_stages、TF32 开关；极小矩阵禁用 swizzling、减小 warp/stage，并默认关闭 TF32。
- L2 swizzling 仅在 block 数足够时启用，避免小规模的额外开销。
- 保留指针预计算与递增，减少循环内地址计算；TF32 改为 constexpr，可按需开启/关闭。

结果展示

优化之后，test_infer.py 的运行时间降低为 **6.29 s**，性能提升约 **25%**。经过优化，发现 linear_lernel_large 和 cudaFree 的时间都显著降低，降幅超过 **100%**，但是 cudaMemcpy、以及 cudaLaunchKerne 这两个操作的时间都略有上升。



(a) 运行结果

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
22.5%	1.871 s	147111	12.717 µs	6.070 µs	4.329 µs	89.076 ms	350.110 µs	CUDA_API	cudaMemcpy
17.5%	1.453 s	211292	6.878 µs	4.820 µs	2.865 µs	57.707 ms	206.184 µs	CUDA_API	cudaLaunchKernel
15.1%	1.255 s	15957	78.672 µs	48.064 µs	44.384 µs	589.887 µs	80.349 µs	CUDA_KERNEL	linear_kernel_large
14.6%	1.217 s	844	1.442 ms	480 ns	352 ns	88.825 ms	6.030 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device]
7.5%	624.585 ms	1085	575.654 µs	10.307 µs	3.534 µs	81.841 ms	3.715 ms	CUDA_API	cudaMemcpyAsync
6.8%	566.045 ms	147023	3.850 µs	1.024 µs	959 ns	1.020 ms	34.317 µs	MEMORY_OPER	[CUDA memcpy Device-to-Device]
3.2%	263.648 ms	107406	2.454 µs	992 ns	832 ns	463.551 µs	17.748 µs	CUDA_KERNEL	void at::native::vectorized_elementwise_kernel
2.5%	211.206 ms	9041	23.360 µs	30.752 µs	6.912 µs	491.456 µs	45.978 µs	CUDA_KERNEL	std::enable_if<!T7, void>::type internal::gemvx::
2.1%	171.711 ms	34344	4.999 µs	4.803 µs	3.330 µs	904.919 µs	5.202 µs	CUDA_API	cuLaunchKernelEx
1.8%	149.512 ms	1	149.512 ms	149.512 ms	149.512 ms	149.512 ms	0 ns	CUDA_API	cudaMemGetInfo
1.5%	125.396 ms	203	617.713 µs	586.306 µs	3.289 µs	2.480 ms	491.234 µs	CUDA_API	cudaFree
0.9%	78.326 ms	2240	34.967 µs	34.944 µs	17.472 µs	36.447 µs	469 ns	CUDA_KERNEL	std::enable_if<!T7, void>::type internal::gemvx::

(b) CUDA Summary(API_Kernels_MemOp)

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
61.8%	1.255 s	15957	78.672 µs	48.064 µs	44.384 µs	589.887 µs	80.349 µs	linear_kernel_large
13.0%	263.648 ms	107406	2.454 µs	992 ns	832 ns	463.551 µs	17.748 µs	void at::native::vectorized_elementwise_kernel<(int)4, at::native::FillFunc
10.4%	211.206 ms	9041	23.360 µs	30.752 µs	6.912 µs	491.456 µs	45.978 µs	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfloa
3.9%	78.326 ms	2240	34.967 µs	34.944 µs	17.472 µs	36.447 µs	469 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfloa
0.8%	15.529 ms	9129	1.701 µs	1.792 µs	1.440 µs	2.432 µs	212 ns	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_l
0.7%	14.862 ms	2240	6.635 µs	6.592 µs	4.480 µs	8.576 µs	1.481 µs	void pytorch_flash::flash_fwd_kernel<Flash_fwd_kernel_traits<(int)128, (
0.6%	13.042 ms	4480	2.911 µs	2.912 µs	2.752 µs	3.232 µs	84 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfloa
0.6%	12.322 ms	2240	5.501 µs	5.664 µs	2.432 µs	8.864 µs	1.849 µs	self_attention_decode_kernel
0.6%	11.289 ms	4536	2.488 µs	2.496 µs	2.432 µs	3.040 µs	57 ns	rope_kernel
0.5%	11.021 ms	4617	2.387 µs	2.368 µs	2.304 µs	3.360 µs	98 ns	void at::native::reduce_kernel<(int)512, (int)1, at::native::ReduceOp<float
0.5%	10.506 ms	9016	1.165 µs	1.152 µs	1.056 µs	1.280 µs	33 ns	void at::native::vectorized_elementwise_kernel<(int)4, at::native::CUDAF
0.5%	10.418 ms	4779	2.180 µs	2.208 µs	1.248 µs	2.496 µs	126 ns	void at::native::unrolled_elementwise_kernel<at::native::direct_copy_ker

(c) CUDA GPU Kernel Summary

图 10: test_infer.py 运行性能分析（第一次改进）

3.3 改进 2

针对上述分析，我做了进一步改进：

改进细节

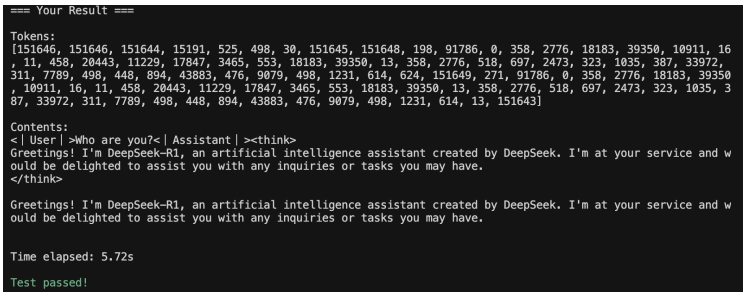
- **Rearrange kernel 优化**。对 rearrange kernel 进行了重构，使其基于真实的张量 stride 进行访问，而不再假设 stride 恒为 1，从而能够正确且高效地处理非连续内存布局。同时，将默认的 `BLOCK_SIZE` 从 1024 提升至 4096，并引入 `num_warps`=4 的 kernel 启动配置，以提高内存访问吞吐率和 warp 级并行度。

- **Add kernel 优化**。对 add kernel 采用与 rearrange kernel 相同的优化策略，将默认 `BLOCK_SIZE` 从 1024 提升至 4096，并配置 `num_warps`=4，以更充分地利用 GPU 计算资源并降低 kernel 启动与调度开销。

- **Tensor 转换逻辑优化**。在 `_llaisys_to_torch_tensor` 函数中，当目标张量为连续（contiguous）内存布局时，使用 `torch.empty` 替代 `torch.zeros` 进行张量分配，从而避免不必要的内存初始化操作，显著降低性能关键路径上的张量创建开销。

结果展示

经过优化，test_infer.py 的运行时间降低为 **5.72 s**。这一次，cudaMemcpy、以及 cudaLaunchKerne 这两个操作的时间都下降了，改进是有效果的。



(a) 运行结果

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
18.1%	1.317 s	147111	8.949 µs	6.781 µs	3.522 µs	18.760 ms	85.855 µs	CUDA_API	cudaMemcpy
17.2%	1.255 s	15957	78.636 µs	48.064 µs	44.352 µs	594.306 µs	80.349 µs	CUDA_KERNEL	linear_kernel_large
14.1%	1.026 s	117332	8.740 µs	4.538 µs	2.864 µs	71.695 ms	333.124 µs	CUDA_API	cudaLaunchKernel
10.7%	777.990 ms	844	921.789 µs	480 ns	352 ns	93.007 ms	4.612 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device]
9.3%	681.149 ms	1085	627.786 µs	8.600 µs	3.420 µs	93.230 ms	4.041 ms	CUDA_API	cudaMemcpyAsync
7.7%	562.969 ms	147023	3.829 µs	1.024 µs	960 ns	991.364 µs	32.925 µs	MEMORY_OPER	[CUDA memcpy Device-to-Device]
5.4%	391.458 ms	4	97.864 ms	99.300 ms	1.929 ms	190.929 ms	104.883 ms	CUDA_API	cudaMallocHost
2.9%	211.224 ms	9041	23.362 µs	30.752 µs	6.880 µs	491.426 µs	45.979 µs	CUDA_KERNEL	std::enable_if<!T7, void>::type internal::gemvx
2.5%	181.921 ms	1	181.921 ms	181.921 ms	181.921 ms	181.921 ms	0 ns	CUDA_API	cudaMemGetInfo
2.5%	178.906 ms	34344	5.209 µs	5.195 µs	3.349 µs	1.300 ms	8.318 µs	CUDA_API	cuLaunchKernelEx
1.9%	140.085 ms	2	70.042 ms	70.042 ms	68.692 ms	71.393 ms	1.910 ms	CUDA_API	cudaFreeHost
1.8%	130.930 ms	203	644.974 µs	584.683 µs	2.856 µs	3.383 ms	562.058 µs	CUDA_API	cudaFree

(b) CUDA Summary(API_Kernels_MemOp)

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
70.5%	1.255 s	15957	78.636 µs	48.064 µs	44.352 µs	594.306 µs	80.349 µs	linear_kernel_large
11.9%	211.224 ms	9041	23.362 µs	30.752 µs	6.880 µs	491.426 µs	45.979 µs	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfloa
4.4%	78.379 ms	2240	34.990 µs	34.976 µs	17.952 µs	36.481 µs	462 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfloa
1.0%	17.735 ms	18144	977 ns	992 ns	832 ns	1.088 µs	33 ns	void at::native::vectorized_elementwise_kernel<(int)4, at::native::FillFunc
0.9%	15.535 ms	9129	1.701 µs	1.792 µs	1.440 µs	2.432 µs	212 ns	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gpu_
0.8%	14.853 ms	2240	6.631 µs	6.592 µs	4.448 µs	8.608 µs	1.480 µs	void pytorch_flash::flash_fwd_kernel<Flash_fwd_kernel_traits<(int)128, (
0.7%	13.077 ms	4480	2.919 µs	2.912 µs	2.752 µs	3.392 µs	85 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfloa
0.7%	12.272 ms	2240	5.478 µs	5.632 µs	2.432 µs	8.832 µs	1.847 µs	self_attention_decode_kernel
0.6%	11.288 ms	4536	2.488 µs	2.496 µs	2.432 µs	3.040 µs	58 ns	rope_kernel
0.6%	11.026 ms	4617	2.388 µs	2.368 µs	2.304 µs	3.328 µs	99 ns	void at::native::reduce_kernel<(int)512, (int)1, at::native::ReduceOp<float
0.6%	10.456 ms	9016	1.159 µs	1.152 µs	1.056 µs	1.312 µs	30 ns	void at::native::vectorized_elementwise_kernel<(int)4, at::native::CUDAF
0.6%	10.433 ms	4779	2.183 µs	2.208 µs	1.248 µs	2.592 µs	126 ns	void at::native::unrolled_elementwise_kernel<at::native::direct_copy_ker

(c) CUDA GPU Kernel Summary

图 11: test_infer.py 运行性能分析（第二次改进）

、

3.4 改进 3

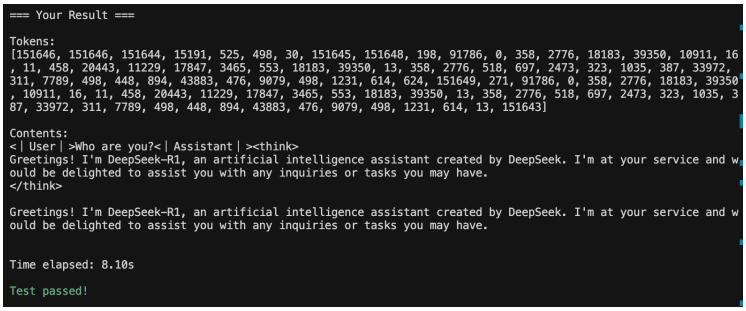
改进细节

经过对 Nsight System 平台的结果的分析，我发现在 kernel 级别的时间消耗主要来自于 linear_kernel_large，占比超过 **60%**，因此，应当主要聚焦于优化该 kernel。于是，我引入了 per-row 的量化手段，具体代码见**附录 A**。具体操作为：仅对 linear 部分的 weight 和 activation 进行同时量化，支持 int8 和 float8 两种量化精度，以加速推理效率。

结果展示

使用 int8 的量化精度对 linear 的 activation 和 weight 同时量化之后，test_infer.py 的运行时间反而增加至 **8.10 s**。可以发现，_per_row_post_process_mm 这个操作用的时间最久（该操作用于时间量化之后的 weight 和 activation 的矩阵乘法），并且在 ker-

nel 中，_per_row_quant_kernel 耗时排第 3（该算子用于对矩阵进行 per-row 量化），比 self_attention 等算子的耗时还要长。说明即使使用了量化方法，但是量化本身就消耗时间（甚至比其他算子的耗时还要长），此外，可能是由于 _per_row_post_process_mm 算子在进行矩阵乘法操作时，没有进行参数搜索等，最终导致了量化方法对性能提升的 failure。



(a) 运行结果

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
27.9%	2.425 s	15957	151.959 µs	89.550 µs	84.586 µs	904.424 µs	157.509 µs	CUDA_KERNEL	_per_row_post_process_mm
14.4%	1.252 s	146394	8.553 µs	5.823 µs	3.856 µs	18.613 ms	96.927 µs	CUDA_API	cudaMemcpy
10.3%	891.857 ms	99188	8.991 µs	3.987 µs	3.052 µs	67.848 ms	356.396 µs	CUDA_API	cudaLaunchKernel
9.3%	809.363 ms	17042	47.492 µs	6.152 µs	3.485 µs	92.365 ms	1.034 ms	CUDA_API	cudaMemcpyAsync
9.3%	804.386 ms	844	953.064 µs	481 ns	352 ns	92.132 ms	4.619 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device]
6.5%	565.240 ms	162263	3.483 µs	1.024 µs	960 ns	999.608 µs	23.839 µs	MEMORY_OPER	[CUDA memcpy Device-to-Device]
4.4%	382.806 ms	4	95.702 ms	92.021 ms	1.887 ms	196.878 ms	103.998 ms	CUDA_API	cudaMallocHost
3.4%	294.202 ms	63698	4.618 µs	4.395 µs	3.521 µs	909.417 µs	3.935 µs	CUDA_API	cuLaunchKernelEx
2.4%	211.392 ms	9041	23.381 µs	30.768 µs	6.947 µs	491.608 µs	45.998 µs	CUDA_KERNEL	std::enable_if<!T7, void>::type internal::gemvx
2.4%	204.199 ms	29354	6.956 µs	2.625 µs	2.241 µs	759.551 µs	39.625 µs	CUDA_KERNEL	_per_row_quant_kernel
2.0%	172.099 ms	1	172.099 ms	172.099 ms	172.099 ms	172.099 ms	0 ns	CUDA_API	cudaMemGetInfo
1.5%	130.701 ms	2	65.351 ms	65.351 ms	64.320 ms	66.381 ms	1.457 ms	CUDA_API	cudaFreeHost

(b) CUDA Summary(API_Kernels_MemOp)

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
77.3%	2.425 s	15957	151.959 µs	89.550 µs	84.586 µs	904.424 µs	157.509 µs	_per_row_post_process_mm
6.7%	211.392 ms	9041	23.381 µs	30.768 µs	6.947 µs	491.608 µs	45.998 µs	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfl
6.5%	204.199 ms	29354	6.956 µs	2.625 µs	2.241 µs	759.551 µs	39.625 µs	_per_row_quant_kernel
2.5%	78.440 ms	2240	35.017 µs	34.993 µs	17.641 µs	36.819 µs	474 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfl
0.5%	15.517 ms	9129	1.699 µs	1.824 µs	1.472 µs	2.466 µs	211 ns	void at::native::elementwise_kernel<(int)128, (int)4, void at::native::gp
0.5%	14.836 ms	2240	6.623 µs	6.564 µs	4.450 µs	8.485 µs	1.476 µs	void pytorch_flash::flash_fwd_kernel<Flash_fwd_kernel_traits<(int)12
0.4%	13.092 ms	4480	2.922 µs	2.913 µs	2.753 µs	3.394 µs	83 ns	std::enable_if<!T7, void>::type internal::gemvx::kernel<int, int, __nv_bfl
0.4%	12.440 ms	2240	5.553 µs	5.699 µs	2.465 µs	8.933 µs	1.859 µs	self_attention_decode_kernel
0.4%	11.439 ms	4536	2.521 µs	2.529 µs	2.497 µs	3.202 µs	64 ns	rope_kernel
0.3%	10.745 ms	4617	2.327 µs	2.305 µs	2.273 µs	3.361 µs	96 ns	void at::native::reduce_kernel<(int)512, (int)1, at::native::ReduceOp<flo
0.3%	10.653 ms	2268	4.697 µs	4.706 µs	4.482 µs	4.802 µs	26 ns	swiglu_kernel
0.3%	10.404 ms	4779	2.177 µs	2.177 µs	1.281 µs	2.561 µs	123 ns	void at::native::unrolled_elementwise_kernel<at::native::direct_copy_k

(c) CUDA GPU Kernel Summary

图 12: test_infer.py 运行性能分析（第三次改进，量化精度为 int8）

A Triton Kernels for Quantized Linear Operations

为保证实现的完整性与可复现性，本文在附录中给出量化线性算子的完整 Triton 实现代码。该实现包括逐行量化（per-row quantization）kernel 以及量化矩阵乘法的后处理 kernel，支持 `int8` 与 `FP8` 两种量化格式，并在 kernel 内完成尺度恢复与偏置融合操作。正文实验中所使用的量化线性算子均基于以下实现。

```
1         import triton
2         import triton.language as tl
3         import torch
4         from triton.language.extra import libdevice
```

```
5
6       @triton.heuristics(values={"PAD_H": lambda args: triton.
    next_power_of_2(args["H"])})
7       @triton.jit
8       def _per_row_quant_kernel(
9       X, Y, S, T, H, BLK_M: tl.constexpr, PAD_H: tl.constexpr, FP8: tl.
    constexpr
10      ):
11      bidx = tl.program_id(0)
12      x_ptrs = (
13      X
14      + (tl.arange(0, BLK_M)[:, None] * H + tl.arange(0, PAD_H)[None,
    :])
15      + bidx * BLK_M * H
16      )
17      s_mask = (tl.arange(0, BLK_M) + bidx * BLK_M) < T
18      mask = ((tl.arange(0, PAD_H) < H)[None, :]) & (s_mask[:, None])
19      x = tl.load(x_ptrs, mask=mask, other=0).to(tl.float32)
20      row_max = tl.maximum(tl.max(tl.abs(x), axis=-1), 1e-8)
21
22      if FP8:
23      scale = row_max / 448
24      scale_inv = 448 / row_max
25      else:
26      scale = row_max / 127
27      scale_inv = 127 / row_max
28
29      scaled_x = x * scale_inv[:, None]
30
31      if FP8:
32      y = tl.cast(scaled_x, tl.float8e4nv)
33      else:
34      y = tl.cast(libdevice.round(scaled_x), tl.int8)
35
36      y_ptrs = (
37      Y
38      + (tl.arange(0, BLK_M)[:, None] * H + tl.arange(0, PAD_H)[None,
    :])
39      + bidx * BLK_M * H
```

```
        )
        s_ptrs = S + (tl.arange(0, BLK_M)) + bidx * BLK_M

        tl.store(y_ptrs, y, mask=mask)
        tl.store(s_ptrs, scale, mask=s_mask)


        @triton.jit
        def _per_row_post_process_mm(
        A, sA, B, sB, Bias, C,
        M, N, K,
        BLK_M: tl.constexpr,
        BLK_N: tl.constexpr,
        BLK_K: tl.constexpr,
        FP8: tl.constexpr,
        ):
        bidy = tl.program_id(0)
        bidx = tl.program_id(1)

        row_start = bidx * BLK_M
        col_start = bidy * BLK_N

        rows = row_start + tl.arange(0, BLK_M)
        cols = col_start + tl.arange(0, BLK_N)

        row_mask = rows < M
        col_mask = cols < N

        acc = tl.zeros((BLK_M, BLK_N), dtype=tl.float32)

        for k_start in range(0, K, BLK_K):
        k_offsets = k_start + tl.arange(0, BLK_K)
        k_mask = k_offsets < K

        a_ptrs = A + rows[:, None] * K + k_offsets[None, :]
        a_mask = row_mask[:, None] & k_mask[None, :]
        if FP8:
        a = tl.load(a_ptrs, mask=a_mask, other=0.0).to(tl.float32)
        else:
```

```
79          a = tl.load(a_ptrs, mask=a_mask, other=0).to(tl.int32).to(tl.
    float32)
80
81          b_ptrs = B + cols[:, None] * K + k_offsets[None, :]
82          b_mask = col_mask[:, None] & k_mask[None, :]
83          if FP8:
84          b = tl.load(b_ptrs, mask=b_mask, other=0.0).to(tl.float32)
85          else:
86          b = tl.load(b_ptrs, mask=b_mask, other=0).to(tl.int32).to(tl.
    float32)
87
88          acc += tl.dot(a, tl.trans(b))
89
90          sa = tl.load(sA + rows, mask=row_mask, other=0.0)
91          sb = tl.load(sB + cols, mask=col_mask, other=0.0)
92
93          c = acc * sa[:, None] * sb[None, :]
94
95          if Bias:
96          bias = tl.load(Bias + cols, mask=col_mask, other=0.0)
97          c += bias[None, :]
98
99          c_ptrs = C + rows[:, None] * N + cols[None, :]
100         c_mask = row_mask[:, None] & col_mask[None, :]
101         tl.store(c_ptrs, c, mask=c_mask)
102
```

Listing 9: Triton Kernels for Quantized Linear Operations