量化推理

量化计算

简单的说，就是算子内部用 int8 做计算，但并不是所有 op，都用 int8。

对于 transformer 架构，或者说绝大部份深度学习模型，主要的计算就是 gemm， 并且 gemm 是线性运算，因此量化 gemm ，并不会造成太多精度损失。

LLM 架构中，主要包含 gemm 运行有： **qkv_proj, output_proj, fc_up, fc_down**

Attention 计算公式上也有两个 gemm（softmax(Q*K^T/sqrt(dq)*V）， 由于内部有 softmax 运算（非线形），因此量化 attention 很难，可能存在较大误差。

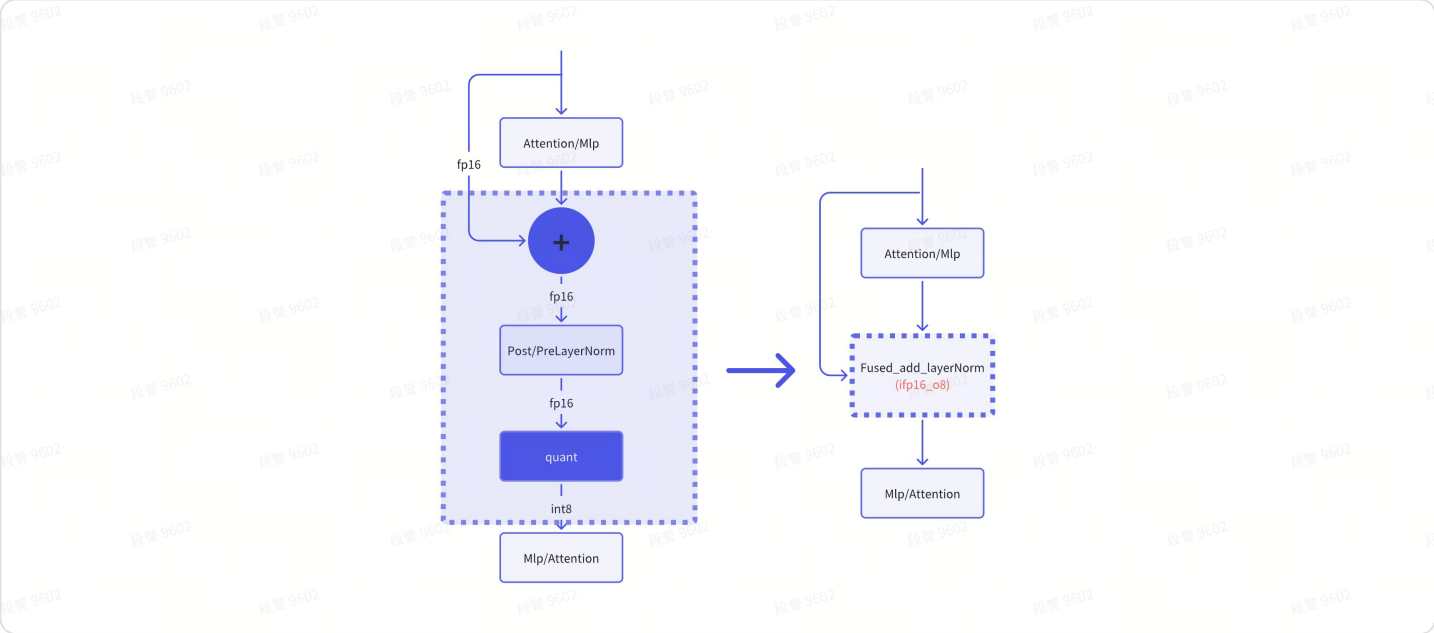
当然也可以把 Attention 里面的主要部分拆成两个 gemm 加一个 softmax，其中的两个 gemm 用量化推理，softmax 由于非线性，依然采用 float32 或者 half 计算，~~但是效果肯定不如一个融合的 half attention 算子（可以结合 int8 kv，减少带宽限制））~~(对prefill阶段可能有较大收益，decode阶段收益可能小写)

当然 attention 的全量化推理，也有一些尝试： IBert：让Transformer实现全量化推理

由于并非整个模型使用全量化推理，有的 op 是 fp16推理，有的 op 是 int8 推理，因此对于这种混合精度计算，从计算逻辑上来说，需要在模型的计算流程上插入一些 quant or dequant 算子，在 llm 推理中，并不需要显示的插入这些算子，而是将这些 quant 和 dequant 计算，融合在一些算子当中（算子融合）。

Per/PostLayerNorm

Llama 中使用的是 RMS Norm，且 layernorm 前面会有一个 redisual add 操作，因此这两个操作可以融合在一个算子中， pre layer norm 前面是上一层 mlp的输出，后面是 attention，post layer norm 前面是 attention 的输出，后面是 mlp，因此，该算子还需要融合对中间值的量化操作，便于后面的 attention or mlp 做 int8_gemm，计算流程图如下：



fused_add_layerNorm op 输入是 fp16，输出是int8（ifp16_o8），作为 mlp 中 gemm 的输入。

这里简单说下 llm 中常见的**量化方式：**

Per tensor：对于中间激活值和权重，每个tensor 共用一个scale，离线量化

Per token：因为hidden state的shape 为（num_tokens, hidden_dim），所以可以选择不同行（token）使用不同的scale，但是由于不同 prompt 包含的 token 个数不同，所以使用推理时**动态量化**，推理时 kernel 内部计算 dynamic scale。

其他量化方式：hidden state的shape 为（num_tokens, hidden_dim），num_tokens 是不固定的，但是 hidden_dim 是固定的，所以可以每一列使用不同的 scale，这种方式为 **Per channel**

该 fused_add_layerNorm kernel 较为简单，详细修改见 Quant_Kernel 部分。

Cutlass gemm

正如上面所说，量化主要是做 gemm_int8 ，CUTLASS是NV开源的高性能计算库，支持各种精度以及混合精度的 gemm 计算，并能够自定义一些和 gemm 的融合操作（epilogue），比如反量化。同时可以指定 Tensor Core，充分发挥硬件性能。

> CUTLASS is a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) and related computations at all levels and scales within CUDA.

Cutlass 最新的 3.0+ 版本，支持 int4，甚至 int2 混合精度的 gemm 计算。
https://github.com/NVIDIA/cutlass/tree/main/examples/55_hopper_mixed_dtype_gemm

参考阅读：

quickstart.md

Efficient GEMM in CUDA

cutlass源码导读（1）——API与设计理念

cutlass源码导读（2）——Gemm的计算流程

cutlass源码导读（3）——核心软件抽象

cutlass源码导读（4）——软件分层与源码目录

将 cutlass gemm 封装为基于 torch tensor 的 api 见算子仓库：https://git.singularity-ai.com/haoran.lin/cutlass_gemm. 如:

`gemm_in8_w8_ofp16_pt`：输入、weight 是int8， 输出是 fp16，量化方式是 per_tensor

`gemm_infp16_w8_ofp16`：weigtOnly量化：输入是 fp16、weight 是int8， 输出是 fp16

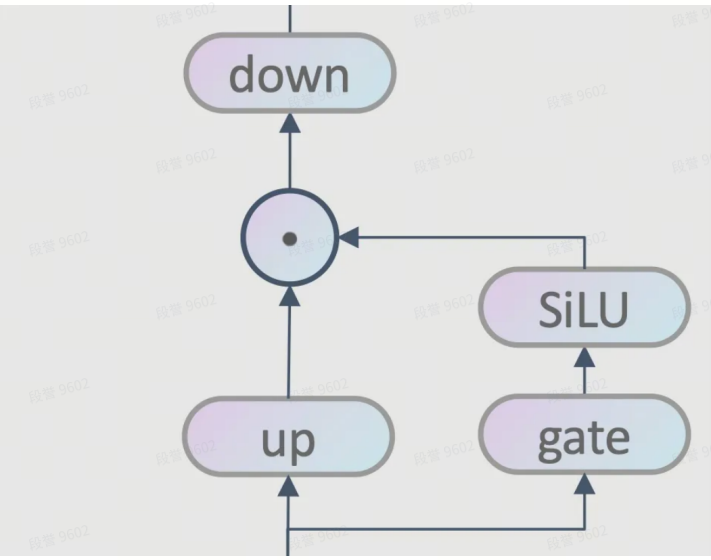
更多细节，可以查看上述抽离出来的算子库。

> Cutlass 将 gemm 的通用优化分解为很多组件，用户可以进行随意配置。国内对cutlass 源码剖析的中文资料很少，而且深度也不够。

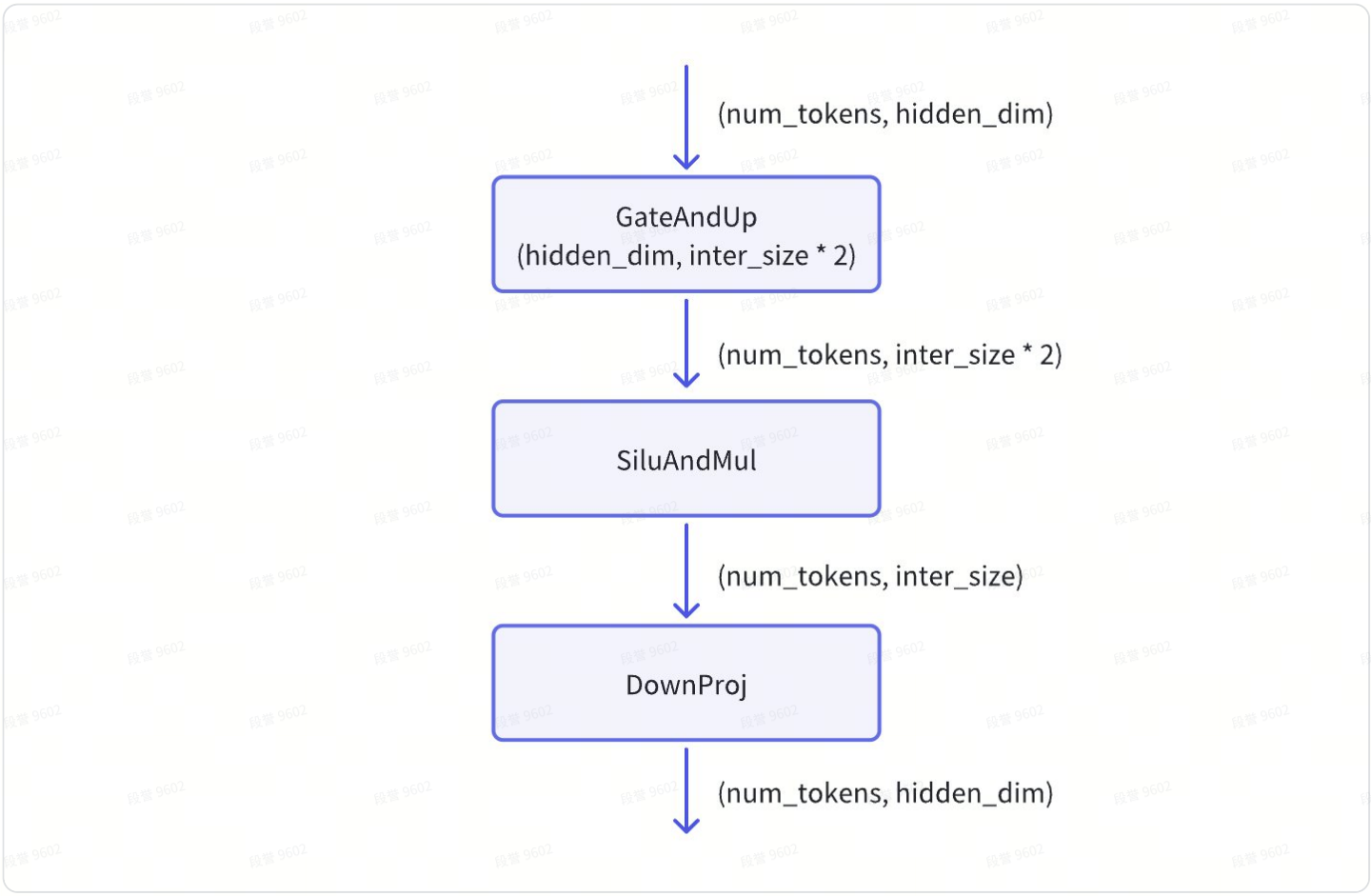
MLP 部分的量化

llama的mlp部分为：

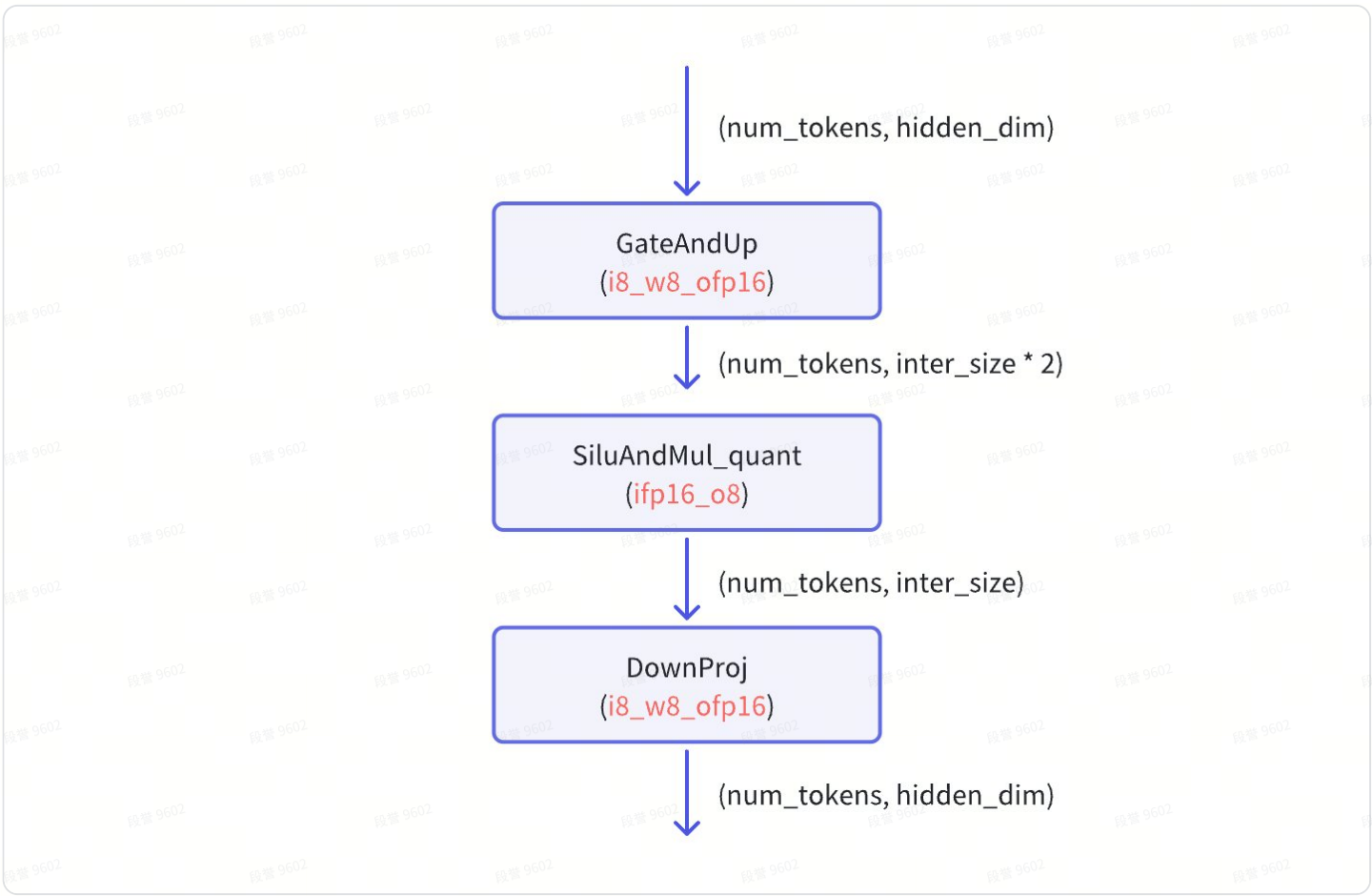
$$down(up(x) \times SiLU(gate(x)))$$



在实现中，往往会把 gate 和 up 两个权重合并成一个（hidden_dim, inter_size * 2）的weight权重矩阵，这样原本三个 gemm 缩减为了两个，计算流程如下。其中 SiluAndMul 会计算前 inter_size 列（相当于 gate 的输出）的 Silu 激活值，并和后 inter_size 列（相当于up的输出）做点乘。vllm 中的实现很 naive，我们也把这部分抽离出来放入了算子库中，具体 kernel 细节可以查看注释。



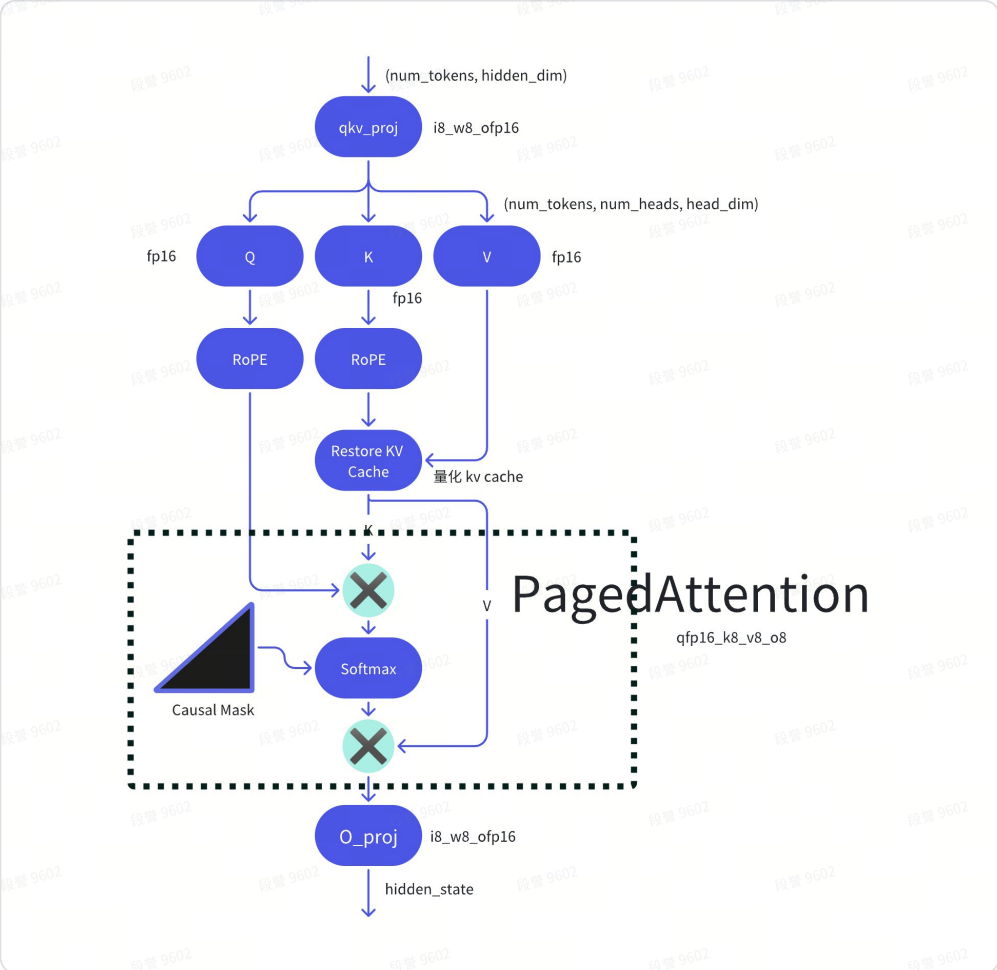
前后两个 gemm 可以用 int8 算，但是中间的激活 SimuAndMul 是非线性激活函数，因此不做量化。GateAndUp 前面的 PostLayerNorm 已经融合了 quant 算子，因此输入就是 int8 的输入，同时 gemm 可以融合反量化操作，所以它的输出就是 fp16。量化计算的计算流程图如下:



注意， SiluAndMul 算子融合了 quant 操作，优化后的该算子也单独封装为了算子库，kernel 中有详细的注释。

Attention 部分的量化

如同 mlp 部分，先梳理 Attention 部分的计算逻辑：



注意，图中的 Restore Kv Cache 算子，就是分散存储 kv cache（Paged），在该算子内部增加 kv cache 量化逻辑，具体代码修改见最后的 Quant_kernel。

这里选择量化 kv cache，因为绝大部份的显存都消耗在了 kv cache 上，在 paged attention 内部，加载 int8 的 kv，转换为 fp16后，再进行计算，最后将输出量化为 int8。因为生成阶段，该 kernel 本身属于 memory bind，将 kv 量化为 int8，可以提高算子性能。

PagedAttion 是 vllm 中最复杂最核心的算子，Quant_kernel中有介绍。

Quant_Kernel

1. RMSNom_quant

融合了residual add 和量化算子的 RMSnorm

RMS：Root Mean Square 均方根：

计算方式：

$$RMS = \sqrt{\frac{x_1^2+x_2^2+...+x_n^2}{n}}$$

均方根标准化的计算公式为：

$$RMSNorm(x_1, x_2, ..., x_n) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2+\epsilon}}$$

ϵ是一个小的正数，用于防止分母为零。

深度学习中，再乘了一个 weight，Computes: `x -> x / sqrt(E[x^2] + eps) * w` where w is the learned weight.

Add + LayerNorm Cuda 实现：

问题输入规模： (num_tokens, hidden_size)

算法设计： 一个 thread block 负责一行，一行负责 hidden_size 个元素，因此：

```
template<typename scalar_t>
__global__ void fused_add_rms_norm_kernel(
  scalar_t* __restrict__ input,           // [..., hidden_size]
  scalar_t* __restrict__ residual,        // [..., hidden_size]
  const scalar_t* __restrict__ weight,    // [hidden_size]
  const float epsilon,
  const int num_tokens,
  const int hidden_size) {
  __shared__ float s_variance;
  float variance = 0.0f;

  for (int idx = threadIdx.x; idx < hidden_size; idx += blockDim.x) {
    float x = (float) input[blockIdx.x * hidden_size + idx];
    x += (float) residual[blockIdx.x * hidden_size + idx];
    variance += x * x;
    residual[blockIdx.x * hidden_size + idx] = (scalar_t) x;
  }
  variance = blockReduceSum<float>(variance);
  if (threadIdx.x == 0) {
    s_variance = rsqrtf(variance / hidden_size + epsilon);
  }
  __syncthreads();

  for (int idx = threadIdx.x; idx < hidden_size; idx += blockDim.x) {
    float x = (float) residual[blockIdx.x * hidden_size + idx];
    input[blockIdx.x * hidden_size + idx] = ((scalar_t) (x * s_variance)) *
weight[idx];
  }
```

```
28  }
```

这个kernel较简单，没有使用向量化读取，计算访存比很低，上述 cuda 代码是计算的基本逻辑。在此基础上使用使用向量化读取并融合量化的代码见：https://git.singularity-ai.com/shengying.wei/ops_library/-/blob/master/csrc/trt_rmsnorm.cu#L229

2. SiluAndMul_quant

如 mlp 量化部分的介绍，vllm 的原生实现也非常简单

问题输入规模：（num_tonkers, inter_size * 2)， ifp16_o8

算法设计: **One CTA handle one row(token)**

```
1   template<typename scalar_t>
2   __global__ void silu_and_mul_kernel(
3     scalar_t* __restrict__ out,                // [..., d]
4     const scalar_t* __restrict__ input,        // [..., 2, d]
5     const int d) {
6     const int64_t token_idx = blockIdx.x;
7     for (int64_t idx = threadIdx.x; idx < d; idx += blockDim.x) {
8       const scalar_t x = __ldg(&input[token_idx * 2 * d + idx]); // __ldg 可以提高缓存命中率
9       const scalar_t y = __ldg(&input[token_idx * 2 * d + d + idx]);
10      out[token_idx * d + idx] = silu(x) * y;
11    }
12  }
```

可以利用向量化读取进行优化，优化后并融合了output 量化的 kernel 可阅读算子库中对应的部分：

https://git.singularity-ai.com/shengying.wei/ops_library/-/blob/master/csrc/trt_silu.cu#L170

3. 在 Restore kv cache 算子中量化 kv

分页存储 kv cache 原理非常简单，可以理解为提前申请一个巨大的 kv 存储池（vllm中就是一个多维torch tensor表示），然后池子分块（block），每个块都有自己独立的编号（0～num_blocks-1），每个块只能存储一定数量token 的kv值（block_size）

问题输入规模: k (num_tokens, num_heads, head_size)

算法设计: **One CTA handle one row(token)**， 输入 slot_mapping 中记录着 kv 中每个token即将在池子中存储的位置。

该 kernel 较为简单，源码如下：

```
1   template<typename scalar_t>
```

```
__global__ void reshape_and_cache_kernel(
  const scalar_t* __restrict__ key,           // [num_tokens, num_heads, head_size]
  const scalar_t* __restrict__ value,         // [num_tokens, num_heads, head_size]
  scalar_t* __restrict__ key_cache,           // [num_blocks, num_heads, head_size/x, block_size, x]
  scalar_t* __restrict__ value_cache,         // [num_blocks, num_heads, head_size, block_size]
  const int64_t* __restrict__ slot_mapping,   // [num_tokens]
  const int key_stride,
  const int value_stride,
  const int num_heads,
  const int head_size,
  const int block_size,
  const int x,
  const float* quant_scale) {
  const int64_t token_idx = blockIdx.x;
  const int64_t slot_idx = slot_mapping[token_idx];
  if (slot_idx < 0) {
    // Padding token that should be ignored.
    return;
  }
  bool quant_output = quant_scale != nullptr;
  char* key_cache_quant;;
  char* value_cache_quant;
  if (quant_output){
    key_cache_quant = reinterpret_cast<char*>(key_cache);
    value_cache_quant = reinterpret_cast<char*>(value_cache);
  }
  const int64_t block_idx = slot_idx / block_size;
  const int64_t block_offset = slot_idx % block_size;

  const int n = num_heads * head_size;
  for (int i = threadIdx.x; i < n; i += blockDim.x) {
    const int64_t src_key_idx = token_idx * key_stride + i;
    const int64_t src_value_idx = token_idx * value_stride + i;

    const int head_idx = i / head_size;
    const int head_offset = i % head_size;
    const int x_idx = head_offset / x;
    const int x_offset = head_offset % x;

    const int64_t tgt_key_idx = block_idx * num_heads * (head_size / x) * block_size * x
                                + head_idx * (head_size / x) * block_size * x
                                + x_idx * block_size * x
```

```
                                    + block_offset * x
                                    + x_offset;
    const int64_t tgt_value_idx = block_idx * num_heads * head_size *
block_size
                                      + head_idx * head_size * block_size
                                      + head_offset * block_size
                                      + block_offset;
    if (quant_output){
      key_cache_quant[tgt_key_idx] =
__float2int_rn(cast_to_float(key[src_key_idx]) * quant_scale[0]);
      value_cache_quant[tgt_value_idx] =
__float2int_rn(cast_to_float(value[src_value_idx])*quant_scale[0]);
    }else{
      key_cache[tgt_key_idx] = key[src_key_idx];
      value_cache[tgt_value_idx] = value[src_value_idx];
    }

  }
}
```

注意量化部分，当前的代码逻辑是 kv 共用一个 scale, quant_scale[0]。

4. PagedAttentionV1

略，基于 vllm 的实现，增加一些反量化（kv）和量化（output）操作，其余逻辑不变

5. PagedAttentionV2

Flash attention版本的 paged attention，之后有空会增加详细剖析

模型组网和切分

除了第一部分 “量化计算” 对计算逻辑、Kernel 的修改之外，适配 int8 量化参数的模型组网和权重加载也有一些改动。组网上多了一些 scale 参数以及使用上述抽离出来的算子库中的 api 进行推理，load weight 上主要 int8 weight 的多卡切分。

这部分看代码即可。

量化方法/工具/以及精度验证

TODO：阅读总结相关论文

性能测试

不同量化方式对性能的影响是不同的，以前面的所说的 per_token 和 per_tensor 为例说明。

对于 gemm，每个 thread 做 FMA 运算：

`c += alpha * (a*b) + beta * c + bias` **a,b,c 为 half 或者 float**

当输入是 int8 时，我们用 q_a 、q_b 表示，它的计算一般为（下面是伪代码）

```
c += alpha * (q_a * s_a * q_b * s_b) + beta * c  + bias
```

不同高性能库计算逻辑可能不同，但都是大同小异。

即等同于：

```
c += new_alpha * (a*b) + beta * c + bias
```

```
new_alpha = alpha * s_a * s_b
```

对于 per tensor 量化来说， s_a 和 s_b 只有一个值，因此这个 scale 会放入寄存器中，计算速度很快。

但对于 per_token 量化来说，在 prefill 阶段，num_token 可能非常大，这么多 scale 不可能全部放入寄存器中，甚至 shared memory 都存不下，所以性能上 per token 是比 per tensor 更差的，甚至比 half 的 fma 要差。

TODO：具体模型测试