如何实现比PyTorch快6倍的Permute/Transpose算子?

撰文 | 郑泽康、柳俊丞、姚迟、郭冉

前言

无论是在统治NLP届的Transformer，还是最近作为在视觉领域的新秀Vision Transformer，我们都能在模型中看到Transpose/Permute算子的身影，特别是在多头注意力机制(Multi-Head Attention)中需要该算子来改变数据维度排布。

显然作为一个被高频使用的算子，其CUDA实现会影响到实际网络的训练速度。本文会介绍OneFlow中优化Permute Kernel的技巧，并跟PyTorch的Permute，原生的Copy操作进行实验对比。结果表明，经过深度优化后的Permute操作在速度和带宽利用率上远超PyTorch，带宽利用率能够接近原生Copy操作。

朴素的Permute实现

Permute算子的作用是变换张量数据维度的顺序，举个例子：

```
x = flow.randn(2, 3)
y = x.permute(1, 0)
y.shape
(3, 2)
```

其实现原理也可以很容易理解，即**输出Tensor的第i维，对应输入Tensor的dims[i]维**，上述例子中 permute 的实现对应的伪代码如下：

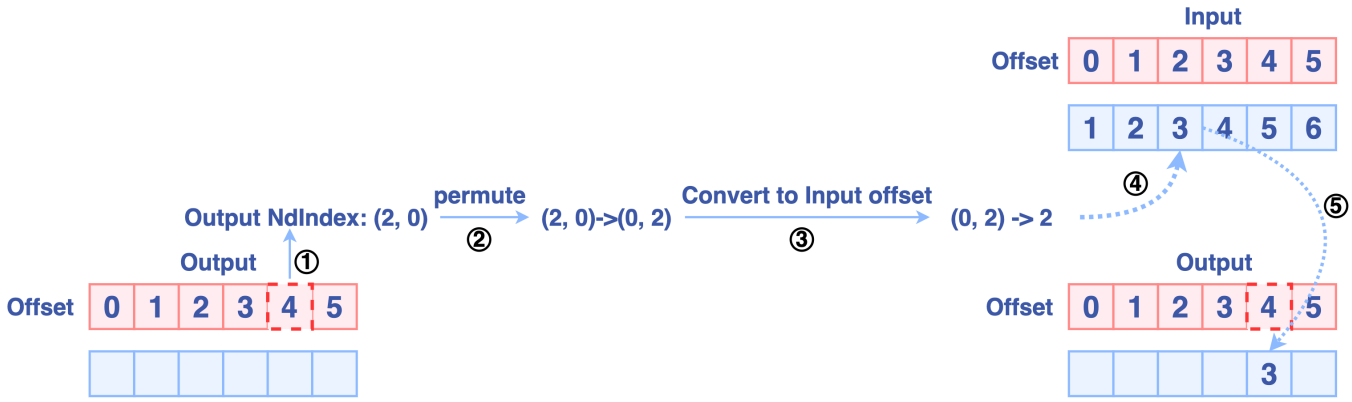
```
for row in x.shape[0]:
  for col in x.shape[1]:
    y[row][col] = x[col][row]
```

但是实际情况与上面的伪代码有出入，张量的Shape是数学上的概念，在物理设备上并不真实存在。在OneFlow中，张量的数据都是保存在一块连续的内存中，下图分别从上层视角和底层视角描述了形状为(2, 3)的张量的存储方式：

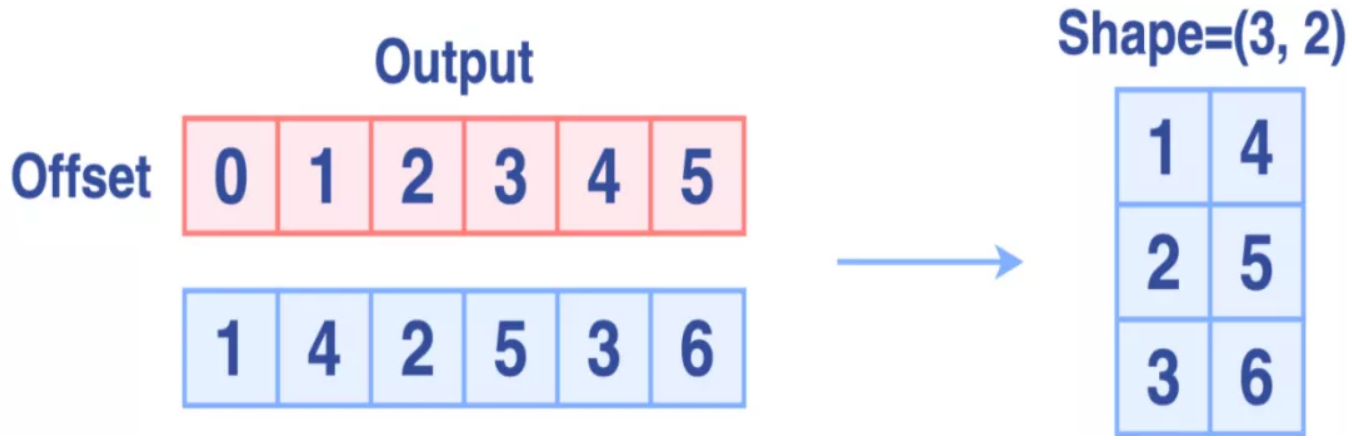
OneFlow的Permute实现原理为：

1. 通过当前输出的一维偏移量(offset)计算对应的高维索引
2. 然后根据参数dims重新排列输出索引，进而得到输入索引。

3．将输入索引转换成输入偏移量
4．最后进行数据移动，整个过程的示意图如下：



完成Permute后，输出如下图所示：



整个 permute 计算过程需要经过多次需要经过多次**一维偏移量offset和高维索引之间的转换**，为了避免一次次手工计算，OneFlow提供了一个工具类**NdIndexOffsetHelper**来方便做上述转换。

NdIndexOffsetHelper

NdIndexOffsetHelper的主体方法如下：

- NdIndexToOffset方法把高维索引转为一维偏移量
- OffsetToNdIndex方法把一维偏移量转为高维索引

有了这么一个工具类，那我们就可以很轻松的写出一版Naive Permute Kernel了，核函数如下：

```
template<size_t num_dims, size_t movement_size, typename IndexType>
__global__ void PermuteKernel(PermuteKernelParams<num_dims, IndexType> params) {
  using T = typename std::aligned_storage<movement_size, movement_size>::type;
  const T* src = reinterpret_cast<const T*>(params.src);
  T* dst = reinterpret_cast<T*>(params.dst);
  IndexType src_index[num_dims];
  IndexType dst_index[num_dims];
  CUDA_1D_KERNEL_LOOP_T(IndexType, i, params.count) {
    params.dst_index_helper.OffsetToNdIndex(i, dst_index);
#pragma unroll
```

```
    for (size_t dim = 0; dim < num_dims; ++dim) {
      src_index[params.permutation[dim]] = dst_index[dim];
    }
    IndexType src_offset = params.src_index_helper.NdIndexToOffset(src_index);
    dst[i] = src[src_offset];
  }
}
```

- PermuteKernelParams是一个结构体，里面有初始化好的NdIndexOffsetHelper(src和dst各一个)，元素总数count还有变换后的维度顺序permutation
- 首先我们取得当前处理输出元素的高维索引dst_index，然后赋给经过Permute后的输入索引src_index
- 再将输入索引转换成一维偏移量src_offset，取到输入元素并赋给对应的输出

常规情况的优化

这种朴素Permute Kernel的计算代价来源于坐标换算，访存开销则来源于数据移动，针对这两个角度我们引入以下优化方案。

1．IndexType静态派发

随着深度学习模型越来越大，参与运算元素的个数可能超过int32_t表示的范围。并且在坐标换算中，不同整数类型的除法运算开销不一样。因此我们给核函数增加了一个模板参数IndexType用于指定索引的数据类型，根据参与Permute的元素个数来决定IndexType是int32_t还是int64_t。

2．合并冗余维度

在一些特殊情形下，Permute维度是可以进行合并的，其规则如下：

1．大小为1的维度可以直接去除
2．连续排列的维度可以合并成一个维度

针对第二条规则，我们考虑以下Permute情况

```
# (0, 1, 2, 3) → (2, 3, 0, 1)
x = flow.randn(3, 4, 5, 6)
y = x.permute(2, 3, 0, 1)
y.shape
(5, 6, 3, 4)
```

显然这是一个四维的Permute情形，但这里第2，3维，第0，1维是一起Permute的，所以我们可以看成是一种二维的Permute情形：

```
# (0, 1, 2, 3) → ((2, 3), (0, 1))
x = x.reshape(x.shape[0]*x.shape[1], x.shape[2]*x.shape[3])
y = x.permute(1, 0)
y = y.reshape(x.shape[2], x.shape[3], x.shape[0], x.shape[1])
```

合并维度后，在利用NdIndexOffsetHelper根据偏移量计算索引时，合并前需要计算成四维索引，而合并后我们只需计算成二维索引。相比合并前**减少除法和乘法的次数**，进而提升速度。

3．使用更大的访问粒度

细心的朋友们可能观察到核函数中有一个模板参数size_t movement_size，它表示的是访问元素的粒度。

在Nvidia性能优化博客increase Performance with Vectorized Memory Access中提到可以通过向量化内存操作来提高CUDA Kernel性能，能够减少指令数，提高带宽利用率。

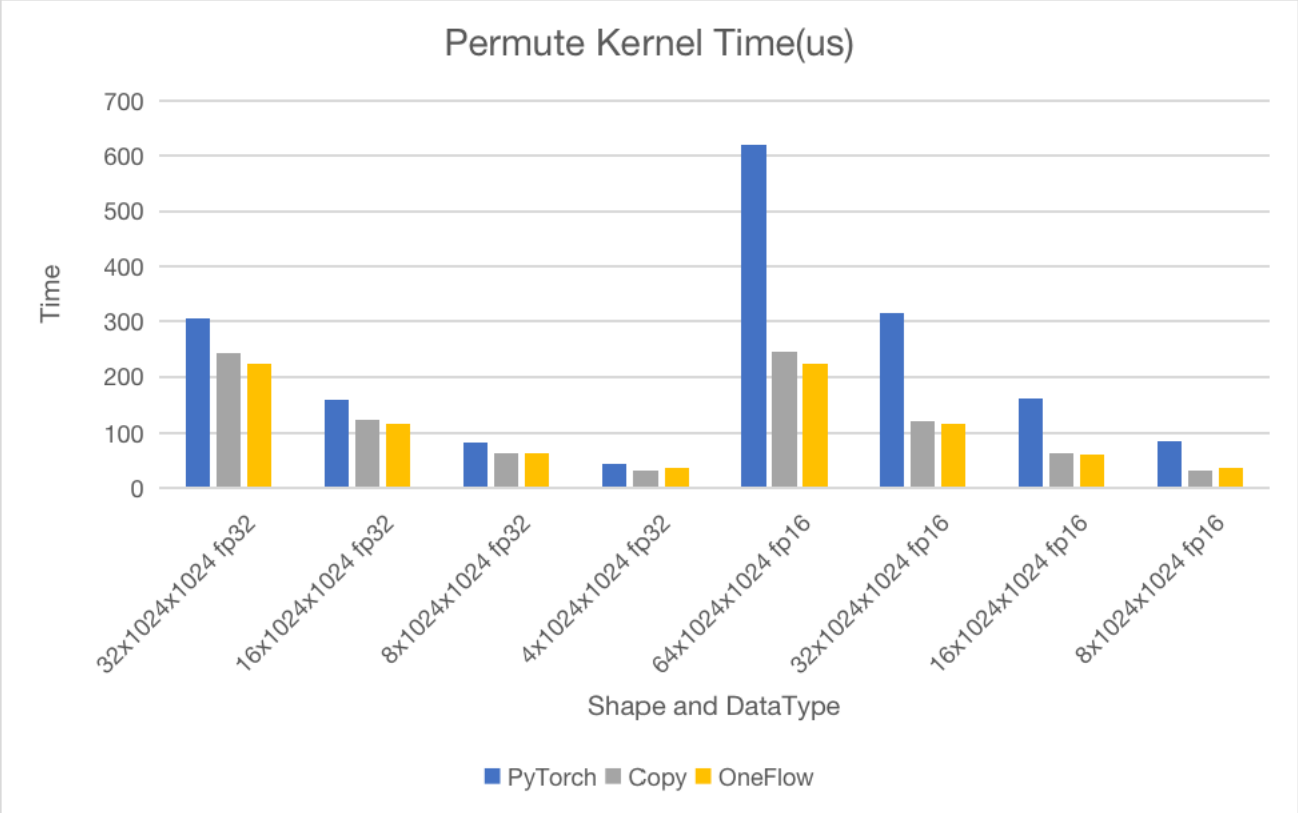
我们设置访问粒度的规则如下：

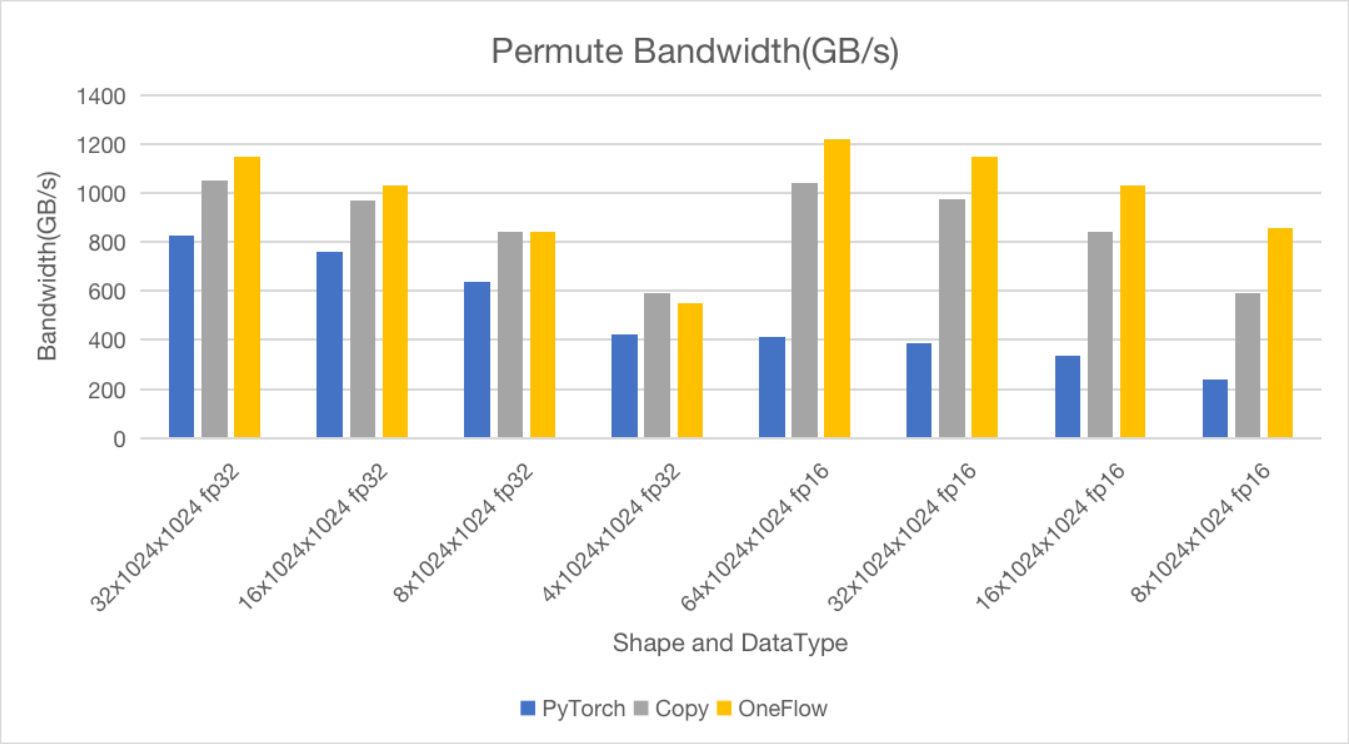
1. CUDA支持的访问粒度为1B，2B，4B，8B，16B，粒度越大性能越好
2. 最后一个维度是作为整体来移动的，即permutation[n-1]==x.dims[n-1]，且大小是新访问粒度的倍数
3. 保证数据指针满足新访问粒度的对齐要求

针对规则2，对应着以下Permute场景：
(0, 1, 2, 3) → (0, 2, 1, 3)

其中**最后一维并没有变化**，仅仅是第1，2维进行交换，那么我们可以**使用更大的访问粒度来读取数据**，再进行Permute操作。代码中我们通过GetMovementSize函数来确定访问粒度的大小。

我们使用Nsight Compute，对PyTorch的Permute和原生Copy操作对比测试运行时间和带宽，测试结果如下：





其中测试环境为NVIDIA A100 40GB，场景为(0, 1, 2)→(1, 0, 2)，横坐标表示数据形状及数据类型。测试数据覆盖了16MB到128MB不同大小的数据，数据类型包含fp32和half两种类型。

从上面两张图可以看到，OneFlow在大部分情况下都可以逼近甚至略高于Copy操作的带宽。在操作耗时上与PyTorch对比，最少快1.24倍，最快能达1.4倍。

> *这里Permute的带宽比原生Copy还高一点是因为Copy Kernel里没有做unroll指令间并行优化，而Permute Kernel内部做了相关优化，这里仅做参考。*

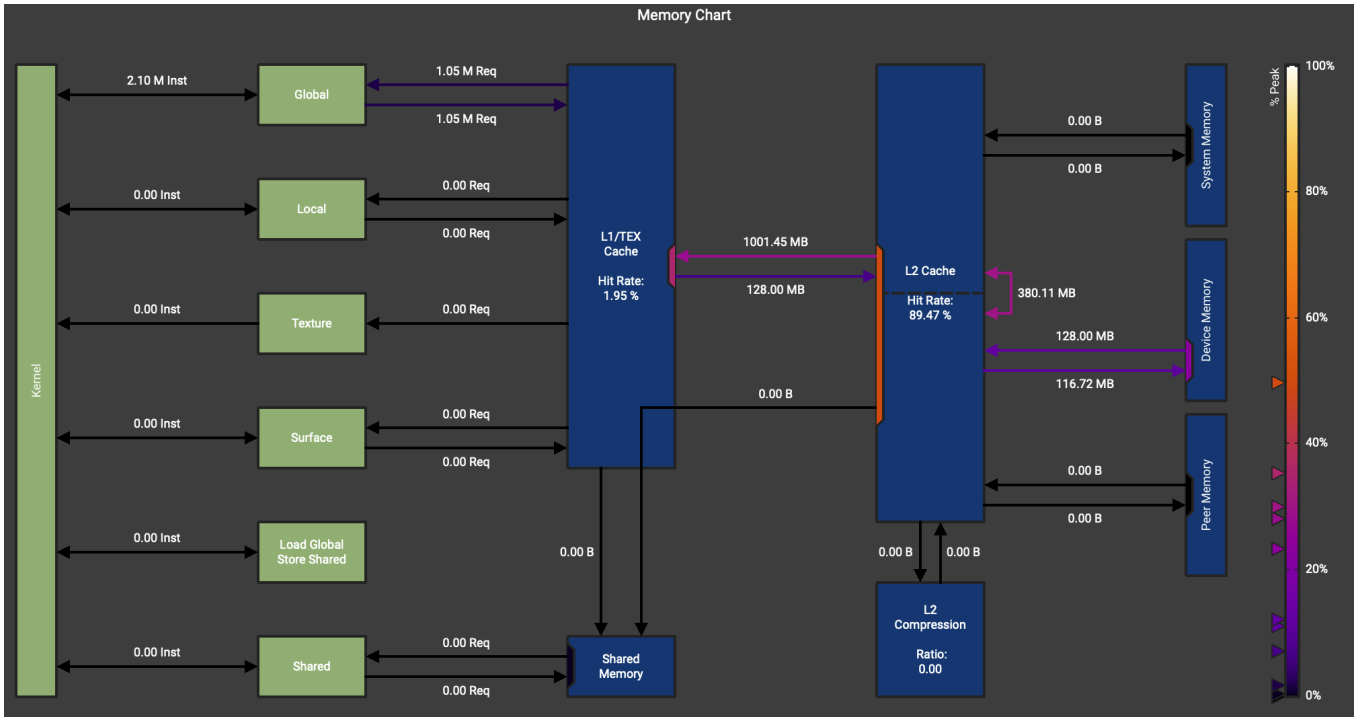
使用上面的两个优化技巧，OneFlow就能轻易做到比PyTorch的实现要快了。常规的Permute适用情况比较广泛，也因此可能存在访存不合并的情况。在一些特殊的场景下，我们可以通过合并访存以提升带宽利用率和速度，这就引出我们下个关于BatchTranspose优化的话题。

BatchTranspose优化

BatchTranspose操作即矩阵转置，**仅交换矩阵最后的两维**，以下情况均符合BatchTranspose的定义，其中括号内容表示维度的顺序：

```
(0, 1) → (1, 0)
(0, 1, 2) → (0, 2, 1)
```

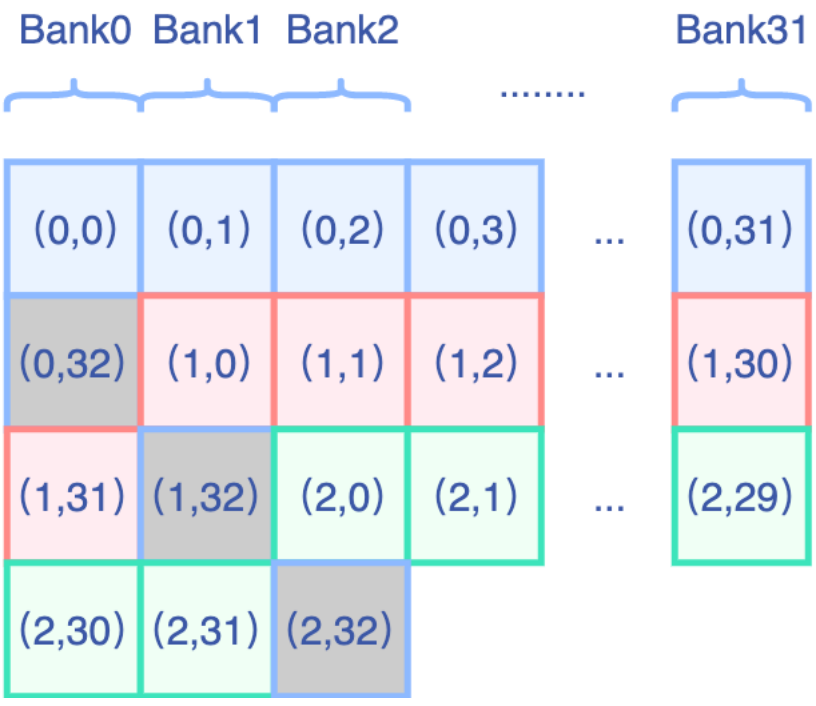
在朴素的Permute方案中，对于最后一维作为整体移动的情况下，已经进行充分的优化。**但实际场景中还存在矩阵转置的情况，此时无法应用第三条增大访问粒度的优化操作，并且不满足访存合并要求，导致性能不佳。**以Pytorch为例子，在数据大小为128MB情况下进行BatchTranspose时，因为未合并的访存导致实际读取数据量远大于写入数据量(7-8倍)。



在英伟达性能优化博客An Efficient Matrix Transpose in CUDA C/C++中，其做法是设置一块Shared Memory，然后将一行数据读取到Shared Memory，再按列顺序将Shared Memory中的元素写回到Global Memory中。得益于Shared Memory访问粒度小的特性(Global Memory是32B，Shared Memory是4B)，进而避免Global Memory的访存不连续的问题。

Shared Memory相比Global Memory有15倍更高的带宽，20-40倍更低的延迟，因此额外引入的读写开销可以忽略不计。

此外我们给Shared Memory多padding了一个元素，**进而让以列顺序访问的元素能够均匀分布在32个bank上，避免bank conflict**。对应的示意图如下(其中灰色部分代表Padding元素)：



基于上述提到的点我们实现了一版BatchTranspose，代码如下：

```
template<size_t num_dims, size_t movement_size, size_t tile_size, typename IndexType>
__global__ void BatchTransposeKernel(const void* src_ptr, void* dst_ptr, IndexType H, IndexType W,
                                     IndexType num_tile_rows, IndexType num_tile_cols,
                                     int32_t block_nums) {
  using T = typename std::aligned_storage<movement_size, movement_size>::type;
  __shared__ T tile[tile_size][tile_size + 1];  // To avoid bank conflict.

  const T* src = reinterpret_cast<const T*>(src_ptr);
  T* dst = reinterpret_cast<T*>(dst_ptr);

  IndexType batch_num_tile = num_tile_rows * num_tile_cols;
  for (int i = blockIdx.x, step = gridDim.x; i < block_nums; i += step) {
    const IndexType batch_index = i / batch_num_tile;  // the index of batch.
    const IndexType flatten_index =
        i - batch_index * batch_num_tile;
    const IndexType row_index = flatten_index / num_tile_cols;  // the row index of tile in a batch.
    const IndexType col_index =
        flatten_index
        - row_index
              * num_tile_cols;  // the col index of tile in a batch.
    const IndexType offset = batch_index * H * W;
    IndexType x = col_index * tile_size + threadIdx.x;
    IndexType y = row_index * tile_size + threadIdx.y;
    if (x < W) {
      IndexType y_range =
          ((tile_size - threadIdx.y) < (H - y)) ? (tile_size - threadIdx.y) : (H - y);
#pragma unroll
      for (int i = 0; i < y_range; i += kBlockRows) {
        tile[threadIdx.y + i][threadIdx.x] = src[offset + (y + i) * W + x];
      }
    }
    __syncthreads();
    x = row_index * tile_size + threadIdx.x;
    y = col_index * tile_size + threadIdx.y;
    if (x < H) {
      IndexType x_range =
          ((tile_size - threadIdx.y) < (W - y)) ? (tile_size - threadIdx.y) : (W - y);
#pragma unroll
      // `i < x_range` equals to: `threadIdx.y + i < tile_size && y + i < W`.
      for (int i = 0; i < x_range; i += kBlockRows) {
        dst[offset + (y + i) * H + x] = tile[threadIdx.x][threadIdx.y + i];
      }
    }
    __syncthreads();
  }
}
```

其中BatchTranspose的优化涉及以下两点：

4．显式展开循环

在先前版本，我们的for循环写法如下：

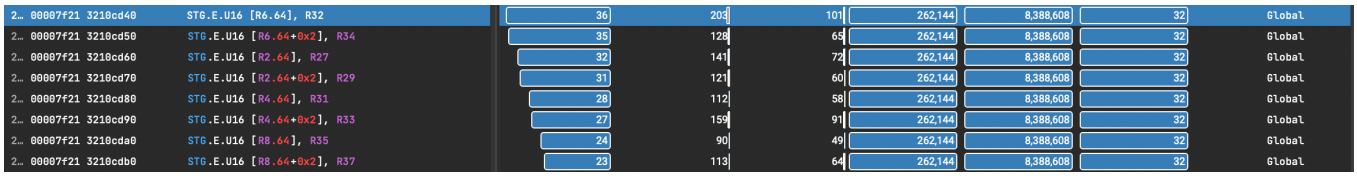
```
#pragma unroll
for (int i = 0; threadIdx.y + i < tile_size && y + i < H; i += kBlockRows) {
    ...
}
```

即便是加入了预编译指令`#pragma unroll`，在Nsight Compute里的汇编代码中，我们也只能看到两条相关指令，也就意味着这部分循环并没有展开。

而for循环里的条件，我们可以化简并提取出来，如下代码所示：

```
IndexType y_range = ((tile_size - threadIdx.y) < (H - y)) ? (tile_size - threadIdx.y) : (H - y);
#pragma unroll
for (int i = 0; i < y_range; i += kBlockRows) {
  ...
}
```

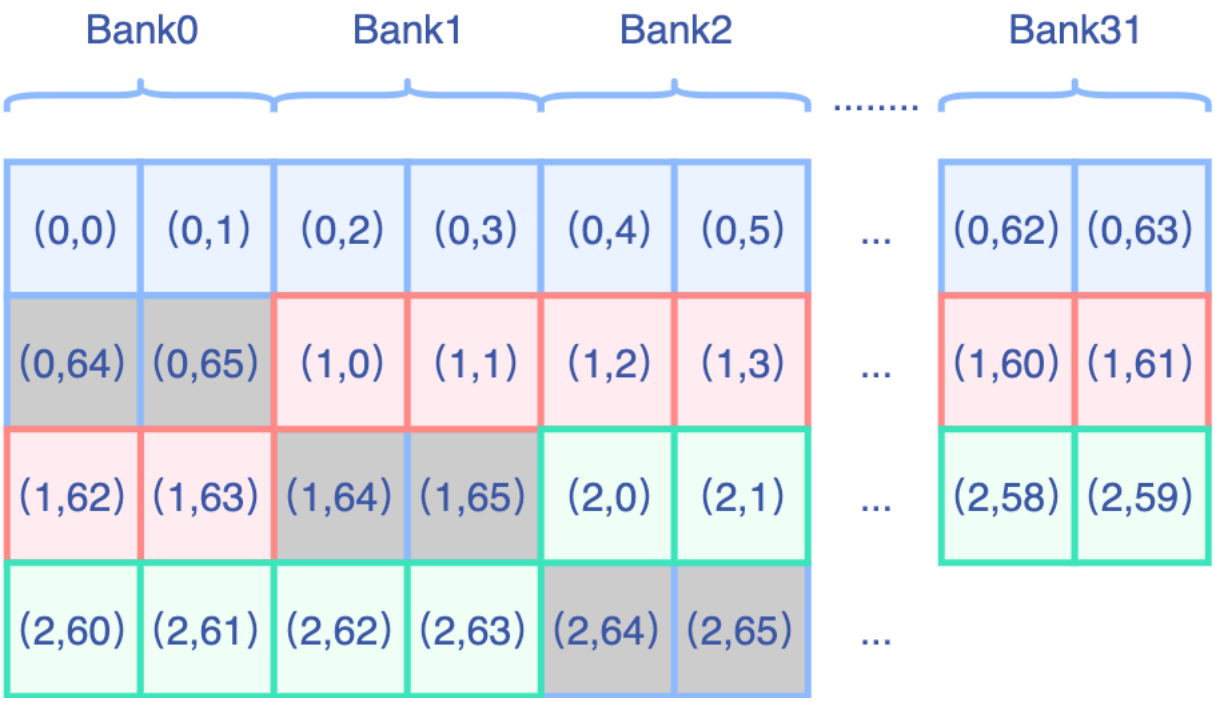
此时对应的汇编代码显示这部分的循环进行了展开，在带宽利用率和速度上有24%的提升。



5. 针对half2版本优化

特别的，针对half数据类型，且转置维度均能被2整除的情况下，我们可以进一步利用half2来合并。

Shared Memory的一个bank宽度为4B，那么一个bank能塞下两个half数据，示意图如下：



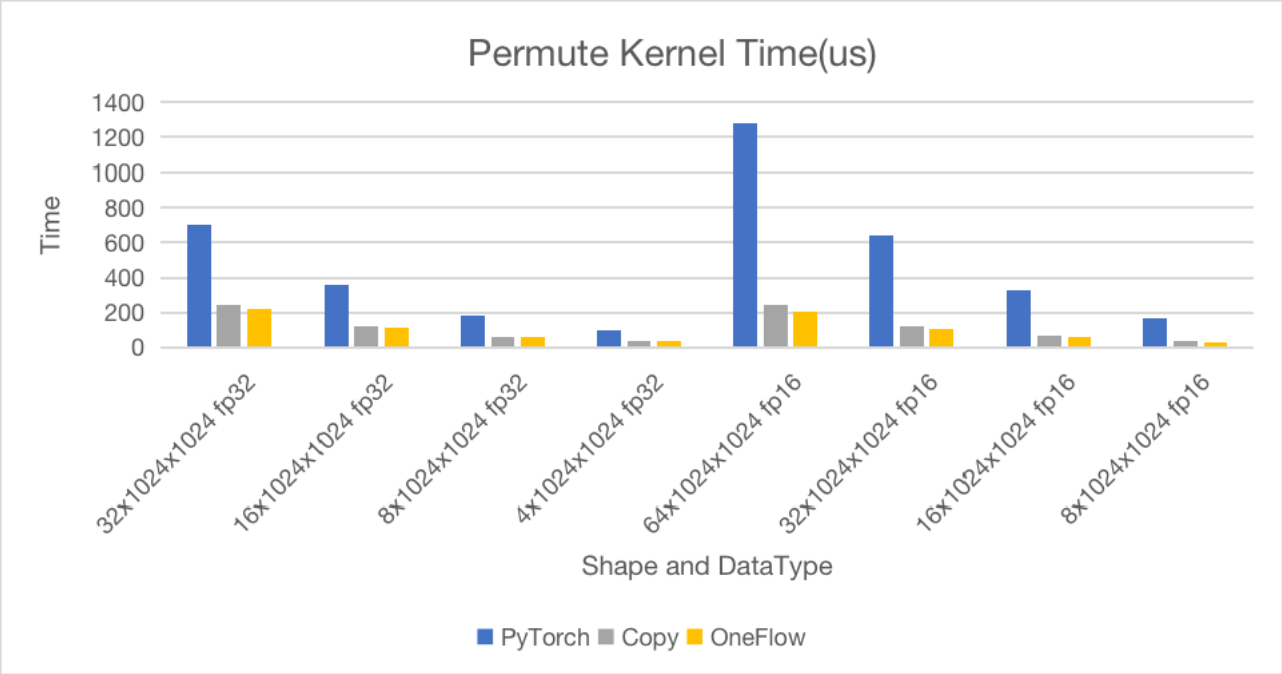
那么加载到Shared Memory的时候，**我们可以将两个half数据合并为half2类型进行加载。**

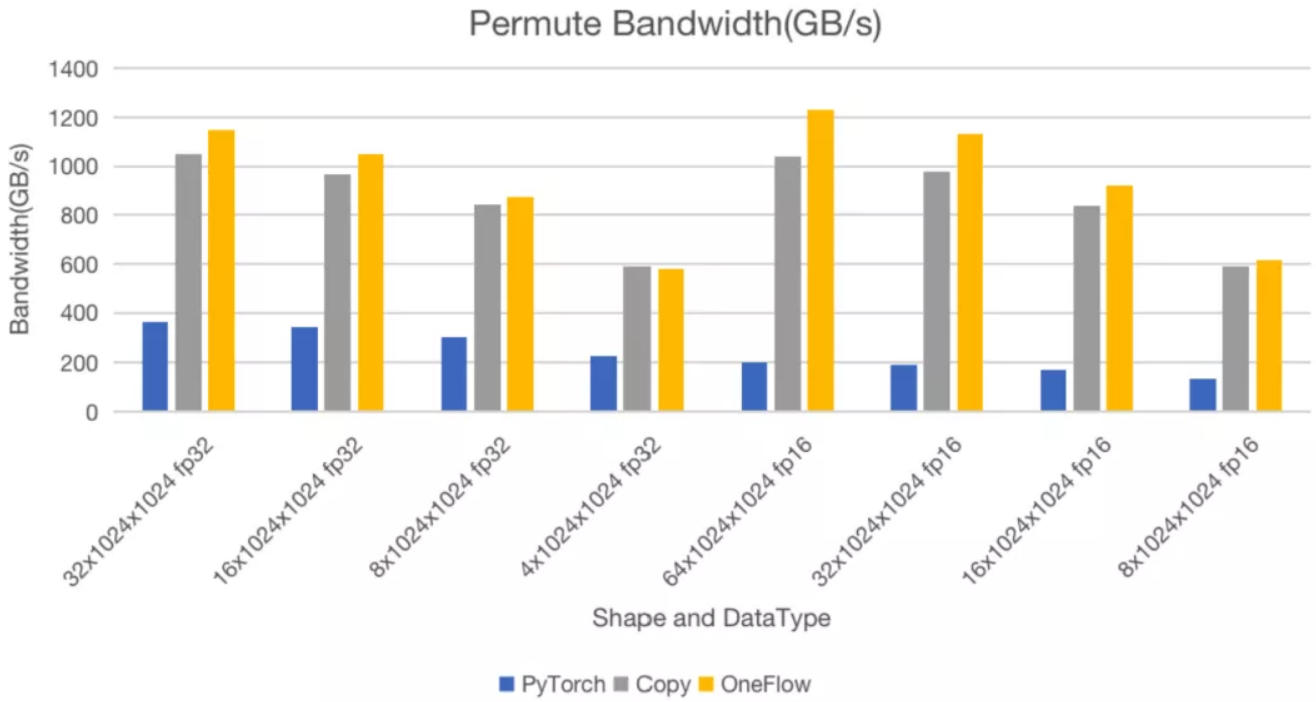
但是取列元素的时候，因为元素分布在两个不同的bank上，不能合并成half2直接取。**需要构造一个临时的half2对象，分别将两个bank上的half元素存储到该half2对象，再写回到Global Memory里。**对应的代码如下：

```
template<size_t num_dims, size_t tile_size, typename IndexType>
__global__ void BatchTransposeMovement2Kernel(const void* src_ptr, void* dst_ptr, IndexType rows,
                                              IndexType cols, IndexType num_tile_rows,
                                              IndexType num_tile_cols, int32_t block_nums) {
  static_assert(tile_size % 2 == 0);
  using T_MOV2 = typename std::aligned_storage<2, 2>::type;
```

```
  using T_MOV4 = typename std::aligned_storage<4, 4>::type;

  const T_MOV4* src = reinterpret_cast<const T_MOV4*>(src_ptr);
  T_MOV4* dst = reinterpret_cast<T_MOV4*>(dst_ptr);

  // Use union structure to process Load and Store.
  __shared__ union {
    T_MOV2 tile_m2[tile_size][tile_size + 2];      // half [64][66]
    T_MOV4 tile_m4[tile_size][tile_size / 2 + 1];  // half2 [64][33]
  } tile_mem;

  IndexType batch_num_tile = num_tile_rows * num_tile_cols;
  for (int i = blockIdx.x, step = gridDim.x; i < block_nums; i += step) {
    const IndexType batch_index = i / batch_num_tile;  // the index of batch.
    const IndexType flatten_index =
        i - batch_index * batch_num_tile;  // the flatten index of tile in a batch.

    const IndexType row_index = flatten_index / num_tile_cols;  // the row index of tile in a batch.
    const IndexType col_index =
        flatten_index
        - row_index
              * num_tile_cols;  // equal to k % num_tile_cols. the col index of tile in a batch.
    const IndexType offset = batch_index * rows * cols;
    IndexType x =
        col_index * tile_size + threadIdx.x * 2;  // cause each thread process a half2 element, we need
    IndexType y = row_index * tile_size + threadIdx.y;
    if (x < cols) {
      // each thread process 4 elements.
      IndexType y_range =
          ((tile_size - threadIdx.y) < (rows - y)) ? (tile_size - threadIdx.y) : (rows - y);
#pragma unroll
      // `i < y_range` equals to: `threadIdx.y + i < tile_size && y + i < rows`.
      for (int i = 0; i < y_range; i += kBlockRows) {
        // each thread load a half2.
        tile_mem.tile_m4[threadIdx.y + i][threadIdx.x] = src[(offset + (y + i) * cols + x) / 2];
      }
    }
    __syncthreads();
    x = row_index * tile_size + threadIdx.x * 2;  // cause each thread process a half2 element, we need
    y = col_index * tile_size + threadIdx.y;
    if (x < rows) {
      IndexType x_range =
          ((tile_size - threadIdx.y) < (cols - y)) ? (tile_size - threadIdx.y) : (cols - y);
#pragma unroll
      // `i < x_range` equals to: `threadIdx.y + i < tile_size && y + i < cols`.
      for (int i = 0; i < x_range; i += kBlockRows) {
        /*
        When write back as column, it cannot be stored as half2 directly.
        So we split as 2 half elements, and write back separately.
        */
        union {
          T_MOV4 m4;
          T_MOV2 m2[2];
        } tmp_storage;
        tmp_storage.m2[0] = tile_mem.tile_m2[threadIdx.x * 2][threadIdx.y + i];
        tmp_storage.m2[1] = tile_mem.tile_m2[threadIdx.x * 2 + 1][threadIdx.y + i];
        dst[(offset + (y + i) * rows + x) / 2] = tmp_storage.m4;
      }
    }
    __syncthreads();
  }
}
```

在前面相同的测试条件下，我们将测试场景设置为`(0, 1, 2)→(0, 2, 1)`，测试结果如下：





可以看到，OneFlow在大部分情况下，无论是计算耗时，还是带宽利用率都可以逼近原生Copy操作。在操作耗时上与PyTorch对比，fp32数据类型情况下最少快3倍，最快能达3.2倍。**而half数据类型情况下OneFlow优势更为明显，最快能达6.3倍。**

未来优化方向

经过我们实际测试，在坐标换算过程中，整数除法的运算开销比较大。而市面上有很多优秀的运算库如Eigen，lemire/fast_division都提供了基于int32，int64类型的快速除法，根据官方提供的benchmark测试结果，快速除法相较于标准除法能提升1-3倍性能。未来我们将探索合适的快速除法用于坐标转换中，进一步提升运算速度。

展望

从本文和之前OneFlow发布的CUDA优化文章中可以看到，在kernel优化过程中有一些常见、通用的手段，如合并冗余以减少计算次数、调整访问粒度以提高访存效率。

这些常见、通用的优化手段，是有可能被提炼出，作为深度学习编译器的组件，来部分替代手工调优工作。

但是，自动优化边界的确定、以及如何自动优化，都提出了比手工调优更高的要求，据我们所知也还是一个半开放的问题。欢迎感兴趣的同道，在OneFlow仓库提issue讨论、研发。

参考资料：

1.CUDA Pro Tip: Increase Performance with Vectorized Memory Access

2.An Efficient Matrix Transpose in CUDA C/C++

3.VOLTA Architecture and performance optimization

题图源自geralt, Pixabay

其他人都在看