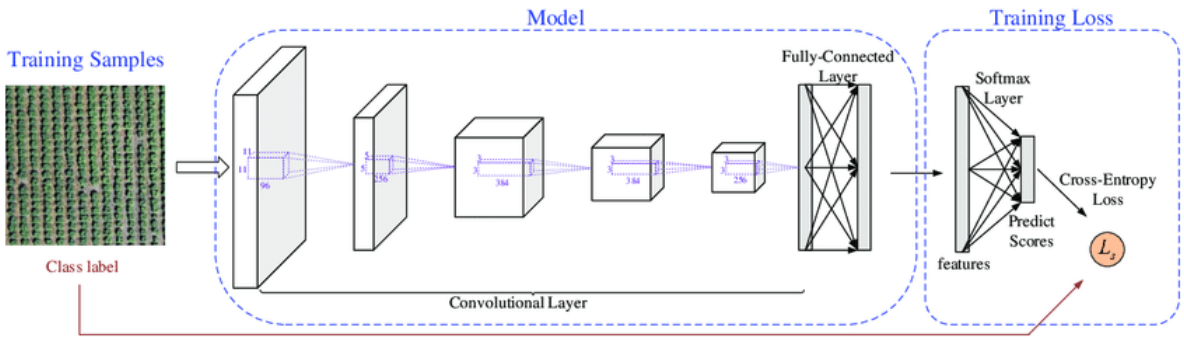
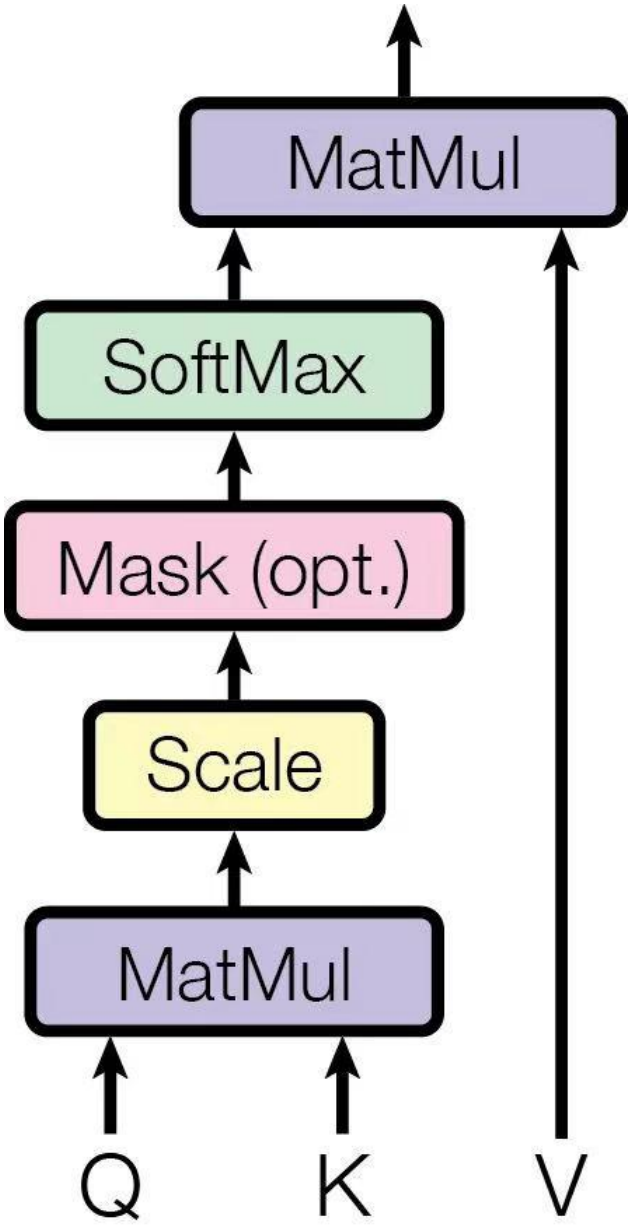
如何实现一个高效的Softmax CUDA kernel? —— OneFlow 性能优化分享

撰文 | 郭冉

Softmax操作是深度学习模型中最常用的操作之一。在深度学习的分类任务中，网络最后的分类器往往是Softmax + CrossEntropy的组合：



尽管当Softmax和CrossEntropy联合使用时，其数学推导可以约简，但还是有很多场景会单独使用Softmax Op。如BERT的Encoder每一层的attention layer中就单独使用了Softmax求解attention的概率分布；GPT-2的attention的multi-head部分也单独使用了Softmax 等等。



深度学习框架中的所有计算的算子都会转化为GPU上的CUDA kernel function，Softmax操作也不例外。Softmax作为一个被大多数网络广泛使用的算子，其CUDA Kernel实现高效性会影响很多网络最终的训练速度。那么如何实现一个高效的Softmax CUDA Kernel？本文将会介绍OneFlow中优化的Softmax CUDA Kernel的技巧，并跟cuDNN中的Softmax操作进行实验对比，结果表明，OneFlow深度优化后的Softmax对显存带宽的利用率可以接近理论上限，远高于cuDNN的实现。

GPU基础知识与CUDA性能优化原则：

对GPU基础知识的介绍以及CUDA性能优化的原则与目标可以参考之前的文章：

https://zhuanlan.zhihu.com/p/271740706

其中简单介绍了GPU的硬件结构与执行原理：

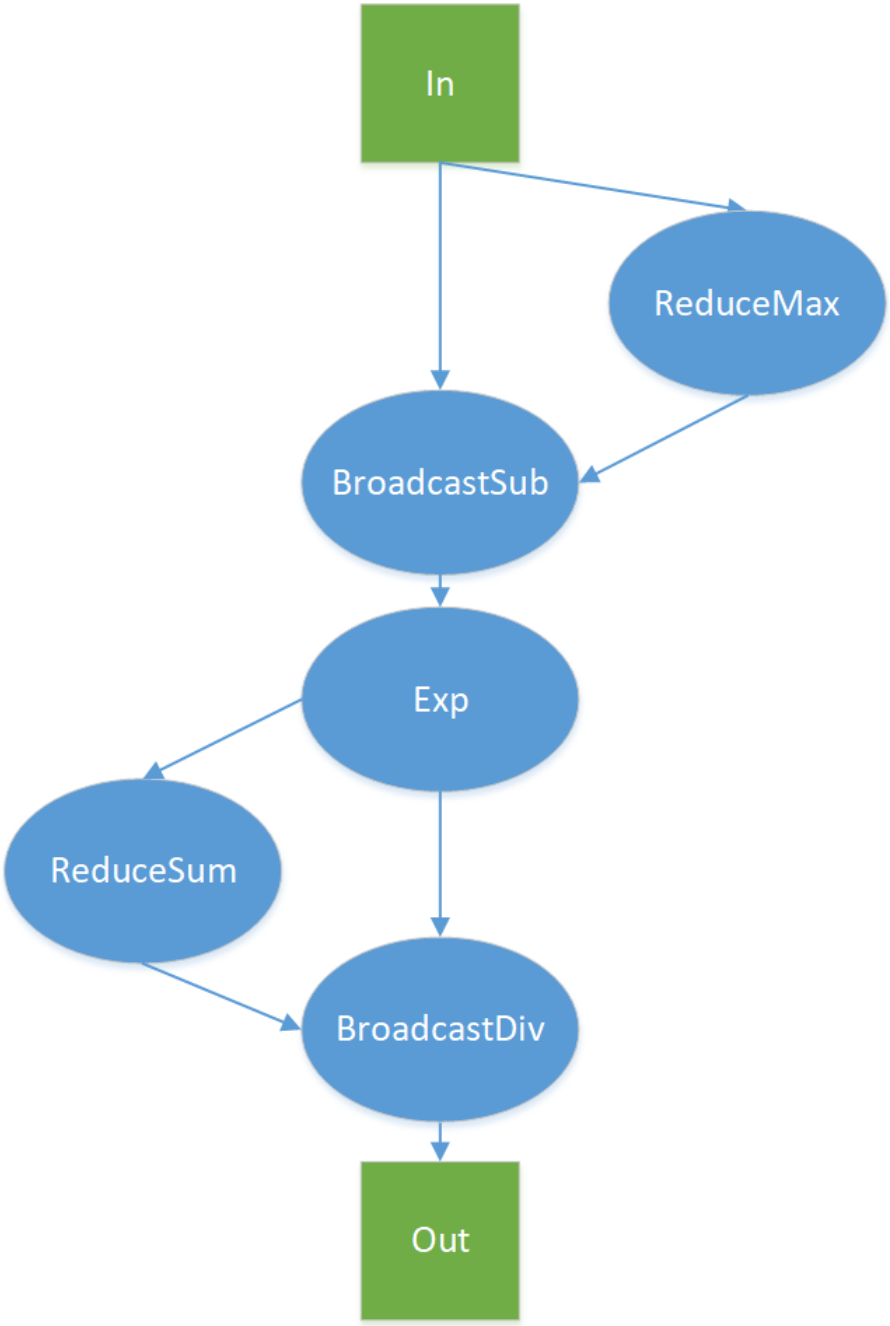
- Kernel：即CUDA Kernel function，是GPU的基本计算任务描述单元。每个Kernel都会根据配置参数在GPU上由非常多个线程并行执行，GPU计算高效就是因为同时可以由数千个core（thread）同时执行，计算效率远超CPU。
- GPU的线程在逻辑上分为Thread、Block、Grid三级，硬件上分为core、**warp**两级；
- GPU内存分为Global memory、Shared memory、Local memory三级。
- GPU最主要提供的是两种资源：**计算资源** 和 **显存带宽资源**。如果我们能将这两种资源充分利用，且对资源的需求无法再降低，那么性能就优化到了极限，执行时间就会最短。在大多数情况下，深度学习训练中的GPU计算资源是被充分利用的，而一个GPU CUDA Kernel的**优化目标**就是尽可能充分利用显存带宽资源。

如何评估一个CUDA Kernel是否充分利用了显存带宽资源?

对于显存带宽资源来说，“充分利用“指的是Kernel的有效显存读写带宽达到了设备显存带宽的**上限**，其中设备显存带宽可以通过执行 cuda中的的bandwidthTest得到。 Kernel的有效显存带宽通过Kernel读写数据量和Kernel执行时间进行评估： $$ 当前Kernel的有效显存带宽 = 读写数据量 / 执行时间 $$

Naive的Softmax实现：

在介绍优化技巧之前，我们先看看一个未经优化的Softmax Kernel的最高理论带宽是多少。如下图所示，一个最简单的Softmax计算实现中，分别调用几个基础的CUDA Kernel function来完成整体的计算：



假设输入的数据大小为D，shape = (num_rows, num_cols)，即 D = num_rows * num_cols ，最Navie的操作会多次访问Global memory，其中：

- ReduceMax = D + num_rows （read 为 D， write 为 num_rows)
- BroadcaseSub = 2 * D + num_rows （read 为 D + num_rows，write 为 D)
- Exp = 2 * D （read 、write 均为D)
- ReduceSum = D + num_rows (read 为 D， write 为 num_rows)
- BroadcastDive = 2 * D + num_rows （read 为 D + num_rows， write 为 D)

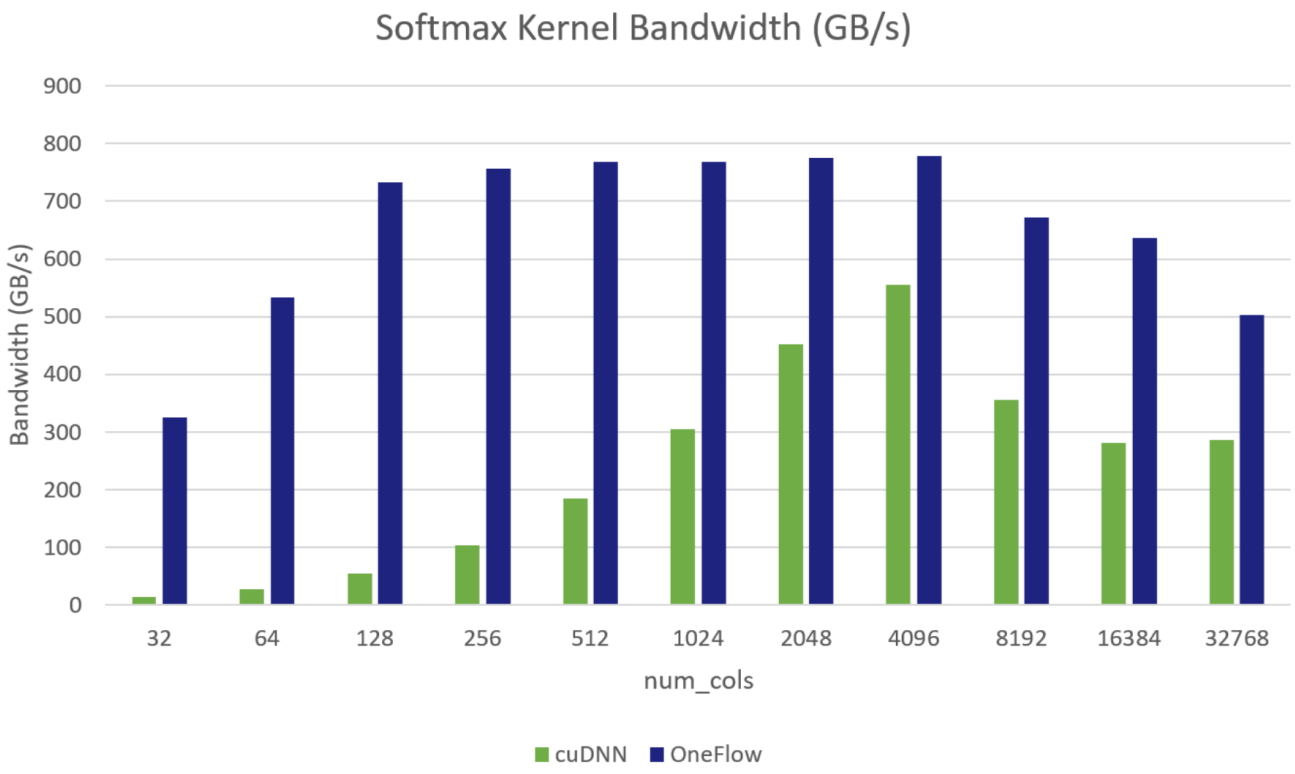
总共需要8 * D + 4 * num_rows 的访存开销。由于num_rows相比于D可以忽略，则Navie版本的Softmax CUDA Kernel需要访问至少8倍数据的显存，即： $$ Naive Softmax Kernel 有效显存带宽 < 理论带宽 / 8 $$ 对于GeForce RTX™ 3090显卡，其理论带宽上限为**936**GB/s，那么Navie版本的Softmax CUDA Kernel利用显存带宽的上界就是936 / 8 = 117 GB/s。

在文章 https://zhuanlan.zhihu.com/p/271740706 里，我们在方法：**借助shared memory合并带有Reduce计算的Kernel**中提到了对Softmax Kernel的访存优化到了 2 * D，但这仍然没有优化到极致。在本文的优化后，大多数场景下OneFlow的Softmax CUDA Kernel的显存带宽利用可以逼近理论带宽。

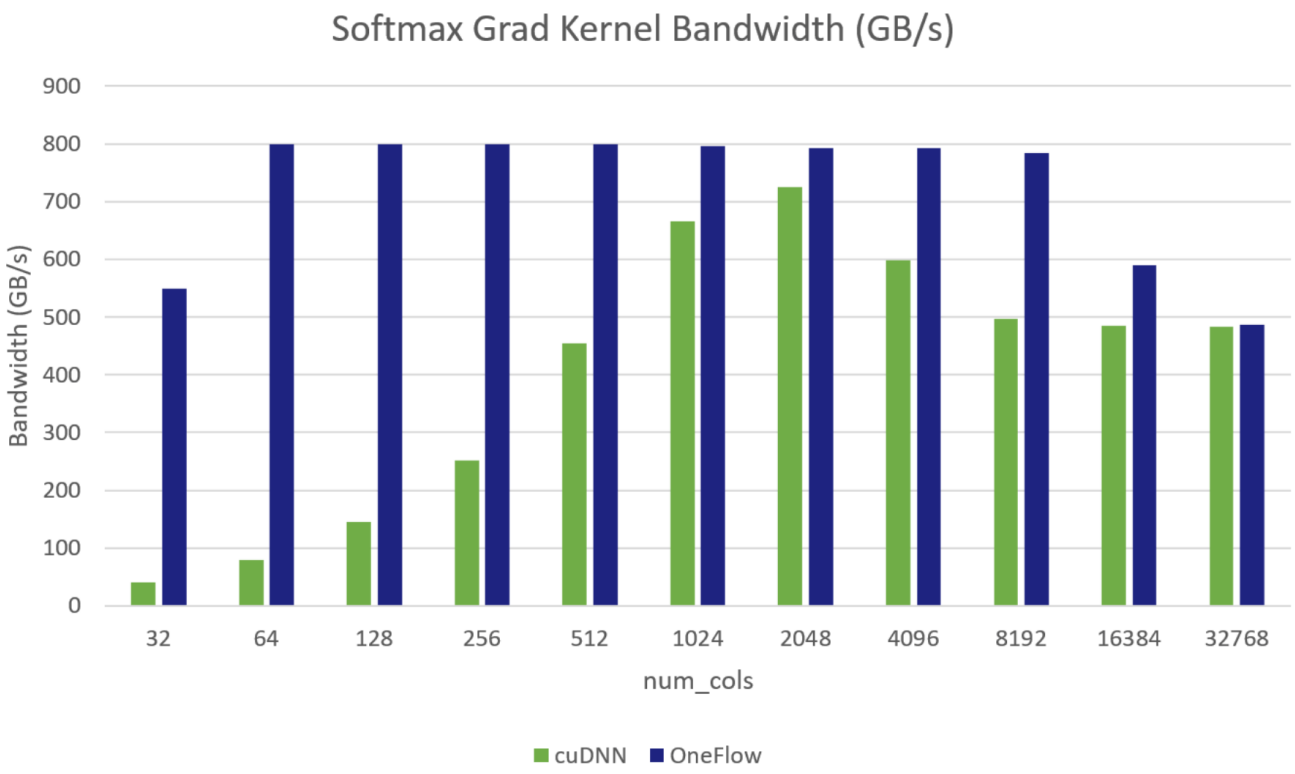
OneFlow 与 cuDNN 的对比结果

我们对OneFlow深度优化后的Softmax CUDA Kernel 与 cuDNN中的Softmax Kernel的访存带宽进行了对比测试，测试结果如下：

Softmax Kernel Bandwidth:



Softmax Grad Kernel Bandwidth:



其中测试环境是 GeForce RTX™ 3090 GPU，数据类型是half，Softmax的Shape = (49152, num_cols)，其中49152 = 32 * 12 * 128，是BERT-base网络中的attention Tensor的前三维，我们固定了前三维，将最后一维动态变化，测试了从32到32768不同大小的Softmax前向Kernel和反向Kernel的有效显存带宽。从上面两张图中可以看出OneFlow在多数情况下都可以逼近理论带宽（800GB/s左右，与cudaMemcpy的访存带宽相当。官方公布的理论带宽为936GB/S）。并且在所有情况下，OneFlow的CUDA Kernel的有效访存带宽都优于cuDNN的实现。

OneFlow深度优化Softmax CUDA Kernel的技巧

Softmax 函数的输入形状为:(num_rows, num_cols)，num_cols的变化，会对有效带宽产生影响；因为，没有一种 **通用** 的优化方法可以实现在所有 num_cols的情况下都是传输最优的。所以，在 OneFlow 中采用分段函数优化SoftmaxKernel：对于不同 num_cols范围，选择不同的实现，以在所有情况下都能达到较高的有效带宽。见 Optimize softmax cuda kernel

OneFlow 中分三种实现，分段对 softmax 进行优化：

(1) 一个 **Warp** 处理一行的计算，适用于 num_cols ≤ 1024 情况

> *硬件上并行执行的32个线程称之为一个warp，同一个warp的32个thread执行同一条指令。 warp是GPU调度执行的基本单元*

(2) 一个 Block 处理一行的计算，借助 Shared Memory 保存中间结果数据，适用于需要的 Shared Memory 资源满足 Kernel Launch 的可启动条件的情况，在本测试环境中是 1024 < num_cols ≤ 4096

(3) 一个 Block 处理一行的计算，不使用 Shared Memory，重复读输入 x，适用于不支持(1)、(2)的情况

下面以前向计算为例分别对三种实现进行介绍：

实现1：每个Warp处理一行或两行元素。

每个 Warp 处理一行或两行元素，每行的Reduce操作 需要做 Warp 内的 Reduce 操作，我们实现 WarpAllReduce 来完成 Warp 内各线程间的求 Global Max 和 Global Sum 操作，WarpAllReduce 是利用Warp级别原语 __shfl_xor_sync 实现的，代码如下。

```
template<template<typename> typename ReductionOp, typename T>
__inline__ __device__ T WarpAllReduce(T val) {
  for (int mask = kWarpSize / 2; mask > 0; mask /= 2) {
    val = ReductionOp<T>()(val, __shfl_xor_sync(0xffffffff, val, mask));
  }
  return val;
}
```

SoftmaxWarpImpl的实现有以下几个模板参数：

LOAD、STORE分别代表输入输出，使用`load.template load<pack_size>(ptr, row_id, col_id);`和`store.template store<pack_size>(ptr, row_id, col_id);` 进行读取和写入。使用LOAD和STORE有两个好处：1、可以在CUDA Kernel中只关心计算类型ComputeType，而不用关心具体的数据类型T。2、只需要加几行代码就可以快速支持Softmax和其他Kernel Fuse，减少带宽需求，提升整体性能。普通的SoftmaxKernel直接使用DirectLoad和DirectStore，FusedSoftmaxKernel如FusedScaleSoftmaxDropoutKernel只需要定义一个ScaleLoad结构和一个DropoutStore结构用于对输入x做Scale预处理和对输出y做Dropout后处理。

ComputeType代表计算类型。pack_size代表向量化访存操作的pack元素的个数，我们将几个元素pack起来读写，提升带宽利用率。cols_per_thread代表每个线程处理的元素个数。

thread_group_width代表处理元素的线程组的宽度，当cols > pack_size * warp_size时，thread_group_width就是warp_size，即32。当cols < pack_size * warp_size时，就根据cols大小用1/2个warp或1/4个warp来处理每行的元素。采用更小的thread_group_width后，WarpAllReduce需要执行的轮次也相应减少。

rows_per_access代表每个thread_group一次处理的行数，当cols较小，thread_group_width不是warp_size 32时，若rows能被2整除，我们就让每个线程处理2行来增加指令并行度，从而提升性能。

padding代表当前是否做了padding，若cols不是warp_size的整数倍，我们会把它padding到最近的整数倍处理。

algorithm代表使用的算法，可选项有Algorithm::kSoftmax或Algorithm::kLogSoftmax。

CUDA Kernel执行的主体循环逻辑如下，首先根据 num_cols信息算出每个线程要处理的 cols_per_thread，每个线程分配`rows_per_access * cols_per_thread`大小的寄存器，将输入x读到寄存器中，后续计算均从寄存器中读取。

理论上来说，以 Warp为单位处理一行元素的速度是最快的，但是由于需要使用寄存器将输入x缓存起来，而寄存器资源是有限的，并且在 num_cols>1024 情况下，使用(2)的 shared memory 方法已经足够快了，因此仅在 num_cols≤1024 时采用 Warp 的实现。

```
template<typename LOAD, typename STORE, typename ComputeType, int pack_size, int cols_per_thread,
         int thread_group_width, int rows_per_access, bool padding, Algorithm algorithm>
__global__ void SoftmaxWarpImpl(LOAD load, STORE store, const int64_t rows, const int64_t cols) {
  static_assert(cols_per_thread % pack_size == 0, "");
  static_assert(thread_group_width <= kWarpSize, "");
  static_assert(kWarpSize % thread_group_width == 0, "");
  constexpr int num_packs = cols_per_thread / pack_size;
  assert(cols <= cols_per_thread * thread_group_width);
```

```
  ComputeType buf[rows_per_access][cols_per_thread];
  const int global_thread_group_id = blockIdx.x * blockDim.y + threadIdx.y;
  const int num_global_thread_group = gridDim.x * blockDim.y;
  const int lane_id = threadIdx.x;
  for (int64_t row = global_thread_group_id * rows_per_access; row < rows;
       row += num_global_thread_group * rows_per_access) {
    ComputeType thread_max[rows_per_access];
#pragma unroll
    for (int row_id = 0; row_id < rows_per_access; ++row_id) {
      thread_max[row_id] = -Inf<ComputeType>();
      ComputeType* row_buf = buf[row_id];
#pragma unroll
      for (int pack_id = 0; pack_id < num_packs; ++pack_id) {
        const int col = (pack_id * thread_group_width + lane_id) * pack_size;
        if (!padding || col < cols) {
          load.template load<pack_size>(row_buf + pack_id * pack_size, row + row_id, col);
#pragma unroll
          for (int i = 0; i < pack_size; ++i) {
            thread_max[row_id] = max(thread_max[row_id], row_buf[pack_id * pack_size + i]);
          }
        } else {
#pragma unroll
          for (int i = 0; i < pack_size; ++i) {
            row_buf[pack_id * pack_size + i] = -Inf<ComputeType>();
          }
        }
      }
    }
    ComputeType warp_max[rows_per_access];
#pragma unroll
    for (int row_id = 0; row_id < rows_per_access; ++row_id) {
      warp_max[row_id] = WarpAllReduce<MaxOp, ComputeType, thread_group_width>(thread_max[row_id]);
    }
    ComputeType thread_sum[rows_per_access];
#pragma unroll
    for (int row_id = 0; row_id < rows_per_access; ++row_id) {
      thread_sum[row_id] = 0;
      ComputeType* row_buf = buf[row_id];
#pragma unroll
      for (int i = 0; i < cols_per_thread; ++i) {
        if (algorithm == Algorithm::kSoftmax) {
          row_buf[i] = Exp(row_buf[i] - warp_max[row_id]);
          thread_sum[row_id] += row_buf[i];
        } else if (algorithm == Algorithm::kLogSoftmax) {
          row_buf[i] -= warp_max[row_id];
          thread_sum[row_id] += Exp(row_buf[i]);
        } else {
          __trap();
        }
      }
    }
    ComputeType warp_sum[rows_per_access];
#pragma unroll
    for (int row_id = 0; row_id < rows_per_access; ++row_id) {
      warp_sum[row_id] = WarpAllReduce<SumOp, ComputeType, thread_group_width>(thread_sum[row_id]);
    }
#pragma unroll
    for (int row_id = 0; row_id < rows_per_access; ++row_id) {
      ComputeType* row_buf = buf[row_id];
#pragma unroll
      for (int i = 0; i < cols_per_thread; ++i) {
        if (algorithm == Algorithm::kSoftmax) {
          row_buf[i] = Div(row_buf[i], warp_sum[row_id]);
```

```
        } else if (algorithm == Algorithm::kLogSoftmax) {
          row_buf[i] -= Log(warp_sum[row_id]);
        } else {
          __trap();
        }
      }
#pragma unroll
      for (int i = 0; i < num_packs; ++i) {
        const int col = (i * thread_group_width + lane_id) * pack_size;
        if (!padding || col < cols) {
          store.template store<pack_size>(row_buf + i * pack_size, row + row_id, col);
        }
      }
    }
  }
}
```

实现2： 一个Block处理一行元素

一个 Block 处理一行元素，行 Reduce 需要做 Block 内的 Reduce 操作，需要做 Block 内线程同步，利用 BlockAllReduce 完成 Warp 内各线程间的求 Global Max 和 Global Sum 操作。BlockAllReduce 是借助 Cub 的 BlockReduce 方法实现的，代码如下：

```
template<template<typename> typename ReductionOp, typename T, int block_size>
__inline__ __device__ T BlockAllReduce(T val) {
  typedef cub::BlockReduce<T, block_size> BlockReduce;
  __shared__ typename BlockReduce::TempStorage temp_storage;
  __shared__ T result_broadcast;
  T result = BlockReduce(temp_storage).Reduce(val, ReductionOp<T>());
  if (threadIdx.x == 0) { result_broadcast = result; }
  __syncthreads();
  return result_broadcast;
}
```

CUDA Kernel的各个模板参数在(1)中做了详细介绍，执行的主体循环逻辑如下，根据 num_cols算出需要的 Shared Memory 大小作为 Launch Kernel 参数，借助 Shared Memory 保存输入，后续计算直接从 Shared Memory 读取。

由于 SM 内的 Shared Memory 资源同样有限，因此当 num_cols超过一定范围，kernel 启动时申请 Shared Memory 超过最大限制，就会出现无法启动的问题，因此，仅在调用 cudaOccupancyMaxActiveBlocksPerMultiprocessor 返回值大于0时采用 Shared Memory 方案。

此外，需要注意的是，由于 Block 内线程要做同步，当 SM 中正在调度执行的一个 Block 到达同步点时，SM 内可执行 Warp 逐渐减少，若同时执行的 Block 只有一个，则 SM 中可同时执行的 Warp 会在此时逐渐降成0，会导致计算资源空闲，造成浪费，若此时同时有其他 Block 在执行，则在一个 Block 到达同步点时仍然有其他 Block 可以执行。当 block_size 越小时，SM 可同时调度的 Block 越多，因此在这种情况下 block_size 越小越好。但是当在调大 block_size，SM 能同时调度的 Block 数不变的情况下，block_size 应该是越大越好，越大就有越好的并行度。因此代码中在选择 block_size 时，对不同 block_size 都计算了 cudaOccupancyMaxActiveBlocksPerMultiprocessor，若结果相同，使用较大的 block_size。

```
template<typename LOAD, typename STORE, typename ComputeType, int pack_size, int block_size,
         Algorithm algorithm>
__global__ void SoftmaxBlockSMemImpl(LOAD load, STORE store, const int64_t rows,
                                     const int64_t cols) {
  extern __shared__ __align__(sizeof(double)) unsigned char shared_buf[];
  auto* buf = reinterpret_cast<ComputeType*>(shared_buf);
  const int tid = threadIdx.x;
  assert(cols % pack_size == 0);
```

```
  const int num_packs = cols / pack_size;
  for (int64_t row = blockIdx.x; row < rows; row += gridDim.x) {
    ComputeType thread_max = -Inf<ComputeType>();
    for (int pack_id = tid; pack_id < num_packs; pack_id += block_size) {
      ComputeType pack[pack_size];
      load.template load<pack_size>(pack, row, pack_id * pack_size);
#pragma unroll
      for (int i = 0; i < pack_size; ++i) {
        buf[i * num_packs + pack_id] = pack[i];
        thread_max = max(thread_max, pack[i]);
      }
    }
    const ComputeType row_max = BlockAllReduce<MaxOp, ComputeType, block_size>(thread_max);
    ComputeType thread_sum = 0;
    for (int col = tid; col < cols; col += block_size) {
      if (algorithm == Algorithm::kSoftmax) {
        const ComputeType exp_x = Exp(buf[col] - row_max);
        buf[col] = exp_x;
        thread_sum += exp_x;
      } else {
        const ComputeType x = buf[col] - row_max;
        buf[col] = x;
        thread_sum += Exp(x);
      }
    }
    const ComputeType row_sum = BlockAllReduce<SumOp, ComputeType, block_size>(thread_sum);
    for (int pack_id = tid; pack_id < num_packs; pack_id += block_size) {
      ComputeType pack[pack_size];
#pragma unroll
      for (int i = 0; i < pack_size; ++i) {
        if (algorithm == Algorithm::kSoftmax) {
          pack[i] = Div(buf[i * num_packs + pack_id], row_sum);
        } else if (algorithm == Algorithm::kLogSoftmax) {
          pack[i] = buf[i * num_packs + pack_id] - Log(row_sum);
        } else {
          __trap();
        }
        thread_max = max(thread_max, pack[i]);
      }
      store.template store<pack_size>(pack, row, pack_id * pack_size);
    }
  }
}
```

实现3：一个Block处理一行的元素，不使用Shared Memory，重复读输入x

和实现2一样，仍然是一个 Block 处理一行元素，不同的是，不再用 Shared Memory 缓存输入x，而是在每次计算时重新读输入 x，这种实现没有最大 num_cols的限制，可以支持任意大小。

此外，需要注意的是，在这种实现中，block_size 应该设越大越好，block_size 越大，SM 中能同时并行执行的 Block 数就越少，对 cache 的需求就越少，就有更多机会命中 Cache，多次读x不会多次访问 Global Memory，因此在实际测试中，在能利用 Cache 情况下，有效带宽不会因为读3次x而降低几倍。

```
template<typename LOAD, typename STORE, typename ComputeType, int pack_size, int block_size,
         Algorithm algorithm>
__global__ void SoftmaxBlockUncachedImpl(LOAD load, STORE store, const int64_t rows,
                                         const int64_t cols) {
  const int tid = threadIdx.x;
  assert(cols % pack_size == 0);
  const int num_packs = cols / pack_size;
  for (int64_t row = blockIdx.x; row < rows; row += gridDim.x) {
    ComputeType thread_max = -Inf<ComputeType>();
```

```
    for (int pack_id = tid; pack_id < num_packs; pack_id += block_size) {
      ComputeType pack[pack_size];
      load.template load<pack_size>(pack, row, pack_id * pack_size);
#pragma unroll
      for (int i = 0; i < pack_size; ++i) { thread_max = max(thread_max, pack[i]); }
    }
    const ComputeType row_max = BlockAllReduce<MaxOp, ComputeType, block_size>(thread_max);
    ComputeType thread_sum = 0;
    for (int pack_id = tid; pack_id < num_packs; pack_id += block_size) {
      ComputeType pack[pack_size];
      load.template load<pack_size>(pack, row, pack_id * pack_size);
#pragma unroll
      for (int i = 0; i < pack_size; ++i) { thread_sum += Exp(pack[i] - row_max); }
    }
    const ComputeType row_sum = BlockAllReduce<SumOp, ComputeType, block_size>(thread_sum);
    for (int pack_id = tid; pack_id < num_packs; pack_id += block_size) {
      ComputeType pack[pack_size];
      load.template load<pack_size>(pack, row, pack_id * pack_size);
#pragma unroll
      for (int i = 0; i < pack_size; ++i) {
        if (algorithm == Algorithm::kSoftmax) {
          pack[i] = Div(Exp(pack[i] - row_max), row_sum);
        } else if (algorithm == Algorithm::kLogSoftmax) {
          pack[i] = (pack[i] - row_max) - Log(row_sum);
        } else {
          __trap();
        }
      }
      store.template store<pack_size>(pack, row, pack_id * pack_size);
    }
  }
}
```

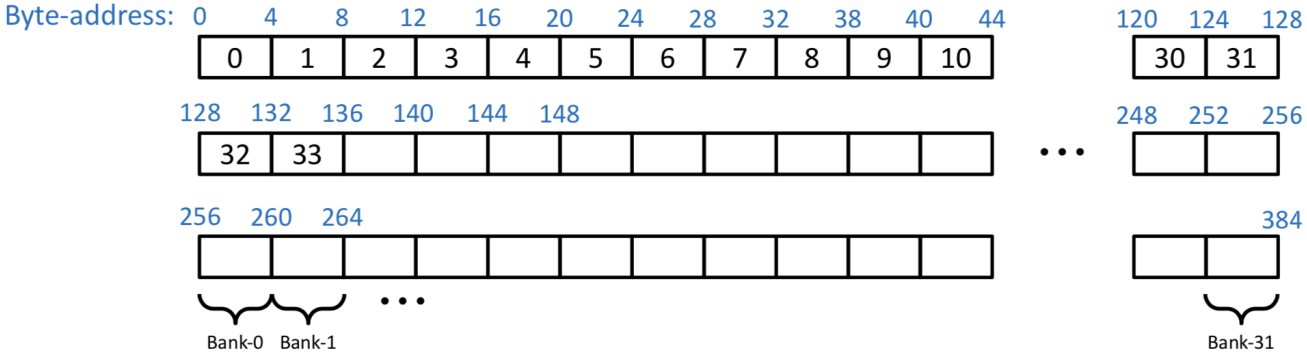
公共优化技巧

除了以上介绍的针对 softmax 的分段优化技巧外，OneFlow 在 softmax 实现中，还使用了一些通用的公共优化技巧，他们不仅应用于 softmax 中，也应用在其它 kernel 实现中。在此介绍两种：

1、将 Half 类型 pack 成 Half2 进行存取，在不改变延迟情况下提高指令吞吐，类似 CUDA template for element-wise kernels 的优化

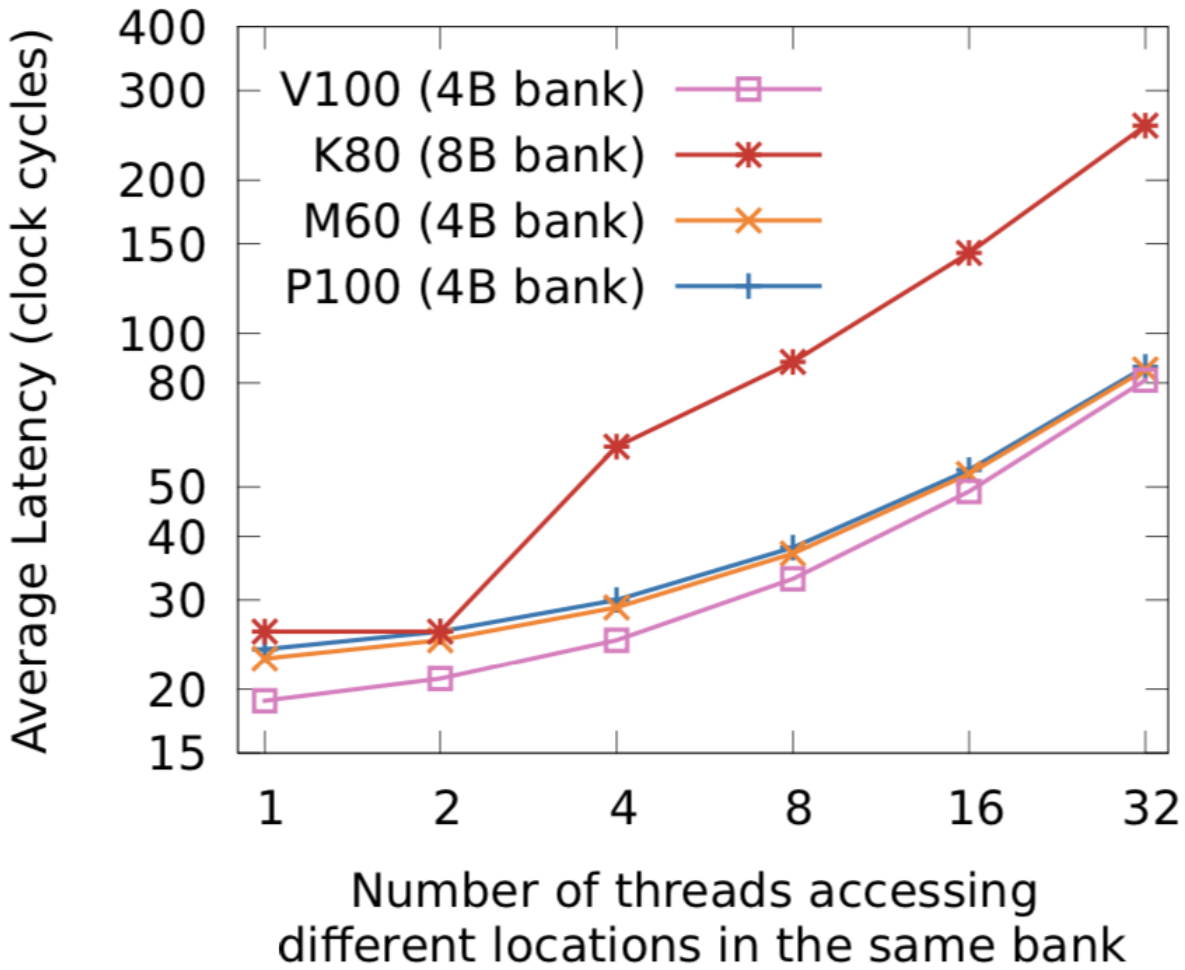
2、Shared Memory 中的 Bank Conflicts

CUDA 将 Shared Memory 按照4字节或8字节大小划分到32个 bank 中，对于 Volta 架构是4字节，这里以4字节为例，如下图所示，0-128 bytes地址分别在bank 0-31中，128-256也分别在 bank0-31 中。



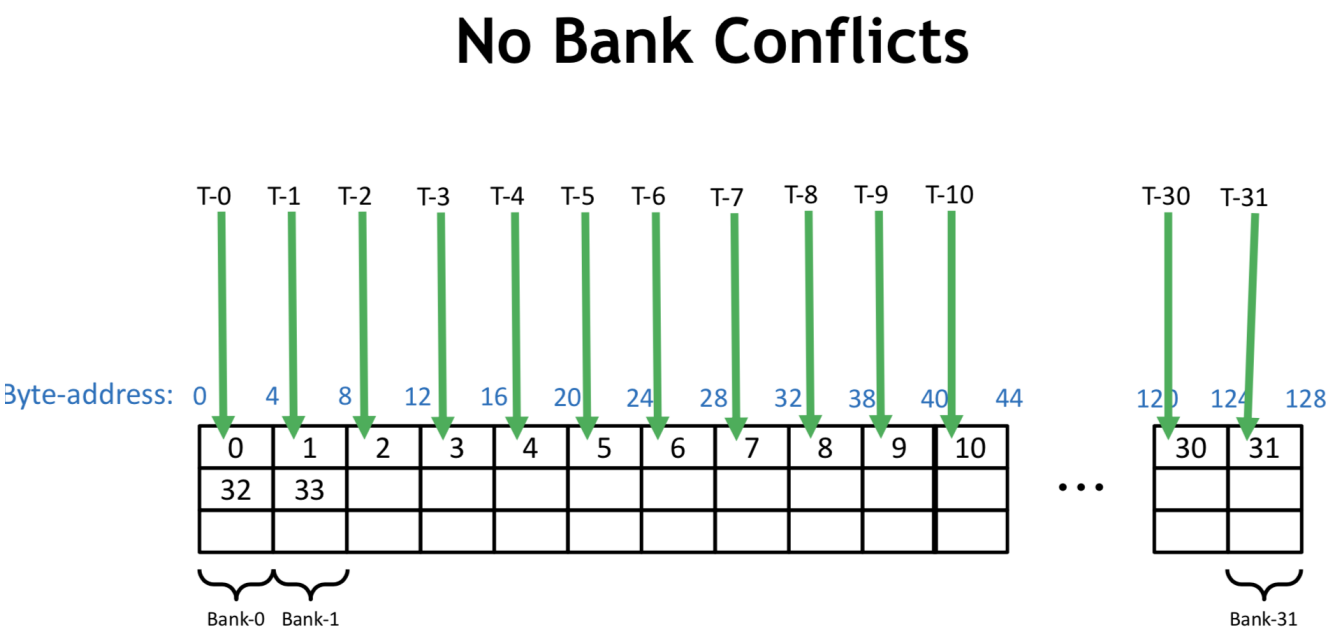
注：此图及以下 Bank Conflicts 图片来自VOLTA Architecture and performance optimization

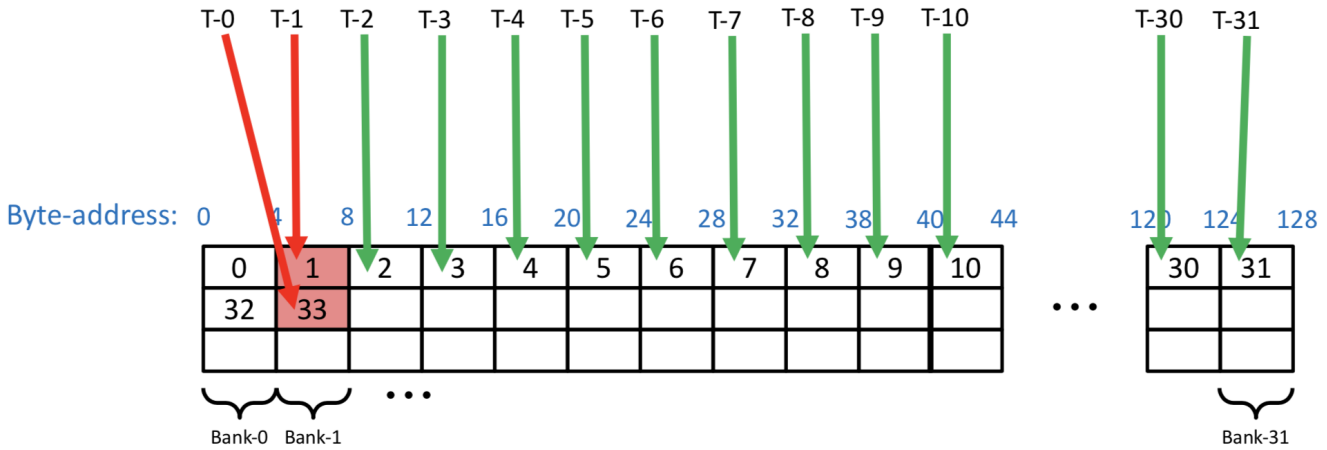
当在一个 Warp 内的不同线程访问同一个 bank 的不同地址，就会出现 Bank Conflicts。当发生 Bank Conflicts 时，线程间只能顺序访问，增加延迟，下图是一个 Warp 中 n 个线程同时访问同一个 bank 中的不同地址时的延迟情况。



注：图来自Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

Bank Conflicts 的几种情况：





若一个 Warp 内每个线程读4个字节，顺序访问，则不会有 Bank Conflicts，若一个 Warp 内每个线程读8个字节，则 Warp 内0号线程访问的地址在第0和第1个 bank，1号线程访问的地址在第2和第3个 bank，以此类推，16号线程访问地址又在第0和第1个 bank内，和0号线程访问了同一个bank的不同地址，此时即产生了 Bank Conflicts。

在前文的实现(2)中，给 Shared memory 赋值过程中，若采用下面方法，当 pack size=2，每个线程写连续两个4 byte 地址，就会产生 Bank Conflicts。

```
#pragma unroll
      for (int j = 0; j < pack_size; ++j) {
        buf[pack_id * pack_size * j] = pack[j];
        thread_max = max(thread_max, pack[j]);
      }
```

因此，在实现(2)中，对Shared memory采用了新的内存布局，避免了同一个Warp访问相同bank的不同地址，避免了Bank Conflicts。

```
#pragma unroll
      for (int j = 0; j < pack_size; ++j) {
        buf[num_packs * j + pack_id] = pack[j];
        thread_max = max(thread_max, pack[j]);
      }
```

参考资料：

Using CUDA Warp-Level Primitives | NVIDIA Developer Blog

CUDA Pro Tip: Increase Performance with Vectorized Memory Access

Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking

VOLTA Architecture and performance optimization

欢迎下载体验OneFlow深度学习框架：