cute 矩阵乘法

Cute 中一些常见的操作

1. partitioning-a-tensor

https://github.com/NVIDIA/cutlass/blob/v3.5.0/media/docs/cute/03_tensor.md#partitioning-a-tensor

这部分主要讲解如何划分一个 tensor。 在 cuda 中，往往需要把一个大的问题划分为很多更小的问题（tile），让多个线程同时并行处理这些小问题。

1.1 local_tile

假设有一个 8*8 的矩阵，按照 4*4 的块进行划分，则可以划分为 4 块。

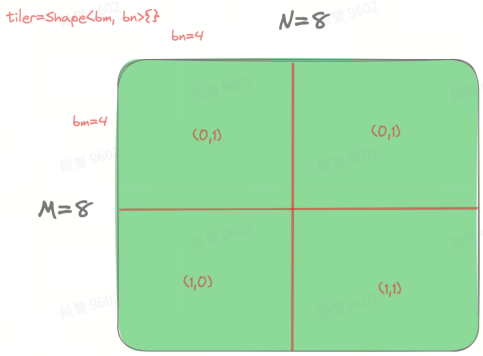
```
TEST(LOCAL_TILE, LOCAL_TILE) {
  thrust::host_vector<int> h_A(8*8);
  for (int j = 0; j < 8*8; ++j) h_A[j] = int(j);

  Tensor A = make_tensor(h_A.data(), make_shape(8,8));  // (8,8) default col-major
  print_tensor(A);
  auto tiler = Shape<_4,_4>{};
  Tensor cta_a = local_tile(A, tiler, make_coord(0, 1));
  print_tensor(cta_a);

  print(cta_a[(make_coord(0,1))]); print("\n");
  print(cta_a[2]); print("\n");
}
```

划分之后，通过二维坐标获取想要的分块，如：

```
Tensor cta_a = local_tile(A, tiler, make_coord(0, 1));
```



然后通过二维坐标或者一维坐标访问对应的元素。

上述测试代码运行结果为：

```
ptr[32b](0x55a395ed59f0) o (8,8):(_1,8):
    0    8   16   24   32   40   48   56
    1    9   17   25   33   41   49   57
    2   10   18   26   34   42   50   58
    3   11   19   27   35   43   51   59
    4   12   20   28   36   44   52   60
    5   13   21   29   37   45   53   61
    6   14   22   30   38   46   54   62
    7   15   23   31   39   47   55   63
ptr[32b](0x55a395ed5a70) o (_4,_4):(_1,8):
   32   40   48   56
   33   41   49   57
   34   42   50   58
   35   43   51   59
40
34
```

也可以一次获取多个划分的 tile：比如下面这个句话可以获取到对应坐标为(0,0), (0,1)的两个tile

`Tensor cta_a = local_tile(A, tiler, make_coord(0, _));`

访问某个tile中的某个元素时可以通过下面代码获取：

`print(cta_a(_,_,0)[(make_coord(0,1))]); print("\n");`

`print(cta_a(_,_,0)[2]); print("\n");`

在很多代码中，通过另一个接口使用 local_tile api

```
TEST(LOCAL_TILE_2, LOCAL_TILE_2) {
  thrust::host_vector<int> h_A(8*8);
  for (int j = 0; j < 8*8; ++j) h_A[j] = int(j);

  Tensor A = make_tensor(h_A.data(), make_shape(8,8), make_stride(8, 1));  // row-major
  print_tensor(A);
  auto tiler = Shape<_4,_1,_4>{};
  auto cta_coord = make_coord(0, 0, _);

  Tensor cta_a = local_tile(A, tiler, cta_coord, Step<_1, X,_1>{});
  /*
```

```
    // 上述代码等同于:
    Tensor cta_a = local_tile(A, Shape<_4, _4>{}, make_coord(0, _));
    */
    print_tensor(cta_a);
}
```

1.2 local_partition

1.1 中每个 tile 是一个 （4，4）小 tensor， 假如现在想用一个 CTA 来处理这个 tile，这个 CTA 里面可能有 4个thread，可能有16个tread，那每个 thread 需要处理那些元素呢。（实际cuda 编程中，CTA 里面一般有256个thread）

参考下面的 demo：

```
TEST(LOCAL_PARTITION, LOCAL_PARTITION) {
  thrust::host_vector<int> h_A(4*4);
  for (int j = 0; j < 4*4; ++j) h_A[j] = int(j);

  Tensor A = make_tensor(h_A.data(), make_shape(4,4), make_stride(4, 1)); // row-major
  print_tensor(A);
  auto bM = Int<2>{};
  auto bN = Int<2>{};
  auto CTA_layout = make_layout(make_shape(bM, bN), LayoutRight{}); // row-major

  Tensor thread_tensor = local_partition(A, CTA_layout, 1);
  //上一行代码等同于: Tensor thread_tensor = local_partition(A, CTA_layout, 1, Step<_1,_1>{});
  print_tensor(thread_tensor);
}
```

输出结果为：

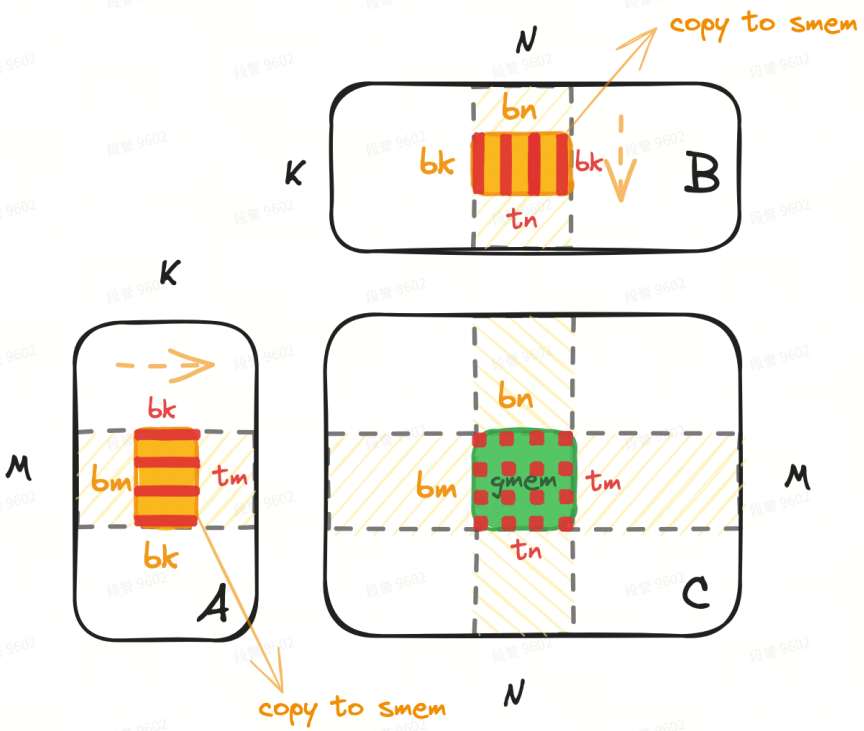
```
ptr[32b](0x557ac2a3b300) o (4,4):(4,1):
    0    1    2    3
    4    5    6    7
    8    9   10   11
   12   13   14   15
ptr[32b](0x557ac2a3b304) o (2,2):(8,2):
    1    3
    9   11
```

注意的两点是：

1. 线程处理的数据是交错的
2. CTA_layout 中的 LayoutRight{} or LayoutLeft() 不改变数据在各个线程之间的分配，只会改变获取不同 thread_tensor 的 idx(如demo中的1)

sgemm_1.cu

这是一个最简单实现的 gemm，它的实现逻辑可以用下图概括：



一个 CTA 只负责计算 C 中绿色的 bm*bn 部分， 一个 Thread 只负责计算 （bm，bn）中的 （tm，tn）部分，但是注意的是， （tm， tn）是分散的，即线程之间处理的数据是交错的。黄色部分表示这块区域的数据会被加载到 shared memory 中。

cute中，每个CTA需要处理的部分和每个thread需要处理的部分都可以抽象为一个 tensor。

代码逻辑如下：

```
# k 纬度的循环
for (i = 0;  i < K/bk; i++) {
   1. 将 A 中的 (bm,bk) 和 B 中 (bk,bn) copy 到shared memory
   2. 每个 thread 负责计算 （tm， bk） * （bk，tn） = （tm，tn）
}
```

以 gemm_tn 为例：， A:（m,k）row-major， B(n,k) row-major == B(k, n) col-major

CTA_Partition

该算法中，cta_tiler, shared_memory, 以及 CTA 线程组织方式设计如下：

```
// Define CTA tile sizes (static)
  auto bM = Int<128>{};
  auto bN = Int<128>{};
  auto bK = Int<  8>{};
  auto cta_tiler = make_shape(bM, bN, bK);                   // (bm, bn, bk)

  // Define the smem layouts (static)
  auto sA = make_layout(make_shape(bM,bK), LayoutRight{});   // (bm,bk) -> smem_idx; k-major
  auto sB = make_layout(make_shape(bN,bK), LayoutRight{});   // (bn,bk) -> smem_idx; k-major
  auto sC = make_layout(make_shape(bM, bN));                 // (bm,bn) -> smem_idx; m-major

  // Define the thread layouts (static)
  auto tA = make_layout(make_shape(Int<32>{}, Int< 8>{}), LayoutRight{});
  auto tB = make_layout(make_shape(Int<32>{}, Int< 8>{}), LayoutRight{});
  auto tC = make_layout(make_shape(Int<16>{}, Int<16>{}));             // (tm, tn) -> thr_idx; m-major

  dim3 dimGrid(size(ceil_div(M, bM)),
               size(ceil_div(N, bN)));
  dim3 dimBlock(size(tC));
```

需要注意一点是， demo 中CTA 中的线程有三种不同的组织方式，其实 tA， tB 是用来把数据从 global memory 搬运到 shared memory的协作方式，tc 才是具体计算的协作方式，但不管线程之间的协作方式怎么样，CTA 中的线程总数一般是相等的。

SMEM Tensor

接下来，在 kernel 中我们需要划分每个 CTA 所需的那部分tensor，以及 shared memory tensor（demo 中没有给申请sc，而是直接把计算结果往 global memory 中写）：

```
// Represent the full tensors
  Tensor mA = make_tensor(make_gmem_ptr(A), select<0,2>(shape_MNK), dA); // (M,K)
  Tensor mB = make_tensor(make_gmem_ptr(B), select<1,2>(shape_MNK), dB); // (N,K)
  Tensor mC = make_tensor(make_gmem_ptr(C), select<0,1>(shape_MNK), dC); // (M,N)

  // Get the appropriate blocks for this thread block
  auto cta_coord = make_coord(blockIdx.x, blockIdx.y, _);              // (m,n,k)

```

```
    // 这里 tile full tensors， 每个 CTA/thread_group 获取了自己需要处理的 subtensor
    Tensor gA = local_tile(mA, cta_tiler, cta_coord, Step<_1, X,_1>{});  // (BLK_M,BLK_K,k)   k=K/BLK_K
    Tensor gB = local_tile(mB, cta_tiler, cta_coord, Step< X,_1,_1>{});  // (BLK_N,BLK_K,k)   k=K/BLK_K
    Tensor gC = local_tile(mC, cta_tiler, cta_coord, Step<_1,_1, X>{});  // (BLK_M,BLK_N)

    // Shared memory buffers
    __shared__ TA smemA[cosize_v<ASmemLayout>];
    __shared__ TB smemB[cosize_v<BSmemLayout>];
    Tensor sA = make_tensor(make_smem_ptr(smemA), sA_layout);            // (BLK_M,BLK_K)
    Tensor sB = make_tensor(make_smem_ptr(smemB), sB_layout);            // (BLK_N,BLK_K)
```

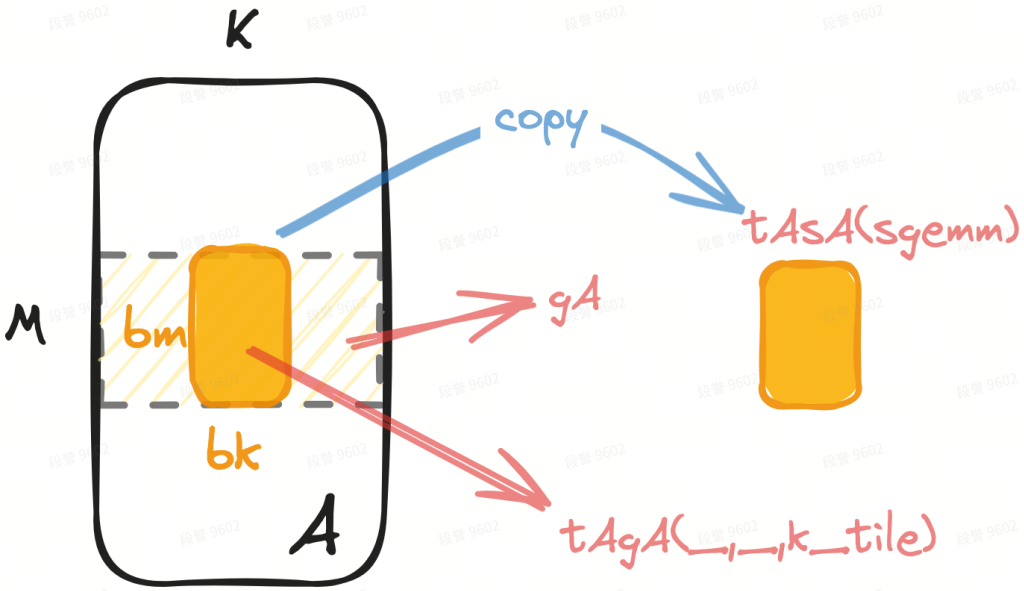
Copy partitioning

接下来，需要把 gA 和 gB 中的某个 tile 拷贝到 sA, sB 中， CTA 按照 tA, tB 的方式协作：

```
    // Partition the copying of A and B tiles across the threads
    Tensor tAgA = local_partition(gA, tA, threadIdx.x);                  // (THR_M,THR_K,k)  k=BLK_K/THR_K
    Tensor tAsA = local_partition(sA, tA, threadIdx.x);                  // (THR_M,THR_K)

    Tensor tBgB = local_partition(gB, tB, threadIdx.x);                  // (THR_N,THR_K,k)  k=BLK_K/THR_K
    Tensor tBsB = local_partition(sB, tB, threadIdx.x);                  // (THR_N,THR_K)
```

用图表示如下：

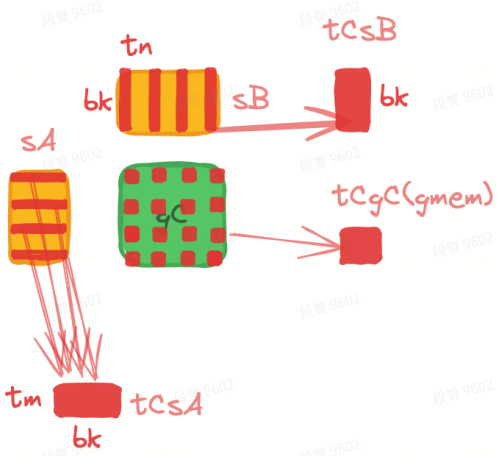


Math partitioning

最后，每次 k 维度的循环中，每个CTA 只需要从 sA， sB 中 load 数据，并把计算结果写入 gC 中对应的部分即可，CTA 中的线程按照 tC 要求的格式协作，同理，也需要划分并抽象出每一个 thread 对应的线程：

```
1    Tensor tCsA = local_partition(sA, tC, threadIdx.x, Step<_1, X>{});   //
   (THR_M,BLK_K)
2    // Partition sB (N,K) by the cols of tC
3    Tensor tCsB = local_partition(sB, tC, threadIdx.x, Step< X,_1>{});   //
   (THR_N,BLK_K)
4    // Partition gC (M,N) by the tile of tC
5    Tensor tCgC = local_partition(gC, tC, threadIdx.x, Step<_1,_1>{});   //
   (THR_M,THR_N)
```

用图表示如下：



最后，就是在 K 维度的循环做 gemm 运算：

```cpp
  auto K_TILE_MAX = size<2>(tAgA);
  for (int k_tile = 0; k_tile < K_TILE_MAX; ++k_tile)
  {
    // Copy gmem to smem with tA|tB thread-partitioned tensors
    copy(tAgA(_,_,k_tile), tAsA);      // A   (THR_M,THR_K) -> (THR_M,THR_K)
    copy(tBgB(_,_,k_tile), tBsB);      // B   (THR_N,THR_K) -> (THR_N,THR_K)

    // TUTORIAL: The above call to copy(tAgA(_,_,k_tile), tAsA) is equivalent to
    //   Tensor tAgAk = tAgA(_,_,k_tile);
    //   CUTE_UNROLL
    //   for (int i = 0; i < size(tAsA); ++i) {
    //     tAsA(i) = tAgAk(i);
    //   }

    cp_async_fence();        // Label the end of (potential) cp.async instructions
    cp_async_wait<0>();      // Sync on all (potential) cp.async instructions
    __syncthreads();         // Wait for all threads to write to smem

    // Compute gemm on tC thread-partitioned smem
    gemm(tCsA, tCsB, tCrC);            // (THR_M,THR_N) += (THR_M,BLK_K) * (THR_N,BLK_K)

    // TUTORIAL: The above call to gemm(tCsA, tCsB, tCrC) is equivalent to
    //   CUTE_UNROLL
    //   for (int k = 0; k < size<1>(tCsA); ++k) {
    //     CUTE_UNROLL
    //     for (int m = 0; m < size<0>(tCrC); ++m) {
    //       CUTE_UNROLL
    //       for (int n = 0; n < size<1>(tCrC); ++n) {
    //         tCrC(m,n) += tCsA(m,k) * tCsB(n,k);
    //       }
    //     }
    //   }
    __syncthreads();         // Wait for all threads to read from smem
  }
```