DeepGEMM学习

核心优化

总结一下deepgemm用到的核心优化。

config选择

block size

限定block_m的范围是[64, 128, 256]，（须为64的倍数）。对于fp8而言，block_n的范围是[16, 24, 32 ,40, ..., 128, 144, 160]。两者的范围是由WGMMA的指令来限制的，WGMMA相关后面会讲。在具体选择best_block_m和best_block_n的时候根据以下的规则：1. 防止块过大，block_m和block_n必须至少有一个<128。2.选择产生最少waves的block_m和block_n。所谓wave，就是gpu所有sm调用一次threadblock。

3.在产生waves数相同的情况下，选择最后一次wave中对sm占用率最大的。

deepgemm固定block_k为128，因为deepseek中block量化的size是128x128，channelgroup量化的size是1x128。

num_stages和share_mem

num_stages从8到1遍历，越大越好，但要满足当前GPU的share_mem的size。如H20的share_mem size最大是233472Bytes。要load到share_mem的数据包括：1.num_stages个激活A的子矩阵[block_m, block_k]。2.num_stages个权重B的子矩阵[block_k, block_n]。3.num_stages个激活A的量化因子[block_m, 1]。4.权重B的量化因子[ceil_div(k, block_k)， 1 or 2]，scale_B选择一次性读入整个K维度的值。5.结果子矩阵[block_m, block_n]。6.num_stages x 2个barrier，每一对barrier用于一组consumer和producer的同步，后面会讲到。每个barrier的size是8bytes。

tma_multicast_config

deepgemm目前采用的多播策略是 在m > 512且sm数可以整除2时，在A上多播两次。也就是在同一个cluster(size = 2)内，一个block发出tma load A的子矩阵的指令，同在该cluster的另一个block也会接收到数据。

生成的best config会与输入参数关联起来，并被lru_cache住，下次调用相同参数时直接返回最佳config。

scheduler

launch

设置了最大dynamic_shared_mem_size。规定了gird--cluster--block层级的dim: <<<num_sms, num_tma_multicast, threads_per_sm>>>，num_sms是用到的gpu sm数量，设置成这个是为了**持久化内核**，这个后面会讲到。num_tma_multicast就是多播的数量，也就是2。thread_per_sm中固定有128个thread用于tma load，另外会有1或者2个用于wgmma的warpgroup，取决于block_m是否大于64，如果是64，则会有1个warpgroup，若大于64，则需要两个warpgroup来计算。每个warpgroup的线程数量是128。

注：cluster层级是compute capability >= 9.0才有的。

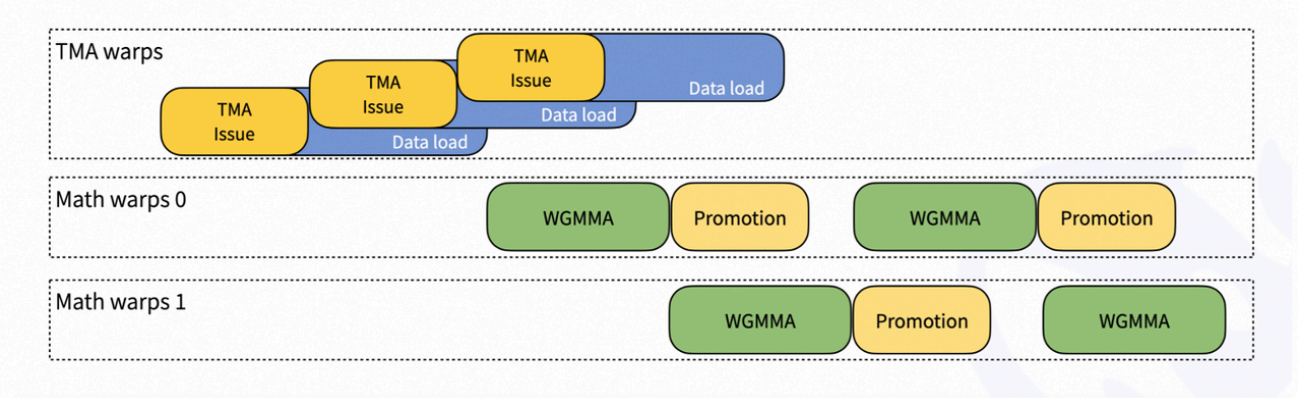
持久化内核

每个sm 只launch一个block，并让该block持续占据该sm，该block会依次处理多个子矩阵的计算。举个例子，假如sm数量是10，结果矩阵需要20个block，每个block负责一个子矩阵，才能全部算完。每个sm上的block0在完成对应的子矩阵计算后，sm并不会调度新的block1替代它，而是让block0继续计算下一块对应的子矩阵，这样一共有两个轮次的计算，都由相同的一批block来完成。

在让sm上的block找对应的子矩阵这里，用到了**swizzle**的技巧，用来提升L2 cache的命中率。若在A上多播，则以[ceil[m, block_m], 16]为group，‘Z’字形顺序计算子块。

consumer and producer

对于一个子块的计算，deepgemm用到了多stages的load/compute的pipeline。对于一个block，会有一个warpgroup 用于TMA load，1到2个warpgrouo(取决于block_m的大小)来用WGMMA指令做compute：



由于两个指令都是异步的，所以在这里设置了barrier来监管指令的完成情况：

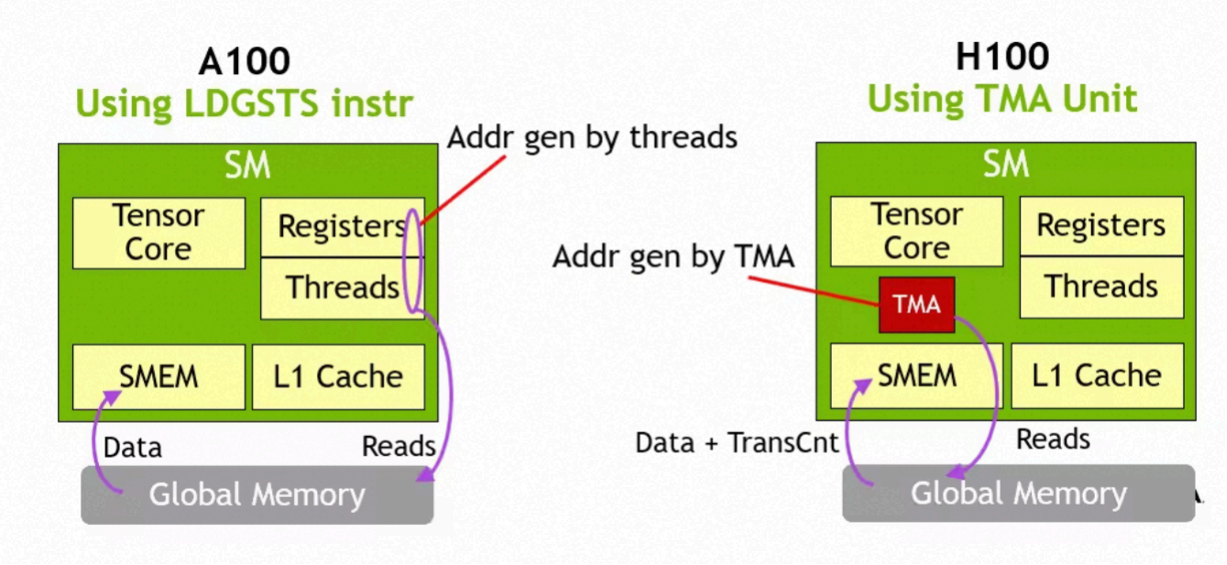
复制代码

```
// about load (producer)
```

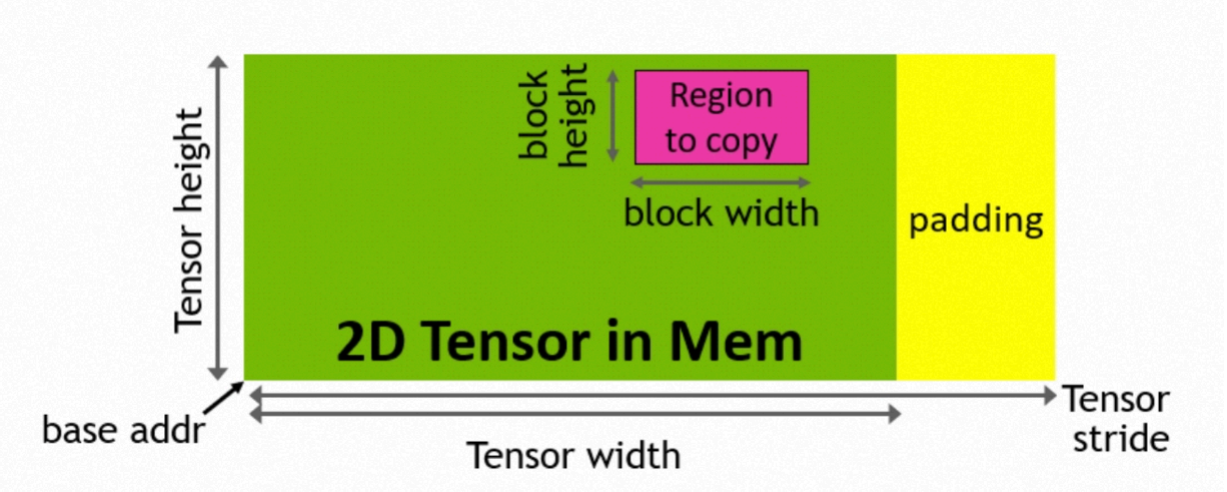
```
while (s in stage N) { // 0-127 threads do
  empty_barrier[s].wait()  // wait compute done
  tma_copy(A)  // issue TMA instuctions
  tma_copy(A_scale)
  tma_copy(B)
  full_barrier[s].arrive()
}
// about compute (consumer)
while (s in stage N) { // 128-255 threads do
  full_barrier[s].wait() // wait load done
  desc_a = make_smem_desc()
  desc_b = make_smem_desc()
  wgmma(desc_a, desc_b, accum) // issue wgmma instuction
  empty_barrier[s].arrive()
}
```

TMA

TMA是hopper架构(sm=90)引入的异步拷贝engine，可以用来将一维或多维的数据从global memory异步拷贝到cluster share memory (multicast)或cta share memory，或者反过来也可以。



tma区别于Ampere架构的async_copy的特点是计算拷贝数据的地址直接在TMA engine上完成，而async_copy需要thread计算出地址，面对一些不连续存储的数据，async_copy需要占用很多thread的寄存器来计算地址，而tma则不需要。在使用的时候要在launch kernel前创建好要load 数据的tensormap：




```
// Create the tensor descriptor.
CUresult res = cuTensorMapEncodeTiled(
  &tensor_map,                // CUtensorMap *tensorMap,
  CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_INT32,
  rank,                       // cuuint32_t tensorRank,
  tensor_ptr,                 // void *globalAddress,
  size,                       // const cuuint64_t *globalDim,
  stride,                     // const cuuint64_t *globalStrides,
  box_size,                   // const cuuint32_t *boxDim,
  elem_stride,                // const cuuint32_t *elementStrides,
  // Interleave patterns can be used to accelerate loading of values that
  // are less than 4 bytes long.
  CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
  // Swizzling can be used to avoid shared memory bank conflicts.
  CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_NONE,
  // L2 Promotion can be used to widen the effect of a cache-policy to a wider
  // set of L2 cache lines.
  CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
  // Any element that is outside of bounds will be set to zero by the TMA transfer.
  CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE
);
```

deepgemm对于A的子矩阵、B的子矩阵、scale_a以及结果子矩阵都是采用tma load的方式，会在host端初始化相关的tensor map，并作为__grid_constant__类型参数传入kernel。tensor map规定了要load的大矩阵的地址，形状，以及每次load 的bulk的形状，swizzle方式等。

deepgemm在kernel一开始prefetch A, B, scale_a, D的tensor map到global memory中。固定128个thread用于tma load。另外，scale_b的读取并不是tma完成，而是传统的线程读取，global memory-->register-->share memory，并与最终计算完成的block 用tma store back有一个overlap。

WGMMA

wgmma是warpgroup参与的tensorcore的mma计算，一个warpgroup是128个threads，对于int8类型，wgmma的计算尺寸固定m=64，k=32，n范围变化：

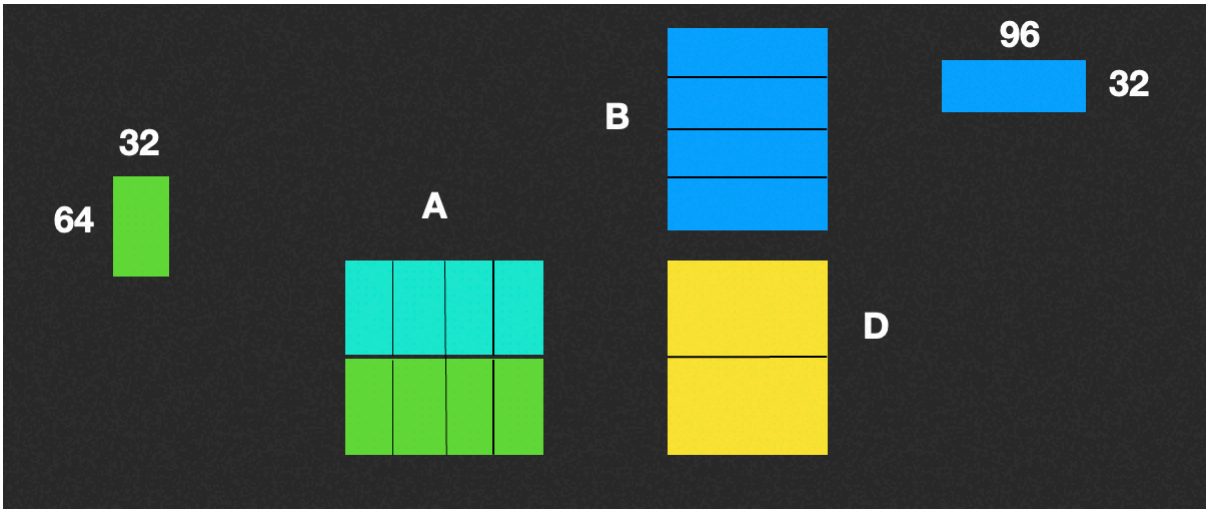
Integer - `.u8` / `.s8`	Dense	`.m64n8k32`, `.m64n16k32`, `.m64n24k32`, `.m64n32k32`, `.m64n48k32`, `.m64n64k32`, `.m64n80k32`, `.m64n96k32`, `.m64n112k32`, `.m64n128k32`, `.m64n144k32`, `.m64n160k32`, `.m64n176k32`, `.m64n192k32`, `.m64n208k32`, `.m64n224k32`, `.m64n240k32`, `.m64n256k32`

结果类型是int32类型：

Data-type	Multiplicands (A or B)	Accumulator (D)
Integer	both .u8 or both .s8	.s32

其他类型的计算尺寸以及结果类型可以在ptx文档中查阅。计算的结果存储在128个thread的寄存器里，每个thread需要存储结果的寄存器数量：m * n / 128。

假如block_n=96，那计算指令就是MMA_64x96x32_S32S8S8_SS_TN，SS表示两个子矩阵都是从share memory中读取。再假如block_m=128, 那么会有两个warp group用于wgmma，一个负责[0:63, block_k] x [block_k, 96]的矩阵计算，另一个负责[64:127, block_k] x [block_k, 96]的矩阵计算。再对于单个warpgroup来说，需要循环 block_k / 32次来累加计算结果，因为每次wgmma指令计算的shape是[64, 32] x [32, 96]：



promotion

为保证精度，promotion的计算在cudacore上进行：wgmma的结果accum * (scale_a * scale_b)。两个warpgroup在计算出结果后不需要同步，先计算完的warpgroup会直接用cudacore来计算promotion，而另一个warpgroup还在tensorcore计算wgmma。所以这里存在一个tensorcore和cudacore计算的overlap。

至于特定线程的accum要乘哪个scale_a和scale_b, 需要知道wgmma之后结果矩阵元素在线程寄存器中的排布。这里举个对齐的例子，block_m = 64， block_k = 128, block_n = 128，scale_a的shape是[64， 1]， scale_b就是一个元素，我们用到的wgmma指令是

wgmma.m64n128k128，那么计算完后结果元素的排布如下：

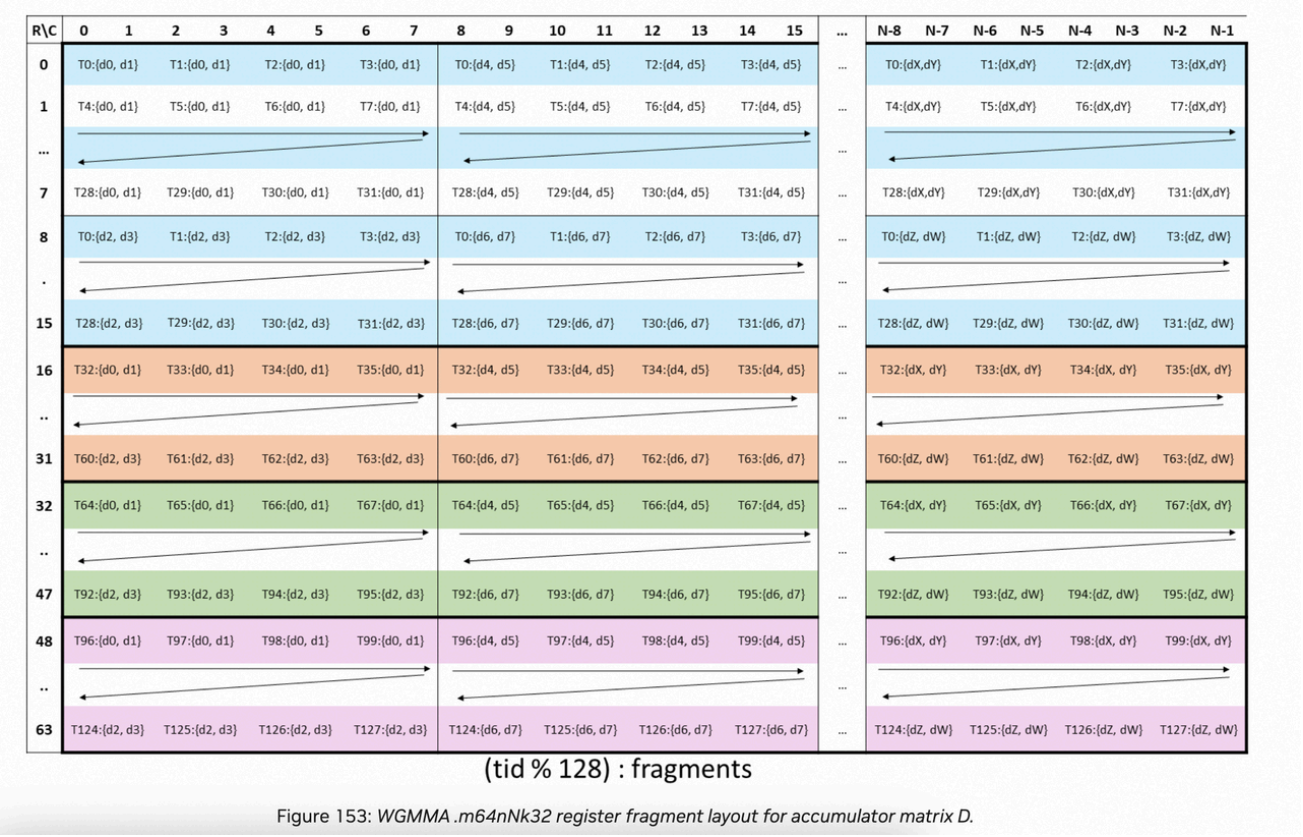


Figure 153: *WGMMA .m64nNk32 register fragment layout for accumulator matrix D.*

thread0的accum寄存器保存着：[d00, d01, d80, d81, d08, d09, d88, d89....] d[rc]: r是结果矩阵element所在的行数，c是结果矩阵element所在的列数。对于每个线程而言，要乘的scale_b都一样，但是scale_a会不同，比如对于thread0，d00和d01,d08和d09都对应着同一行，因此要乘相同的scale_a，但是d00和d80乘的scale_a就不同了，因为是不同行的，d80和d81乘的scale_a是相同的。故有如下的promotion计算代码：

```
#pragma unroll
for (uint32_t i = 0; i < WGMMA::kNumAccum / 4; ++ i) {
    // NOTES: for unrolled `num_former_iters` cases, we expect the compiler to automatically make it a constant
    bool predicate = kMustUseUniformedScaleB or i < num_former_iters;
    shifted_accum[i * 4 + 0] += (predicate ? scale_0_0 : scale_0_1) * static_cast<int32_t>(accum[i * 4 + 0]);
    shifted_accum[i * 4 + 1] += (predicate ? scale_0_0 : scale_0_1) * static_cast<int32_t>(accum[i * 4 + 1]);
    shifted_accum[i * 4 + 2] += (predicate ? scale_1_0 : scale_1_1) * static_cast<int32_t>(accum[i * 4 + 2]);
    shifted_accum[i * 4 + 3] += (predicate ? scale_1_0 : scale_1_1) * static_cast<int32_t>(accum[i * 4 + 3]);
}
```

这里的i*4+0和i*4+1对应thread0就是第一行每8列的前两个元素：d00和d01(i = 0), d08和d09(i = 1)....

i*4+2和i*4+3对应thread0就是第8行每8列的前两个元素：d80和d81(i = 0), d88和d89(i = 1).....

store

在完成k维度上的循环计算后，结果block会用tma来存回global memory。使用第0个warp(threadIdx: 0-31) tma存回结果块和使用其他线程(threadIdx: 32-255) load 下一个计算块的scale_b到share memory会形成一个overlap。