15-779 Lecture 6: Advanced CUDA Programming: Warp Specialization

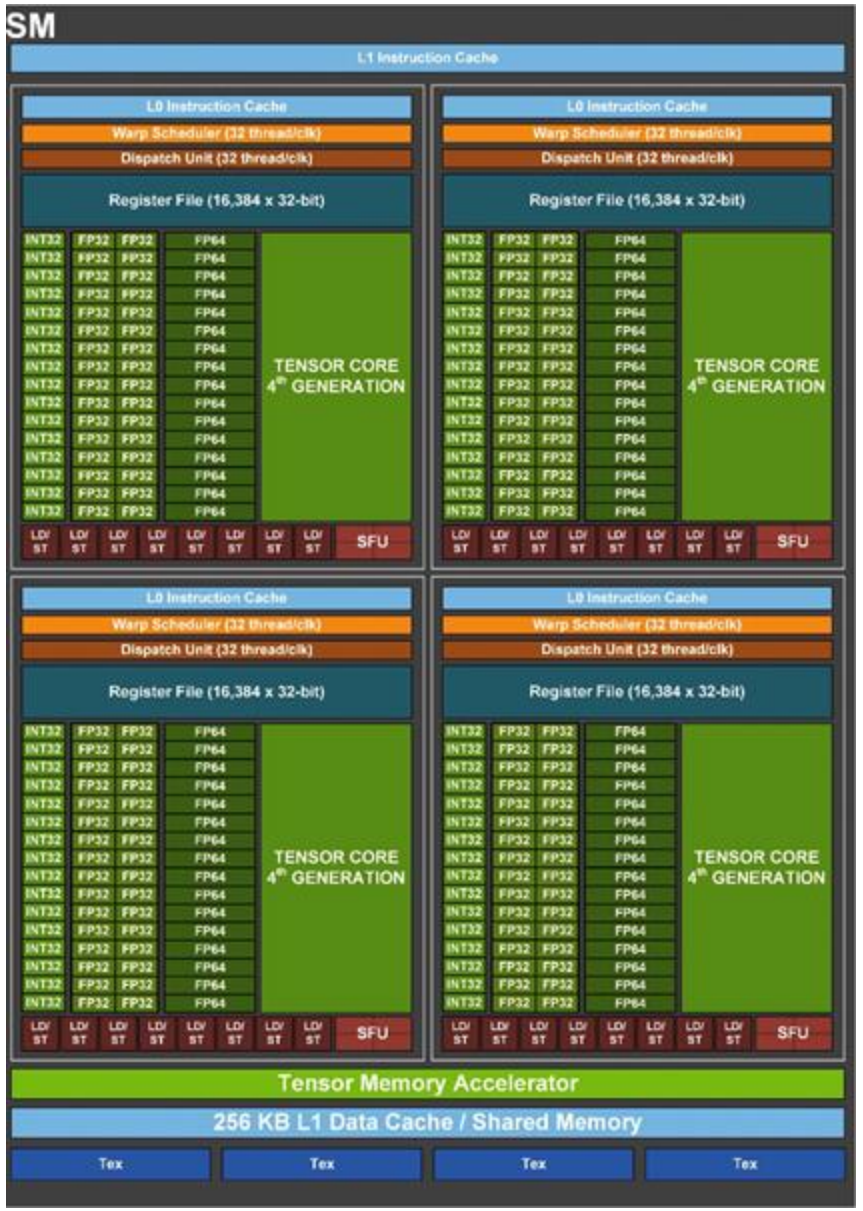
Xinhao Cheng

Computer Science Department

Carnegie Mellon University

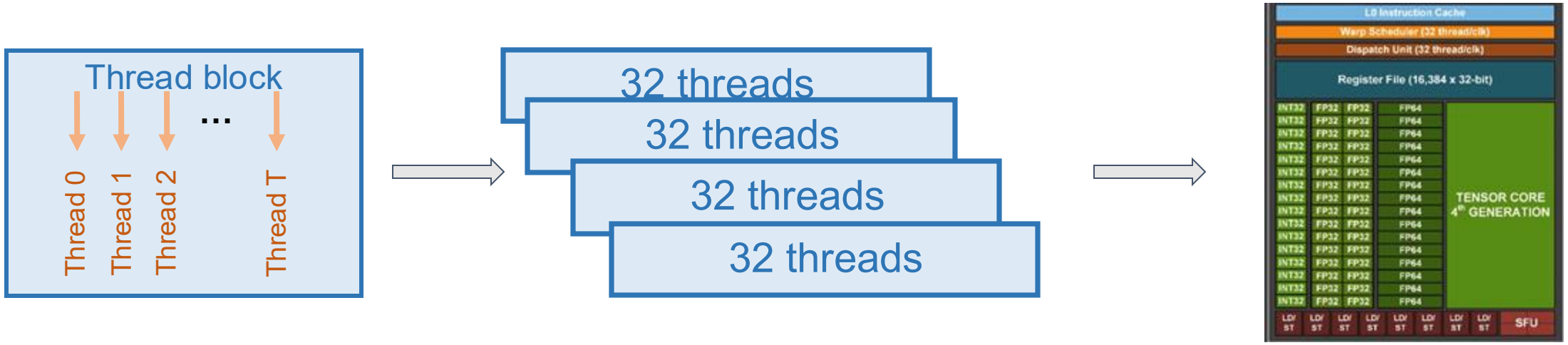
Background: Streaming multiprocessor architecture

- Processing block
- Cuda cores, tensor cores, SFU
- Load/Store unit
- Warp scheduler
- Register file
- Tensor memory accelerator
- …



https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

Background: Thread Block and Warp



number of warps = ceil(threads per block / warp size(32))

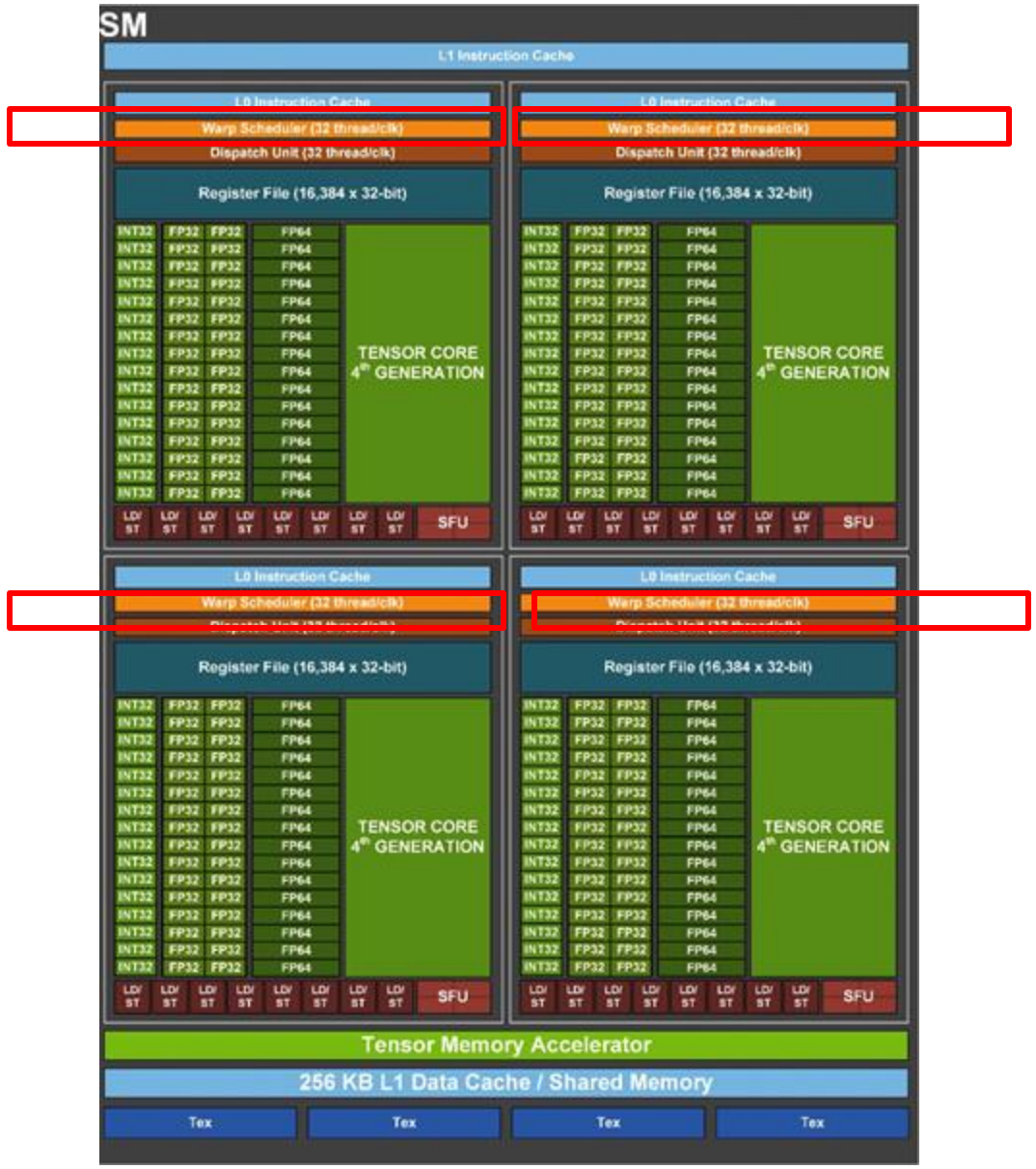
Background: Warp Scheduling

warp status

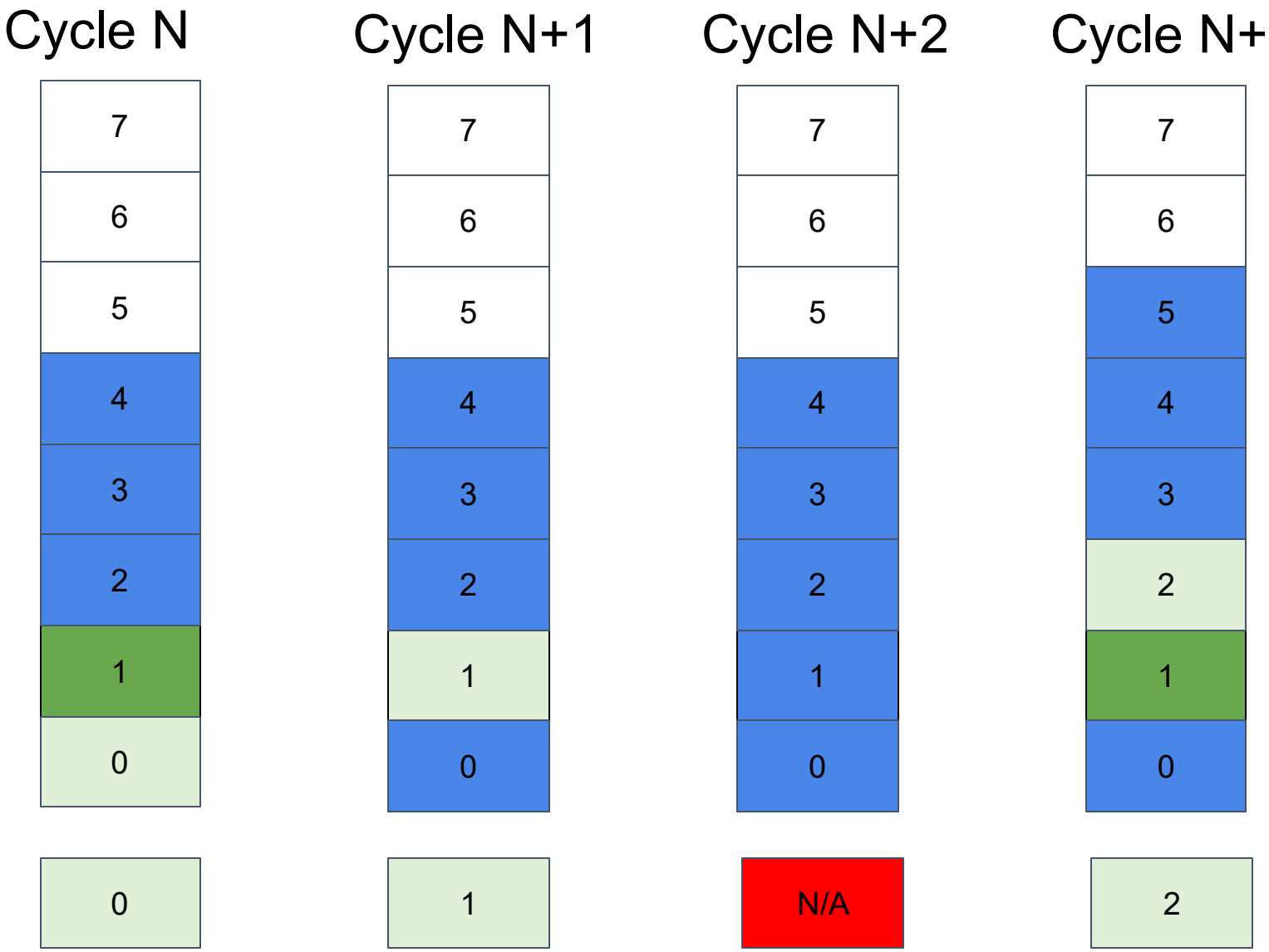
- Unused
- Active
- Stalled
- Eligible
- Selected

warp slots

7
6
5
4
3
2
1
0



Background: Warp Scheduling



warps stalled: 16
warps eligible: 5
warps issued: 3

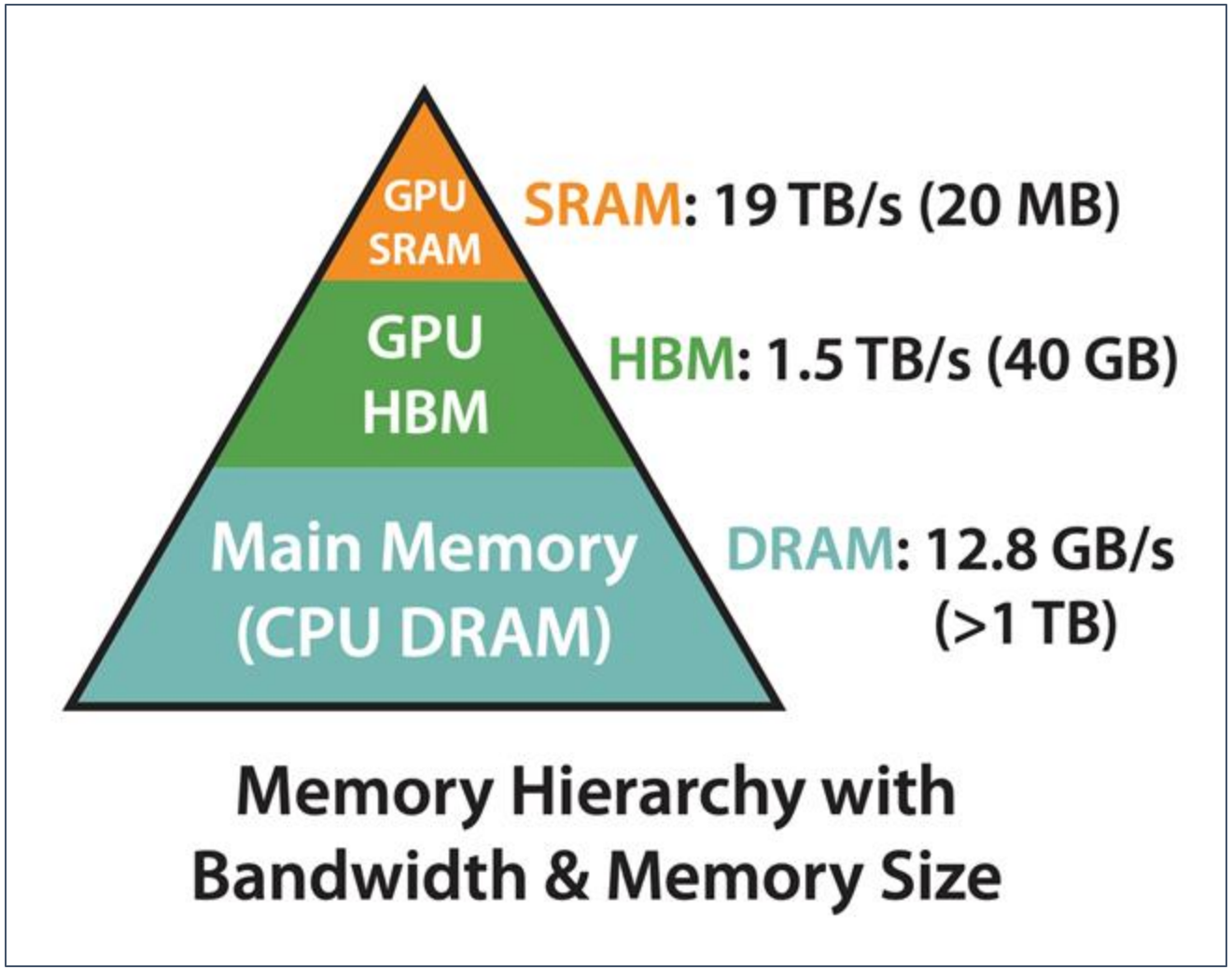
issue_slot_utlization: 75%

we need more eligible warps to increase the slot utilization

Key insight: more independent instructions across warps

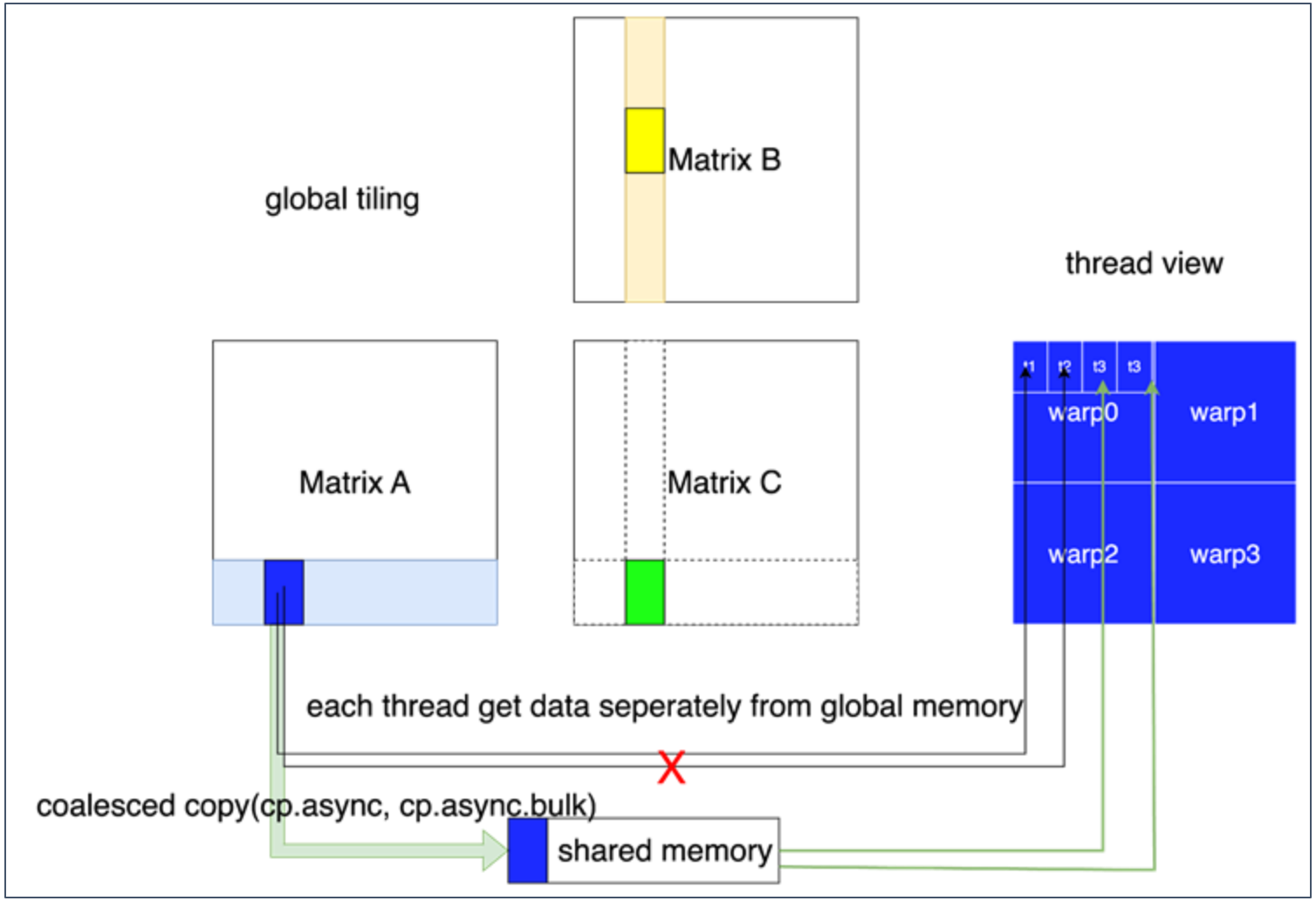
Background: Data Movement From Host to Device

- on chip memory has highest bandwidth
- 20MB = 192KB * 108 SMs



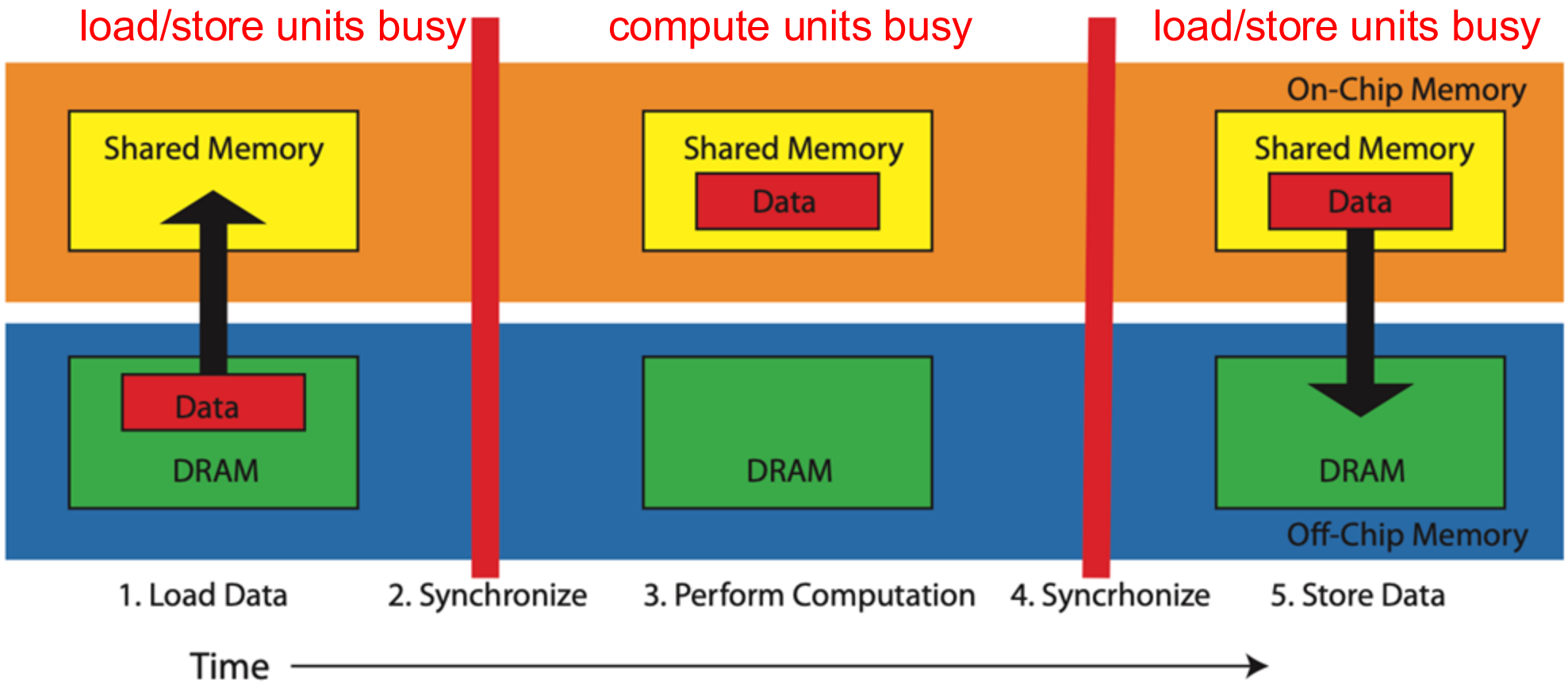
Background: Data Movement From Host to Device

- coalesced copy data to shared memory
- each thread read data with low latency
- …



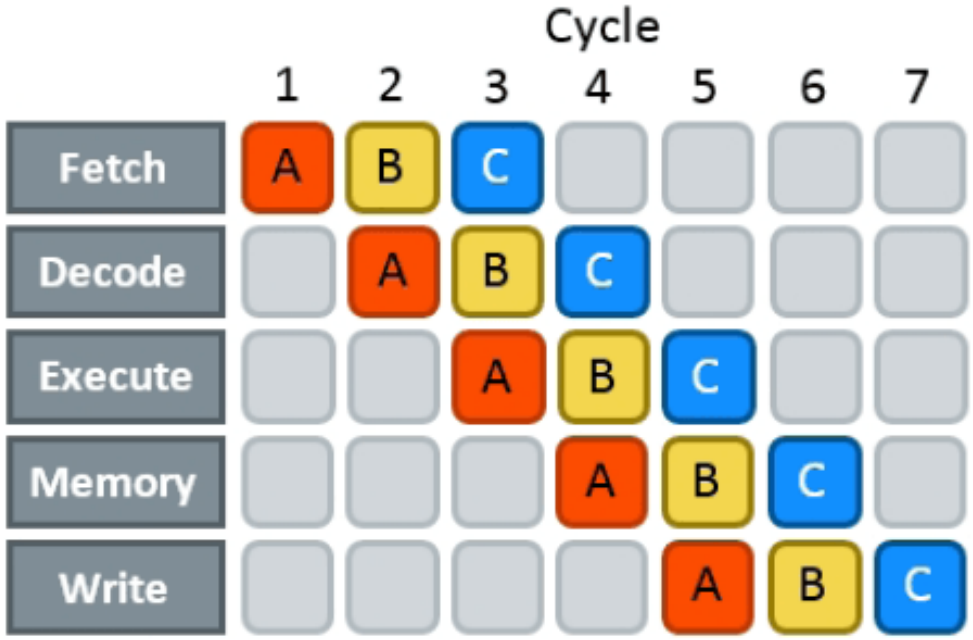
A Common Data Movement Paradigm

use __syncthreads() to sync the threads inside a thread block



Software Pipelining

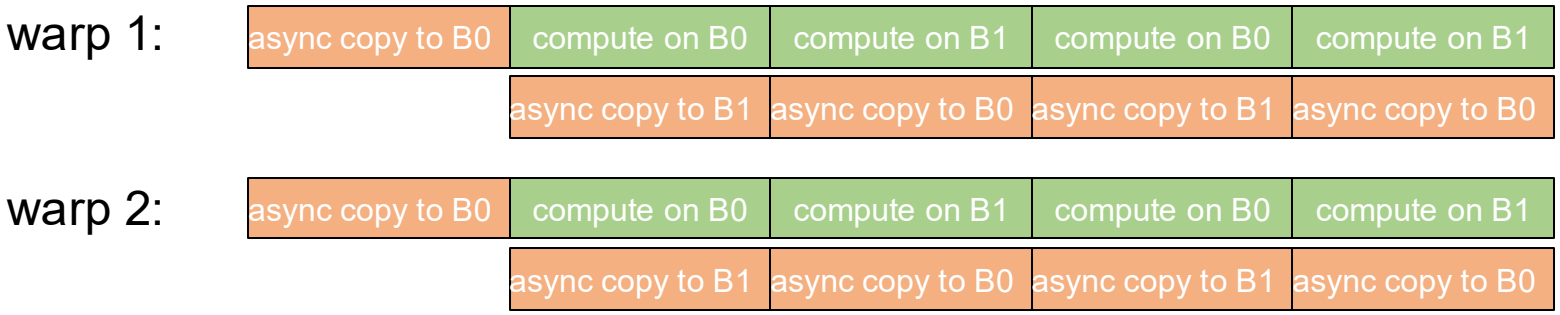
- instruction level parallelism
- launch next instruction without waiting for the previous one to complete
- not “reduce” latency but “hide” latency



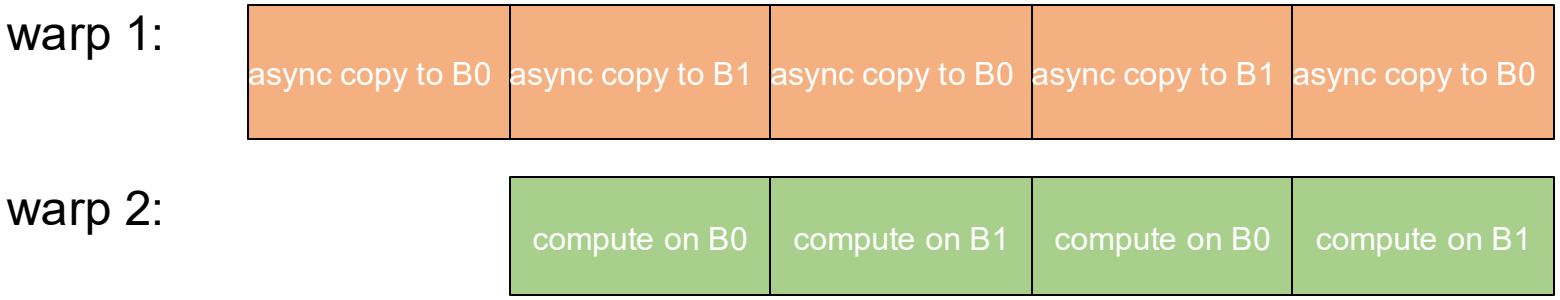
https://en.algorithmica.org/hpc/pipelining/

Two Approaches to Implement Software Pipelining in CUDA

- Multi-stage pipelining

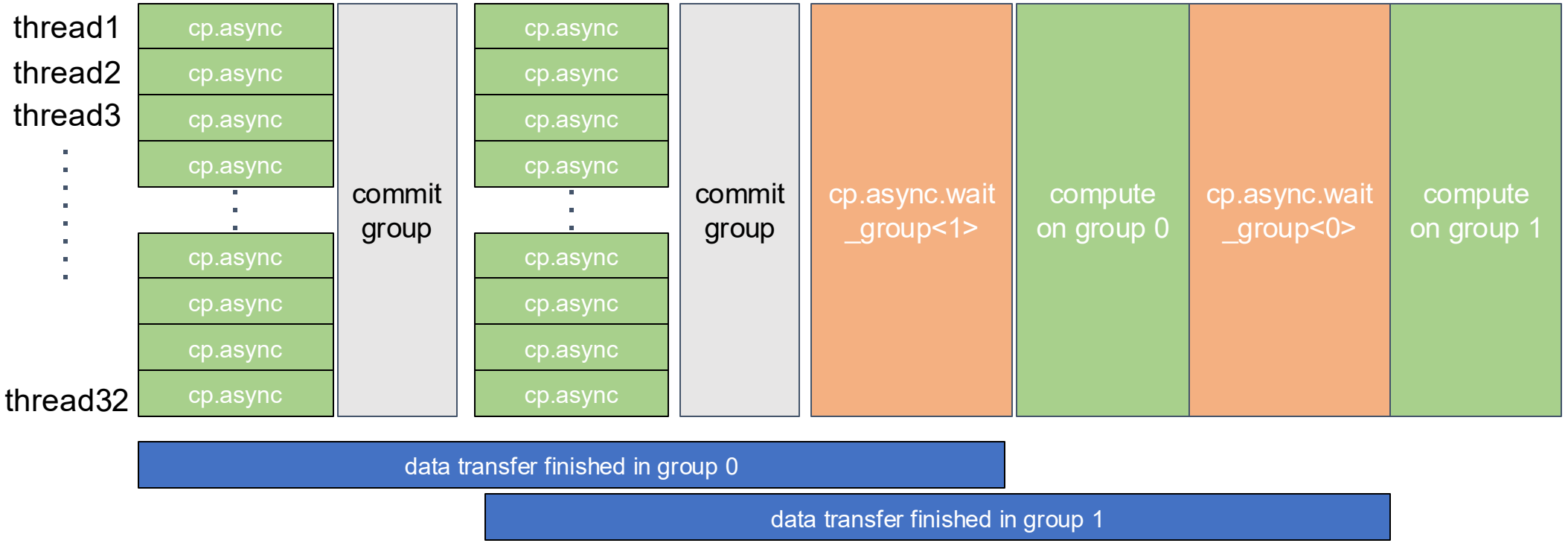


- Warp specialization



Asynchronous copy in CUDA

- **cp.async**: initiate an asynchronous copy
- **cp.async.commit_group**: batches all prior cp.async instructions into a group
- **cp.async.wait_group<N>**: let executing threads wait for **at most N groups pending**.
- **cp.async.wait_all**: equals to cp.async.wait_group<0>

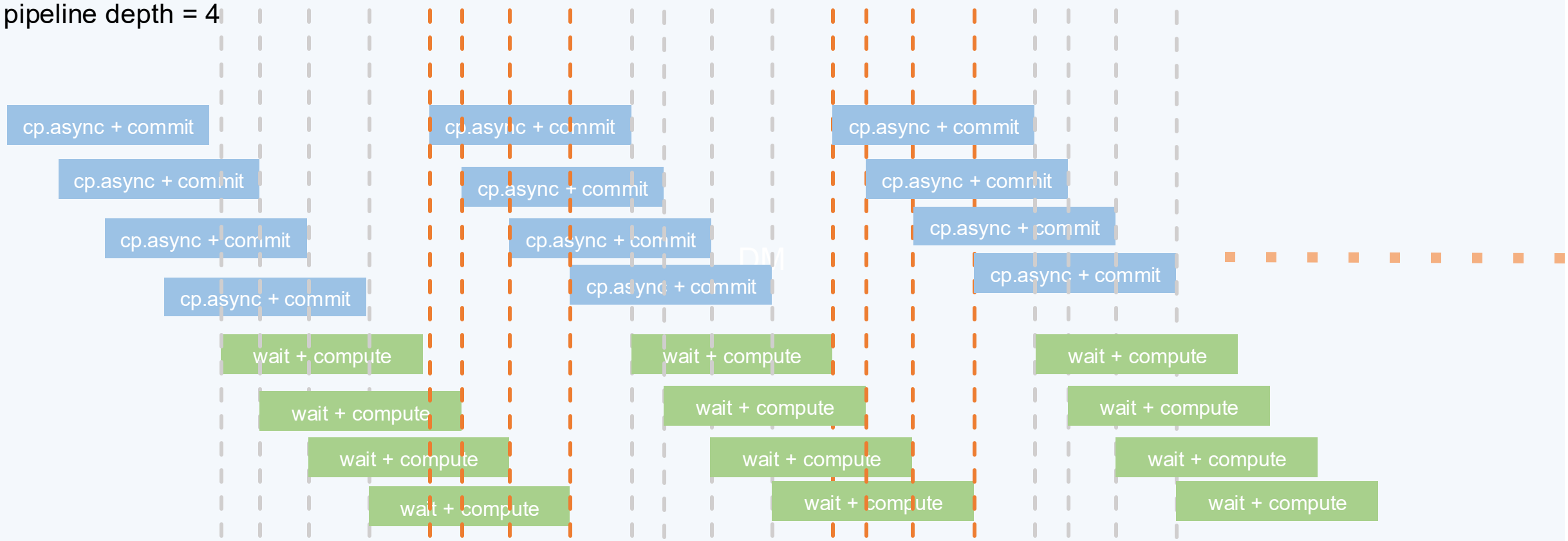


MatMul on Ampere GPU with Multi-stage Pipelining

- Asynchronous data copy instruction provides opportunity for intra-warp overlapping
- Use cp.async, cp.async.commit_group and cp.async.wait to construct an async pipeline

```
//prefetch data from device memory to shared memory
for (int k_pipe = 0; k_pipe < PIPELINE_DEPTH - 1; k_pipe++) {
  //compute the shared memory and device memory address
  for (int i = threadIdx.x; i < NUM_TOTAL_CHUNKS; i += NUM_THREADS) {
    // issue the async copy in chunk of 128B
    asm volatile("cp.async.cg.shared.global.L2…”);
  }
  asm volatile("cp.async.commit_group;\n" ::);
}

for (int for_idx = 0; for_idx < FORLOOP_RANGE; for_idx++) {
  if (for_idx + PIPELINE_DEPTH - 1 < FORLOOP_RANGE) {
    for (int i = threadIdx.x; i < NUM_TOTAL_CHUNKS; i += NUM_THREADS) {
      // issue the async copy in chunk of 128B
      asm volatile("cp.async.cg.shared.global.L2…”);
    }
    asm volatile("cp.async.commit_group;\n" ::);
    asm volatile("cp.async.wait_group %0;\n" ::"n"(PIPELINE_DEPTH -
1));
  } else{
    asm volatile("cp.async.wait_all;\n" ::);
  }

  //tensor core instruction
  asm volatile("mma.sync.aligned.m16n8k16…”);
}
```

MatMul on Ampere GPU with Multi-stage Pipelining

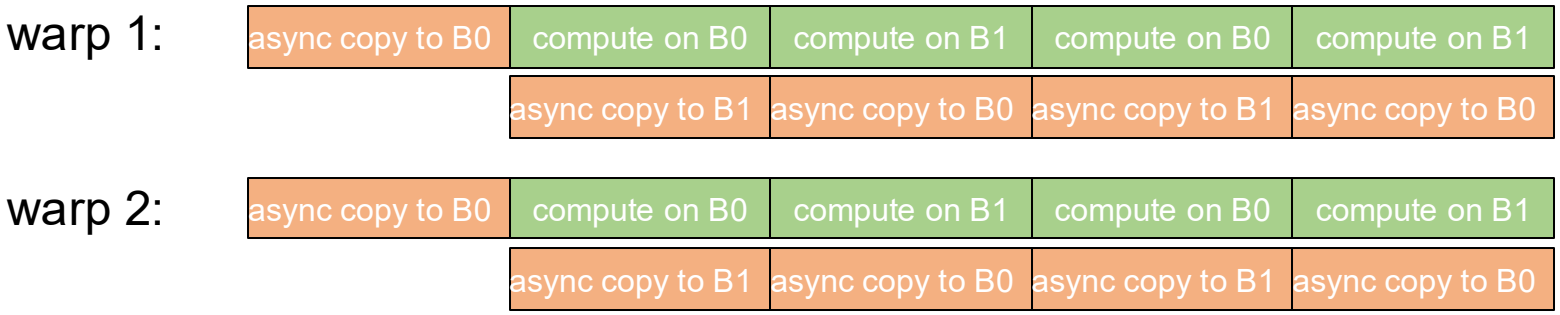


Issues with Multi-Stage Pipelining

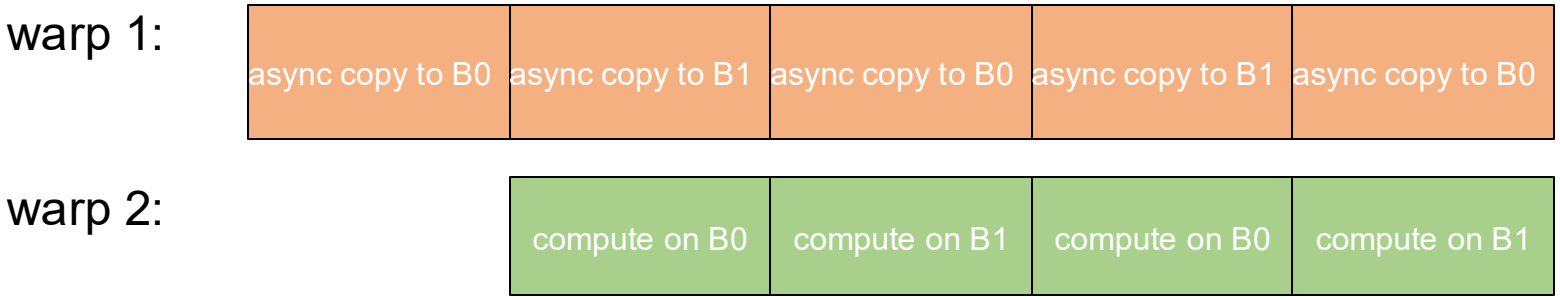
- coarse grained synchronization
- address generation overhead
- hard to decouple instruction streams

Two Approaches to Implement Software Pipelining in CUDA

- Multi-stage pipelining



- Warp specialization



Warp Specialization (Spatial Partitioning)

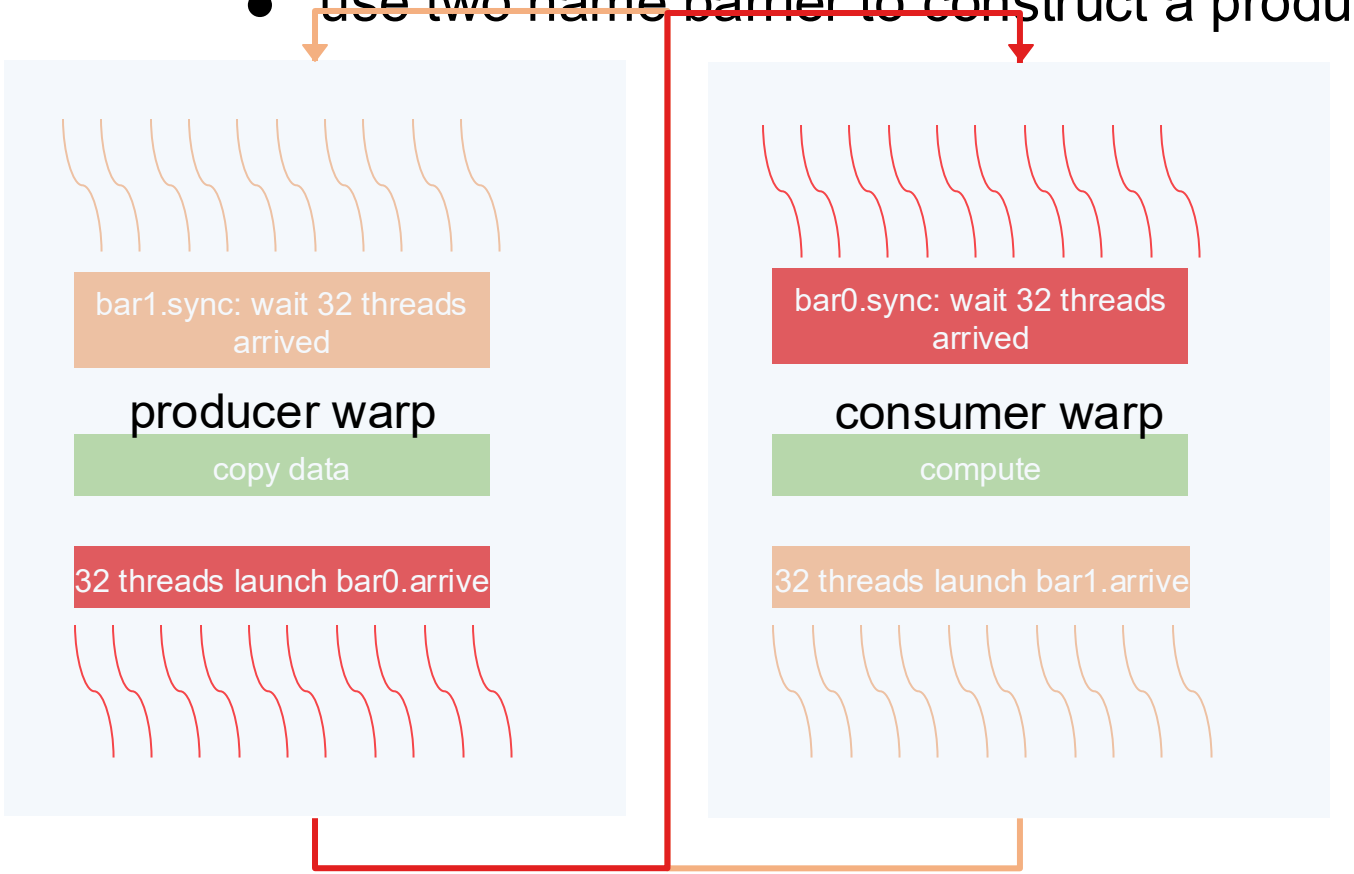
- A thread block can be spatially partitioned to perform independent computations.

- One subset of threads produces data that is concurrently consumed by the other subset of threads.

Producer Warp	Consumer Warp
wait for buffer to be ready to be filled	signal buffer is ready to be filled
produce data and fill the buffer	wait for buffer to be filled
signal buffer is filled	consume data in filled buffer

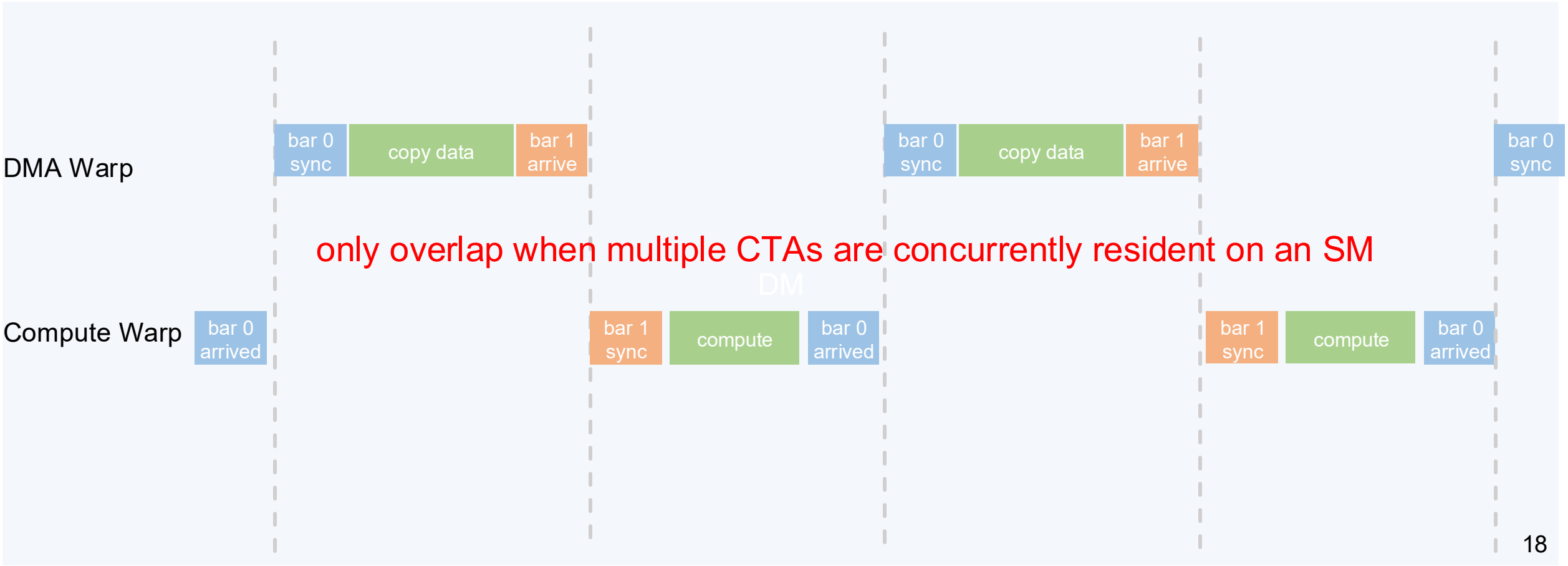
Synchronization across Warps in a Threadblock

- bar{.cta}.arrive: signal the arrival threads (data transfer has finished filling a buffer/computation finished)
- bar{.cta}.sync: block the threads until the name barrier flipped
- use two name barrier to construct a producer/consumer pipeline



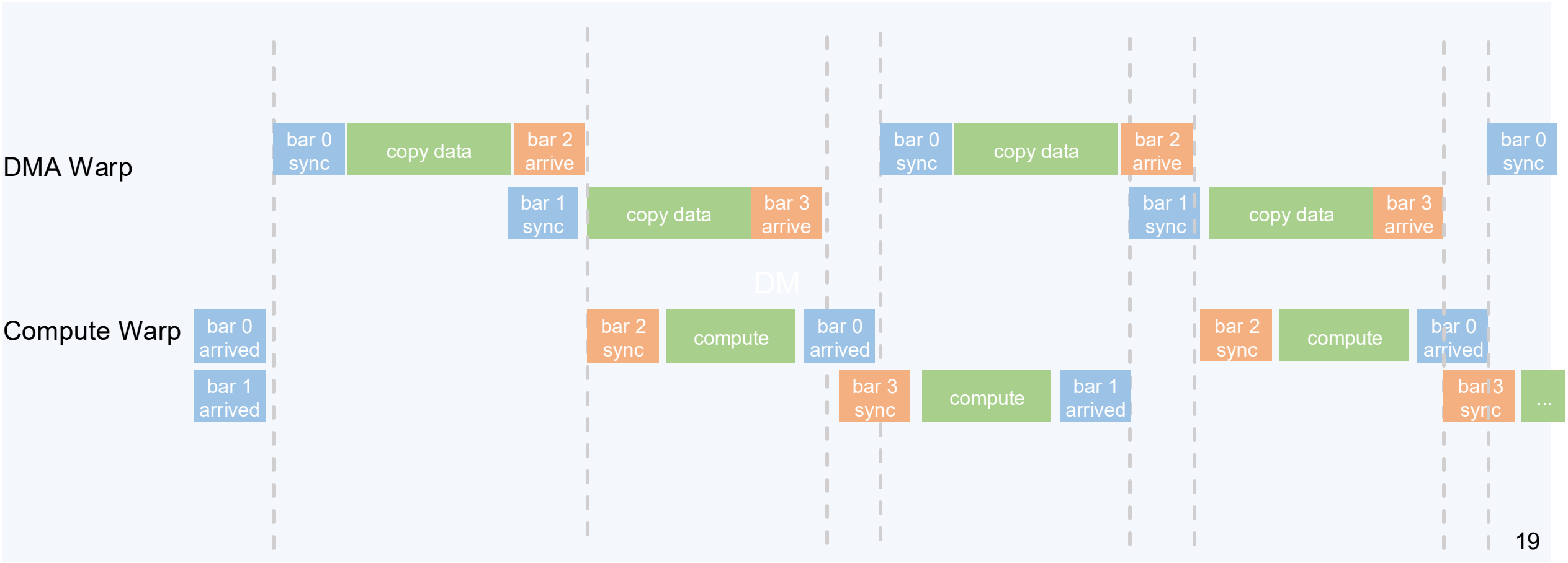

```
if(threadIdx.x < COMPUTE_THREADS){
    //wait for dma finished
    asm volatile("bar.sync [%0], %1”, barrier_0, 32)
    //do computation
   asm volatile("wmma.mma…”);
   //notify computation is finished
    asm volatile("bar.arrive [%0], %1”, barrier_1, 32)
} else {
    //wait for computation finished
    asm volatile("bar.sync [%0], %1”, barrier_1, 32)
    // do data copy
    asm volatile("ld.global…”);
    //notify data copy is finished
    asm volatile("bar.arrive [%0], %1”, barrier_0, 32)
}
```

CudaDMA:
Overlap Data Load and Compute with Warp Specialization



More Optimizations: Warp Specialization + Double Buffer

- more overlapping, more independent instructions
- but use more barriers, more shared memory reserved



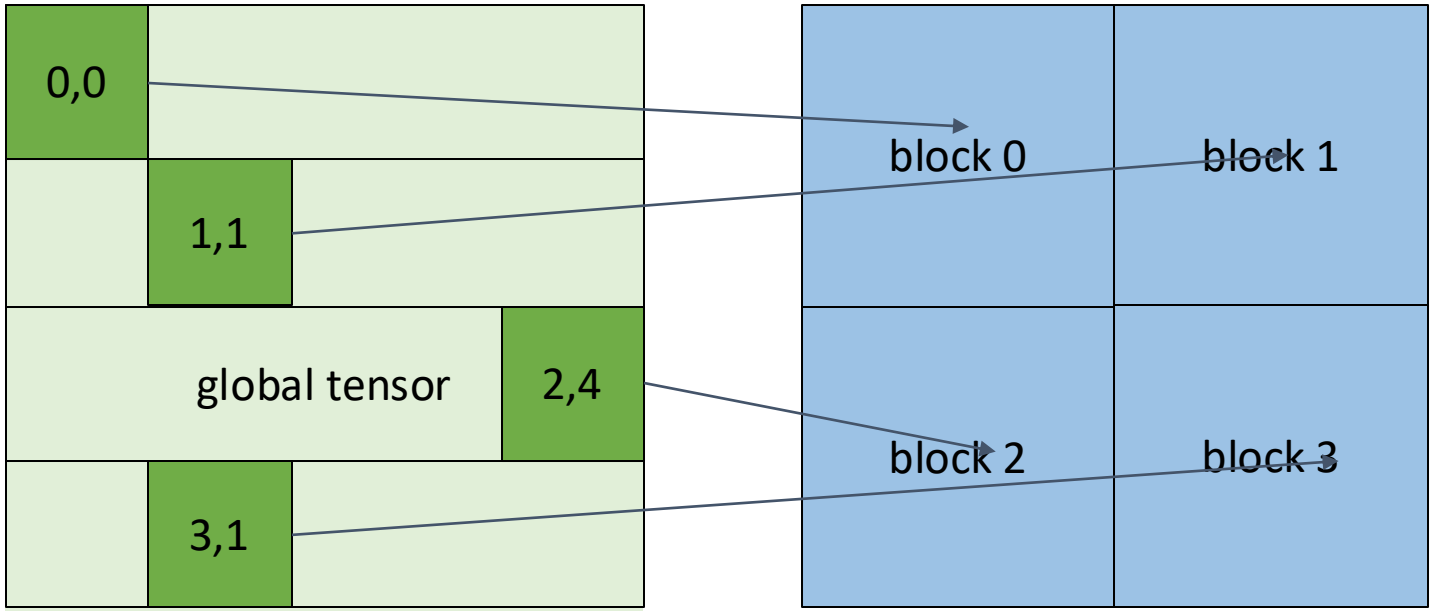
Nvidia Hopper GPU: Tensor Memory Accelerator(TMA)

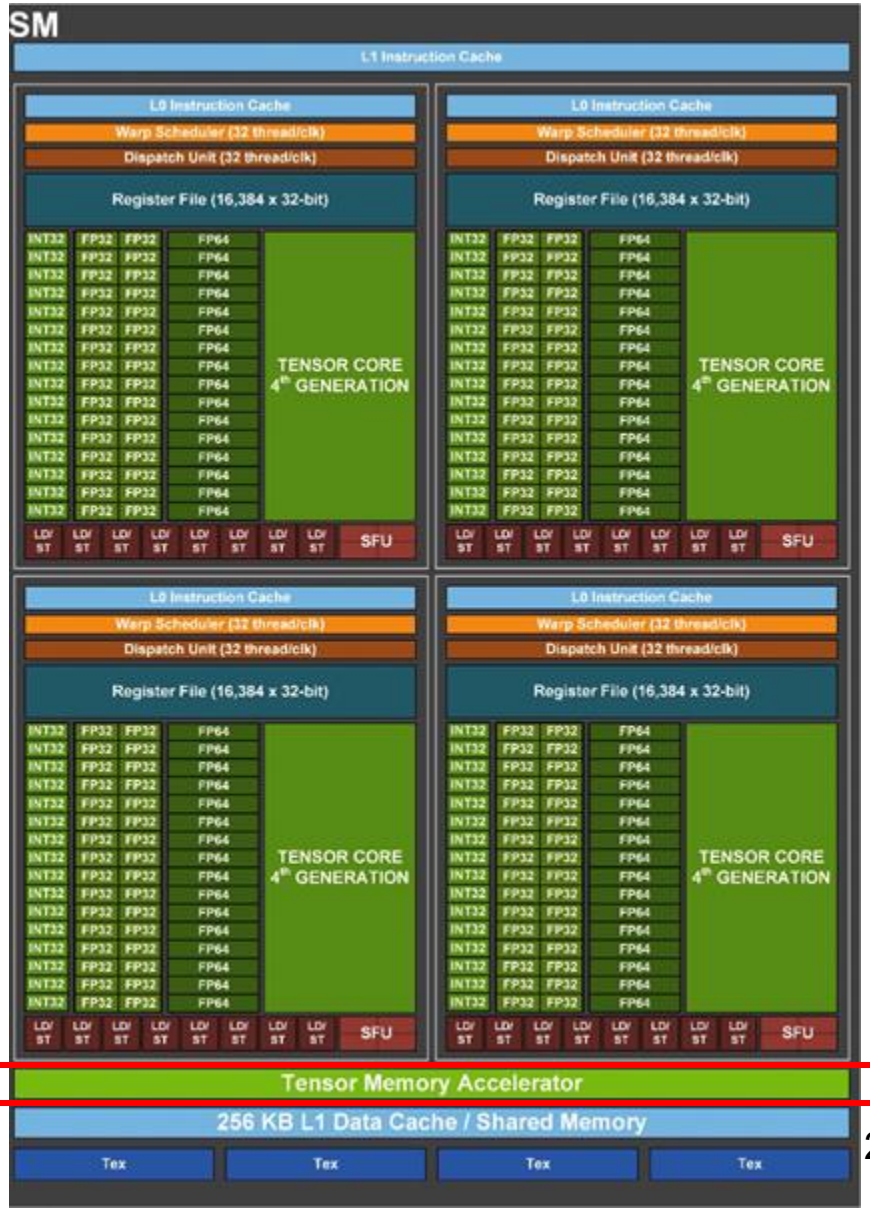
tiling information stored in TMA descriptor(CUtensorMap)

```
CUresult result = cuTensorMapEncodeTiled(tma_desc,
                                         tma_format, // type of data transmitted by TMA
                                         tma_dim, // tma dimension
                                         global_addr, //device tensor address
                                         gmem_shape_ptr, //device tensor shape
                                         gmem_stride_ptr + 1, // device tensor stride
                                         smem_box_shape_ptr, // shared tensor shape
                                         smem_box_stride_ptr, //shared tensor stride
                                         CU_TENSOR_MAP_INTERLEAVE_NONE,
                                         tma_swizzle, // swizzle mode
                                         CU_TENSOR_MAP_L2_PROMOTION_NONE,
                                         CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE);
```

Nvidia Hopper GPU: Tensor Memory Accelerator(TMA)

Use coordinate to determine with tile is loaded in the current thread block, address generation handled by TMA hardware





TMA Reduces Register Usage

- cp.async needs to allocate registers for addressing data copy
- TMA provides hardware support for address calculation

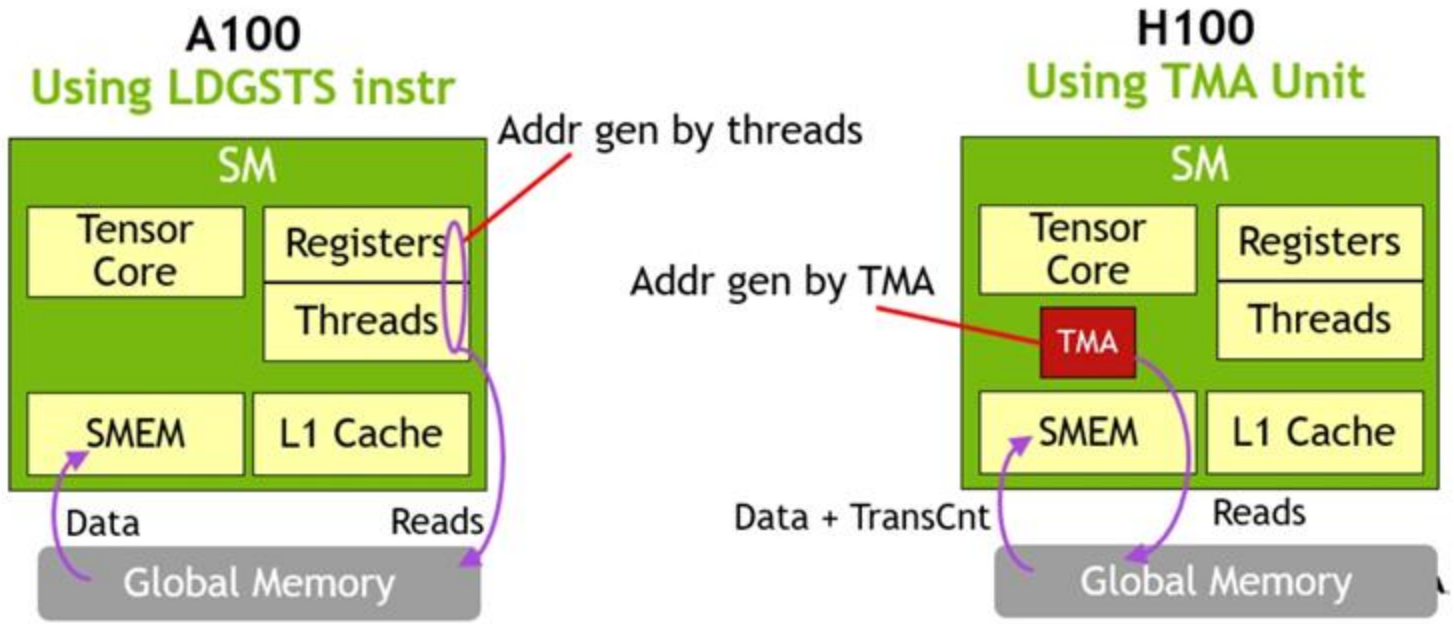
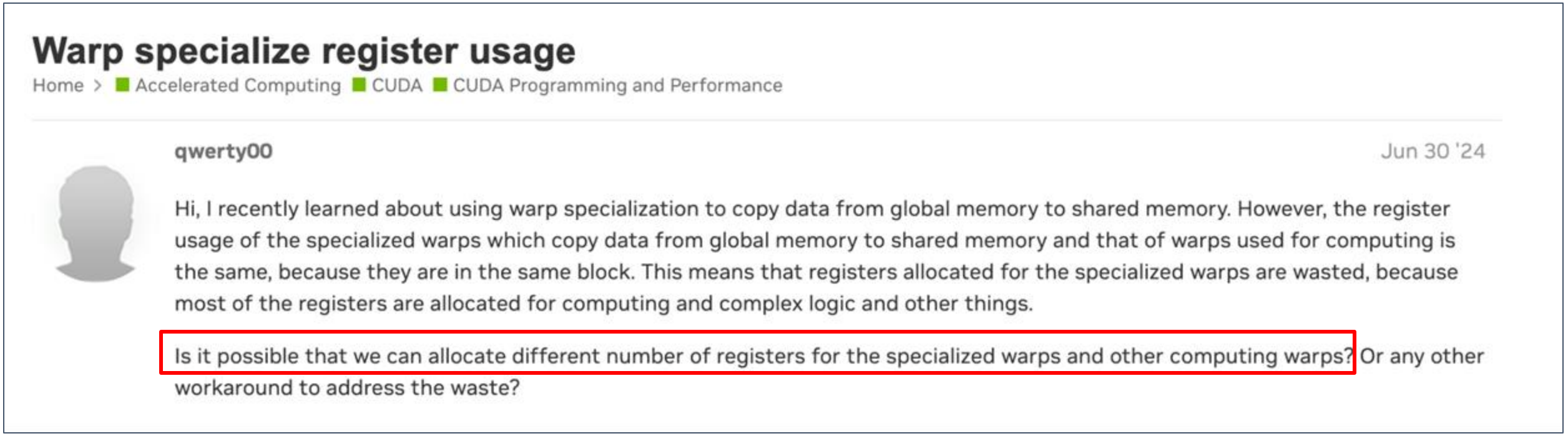


Figure 16. Asynchronous memory copy with TMA on H100 vs. LDGSTS on A100

https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

Nvidia Hopper GPU: Customize Register Allocation Across Warps

A Hopper GPU has 64K registers per SM



Warp specialize register usage

Home > Accelerated Computing > CUDA > CUDA Programming and Performance

qwerty00 Jun 30 '24

Hi, I recently learned about using warp specialization to copy data from global memory to shared memory. However, the register usage of the specialized warps which copy data from global memory to shared memory and that of warps used for computing is the same, because they are in the same block. This means that registers allocated for the specialized warps are wasted, because most of the registers are allocated for computing and complex logic and other things.

Is it possible that we can allocate different number of registers for the specialized warps and other computing warps? Or any other workaround to address the waste?

https://forums.developer.nvidia.com/t/warp-specialize-register-usage/298190

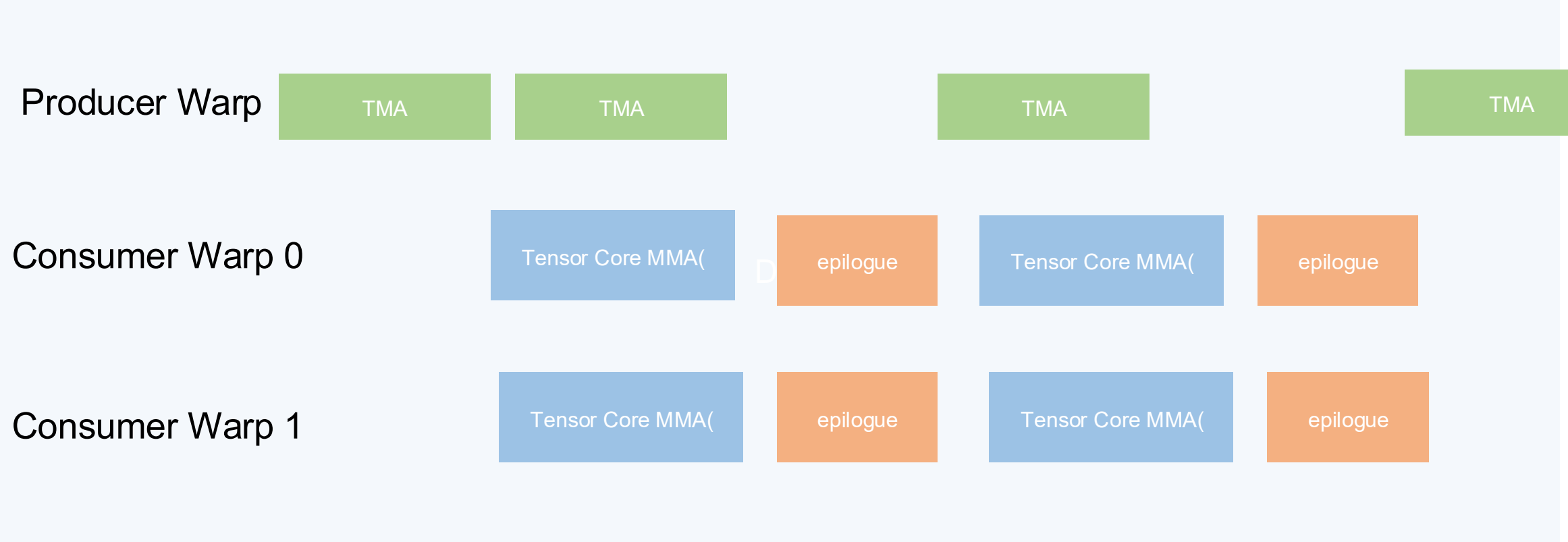
Nvidia Hopper GPU: save register usage by setmaxnreg

40 * 128(producer threads) + 232 * 256(consumer threads) = 64512 (< 65536 registers per SM)

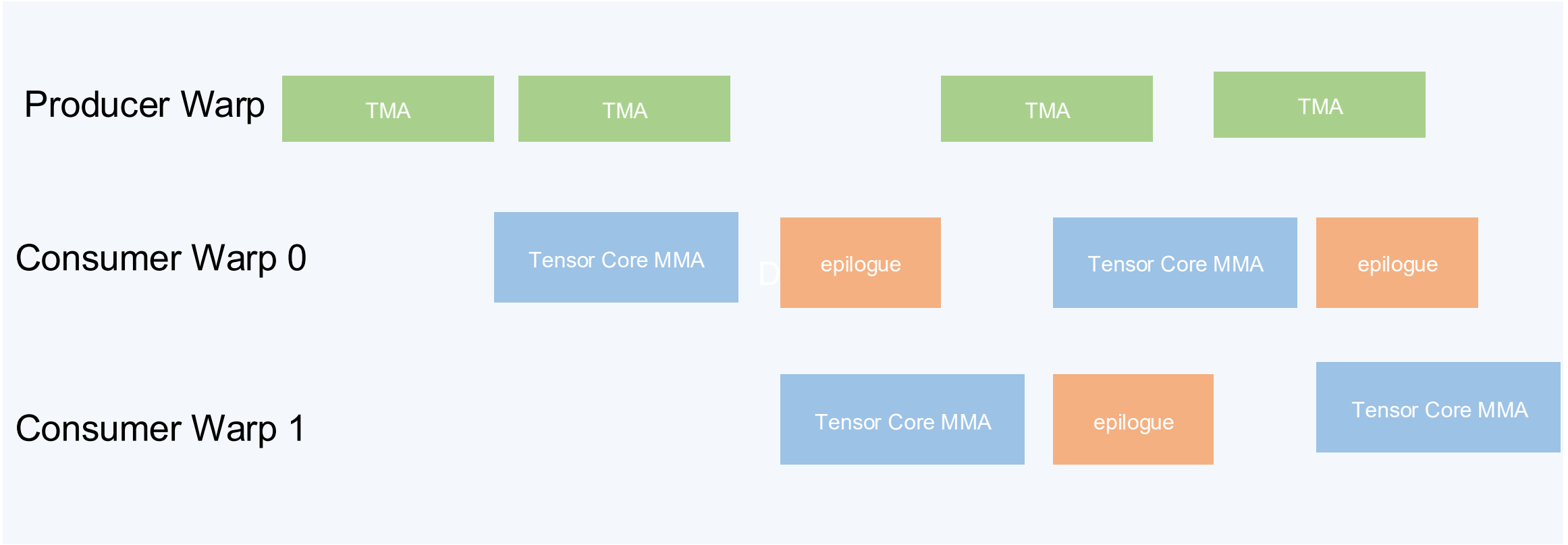
```
template<uint32_t RegCount>
void warpgroup_reg_alloc(){
  //increase the maximum number of registers in this warp
  asm volatile("setmaxnreg.inc.sync.aligned.u32 %0;\n" : : "n"(RegCount) );
}

template<uint32_t RegCount>
void warpgroup_reg_dealloc(){
  //decrease the maximum number of registers in this warp
  asm volatile("setmaxnreg.dec.sync.aligned.u32 %0;\n" : : "n"(RegCount) );
}

// usage
if (warp_group_role == WarpGroupRole::Producer) {
      cutlass::arch::warpgroup_reg_dealloc<LoadRegisterRequirement>(); // LoadRegisterRequirement = 40
} else if (warp_group_role == WarpGroupRole::Consumer0 || warp_group_role == WarpGroupRole::Consumer1) {
      cutlass::arch::warpgroup_reg_alloc<MmaRegisterRequirement>(); // MmaRegisterRequirement = 232
}
```

https://github.com/NVIDIA/cutlass

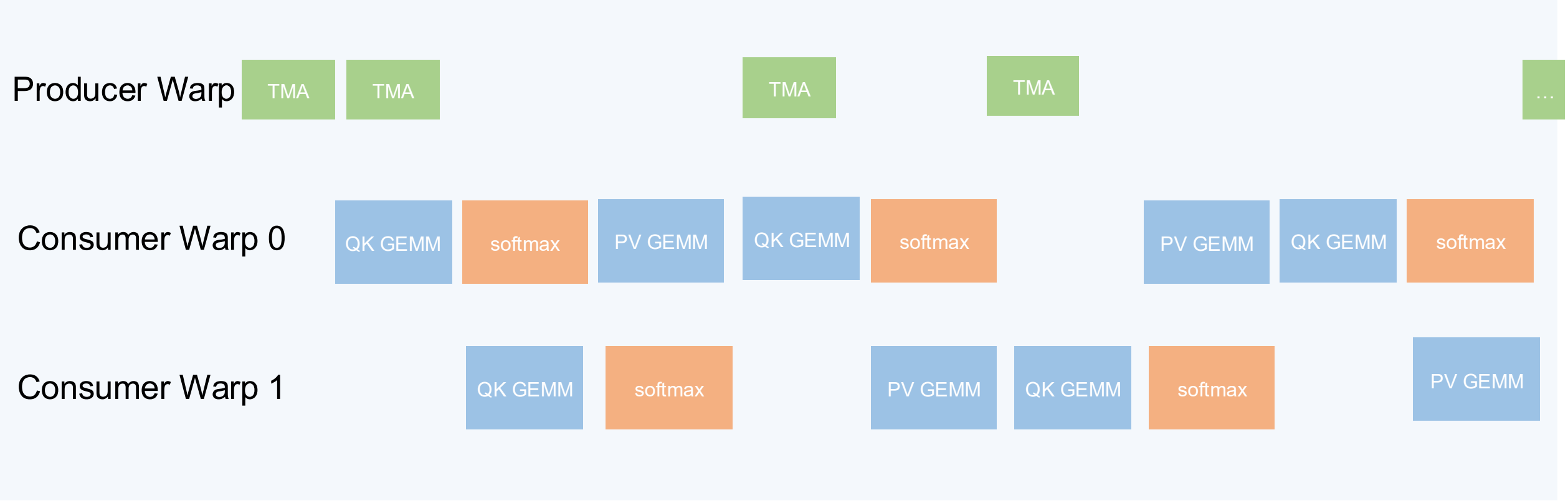
MatMul on Hopper with Warp Specialization



MatMul on Hopper with Warp Specialization + Pingpong Scheduling



FlashAttention 3: Attention with Warp Specialization + Pingpong Scheduling



FlashAttention 3: Intra-Warpgroup Overlapping