Evolving Ampere to Hopper

MIT 6.S894: Accelerated Computing Lecture 8 - 2025/10/23

Vijay Thakkar (@__tensorcore__)

- Senior DL Compute Architect @ NVIDIA

- “Part time” PhD student @ HPC Garage, GaTech

How do we achieve this?

(measurements in TFLOPS)	A100	A100 Sparse	H100 SXM5[1]	H100 SXM5[1] Sparse	H100 SXM5[1] Speedup vs A100
FP8 Tensor Core			2000	4000	6.4x vs A100 FP16
FP16	78		120		1.5x
FP16 Tensor Core	312	624	1000	2000	3.2x
BF16 Tensor Core	312	624	1000	2000	3.2x
FP32	19.5		60		3.1x
TF32 Tensor Core	156	312	500	1000	3.2x
FP64	9.7		30		3.1x
FP64 Tensor Core	19.5		60		3.1x
INT8 Tensor Core	624 TOPS	1248 TOPS	2000	4000	3.2x

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

Making number go up

Aka how do we fight physics

First Principles

These are all you need to know

Hardware and physics:

- Moore’s law is still alive and kicking but slowing down
- Dennard scaling is dead – per xtor energy will scale extremely slowly now
  - We are in a power limited regime of silicon engineering
- Data movement is sin
  - Cost of moving data is 100x-100000x higher that computing on it
  - Latency of moving data cannot be improved

Software and algorithmic:

- Matix multiply has infinite potential for data-reuse
  - Arbitrarily high arithmetic intensity
- Ahmdal’s law is the death of embarassingly parallel workloads

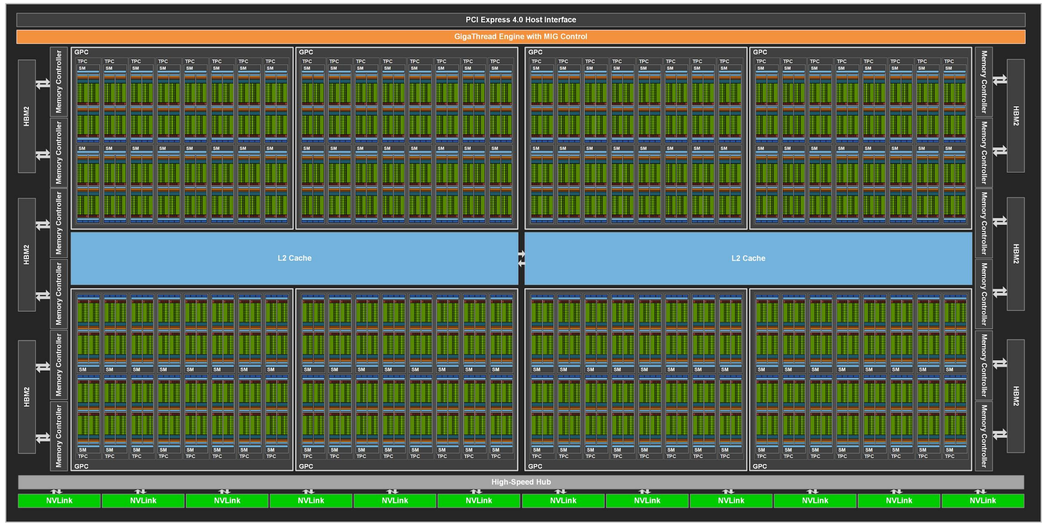
Lowest hanging fruit first

Can we just spam more cores???

A100: 2019
Reticle size die on TSMC N7
52 giga-xTors

Use it to add 22% more SMs (108 -> 132)

We are still 2.3x away from our scaling goal



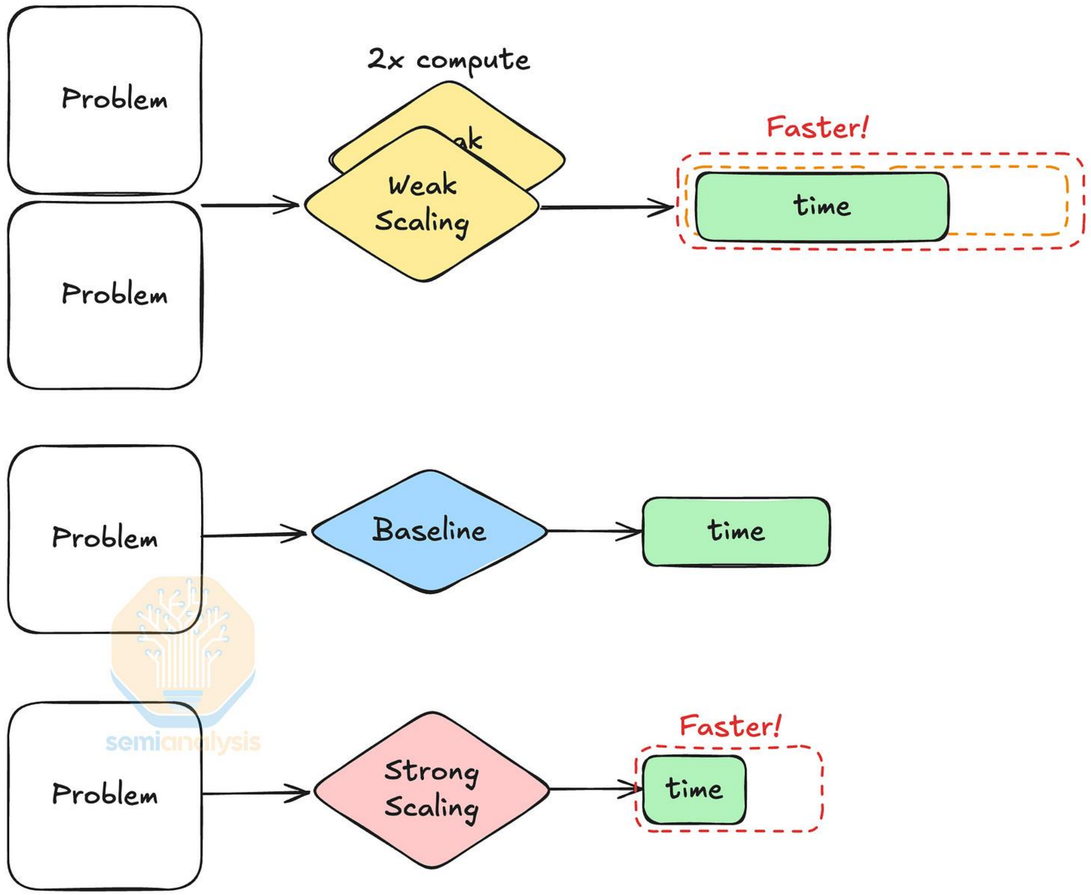
H100: 2021
Reticle size die on TSMC N5
80 giga-xTors

~ 54% density increase

Aside: Strong v. Weak Scaling

Would you rather fight 100 🦆 sized 🐕 or 1 🐕 sized 🦆

- Weak scaling:
  - Same time to solution for 2x larger problem with 2x more workers
  - Requires embarassingly parallel workload AND
  - An opulence of scaling execution units
- Strong scaling:
  - 2x lower time to solution for the same problem size



https://newsletter.semianalysis.com/p/nvidia-tensor-core-evolution-from-volta-to-blackwell

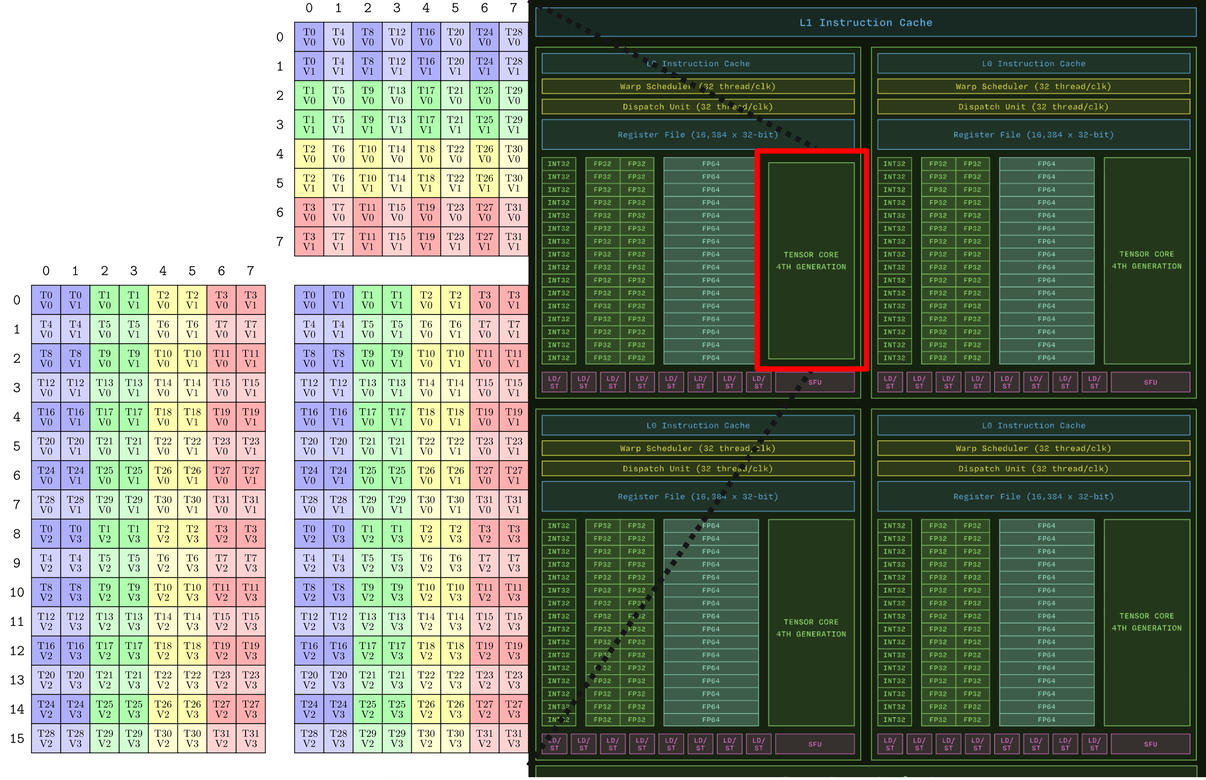
We have to strong scale

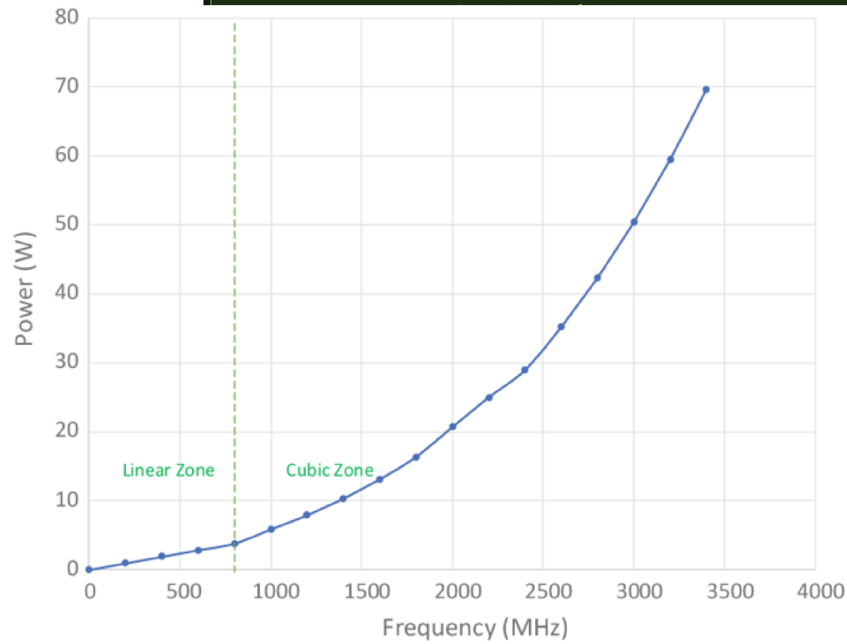
And there is no free lunch here …

- Ampere tensor core runs at 8 clocks per 16x8x16 instruction
- Can we just make it run in 4 clk instead? … Not so simple

- Increases b/w requirements on the VRF by 2x
- Data must load from SMEM -> RMEM 2x fewer clocks too
  - This is _insanely_ hard to do
  - Limited by wire parasitics and speed of electrons
- Pipeline deeper instead – grow RMEM size???
  - Add banks?
- How do we even hide issue latencies of HMMA/LDSM

- Can’t scale clock speeds either – power is O(frequency^3)



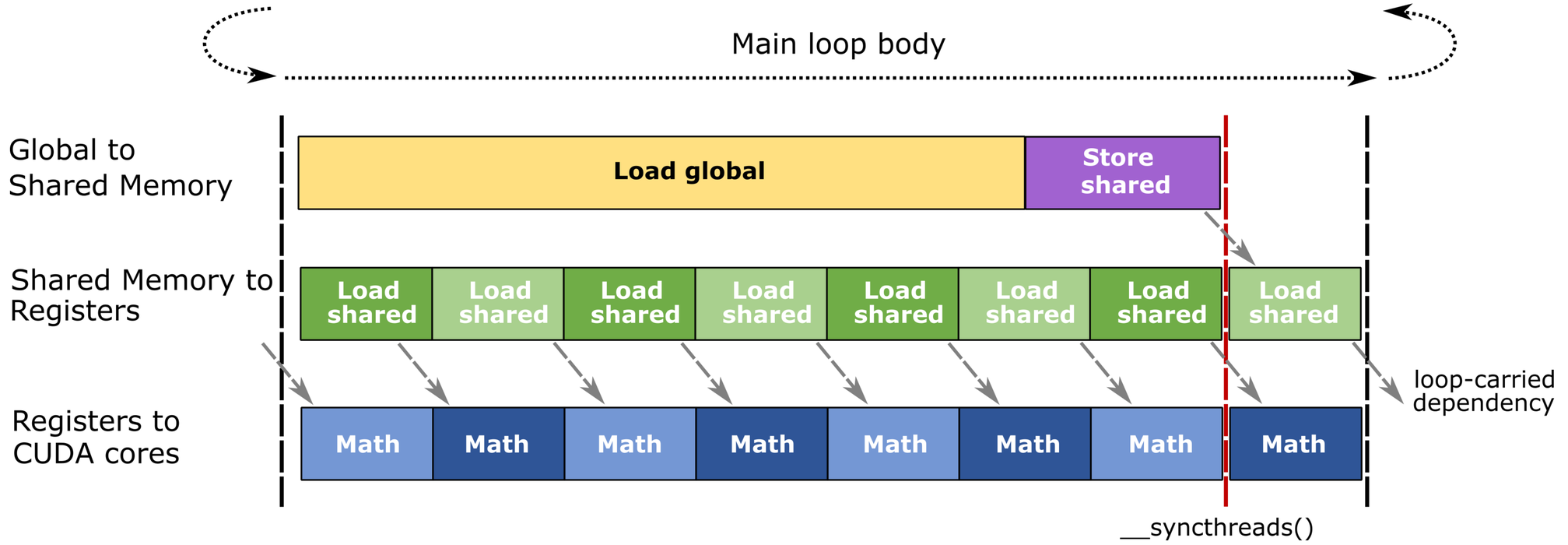


https://www.researchgate.net/figure/Power-and-frequency-relationship-at-linear-and-cubic-zones_fig3_323940362

Ampere Mainloop

ASYNC in SMEM and ILP in RMEM

- Async gmem->smem copy (3 stage pipeline)
- Deeper pipelines from async copies and less register pressure
- Tight instruction interleaving between LDSM+HMMA for peak utilization
- 2 CTA / SM occupancy for hiding epilogue
- 256 thread CTAs



Speed of Light Ampere Mainloop

Async in SMEM + ILP in RMEM

```
CUTLASS_PRAGMA_NO_UNROLL
for ( ; k_tile_count > -(TILE_STAGES-1); --k_tile_count)
{
  // Pipeline the outer products with a static for loop.
  // Note, the for_each() function is required here to ensure `k_block` is of type Int<x>.
  for_each(make_int_sequence<K_BLOCK_MAX>{}, [&] (auto k_block)
  {
    if (k_block == K_BLOCK_MAX - 1)
    {
      // Slice the smem_pipe_read smem
      tCsA_p = tCsA(_,_,_,smem_pipe_read);
      tCsB_p = tCsB(_,_,_,smem_pipe_read);

      // Commit the smem for smem_pipe_read
      cp_async_wait<TILE_STAGES-2>();
      __syncthreads();
    }

    // Load A, B shmem->regs for k_block+1
    auto k_block_next = (k_block + Int<1>{}) % K_BLOCK_MAX; // static
    copy(smem_tiled_copy_A, tCsA_p(_,_,k_block_next), tCrA_copy_view(_,_,k_block_next));
    copy(smem_tiled_copy_B, tCsB_p(_,_,k_block_next), tCrB_copy_view(_,_,k_block_next));

    // Copy gmem to smem before computing gemm on each k-pipe
    if (k_block == 0)
    {
      copy(gmem_tiled_copy_A, tAgA(_,_,_,*k_tile_iter), tAsA(_,_,_,smem_pipe_write)); // LDGSTS
      copy(gmem_tiled_copy_B, tBgB(_,_,_,*k_tile_iter), tBsB(_,_,_,smem_pipe_write)); // LDGSTS
      cp_async_fence();
      if (k_tile_count > 0) { ++k_tile_iter; }

      // Advance the pipe -- Doing it here accounts for K_BLOCK_MAX = 1 (no rmem pipe)
      smem_pipe_write = smem_pipe_read;
      ++smem_pipe_read;
      smem_pipe_read = (smem_pipe_read == TILE_STAGES) ? 0 : smem_pipe_read;
    }

    // Thread-level register rank-2 gemm for k_block
    cute::gemm(tiled_mma, accum, tCrA(_,_,k_block), tCrB(_,_,k_block), src_accum);

  }); // k_block
} // k_tile
```

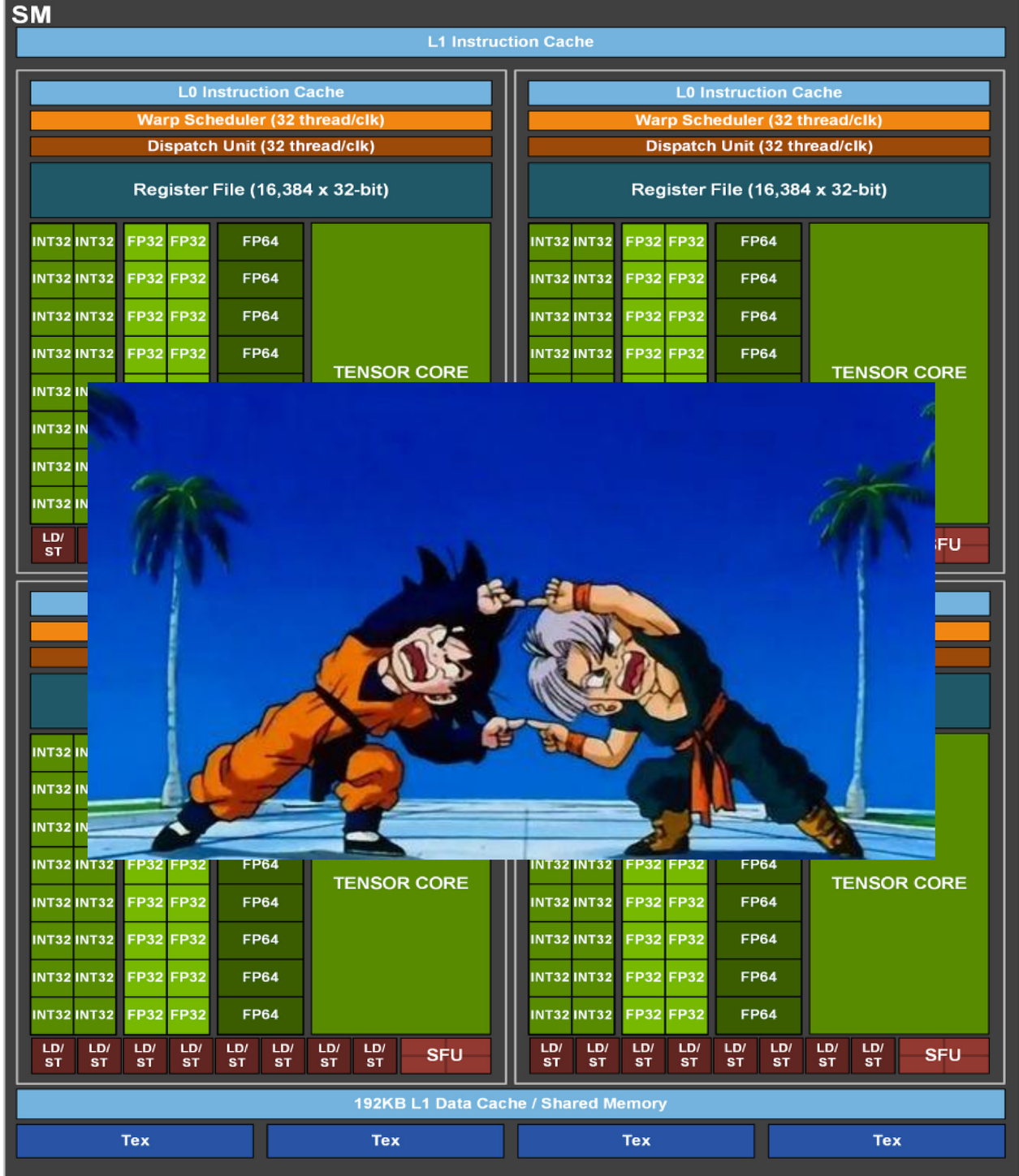
```
// Issue async gmem loads for PIPE+2
@p0 LDGSTS gmem[Rx+0], smem[Rx+0];
@p1 LDGSTS gmem[Rx+1], smem[Rx+1];
@p2 LDGSTS gmem[Rx+3], smem[Rx+2];
@p3 LDGSTS gmem[Rx+4], smem[Rx+3];

// Wait for PIPE to be visible
__cp_async_fence();
__cp_async_wait<N-2>();


// Issue “async” smem loads for HMMA+1/2
LDSM smem[Rm+0], Ra0, Ra1, Ra2, Ra3
LDSM smem[Rm+1], Rb0, Rb1, Rb2, Rb3


// Issue MMA
HMMA Ra0, Rb01, Rb02, Rc0
HMMA Ra1, Rb01.reuse, Rb02.reuse, Rc0
HMMA Ra2, Rb02, Rb03, Rc0
HMMA Ra3, Rb01.reuse, Rb02.reuse, Rc0

// repeat 2xLDSM+4xHMMA over the tile
```

Exploit the workload

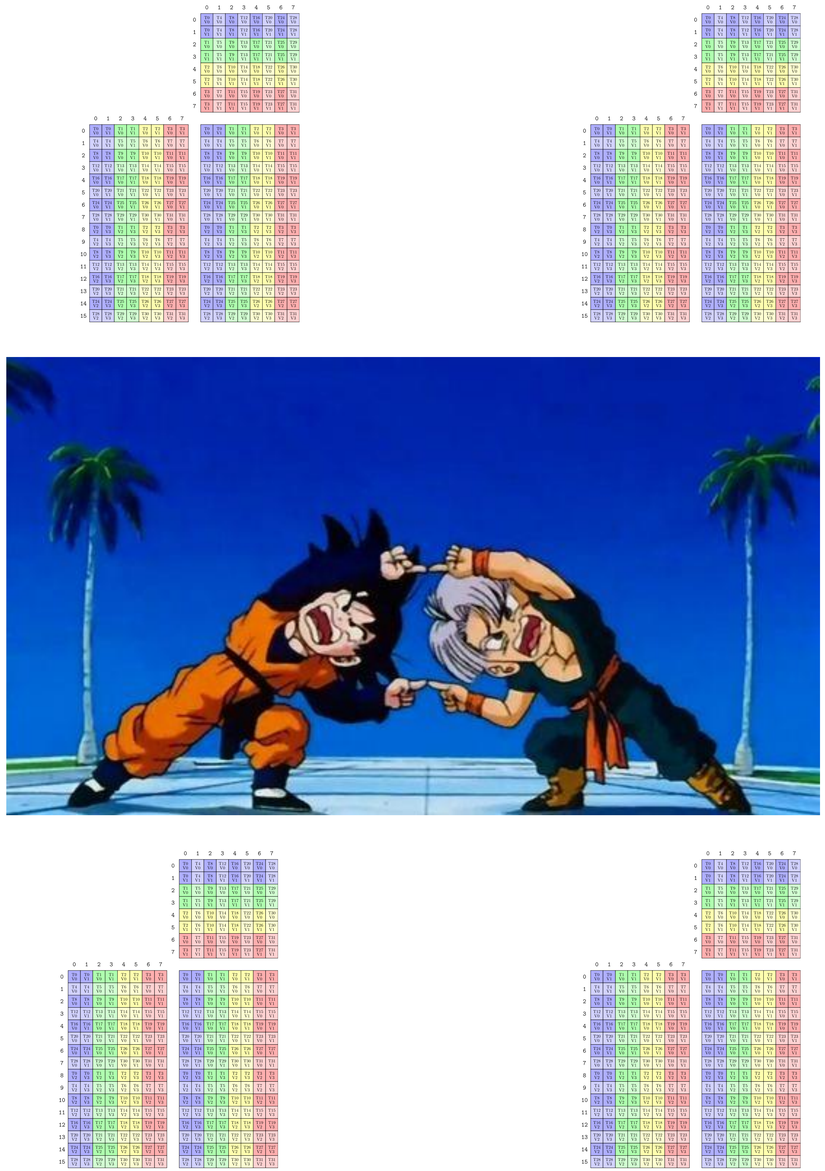
- Matmul is specifically amenable to strong scaling
- We have “infinite” arithmetic intensity
- AI = $\theta((m * n) \ / \ k)$

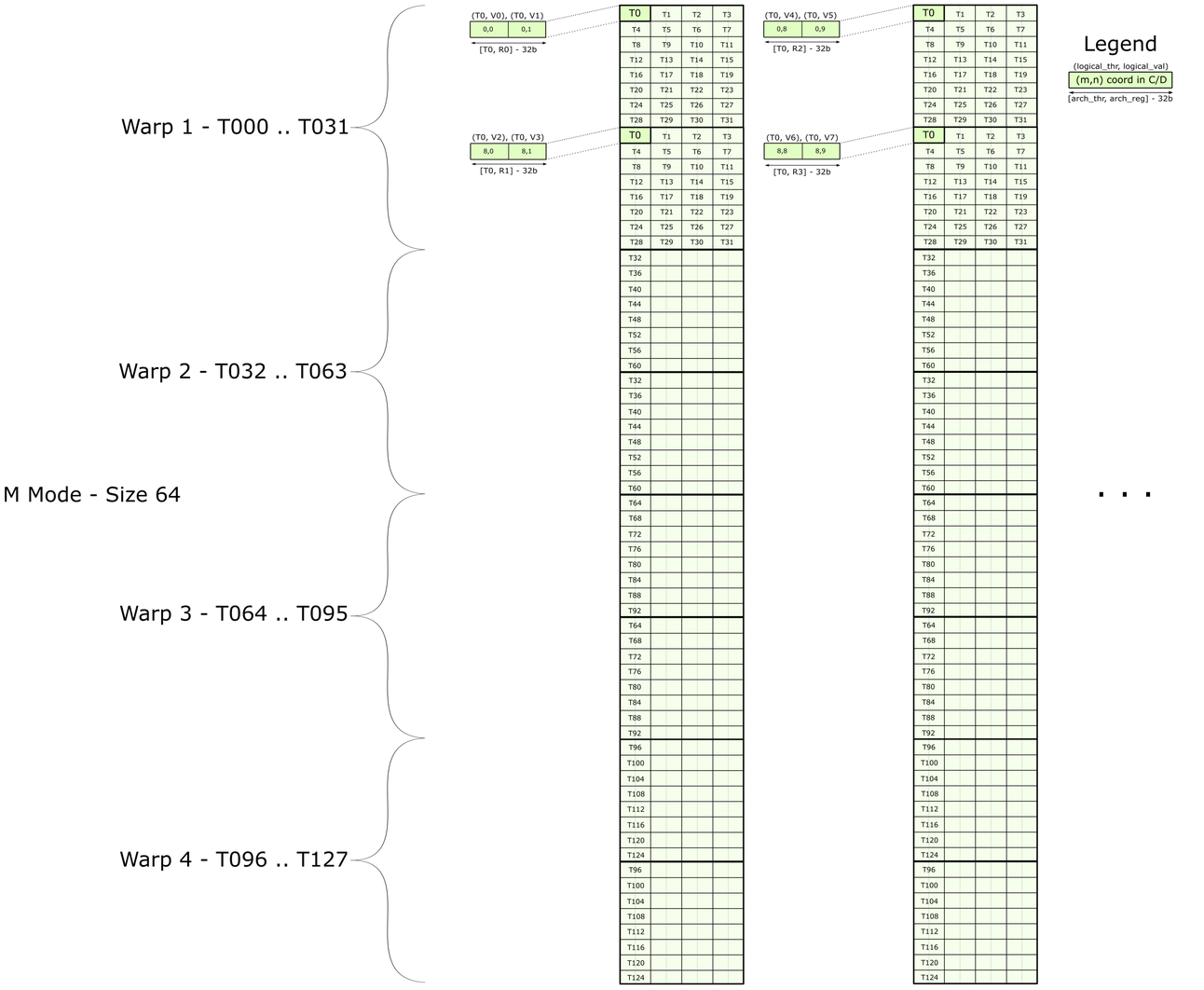
- Ampere TC is warp-wide
  - 32 threads
  - 16x8x16
  - 8 clocks

- But our SMs have 4 sub-partitions that run in parallel
- Already issuing 4 separate MMAs across them



SM90 WGMMA spans all 4 sub-partitions





GMMA Warpgroup Level C/D Matrix Layout (16b dstfmt)

Configurable N dimension in range(8, 256, 8)

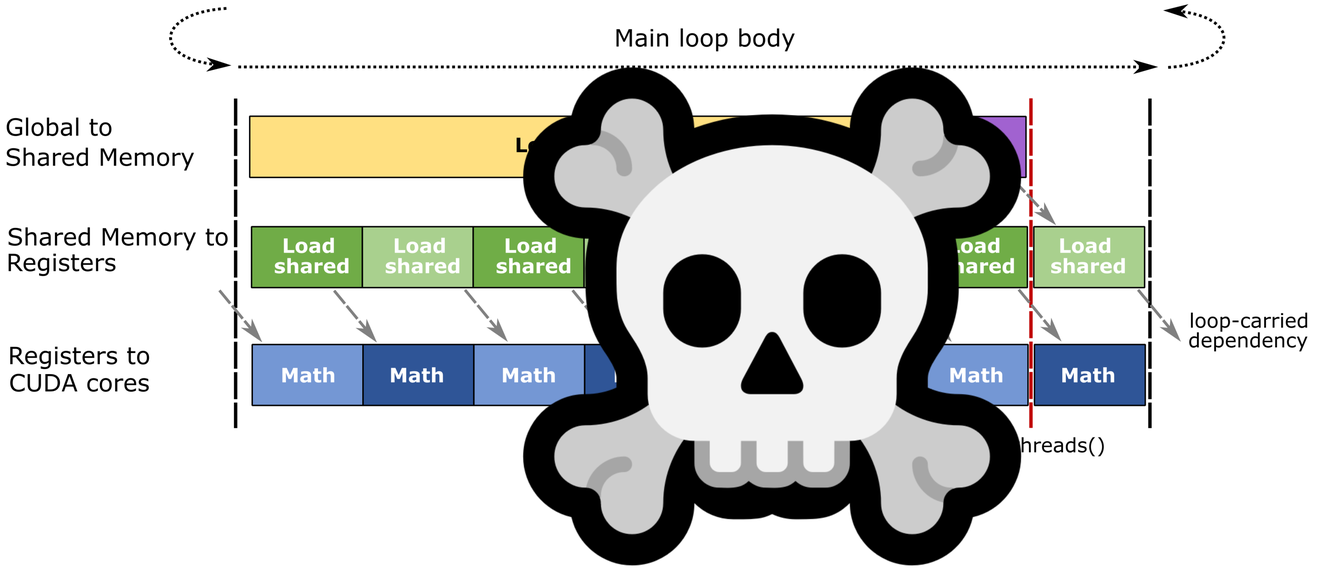
But how do we feed the beast?

- We cut B matrix reads by a factor of 4 BUT
- The instruction is now massive
  - Ampere: 16x8x16
  - Hopper: 64x128x16
  - 8x more data per MMA
- And it runs 2x faster per clock

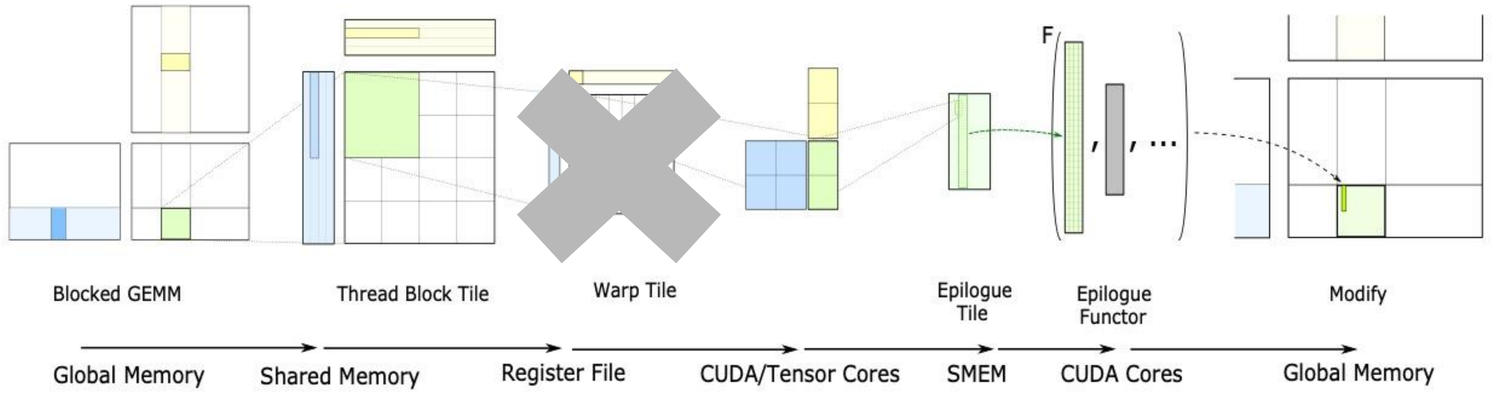
- Could we load 64x16 + 128x16 tiles for each into RMEM?

- We have to keep (64x16) + (128x16) + (64x128) live in registers
  - Blows up register budgets

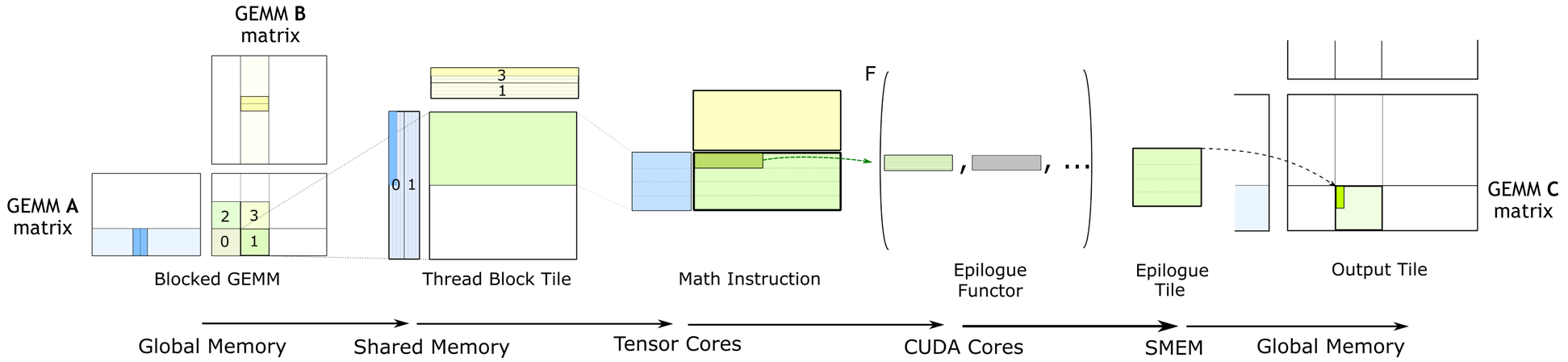
- **Exposes the entire load latency – Ahmdal's law**



What's new in Hopper



Tensor cores reuse data directly from in shared memory



Corollary: the tensor core is now explicitly async too in the same way LDGSTS is!

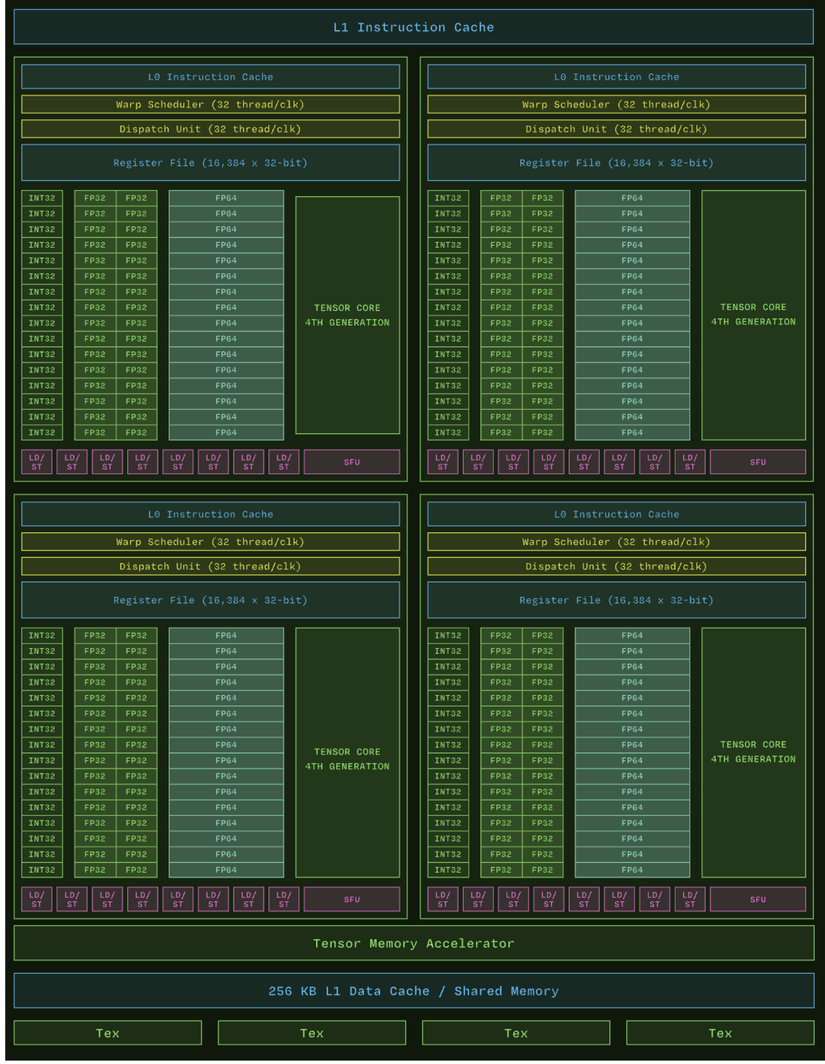
We strong scaled our tensor core

- Fed it with data from smem directly
- Made it async since its so big now
- Victory?!

- But if you were to run this on H100 silicon (LDGSTS + GMMA)
- You would get no extra performance *at all*

- We still have latencies to contend with
- GMEM load latencies did not improve gen-on-gen
- If anything they got worse
- Moving data around in wires cannot get faster
- How do we tolerate higher gmem latencies?
  - Builder smaller chips?
  - Stack memory?
  - Grow caches?

- **Asynchrony is our best countermeasure to Ahmdal's law**



A100 SMEM: 164 KiB / SM

1.56x

H100 SMEM: 256 KiB / SM

NVIDIA

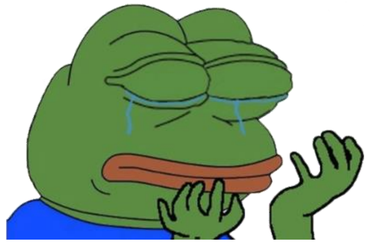
Are we there yet?

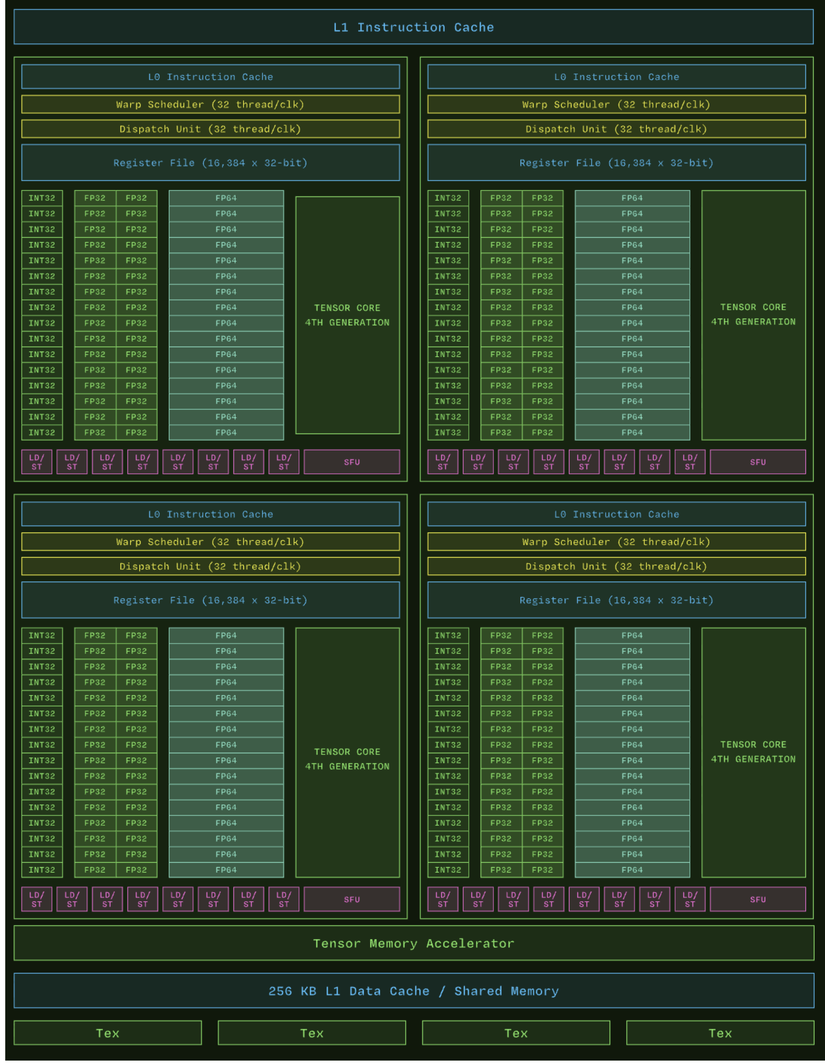
- We strong scaled our tensor core
- Fed it with data from smem directly
- Grew pipeline stage counts by 1.56x (3->4/5)

- Still not enough, but we get 20% faster
- How do we get even faster?
  - MOAR cache!
  - Give up on 2 CTA / SM occupancy

- Write kernels with 1 CTA / SM occupancy
- Use all the 256 KiB of SMEM for 1 CTA
- Gets us to 6/7 stages – good enough!

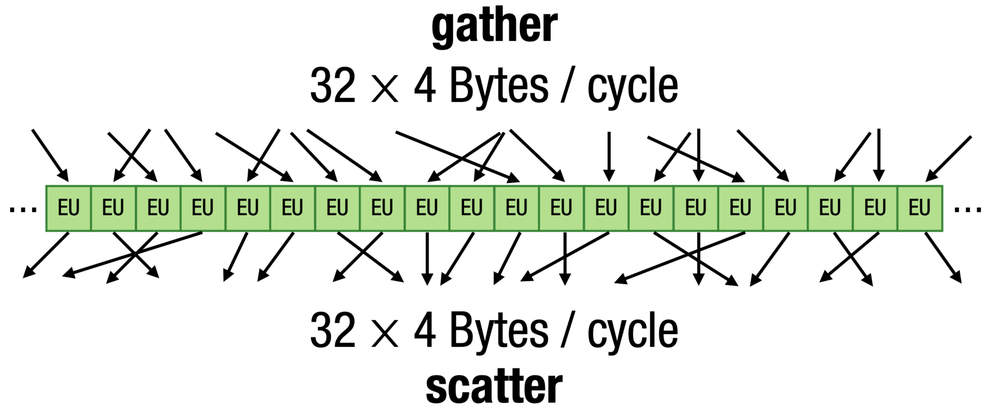
- But exposes the epilogue and prologue …
  - We will come back to this later, let’s press on for now
- We have bigger issues for now





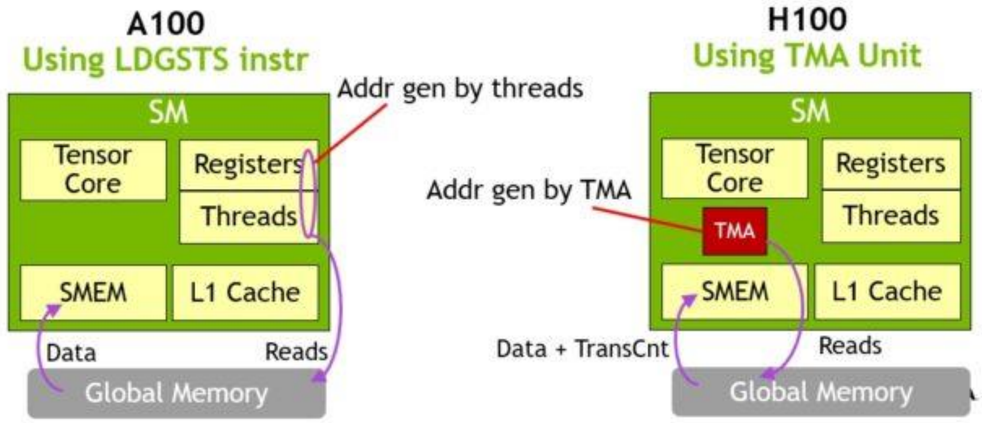
We have not improved GMEM loads

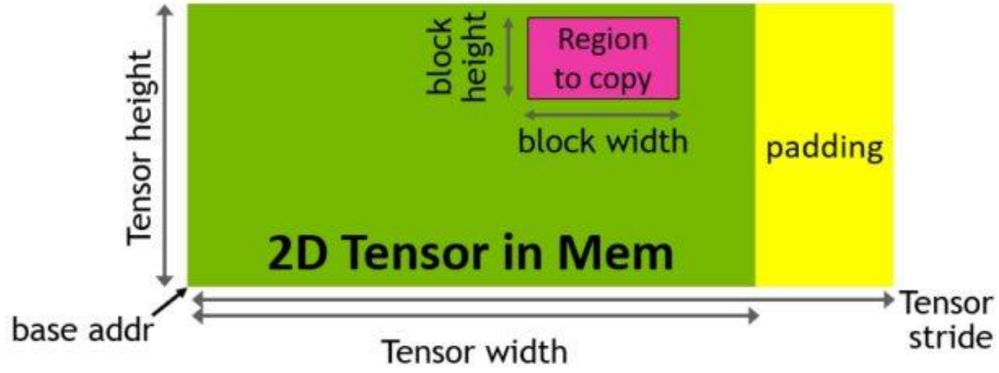
- LDGSTS is now the main bottleneck in many ways:
  - Uses too many vector registers
  - Uses too many issue slots for address computation
  - Predication is hard to do and takes up predicate registers



- But we know what layout of smem our tensor core is going to consume
  - Not much programmer freedom here
  - We don't need the generality of SIMT load

- Introduce a new data loading engine specifically for affine tensor load/store
- Introduce a "Uniform Datapath" and "Uniform Registers"

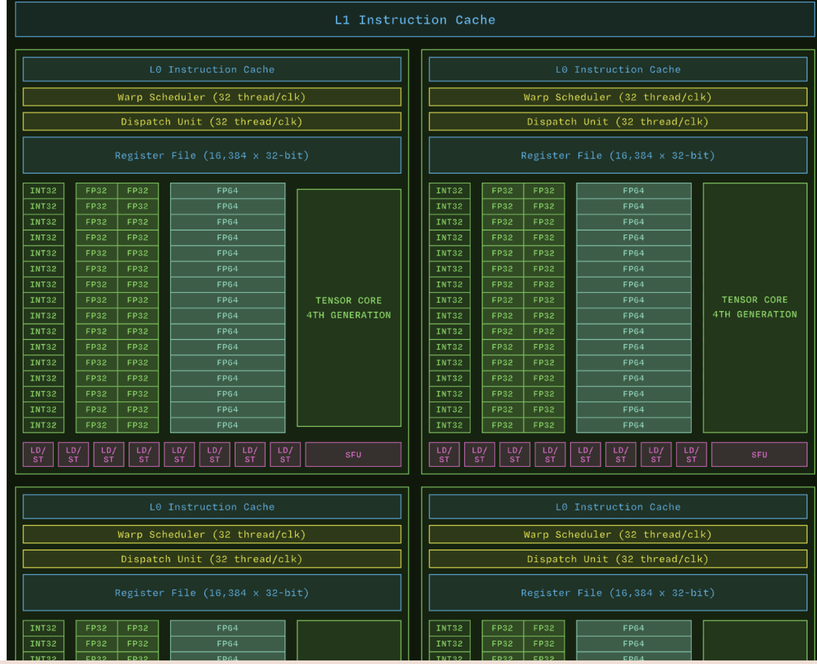




This solves our register spills

And large tile perf is now SOL but …

- Small tile shape performance still sucks
- You profile the workload and find out that the loop overheads are killing you now
- Overhead clocks longer latency than the tile shape MMA clocks
  - Setting up descriptors, incrementing offsets, arriving on barriers etc.

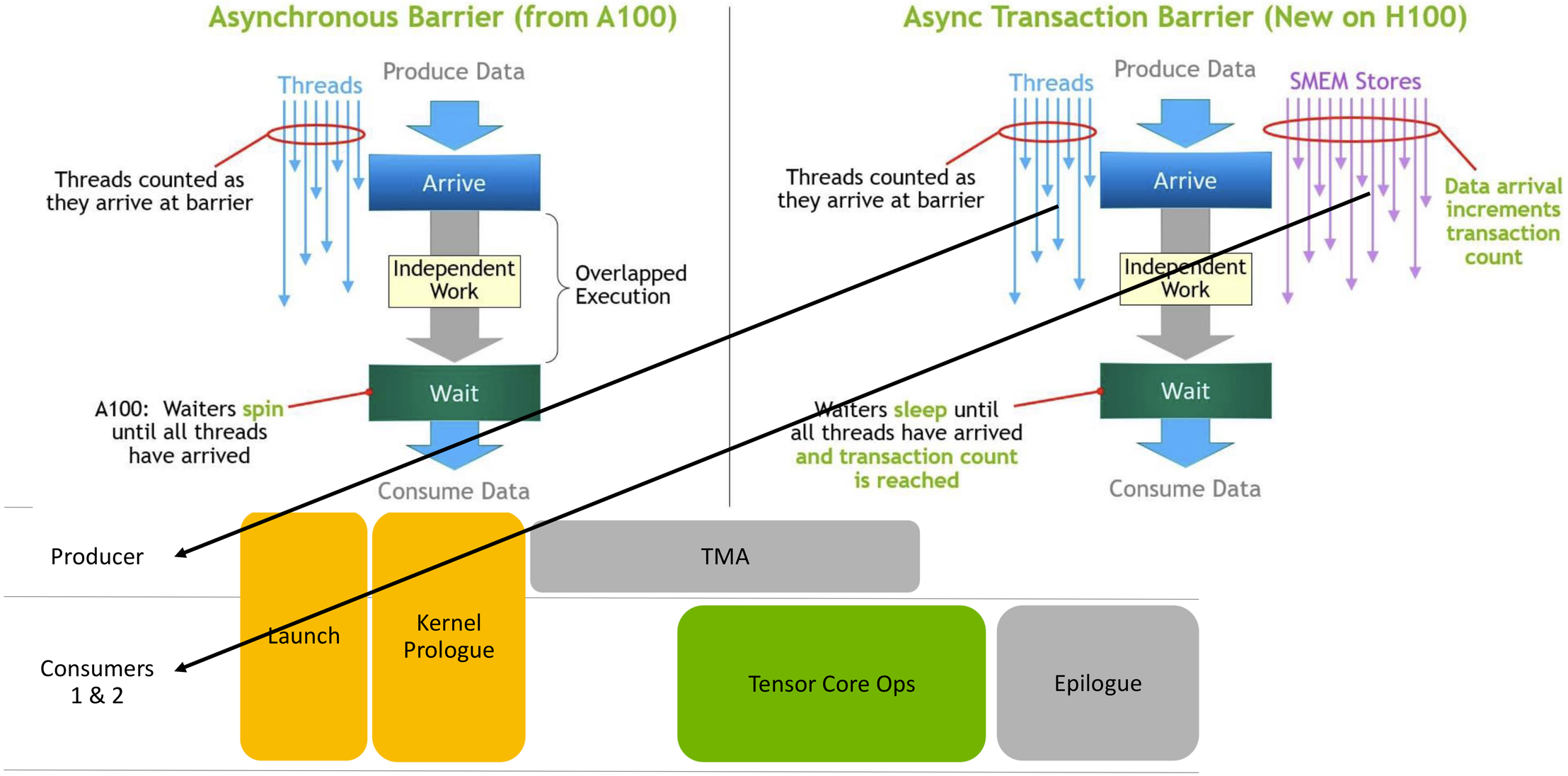


- How do we fix this?
- We could just use big tile shapes only – bigger MMAs are async for longer
- BUT that’s not strong scaling
- How do we strong scale?

- Core problem: TMA -> MMA -> TMA takes too many cycles
- We are bound by issue latencies of a long serial program

- These are not true deps – barriers are what establish ordering already
- What if we could break the loops into separate loops to issue asap!?
- TMA is single thread and we have warp-scheduling already?

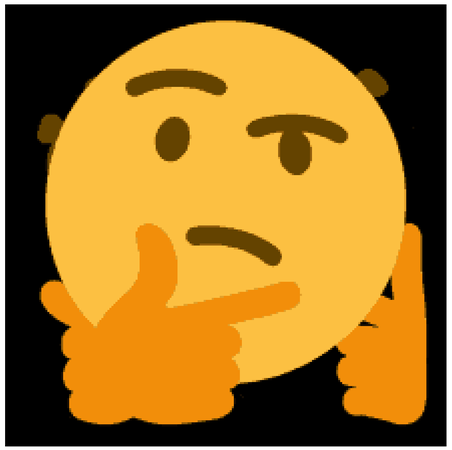
What if we split the TMA and MMA loops into separate loops?



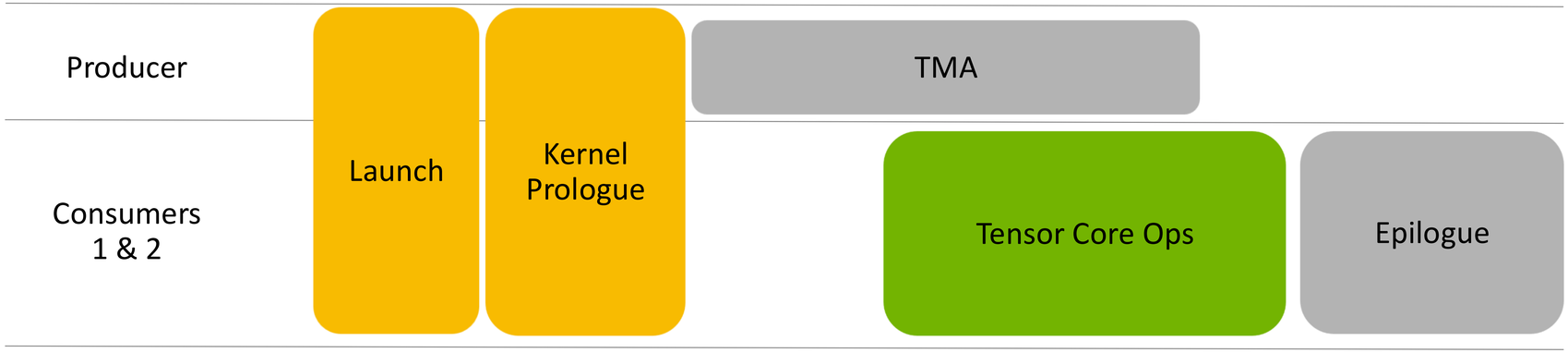
But we are spilling registers again!!!

Warp-Specialized Kernels

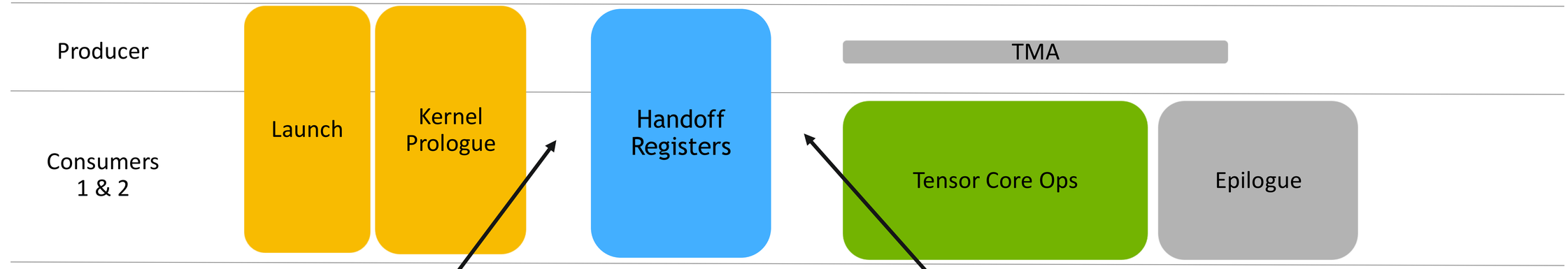
- 256 threads are still needed for MMA to use all 512 registers in the SM
- But we now added a new workgroup for TMA for a total of 384 threads / CTA
- CUDA programming model requires all warps use the same number of registers
- round(512 / 3, 8) = 168 registers / thread
- Not enough for our massive MMA output matrices + other epilogue stuff



- Grow register file? But that would be so wasteful, we just added uniform registers
- TMA uses no registers – all 168 registers in that warpgroup are wasted



What if we could reallocate registers between warps?



Start off all threads in the CTA with 168 registers / thread

Donate register ownership from TMA to MMA threads the “prologue”

Phew, are we there now??

- We have strong scaled our tensor core
- TMA: data loading accelerator with minimal overheads
- Added new barriers for sub-CTA sync
- Implemented register reconfiguration
- Changed kernels to be warp-specialized
- So like any good engineer, you go and profile your workload

- Only to find that for very large problems, L2 read b/w is our bottleneck now
- Solution: widen L2 read ports?

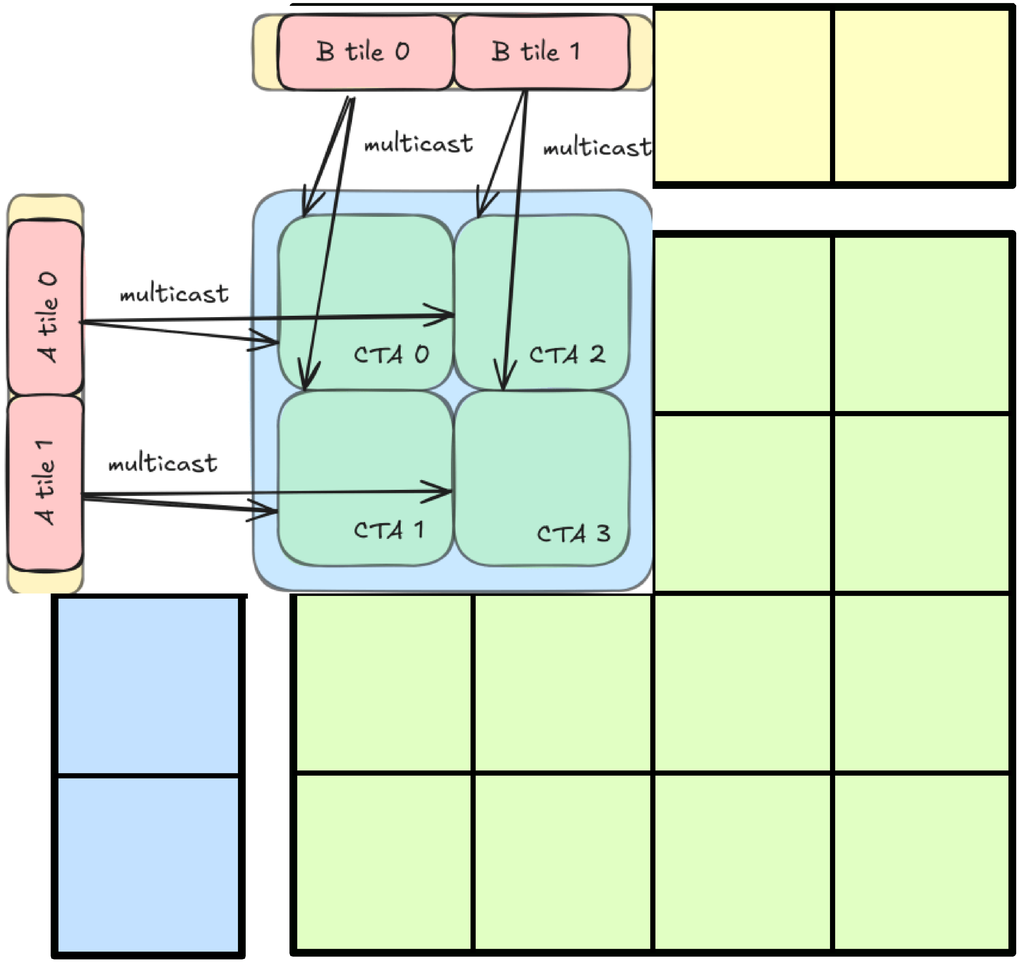
Overcoming L2 b/w limits

Exploit the workload properties

- L2 <> GPC is a *massive* crossbar – widening it is not an option
- Recall, we have infinite arithmetic intensity
- How can we exploit that to get L2 data reuse?

- We have many separate CTAs
- We usually tile the output MxN in a 2D grid
- Projections of CTAs along A and B load the same data
- Our GPU already has GPCs as a hierarchy

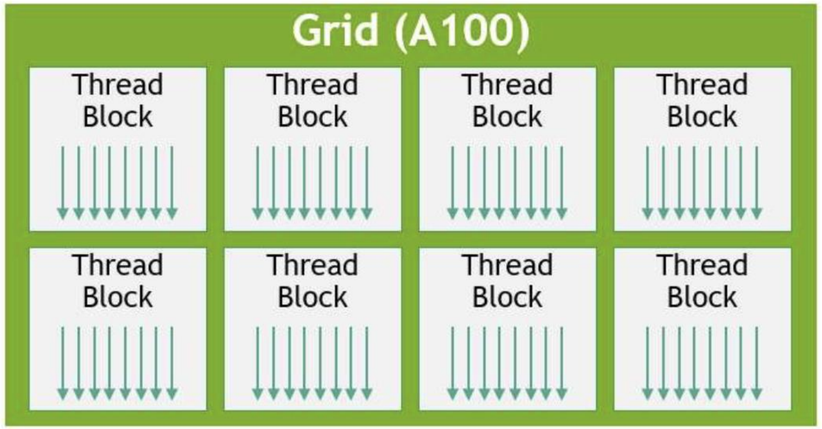
- What if we could load from L2 only once to the GPC
- And then broadcast that data to all the SMs that need it?

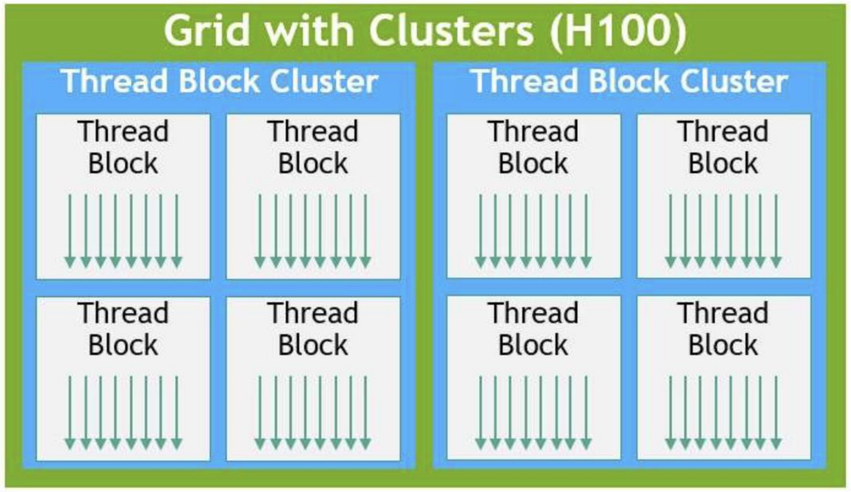


But how do we even represent a program that does this?

- We do have GPCs that could support this
- But CUDA says all CTAs are concurrent and independently scheduled
- We have no way of forcing simultaneous execution
- Solution: expose this as a programmable hierarchy

CUDA View	HW Hierarchy
Grid	GPU
Cluster	GPC
Thread Block	SM
Threads	Lanes





Aside: Think about our barriers

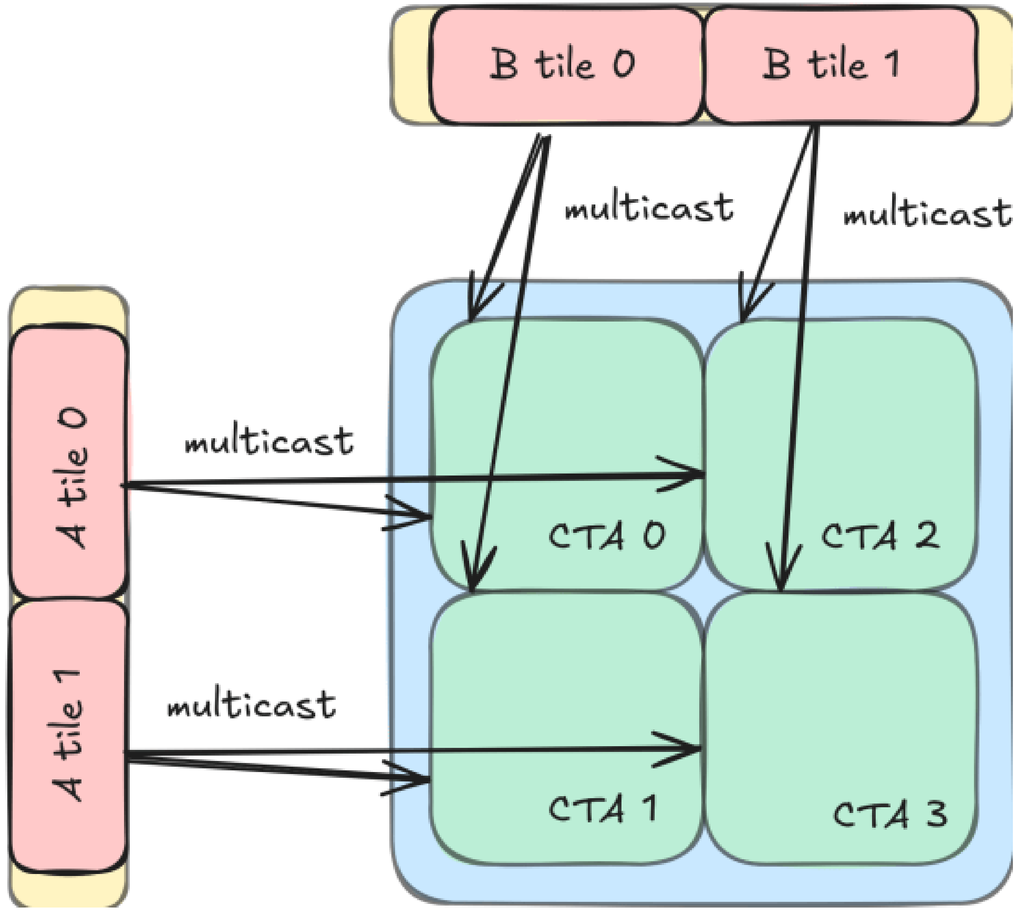
- CTA0 is loading half of A for both CTAs 0 and 2
- CTA0 is loading half of B for both CTAs 0 and 1

- With our current barriers, how do we wait on CTA0 from CTA 1/2?

- Solution: Atomic in global memory?
- Solution: Atomic in L2?
- Solution: DSMEM – distributed shared memory

- CTAs can treat remote SMEM as local SMEM
- PGAS programming model
- Local crossbar within a GPC – much more palatable

- This allows our barriers to arrive on remote barriers
- Wait is still on local barrier only



Anatomy of an H100 GEMM

Speed of Light Hopper Mainloop

ASYNC in everything

MMA

```
// Mainloop
CUTLASS_PRAGMA_NO_UNROLL
for ( ; k_tile_count > 0; --k_tile_count)
{
  // WAIT on smem_pipe_read until its data are available
  auto barrier_token = pipeline.consumer_try_wait(smem_pipe_read);
  pipeline.consumer_wait(smem_pipe_read, barrier_token);

  int read_stage = smem_pipe_read.index();
  warpgroup_fence_operand(accum);
  warpgroup_arrive();
  // Unroll the K mode manually to set scale D to 1
  CUTLASS_PRAGMA_UNROLL
  for (int k_block = 0; k_block < size<2>(tCrA); ++k_block) {
    // (V,M,K) x (V,N,K) => (V,M,N)
    cute::gemm(tiled_mma, tCrA(_,_,k_block,read_stage), tCrB(_,_,k_block,read_stage), accum);
    tiled_mma.accumulate_ = GMMA::ScaleOut::One;
  }
  warpgroup_commit_batch();

  /// Wait on the GMMA barrier for K_PIPE_MMAS (or fewer) outstanding
  ///   to ensure smem_pipe_write is consumed
  warpgroup_wait<K_PIPE_MMAS>();
  warpgroup_fence_operand(accum);

  // UNLOCK smem_pipe_release, done _computing_ on it
  pipeline.consumer_release(smem_pipe_release);

  // Advance smem_pipe_read and smem_pipe_release
  ++smem_pipe_read; ++smem_pipe_release;
}
```

```
// Wait on PIPE to be read
SYNCS.PHASECHK.TRANS64.TRYWAIT smem_bar_ptr_full, phase;

// Fill up MMA command pipeline
WARPGROUP.ARRIVE
HGMMA.64x128x16.F16 Rc, sdesc[URab].tnspA.tnspB, Rc, UP0
UIADD3.32 sdesc[Urab+0], #immA
UIADD3.32 sdesc[Urab+1], #immB
// repeat 1xHGMMA+2xUIDADD3 above 1/3/7 more times
WARPGROUP.DEPBAR.LE gsb0, 0x1

// Release SMEM stage for TMA to load into
SYNCS.ARRIVE.A1T0 smem_bar_ptr_empty_sm0, phase;
SYNCS.ARRIVE.A1T0 smem_bar_ptr_empty_sm1, phase;
```

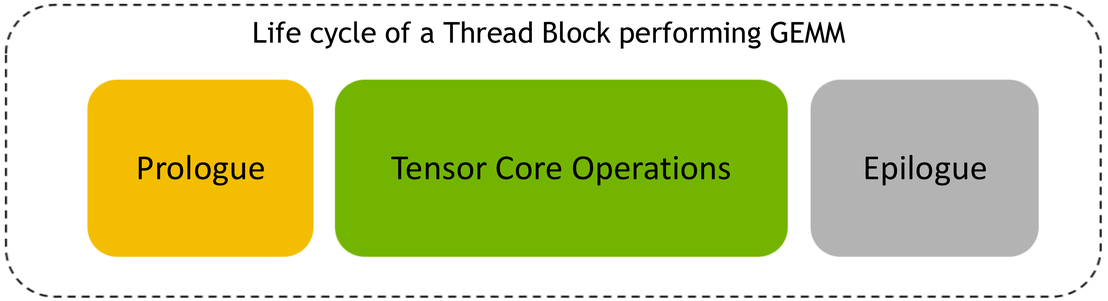
TMA

```
// TMA threads
// Acquire for PIPE+N to be consumed
SYNCS.TRY_WAIT.PHSCK smem_bar_ptr, phase;

// Set txcount for PIPE+N
SYNCS.ARRIVE.TRANS64 smem_bar_ptr, txcount;

// Issue TMA loads that will commit themselves
UTMALDG.2D URa, UR[smem_ptr], smem_bar_ptr, UR[gdesca]
UTMALDG.2D URb, UR[smem_ptr], smem_bar_ptr, UR[gdescb]
```

```
// Mainloop
CUTLASS_PRAGMA_NO_UNROLL
for ( ; k_tile_count > 0; --k_tile_count)
{
  // LOCK smem_pipe_write for _writing_
  pipeline.producer_acquire(smem_pipe_write);

  //
  // Copy gmem to smem for *k_tile_iter
  //

  using BarrierType = typename MainloopPipeline::ProducerBarrierType;
  BarrierType* tma_barrier = pipeline.producer_get_barrier(smem_pipe_write);

  int write_stage = smem_pipe_write.index();
  copy(mainloop_params.tma_load_a.with(*tma_barrier, mcast_mask_a),
       tAgA(_,_,_,*k_tile_iter), tAsA(_,_,_,write_stage));
  copy(mainloop_params.tma_load_b.with(*tma_barrier, mcast_mask_b),
       tBgB(_,_,_,*k_tile_iter), tBsB(_,_,_,write_stage));
  ++k_tile_iter;

  // Advance smem_pipe_write
  ++smem_pipe_write;
}
```

End-to-end performance

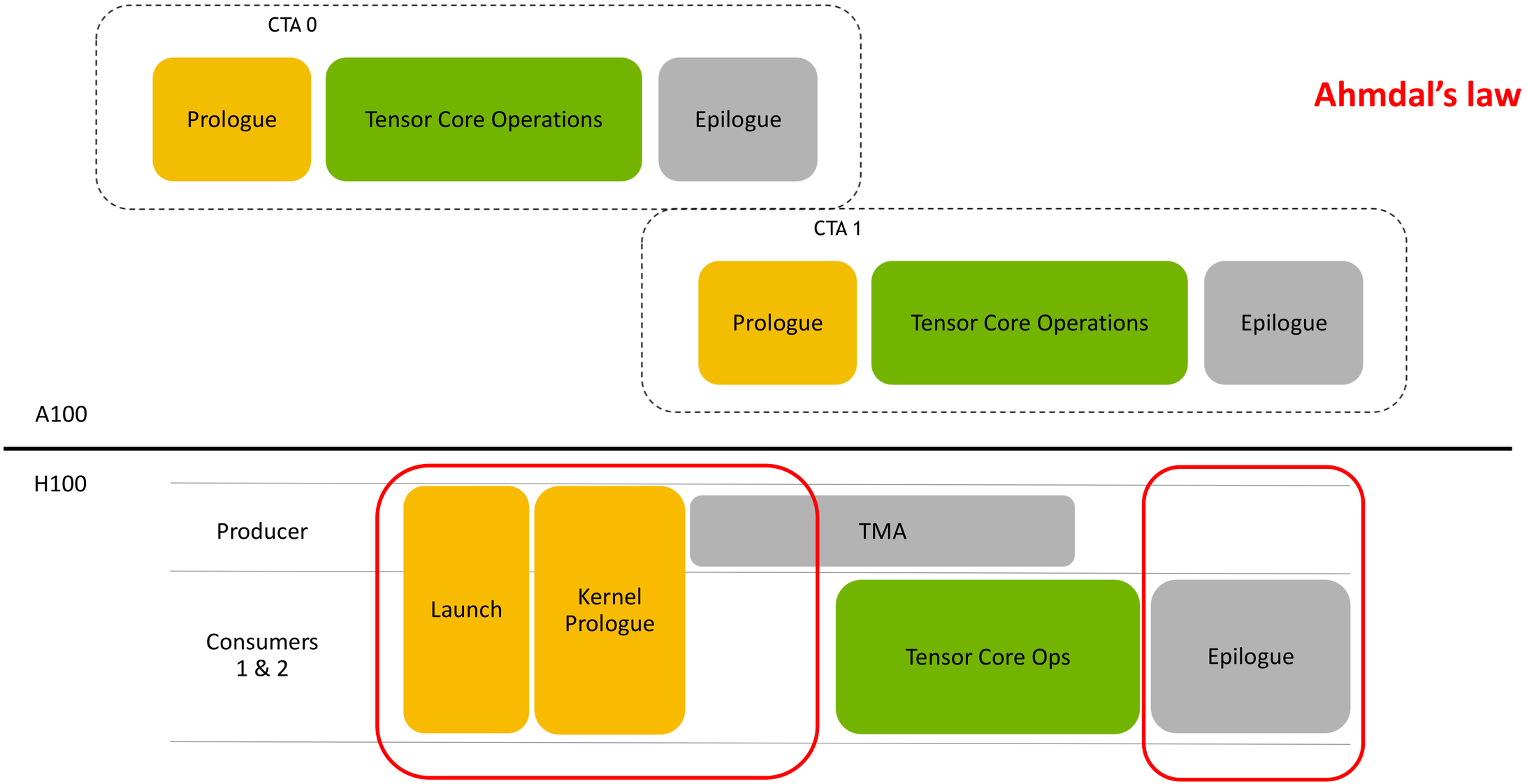
Life of a GEMM kernel



- Prologue and Epilogue are components of the GEMM kernel which involve non-tensor core operations and often latency or bandwidth bound.
- Typically hidden via multiple concurrently running ThreadBlock / SM
- With deep software pipelines – it becomes a tricky problem (due to lack of shared memory capacity)

We lost our 2 CTA / SM occupancy

Don’t have enough SMEM/RMEM for > 1 CTA / SM

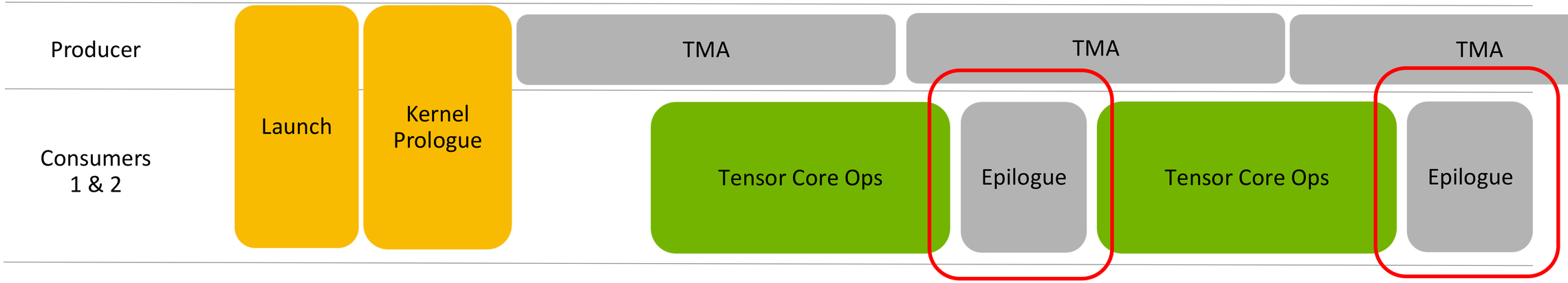


Ahmdal’s law

Warp-Specialized Persistent Kernels

- Idea: amortize the fixed costs across multiple output tiles
- Instead of launching all the CTAs, only Lauch as many CTAs as the number of SMs
- Implement tile scheduling in software instead of relying on `blockIdx.x/y/z`

Ahmdal's law



Producer	Consumer #1	Consumer #2
PersistentTileSchedulerSm90 **scheduler(problem_shape, blk_shape, cluster_shape)**		

```
// Data in via TMA

while (work_tile_info.is_valid_tile) {
  collective_mainloop.dma()
  scheduler.advance_to_next_work()
  work_tile_info = scheduler.get_current_work()
}
```

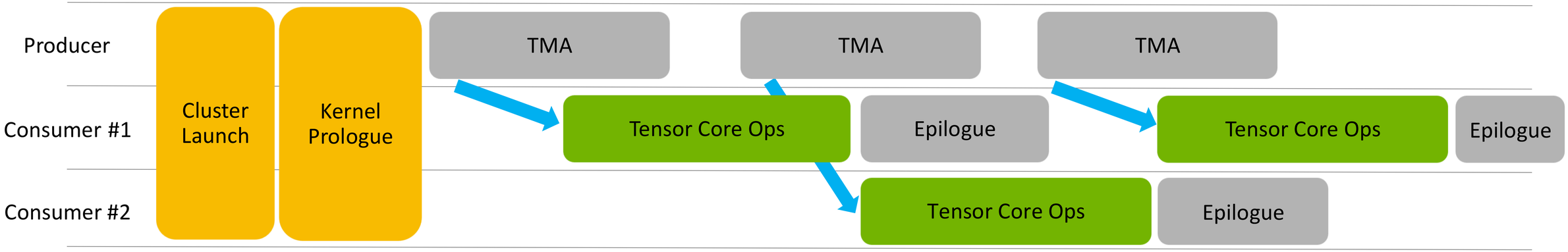
```
// Mainloop, epilogue, and data out

while (work_tile_info.is_valid_tile) {
  collective_mainloop.mma()
  scheduler.advance_to_next_work(NumConsumers)
  work_tile_info = scheduler.get_current_work()
}
```

How do we hide the epilogue now?

Warp-specialized Ping Pong Persistent Schedule

- Observe: we already have software persistent scheduling now
- What if we pipelined the execution of multiple tiles w.r.t. each other?

- Ping Pong: 2 consumers warp-groups alternate between math and epilogue of different work tiles
- Tradeoff: smaller tile shapes for better epilogue hiding
- Heuristic: smaller K shapes better with ping pong, larger better with non-ping-pong (cooperative)

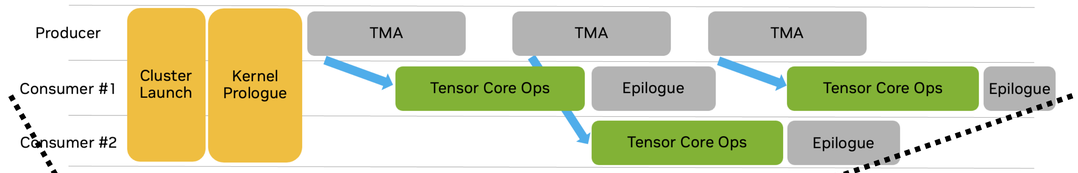


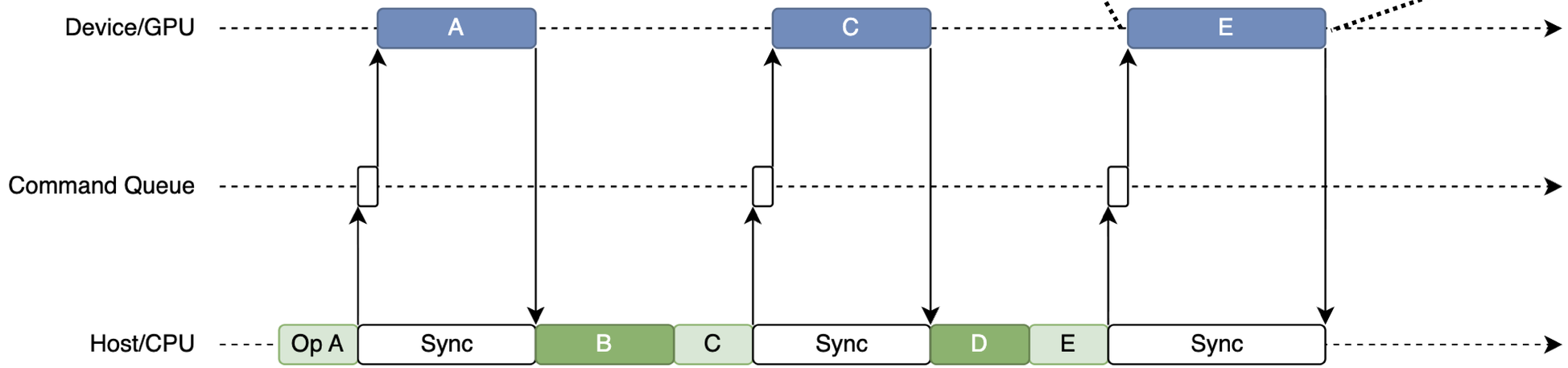
End to end optimization requires overlapping across kernels

Optimizing a model graph end to end

- Persistence only helps for multi—wave problems
- Strong scaling for single wave problems is hard
- The problem is exacerbated every time you add more SMs

- End to end optimization requires overlapping across kernels

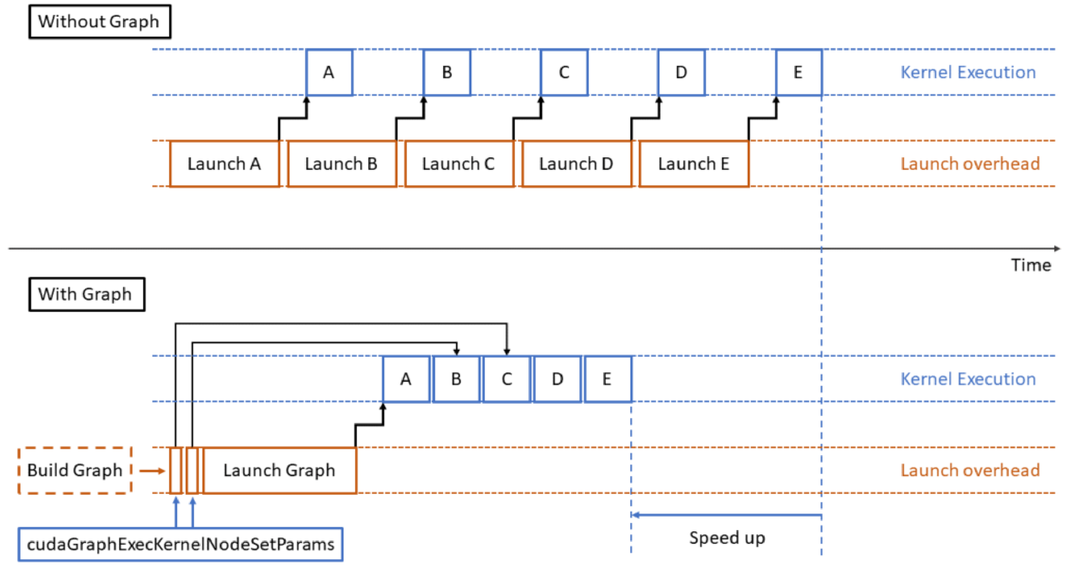




End to end model optimization

Zooming out beyond just a single kernel

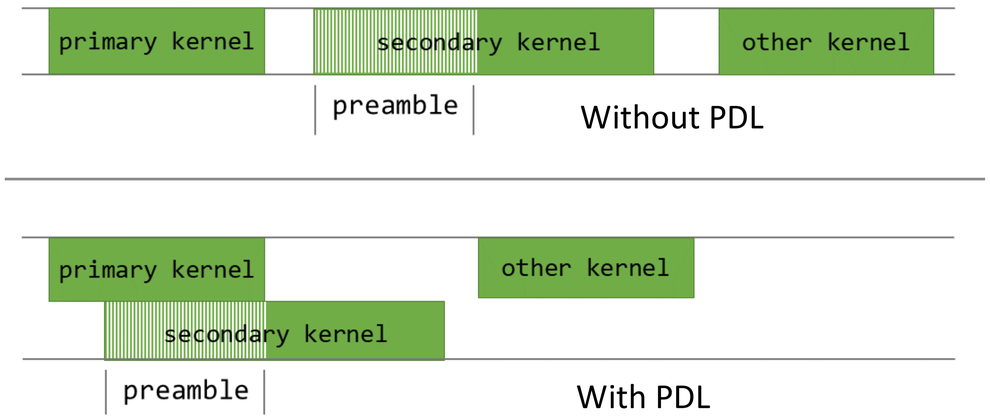
- Grid launches take 2-6 microseconds
- This sounds tiny but if you have tons of small kernels b2b it adds up
- What if we could enqueue an entire graph of kernels onto the GPU?
- CUDA Graphs!



B2B grids with and without CUDA graphs

- Cold start prologue can be long
- Even within a stream, there are finer grained dependencies
- E.g. inference kernels only need to wait on activations
  - Weights are unchanging and can be loaded
- Programmatic Dependent Launch (PDL) forgoes member.gpu
- Kernels can launch without satisfying stream ordering
- Programmer inserts dep points in the kernels

B2B grids with and without PDL



https://developer.nvidia.com/blog/constructing-cuda-graphs-with-dynamic-parameters/

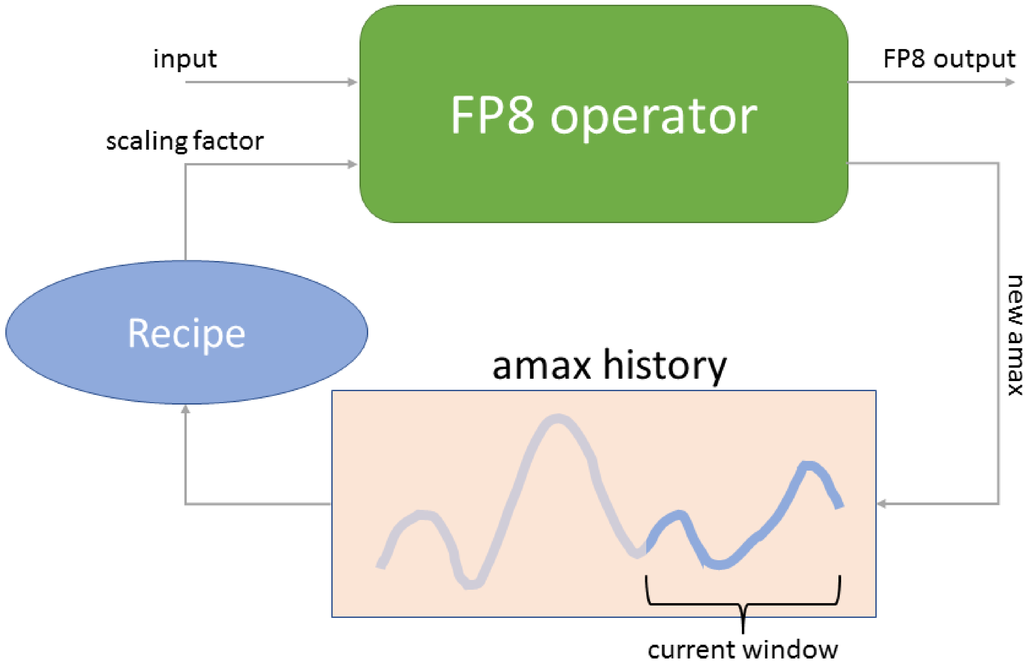
Workload Codesign

16b floats are so wasteful

What if we could run with 8b floats instead

- Lower precision add/mul is way faster to scale with
- Why?
- Data movement is sin
- Multiplicative FLOP/s gain for bit width reduction => higher perf/W

- But convergence gets really hard at low precisions
- Solution: co-design the model training recipe

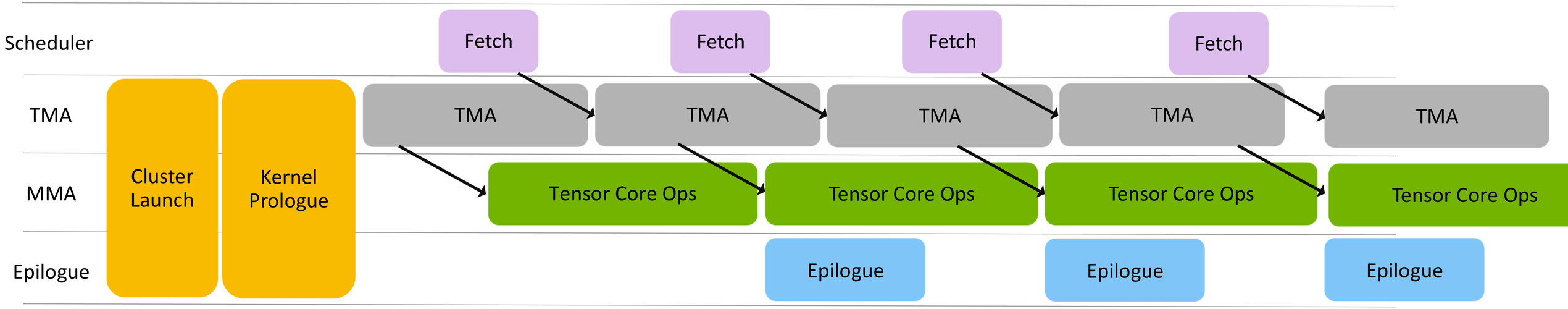



NVIDIA
Blackwell

Blackwell kernel schedule at a glance

Generalization of the Hopper ping-pong kernel enabled by TMEM

- Accumulators being in TMEM allows for separate threads to access it as a shared resource
- Concurrent execution of MMA and epilogue on different work units



Conclusion

First principles are all you need to know
Everything else is all *details*
But the details are so so fun :D