TLX:

Minimally Invasive Paths to Performance Portability

Hongtao Yu (Meta)

Outline

- Motivation
- Overview of TLX
- Building blocks
- Kernel performance study
- What’s Next
- Key takeaways

Extending Triton’s Power

- Triton is great
  - Exceptional productivity through tile-level abstraction and automatic thread mapping
  - Hardware-agnostic design — write once, run anywhere
  - Delivers effortless high performance on CUDA cores
- Triton can be greater if
  - Attainable path to ultimate performance on tensor cores
  - Stable and predictable performance in production
  - Fast path to new hardware features amid short GPU lifecycles

TLX : Triton Low-level Language Extension

- Extends Triton DSL with low-level hardware-specific primitives
- Exposes fine-grained control over hardware features for performance
- (+) Abstracts away unnecessary hardware details (e.g., layout encoding)
- (+) Seamless integration with Triton DSL and compiler
- (+) Preserves Triton’s composable semantics
- (+) Enables incremental development and performance tuning

Example: Dot Compression++

Computes $X@(X^T@Y)$

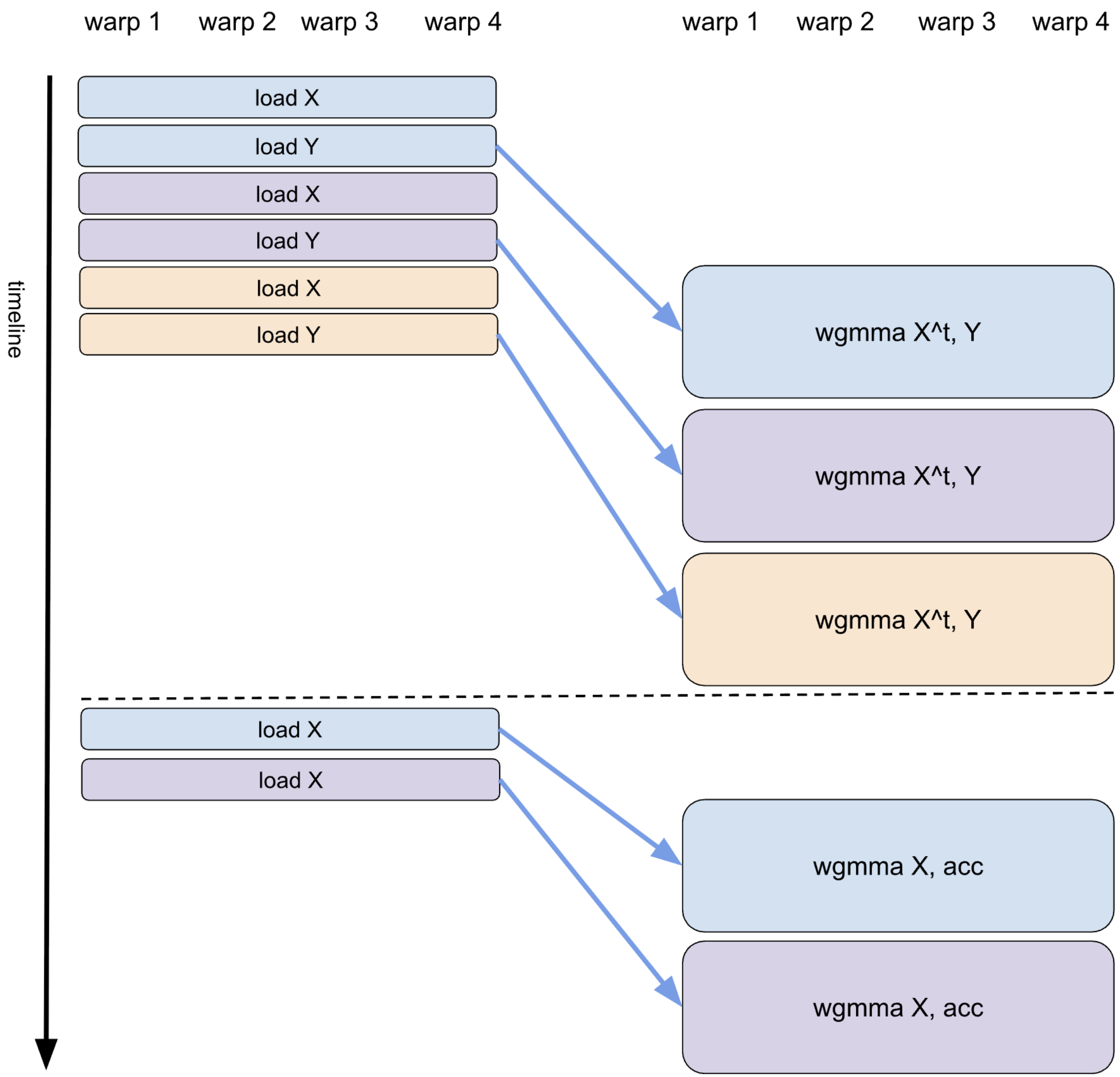
```
@triton.jit
def dcpp(x_desc, y_desc, o_desc, BLOCK_K: tl.constexpr,
):
 pid = tl.program_id(0)

# compute X^T @ Y
for k in range(0, tl.cdiv(K, BLOCK_K)):
  x = x_desc.load([k + pid * K, 0])
  y = y_desc.load([k + pid * K, 0])
  acc = tl.dot(x.T, y, acc)

acc = acc.to(tl.float16)

# compute X @ (X^T @ Y)
for k in range(0, tl.cdiv(K, BLOCK_K)):
  x = x_desc.load([k + pid * K, 0])
  acc = tl.dot(x, acc)
  o_desc.store([k + pid * K, 0], out.to(tl.float16))
```

Example: Dot Compression++

num_warps=4

Computes X@(X^T@Y)

```
@triton.jit
def dcpp(x_desc, y_desc, o_desc, BLOCK_K: tl.constexpr,
):
 pid = tl.program_id(0)

# compute X^T @ Y
for k in range(0, tl.cdiv(K, BLOCK_K)):
  x = x_desc.load([k + pid * K, 0])
  y = y_desc.load([k + pid * K, 0])
  acc = tl.dot(x.T, y, acc)

acc = acc.to(tl.float16)

# compute X @ (X^T @ Y)
for k in range(0, tl.cdiv(K, BLOCK_K)):
  x = x_desc.load([k + pid * K, 0])
  acc = tl.dot(x, acc)
  o_desc.store([k + pid * K, 0], out.to(tl.float16))
```



warp 1 warp 2 warp 3 warp 4

warp 1 warp 2 warp 3 warp 4

timeline

load X

load Y

load X

load Y

load X

load Y

wgmma X^t, Y

wgmma X^t, Y

wgmma X^t, Y

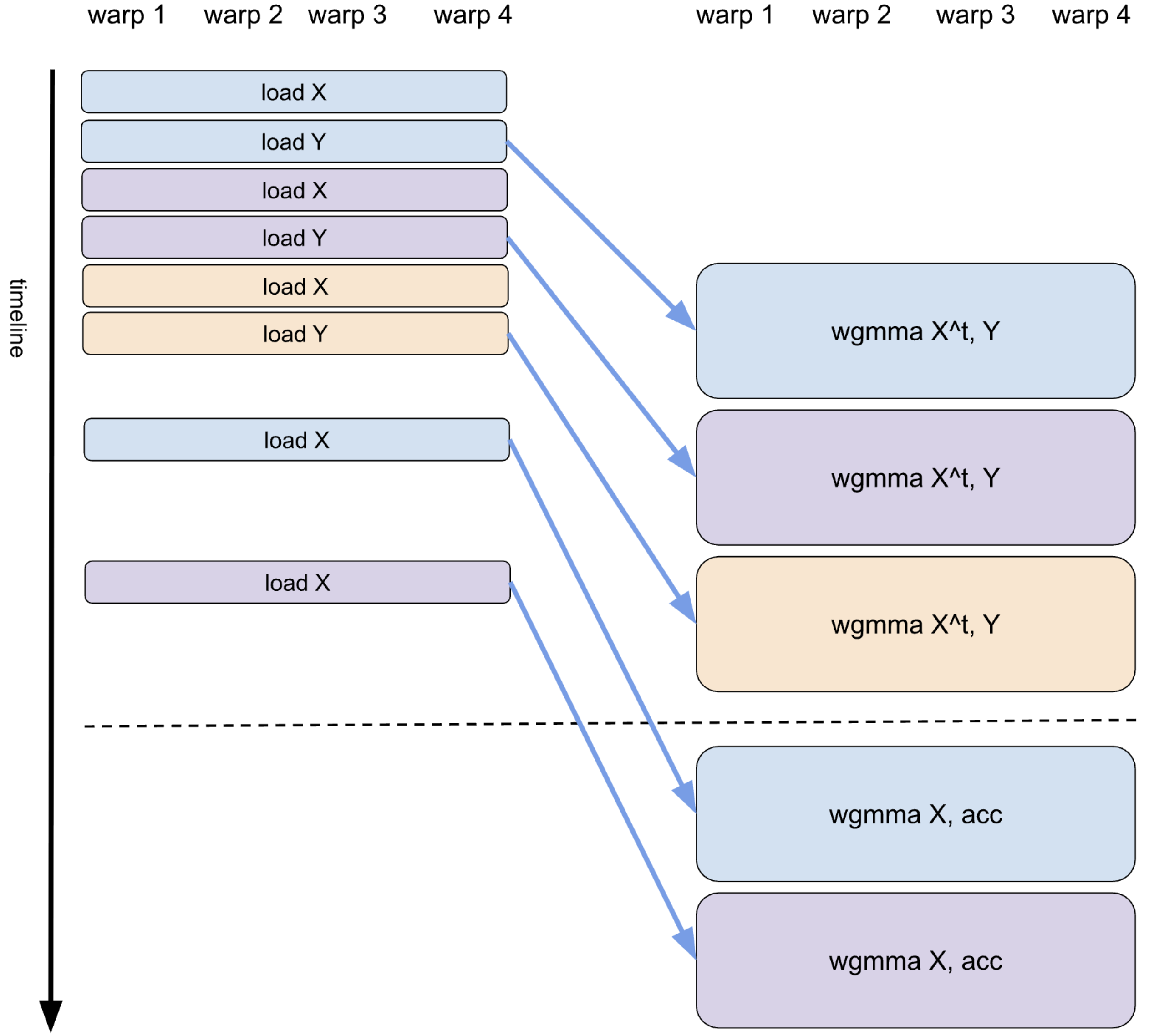
load X

load X

wgmma X, acc

wgmma X, acc

Continuous Pipelining



20% speedup

Example: Dot Compression++

Computes X@(X^T@Y)

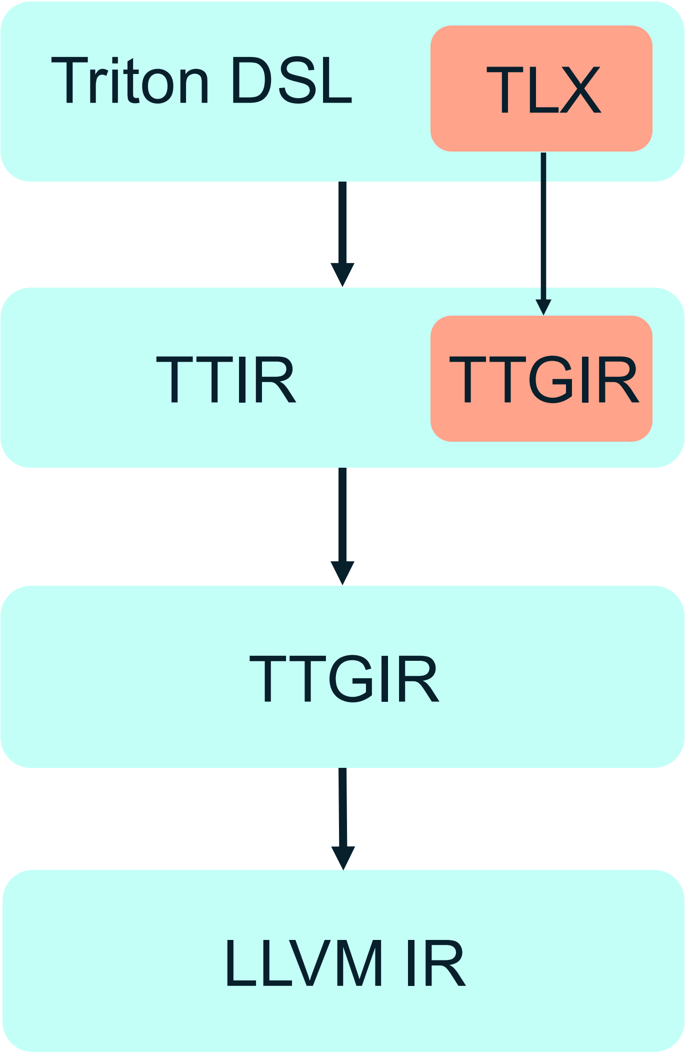
```
@triton.jit
def dcpp(x_desc, y_desc, o_desc, BLOCK_K: tl.constexpr,
):
 pid = tl.program_id(0)

# compute X^T @ Y
for k in range(0, tl.cdiv(K, BLOCK_K)):
  x = x_desc.load([k + pid * K, 0])
  y = y_desc.load([k + pid * K, 0])
  acc = tl.dot(x.T, y, acc)

acc = acc.to(tl.float16)

# compute X @ (X^T @ Y)
for k in range(0, tl.cdiv(K, BLOCK_K)):
  x = x_desc.load([k + pid * K, 0])
  acc = tl.dot(x, acc)
  o_desc.store([k + pid * K, 0], out.to(tl.float16))
```

```
@triton.jit
def dcpp_tlx(x_desc, y_desc, o_desc ..., NB: tl.constexpr):
  pid = tl.program_id(0)
  xt= tlx.local_alloc((BM, BK), tlx.dtype_of(x_desc), NB)
  y = tlx.local_alloc((BK, BN), tlx.dtype_of(y_desc), NB)
  x= tlx.local_alloc((BM, BK), tlx.dtype_of(x_desc), NB, reuse=x)
  bar = tlx.alloc_barriers(NB)

  # prefetch X^T and Y
  for k in tl.static_range(0, NB - 1):
    _load_x_y(x_desc, y_desc, xt, y, bar, k, k + pid * K)

  k_iters = tl.cdiv(K, BK)
  for k in range(0, k_iters – NB + 1):
    acc = _dot_xt_y(x, y, acc, bar, k)
     # prefetch X^T and Y
    _load_x_y(x_desc, y_desc, xt, y, bar, k, k + pid * K)

  for k in tl.static_range(k_iters – NB + 1, k_iters):
    acc = _dot_xt_y(x, y, acc, bar, k)
    # prefetch X
    _load_x(x_desc, x, bar, k + NB – 1, k - k_iters + NB - 1 + pid * K)

  acc = tlx.async_dot_wait(0, acc)
  acc = acc.to(tl.float16)
  for k in range(0, tl.cdiv(K, BK)):
    tlx.barrier_wait(bar[k], (k + NB – 1) // NB)
    out = tlx.async_dot(x[k], y[k], acc)
    _load_x(x_desc, x, bar, k + NB – 1, k - k_iters + NB - 1 + pid * K)
    out = tlx.async_dot_wait(0, out)
    o_desc.store([k + pid * K, 0], out.to(tl.float16))
```

```
@triton.jit
def _load_x_y(x_desc, y_desc, x, y, bar, idx, offset)
  buf = idx % NB
  tlx.barrier_expect_bytes(bar[buf], (BM + BN) * BK)
  tlx.async_descriptor_load(x_desc, x[buf], [offset, 0], bar[buf])
  tlx.async_descriptor_load(y_desc, y[buf], [offset, 0], bar[buf])


@triton.jit
def _dot_xt_y(x, y, acc, bar, idx)
  buf = idx % NB
  tlx.barrier_wait(bar[buf], phase = buf // NB)
  acc = tlx.async_dot(tlx.local_trans(x[buf]), y[buf], acc)
  return tlx.async_dot_wait(1, acc)


@triton.jit
def _load_x(x_desc, x, bar, idx, offset)
  buf = idx % NB
  tlx.barrier_expect_bytes(bar[buf], 2 * BM * BK)
  tlx.async_descriptor_load(x_desc, x[buf], [offset, 0], bar[buf])
```

TLX Operations for NVGPU

- Local memory accesses
- Asynchronous global memory accesses
- Asynchronous tensor core operations
- Asynchronous task operation for warp specialization
- Cross-warp group synchronization



Automatic Layout Encoding Assignment

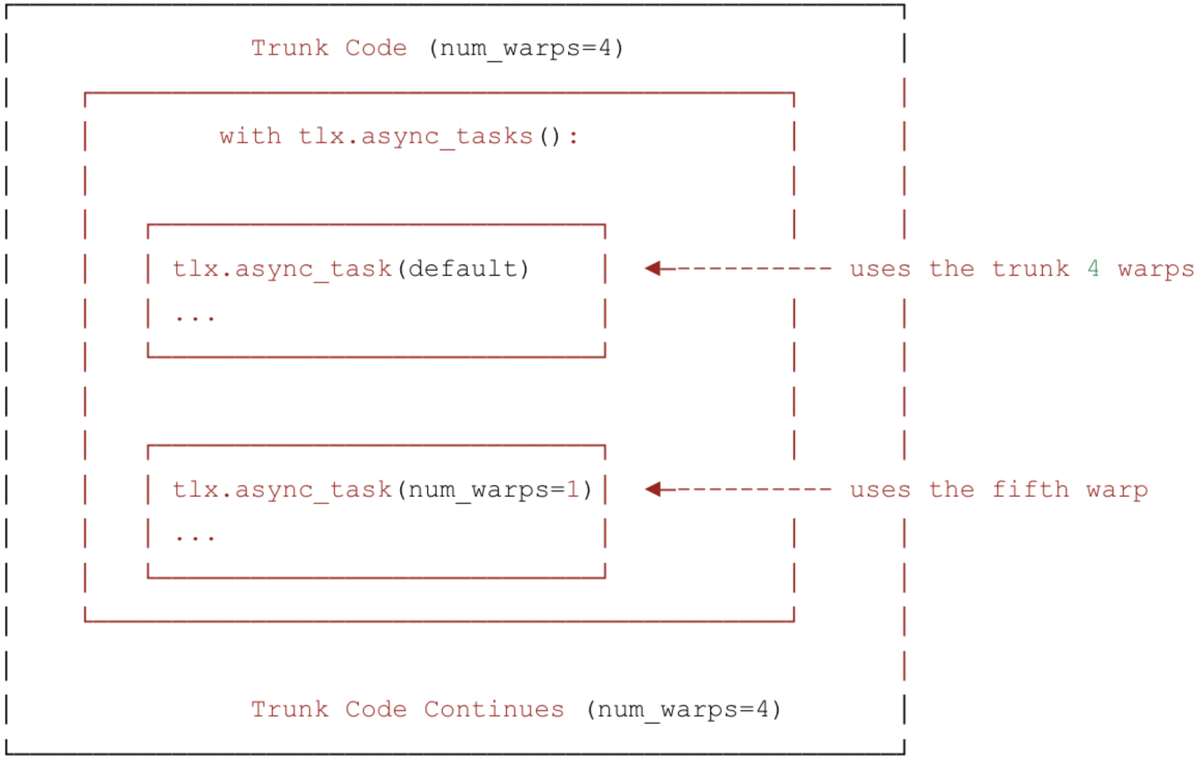
- For register tensors (tl.tensor)
  - Non-coalesced placeholder layout introduced during TTIR -> TTGIR
  - Optimized by the memory coalescing pass; or
  - Inferred based on the MMA operations;
  - Reconciled by the removal of layout conversion pass
- For local-memory buffers (tlx.buffered_tensor)
  - Non-swizzled placeholder layout introduced during DSL -> TTIR
  - Required by consumers (MMA operations, …)
  - Propagated and reconciled by layout propagation pass, priority-aware

Outline

- Motivation
- Overview of TLX
- **Building blocks**
- Kernel performance study
- What's Next
- Key takeaways

Warp Specialization

```
# trunk code with num_warps=4
...

with tlx.async_tasks():
    with tlx.async_task("default"):
        ...
    with tlx.async_task(num_warps=1):
        ...

# trunk code
...
```



Trunk Code (num_warps=4)

with tlx.async_tasks():

tlx.async_task(default)
...

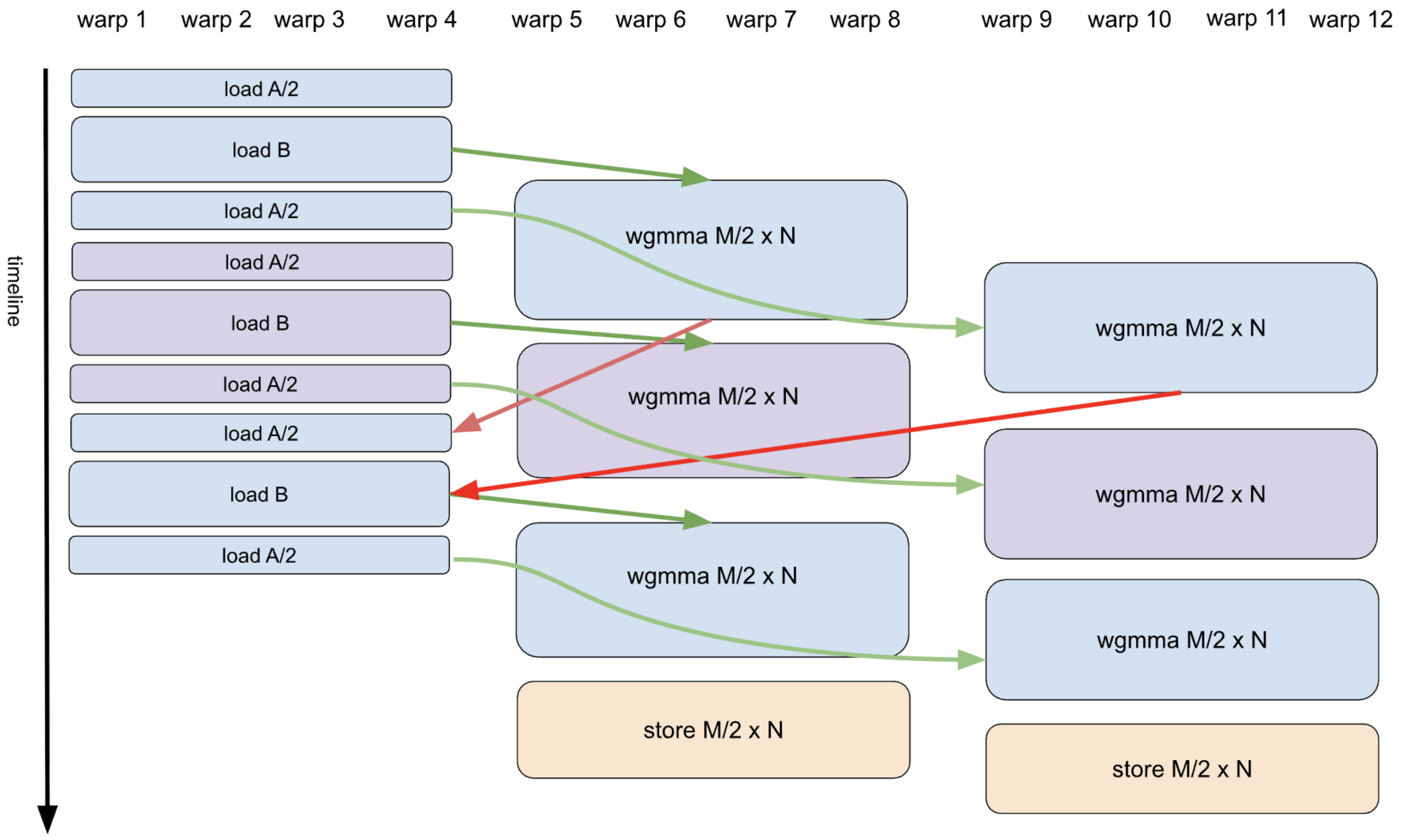
uses the trunk 4 warps

tlx.async_task(num_warps=1)
...

uses the fifth warp

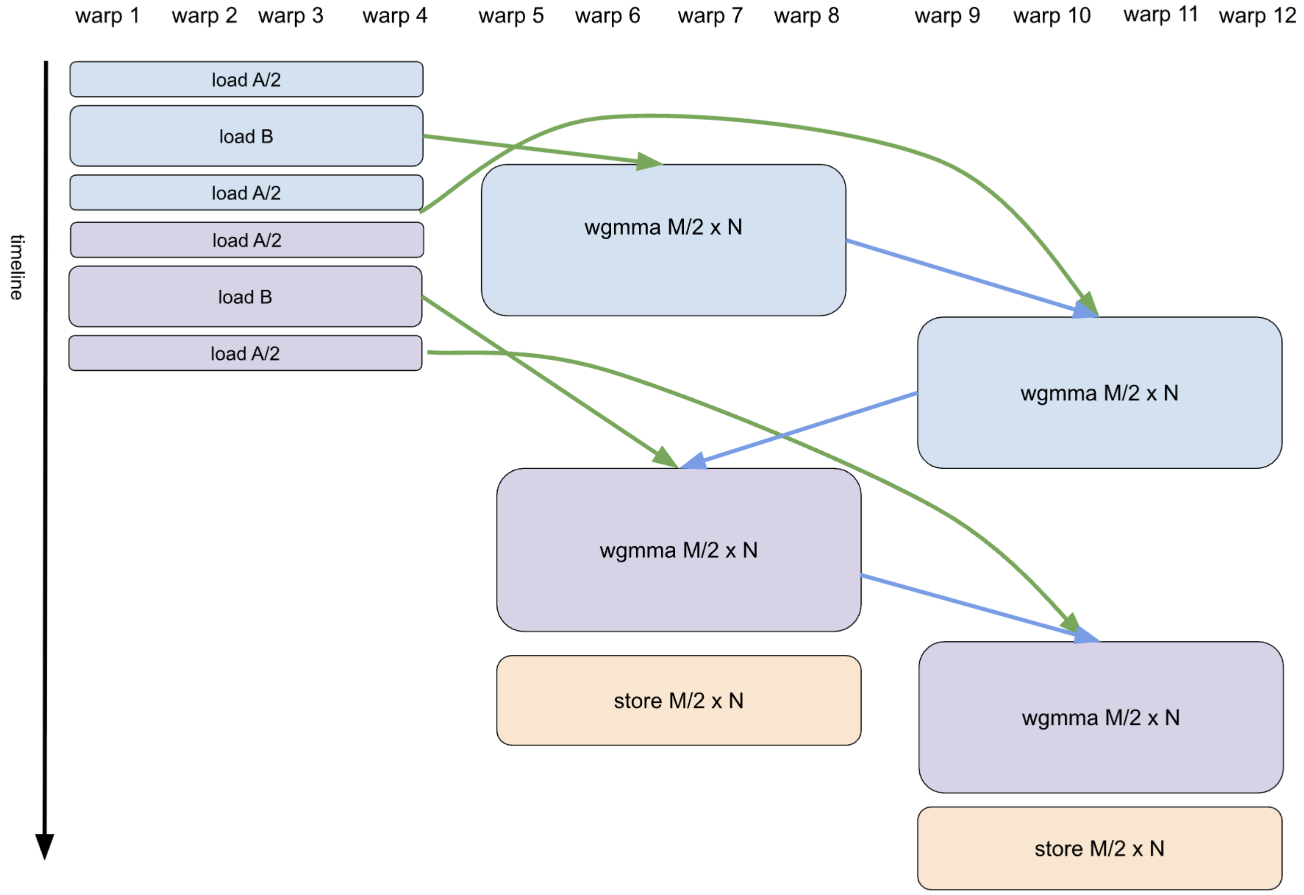
Trunk Code Continues (num_warps=4)

Cooperative Warp Specialization



```
@triton.jit
def _load(desc, buffer, bar_empty, bar_full, idx, size, offsets, NUM_BUFFERS)
    buf_id = idx % NUM_BUFFERS
    phase = idx // NUM_BUFFERS
    tlx.barrier_wait(bar_empty[buf_id], phase ^ 1)
    tlx.barrier_expect_bytes(bar_full[buf_id], size)
    tlx.async_descriptor_load(desc, x[buf_id], offsets, bar_full[buf_id])

@triton.jit
def hopper_gemm_ws(a_desc, b_desc, c_desc ... M, N, K, NB: tl.constexpr):

   a = tlx.local_alloc((BLOCK_M // 2, BLOCK_K), tlx.dtype_of(x_desc), NB*2)
   b = tlx.local_alloc((BLOCK_K, BLOCK_N), tlx.dtype_of(y_desc), NB)
   a_full = tlx.alloc_barriers(num_barriers=NB *2)
   a_empty = tlx.alloc_barriers(num_barriers=NB *2)
   b_full = tlx.alloc_barriers(num_barriers=NB, arrive_count=2)
   b_empty = tlx.alloc_barriers(num_barriers=NB, arrive_count=2)

   with tlx.async_tasks():
          with tlx.async_task("default") :
         for k in range(0, tl.cdiv(K, BK))
           _load(a_desc, a, a_empty, a_full, k, BM // 2 * BK, [k * BK])
           _load(b_desc, b, b_empty, b_full, k, BN * BK, [k * BK])
           _load(a_desc, a+NB, a_empty+NB, a_full+NB, k, BM // 2 * BK, [k * BK])

      with tlx.async_task(num_warps=4, num_regs=168, replicate=2):
         cid = tlx.async_task_replica_id()
         for k in range(0, tl.cdiv(K, BK) )
            buf_id = idx % NUM_BUFFERS
            phase = idx // NUM_BUFFERS
            tlx.barrier_wait(a_full[buf_id + cid * NB], phase)
            tlx.barrier_wait(b_full[buf_id], phase)
            acc = tlx.async_dot(a[buf_id]), b[buf_id], acc)
            acc = tlx.async_dot_wait(0, acc)
            tlx.barrier_arrive(a_empty[buf_id + cid * NB])
            tlx.barrier_arrive(b_empty[buf_id])
             o_desc.store([offset_am, offset_bn], acc)
```

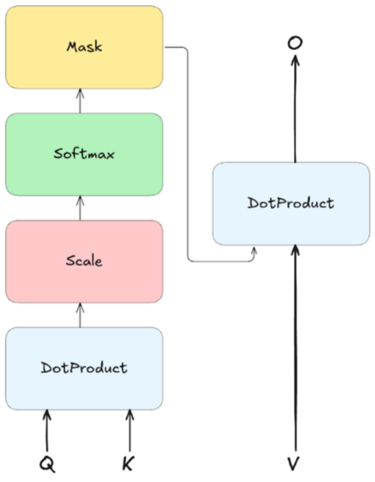
Ping-Pong Schedule



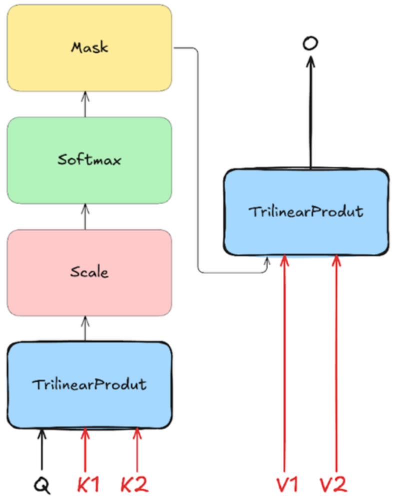
```
with tlx.async_tasks():
    with tlx.async_task("default") :
    for k in range(0, tl.cdiv(K, BK))
      _load(a_desc, a, a_empty, a_full, k, BM // 2 * BK, [k * BK])
      _load(b_desc, b, b_empty, b_full, k, BN * BK, [k * BK])
      _load(a_desc, a+NB, a_empty+NB, a_full+NB, k, BM // 2 * BK, [k * BK])

  with tlx.async_task(num_warps=4, num_regs=168, replicate=2):
    cid = tlx.async_task_replica_id()
    if cid == 1:
      tlx.named_barrier_arrive(9, 256)
    for k in range(0, tl.cdiv(K, BK))
       buf_id = idx % NUM_BUFFERS
       phase = idx // NUM_BUFFERS
       tlx.barrier_wait(a_full[buf_id + cid * NB], phase)
       tlx.barrier_wait(b_full[buf_id], phase)
       if cid == 0:
         tlx.named_barrier_wait(9, 256)
       else:
         tlx.named_barrier_arrive(10, 256)
       acc = tlx.async_dot(a[buf_id]), b[buf_id], acc)
       if cid == 0:
         tlx.named_barrier_arrive(10, 256)
       else:
         tlx.named_barrier_arrive(9, 256)
       acc = tlx.async_dot_wait(0, acc)
       tlx.barrier_arrive(a_empty[buf_id + cid * NB])
       tlx.barrier_arrive(b_empty[buf_id])
      o_desc.store([offset_am, offset_bn], acc)
```

TLX Kernel Performance Study

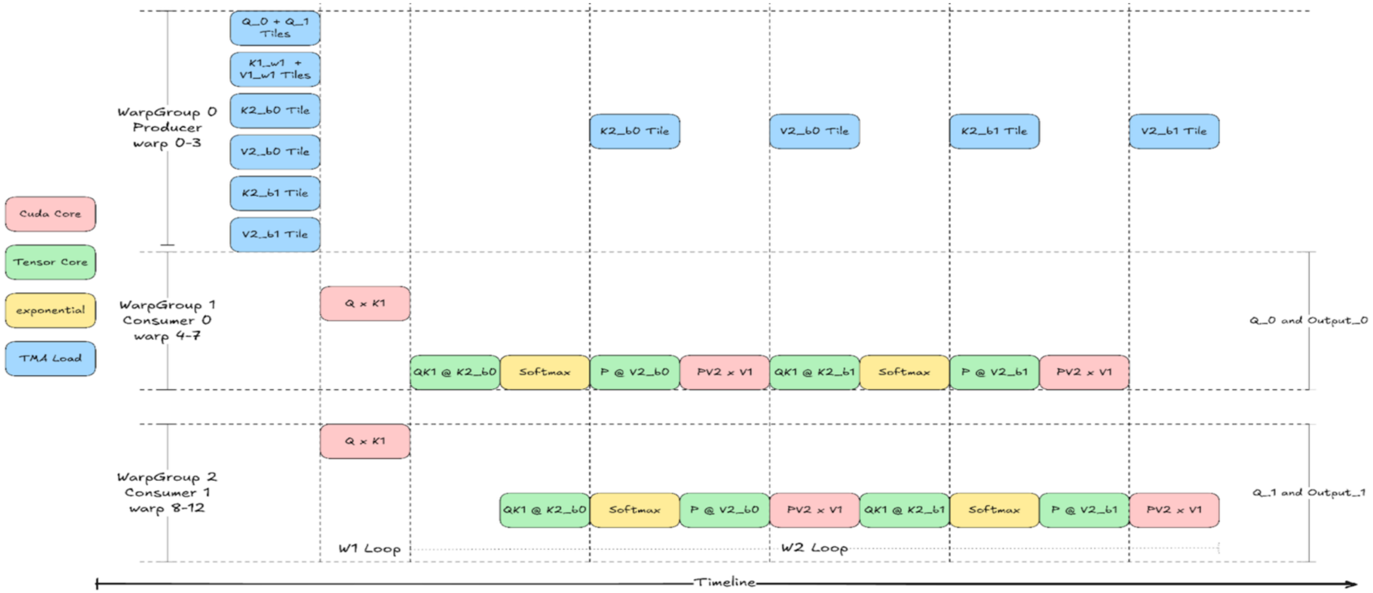
2-Simplicial Attention on H100: https://pytorch.org/blog/fast-2-simplicial-attention-hardware-efficient-kernels-in-tlx/

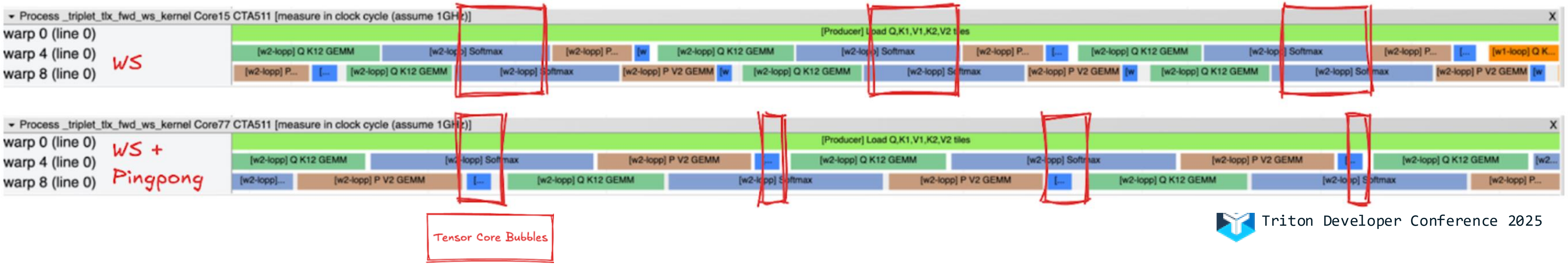


Standard Dot-Product Attention



2-Simplicial Attention




Process _triplet_tlx_fwd_ws_kernel Core15 CTA511 [measure in clock cycle (assume 1GHz)]

Process _triplet_tlx_fwd_ws_kernel Core77 CTA511 [measure in clock cycle (assume 1GHz)]

Tensor Core Bubbles

TLX Kernels

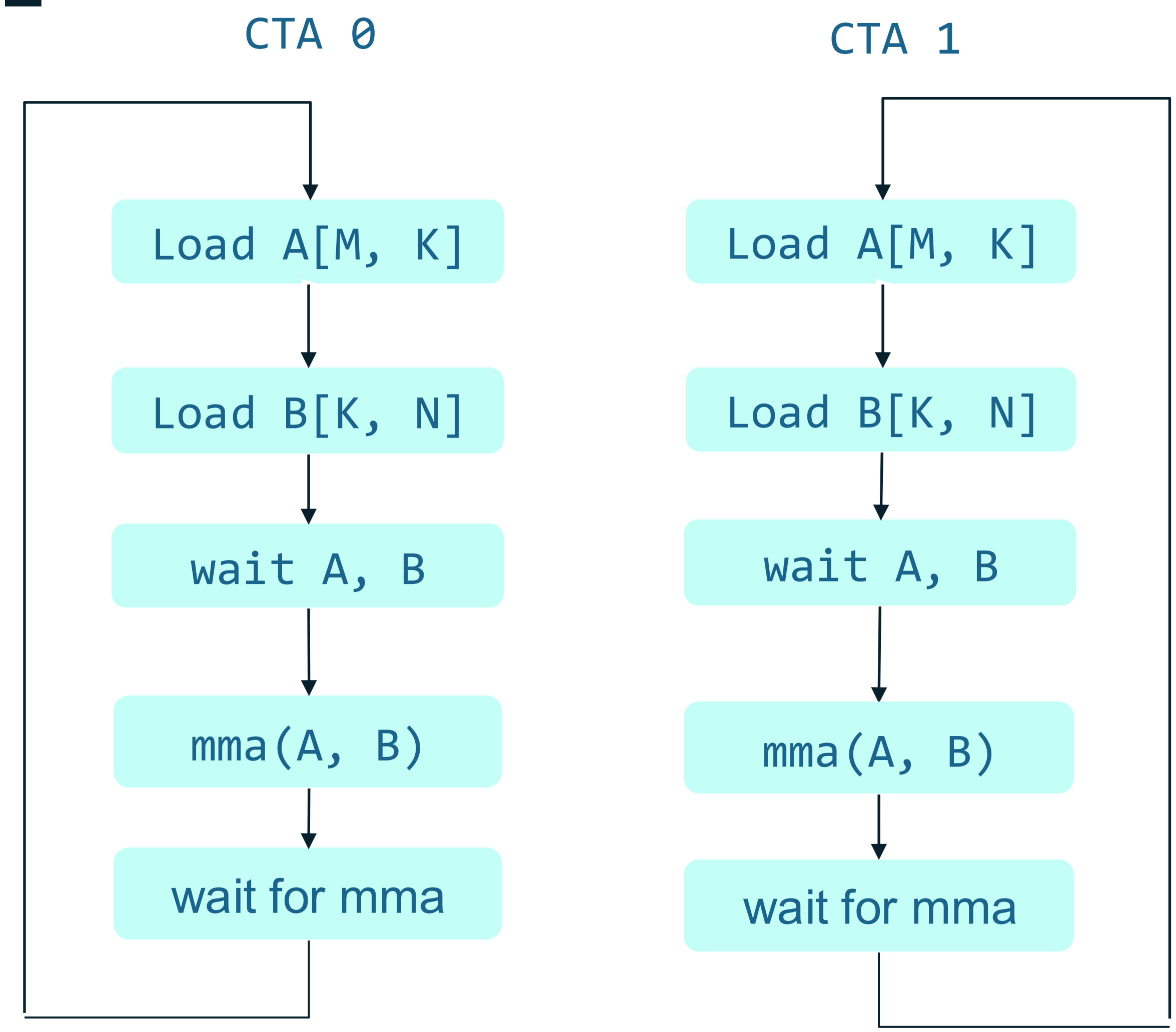
	Triton	TLX
2-Simplicial Attention on H100	**337** TFLOPS	**588** TFLOPS warp specialization computation pipelining ping-pong K1/V1 buffer prefetch
Flash Attention Fwd on H100	**446** TFLOPS	**548** TFLOPS warp specialization computation pipelining ping-pong
Flash Attention Fwd on B200	**789** TFLOPS warp specialization	**930** TFLOPS warp specialization mma/load pipelining fine-grain barrier on tile slices

Where We’re Going

- More hardware features
  - Paired-CTA
  - Cluster launch control (CLC)
  - Distributed shared memory
- Supporting AMDGPU
  - Same user interface
  - Software-based wave specialization support

Paired-CTA MMA on Blackwell

- Single-CTA
  - Each CTA computes **M*N**
  - Two CTAs compute **2*M*N**

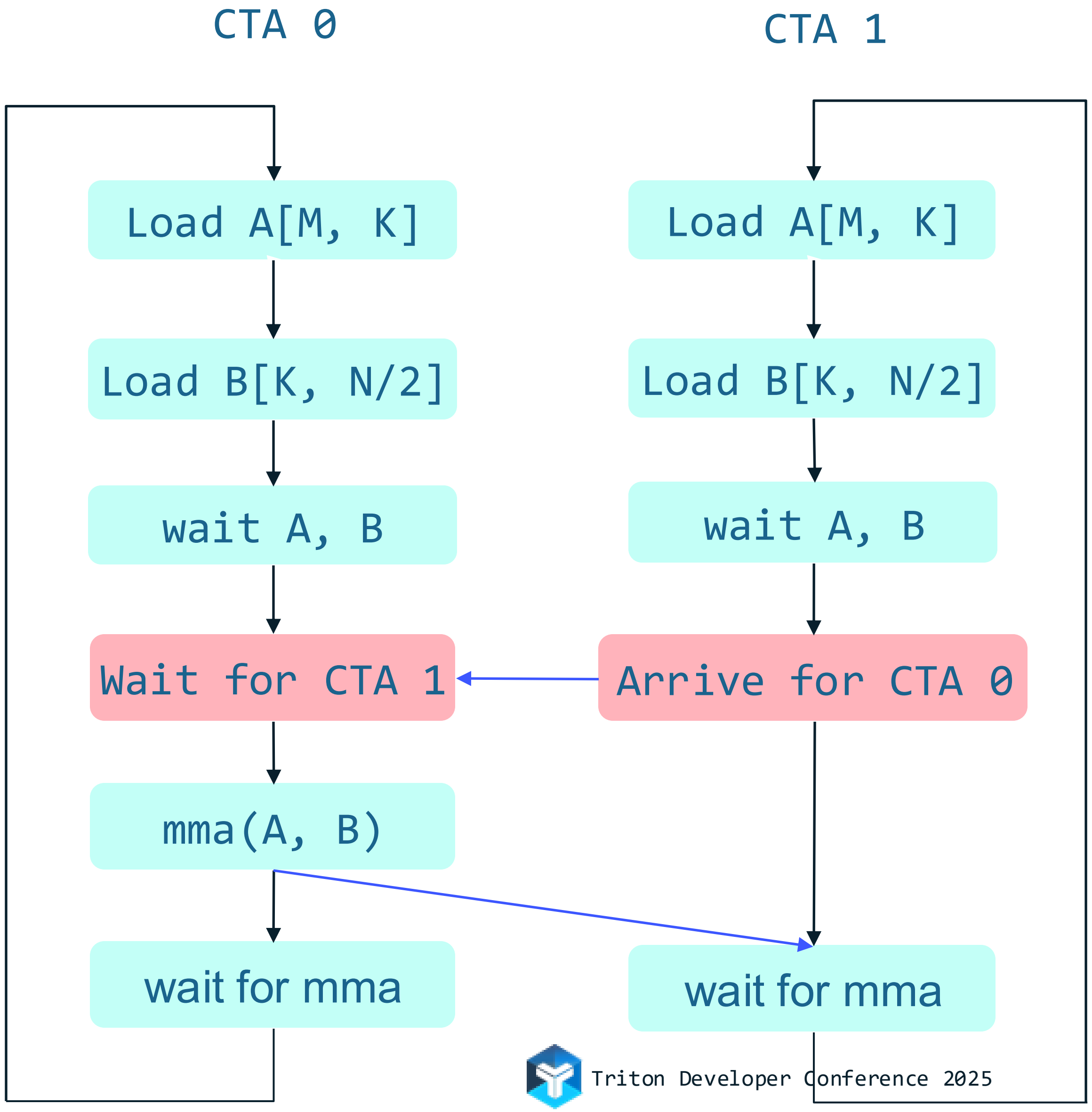


Paired-CTA MMA on Blackwell

- Paired-CTA
  - Each CTA computes **M*N**
  - Two CTAs co-compute **2*M*N**
  - **B** is shared, half stored in CTA 0, half in CTA 1
  - Only one extra barrier arrive/wait

- Less memory traffic
- Less shared memory usage

- +2% speedup over single-CTA
- 98% of cuBLAS GEMM performance


CTA 0

CTA 1

Load A[M, K]

Load B[K, N/2]

wait A, B

Wait for CTA 1

mma(A, B)

wait for mma

Load A[M, K]

Load B[K, N/2]

wait A, B

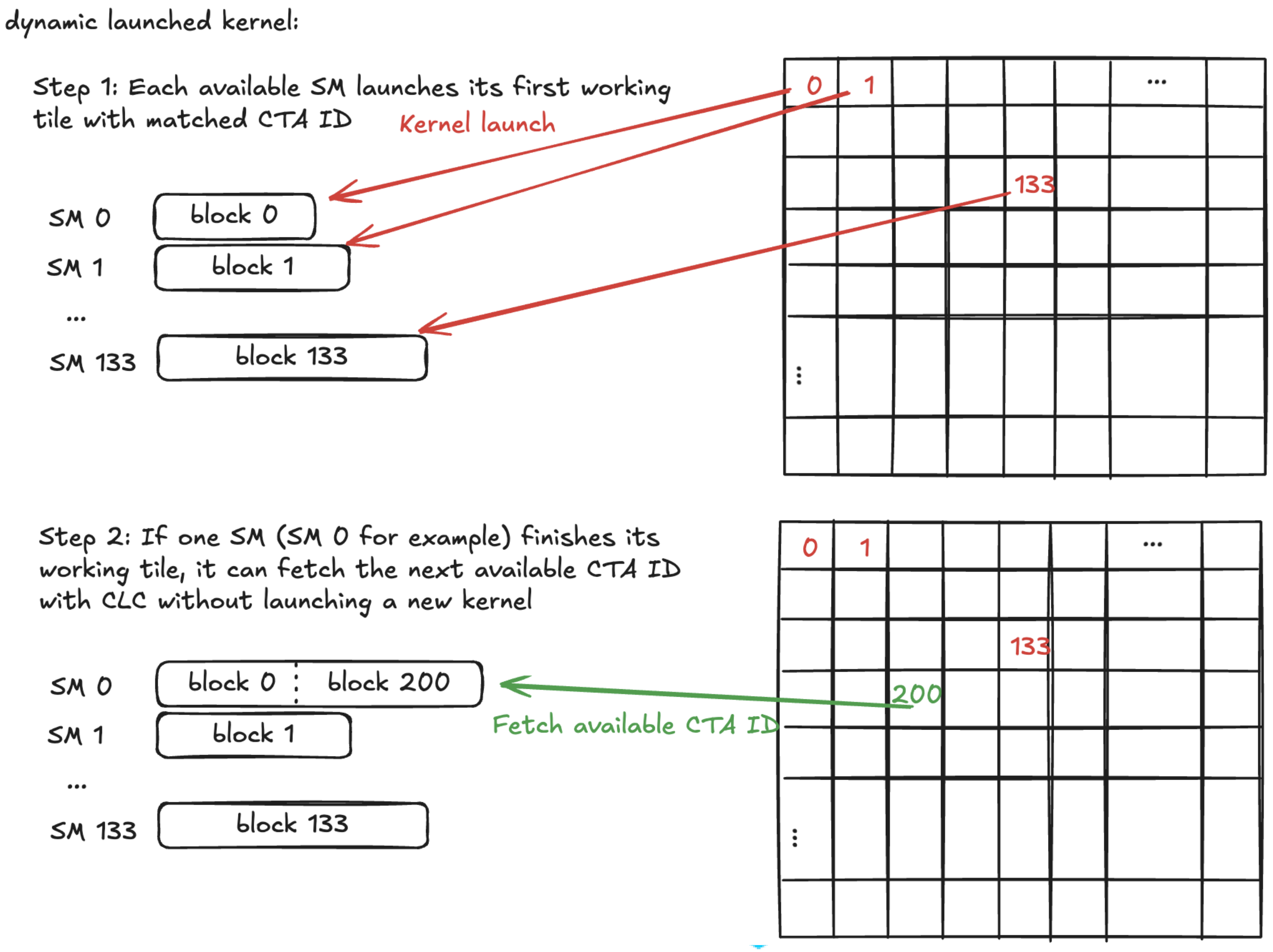
Arrive for CTA 0

wait for mma

Cluster Launch Control (CLC)

- Enables dynamic tile scheduling for imbalanced workloads

```
start_pid = tl.program_id(axis=0)
phase = 0
while start_pid != -1 :
  # compute tile offset based on start_pid
  ...
  # GEMM K-loop
  for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
    ...


  # Issue async clc.try_cancel for the next
  # available CTA
  tlx.barrier_expect_bytes(clc_mbars, 16)
  tlx.clc_issue(clc_responses, clc_mbars)
  tlx.barrier_wait(clc_mbars, phase)

    # Extract CTA ID from CLC response
  start_pid = tlx.clc_query(clc_responses)
  phase = phase ^ 1
```



dynamic launched kernel:

Step 1: Each available SM launches its first working tile with matched CTA ID

Step 2: If one SM (SM 0 for example) finishes its working tile, it can fetch the next available CTA ID with CLC without launching a new kernel

Main Takeaways

- TLX = Triton Low-level Language Extension
  - Hardware-level control within the Triton ecosystem
- Available now!
  - Download : https://github.com/facebookexperimental/triton or QR code below.
  - Spec: https://github.com/facebookexperimental/triton/blob/main/README.md
  - Barrier manual: https://github.com/facebookexperimental/triton/blob/main/third_party/tlx/tlx_barriers.md
- Supports NVIDIA Hopper/Blackwell
  - AMD GPU support is in development.
- Try it out and tell us what you think!

Triton Developer Conference 2025

Hongtao Yu (Meta)