TLX: Triton-Like Simplicity, a Clear Path to Peak Performance

Hongtao Yu Daohang Shi Peng Chen
Karthik Manivannan Nick Riasanovsky Manman Ren

Outline

- Triton’s Model vs. Modern GPU Reality
  - Tile-based kernels vs. warp-groups and async pipelines
- TLX: From Tiles to Schedules
  - Hardware pipelines without losing Triton simplicity
- Kernel Optimization in Practice
  - Pipelining, specialization, and memory orchestration with TLX

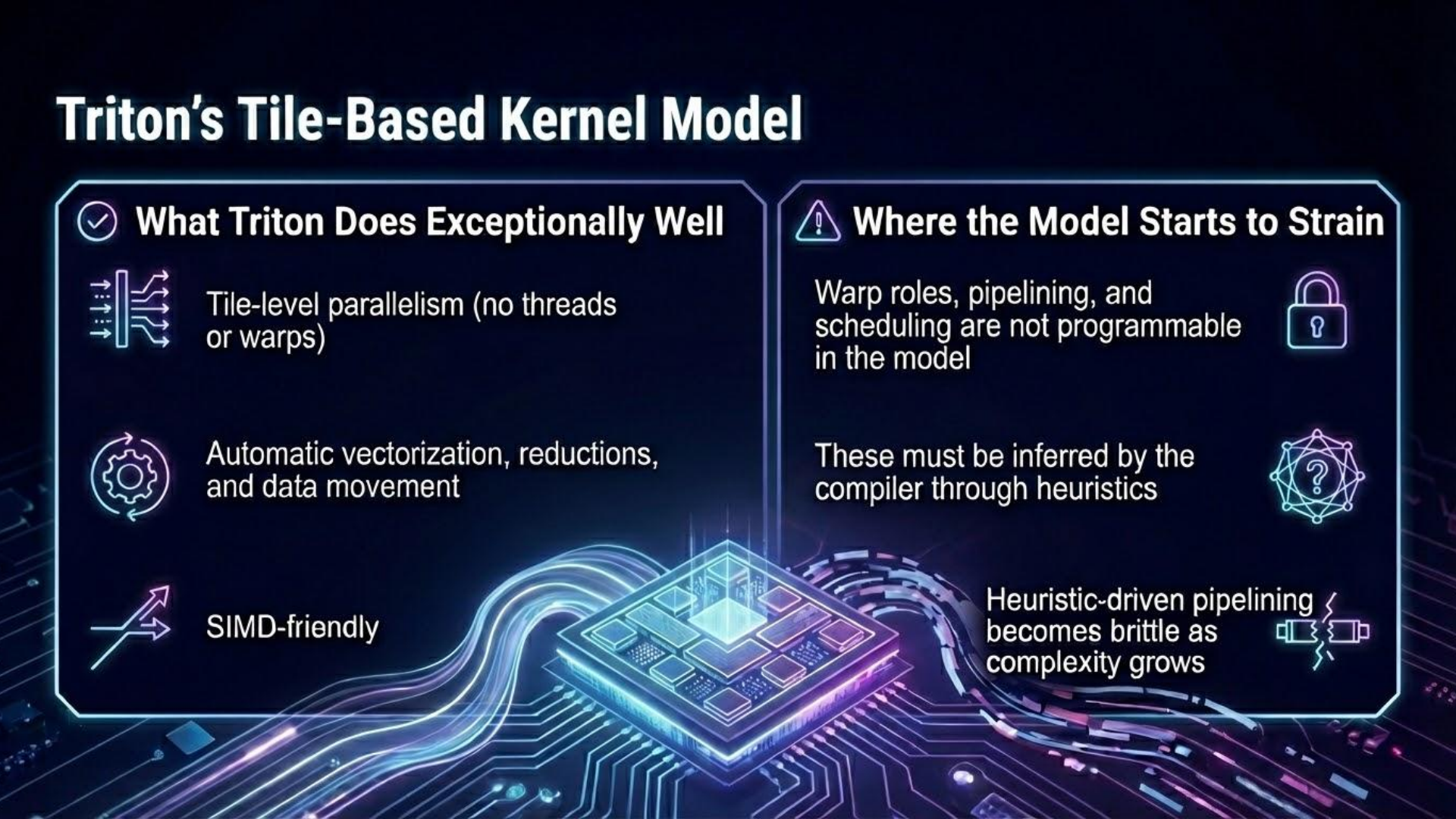
Triton's Tile-Based Kernel Model

What Triton Does Exceptionally Well

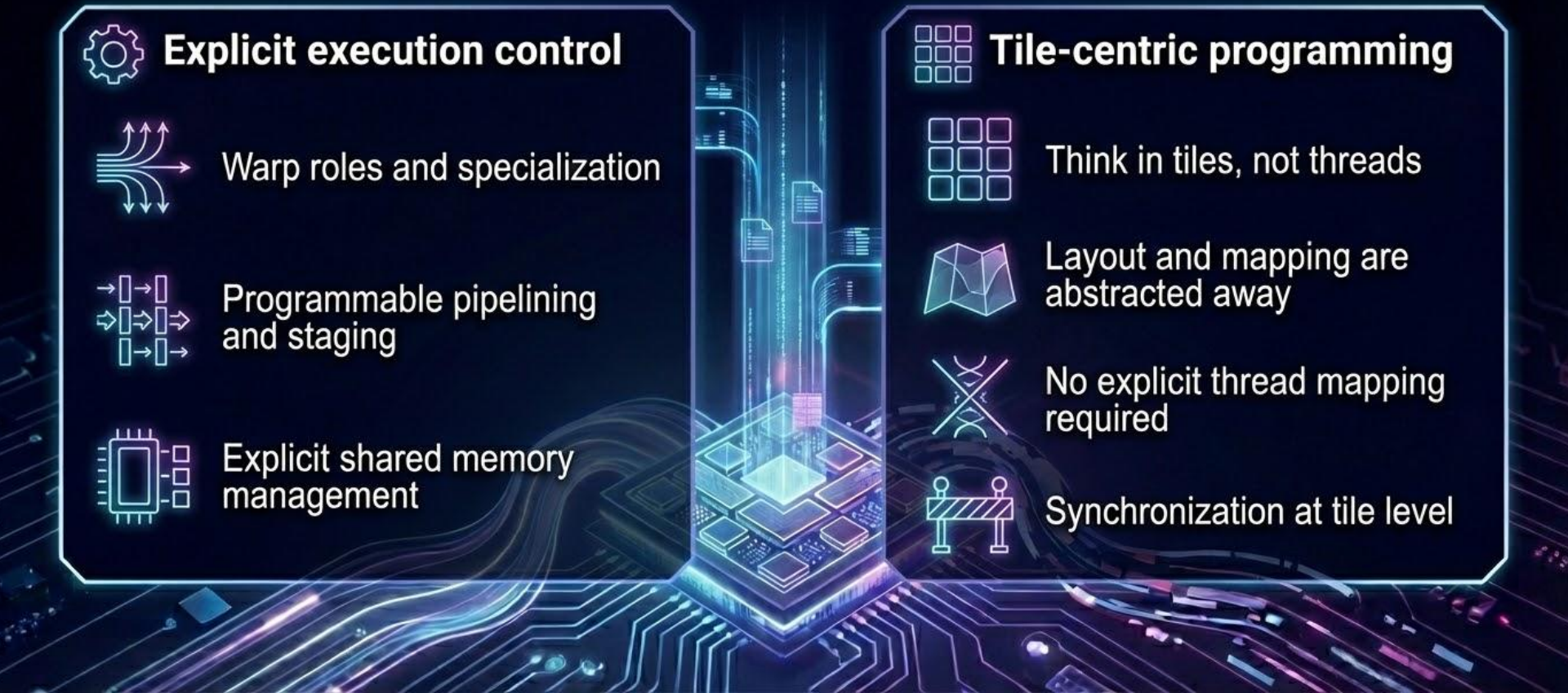
- Tile-level parallelism (no threads or warps)
- Automatic vectorization, reductions, and data movement
- SIMD-friendly

Where the Model Starts to Strain

- Warp roles, pipelining, and scheduling are not programmable in the model
- These must be inferred by the compiler through heuristics
- Heuristic-driven pipelining becomes brittle as complexity grows



TLX: Triton Low-level Language Extension



Explicit execution control

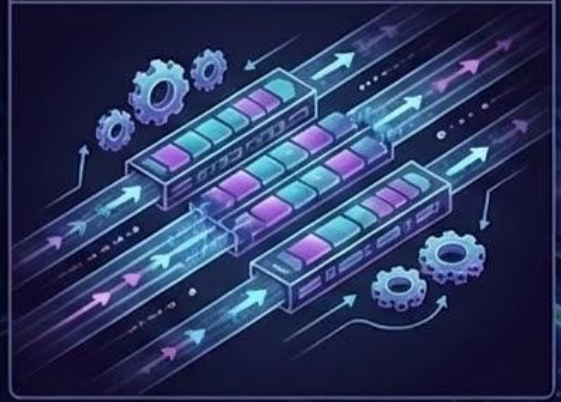
- Warp roles and specialization
- Programmable pipelining and staging
- Explicit shared memory management

Tile-centric programming

- Think in tiles, not threads
- Layout and mapping are abstracted away
- No explicit thread mapping required
- Synchronization at tile level

Kernel Optimization with TLX

Improving tile-level parallelism



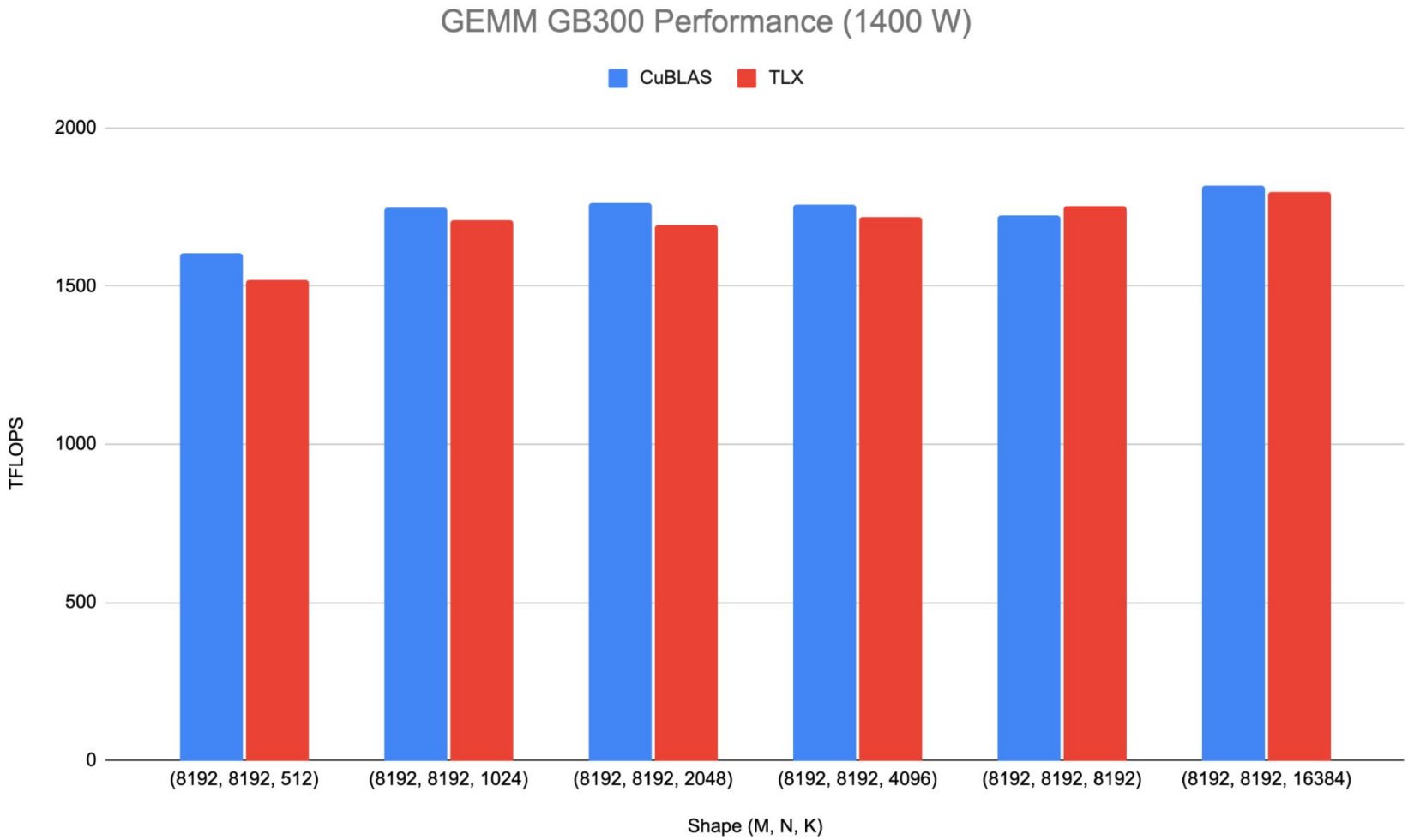
- Pipelining
- Warp specialization
- Cooperative Data partition
- Pingpong schedule

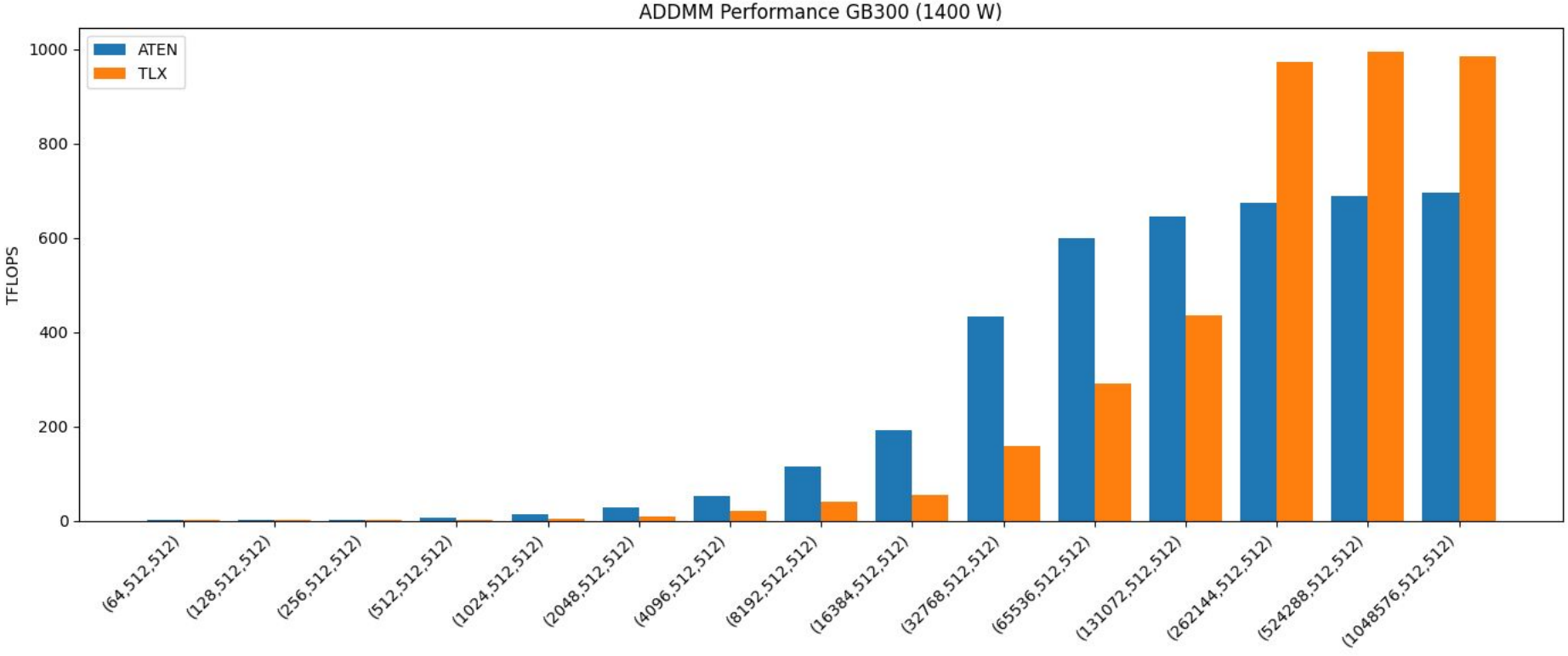
Managing shared Memory

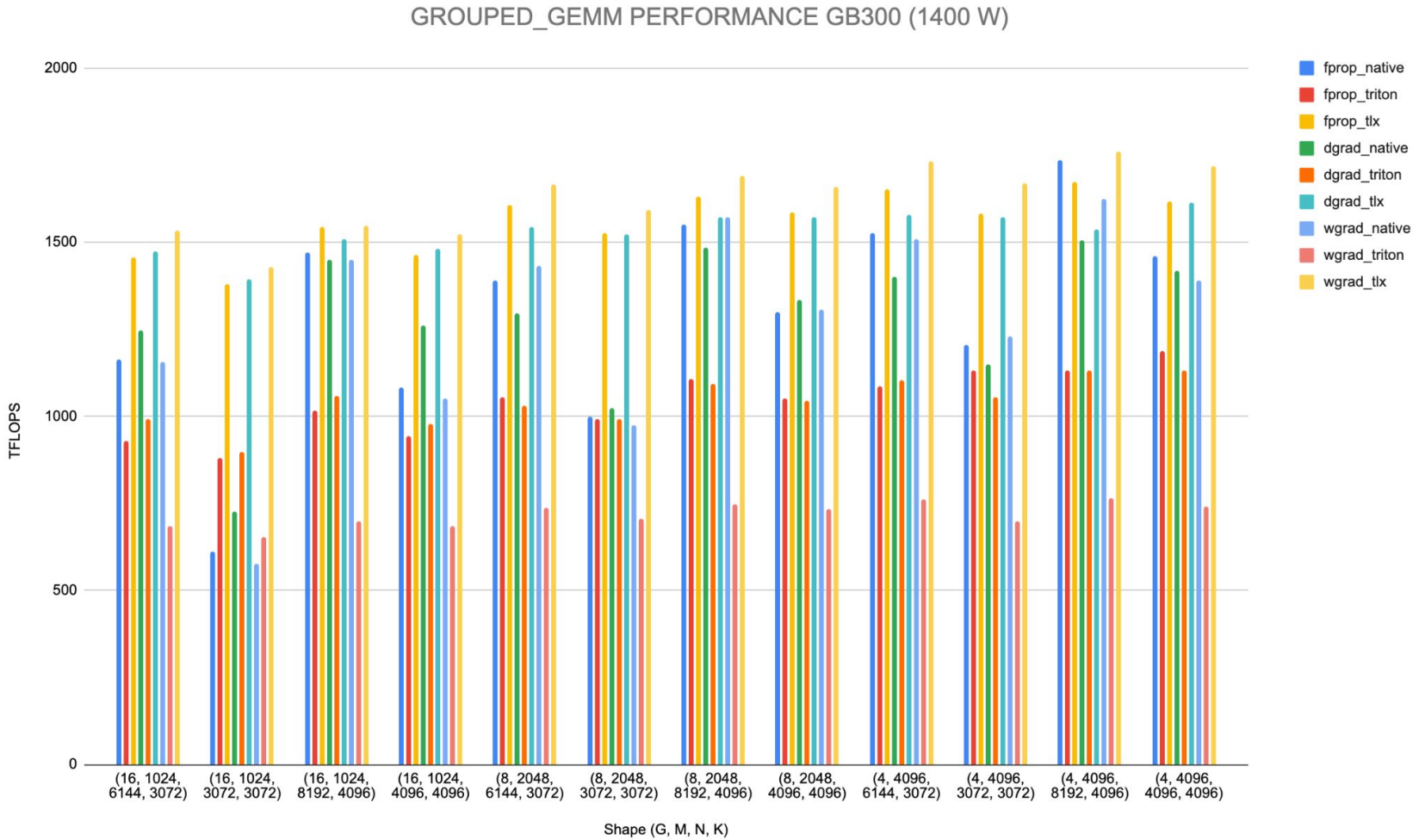
- Reducing register pressure
- Increasing tile size to improve tensor core utilization
- Advancing with distributed shared memory

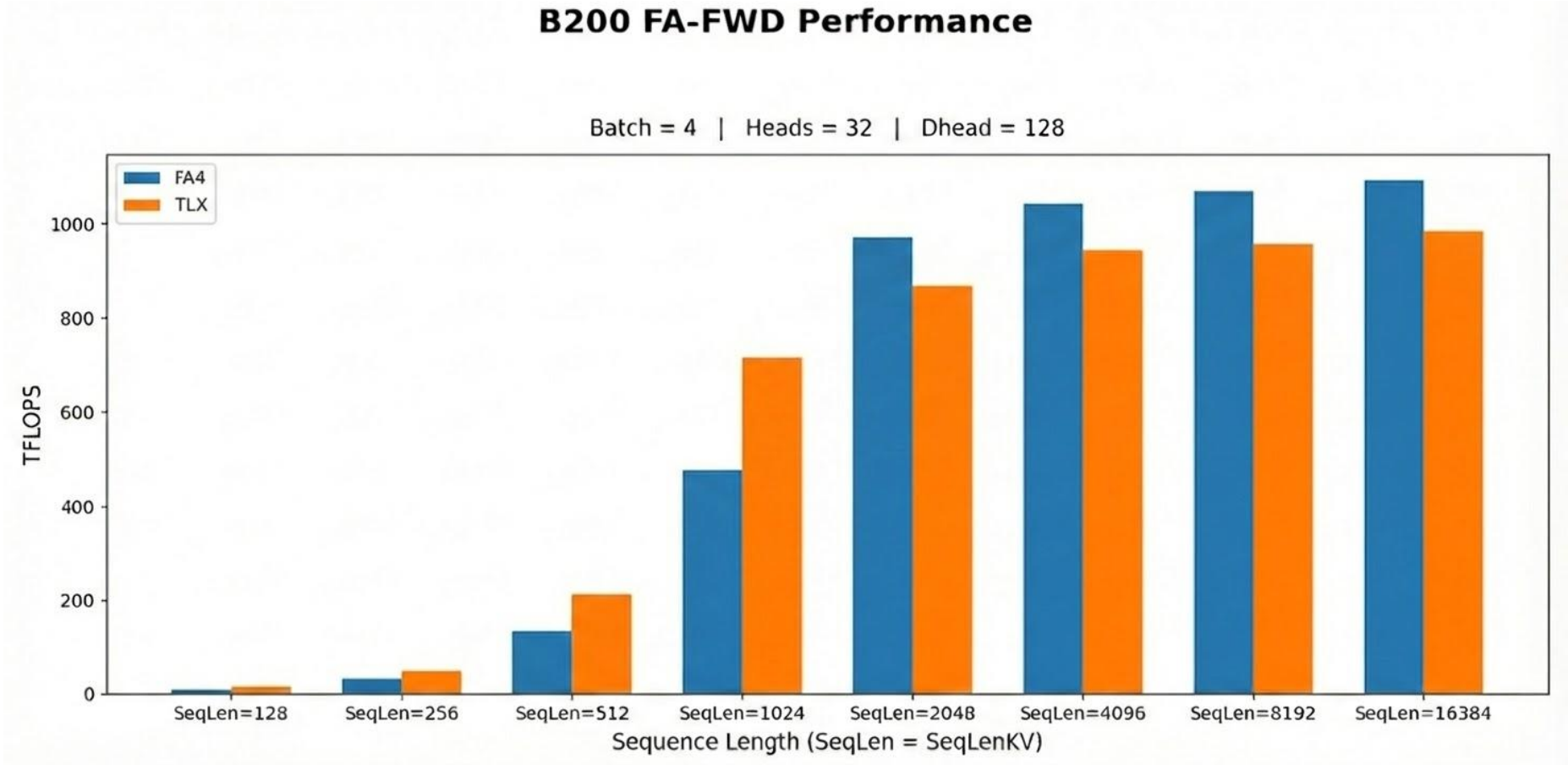
Improving load balancing

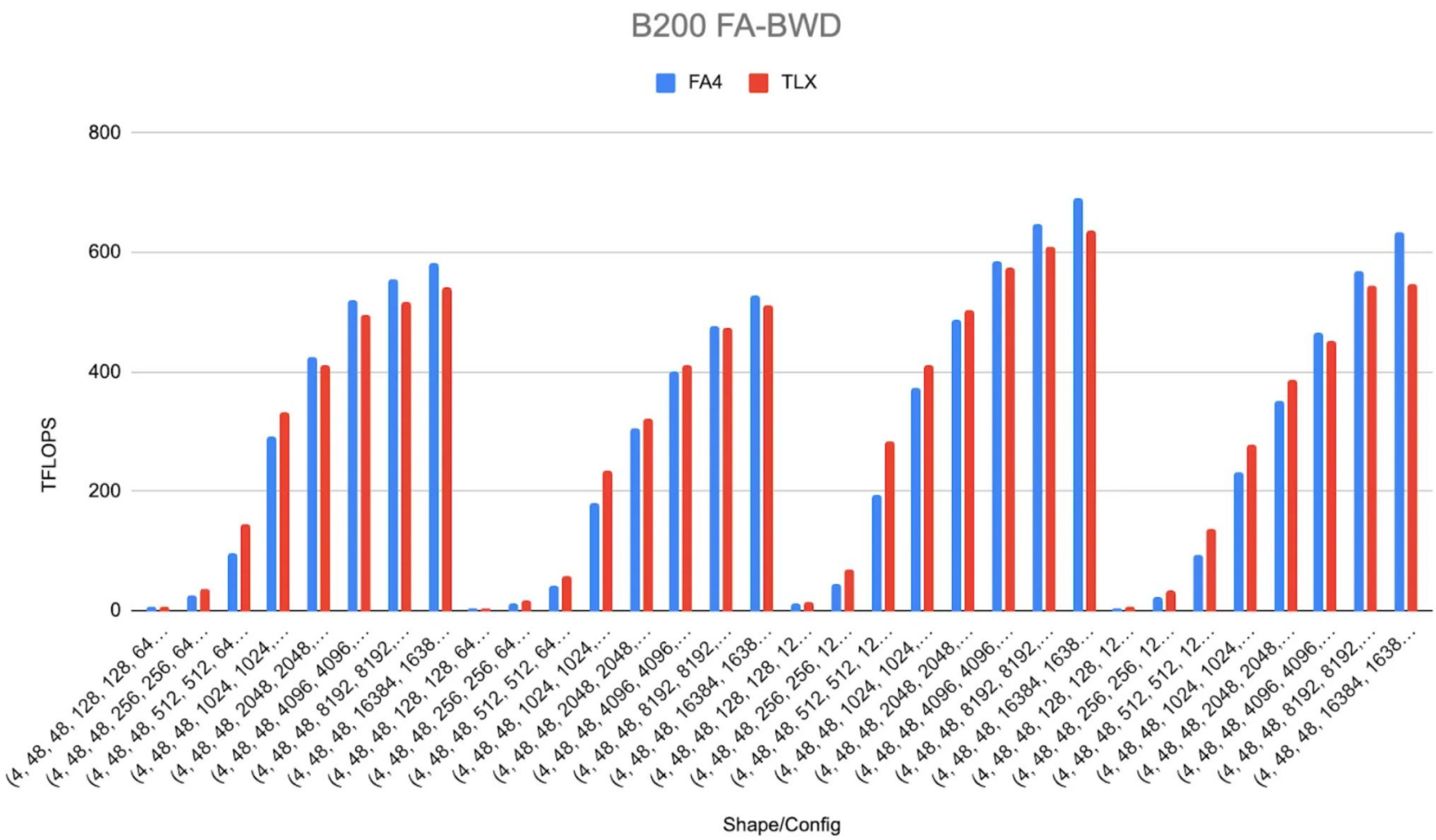
- Dynamic Launch control

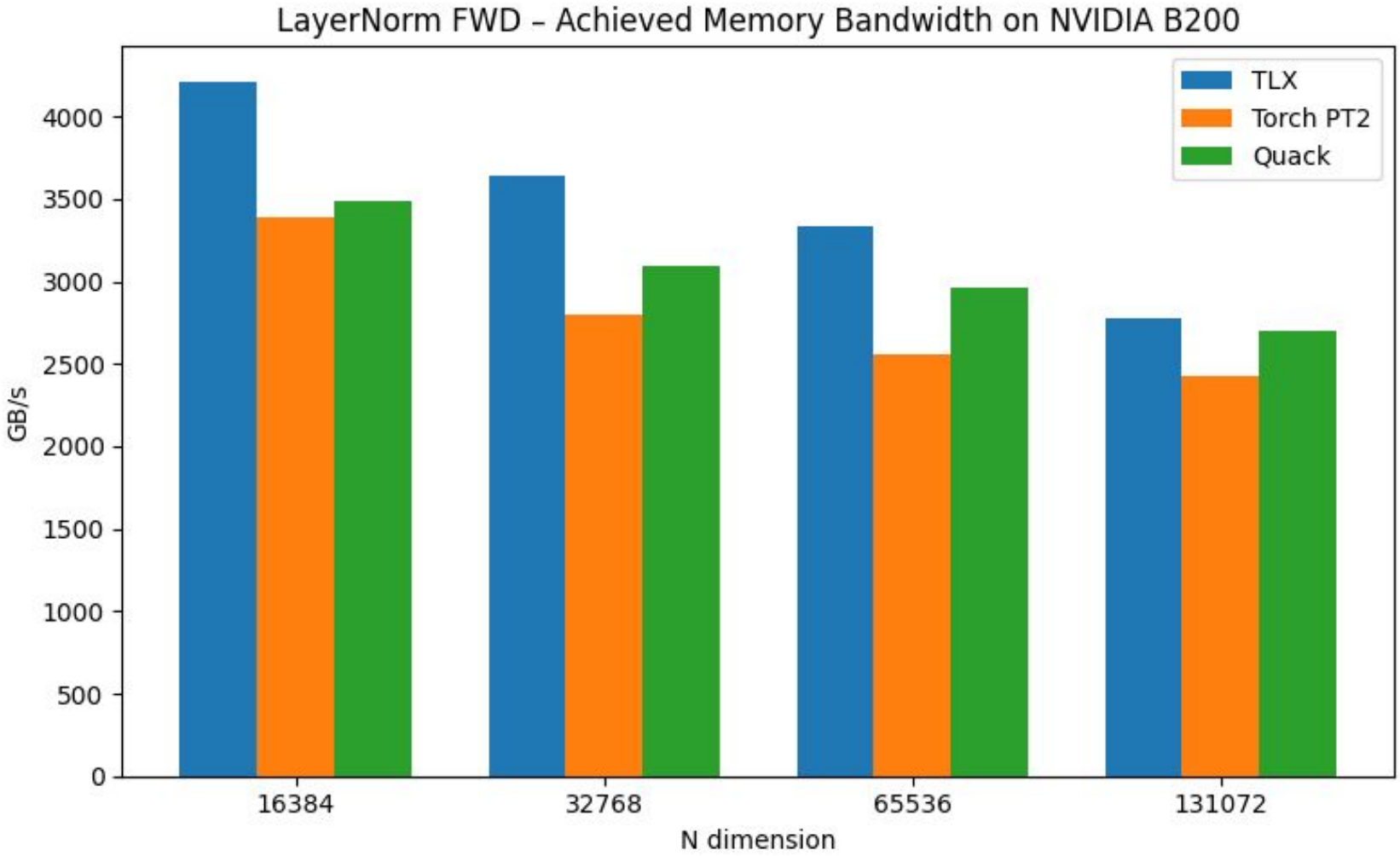

GEMM GB300 Performance (1400 W)
CuBLAS
TLX
2000
1500
1000
500
0
TFLOPS
(8192, 8192, 512)
(8192, 8192, 1024)
(8192, 8192, 2048)
(8192, 8192, 4096)
(8192, 8192, 8192)
(8192, 8192, 16384)
Shape (M, N, K)

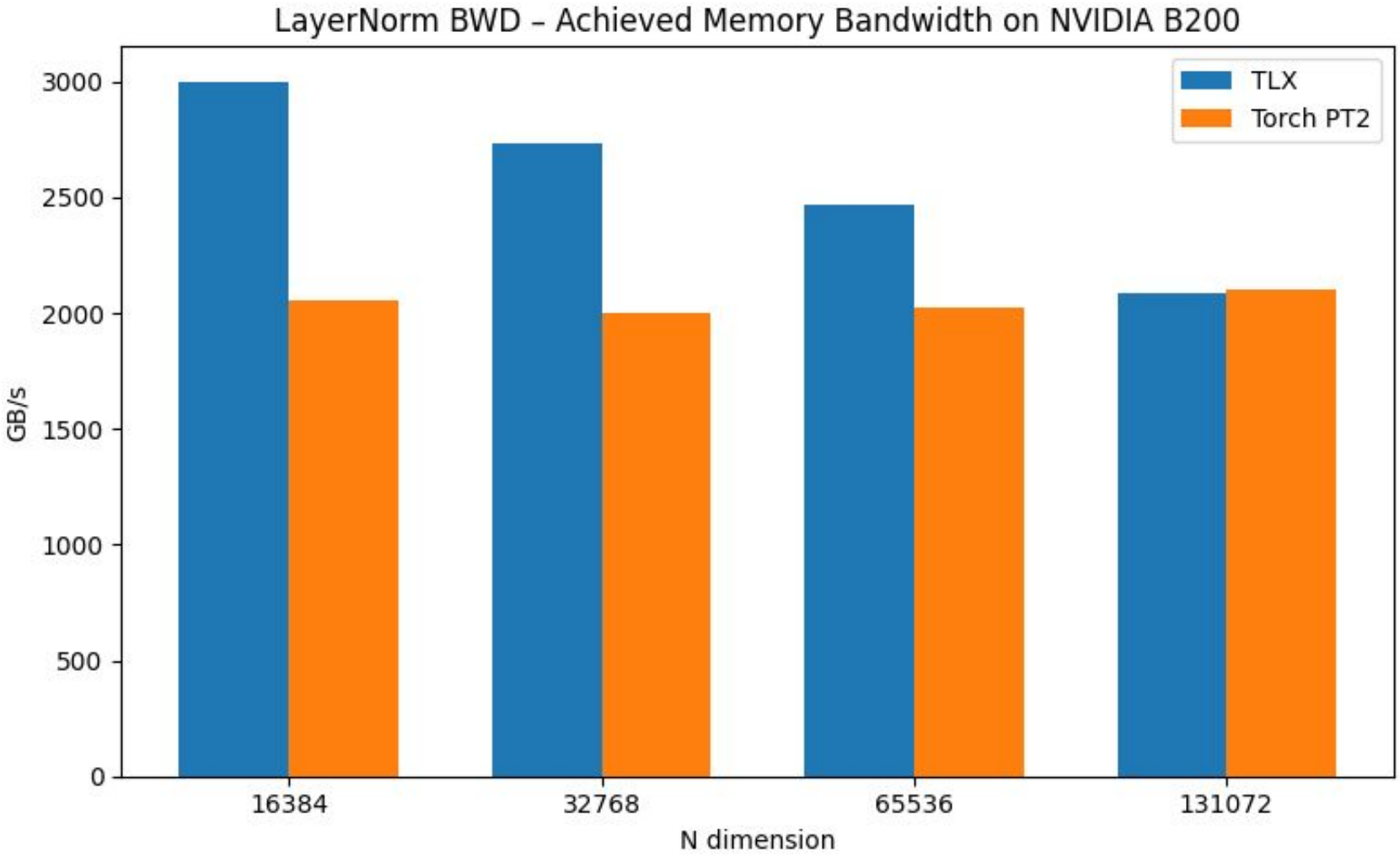

ADDMM Performance GB300 (1400 W)
ATEN
TLX
TFLOPS
0
200
400
600
800
1000
(64,512,512)
(128,512,512)
(256,512,512)
(512,512,512)
(1024,512,512)
(2048,512,512)
(4096,512,512)
(8192,512,512)
(16384,512,512)
(32768,512,512)
(65536,512,512)
(131072,512,512)
(262144,512,512)
(524288,512,512)
(1048576,512,512)


GROUPED_GEMM PERFORMANCE GB300 (1400 W)
TFLOPS
2000
1500
1000
500
0
fprop_native
fprop_triton
fprop_tlx
dgrad_native
dgrad_triton
dgrad_tlx
wgrad_native
wgrad_triton
wgrad_tlx
(16, 1024, 6144, 3072)
(16, 1024, 3072, 3072)
(16, 1024, 8192, 4096)
(16, 1024, 4096, 4096)
(8, 2048, 6144, 3072)
(8, 2048, 3072, 3072)
(8, 2048, 8192, 4096)
(8, 2048, 4096, 4096)
(4, 4096, 6144, 3072)
(4, 4096, 3072, 3072)
(4, 4096, 8192, 4096)
(4, 4096, 4096, 4096)
Shape (G, M, N, K)


B200 FA-FWD Performance
Batch = 4 | Heads = 32 | Dhead = 128
FA4
TLX
TFLOPS
0
200
400
600
800
1000
SeqLen=128
SeqLen=256
SeqLen=512
SeqLen=1024
SeqLen=2048
SeqLen=4096
SeqLen=8192
SeqLen=16384
Sequence Length (SeqLen = SeqLenKV)


B200 FA-BWD
FA4
TLX
TFLOPS
800
600
400
200
0
(4, 48, 48, 128, 128, 64…
(4, 48, 48, 256, 256, 64…
(4, 48, 48, 512, 512, 64…
(4, 48, 48, 1024, 1024,…
(4, 48, 48, 2048, 2048,…
(4, 48, 48, 4096, 4096,…
(4, 48, 48, 8192, 8192,…
(4, 48, 48, 16384, 1638…
(4, 48, 48, 128, 128, 64…
(4, 48, 48, 256, 256, 64…
(4, 48, 48, 512, 512, 64…
(4, 48, 48, 1024, 1024,…
(4, 48, 48, 2048, 2048,…
(4, 48, 48, 4096, 4096,…
(4, 48, 48, 8192, 8192,…
(4, 48, 48, 16384, 1638…
(4, 48, 48, 128, 128, 12…
(4, 48, 48, 256, 256, 12…
(4, 48, 48, 512, 512, 12…
(4, 48, 48, 1024, 1024,…
(4, 48, 48, 2048, 2048,…
(4, 48, 48, 4096, 4096,…
(4, 48, 48, 8192, 8192,…
(4, 48, 48, 16384, 1638…
(4, 48, 48, 128, 128, 12…
(4, 48, 48, 256, 256, 12…
(4, 48, 48, 512, 512, 12…
(4, 48, 48, 1024, 1024,…
(4, 48, 48, 2048, 2048,…
(4, 48, 48, 4096, 4096,…
(4, 48, 48, 8192, 8192,…
(4, 48, 48, 16384, 1638…
Shape/Config


LayerNorm FWD – Achieved Memory Bandwidth on NVIDIA B200
TLX
Torch PT2
Quack
GB/s
0
500
1000
1500
2000
2500
3000
3500
4000
16384
32768
65536
131072
N dimension


LayerNorm BWD – Achieved Memory Bandwidth on NVIDIA B200
TLX
Torch PT2
GB/s
3000
2500
2000
1500
1000
500
0
16384
32768
65536
131072
N dimension

Kernel Optimization in Practice

- Layer Norm on Hopper/Blackwell
  - Using shared memory to reduce register pressure
  - Multi-CTA with distributed shared memory
  - Asyncrhonous global memory load
- GEMM on Blackwell
  - Warp specialization
  - Persistent with CLC
  - Paired-CTA MMA
  - Cooperative data partitioning
  - Producer pipelining
  - Epilogue subtiling

Optimizing Layer Norm FWD for Hopper/Blackwell

- torch.nn.functional.layer_norm

```
mean    = avg(x)
var     = avg((x - mean) * (x - mean))
inv_std = rsqrt(var + epsilon)
y = (x - mean) * inv_std * weight + bias
```

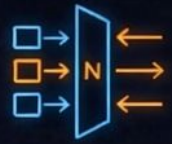
```
@triton.jit
layer_norm_fwd_fused_kernel(X,Y,W,B,Mean, Rstd, stride, N, eps.BLOCK_SIZE::tl.constexpr):
```

Step 1: Compute Mean

```
row_idx = tl.program_id(0)
row_start_ptr = X + row_idx * stride
mean = 0.0
for offset in range(0, N, BLOCK_SIZE)
    cols = offset + tl.arange(0, BLOCK_SIZE)
    x = tl.load(row_start_ptr + cols)
    mean += tl.sum(x, axis=0)
mean = mean / N
```

Step 2: Compute Variance & Rstd

```
var = 0.0
for offset in range(0, N, BLOCK_SIZE):
    cols = offset + tl.arange(0, BLOCK_SIZE)
    x = tl.load(row_start_ptr + cols)
    x = x - mean
    var += tl.sum(x * x, axis=0)
var = var / N
rstd = 1.0 / tl.sqrt(var + eps)
```

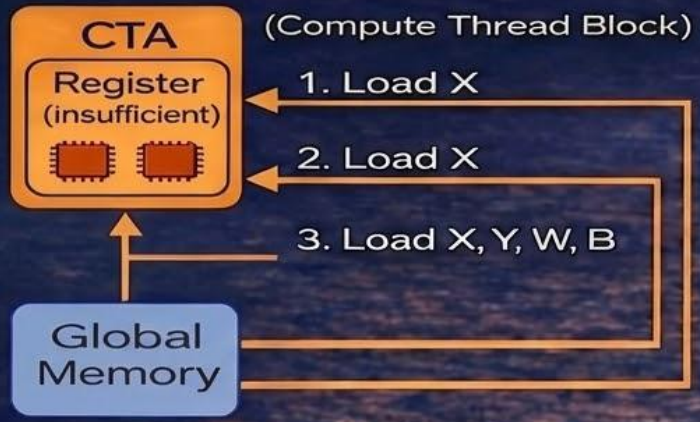


Step 3: Normalize & Apply Affine Transform

```
out_ptr = Y + row_idx * stride
for offset in range(0, N, BLOCK_SIZE):
    cols = offset + tl.arange(0, BLOCK_SIZE)
    x = tl.load(W + cols)
    w = tl.load(W + cols)
    b = tl.load(B + cols)
    x_norm = (x - mean) * rstd
    y = x_norm * w + b
    tl.store(out_ptr + cols, y)
```

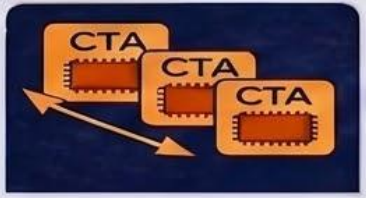
Triton Kernel Current Challenges & Optimization Path

Current Problem: repeated Data Load



Three repeated loads; single CTA registers are limited, data cannot be reused across steps, leading to high memory bandwidth consumption.

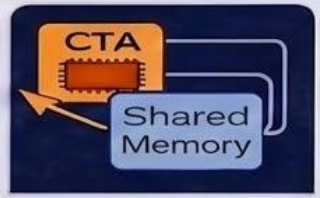
Optimization: Improving Reuse & Parallelism



Optimization 1: Multi-CTA Parallelism

Decompose tasks, utilize multiple CTAs to reduce register pressure.

- Further reduces register pressure



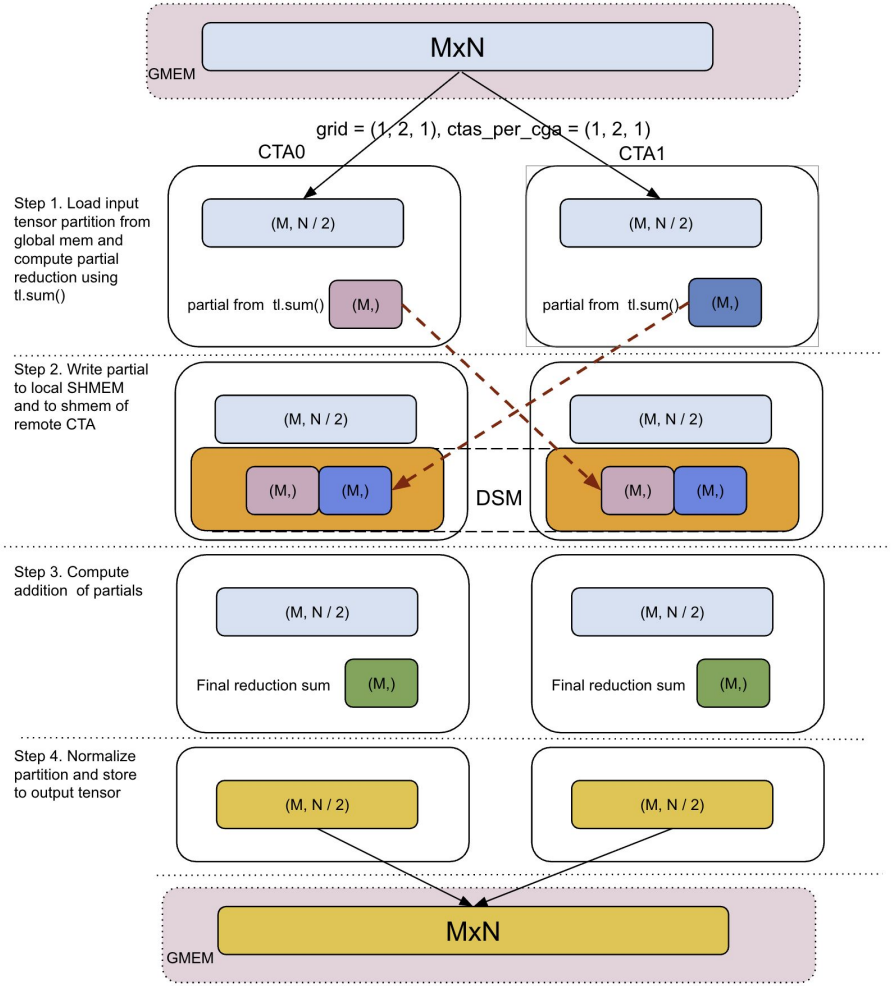
Optimization 2: Local Shared Memory Caching

Use Local Shared Memory within same CTA to temporarily store intermediate data (e.g. Mean, Rstd)

Optimization 3: Copy Async (Asynchronous Copy)

Use asynchronous copy instructions to prefetch data while computing, hiding memory latency, increasing throughput

All-Reduce with Multi-CTA + DSM



Reduction with Multi-CTA + DSM

```
grid = lambda META: (triton.cdiv(M, META['BLOCK_M']), NUM_CTAS, 1)
compute_multi_cta_sum(grid){..., ctas_per_cga=(1, NUM_CTAS, 1)}
```

Step 1: Compute local partial sum

```
# Kernel: Reduce sum (BLOCK_M,
BLOCK_N) tensor x to (BLOCK_M,)

@triton.jit
def compute_multi_cta_sum(x):
  x = tl.load(row_start_ptr + cols)
  local_buff =
    tlx.local_alloc((BLOCK_M, 1),
    tlx.dtype_of(x), num=NUM_CTAS)
  local_sum = tl.sum(x, axis=1)
  cta_rank = tlx.cluster_cta_rank()
  tlx.local_store(local_buff[cta_rank],
    local_sum)
```

Step 2. Remote store local_partial to all other CTAs in the cluster

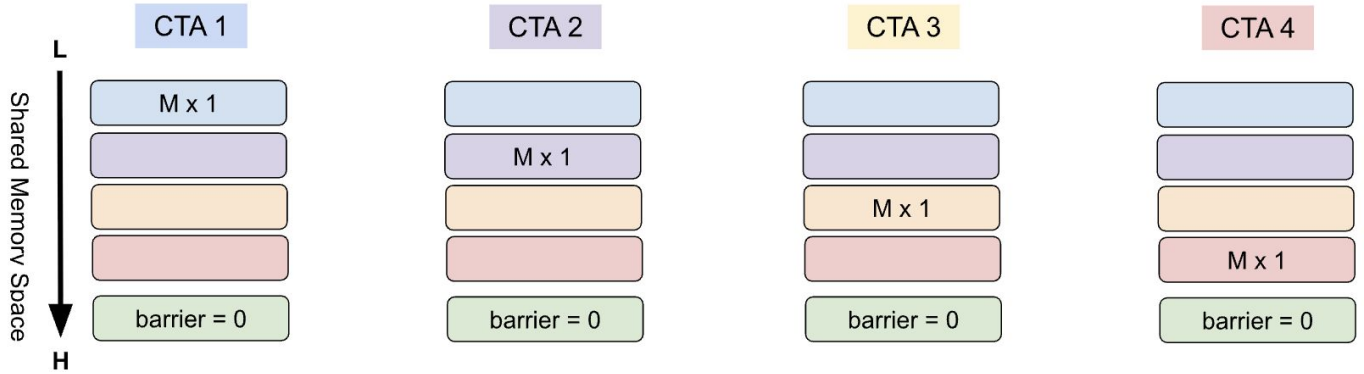
```
barrier = tlx.alloc_barriers(num_bar-
 riers=1)
store_bytes = BLOCK_SIZE_M *
  tlx.size_of(x) * (NUM_CTAS - 1))
tlx.barrier_expect_bytes(barrier[0],
  size=store_bytes)
tlx.cluster_barrier()
for i in tl.static_range(NUM_CTAS):
    if cta_rank != i:
        tlx.async_remote_shmem_store(
            dst=local_buff[cta_rank],
            src=local_sum,
            remote_cta_rank=i,
            barrier=barrier)
```

Step 3. All remote CTAs have written partials, calculate final sum

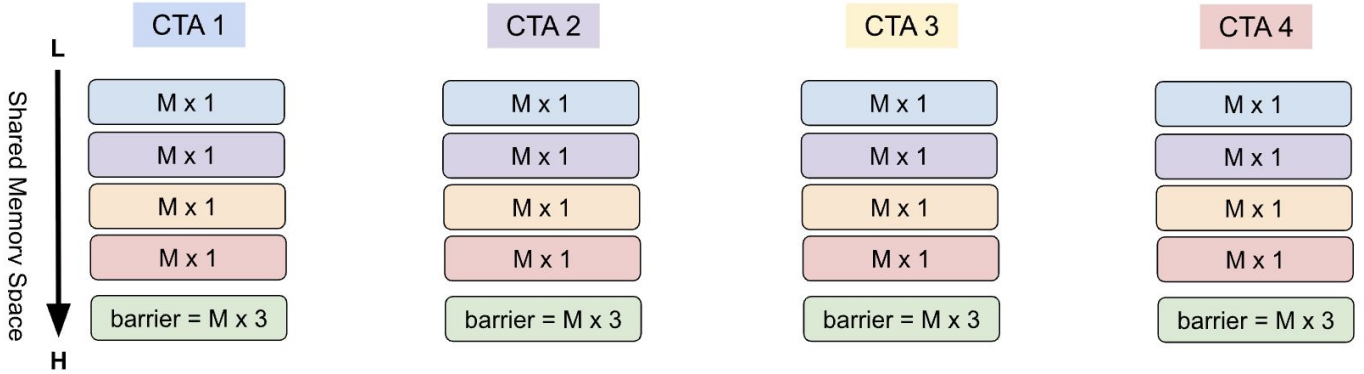
```
tlx.barrier_wait(barrier, phase=
phase)

final_sum = tl.zeros((BLOCK_SIZE_M,
  1), dtype=tlx.dtype_of(x))
for i in tl.static_range(NUM_CTAS):
    final_sum += tlx.local_load(
    local_buff[i])

return final_sum
```

Asynchronous Remote Shared Memory Store

Before



After



TLX Layer Norm

```
@triton.jit                                X, Y, W, B, Mean, Rstd, stride, N, eps,
def layer_norm_fwd_fused_kernel(    BLOCK_SIZE: tl.constexpr, num_reduction_ctas: tl.constexpr, ):
```

Step 1: Cross-CTA reduction for mean

```
# Load data
x = tl.load(row_start_ptr + cols)

# Step 1: Cross-CTA reduction for
mean
mean = compute_multi_cta_sum(x) /
  N
```

Step 2: Cross-CTA reduction for variance

```
# Step 2: Cross-CTA reduction for
variance
x_centered = x - mean
var = compute_multi_cta_sum(x_cen-
  tered * x_centered) / N

var = total_var
rstd = 1.0 / tl.sqrt(var + eps)
```

Step 3: Normalize and apply affine transform

```
# Step 3: Normalize and apply
affine transform
w = tl.load(W + col_start + cols)
b = tl.load(B + col_start + cols)

x_norm = (x - mean) * rstd
y = x_norm * w + b
y = x_norm * w + b

tl.store(out_ptr + cols, y)
```

Leverage Shared Memory to Reduce Register Pressure

```python
@triton.jit
def layer_norm_fwd_fused_kernel(
   X, Y, W, B,...
):
   # Load data
   x = tl.load(row_start_ptr + cols)
   x_buffer = tlx.local_alloc((BLOCK_SIZE_M, BLOCK_SIZE_N),
tl.dtype_of(X), 1)
   tlx.local_store(x_buffer[0], x)

   # Step 1: Cross-CTA reduction for mean
   mean = compute_multi_cta_sum(x) / N
```

Initial Load & Mean

Data loaded once, re-used here

```python
   # Step 2: Cross-CTA reduction for variance
   # x_centered and x share same registers
   x_centered = x - mean
   var = compute_multi_cta_sum(x_centered *
x_centered) / N
   rstd = 1.0 / tl.sqrt(var + eps)

   # Step 3: Normalize and apply affine transform
   w = tl.load(W + col_start + cols)
   b = tl.load(B + col_start + cols)
   x = tlx.local_load(x_buffer[0])
   x_norm = (x - mean) * rstd
   y = x_norm * w + b
   tl.store(out_ptr + cols, y)
```

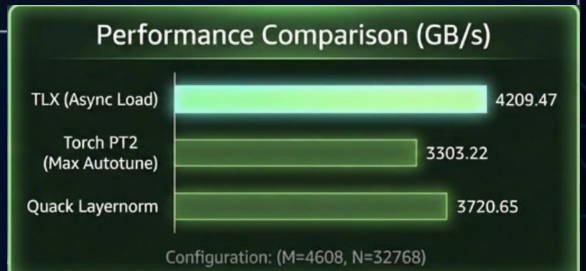
Normalize, Transform & Store

(M, N)	tlx_layernorm-gbps	torch_layernorm_pt2_max_autotune-gbps	quack_layernorm-gbps
(4608, 32768)	4013.9	3259.27	3689.2

Use Async Global Memory Copy

```
@triton.jit
def layer_norm_fwd_fused_kernel(
   X, Y, W, B, Mean, Rstd,
   stride, N, eps,
   BLOCK_SIZE: tl.constexpr,
   num_reduction_ctas: tl.constexpr,
);
   # Load data
   x_buffer = tlx.local_alloc((BLOCK_SIZE_ M, BLOCK_SIZE_N),
X.dtype.element_ty, 1)
   tlx.async_load(x_ptrs, x_buffer[0])
   tlx.async_load_commit_group()
   tlx.async_load_wait_group(0)

   # Step 1: Cross-CTA reduction for mean
   x = tlx.local_load(x_buffer[0])
   mean = compute_multi_cta_sum(x) / N
```

```
# Step 2: Cross-CTA reduction for variance
x_centered = x - mean
var = compute_multi_cta_sum(x_centered * x_centered) / N
rstd = 1.0 / tl.sqrt(var + eps)

# Step 3: Normalize and apply affine transform
w = tl.load(W + col_start + cols)
b = tl.load(B + col_start + cols)
x = tlx.local_load(x_buffer[0])
x_norm = (x - mean) * rstd
y = x_norm * w + b
tl.store(out_ptr + cols, y)
```



Optimizing GEMM on Blackwell

- Triton GEMM kernel
  - L2 swizzling
  - Host-side TMA
  - Compiler optimizations (Loop pipelining or warp specialization)
- TLX GEMM kernel
  - Warp specialization
  - Persistent with CLC
  - Paired-CTA MMA
  - Cooperative data partitioning
  - Producer pipelining

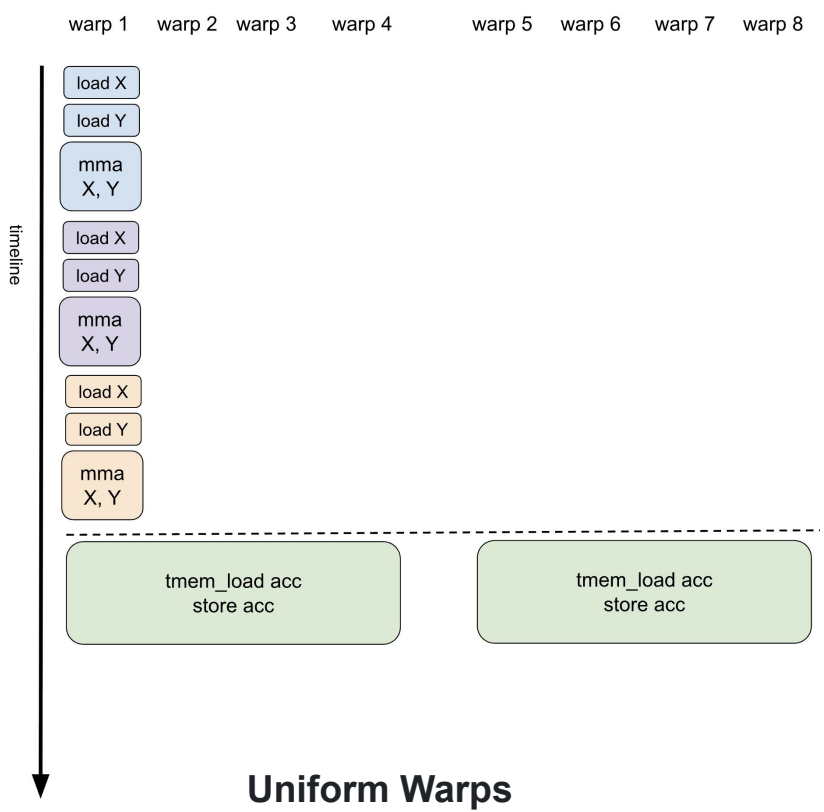
(M, N, K)	aten_matmul-tflops	tlx_matmul-tflops	triton_tma_persistent_matmul-tflops
(8191, 8192, 8192)	1143.46	1191.39	1037.42

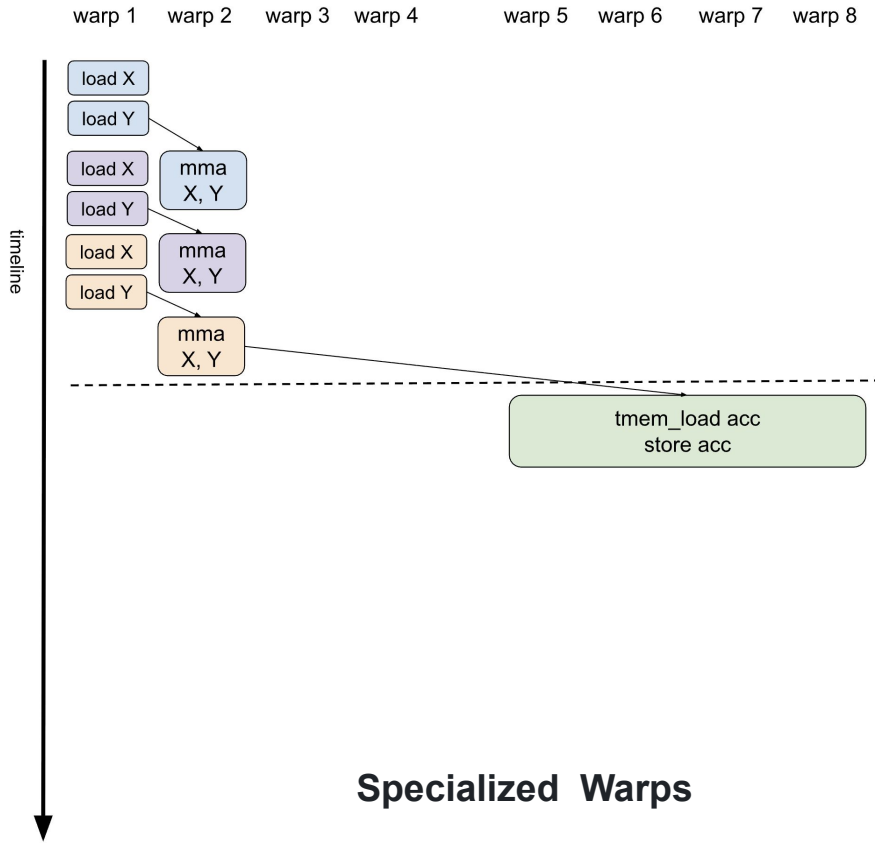
Basic GEMM Kernel

```
@triton.jit
def matmul_kernel_tma(
    a_desc, b_desc, c_desc, M, N, K,
    BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
    GROUP_SIZE_M: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_M)
    num_pid_n = tl.cdiv(N, BLOCK_N)

    # Apply swizzle for better L2 cache behavior
    pid_m, pid_n = tl.swizzle2d(pid_m, pid_n, num_pid_m, num_pid_n, GROUP_SIZE_M)

    offs_am = pid_m * BLOCK_M
    offs_bn = pid_n * BLOCK_N
    acc = tl.zeros((BLOCK_M, BLOCK_N), tl.float32)
    for k in range(tl.cdiv(K, BLOCK_K)):
        offs_k = k * BLOCK_K
        a = a_desc.load([offs_am, offs_k])
        b = b_desc.load([offs_bn, offs_k])
        acc = tl.dot(a, b, acc)
    c = acc.to(tl.float16)
    c_desc.store([offs_am, offs_bn], c)
```

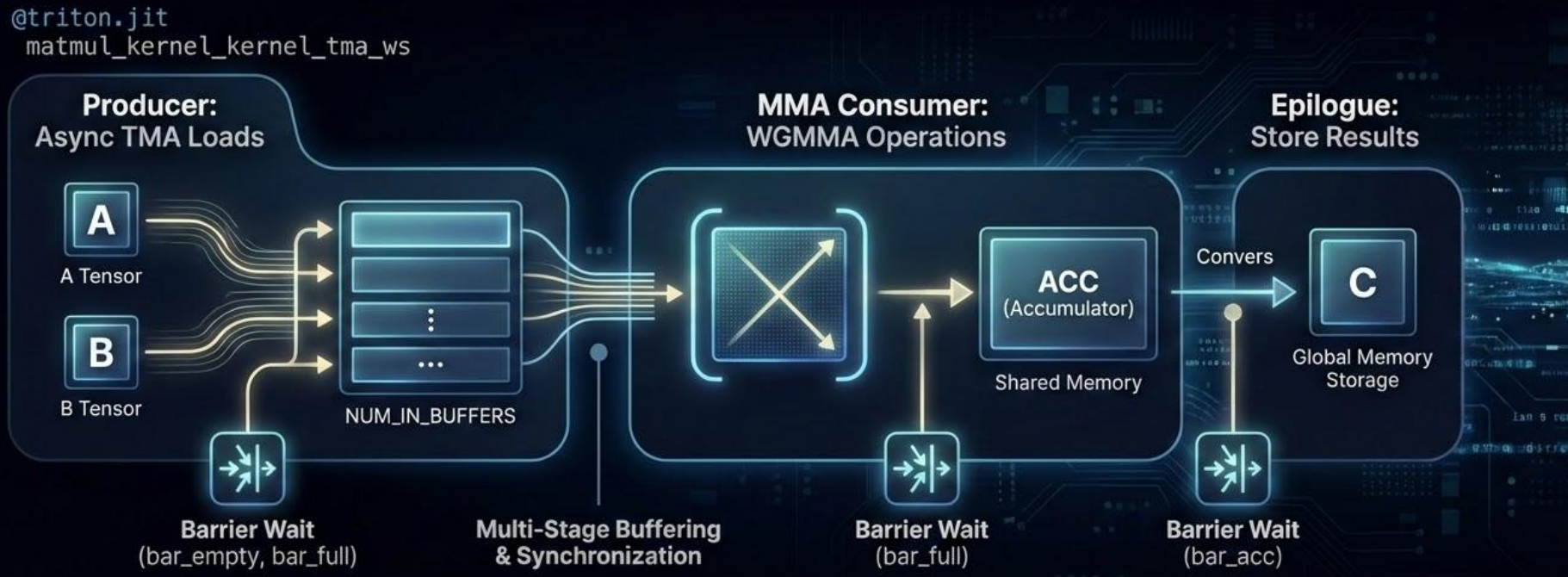
(M, N, K)	aten_matmul-tflops	tlx_matmul-tflops
(8192, 8192, 8192)	1131.45	472.15

Pipelining with Warp Specialization





Warp Specialization



TLX GEMM: Warp-Specialized Pipelining

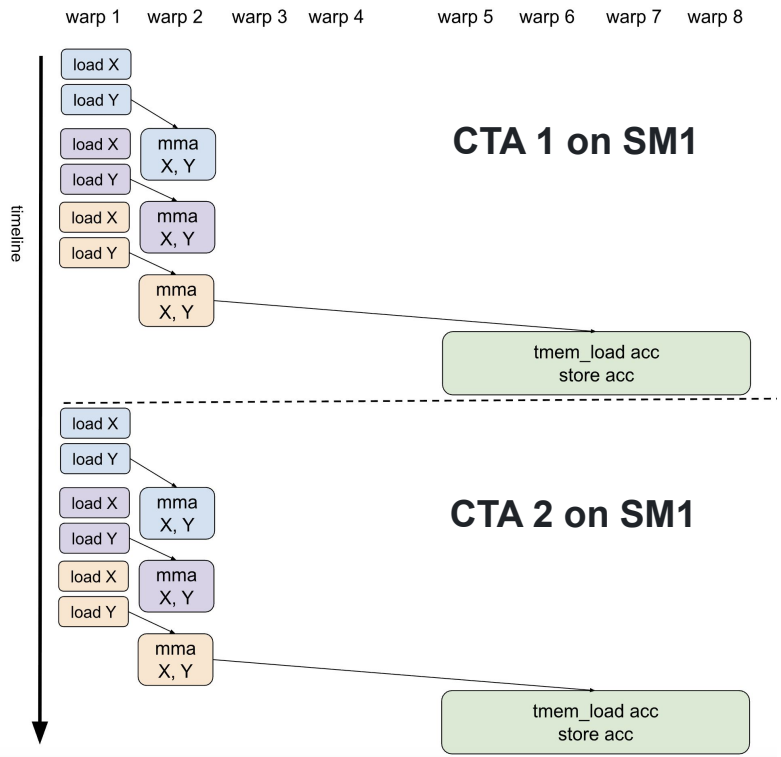
```
@triton.jit
def tlx_matmul_ws(...):
    a = tlx.local_alloc((BLOCK_M, BLOCK_K), dtype_a, NUM_IN_BUFFERS)
    b = tlx.local_alloc((BLOCK_K, BLOCK_N), dtype_b, NUM_IN_BUFFERS)
    acc = tlx.local_alloc((BLOCK_M, BLOCK_N), tl.float32, 1, tlx.storage_kind.tmem)

    bar_full = tlx.alloc_barriers(num_barriers=NUM_IN_BUFFERS, arrive_count=1)
    bar_empty = tlx.alloc_barriers(num_barriers=NUM_IN_BUFFERS, arrive_count=1)
    bar_acc = tlx.alloc_barriers(num_barriers=1, arrive_count=1)

    with tlx.async_tasks():
        # Producer: TMA loads
        with tlx.async_task(num_warps=1):
            for k in range(num_k_iters):
                buf, phase = _get_bufidx_phase(k, NUM_IN_BUFFERS)
                tlx.barrier_wait(bar_empty[buf], phase ^ 1)
                tlx.barrier_expect_bytes(
                    bar_full[buf],
                    (BLOCK_M + BLOCK_N) * BLOCK_K * tlx.size_of(dtype_a),
                )
                offs_k = k * BLOCK_K
                tlx.async_descriptor_load(a_desc, a[buf], [offs_am, offs_k], bar_full[buf])
                tlx.async_descriptor_load(b_desc, b[buf], [offs_k, offs_bn], bar_full[buf])

        # MMA Consumer:
        with tlx.async_task(num_warps=1):
            for k in range(num_k_iters):
                buf, phase = _get_bufidx_phase(k, NUM_IN_BUFFERS)
                tlx.barrier_wait(bar_full[buf], phase)
                acc = tlx.async_dot(a[buf], b[buf], acc[0], use_acc=k > 0, mBarriers=[bar_empty[buf]])
            tlx.tcgen05_commit(bar_acc[0])

        # Epilogue: store results to global memory
        with tlx.async_task("default"):
            tlx.barrier_wait(bar_acc[0], phase=0)
            c = tlx.local_load(acc[0])
            c_desc.store([offs_am, offs_bn], c.to(tl.float16))
```

(M, N, K)	aten_matmul-tflops	tlx_matmul-tflops
(8192, 8192, 8192)	1131.45	790.642

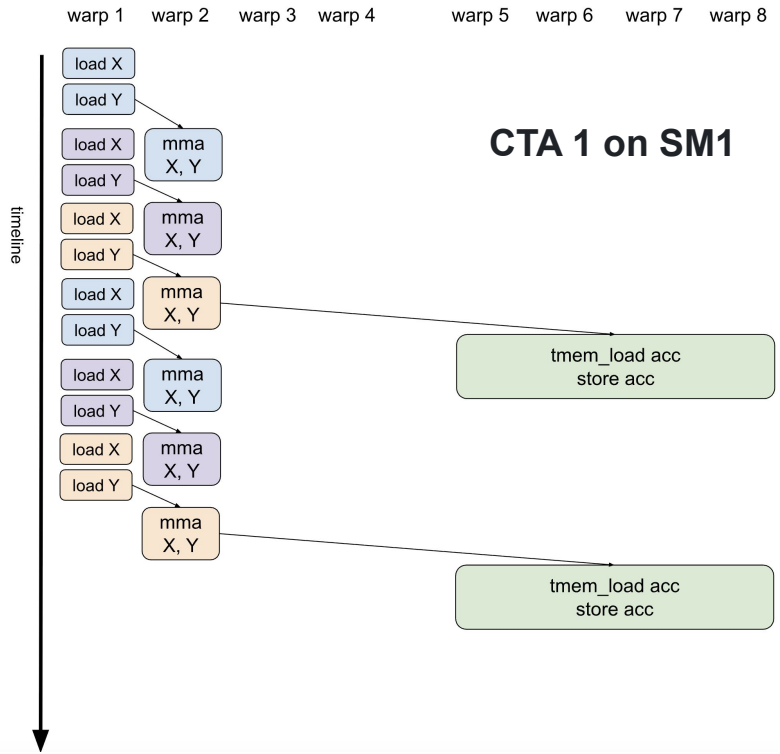
Pipelining Prolog & Epilog with Persistent loop

- Resource constraint
  - Only one CTA can reside on an SM due to shared memory / register limits
- Performance issue
  - Prolog and epilog cannot overlap across CTAs, causing pipeline bubbles
- Key idea
  - Use a persistent loop to reuse the same CTA and overlap prolog/epilog with steady-state execution

Pipelining with Persistent Loop



Non-persistent



Persistent

Persistent with CLC

Producer: Async TMA Loads

```
# Producer: async TMA loads
with tlx.async_task(num_warps=1):
    tile_id = start_pid
    clc_phase_consumer = 0
    while tile_id != -1:
      for k in range(num_k_iters):
          tlx.barrier_wait(bar_empty[buf], phase ^
              1)
          tlx.barrier_expect_bytes(bar_full[buf],
              (BLOCK_N * BLOCK_N) * BLOCK_K *
              tlx.size_of(dtype_a))
          tlx.async_descriptor_load(a_desc, a[buf],
              [offs_aa, offs_k], bar_full[buf])
          tlx.async_descriptor_load(b_desc, b[buf],
              [offs_k, offs_bn], bar_full[buf])

      tile_id = tlx.clc_consumer(clc_context, 0,
          clc_phase_coosumer)
      clc_phase_consumer ^= 1
```

MMA Consumer: WGMMA Operations

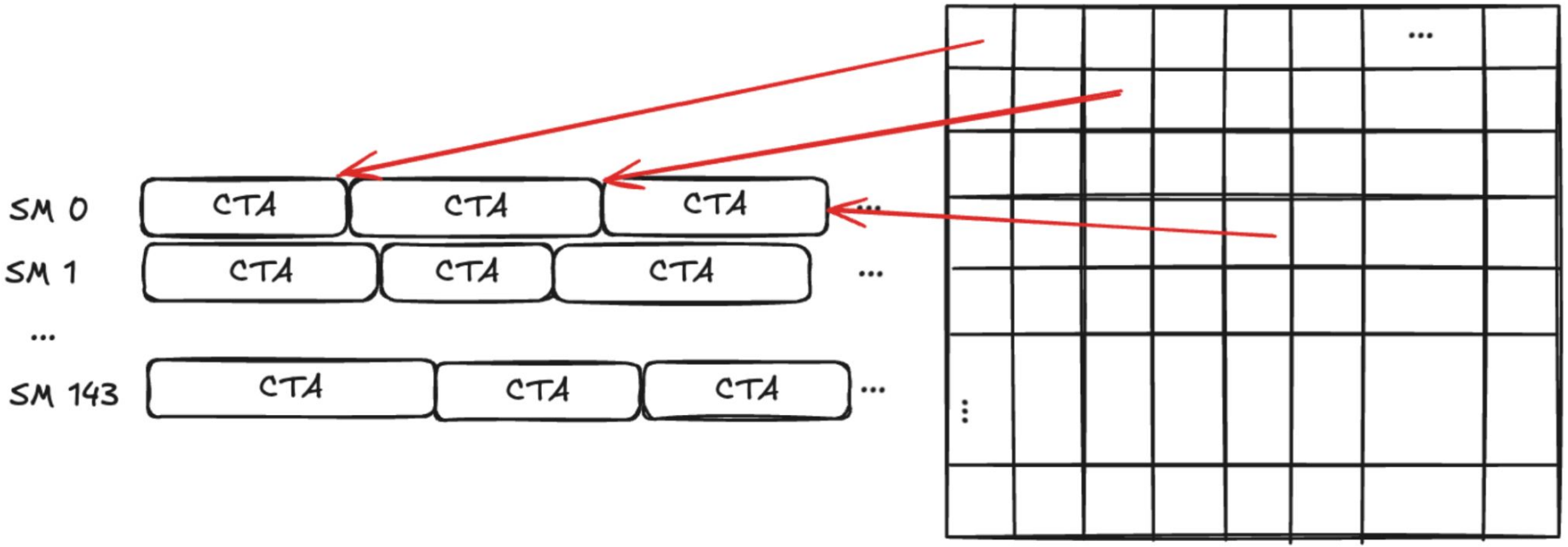
```
# MMA Consumer: wgmma operations
with tlx.async_task(num_warps=1):
    tile_id = start_pid
    clc_phase_consumer = 0
    while tile_id != -1:
      for k in range(num_k_iters):
          tlx.barrier_wait(bar_full[buf], phase)
          acc = tlx.async_dot(a[buf], b[buf], acc[0],
              use_acc=k > 0,
              mBarriers=[bar_empty[buf]])
      tlx.tcgen05_commit(bar_acc[0])
      ...
      tile_id = tlx.clc_consumer(clc_context, 0,
          clc_phase_consumer)
      clc_phase_consumer ^= 1
```

Epilogue: Store to Global Memory

```
# CLC context: 3 consumers (epilogue, MMA, TMA
    producer), 1 stage
clc_context = tlx.clc_create_context(1, 3)
# Epilogue: store results to global memory
with tlx.async_task("default"):
    tile_id = start_pid
    clc_phase_producer = 1
    clc_phase_consumer = 0
    while tile_id != -1:
      # Only epilogue task issues clc_preducer
      tlx.clc_producer(clc_context, 0, clc_phase_producer)
      clc_phase_producer ^= 1
      ...
      tlx.barrier_wait(bar_acc[0], phase=0)
      c = tlx.local_load(acc[0])
      c_desc.store([offs_aa, offs_bn], c.to(tl.float16))

      tile_id = tlx.clc_censuner(clc_context, 0,
          clc_phase_consuxer)
      clc_phase_consumer ^= 1
```

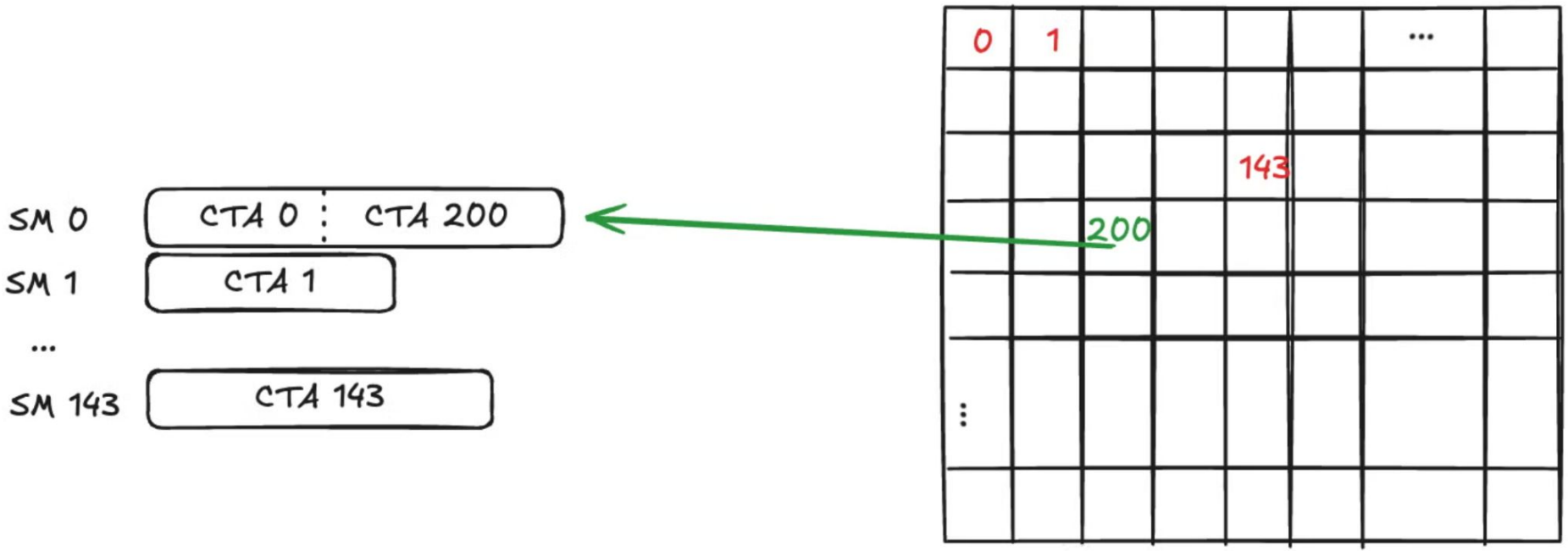
(M, N, K)	aten_matmul-tflops	tlx_matmul-tflops
(8192, 8192, 8192)	1143.5	832.318

Cluster Launch Control (CLC)



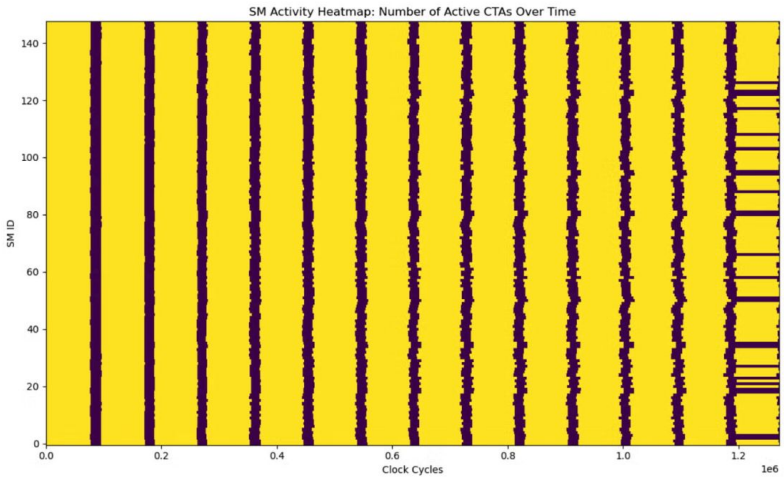
Non-persistent

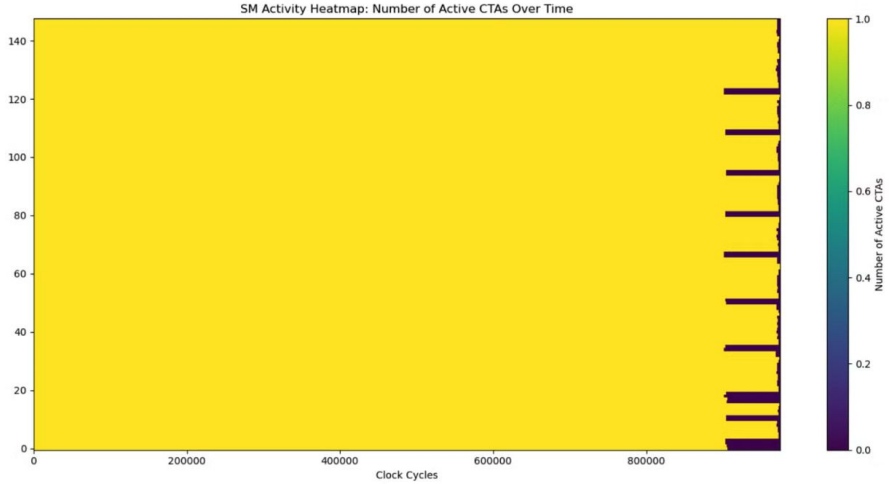
Cluster Launch Control (CLC)



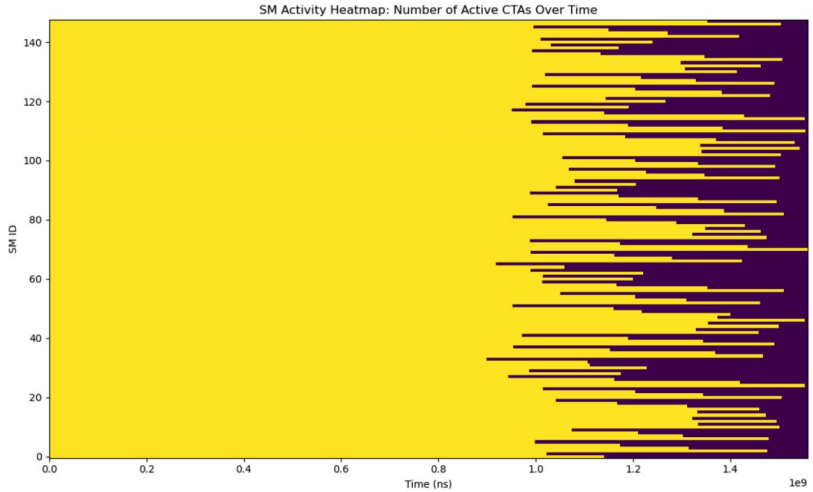
Persistent

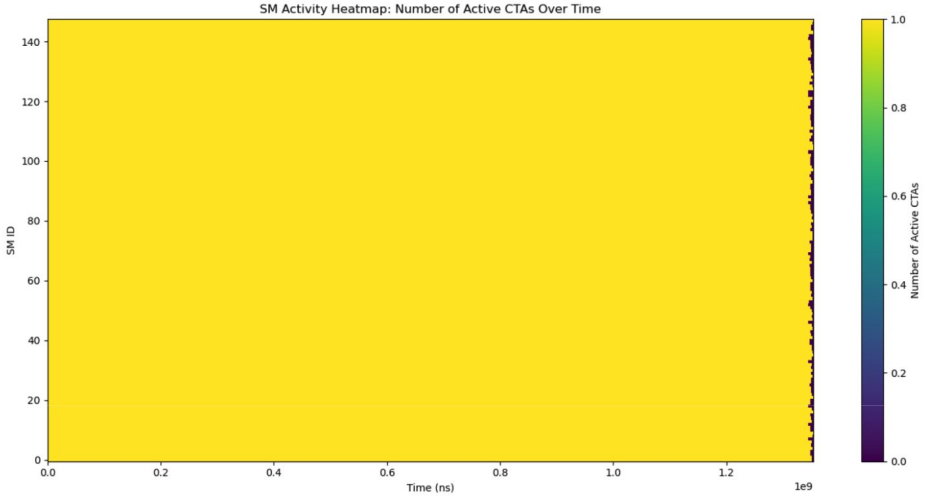
SM Heatmaps: non-persistent vs CLC (GEMM)



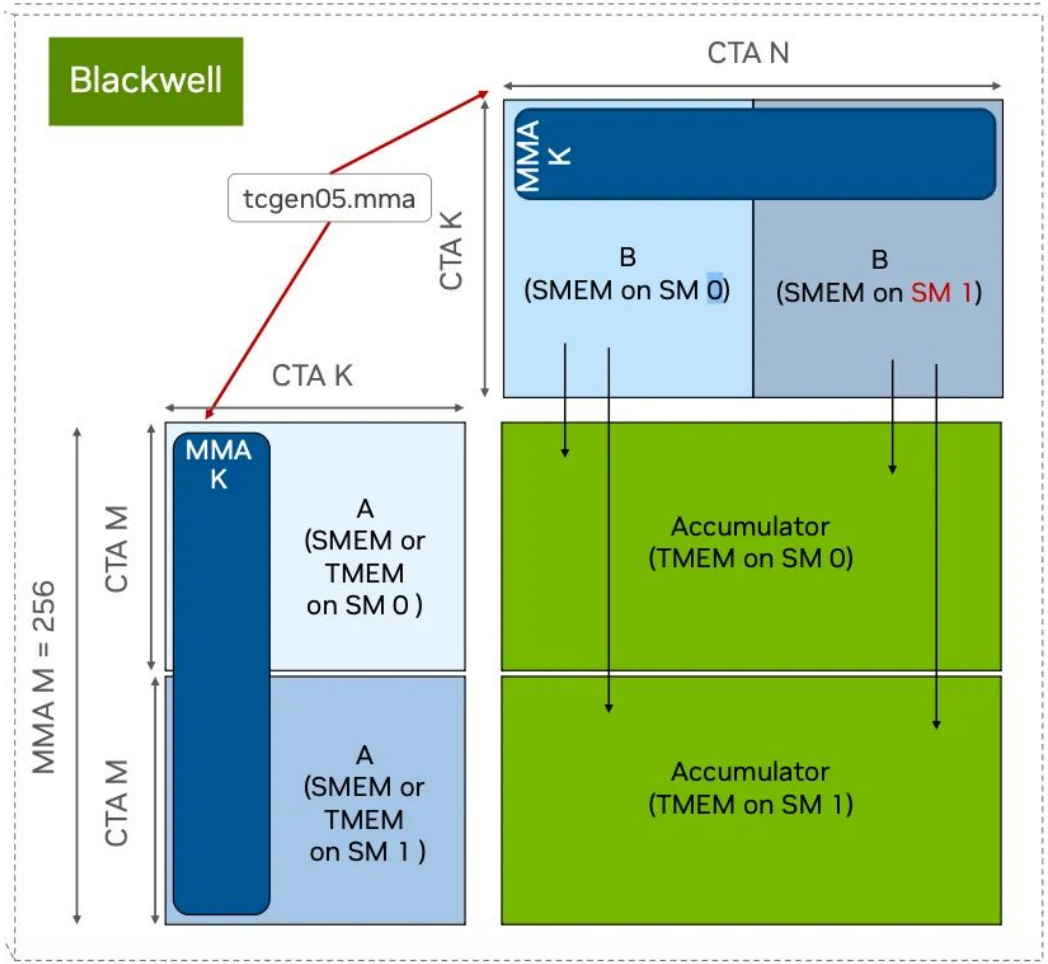


SM Heatmaps: non LB vs CLC (internal kernel)

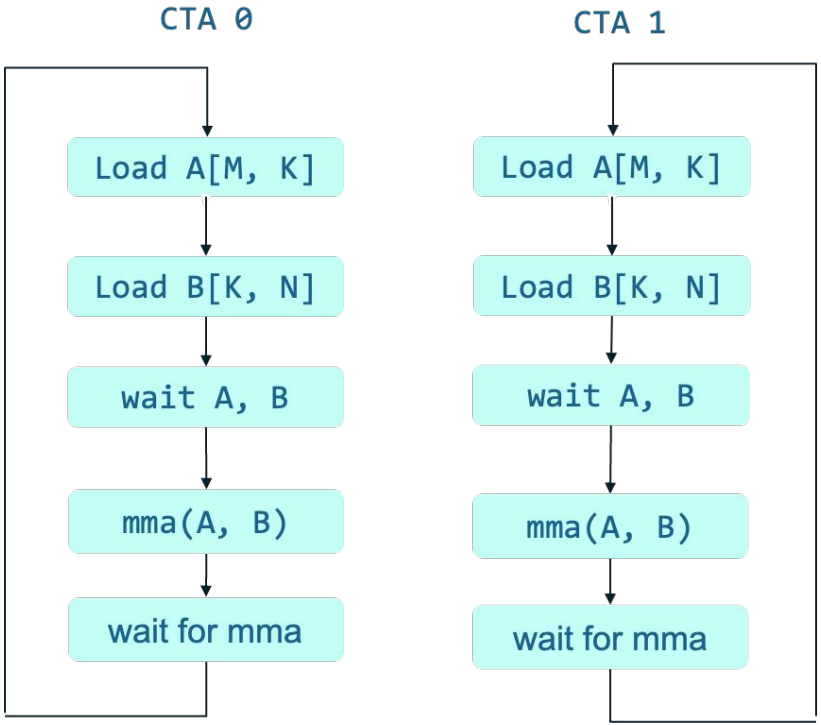




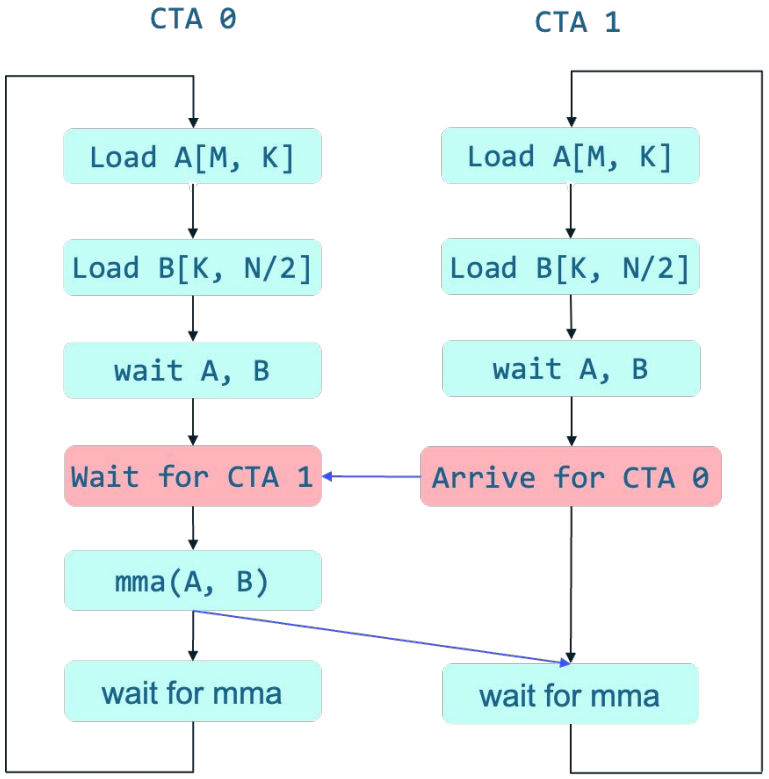
Increasing Tile Size with Paired-CTA MMA



Paired-CTA MMA with TLX



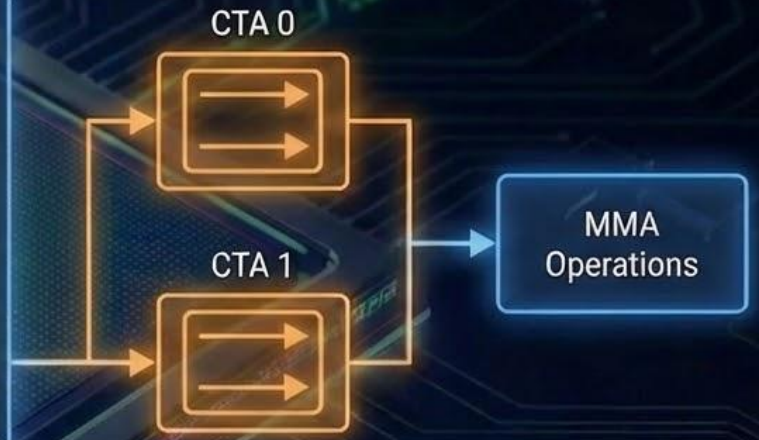
Single-CTA



Paired-CTA

Paired-CTA MMA with TLX

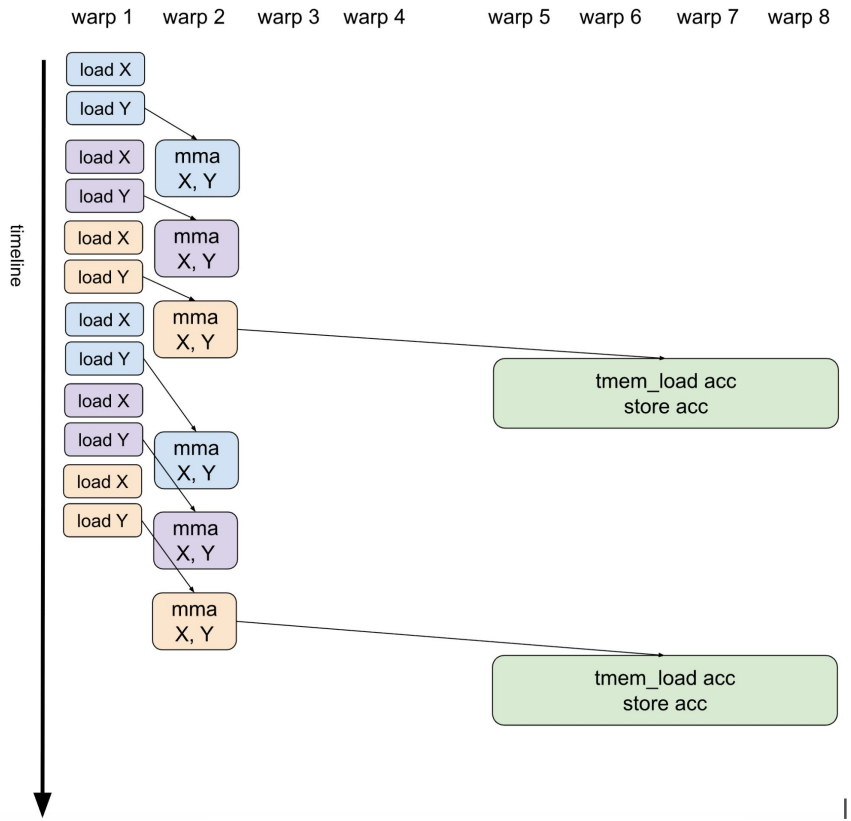
```
# MMA Consumer
with tlx.async_task(num_warps=1):
    for k in range(num_k_iters):
        buf, phase = _get_bufidx_phase(k, NUM_IN_BUFFERS)
        tlx.barrier_wait(bar_full[buf], phase)
        # CTA0 waits for CTA0 and CTA1 to finish loading A and B before
issuing dot op
        if PAIR_CTA:
            tlx.barrier_arrive(cta_bars[buf], arrive_count=1,
remote_cta_rank=0)
            tlx.barrier_wait(cta_bars[buf], phase=phase, pred=pred_cta0)
        acc = tlx.async_dot(a[buf], b[buf], acc[0], use_acc=k > 0,
mBarriers=[bar_empty[buf]], two_ctas=PAIR_CTA)
        tlx.tcgen05_commit(bar_acc[0], two_ctas=PAIR_CTA)
```



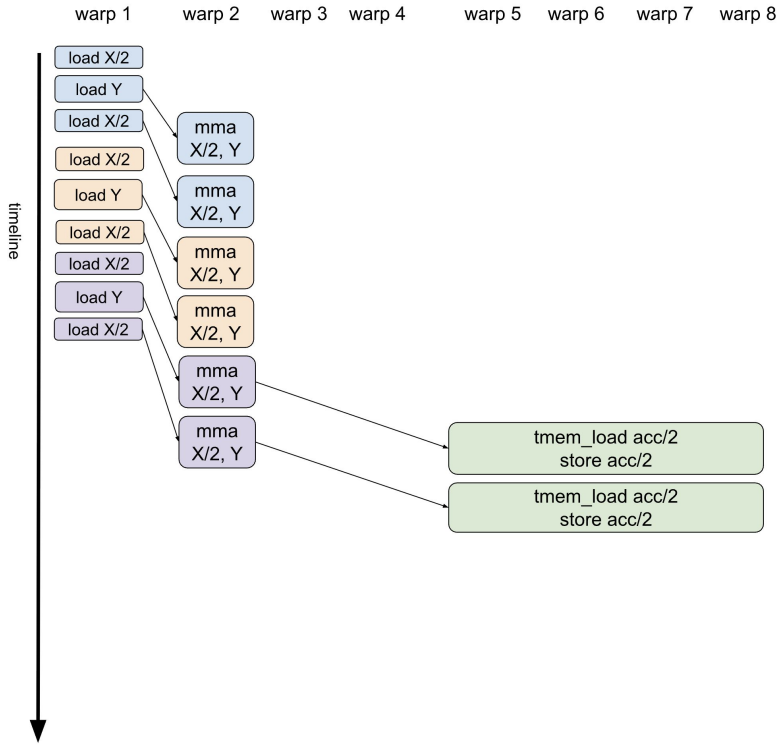
best config selected: BLOCK_SIZE_M: 128, BLOCK_SIZE_N: 256, BLOCK_SIZE_K: 64

(M, N, K)	aten_matmul-tflops	tlx_matmul-tflops
(8192, 8192, 8192)	1143.5	1075.89

Cooperative Data Partitioning



With one tmem buffer 128x256



With Data partitioning 256x256

Producer with Data Partitioning

```
a = tlx.local_alloc((BLOCK_M // NUM_MMA_GROUPS, BLOCK_K), dtype_a, NUM_IN_BUFFERS * NUM_MMA_GROUPS)
b = tlx.local_alloc((BLOCK_K, BLOCK_N), dtype_b, NUM_IN_BUFFERS)

bar_full_A = tlx.alloc_barriers(num_barriers=NUM_IN_BUFFERS * NUM_MMA_GROUPS)
bar_empty_A = tlx.alloc_barriers(num_barriers=NUM_IN_BUFFERS * NUM_MMA_GROUPS)
bar_full_B = tlx.alloc_barriers(num_barriers=NUM_IN_BUFFERS)
bar_empty_B = tlx.alloc_barriers(num_barriers=NUM_IN_BUFFERS)
```

```
with tlx.async_task(num_warps=1):
    for k in range(num_k_iters):
        buf, phase = _get_bufidx_phase(k, NUM_IN_BUFFERS)
        offs_k = k * BLOCK_K
        for group_id in tl.static_range(1, NUM_MMA_GROUPS):
            a_buf = group_id * NUM_SMEM_BUFFERS + buf
            tlx.barrier_wait(bar_empty_A[a_buf], phase ^ 1)
            tlx.barrier_expect_bytes(bar_full_A[a_buf], dsize * BLOCK_M_SPLIT * BLOCK_SIZE_K)
            offs_am2 = offs_am + group_id * BLOCK_M_SPLIT
            tlx.async_descriptor_load(a_desc, buffers_A[a_buf], [offs_am2, offs_k], bar_full_A[a_buf])
        tlx.barrier_wait(bar_empty_B[buf], phase ^ 1)
        tlx.barrier_expect_bytes(bar_full_B[buf], (BLOCK_M + BLOCK_N) * BLOCK_K) * tlx.size_of(dtype_b)
        tlx.async_descriptor_load(b_desc, b[buf], [offs_k, offs_bn], bar_full_B[buf])
```

MMA with Data Partitioning

```
acc = tlx.local_alloc((BLOCK_M // NUM_MMA_GROUPS, BLOCK_N), tl.float32, 1, tlx.storage_kind.tmem)
bar_acc = tlx.alloc_barriers(num_barriers=NUM_MMA_GROUPS)
```

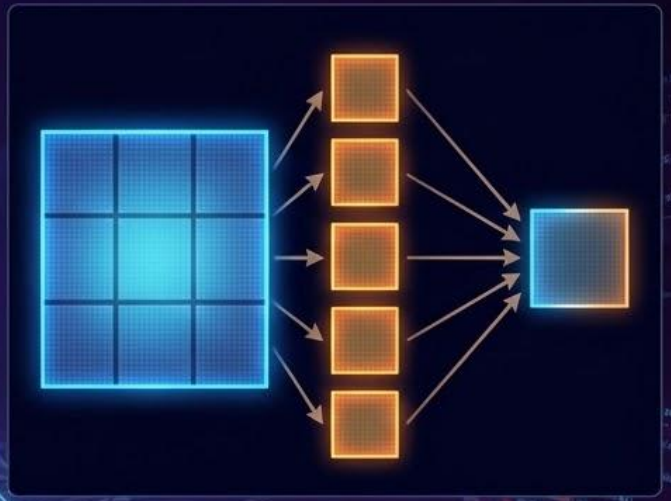
```
# MMA Consumer:
with tlx.async_task(num_warps=1):
    for k in range(num_k_iters):
        buf, phase = _get_bufidx_phase(k, NUM_IN_BUFFERS)
        for group_id in tl.static_range(1, NUM_MMA_GROUPS):
            a_buf = group_id * NUM_SMEM_BUFFERS + buf
            acc_buf = group_id * NUM_TMEM_BUFFERS + cur_tmem_buf
            tlx.barrier_wait(bar_full_A[a_buf], phase)
            tlx.barrier_wait(bar_full_B[buf], phase)
            if group_id == NUM_MMA_GROUPS - 1:
                acc = tlx.async_dot(a[a_buf], b[buf], acc[0], use_acc=k > 0,
                  mBarriers=[bar_empty_A[a_buf], bar_empty_B[buf]])
            else:
                acc = tlx.async_dot(a[a_buf], b[buf], acc[0], use_acc=k > 0,
                 mBarriers=[bar_empty_A[a_buf]])
    for group_id in tl.static_range(1, NUM_MMA_GROUPS):
        a_buf = group_id * NUM_SMEM_BUFFERS + buf
        tlx.tcgen05_commit(bar_acc[a_buf])
```

Epilog with Data Partitioning

```
acc = tlx.local_alloc((BLOCK_M // NUM_MMA_GROUPS, BLOCK_N),
      tl.float32, 1, tlx.storage_kind.tmem)
bar_acc = tlx.alloc_barriers(num_barriers=NUM_MMA_GROUPS)


# Epilog:
with tlx.async_task("default"):
    for group_id in tl.static_range(1, NUM_MMA_GROUPS):
        buf = group_id * NUM_MMA_GROUPS
        tlx.barrier_wait(bar_acc[buf], phase=0)
        c = tlx.local_load(acc[buf])
        offs_am2 = offs_am + group_id * BLOCK_M // NUM_MMA_GROUPS
        c_desc.store([offs_am2, offs_bn], c.to(tl.float16))
```

Epilog Sub Slicing

```
with tlx.async_task("default"):
       for group_id in tl.static_range(1, NUM_MMA_GROUPS):
          buf = group_id * NUM_MMA_GROUPS
          tlx.barrier_wait(bar_acc[buf], phase=0)
          for slice_id in tl.static_range(EPILOQUE_SUBTILE):
             acc_subslice = tlx.local_slice(
                 acc[buf],
                 [0, slice_id * slice_size],
                 [BLOCK_M_SPLIT, slice_size],
             )
             c = tlx.local_load(acc_subslice)
             offs_am2 = offs_am + group_id * BLOCK_M // NUM_MMA_GROUPS
             c_desc.store([offs_am, offs_bn+ slice_id * slice_size
c.to(tl.float6))
```



best config selected: BLOCK_SIZE_M: 256, BLOCK_SIZE_N: 256, BLOCK_SIZE_K: 64

(M, N, K)	aten_matmul-tflops	tlx_matmul-tflops
(8192, 8192, 8192)	1143.46	1191.39

Tunable Optimization Techniques

https://github.com/facebookexperimental/triton/blob/main/third_party/tlx/tutorials/blackwell_gemm_ws.py

```
def get_cuda_autotune_config():
    return [
        triton.Config(
            {
                "BLOCK_SIZE_M": 256,
                "BLOCK_SIZE_N": 256,
                "BLOCK_SIZE_K": 64,
                "GROUP_SIZE_M": 8,
                "NUM_SMEM_BUFFERS": 4,
                "NUM_TMEM_BUFFERS": 1,
                "NUM_MMA_GROUPS": 2,
                "EPILOGUE_SUBTILE": 4,
                "PAIR_CTA": True,
                "CLUSTER_LAUNCH_CONTROL": True,
            },
            num_warps=4,
            num_stages=1,
            pre_hook=matmul_tma_set_block_size_hook,
            ctas_per_cga=(2, 1, 1),
        )
    ]
```

TLX Resources

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