

High-Velocity GPU Kernel Authoring with CUTLASS Python

CuTe DSL extensions – CUTLASS Primitives – Task Scheduling

Guray Ozen

Principal Compiler Engineer

Michael Goldfarb

Senior Deep Learning Performance Engineer

PyTorch Conference North America | 20 October 2026

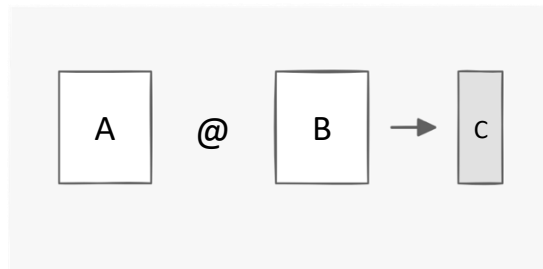


Programming Tensor Cores

Pick the level of abstraction based on your needs

PyTorch

Tensor operators

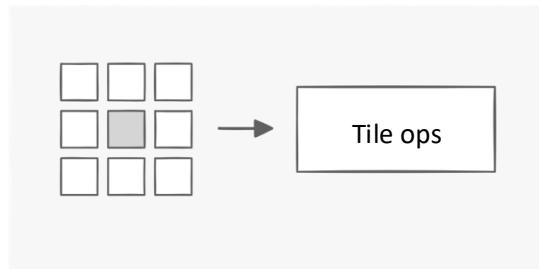


Automatic

Framework / graph compiler
chooses the kernel

cuTile

Tile programs

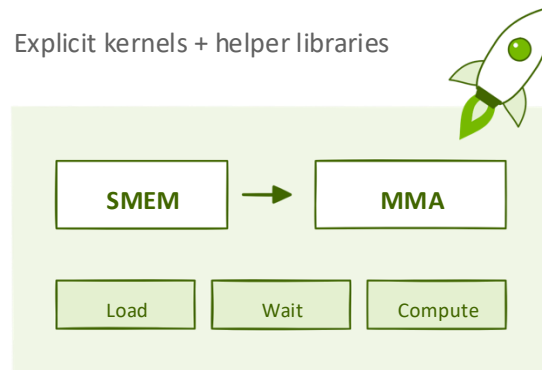


Block-level control

You choose tile operations;
compiler maps the threads

CUTLASS Python

Explicit kernels + helper libraries



Full kernel control

You choose layouts, movement
and instruction-level behavior

PyTorch + CUTLASS Python

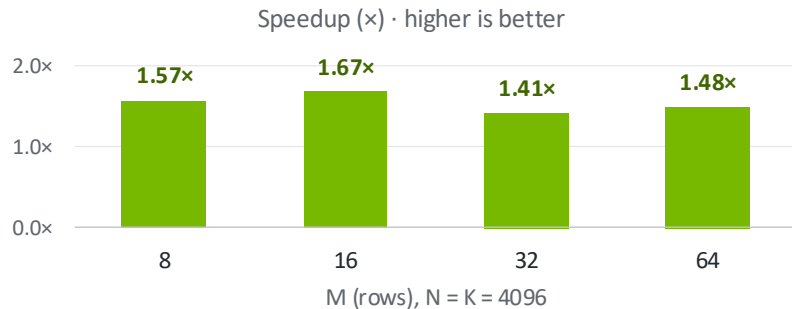


CUTLASS Python supported natively in PyTorch and FlexAttention

[Blog](#)

Generating State-of-the-Art GEMMs with TorchInductor's CuteDSL backend

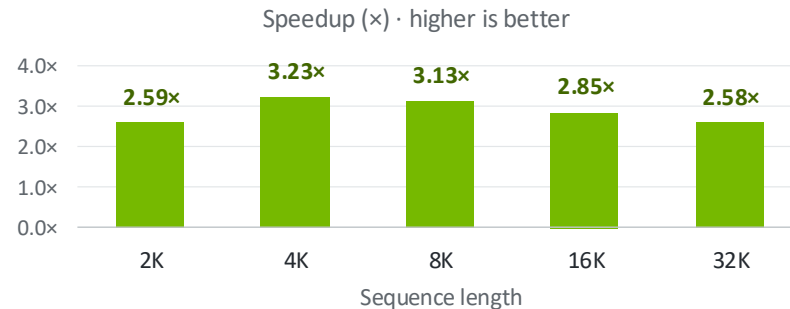
Selected BF16 GEMMs · B200 · vs best ATen/Triton



[Blog](#)

FlexAttention + FlashAttention-4: Fast and Flexible

Dense forward · GB200 · BF16 · vs Triton

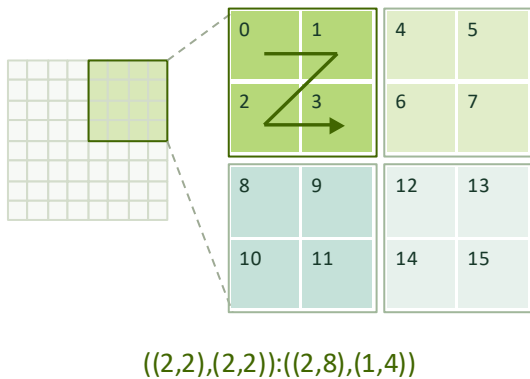


CUTLASS Python

A collection of Python DSLs and libraries for high-performance Tensor Core programming.

CuTe

Verifiable layouts and computations



Primitives

Typed GPU instructions

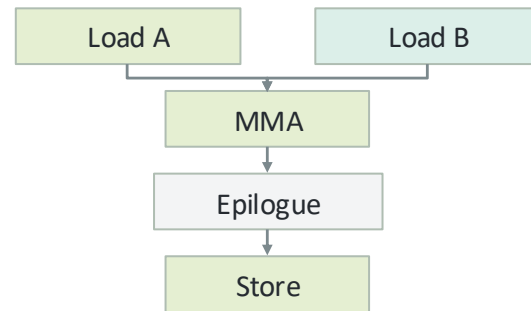
```
fp8 = prims.convert(  
    source, cutlass.Float8E4M3FN,  
    random_bits=entropy,  
)
```

PTX

```
cvt.rs.satfinite.e4m3x4.f32  
%r11, {%r9, %r8, %r7, %r6},  
%r10;
```

Libraries

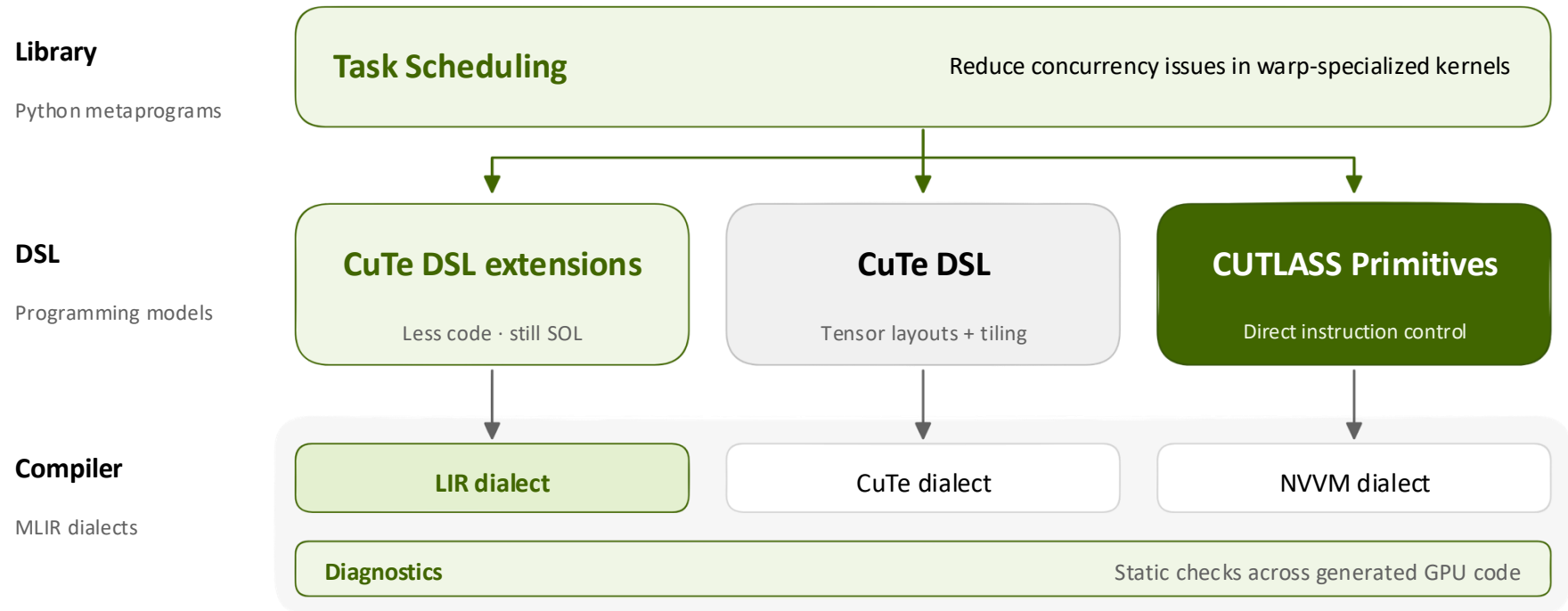
Reusable kernel building blocks



Tasks, pipelines and epilogues and operators

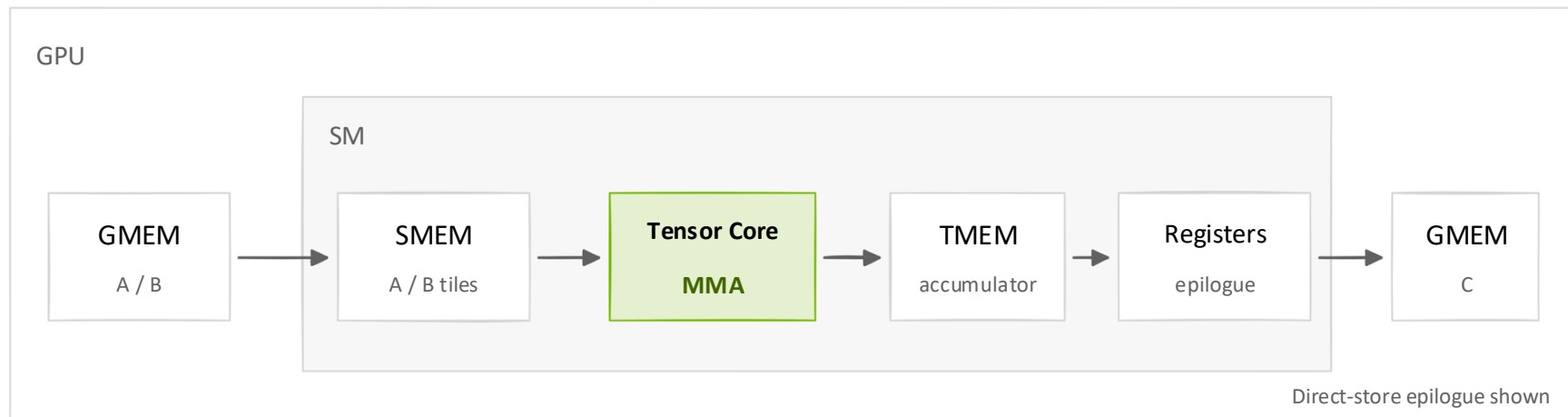
The CUTLASS Python Stack

Extensible Python Libraries build on DSLs backed by MLIR compiler foundation



A Typical GEMM

Orchestration of data movement and compute.



1. TMA load

Async copy + completion barrier

2. Tensor Core MMA

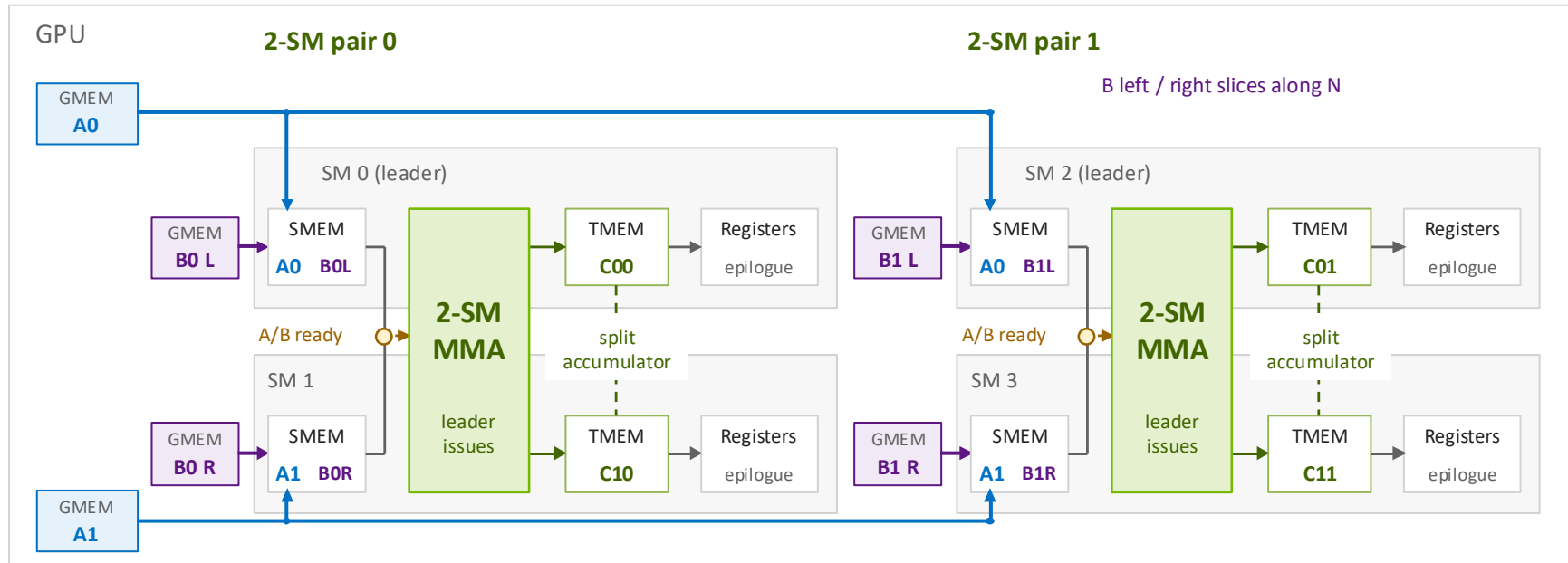
Shared tiles → TMEM accumulator

3. Epilogue + store

Registers → output

A Performant GEMM

Realistic example of Blackwell GEMM



1. A multicast, B per SM

B slices complete the pair's input tile

2. One MMA per pair

Leader waits for both CTAs' inputs

3. Paired TMEM lifetime

Allocate together, drain, then free

New CuTe DSL Extensions

Zero cost abstractions to simplify peak-performance kernels

50+ operations and helpers

26

Pipeline +
synchronization

7

Memory
+ TMA

6

Tensor
algorithms

6

Layout + cluster
helpers

5

Tensor Core
compute

2

Buffering +
mailbox helpers

TMA Atom and Coordinate Inferred from Memory Views

Compiler derives TMA atoms, coordinate tensors and copy partitions.

CuTe DSL: construct TMA objects

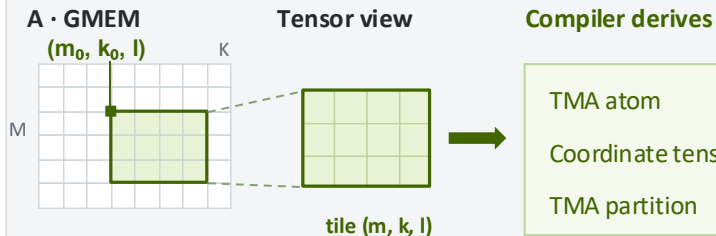
```
@cute.jit
def launch_gemm(A):
    atom_A, coords_A = cute.nvgpu.make_tiled_tma_atom_A(
        tma_op_A, A, smem_layout_A, ...)
    kernel(atom_A, coords_A).launch(...)
```

```
@cute.kernel
def kernel(atom_A, coords_A):
    sA = smem.allocate_tensor(...)
    gA = cute.local_tile(coords_A, tile_mk, tile_coord)
    gA = thr_mma.partition_A(gA)
    a_cta = cute.make_layout(...)
    sTMA, gTMA = cpasync.tma_partition(
        atom_A, cta_coord[2], a_cta,
        cute.group_modes(sA, 0, 3),
        cute.group_modes(gA, 0, 3))
    mask_A = cpasync.create_tma_multicast_mask(
        cluster, cta_coord, mcast_mode=2)
    cute.copy(atom_A, gTMA[None, m, k, l],
              sTMA[None, stage], tma_bar_ptr=bar,
              mcast_mask=mask_A)
```

CuTe extensions: pass and load tensor views

```
@cute.jit
def launch_gemm(A):
    kernel(A).launch(...)

@cute.kernel
def kernel(A):
    sA = smem.allocate_tensor(...)
    gA = cute.local_tile(A, tile_mk, tile_coord)
    gA = thr_mma.partition_A(gA)
    cute_ext.tma_load(gA[None, None, None, m, k, l],
                      sA[None, None, None, stage], bar,
                      vmnk_layout=cluster, multicast_mode=2,
                      tma_operation_type=Op.SM90_TMA_LOAD_MULTICAST)
```



TMA Update for Efficient Group GEMMs

Simplified management and update of Tensor Maps

CuTe DSL: manage descriptor slots, pointers and fencing

```
desc_A = tm.get_tensormap_ptr(ws[cta, 0, None].iterator)
desc_B = tm.get_tensormap_ptr(ws[cta, 1, None].iterator)
```

if group_changed:

```
A = cute.make_tensor(ptr_A, layout_A)
B = cute.make_tensor(ptr_B, layout_B)
```

```
tm.update_tensormap((A, B), (atom_A, atom_B),
                    (desc_A, desc_B), tma_warp, (sdesc_A, sdesc_B))
tm.fence_tensormap_update(desc_A)
tm.fence_tensormap_update(desc_B)
```

```
cute.copy(atom_A, coords_A_k, sA_stage,
          tma_bar_ptr=bar,
          tma_desc_ptr=tm.get_tensormap_ptr(desc_A, generic))
cute.copy(atom_B, coords_B_k, sB_stage,
          tma_bar_ptr=bar,
          tma_desc_ptr=tm.get_tensormap_ptr(desc_B, generic))
```

Host Workspace Allocation

```
req = query_host_workspace(A, B) # Generated by compiler
workspace = torch.empty(req.size_in_bytes, dtype=torch.uint8, device="cuda")
```

CuTe DSL extensions: Publish views directly to TMA

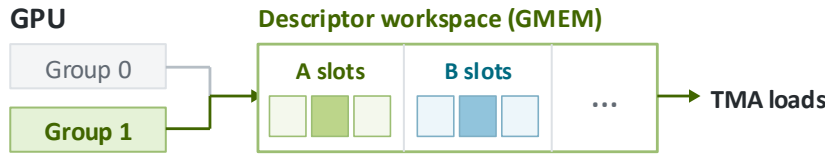
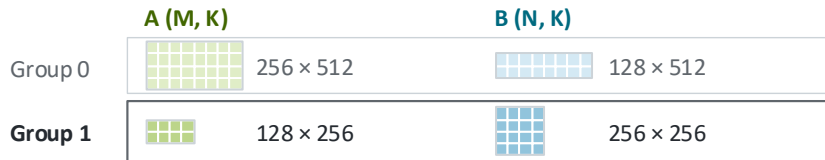
if group_changed:

```
A = cute.make_tensor(ptr_A, layout_A)
B = cute.make_tensor(ptr_B, layout_B)
```

```
A = cute_ext.tma_update(A)
B = cute_ext.tma_update(B)
```

Tile A and B, then load as usual

```
cute_ext.tma_load(a_tile, sA_stage, bar)
cute_ext.tma_load(b_tile, sB_stage, bar)
```



Mixed Cluster Shapes from One Kernel

preferred = (2, 2, 1), fallback = (2, 1, 1). Hardware selects the shape of each cluster.

CuTe DSL: write a code path per cluster shape

```
@cute.jit
def launch_gemm(A, B, ...):
    atom_A_p, tA_p = cute.nvgpu.make_tiled_tma_atom_A(
        op_A_p, A, smem_layout_A, ..., vmnk_p.shape)
    atom_A_f, tA_f = cute.nvgpu.make_tiled_tma_atom_A(
        op_A_f, A, smem_layout_A, ..., vmnk_f.shape)
    ... # same for B; scheduler params per shape
    kernel(atom_A_p, tA_p, atom_A_f, tA_f, ...).launch(
        cluster=preferred, fallback_cluster=fallback,
        smem_merge_branch_allocs=True)
```

```
@cute.kernel
def kernel(atom_A_p, tA_p, atom_A_f, tA_f, ...):
    cx, cy, _ = cute.arch.block_in_cluster_dim()
    if cx == preferred[0] and cy == preferred[1]:
        gemm_body(atom_A_p, tA_p, ..., vmnk_p)
    else:
        gemm_body(atom_A_f, tA_f, ..., vmnk_f)
```

CuTe DSL extensions: one body for both shapes

```
@cute.jit
def launch_gemm(A, B, ...):
    kernel(A, B, ...).launch(grid=grid, ...,
        cluster=preferred, fallback_cluster=fallback)
```

```
@cute.kernel
def kernel(A, B, ...):
    shape = cute_ext.get_cluster_shape(
        preferred, fallback)
    cluster = cute.tiled_divide(
        cute.make_layout(shape), ...)
    ... # same code as a single-cluster kernel
    cute_ext.tma_load(gA_k, sA_stage, bar,
        vmnk_layout=cluster, multicast_mode=2, ...)
```

Example: 18 SMs in one GPC



■ 2x2 preferred

■ 2x1 fallback

Compiler derives per shape

- TMA descriptors
- Multicast masks + layouts
- Specialized kernel body

TMEM Allocator Planning

One plan defines overlapping TMEM views and their safe reuse

CuTe DSL: manual synchronized plan and allocation

```
overlap_cols = 2 * tile_n + aux_cols - max_cols
stage_stride = (
    (tile_n - overlap_cols) * one_stage.stride[0][1])
acc_layout = cute.make_layout(
    cute.append(one_stage.shape, 2),
    stride=cute.append(one_stage.stride, stage_stride))
```

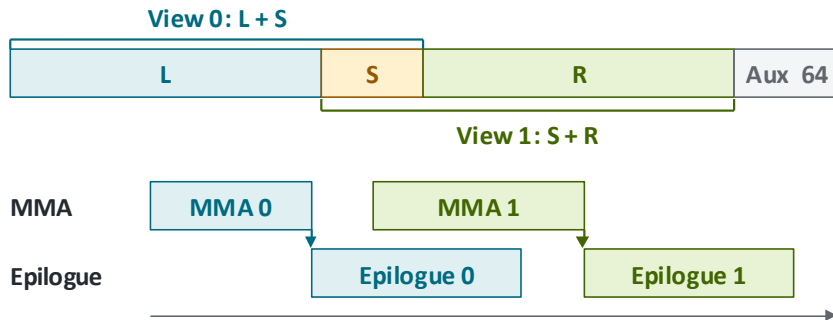
```
alloc_sync = NamedBarrier(2, 32 + 128)
epi_sync = NamedBarrier(1, 128)
tmem = TmemAllocator(
    barrier_for_retrieve=alloc_sync, is_two_cta=True)
if is_epi_warp:
    tmem.allocate(max_cols)
if is_mma_warp or is_epi_warp:
    tmem.wait_for_alloc()
    ptr = tmem.retrieve_ptr(acc_dtype)
    acc_buffer = cute.make_tensor(ptr, acc_layout)
... # Auxiliary views, MMA and epilogue.
```

```
if is_epi_warp:
    tmem.relinquish_alloc_permit()
    epi_sync.arrive_and_wait()
    tmem.free(ptr)
```

CuTe DSL extensions: define a plan and buffer

```
plan = cute_ext.plan_accumulator_buffering(
    one_stage_layout=one_stage,
    accumulator_dtype=acc_dtype, epilogue_tile=epi_tile,
    auxiliary_tmem_allocations=aux_allocations,
    allow_overlap=True)
```

```
acc_buffer = cute_ext.allocate(
    acc_dtype, cutlass.AddressSpace.tmem,
    plan.storage_layout, alignment=16, is2cta=True)
```



Epilogue Pipeline Consumes TMEM Plan

Epilogue safely consumes TMEM buffer to unblock next MMA

CuTe DSL: explicit overlap epilogue

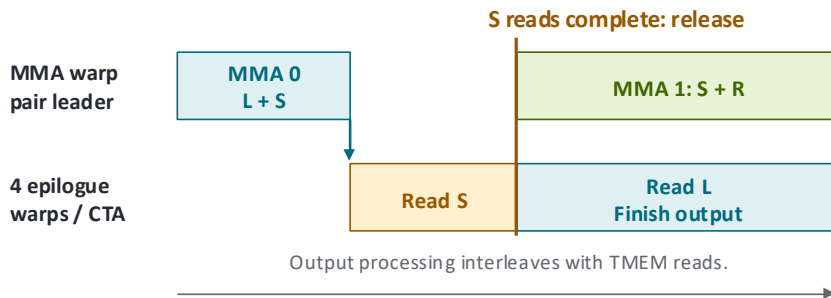
```
acc_epi = cute.flat_divide(
    acc_buffer[(None, None), 0, 0, None], epi_tile)
acc_views = cute.group_modes(
    t2r.get_slice(tid).partition_S(acc_epi), 3, 5)
acc_pipe.consumer_wait(cons)
phase = cons.phase
for i in cutlass.range(subtiles):
    j = i
    if phase == 0:
        j = subtiles - 1 - i
    cute.copy(t2r, acc_views[None, None, None, j, phase], rAcc)
    if i == release_at:
        cute.arch.fence_view_async_tmemb_load()
        acc_pipe.consumer_release(cons)
        cons.advance()
    values = r2s.retile(rAcc).load()
    rD.store(epilogue_op(values.to(d_dtype)))
    stage = (tile_count * subtiles + i) % store_stages
    cute.copy(r2s, rD, sD_part[None, None, None, stage])
    cute.arch.fence_proxy("async.shared", space="cta")
    epi_sync.arrive_and_wait()
    if warp_idx == store_warp:
        cute.copy(atom_D, sD_tma[None, stage], gD_tma[None, j])
        store_pipe.producer_commit()
        store_pipe.producer_acquire()
    epi_sync.arrive_and_wait()
```

CuTe DSL extensions: simplify efficient epilogue

```
view, drain = plan.consumer_wait_and_get_view(
    acc_pipe, acc_buffer)
```

```
store_pipe, acc_pipe = cute_ext.epilogue_tma_store(
    tile_shape, True, view[(None, None), 0, 0],
    D, tile_coord, store_pipe, 0, epilogue_op,
    accumulator_drain=drain)
```

TMEM reuse after shared reads



New CUTLASS Primitives

- 1 Explicit control
- 2 SIMT programming model
- 3 Toward full instruction set coverage

160+ operation APIs

49

Synchronization

45

Math +
conversion

28

Tensor Core
+ TMEM

24

Memory +
atomics

14

Control +
assembly

5

Warp
collectives

Three new types: Array, Pointer, Vector

Storage, addresses and register values without CuTe layout algebra.

cutlass.Array

Memory

3	1	4	1
5	9	2	6

Own storage or view it

Shape · strides · dtype

cutlass.Pointer

$p + 2$



3	1	4	1
---	---	---	---

Typed element address

Move with element offsets

cutlass.Vector

One thread

1.0	2.0	3.0	4.0
v[0]	v[1]	v[2]	v[3]

Register values

Elementwise arithmetic

TMA load in CUTLASS Primitives

Inside the same CuTe pipeline API, CUTLASS Primitives replaces `cute.copy` with the instruction itself.

```
CuTe    cute.copy(tma_atom_a, tAgA[(None, ab_empty.count)], tAsA[(None, ab_empty.index)],  
               tma_bar_ptr=ab_empty.barrier)
```

CUTLASS Primitives

TMA producer · A tile

```
ab_empty = ab_producer.acquire_and_advance() # CuTe pipeline call  
1 if prims.elect_sync():  
    k_coord = k * _K_TILE  
    3 prims.cp_async_bulk_tensor_shared_cta_global(  
        smem_a_ptr + ab_empty.index * _M_TILE * _K_TILE,  
        a_ptr,  
        2 (k_coord, coord_m),  
        ab_empty.barrier,  
    )
```

1 Explicit control

You pick the thread and issue the copy yourself.

2 No CuTe layout knowledge

Plain coordinates and a pointer:
no copy atom, no partitions.

3 Full instruction set

The call is the PTX instruction
`cp.async.bulk.tensor`.

MMA in CUTLASS Primitives

Inside the CuTe pipeline, CUTLASS Primitives replaces `cute.gemm` and `tiled_mma.set` with the instruction itself.

```
CuTe for k_block_idx in cutlass.range_constexpr(num_k_blocks):  
    cute.gemm(tiled_mma, tCtAcc, tCrA[k_block_coord], tCrB[k_block_coord], tCtAcc)  
    tiled_mma.set(tcgen05.Field.ACCUMULATE, True)
```

CUTLASS Primitives

MMA Warp

```
ab_full = ab_consumer.wait_and_advance() # CuTe pipeline call  
:  
for kb in cutlass.range_constexpr(_K_BLOCKS):  
    inc = cutlass.Int64(_K_BLOCK_OFFSET_BYTES * kb)  
    1 if prims.elect_sync():  
        3 prims.tcgen05_mma(  
            prims.Tcgen05MMAKind.F16, prims.CTAGroup.CTA_1,  
            tmem_ptr,  
            2 desc_a_base.advance_start_address(inc),  
            desc_b_base.advance_start_address(inc),  
            _IDESC, scale_d,  
        )  
    scale_d = cutlass.Boolean(True)  
ab_full.release() # CuTe pipeline call
```

1 Explicit control

You write the thread guard and every operand of the call.

2 No CuTe layout knowledge

Descriptors and byte offsets:
no tiled MMA, no partitions.

3 Full instruction set

The call is the PTX instruction `tcgen05.mma`.

Compiler diagnostics for CUTLASS Python

Every call is typed, so the compiler knows what it means and can check how you combine them.

20

diagnostics for Primitives

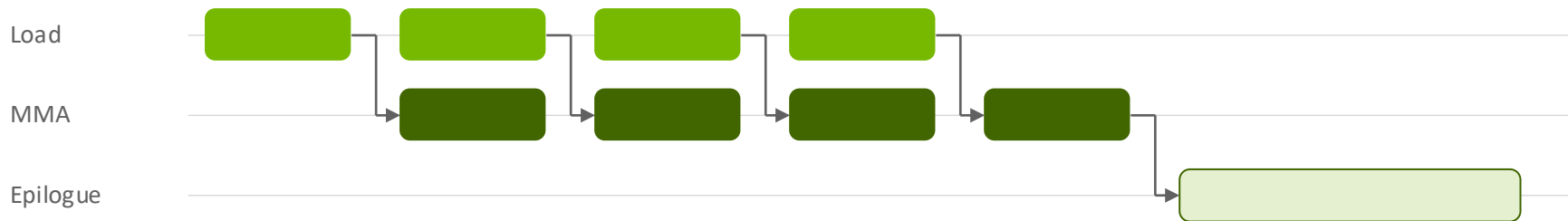
- 3 errors stop the build
- 14 warnings flag hangs and races
- 3 remarks report resources

7	Barrier protocol	1 error · 6 warnings
4	Elections + participation	2 errors · 2 warnings
1	TMA / bulk copy	1 warning
4	Tensor Core + TMEM	4 warnings
1	Memory ordering	1 warning
3	Resources (any kernel)	3 remarks

opt-in: CUTE_DSL_COMPILER_OPT="warnings{nvvm}" · resources: remarks{ptx}.

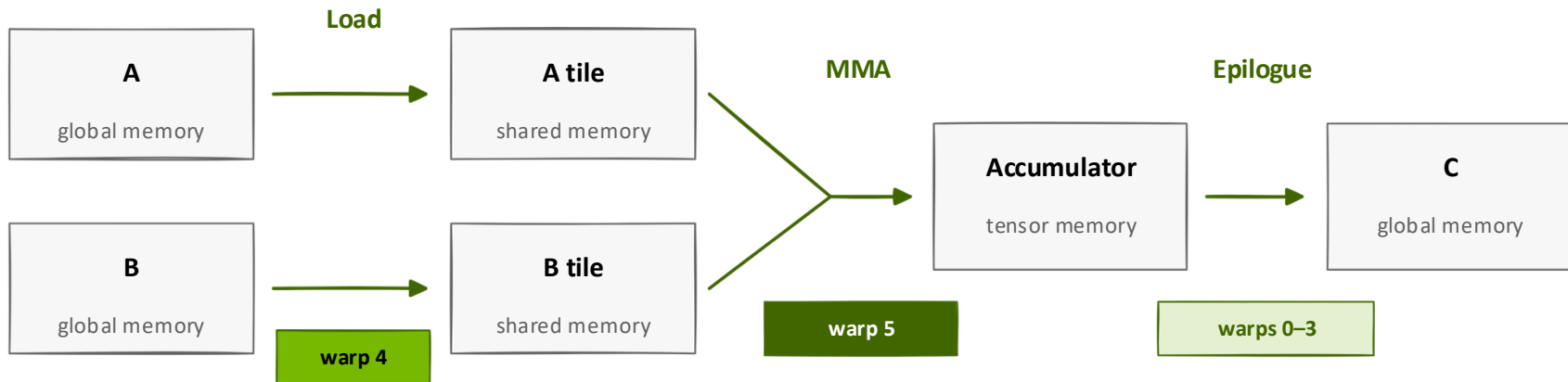
The Task Scheduling library

A compile-time framework for catching
deadlocks and memory races



Start with the GEMM dependency graph

Design the resources, parallel work, and warp roles together.



Which resources?

Resources and their dependencies

What can overlap?

Independent work and tile lifetimes

Which warps?

A warp range for each task

Then make the access order explicit.

A task runs its schedule on warps you choose

A Python library constructs the schedule; the GPU executes the generated kernel.

Resource

ab = staged SMEM A/B + TmaUmma protocol

EXAMPLE-DEFINED RESOURCE HELPER

```
ab = create_smem_ab_resource(  
    tma_a_desc, tma_b_desc)
```

Callbacks: tma_load / build_descriptors

Schedule

Ordered work + synchronization

Task

Schedule + assigned warp range

PYTHON · TASK CONSTRUCTION

```
load_task = Task(  
    src_resources=[gmem], dst_resources=[ab],  
    warp_idx=4, num_warps=1,  
    schedule=load(gmem, ab),  
    num_registers=40, name="LoadTask")
```

ONE CTA

Load

MMA

Epilogue

warp 4

warp 5

warps 0–3

Record → check → emit GPU work + sync

The schedules make both handoffs explicit

Commit → wait hands over data; release → acquire permits stage reuse.

LOAD · warp 4

@schedule

```
def load(gmem, ab):
    ab.init_load_state()
    with domain_loop(num_k_tiles):
        k, m, n = gmem.compute_coords()
        ab.try_acquire()
        ab.acquire()
        ab.tma_load(coord_k=k, coord_m=m,
                    coord_n=n)
        ab.commit()
```

MMA · warp 5

@schedule

```
def mma(ab, acc):
    ab.init_descriptors()
    acc.init_accumulator_state()
    acc.init_work_tile_state()
    acc.try_acquire()
    acc.acquire()
    with domain_loop(num_k_tiles):
        ab.try_wait()
        ab.wait()
        da, db = ab.build_descriptors()
        acc.mma(desc_a_base=da,
                desc_b_base=db)
        ab.release()
    acc.commit()
```

The schedules make both handoffs explicit

Commit → wait hands over data; release → acquire permits stage reuse.

LOAD · warp 4

@schedule

```
def load(gmem, ab):  
    ab.init_load_state()  
    with domain_loop(num_k_tiles):  
        k, m, n = gmem.compute_coords()  
        ab.try_acquire()  
        ab.acquire()  
        ab.tma_load(coord_k=k, coord_m=m,  
                    coord_n=n)  
        ab.commit() 1
```

MMA · warp 5

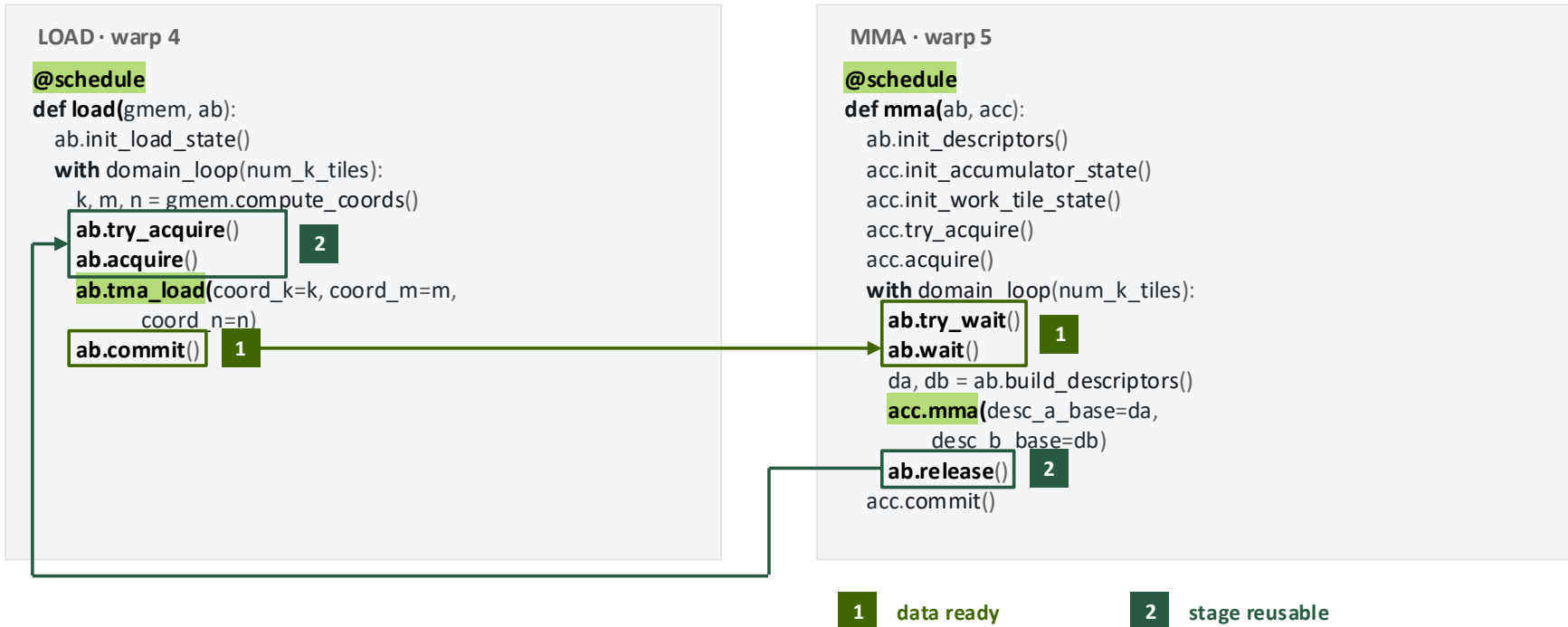
@schedule

```
def mma(ab, acc):  
    ab.init_descriptors()  
    acc.init_accumulator_state()  
    acc.init_work_tile_state()  
    acc.try_acquire()  
    acc.acquire()  
    with domain_loop(num_k_tiles):  
        ab.try_wait() 1  
        ab.wait()  
        da, db = ab.build_descriptors()  
        acc.mma(desc_a_base=da,  
                desc_b_base=db)  
        ab.release()  
    acc.commit()
```

1 data ready

The schedules make both handoffs explicit

Commit → wait hands over data; release → acquire permits stage reuse.



Check the protocol before the kernel runs

Deadlocks and memory races stop the build.

Concurrency issues are checked for in two ways:

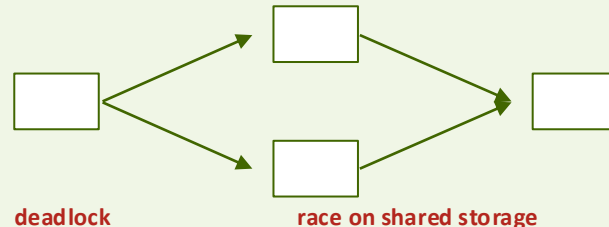
Static checks

One pass over the declared schedules

- Balanced commit / wait and release / acquire **deadlock**
- Work inside valid acquire / commit brackets **race**
- Work inside valid wait / release brackets **race**
- Pipeline participation + DMA ordering **deadlock · race**
- Register, SMEM and TMEM limits

Exhaustive check

Explore every enabled task ordering



Compile-time schedule verification

Exhaustive checker example: each task keeps its own order.

Task Consume

C.work

A.wait

A.work

A.release

Task Produce

D.work

A.acquire

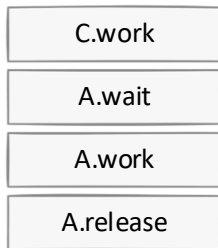
A.work

A.commit

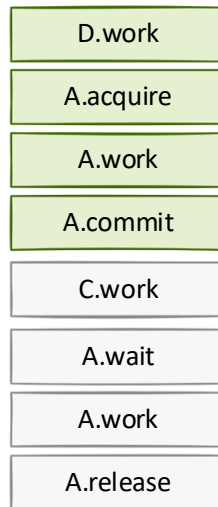
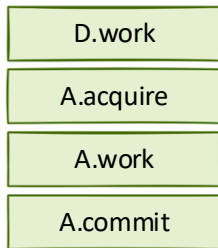
Compile-time schedule verification

One possible ordering executes the producer before the consumer.

Task Consume



Task Produce

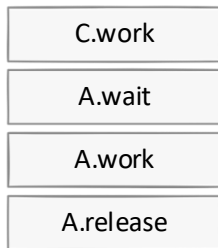


Time
26

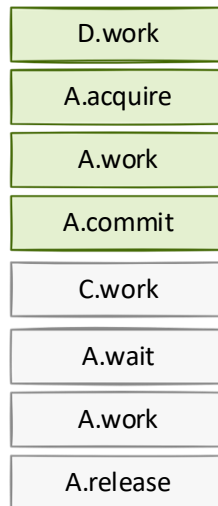
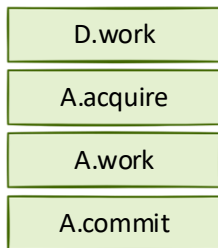
Compile-time schedule verification

Or the consumer starts with C.work before the producer runs.

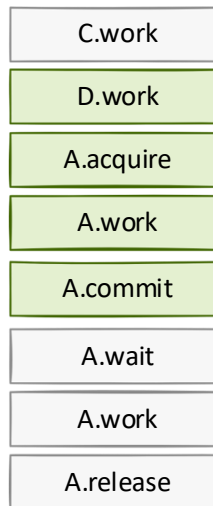
Task Consume



Task Produce



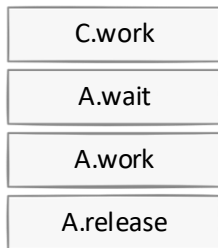
Or



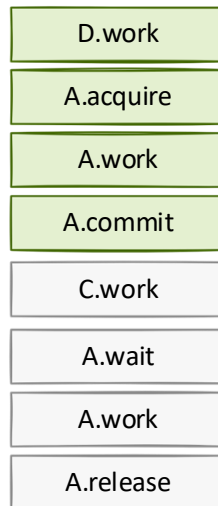
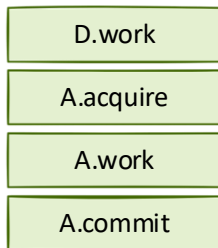
Compile-time schedule verification

Or C.work runs between D.work and A.acquire.

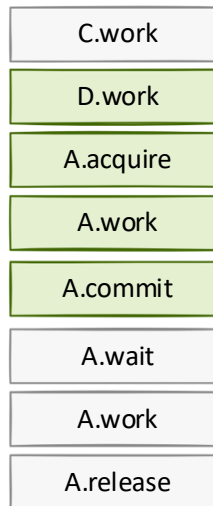
Task Consume



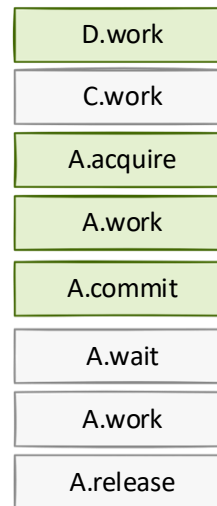
Task Produce



Or



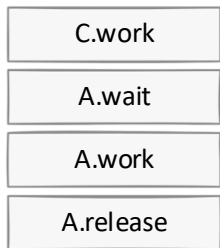
Or



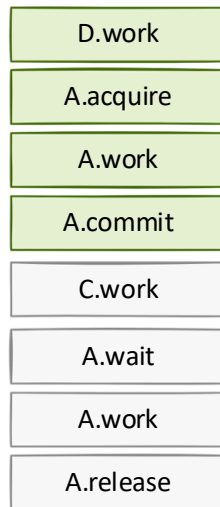
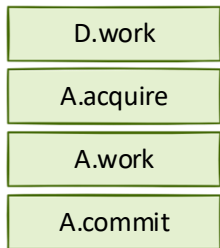
Compile-time schedule verification

Or C.work follows A.acquire, before the producer's A.work.

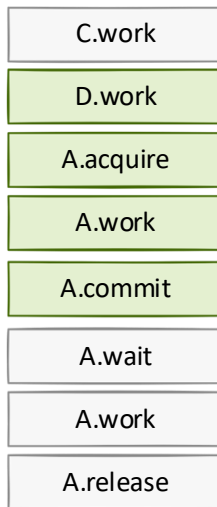
Task Consume



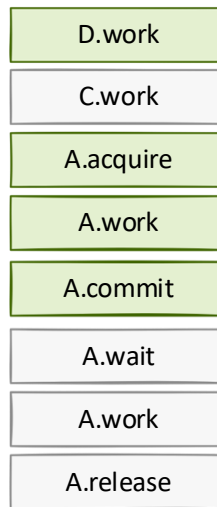
Task Produce



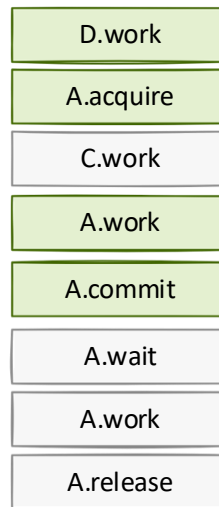
Or



Or



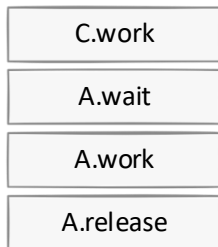
Or



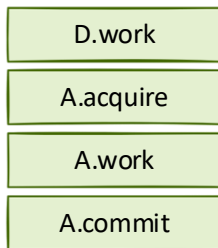
Compile-time schedule verification

C.work is still reading when A.work begins.

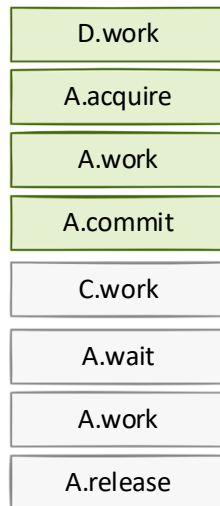
Task Consume



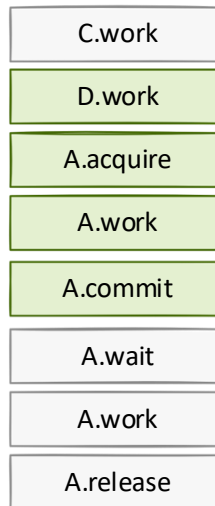
Task Produce



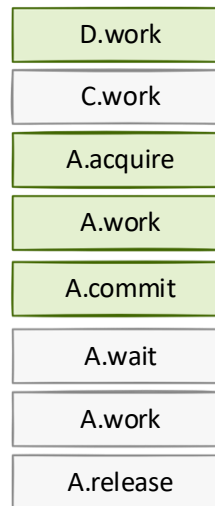
Work boxes show start order.



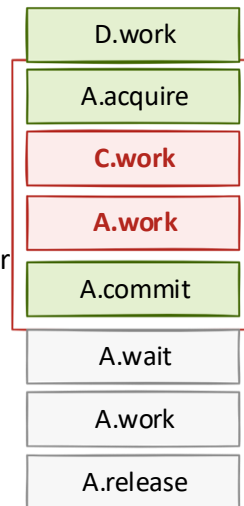
Or



Or



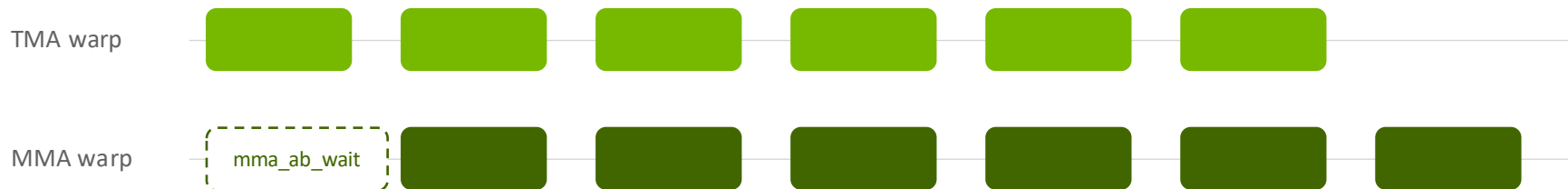
Or



Time
30

Device-side profiling

See where each warp spends its time.



Device-side profiling

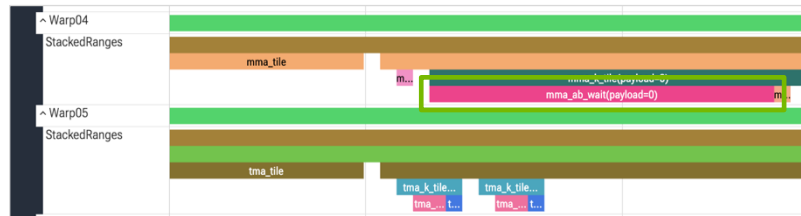
Warp-level ranges and markers inside a CUTLASS Python kernel

MMA mainloop

```
import cutlass.cute.experimental.iket as profiler

@cute.kernel
def kernel(...):
    if warp_idx == mma_warp_id and is_leader_cta:
        for k in range(num_k_tiles):
            profiler.range_push("mma_k_tile", k)
            profiler.range_push("mma_ab_wait", k)
            h = ab_consumer.wait_and_advance()
            profiler.range_pop()
            profiler.range_push("mma_issue", k)
            cute.gemm(...)
            profiler.range_pop()
            h.release()
            profiler.range_pop()
```

Perfetto timeline



mma_ab_wait(payload=0) MMA warp waits for input tile 0

Main APIs

mark(name[, payload])

One named timestamp for a point event

range_push(name[, payload]) / range_pop()

Nested scopes; pop closes the most recent range

token = range_start(name[, payload])

range_end(token[, payload])

Explicit pairing across a wait or loop iteration

Payloads attach scalars; start/end must match in presence and type.

Cleaner code. More control. Fewer concurrency bugs.

New programming models, device-side profiling, and a task-based Python library.

CuTe DSL extensions	Clean code, still SOL
CUTLASS Primitives	Direct instructions, more control
Task Scheduler	Task-based library, simpler async coordination
In kernel Profiler	Profiler code regions inside kernel

CuTe DSL extensions kernels

GEMM · grouped GEMM · attention

NVIDIA / cutlass



CUTLASS Primitives + TS kernels

Batched / grouped GEMM for MoE

flashinfer-ai / flashinfer

