CuTe DSL Introduce

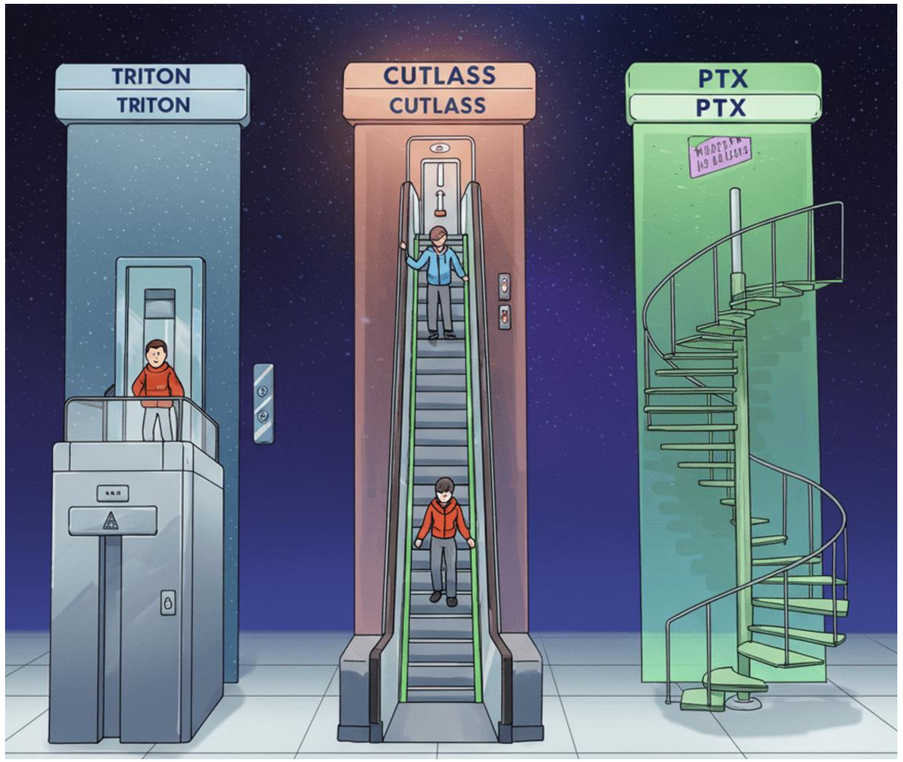
Vicki Wang: 2025-12-06

Why CUTLASS?

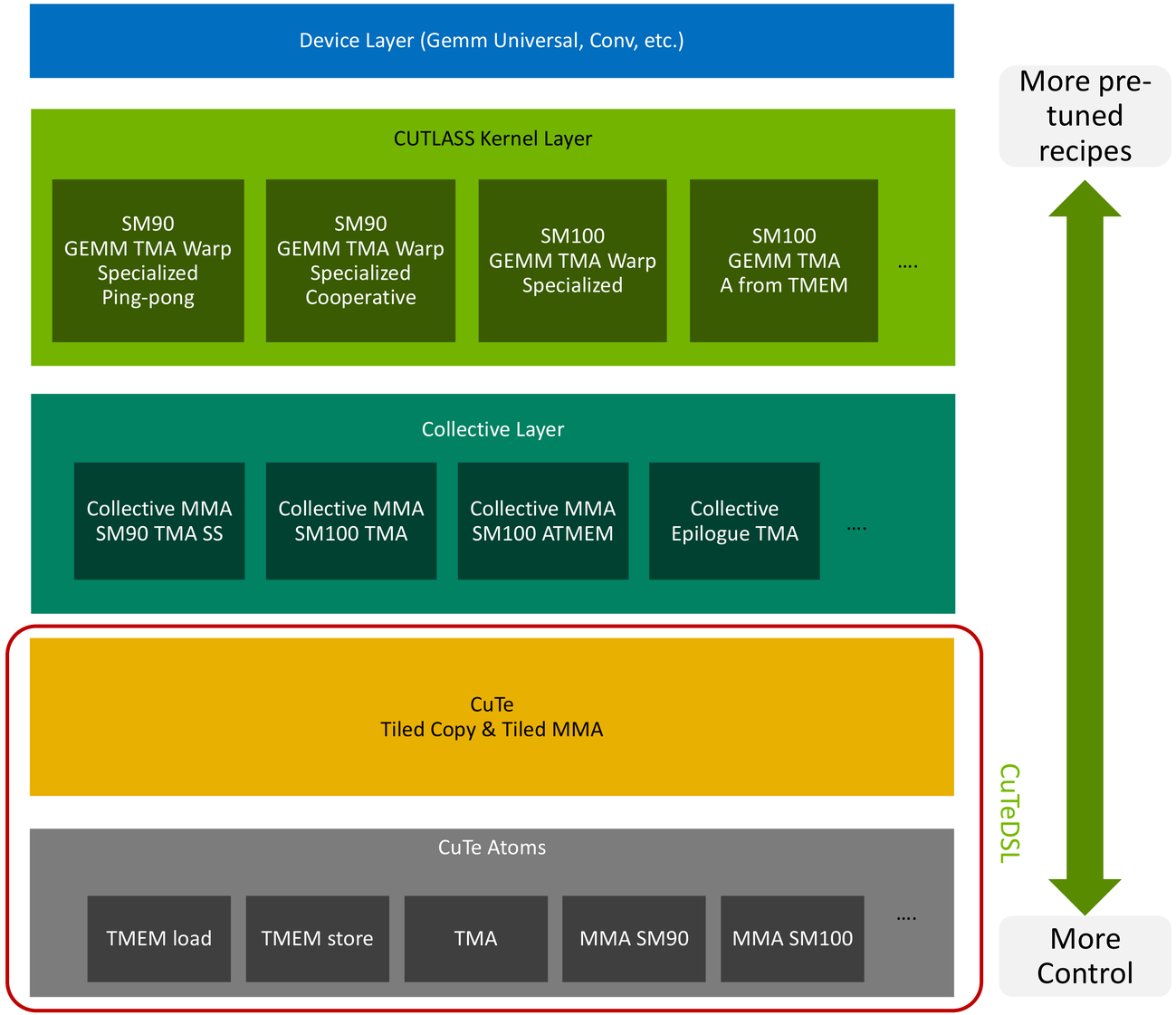
Enabling innovation for SOL performance

- High-level generators leveraging compilers are popular
  - Hide and automate a lot of details
  - Get excellent performance on common use cases, but…
  - Algorithmic innovations require lower-level ***abstractions***
  - Advanced features like PDL require fine grain control

- With CUTLASS
  - Available on day 0 with full control!
  - Expressive abstractions for performance in all cases
  - Modular and extensible design robust throughout GPU generations



CUTLASS in Python



- CUTLASS: A set of useful abstractions for productivity and performance at all scopes and scales
  - Open source https://github.com/NVIDIA/cutlass
  - Multiple entry points depending on your needs
- CuTe DSL
  - This first release makes available a ***mature*** low-level tensor programming model, giving access to tensor cores with full control.
  - Empower AI researchers and engineers to facilitate rapid prototyping
  - GTC'25

CUTLASS in Python

How will you get started?

```
pip install nvidia-cutlass-dsl
```

```python
import cutlass
import cutlass.cute as cute

@cute.kernel
def kernel():
  tidx, _, _ = cutlass.nvvm.thread_idx()
  if tidx == 0:
    cute.print_("Hello world")

@cute.jit
def host():
  kernel().launch(
    grid=(1, 1, 1), block=(32, 1, 1)))

host()
```

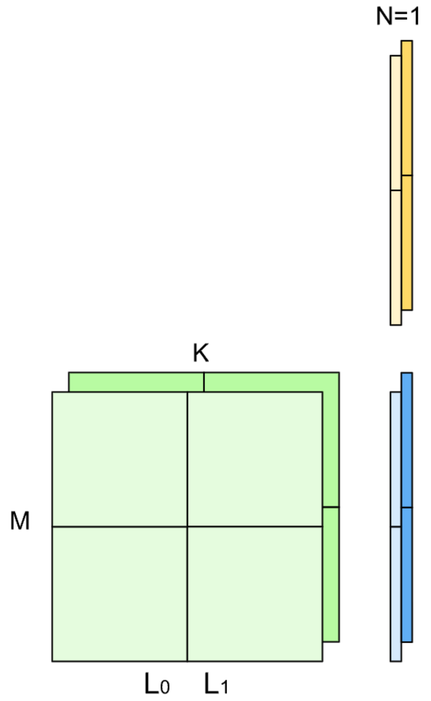
```
python3 hello_world.py
```

Writing GEMV in Python

@cute.jit / @cute.kernel

```python
@cute.jit
def my_kernel(
    a_ptr: cute.Pointer,
    b_ptr: cute.Pointer,
    sfa_ptr: cute.Pointer,
    sfb_ptr: cute.Pointer,
    c_ptr: cute.Pointer,
    problem_size: tuple,
):
    # Unpack the problem size.
    m, _, k, l = problem_size

    # Create the A Tensor with (M, K, L) : (K, 1, L) layout.
    a_tensor = cute.make_tensor(
        a_ptr, cute.make_layout((m, cute.assume(k, 32), l), stride=(cute.assume(k, 32), 1, cute.assume(m * k, 32)),)
    )
    # Create the scale A Tensor with (((32, 4), REST_M), ((SF_K, 4), REST_K), (1, REST_L)) shape.
    # Details see https://docs.nvidia.com/cuda/cublas/index.html?highlight=fp4#d-block-scaling-factors-layout
    sfa_layout = blockscaled_utils.tile_atom_to_shape_SF(a_tensor.shape, sf_vec_size)
    # ... manage other parameters

    # Compute grid shape, let each block handle 128 output elements
    grid = (cute.ceil_div(c_tensor.shape[0], 128), 1, c_tensor.shape[2])

    # Launch the CUDA kernel
    kernel(a_tensor, b_tensor, sfa_tensor, sfb_tensor, c_tensor).launch(
        grid=grid,
        block=[threads_per_cta, 1, 1],
        cluster=(1, 1, 1),
    )
    return
```

- cute.jit is the entry point from host side
  - Prepare parameters for the kernel
  - Called by other DSL function



Native Implementation: Load data from global buffer to register, do FMA and store results to global memory

Writing GEMV in Python

@cute.jit / @cute.kernel

```python
@cute.kernel
def kernel(
    mA_mkl: cute.Tensor,
    mB_nkl: cute.Tensor,
    mSFA_mkl: cute.Tensor,
    mSFB_nkl: cute.Tensor,
    mC_mnl: cute.Tensor,
):
    # Get CUDA block and thread indices
    bidx, bidy, bidz = cute.arch.block_idx()
    tidx, _, _ = cute.arch.thread_idx()

    # Do other computation ...
```

- cute.kernel is the GPU kernel
  - Equal to __global__ void kernel(...)

Caller-Callee Compatibility

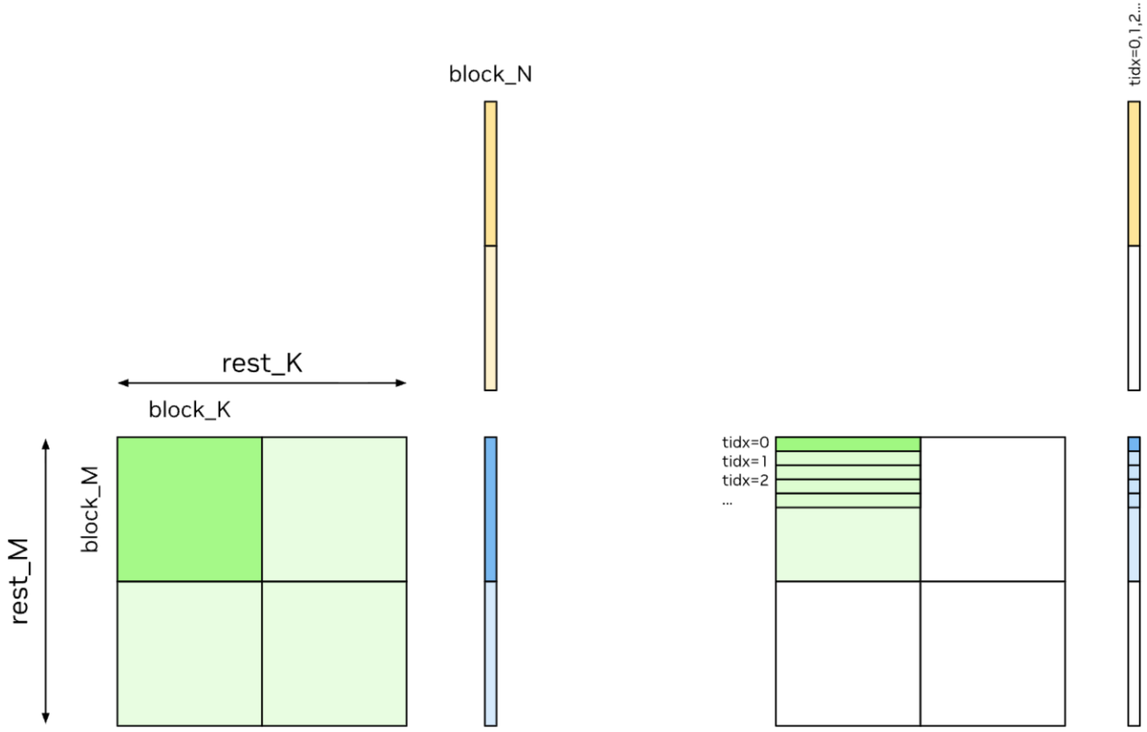
Caller	Callee	Allowed	Compilation/Runtime
Python function	`@jit`	✅	DSL runtime
Python function	`@kernel`	❌	N/A (error raised)
`@jit`	`@jit`	✅	Compile-time call, inlined
`@jit`	Python function	✅	Compile-time call, inlined, tracing-only
`@jit`	`@kernel`	✅	Dynamic call via GPU driver or runtime
`@kernel`	`@jit`	✅	Compile-time call, inlined
`@kernel`	Python function	✅	Compile-time call, inlined, tracing-only
`@kernel`	`@kernel`	❌	N/A (error raised)

Writing GEMV in Python

CuTe Basic Layout

```
# Partition matrix A into local tiles (shape: [block_M, block_K, rest_M, rest_K, rest_L])
gA_mkl = cute.local_tile(
    mA_mkl, cute.slice_(mma_tiler_mnk, (None, 0, None)), (None, None, None)
)
# Each block handles a tile of matrix A
# Each thread handles one row inside the tile
tAgA = gA_mkl[tidx, None, bidx, None, bidz]
```

- Do partition to split buffer before doing cute.copy and cute.gemm
- Using slice or [] to index the buffer for each block, warp, or threads



Block view of partition and slice per Batch

Thread view of partition and slice per Block

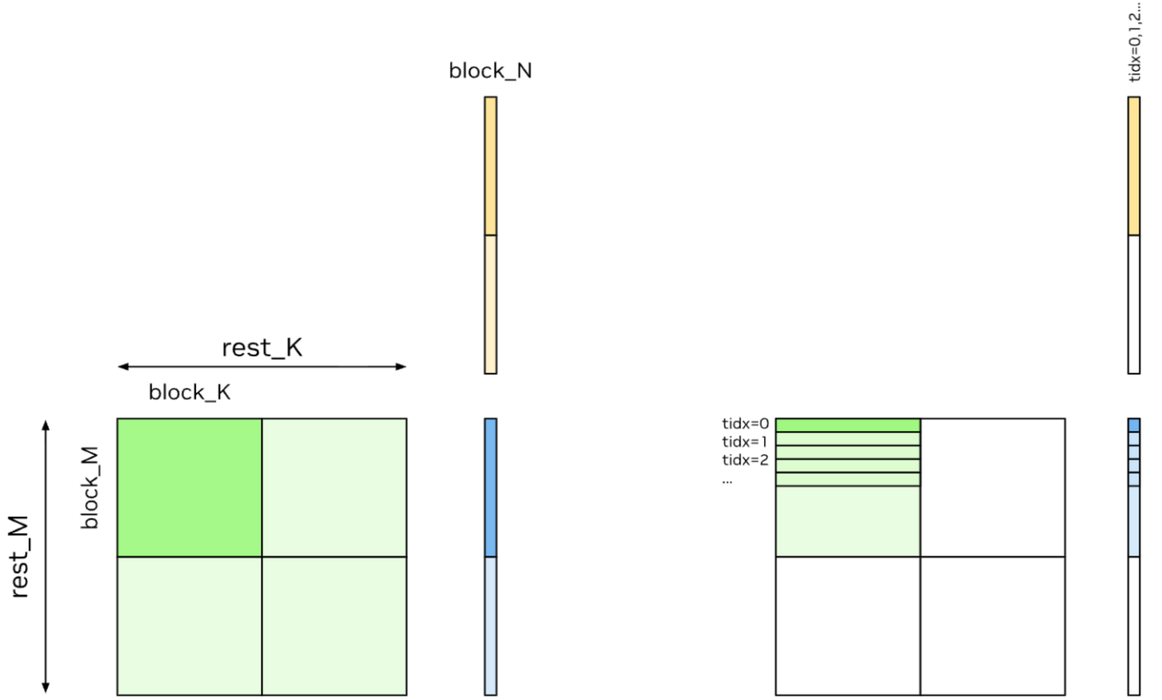
Writing GEMV in Python

Tensor and Tensor SSA

```
res = cute.zeros_like(tCgC, cutlass.Float32)
# Get the number of k tiles of the reduction loop
k_tile_cnt = gA_mkl.layout[3].shape
for k_tile in range(k_tile_cnt):
    # Slice data on k_dimension
    tAgA = gA_mkl[tidx, None, bidx, k_tile, bidz]
    tBgB = gB_nkl[0, None, bidy, k_tile, bidz]
    # Take tAgA/tBgB layotus and create Rmem Tensor
    # tArA.layout: (64): (1)
    # tBrB.layout: (64): (1)
    tArA = cute.make_rmem_tensor_like(tAgA, cutlass.Float32)
    tBrB = cute.make_rmem_tensor_like(tBgB, cutlass.Float32)
    # Load NVFP4 into register (TensorSSA)
    a_val_nvfp4 = tAgA.load() # vector<64xf4E2M1FN>
    b_val_nvfp4 = tBgB.load() # vector<64xf4E2M1FN>
    # Convert float32 for computation (FFMA)
    a_val = a_val_nvfp4.to(cutlass.Float32) # vector<64xf32>
    b_val = b_val_nvfp4.to(cutlass.Float32) # vector<64xf32>
    # Store the converted values to Rmem CuTe tensors
    tArA.store(a_val)
    tBrB.store(b_val)

    # Iterate over SF vector tiles and compute the scale&matmul accumulation
    for i in cutlass.range_constexpr(mma_tiler_mnk[2]):
        res += tArA[i] * tBrB[i]
```

The code is simplified to show key APIs and concepts

- Tensor is the composition of an iterator and a layout
- TensorSSA thread local data modeling for CuTe Tensor in value semantics and immutable
  - Vector based with nested CuTe shape support
  - Load tensor elements as vector / store vector data into tensor
  - Operator overloading for vectorized operations



Block view of partition and slice per Batch

Thread view of partition and slice per Block

NVIDIA

Writing GEMV in Python

Python to Python code generation: for/if/while

```
res = cute.zeros_like(tCgC, cutlass.Float32)
# Get the number of k tiles of the reduction loop
k_tile_cnt = gA_mkl.layout[3].shape
for k_tile in range(k_tile_cnt):
    # Slice data on k_dimension
    tAgA = gA_mkl[tidx, None, bidx, k_tile, bidz]
    tBgB = gB_nkl[0, None, bidy, k_tile, bidz]
    # Take tAgA/tBgB layotus and create Rmem Tensor
    # tArA.layout: (64): (1)
    # tBrB.layout: (64): (1)
    tArA = cute.make_rmem_tensor_like(tAgA, cutlass.Float32)
    tBrB = cute.make_rmem_tensor_like(tBgB, cutlass.Float32)
    # Load NVFP4 into register (TensorSSA)
    a_val_nvfp4 = tAgA.load() # vector<64xf4E2M1FN>
    b_val_nvfp4 = tBgB.load() # vector<64xf4E2M1FN>
    # Convert float32 for computation (FFMA)
    a_val = a_val_nvfp4.to(cutlass.Float32) # vector<64xf32>
    b_val = b_val_nvfp4.to(cutlass.Float32) # vector<64xf32>
    # Store the converted values to Rmem CuTe tensors
    tArA.store(a_val)
    tBrB.store(b_val)

    # Iterate over SF vector tiles and compute the scale&matmul accumulation
    for i in cutlass.range_constexpr(mma_tiler_mnk[2]):
        res += tArA[i] * tBrB[i]
```

- Control flow
  - CuTe DSL navigates through Python's AST and transforms each control-flow construct it encounters into a structured intermediate representation (IR). This allows users to write standard Python loops and branches, while the compiler determines whether to evaluate it at compile time if it's a native Python control flow, or to generate intermediate representation (IR) if the control flow is dynamic.
    - **cutlass.rang**: supports advanced unrolling and pipelining control
    - Software Pipeline example

Control Flow Process

Control Flow	AST Rewrite	Python Evaluation	Scf Node Generation
if cutlass.const_expr()	❌	✅	❌
if pred	✅	❌	✅
while cutlass.const_expr()	❌	✅	❌
while pred	✅	❌	✅
for i in cutlass.range_constexpr()	❌	✅	❌
for i in range()	✅	❌	✅
for i in cutlass.range()	✅	❌	✅

Writing GEMV in Python

Software Pipeline

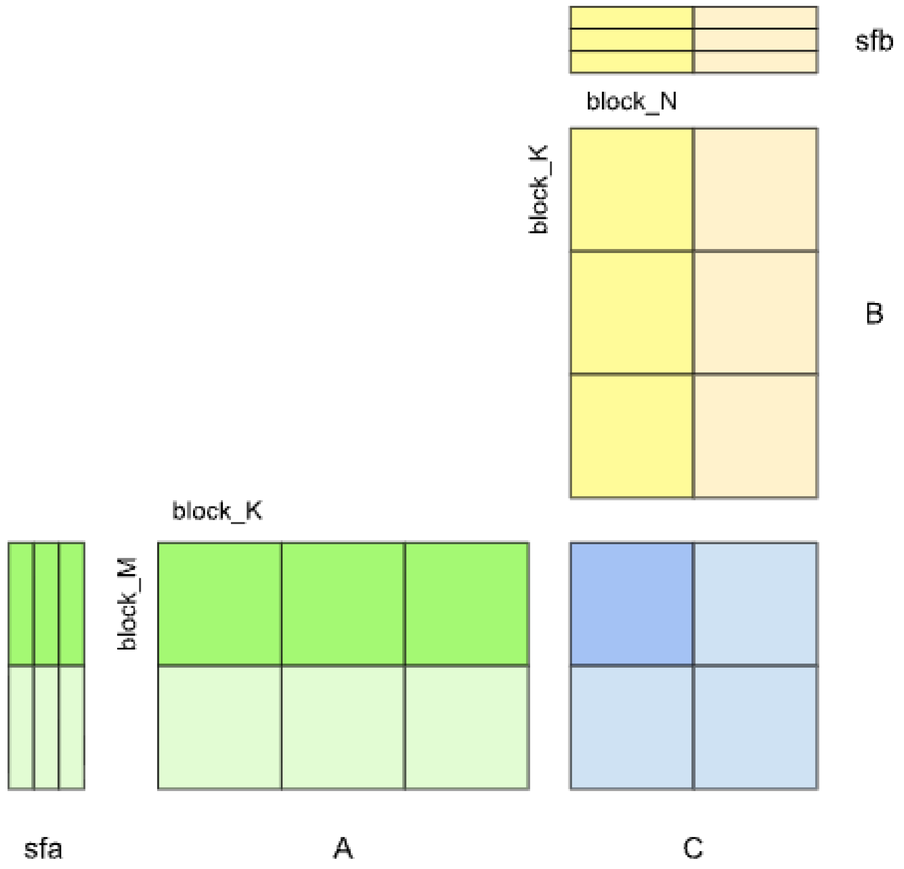
```
# ((atom_v, rest_v), RestK)
tBgB = tBgB[(None, mma_tile_coord_mnl[1], None, mma_tile_coord_mnl[2])]

if warp_idx == 0:
    # ////////////////////////////////////////////////////////////////////////////
    #  MAINLOOP
    # ////////////////////////////////////////////////////////////////////////////
    for k_tile in cutlass.range(
        k_tile_cnt, prefetch_stages=self.num_ab_stage - 2
    ):
        # wait for AB buffer empty
        producer_handle = ab_producer.acquire_and_advance()

        # TMA load A/B
        cute.copy(…
        )
        cute.copy(…
        )

        if is_leader_cta:
            #  Wait for AB buffer full
            consumer_handle = ab_consumer.wait_and_advance()

            # tCtAcc += tCrA * tCrB
            num_kblks = cute.size(tCrA, mode=[2])
            for kblk_idx in cutlass.range(num_kblks, unroll_full=True):
                kblk_crd = (None, None, kblk_idx, consumer_handle.index)

                cute.gemm(…
                )
                # Enable accumulate on tCtAcc after first kblock
                tiled_mma.set(tcgen05.Field.ACCUMULATE, True)

            # Async arrive AB buffer empty
            consumer_handle.release()

    # Async arrive accumulator buffer full
    if is_leader_cta:
        acc_pipeline.producer_commit(acc_producer_state)
```

```
tBgB = tBgB[(None, mma_tile_coord_mnl[1], None, mma_tile_coord_mnl[2])]

#
# Pipelining TMA load A/B and MMA mainloop
#
prefetch_k_tile_cnt = cutlass.min(self.num_ab_stage - 2, k_tile_cnt)
if warp_idx == 0:
    #
    # Prefetch TMA load A/B
    #
    for k_tile_idx in cutlass.range(prefetch_k_tile_cnt, unroll=1):
        # Conditionally wait for AB buffer empty
        producer_handle = ab_producer.acquire_and_advance()

        # TMA load A/B
        cute.copy(…
        )
        cute.copy(…
        )

    peek_ab_full_status = cutlass.Boolean(False)
    if is_leader_cta:
        peek_ab_full_status = ab_consumer.try_wait()

    # Peek (try_wait) AB buffer empty for k_tile = prefetch_k_tile_cnt + k_tile + 1
    peek_ab_empty_status = ab_producer.try_acquire()

    #
    # MMA mainloop
    #
    for k_tile_idx in cutlass.range(k_tile_cnt):
        # Conditionally wait for AB buffer empty
        if k_tile_idx < k_tile_cnt - prefetch_k_tile_cnt:
            producer_handle = ab_producer.acquire_and_advance(
                peek_ab_empty_status
            )

            # TMA load A/B
            cute.copy(…
            )
            cute.copy(…
            )

        if is_leader_cta:
            # Conditionally wait for AB buffer full
            consumer_handle = ab_consumer.wait_and_advance(peek_ab_full_status)

            # tCtAcc += tCrA * tCrB
            num_kblks = cute.size(tCrA, mode=[2])
            for kblk_idx in cutlass.range(num_kblks, unroll_full=True):
                kblk_crd = (None, None, kblk_idx, consumer_handle.index)

                cute.gemm(…
                )
                # Enable accumulate on tCtAcc after first kblock
                tiled_mma.set(tcgen05.Field.ACCUMULATE, True)

            # Async arrive AB buffer empty
            consumer_handle.release()

        # Peek (try_wait) AB buffer empty for k_tile = prefetch_k_tile_cnt + k_tile + 1
        peek_ab_empty_status = ab_producer.try_acquire()

        # Peek (try_wait) AB buffer full for k_tile = k_tile + 1
        peek_ab_full_status = ab_consumer.try_wait()

    # Async arrive accumulator buffer full
    if is_leader_cta:
        acc_pipeline.producer_commit(acc_producer_state)
```

Writing GEMM in Python

Use advance GPU feature: Tensor core

- Using TMA Load to load A, B, scaleA, scaleB tensors from global memory to shared memory
- Using S2T Load to load scaleA, scaleB tensors from shared momory to tensor memory
- Using tensor core instruction to do math computation
- Using T2R Load to load accumulation result from tensor memory to register
- Store results from register to global C tensor



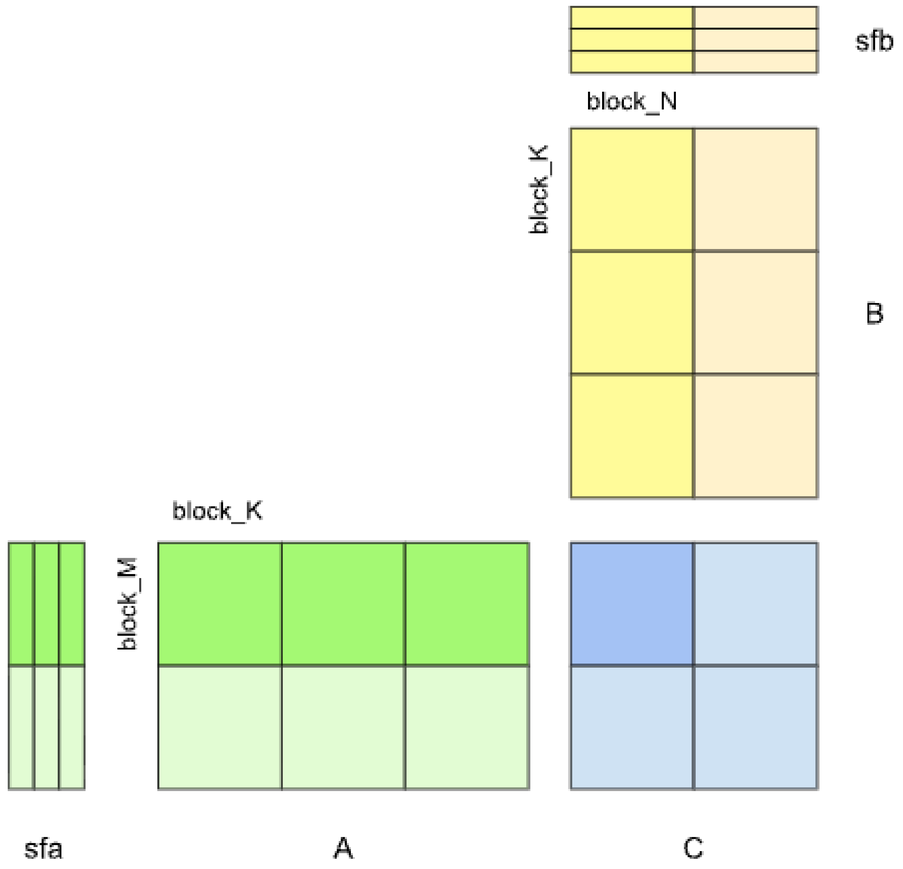
Writing GEMM in Python

Use advance GPU feature: TMA load

```python
# Setup TMA for A
tma_atom_a, tma_tensor_a = cute.nvgpu.make_tiled_tma_atom_A(
    cpasync.CopyBulkTensorTileG2SOp(tcgen05.CtaGroup.ONE),
    a_tensor,      # a_tensor.layout:(M, K, L): (K, 1, M * K)
    a_smem_layout, # a_smem_layout: S<3,4,3> o 0 o ((128,64),1,4):((256,1),0,64)
    mma_tiler_mnk, # (128, 128, 256)
    tiled_mma,     # mma_op: MmaMXF4NVF4Op(sf_dtype, (128, 128, 64),
                   #           CtaGroup.ONE, OperandSource.SMEM)
)
```

Host codes to setup TMA atom and TMA coord tensor (in @cute.jit)

```python
# Partition global tensor via mma_tiler shape(128, 128, 256)
# (bM, bK, RestM, RestK, RestL)
gA_mkl = cute.local_tile(
    mA_mkl, cute.slice_(self.mma_tiler, (None, 0, None)), (None, None, None)
)
# Partition global tensor via MMA op shape(128, 128, 64)
# (MMA, MMA_M, MMA_K, RestM, RestK, RestL)
tCgA = thr_mma.partition_A(gA_mkl)
# Partition global/shared tensor to fit TMA load
# ((atom_v, rest_v), STAGE)
# ((atom_v, rest_v), RestM, RestK, RestL)
tAsA, tAgA = cpasync.tma_partition(
    tma_atom_a,
    0,                          # Used for CGA config
    cute.make_layout(1),        # Used for CGA config
    cute.group_modes(sA, 0, 3),   # sA.shape:((128,64),1,4,1)
                                  #          (MMA, MMA_M, MMA_K, STAGE)
    cute.group_modes(tCgA, 0, 3), # tCgA.shape:((128,64),1,4,1,1,1)
                                  #            (MMA, MMA_M, MMA_K, RestM, RestK, RestL)
)
# Execute k_tile loop
for k_tile in range(k_tile_cnt):
    #  TMA Load A Tensor
    cute.copy(
        tma_atom_a,
        tAgA[(None, k_tile)],
        tAsA[(None, ab_empty.index)],
        tma_bar_ptr=ab_empty.barrier, # TMA barrier pointer
    )
```



The code is simplified to show key APIs and concepts

Writing GEMM in Python

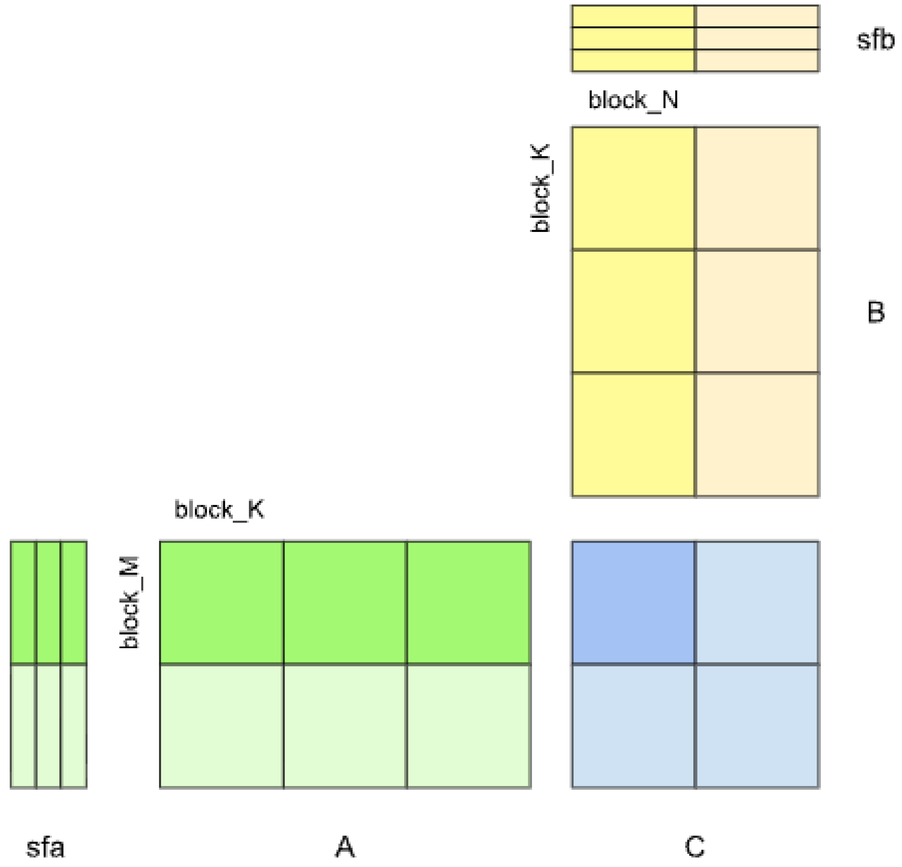
Use advance GPU feature: TMA load

```
# Setup TMA for A
tma_atom_a, tma_tensor_a = cute.nvgpu.make_tiled_tma_atom_A(
    cpasync.CopyBulkTensorTileG2SOp(tcgen05.CtaGroup.ONE),
    a_tensor,      # a_tensor.layout:(M, K, L): (K, 1, M * K)
    a_smem_layout, # a_smem_layout: S<3,4,3> o 0 o ((128,64),1,4):((256,1),0,64)
    mma_tiler_mnk, # (128, 128, 256)
    tiled_mma,     # mma_op: MmaMXF4NVF4Op(sf_dtype, (128, 128, 64),
                   #           CtaGroup.ONE, OperandSource.SMEM)
)
```

Host codes to setup TMA atom and TMA coord tensor (in @cute.jit)

```
# Partition global tensor via mma_tiler shape(128, 128, 256)
# (bM, bK, RestM, RestK, RestL)
gA_mkl = cute.local_tile(
    mA_mkl, cute.slice_(self.mma_tiler, (None, 0, None)), (None, None, None)
)
# Partition global tensor via MMA op shape(128, 128, 64)
# (MMA, MMA_M, MMA_K, RestM, RestK, RestL)
tCgA = thr_mma.partition_A(gA_mkl)
# Partition global/shared tensor to fit TMA load
# ((atom_v, rest_v), STAGE)
# ((atom_v, rest_v), RestM, RestK, RestL)
tAsA, tAgA = cpasync.tma_partition(
    tma_atom_a,
    0,                          # Used for CGA config
    cute.make_layout(1),        # Used for CGA config
    cute.group_modes(sA, 0, 3),   # sA.shape:((128,64),1,4,1)
                                #          (MMA, MMA_M, MMA_K, STAGE)
    cute.group_modes(tCgA, 0, 3), # tCgA.shape:((128,64),1,4,1,1,1)
                                #            (MMA, MMA_M, MMA_K, RestM, RestK, RestL)
)
# Execute k_tile loop
for k_tile in range(k_tile_cnt):
    #  TMA Load A Tensor
    cute.copy(
        tma_atom_a,
        tAgA[(None, k_tile)],
        tAsA[(None, ab_empty.index)],
        tma_bar_ptr=ab_empty.barrier, # TMA barrier pointer
    )
```

GPU codes: Partition Gmem and Smem buffer to prepare inputs for TMA load instruction



The code is simplified to show key APIs and concepts

Writing GEMM in Python

Use advance GPU feature: TMA load

```
# Setup TMA for A
tma_atom_a, tma_tensor_a = cute.nvgpu.make_tiled_tma_atom_A(
    cpasync.CopyBulkTensorTileG2SOp(tcgen05.CtaGroup.ONE),
    a_tensor,      # a_tensor.layout:(M, K, L): (K, 1, M * K)
    a_smem_layout, # a_smem_layout: S<3,4,3> o 0 o ((128,64),1,4):((256,1),0,64)
    mma_tiler_mnk, # (128, 128, 256)
    tiled_mma,     # mma_op: MmaMXF4NVF4Op(sf_dtype, (128, 128, 64),
                   #           CtaGroup.ONE, OperandSource.SMEM)
)
```

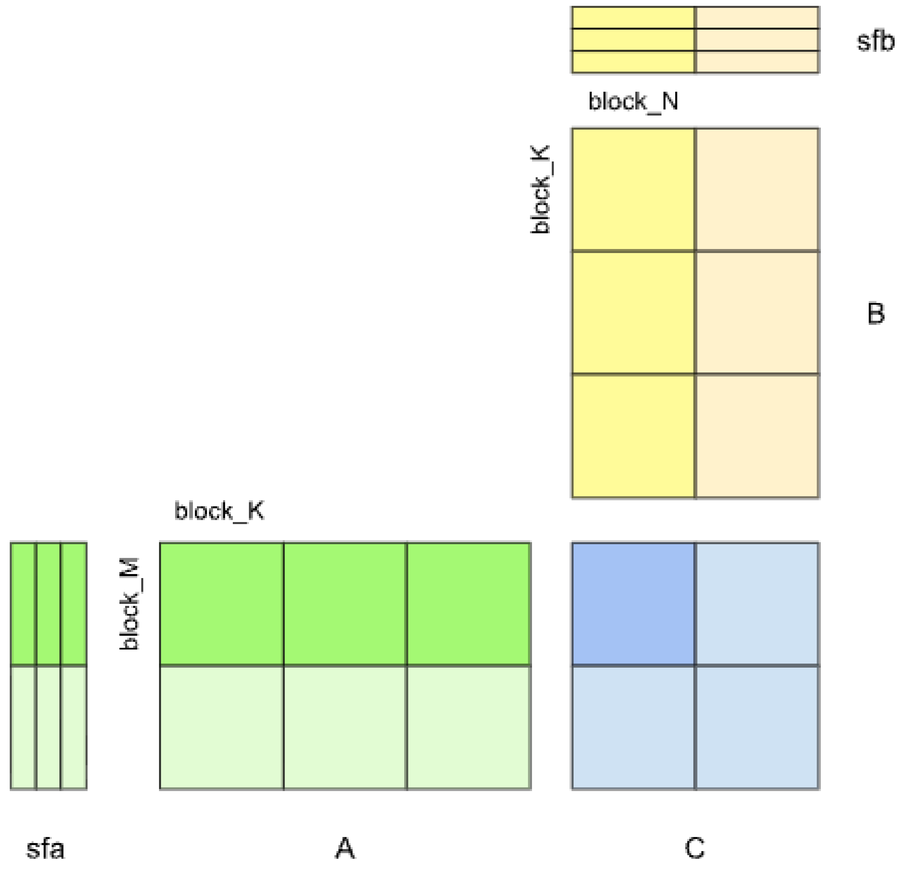
Host codes to setup TMA atom and TMA coord tensor (in @cute.jit)

```
# Partition global tensor via mma_tiler shape(128, 128, 256)
# (bM, bK, RestM, RestK, RestL)
gA_mkl = cute.local_tile(
    mA_mkl, cute.slice_(self.mma_tiler, (None, 0, None)), (None, None, None)
)
# Partition global tensor via MMA op shape(128, 128, 64)
# (MMA, MMA_M, MMA_K, RestM, RestK, RestL)
tCgA = thr_mma.partition_A(gA_mkl)
# Partition global/shared tensor to fit TMA load
# ((atom_v, rest_v), STAGE)
# ((atom_v, rest_v), RestM, RestK, RestL)
tAsA, tAgA = cpasync.tma_partition(
    tma_atom_a,
    0,                          # Used for CGA config
    cute.make_layout(1),        # Used for CGA config
    cute.group_modes(sA, 0, 3),   # sA.shape:((128,64),1,4,1)
                                  #          (MMA, MMA_M, MMA_K, STAGE)
    cute.group_modes(tCgA, 0, 3), # tCgA.shape:((128,64),1,4,1,1,1)
                                  #            (MMA, MMA_M, MMA_K, RestM, RestK, RestL)
)
```

GPU codes: Partition Gmem and Smem buffer to prepare inputs for TMA load instruction

```
# Execute k_tile loop
for k_tile in range(k_tile_cnt):
    #  TMA Load A Tensor
    cute.copy(
        tma_atom_a,
        tAgA[(None, k_tile)],
        tAsA[(None, ab_empty.index)],
        tma_bar_ptr=ab_empty.barrier, # TMA barrier pointer
    )
```

GPU codes: Call TMA load instruction For each k_tile



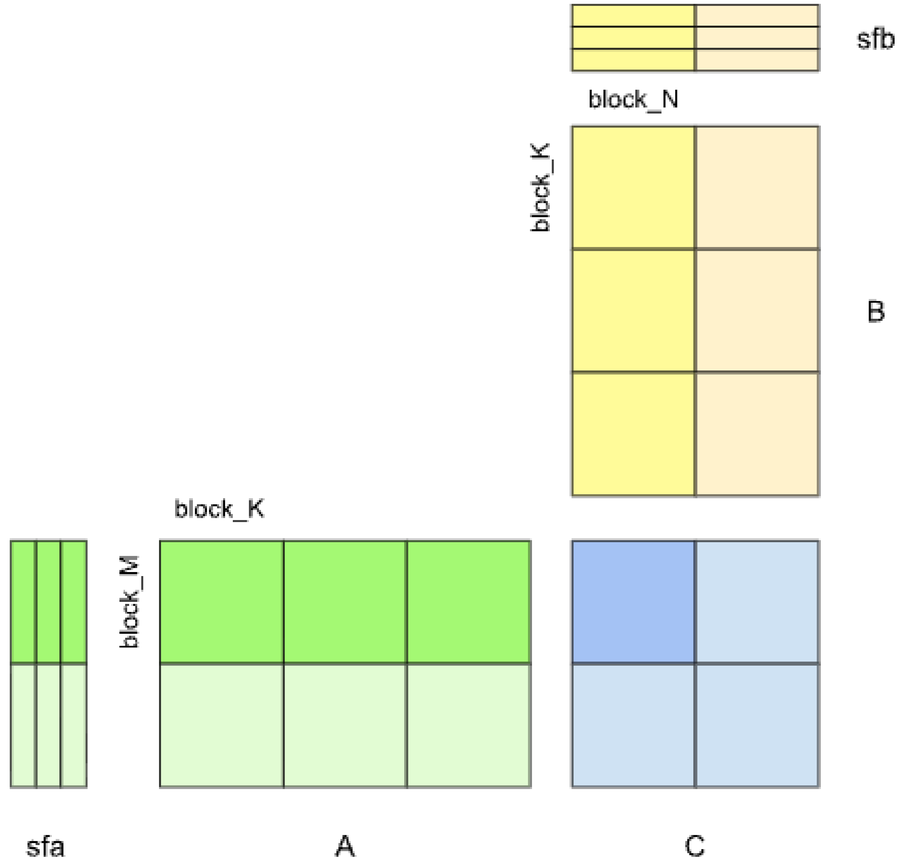
The code is simplified to show key APIs and concepts

Writing GEMM in Python

Use advance GPU feature: S2T copy

```
#
# Partition for S2T copy of SFA/SFB
#
# Make S2T CopyAtom
copy_atom_s2t = cute.make_copy_atom(
    tcgen05.Cp4x32x128bOp(tcgen05.CtaGroup.ONE),
    sf_dtype,
)
tiled_copy_s2t_sfa = tcgen05.make_s2t_copy(copy_atom_s2t, tCtSFA)
thr_copy_s2t_sfa = tiled_copy_s2t_sfa.get_slice(0)

# ((ATOM_V, REST_V), Rest_Tiler, MMA_MN, MMA_K, STAGE)
tCsSFA_s2t = thr_copy_s2t_sfa.partition_S(sSFA)
# ((ATOM_V, REST_V), Rest_Tiler, MMA_MN, MMA_K)
tCtSFA_s2t = thr_copy_s2t_sfa.partition_D(tCtSFA)

for k_tile in cutlass.range(k_tile_cnt):
    # ... copy data from global to shared memory

    #  Copy SFA/SFB to tmem
    s2t_stage_coord = (None, None, None, None, ab_full.index)
    tCsSFA_s2t_staged = tCsSFA_s2t[s2t_stage_coord]
    cute.copy(
        tiled_copy_s2t_sfa,
        tCsSFA_s2t_staged,
        tCtSFA_s2t,
    )
```

Construct the Tiled S2T copy with proper configure



The code is simplified to show key APIs and concepts

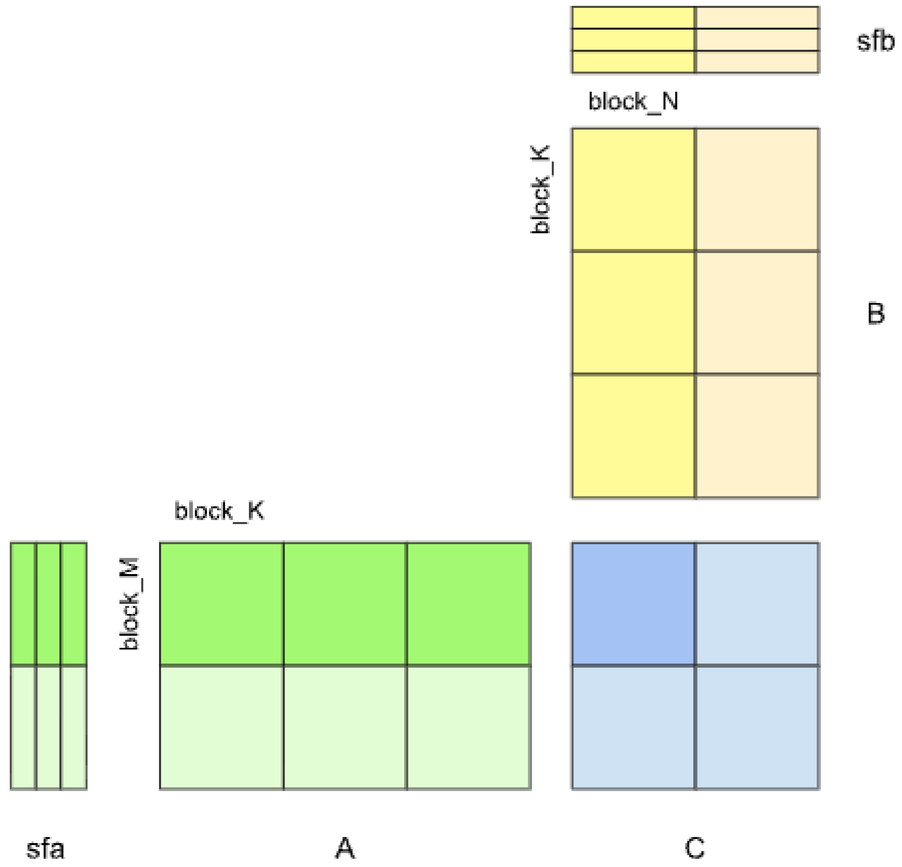
Writing GEMM in Python

Use advance GPU feature: S2T copy

```
#
# Partition for S2T copy of SFA/SFB
#
# Make S2T CopyAtom
copy_atom_s2t = cute.make_copy_atom(
    tcgen05.Cp4x32x128bOp(tcgen05.CtaGroup.ONE),
    sf_dtype,
)
tiled_copy_s2t_sfa = tcgen05.make_s2t_copy(copy_atom_s2t, tCtSFA)
thr_copy_s2t_sfa = tiled_copy_s2t_sfa.get_slice(0)

# ((ATOM_V, REST_V), Rest_Tiler, MMA_MN, MMA_K, STAGE)
tCsSFA_s2t = thr_copy_s2t_sfa.partition_S(sSFA)
# ((ATOM_V, REST_V), Rest_Tiler, MMA_MN, MMA_K)
tCtSFA_s2t = thr_copy_s2t_sfa.partition_D(tCtSFA)

for k_tile in cutlass.range(k_tile_cnt):
    # ... copy data from global to shared memory

    #  Copy SFA/SFB to tmem
    s2t_stage_coord = (None, None, None, None, ab_full.index)
    tCsSFA_s2t_staged = tCsSFA_s2t[s2t_stage_coord]
    cute.copy(
        tiled_copy_s2t_sfa,
        tCsSFA_s2t_staged,
        tCtSFA_s2t,
    )
```

Construct the Tiled S2T copy with proper configure

Partition the shared memory and tensor memory for S2T copy



The code is simplified to show key APIs and concepts

Writing GEMM in Python

Use advance GPU feature: S2T copy

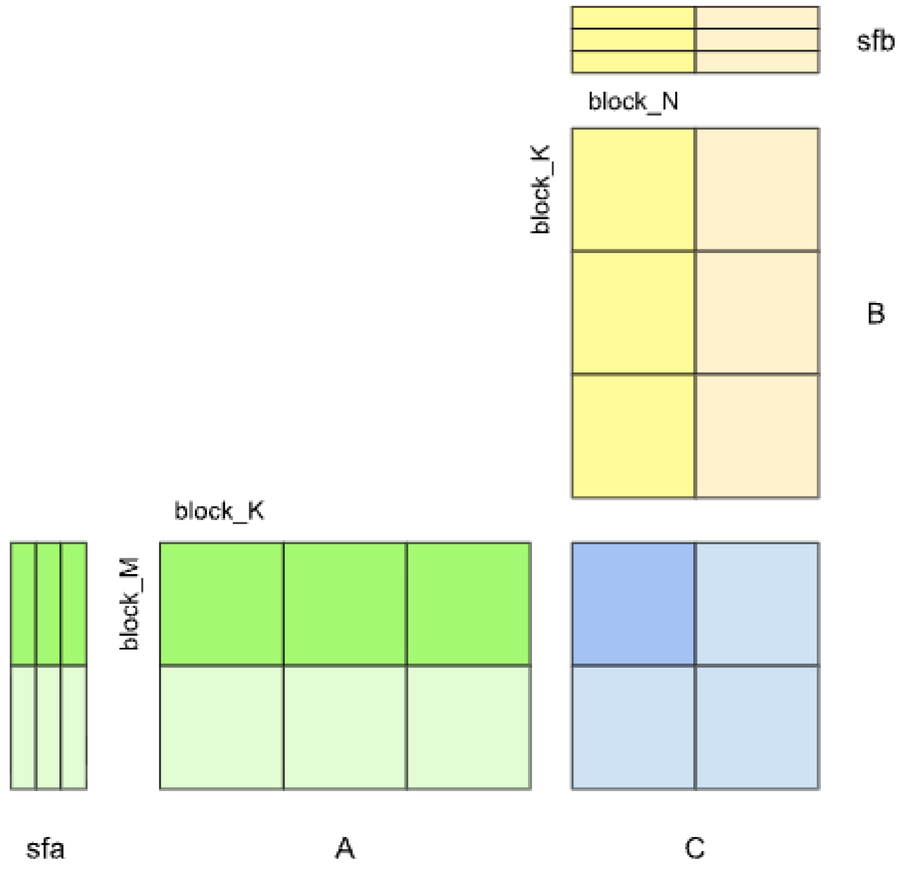
```
#
# Partition for S2T copy of SFA/SFB
#
# Make S2T CopyAtom
copy_atom_s2t = cute.make_copy_atom(
    tcgen05.Cp4x32x128bOp(tcgen05.CtaGroup.ONE),
    sf_dtype,
)
tiled_copy_s2t_sfa = tcgen05.make_s2t_copy(copy_atom_s2t, tCtSFA)
thr_copy_s2t_sfa = tiled_copy_s2t_sfa.get_slice(0)

# ((ATOM_V, REST_V), Rest_Tiler, MMA_MN, MMA_K, STAGE)
tCsSFA_s2t = thr_copy_s2t_sfa.partition_S(sSFA)
# ((ATOM_V, REST_V), Rest_Tiler, MMA_MN, MMA_K)
tCtSFA_s2t = thr_copy_s2t_sfa.partition_D(tCtSFA)
```

```
for k_tile in cutlass.range(k_tile_cnt):
    # ... copy data from global to shared memory

    #  Copy SFA/SFB to tmem
    s2t_stage_coord = (None, None, None, None, ab_full.index)
    tCsSFA_s2t_staged = tCsSFA_s2t[s2t_stage_coord]
    cute.copy(
        tiled_copy_s2t_sfa,
        tCsSFA_s2t_staged,
        tCtSFA_s2t,
    )
```

Construct the Tiled S2T copy with proper configure

Partition the shared memory and tensor memory for S2T copy

Call S2T copy for each kBlock



The code is simplified to show key APIs and concepts

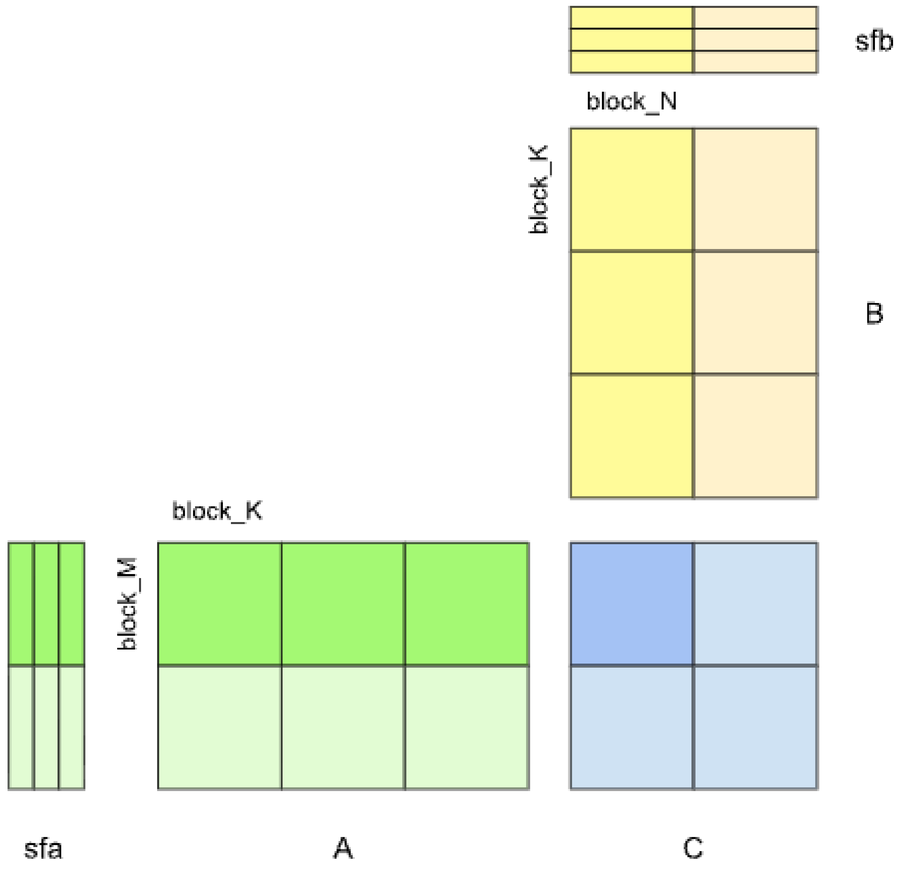
Writing GEMM in Python

Use advance GPU feature: Tensor core

```
# Pick the right tensor core operation
mma_op = tcgen05.MmaMXF4NVF4Op(
    sf_dtype,
    (128, 128, 64), # (MMA, MMA_M, MMA_K)
    tcgen05.CtaGroup.ONE,
    tcgen05.OperandSource.SMEM,
)
tiled_mma = cute.make_tiled_mma(mma_op)
```

Construct the Tiled MMA with right TensorCore configure

```
# (MMA, MMA_M, MMA_K, STAGE)
tCrA = tiled_mma.make_fragment_A(sA)
# (MMA, MMA_N, MMA_K, STAGE)
tCrB = tiled_mma.make_fragment_B(sB)
# (MMA, MMA_M, MMA_N)
tCtAcc = tiled_mma.make_fragment_C(mma_tiler_mnk[:2])

# For each k block, set the scale factor tensors iterator in tiled_mma
# and perform the blockscaled GEMM tensor-core operation.
num_kblocks = cute.size(tCrA, mode=[2])
for kblock_idx in cutlass.range(num_kblocks, unroll_full=True):
    tiled_mma.set(
        tcgen05.Field.SFA,
        gCtSFA[kblock_idx].iterator,
    )
    tiled_mma.set(
        tcgen05.Field.SFB,
        gCtSFB[kblock_idx].iterator,
    )
    cute.gemm(
        tiled_mma,
        tCtAcc,
        tCrA[kblock_idx],
        tCrB[kblock_idx],
        tCtAcc,
    )
```



The code is simplified to show key APIs and concepts

Writing GEMM in Python

Use advance GPU feature: Tensor core

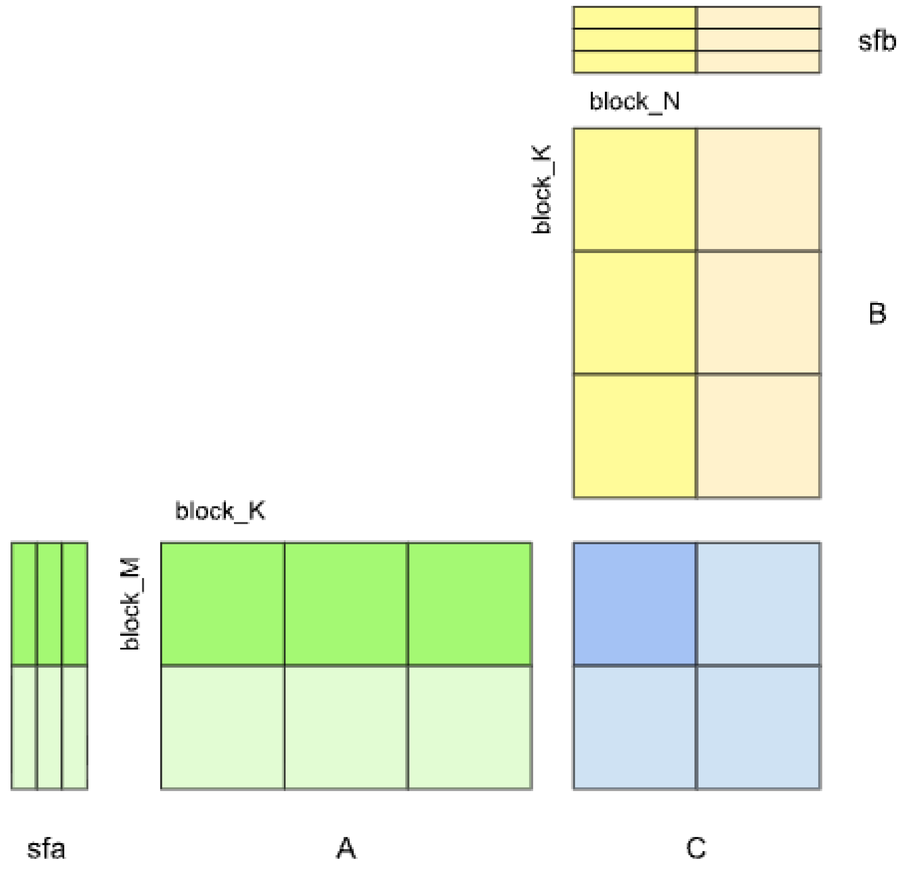
```
# Pick the right tensor core operation
mma_op = tcgen05.MmaMXF4NVF4Op(
    sf_dtype,
    (128, 128, 64), # (MMA, MMA_M, MMA_K)
    tcgen05.CtaGroup.ONE,
    tcgen05.OperandSource.SMEM,
)
tiled_mma = cute.make_tiled_mma(mma_op)
```

Construct the Tiled MMA with right TensorCore configure

```
# (MMA, MMA_M, MMA_K, STAGE)
tCrA = tiled_mma.make_fragment_A(sA)
# (MMA, MMA_N, MMA_K, STAGE)
tCrB = tiled_mma.make_fragment_B(sB)
# (MMA, MMA_M, MMA_N)
tCtAcc = tiled_mma.make_fragment_C(mma_tiler_mnk[:2])
```

Construct A, B, C operands for the TensorCore, A and B come from Smem and Acc comes from TMEM

```
# For each k block, set the scale factor tensors iterator in tiled_mma
# and perform the blockscaled GEMM tensor-core operation.
num_kblocks = cute.size(tCrA, mode=[2])
for kblock_idx in cutlass.range(num_kblocks, unroll_full=True):
    tiled_mma.set(
        tcgen05.Field.SFA,
        gCtSFA[kblock_idx].iterator,
    )
    tiled_mma.set(
        tcgen05.Field.SFB,
        gCtSFB[kblock_idx].iterator,
    )
    cute.gemm(
        tiled_mma,
        tCtAcc,
        tCrA[kblock_idx],
        tCrB[kblock_idx],
        tCtAcc,
    )
```



The code is simplified to show key APIs and concepts

Writing GEMM in Python

Use advance GPU feature: Tensor core

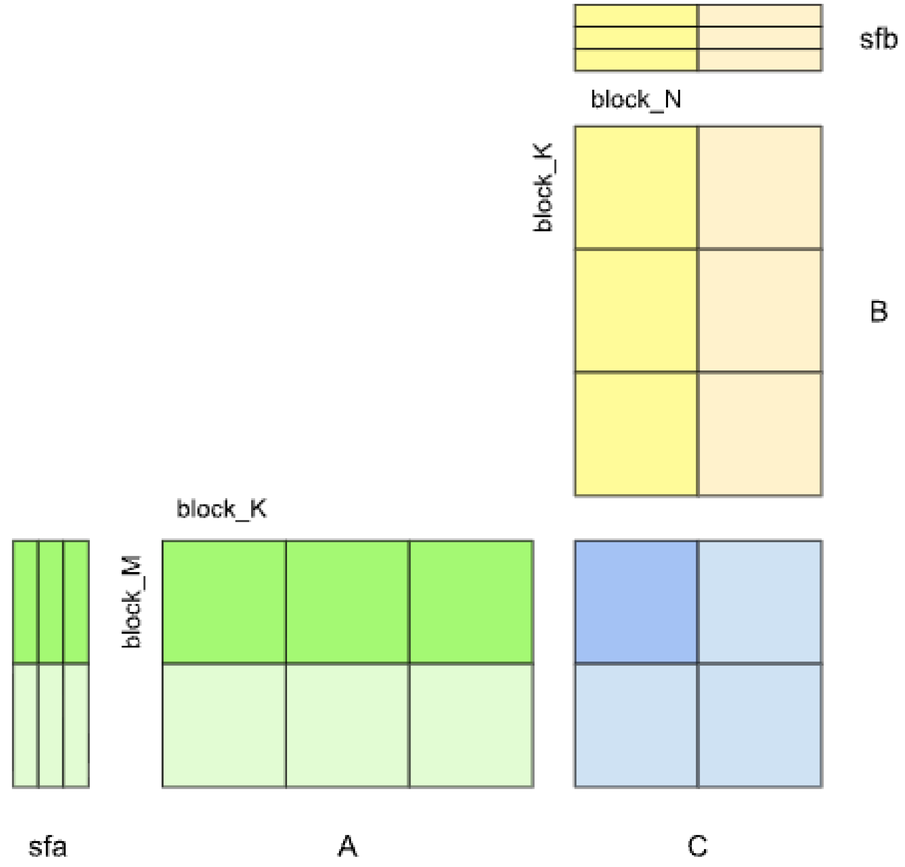
```
# Pick the right tensor core operation
mma_op = tcgen05.MmaMXF4NVF4Op(
    sf_dtype,
    (128, 128, 64), # (MMA, MMA_M, MMA_K)
    tcgen05.CtaGroup.ONE,
    tcgen05.OperandSource.SMEM,
)
tiled_mma = cute.make_tiled_mma(mma_op)

# (MMA, MMA_M, MMA_K, STAGE)
tCrA = tiled_mma.make_fragment_A(sA)
# (MMA, MMA_N, MMA_K, STAGE)
tCrB = tiled_mma.make_fragment_B(sB)
# (MMA, MMA_M, MMA_N)
tCtAcc = tiled_mma.make_fragment_C(mma_tiler_mnk[:2])
```

Construct the Tiled MMA with right TensorCore configure

Construct A, B, C operands for the TensorCore, A and B come from Smem and Acc comes from TMEM

```
# For each k block, set the scale factor tensors iterator in tiled_mma
# and perform the blockscaled GEMM tensor-core operation.
num_kblocks = cute.size(tCrA, mode=[2])
for kblock_idx in cutlass.range(num_kblocks, unroll_full=True):
    tiled_mma.set(
        tcgen05.Field.SFA,
        gCtSFA[kblock_idx].iterator,
    )
    tiled_mma.set(
        tcgen05.Field.SFB,
        gCtSFB[kblock_idx].iterator,
    )
    cute.gemm(
        tiled_mma,
        tCtAcc,
        tCrA[kblock_idx],
        tCrB[kblock_idx],
        tCtAcc,
    )
```

For each TensorCore in the kBlock, update scaling factor iterators and do the accumuation



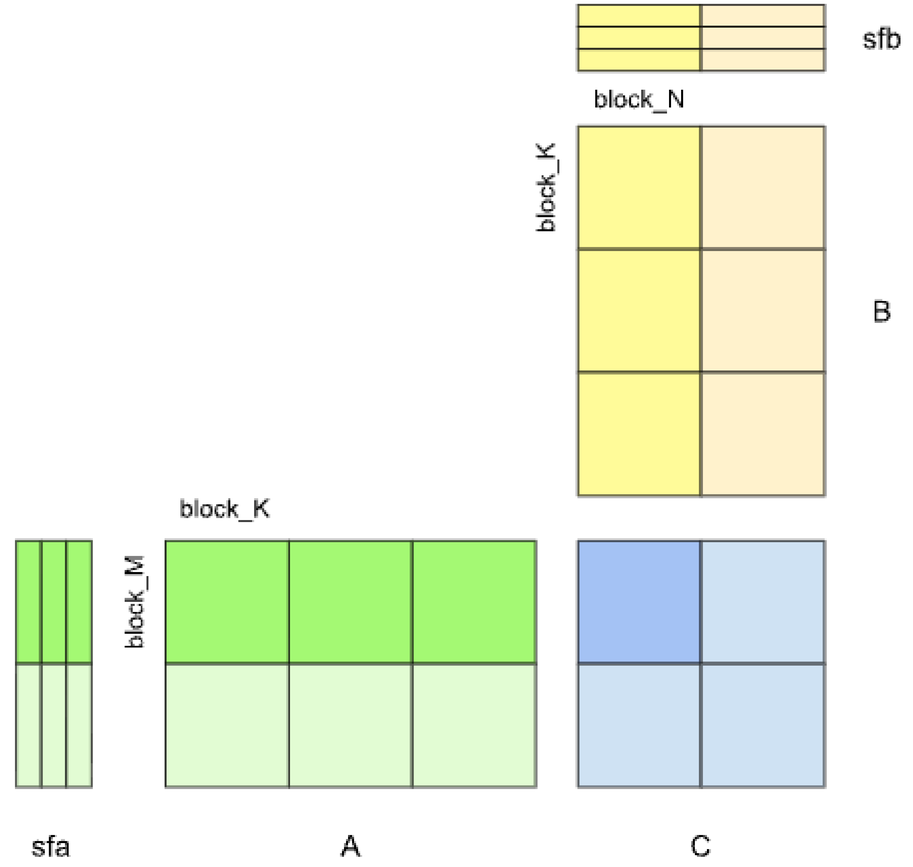
The code is simplified to show key APIs and concepts

Writing GEMM in Pyhton

Use advance GPU feature: T2R copy

```
#
# Epilogue
# Partition for epilogue
#
op = tcgen05.Ld32x32bOp(tcgen05.Repetition.x128, tcgen05.Pack.NONE)
copy_atom_t2r = cute.make_copy_atom(op, cutlass.Float32)
tiled_copy_t2r = tcgen05.make_tmem_copy(copy_atom_t2r, tCtAcc)
thr_copy_t2r = tiled_copy_t2r.get_slice(tidx)

# (T2R_M, T2R_N, EPI_M, EPI_M)
tTR_tAcc = thr_copy_t2r.partition_S(tCtAcc)
# (T2R_M, T2R_N, EPI_M, EPI_N, RestM, RestN, RestL)
tTR_gC = thr_copy_t2r.partition_D(tCgC)

# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)
# Copy accumulator to register
cute.copy(tiled_copy_t2r, tTR_tAcc, tTR_rAcc)
```

Construct the Tiled T2R copy with proper configure



The code is simplified to show key APIs and concepts

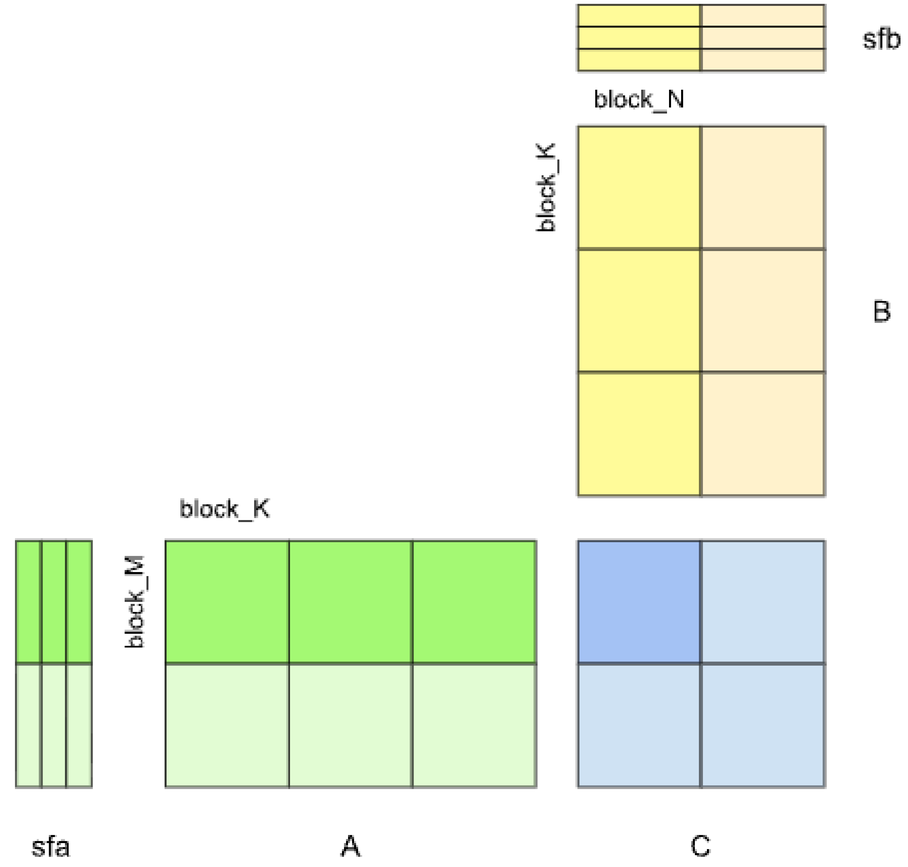
Writing GEMM in Pyhton

Use advance GPU feature: T2R copy

```
#
# Epilogue
# Partition for epilogue
#
op = tcgen05.Ld32x32bOp(tcgen05.Repetition.x128, tcgen05.Pack.NONE)
copy_atom_t2r = cute.make_copy_atom(op, cutlass.Float32)
tiled_copy_t2r = tcgen05.make_tmem_copy(copy_atom_t2r, tCtAcc)
thr_copy_t2r = tiled_copy_t2r.get_slice(tidx)

# (T2R_M, T2R_N, EPI_M, EPI_M)
tTR_tAcc = thr_copy_t2r.partition_S(tCtAcc)
# (T2R_M, T2R_N, EPI_M, EPI_N, RestM, RestN, RestL)
tTR_gC = thr_copy_t2r.partition_D(tCgC)

# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)
# Copy accumulator to register
cute.copy(tiled_copy_t2r, tTR_tAcc, tTR_rAcc)
```

Construct the Tiled T2R copy with proper configure

Slice via thread view as T2R is SIMT instruction



The code is simplified to show key APIs and concepts

Writing GEMM in Pyhton

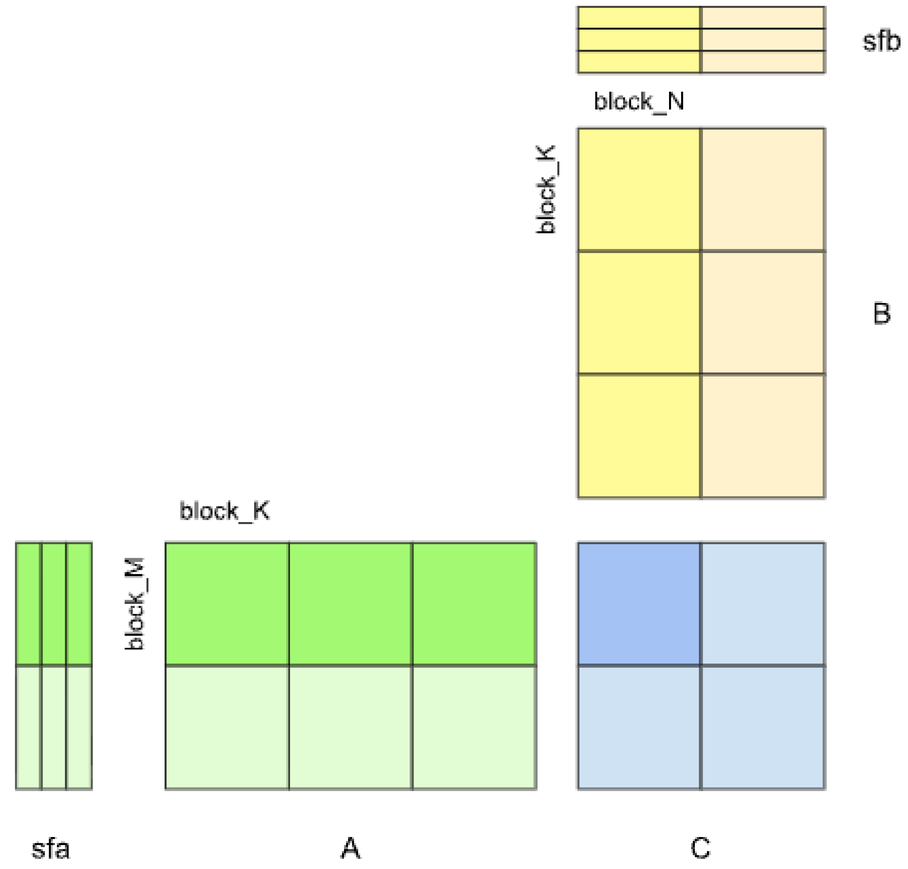
Use advance GPU feature: T2R copy

```
#
# Epilogue
# Partition for epilogue
#
op = tcgen05.Ld32x32bOp(tcgen05.Repetition.x128, tcgen05.Pack.NONE)
copy_atom_t2r = cute.make_copy_atom(op, cutlass.Float32)
tiled_copy_t2r = tcgen05.make_tmem_copy(copy_atom_t2r, tCtAcc)
thr_copy_t2r = tiled_copy_t2r.get_slice(tidx)

# (T2R_M, T2R_N, EPI_M, EPI_M)
tTR_tAcc = thr_copy_t2r.partition_S(tCtAcc)
# (T2R_M, T2R_N, EPI_M, EPI_N, RestM, RestN, RestL)
tTR_gC = thr_copy_t2r.partition_D(tCgC)

# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)
# Copy accumulator to register
cute.copy(tiled_copy_t2r, tTR_tAcc, tTR_rAcc)
```

Construct the Tiled T2R copy with proper configure

Slice via thread view as T2R is SIMT instruction

Partition the tensor memory and global tensor and make register tensor based on global tensor shape for T2R copy



The code is simplified to show key APIs and concepts

Writing GEMM in Pyhton

Use advance GPU feature: T2R copy

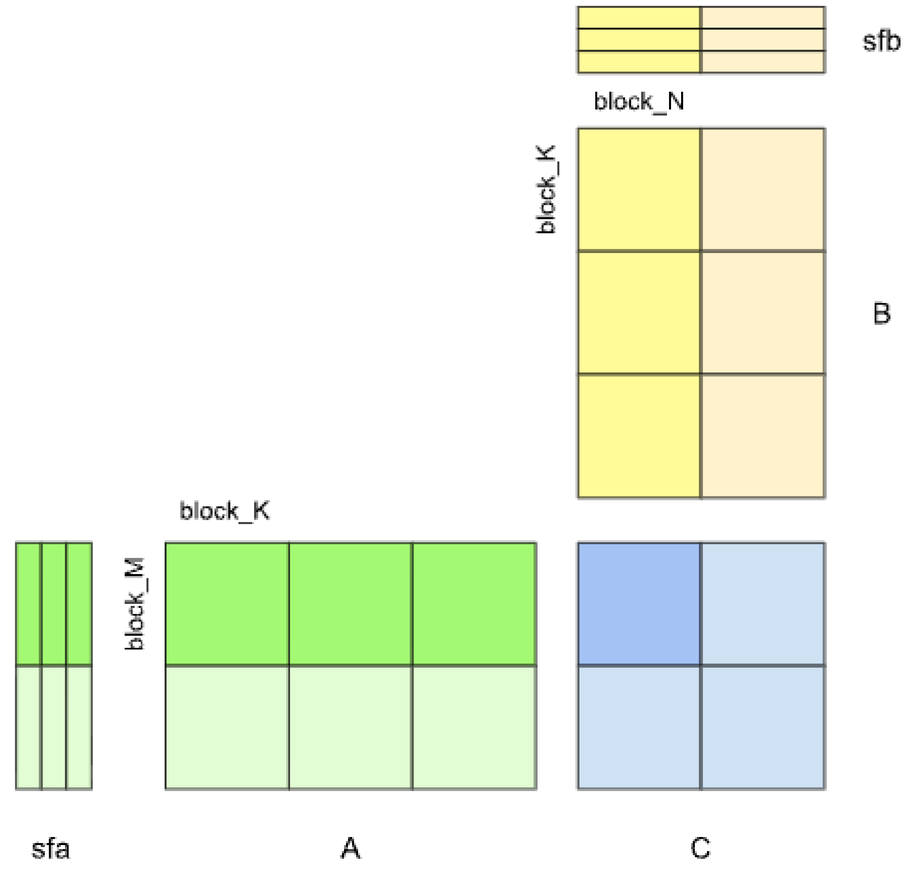
```
#
# Epilogue
# Partition for epilogue
#
op = tcgen05.Ld32x32bOp(tcgen05.Repetition.x128, tcgen05.Pack.NONE)
copy_atom_t2r = cute.make_copy_atom(op, cutlass.Float32)
tiled_copy_t2r = tcgen05.make_tmem_copy(copy_atom_t2r, tCtAcc)
thr_copy_t2r = tiled_copy_t2r.get_slice(tidx)

# (T2R_M, T2R_N, EPI_M, EPI_M)
tTR_tAcc = thr_copy_t2r.partition_S(tCtAcc)
# (T2R_M, T2R_N, EPI_M, EPI_N, RestM, RestN, RestL)
tTR_gC = thr_copy_t2r.partition_D(tCgC)

# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)
# Copy accumulator to register
cute.copy(tiled_copy_t2r, tTR_tAcc, tTR_rAcc)
```

Construct the Tiled T2R copy with proper configure

Slice via thread view as T2R is SIMT instruction

Partition the tensor memory and global tensor and make register tensor based on global tensor shape for T2R copy

Call the T2R copy



The code is simplified to show key APIs and concepts

Writing GEMM in Python

Cute Programming Model

- Typical code sequence (APIs) used to do copy and gemm
  - Copy
    - local_tile (tiling global tensor)
    - partition_S, partition_D **Or** tma_partition
    - cute.copy
  - Mma
    - make_fragment_A
    - make_fragment_B
    - make_fragment_C
    - cute.gemm

- Variable Naming Convention
  - tXaY : Partitioning pattern tX applied to tensor aY
    - ***a:*** is r for register, g for Gmem, s for Smem, c for coordinate, p for predicate, t for Tmem
  - For example:

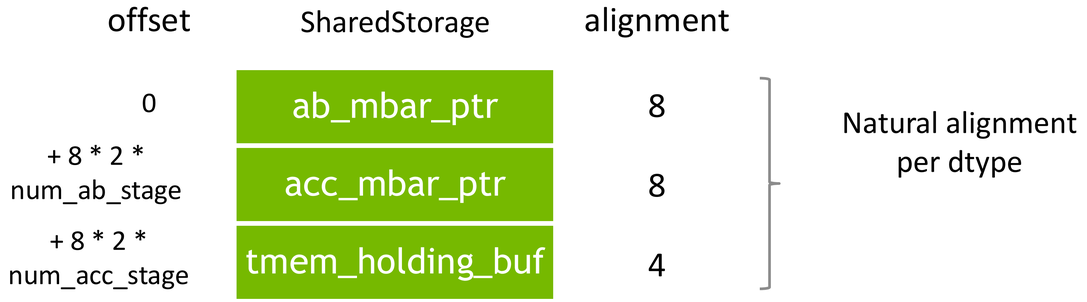
```
# Define thread layouts (static)
tC = make_layout((16, 16), stride=(16, 1));
# Partition gC (M,N) by the tile of tC
tCgC = cute.local_partition(gC, tC, tidx, (1, 1));
```

Writing GEMM in Python

@cute.struct - Customized C struct like data types

- cute.struct decorator
  - Transform a Python class into a memory-mapped structure with precise control over memory layout, alignment and offsets
  - Support scalar, MemRange or nested struct as data members
  - Allow customized data alignment which is essential for better performance
- SmemAllocator
  - Manage shared memory allocation of raw bytes, numeric types, and tensors with static layout

```
@cute.struct
class SharedStorage:
    ab_mbar_ptr: cute.struct.MemRange[cutlass.Int64, self.num_ab_stage * 2]
    acc_mbar_ptr: cute.struct.MemRange[cutlass.Int64, self.num_acc_stage * 2]
    tmem_holding_buf: cutlass.Int32

smem = utils.SmemAllocator()
storage = smem.allocate(SharedStorage)
# (MMA, MMA_M, MMA_K, STAGE)
sA = smem.allocate_tensor(
    element_type=ab_dtype,
    layout=a_smem_layout_staged.outer,
    byte_alignment=128,
    swizzle=a_smem_layout_staged.inner,
)
```



Writing GEMM in Python

Pipeline – warp specialization

```
#
# Initialize mainloop ab_pipeline, acc_pipeline and their states
#
ab_pipeline_producer_group = pipeline.CooperativeGroup(pipeline.Agent.Thread)
ab_pipeline_consumer_group = pipeline.CooperativeGroup(pipeline.Agent.Thread, 1)
ab_producer, ab_consumer = pipeline.PipelineTmaUmma.create(
    barrier_storage=storage.ab_mbar_ptr.data_ptr(),
    num_stages=self.num_ab_stage,
    producer_group=ab_pipeline_producer_group,
    consumer_group=ab_pipeline_consumer_group,
    tx_count=self.num_tma_load_bytes,
).make_participants()
acc_producer, acc_consumer = pipeline.PipelineUmmaAsync.create(
    barrier_storage=storage.acc_mbar_ptr.data_ptr(),
    num_stages=self.num_acc_stage,
    producer_group=ab_pipeline_producer_group,
    consumer_group=pipeline.CooperativeGroup(
        pipeline.Agent.Thread,
        self.threads_per_cta,
    ),
).make_participants()
```

Async pipeline allow producer and consumer work concurrently for best performance

Between TMA and Tensor Core (PipelineTmaUmma)

```
if warp_idx == 0:
    # Execute k_tile loop
    for k_tile in range(k_tile_cnt):
        # Wait for AB buffer empty
        ab_empty = ab_producer.acquire_and_advance()

        #  TMA Load A/B/SFA/SFB
        cute.copy(…
        )
        cute.copy(…
        )
        cute.copy(…
        )
        cute.copy(…
        )
```

```
# Wait for AB buffer full
ab_full = ab_consumer.wait_and_advance()
# tCtAcc += tCrA * tCrSFA * tCrB * tCrSFB
num_kblocks = cute.size(tCrA, mode=[2])
for kblock_idx in cutlass.range(num_kblocks, unroll_full=True):
    kblock_coord = (…
    )

    # Set SFA/SFB tensor to tiled_mma
    sf_kblock_coord = (None, None, kblock_idx)
    tiled_mma.set(…
    )
    tiled_mma.set(…
    )

    cute.gemm(…
    )
    # Enable accumulate on tCtAcc after first kblock
    tiled_mma.set(tcgen05.Field.ACCUMULATE, True)
# Async arrive AB buffer empty
ab_full.release()
```

Between Tensor Core and Epilogue (PipelineUmmaAsync)

```
# Wait for accumulator buffer empty
acc_empty = acc_producer.acquire_and_advance()
# Set ACCUMULATE field to False for the first k_tile iteration
tiled_mma.set(tcgen05.Field.ACCUMULATE, False)
# Execute k_tile loop
for k_tile in range(k_tile_cnt):
    # Wait for AB buffer empty
    ab_empty = ab_producer.acquire_and_advance()

    #  TMA Load A/B/SFA/SFB
    cute.copy(…
    )
    # ...
    # Wait for AB buffer full
    ab_full = ab_consumer.wait_and_advance()

    # tCtAcc += tCrA * tCrSFA * tCrB * tCrSFB
    num_kblocks = cute.size(tCrA, mode=[2])
    for kblock_idx in cutlass.range(num_kblocks, unroll_full=True):
        # ...
        cute.gemm(…
        )
        # Enable accumulate on tCtAcc after first kblock
        tiled_mma.set(tcgen05.Field.ACCUMULATE, True)

    # Async arrive AB buffer empty
    ab_full.release()
acc_empty.commit()
```

```
# Wait for accumulator buffer full
acc_full = acc_consumer.wait_and_advance()

# Copy accumulator to register
cute.copy(tiled_copy_t2r, tTR_tAcc, tTR_rAcc)
acc_vec = epilogue_op(tTR_rAcc.load().to(c_dtype))
tTR_rC.store(acc_vec)
# Store C to global memory
cute.copy(simt_atom, tTR_rC, tTR_gC)

acc_full.release()
```

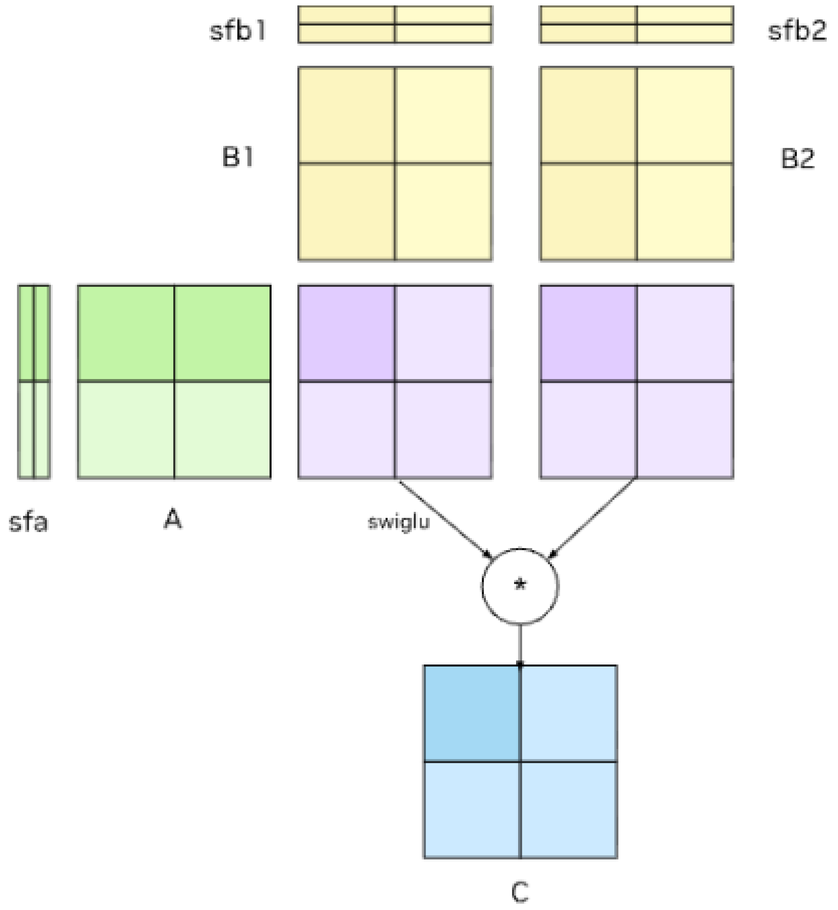
Writing Dual GEMM in Python

Static argument vs. Dynamic argument (doc)

- **Static arguments** hold values that are known at compile time. It is not included in the generated JIT function signature.
- **Dynamic arguments** hold values that are only known at runtime (default behavior)
- cutlass.Constexpr can be used to specify a static argument.

```
#  GPU device kernel
@cute.kernel
def kernel(
    self,
    tiled_mma: cute.TiledMma,
    tma_atom_a: cute.CopyAtom,
    # ...
    epilogue_op: cutlass.Constexpr = lambda x: x
    * (1.0 / (1.0 + cute.math.exp(-x, fastmath=True))),
):

# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc1 = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)
# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc2 = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)

# Copy accumulator from Tmem to Rmem
cute.copy(tiled_copy_t2r, tTR_tAcc1, tTR_rAcc1)
cute.copy(tiled_copy_t2r, tTR_tAcc2, tTR_rAcc2)

# Silu activation on acc1 and multiply with acc2
acc_vec1 = epilogue_op(tTR_rAcc1.load())
acc_vec2 = tTR_rAcc2.load()
acc_vec = acc_vec1 * acc_vec2
```

Constexpr to define swiglu fusion in epilogue



Writing Dual GEMM in Python

Tensor and TensorSSA (doc)

- TensorSSA thread local data modeling for CuTe Tensor in value semantics and immutable
  - Vector based with nested CuTe shape support
  - Load tensor elements as vector / store vector data into tensor
  - Operator overloading for vectorized operations

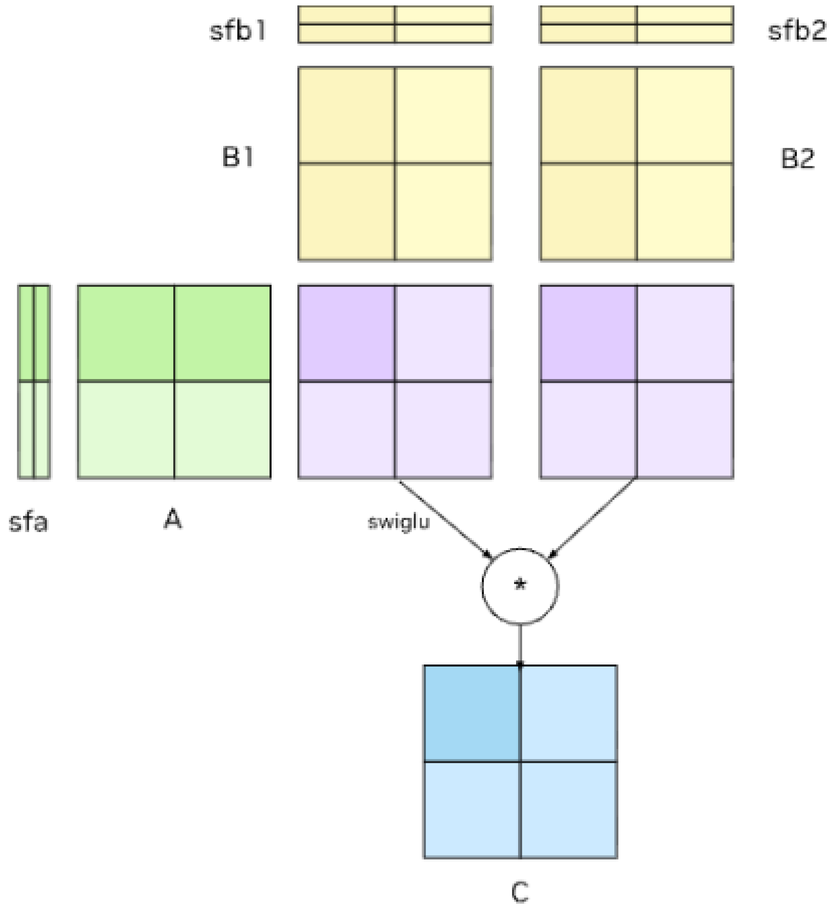
```python
#  GPU device kernel
@cute.kernel
def kernel(
    self,
    tiled_mma: cute.TiledMma,
    tma_atom_a: cute.CopyAtom,
    # ...
    epilogue_op: cutlass.Constexpr = lambda x: x
    * (1.0 / (1.0 + cute.math.exp(-x, fastmath=True))),
):
```

Constexpr to define swiglu fusion in epilogue

```python
# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc1 = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)
# (T2R_M, T2R_N, EPI_M, EPI_N)
tTR_rAcc2 = cute.make_rmem_tensor(
    tTR_gC[None, None, None, None, 0, 0, 0].shape, cutlass.Float32
)

# Copy accumulator from Tmem to Rmem
cute.copy(tiled_copy_t2r, tTR_tAcc1, tTR_rAcc1)
cute.copy(tiled_copy_t2r, tTR_tAcc2, tTR_rAcc2)
```

```python
# Silu activation on acc1 and multiply with acc2
acc_vec1 = epilogue_op(tTR_rAcc1.load())
acc_vec2 = tTR_rAcc2.load()
acc_vec = acc_vec1 * acc_vec2
```

Load data to register, do swiglu on Acc1 and then do elementwise multiply for Acc1 and Acc2

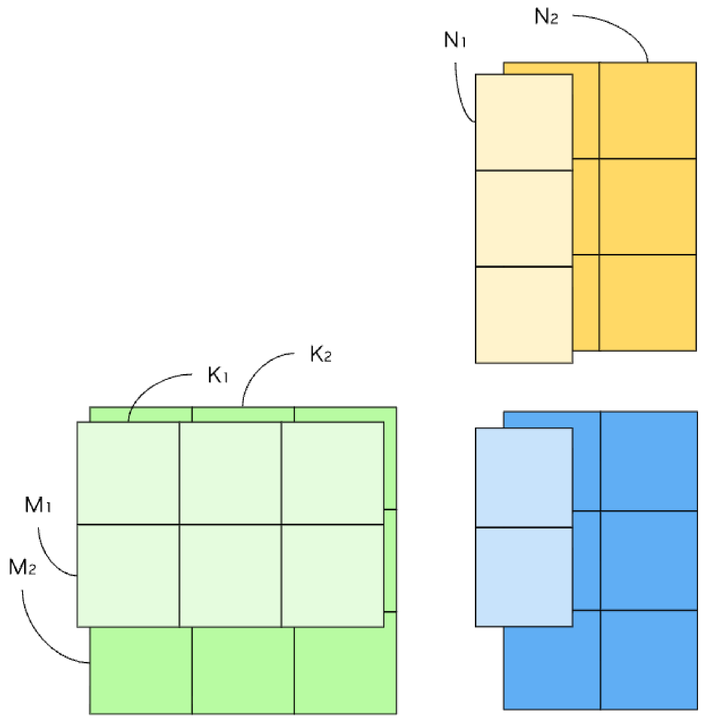


Writing Group GEMM in Python

Update TMA on the fly

- CTA will figure which group to compute on device and thus need to update TMA desc from device before doing cute.copy
  - **TensorMapManager**: provides a set of utils to manage TensorMap operations, including initialization and updates
  - **Step1**
    - Each CTA gets its own Gmem tensormap pointers for A/B/SFA/SFB tensors

```
tensormap_manager = utils.TensorMapManager(
    utils.TensorMapUpdateMode.SMEM,
    128,
)
tensormap_a_gmem_ptr = tensormap_manager.get_tensormap_ptr(
    tensormaps[(bidz, 0, None)].iterator
)
tensormap_b_gmem_ptr = tensormap_manager.get_tensormap_ptr(
    tensormaps[(bidz, 1, None)].iterator
)
tensormap_sfa_gmem_ptr = tensormap_manager.get_tensormap_ptr(
    tensormaps[(bidz, 2, None)].iterator
)
tensormap_sfb_gmem_ptr = tensormap_manager.get_tensormap_ptr(
    tensormaps[(bidz, 3, None)].iterator
)
```

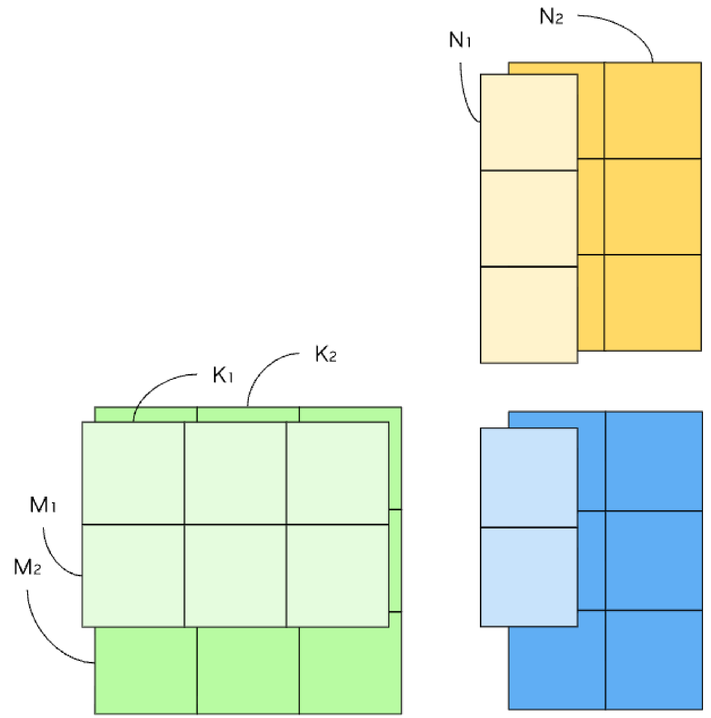


Writing Group GEMM in Python

Update TMA on the fly

- CTA will figure which group to compute on device and thus need to update TMA desc from device before doing cute.copy
  - **TensorMapManager**: provides a set of utils to manage TensorMap operations, including initialization and updates
  - **Step2**
    - Construct tma desc in shared memory
    - Copying tensormap (with fake globalAddress, globalDim, globalStrides) from const memory to Smem

```
if warp_idx == 0:
    tensormap_manager.init_tensormap_from_atom(
        tma_atom_a, tensormap_a_smem_ptr, 0
    )
    tensormap_manager.init_tensormap_from_atom(
        tma_atom_b, tensormap_b_smem_ptr, 0
    )
    tensormap_manager.init_tensormap_from_atom(
        tma_atom_sfa, tensormap_sfa_smem_ptr, 0
    )
    tensormap_manager.init_tensormap_from_atom(
        tma_atom_sfb, tensormap_sfb_smem_ptr, 0
    )
```

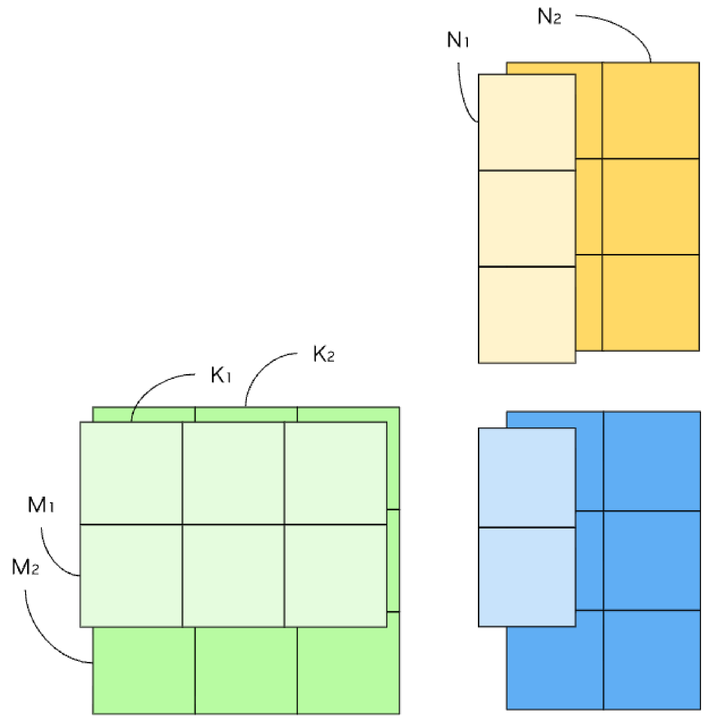


Writing Group GEMM in Python

Update TMA on the fly

- CTA will figure which group to compute on device and thus need to update TMA desc from device before doing cute.copy
  - **TensorMapManager**: provides a set of utils to manage TensorMap operations, including initialization and updates
  - **Step3**
    - Update tensormaps' globalAddress, globalDim, globalStrides with real global memory addresses and problem sizes.
    - Store tma desc to global tensormap space

```
tensormap_manager.update_tensormap(
    (
        real_tensor_a,
        real_tensor_b,
        real_tensor_sfa,
        real_tensor_sfb,
    ),
    (tma_atom_a, tma_atom_b, tma_atom_sfa, tma_atom_sfb),
    (
        tensormap_a_gmem_ptr,
        tensormap_b_gmem_ptr,
        tensormap_sfa_gmem_ptr,
        tensormap_sfb_gmem_ptr,
    ),
    0,  # tma warp id
    (
        tensormap_a_smem_ptr,
        tensormap_b_smem_ptr,
        tensormap_sfa_smem_ptr,
        tensormap_sfb_smem_ptr,
    ),
)
```

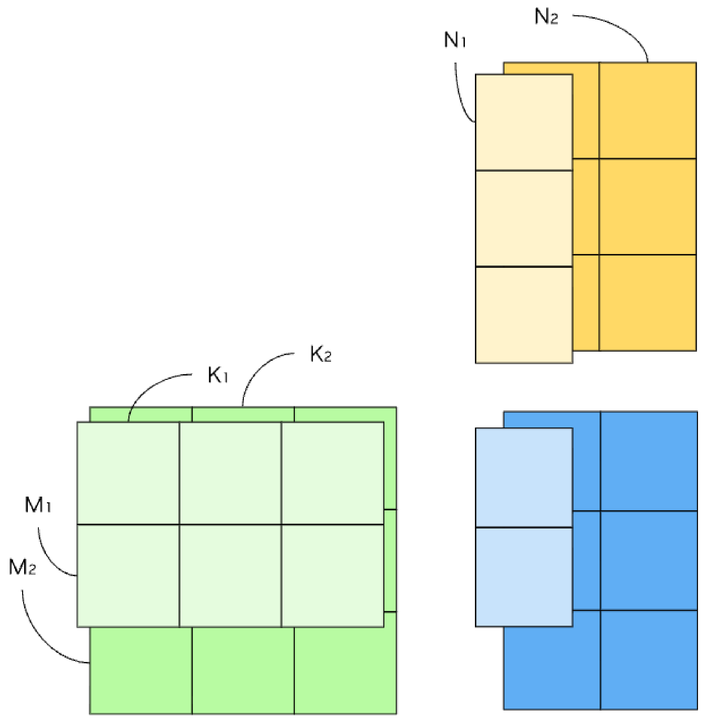


Writing Group GEMM in Python

Update TMA on the fly

- CTA will figure which group to compute on device and thus need to update TMA desc from device before doing cute.copy
  - **TensorMapManager**: provides a set of utils to manage TensorMap operations, including initialization and updates
  - **Step4**
    - Ensure TMA desc are stored in Gmem

```
# Let warp 0 initialize tensormap
if warp_idx == 0:
    tensormap_manager.fence_tensormap_update(tensormap_a_gmem_ptr)
    tensormap_manager.fence_tensormap_update(tensormap_b_gmem_ptr)
    tensormap_manager.fence_tensormap_update(tensormap_sfa_gmem_ptr)
    tensormap_manager.fence_tensormap_update(tensormap_sfb_gmem_ptr)
```



Compile and Run

Cache

```python
# Global cache for compiled kernels (keyed by group size)
_compiled_kernel_cache = {}
# This function is used to compile the kernel once and cache it and then allow users to
# run the kernel multiple times to get more accurate timing results.
def compile_kernel(problem_sizes):
    global _compiled_kernel_cache

    # Convert problem_sizes list to a hashable tuple for use as dictionary key
    cache_key = f"{len(problem_sizes)}"

    # Check if we already have a compiled kernel for these problem sizes
    if cache_key in _compiled_kernel_cache:
        return _compiled_kernel_cache[cache_key]

    cute_ptr_of_tensor_of_problem_sizes = make_ptr(
        cutlass.Int32, 0, cute.AddressSpace.gmem, assumed_align=16,
    )
    # Create fake tensors and pointers to compile the kernel
    compiled_func = cute.compile(
        my_kernel,
        cute_ptr_of_tensor_of_problem_sizes,
        cute_ptr_of_tensor_of_abc_ptrs,
        cute_ptr_of_tensor_of_sfasfb_ptrs,
        cute_ptr_of_tensor_of_tensormap,
        total_num_clusters,
        problem_sizes,
        num_groups
    )
    # Store compiled kernel in cache with problem_sizes as key
    _compiled_kernel_cache[cache_key] = compiled_func
    return compiled_func
```

Check if the cache contain the cubin

Use nullptr to compile

Cache the kernel based on group sizes

```python
def custom_kernel(data: input_t) -> output_t:
    abc_tensors, _, sfasfb_reordered_tensors, problem_sizes = data

    compiled_func = compile_kernel(problem_sizes)

    # Extract raw data pointers from all input tensors for each group
    # These will be passed to the GPU kernel to access the actual tensor data
    abc_ptrs = []
    sfasfb_ptrs = []
    for i, ((a, b, c), (sfa_reordered, sfb_reordered), (m, n, k, l)) in
        enumerate(zip(abc_tensors, sfasfb_reordered_tensors, problem_sizes)):
        # Store pointers to A, B, and C matrices for this group
        abc_ptrs.append((a.data_ptr(), b.data_ptr(), c.data_ptr()))
        # Store pointers to scale factor tensors for this group
        sfasfb_ptrs.append((sfa_reordered.data_ptr(), sfb_reordered.data_ptr()))

    # ... codes to prepare arguments of the kernel

    # Create CuTe pointers to the metadata tensors that will be passed to the kernel
    # These allow the GPU kernel to read problem sizes and tensor pointers
    cute_ptr_of_tensor_of_abc_ptrs = make_ptr(
        cutlass.Int64,
        tensor_of_abc_ptrs.data_ptr(),
        cute.AddressSpace.gmem,
        assumed_align=16,
    )
    # ... codes to prepare arguments of the kernel

    # Launch the JIT-compiled GPU kernel with all prepared data
    # The kernel will perform block-scaled group GEMM: C = A * SFA * B * SFB for all groups
    compiled_func(
        cute_ptr_of_tensor_of_problem_sizes, # Pointer to problem sizes array
        cute_ptr_of_tensor_of_abc_ptrs,      # Pointer to ABC tensor pointers array
        cute_ptr_of_tensor_of_sfasfb_ptrs,   # Pointer to scale factor pointers array
        cute_ptr_of_tensor_of_tensormap,     # Pointer to tensormap buffer
        total_num_clusters,                  # Total number of CTAs to launch
        problem_sizes,                       # Problem sizes list (for host-side processing)
        num_groups,                          # Number of groups in this batch
    )
```

Pass the real data when run the kernel

Debug in Python

cute.print/cute.print_tensor

- **print**: Can only show static values at compile time
- **cute.printf**: Can display both static and dynamic values at runtime
- **cute.print_tensor**: Dump contents of the tensor
  - Only support Float16/Float32/Float64 data type

```python
# (bM, bN, RestM, RestN, RestL)
gC_mnl = cute.local_tile(
    mC_mnl, cute.slice_(self.mma_tiler, (None, None, 0)), (None, None, None)
)
print(f"mC_mnl: {mC_mnl}")
print(f"gC_mnl: {gC_mnl}")
if tidx == 0:
    cute.printf("mC_mnl: {}", mC_mnl)
    cute.printf("gC_mnl: {}", gC_mnl)
    cute.print_tensor(mC_mnl)
    cute.print_tensor(gC_mnl[None, 0, 0, 0, 0])
    cute.print_tensor(gC_mnl[None, 0, 0, 0, 0], verbose=True)
```

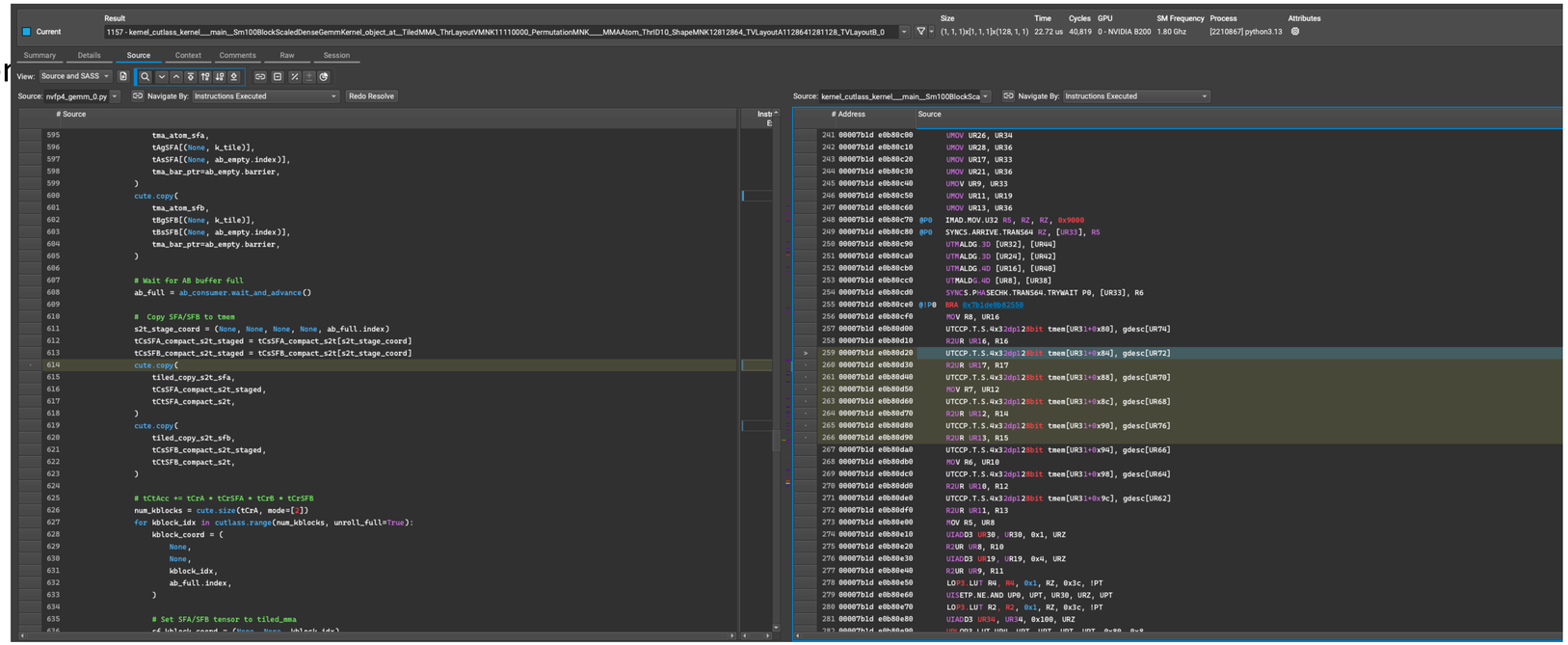
```
mC_mnl: tensor<ptr<f16, gmem, align<16>> o (?{div=32},?,?):(?,1,?)>
gC_mnl: tensor<ptr<f16, gmem, align<16>> o (128,128,?,?,?):(?,1,?{div=128},128,?)>
mC_mnl: raw_ptr(0x0000738313008000: f16, gmem, align<16>) o (128,128,1):(128,1,16384)
gC_mnl: raw_ptr(0x0000738313008000: f16, gmem, align<16>) o (128,128,1,1,1):(128,1,16384,128,16384)
tensor(raw_ptr(0x0000738313008000: f16, gmem, align<16>) o (128,128,1):(128,1,16384), data=
       [[[-0.017166, -0.003027,  0.513672, ..., -1.078125,  0.255615,  0.859375, ],
         [ 0.047668, -0.700684,  0.719238, ..., -1.219727,  0.462402,  1.126953, ],
         [-1.769531,  0.977051,  0.722656, ..., -1.695312,  0.187012, -0.850586, ],
         ...
         [ 0.396484,  0.466309, -0.545898, ..., -1.271484,  0.546875, -0.288818, ],
         [-0.044281, -1.361328, -0.631836, ..., -0.820801,  1.089844,  0.040985, ],
         [-0.324707, -1.977539,  1.054688, ...,  0.364746,  0.242188, -1.424805, ]]])
tensor(raw_ptr(0x0000738313008000: f16, gmem, align<16>) o (128):(128), data=
       [-0.017166, ],
       [ 0.047668, ],
       [-1.769531, ],
       ...
       [ 0.396484, ],
       [-0.044281, ],
       [-0.324707, ])
tensor(raw_ptr(0x0000738313008000: f16, gmem, align<16>) o (128):(128), data= (
        (0)=-0.017166
        (1)= 0.047668
        (2)=-1.769531
        (3)= 0.019379
        (4)=-0.750488
        (5)=-0.035400
        (6)=-1.210938
        (7)= 1.065430
        (8)=-0.353271
        (9)= 0.181519
        (10)= 0.140259
        (11)=-0.308838
        (12)=-1.227539
        (13)= 0.475342
        (14)= 0.270996
        (15)=-0.618652
        (16)=-0.033936
        (17)=-0.113770
        (18)=-0.421387
        (19)=-0.118774
        (20)= 2.476562
```

Debug in Python

Debugging

- Some useful flag
  - export CUTE_DSL_KEEP_IR=1
    - Keep generated CuTe IR
  - export CUTE_DSL_KEEP_PTX=1
    - Keep ptx in a *.ptx file
  - export CUTE_DSL_KEEP_CUBIN=1
    - Keep cubin in a *.cubin file
- Enable lineinfo to do source/sass correlation in nsight-compute

```
# Compile gemm kernel
compiled_gemm = cute.compile[cute.GenerateLineInfo(True)](
    gemm,
    a_ptr,
    b_ptr,
    sfa_ptr,
    sfb_ptr,
    c_ptr,
    (m, n, k, l),
    current_stream,
)
```



Add new feature in Python

Inline ptx

- For cases that lack feature supports in wheel, users can inline ptx to unblock themselves

```
# ////////////////////////////////////////////////////////////////////////////////
# silu activation
# ////////////////////////////////////////////////////////////////////////////////
if self._enable_fast_sigmoid:
    t1 = acc_S.load()
    t2 = t1 * 0.5
    acc_S.store(t2)
    for i in cutlass.range_constexpr(cute.size(acc_S.shape[0])):
        for j in cutlass.range_constexpr(cute.size(acc_S.shape[1])):
            for k in cutlass.range_constexpr(cute.size(acc_S.shape[2])):
                ret = llvm.inline_asm(
                    cutlass.Float32.mlir_type,
                    [acc_S[i, j, k].ir_value()],
                    "tanh.approx.f32 $0, $1;",
                    "=f,f",
                    has_side_effects=False,
                    is_align_stack=False,
                    asm_dialect=llvm.AsmDialect.AD_ATT,
                )
                acc_S[i, j, k] = ret
    t3 = acc_S.load()
    t4 = t2 * t3 + t2
    acc_S.store(t4)
```

hstu_attention

Other Resources

- CuTe at GPU Mode
- CuTeDSL documentations
- CuTeDSL examples

THANK YOU !