Enable Tensor Core Programming in Python with CUTLASS 4.0

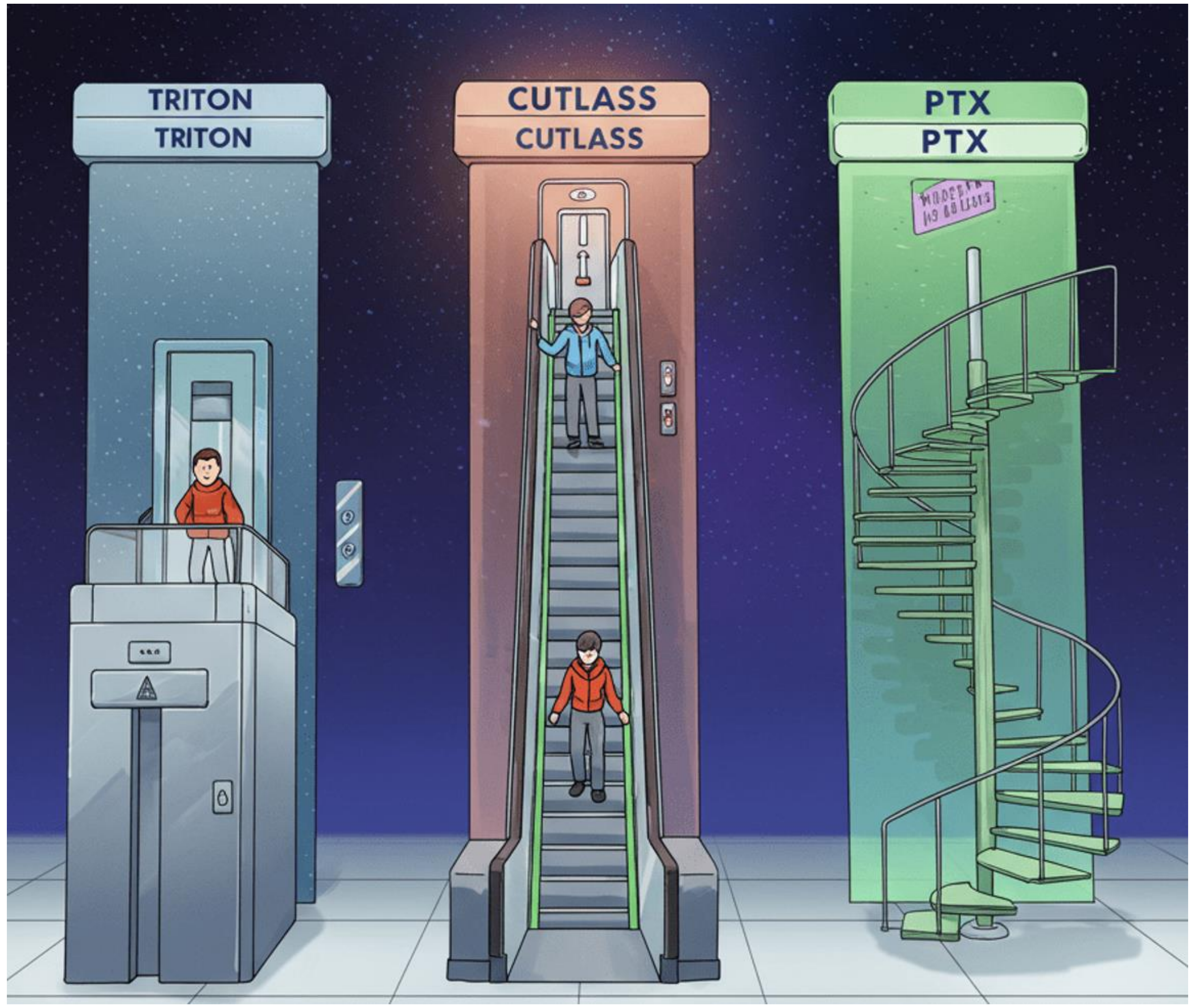
Kihiro Bando, Brandon Sun | 2025-03-21

Why CUTLASS?

Enabling innovation for SOL performance

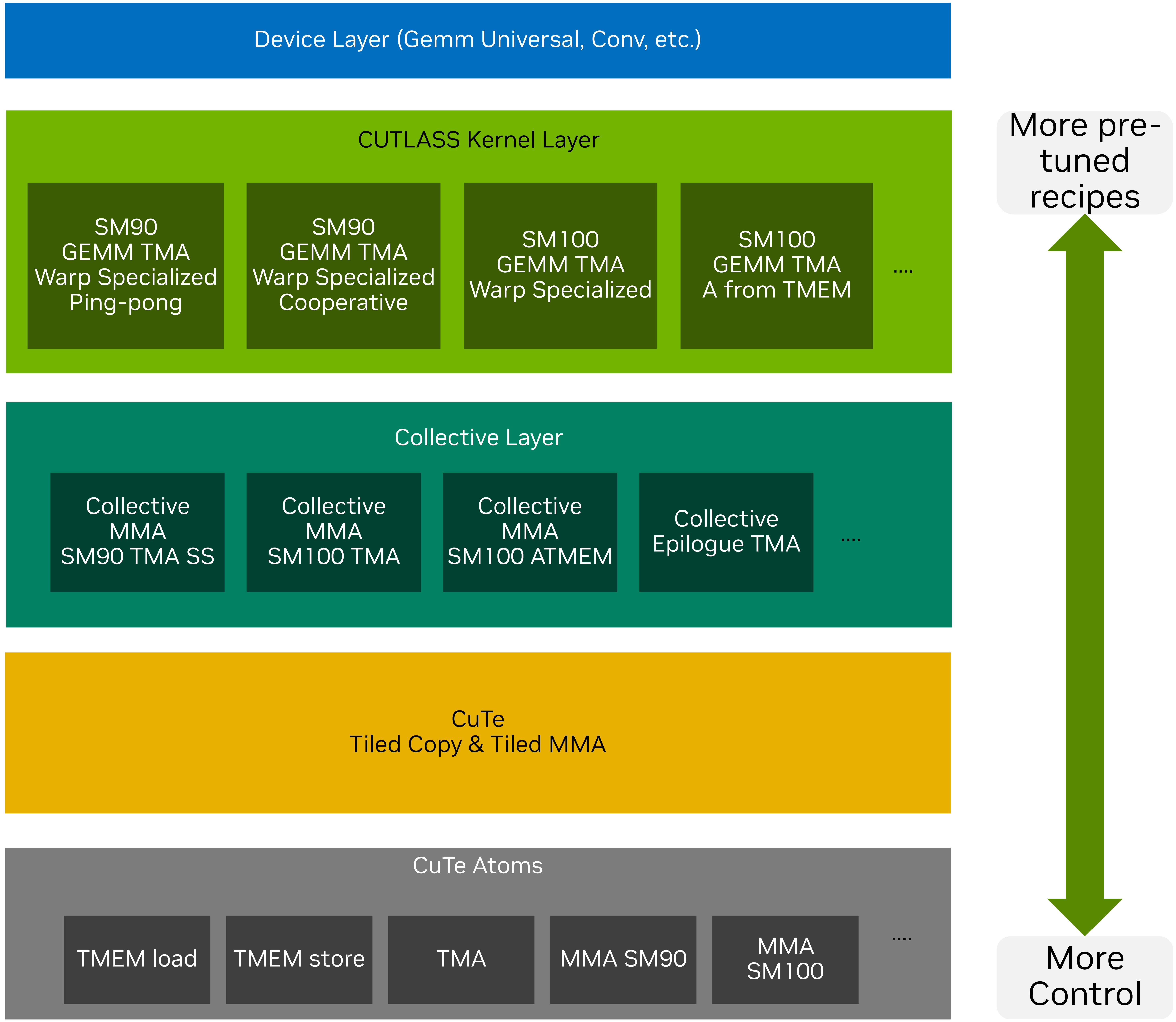
- High-level generators leveraging compilers are popular
  - Hide and automate a lot of details
  - Get excellent performance on common use cases, but…
  - Algorithmic innovations require lower-level ***abstractions***
  - Advanced features like PDL require fine grain control

- 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

A set of useful abstractions for productivity and performance at all scopes and scales



Device Layer (Gemm Universal, Conv, etc.)

CUTLASS Kernel Layer

SM90 GEMM TMA Warp Specialized Ping-pong

SM90 GEMM TMA Warp Specialized Cooperative

SM100 GEMM TMA Warp Specialized

SM100 GEMM TMA A from TMEM

....

Collective Layer

Collective MMA SM90 TMA SS

Collective MMA SM100 TMA

Collective MMA SM100 ATMEM

Collective Epilogue TMA

....

CuTe
Tiled Copy & Tiled MMA

CuTe Atoms

TMEM load

TMEM store

TMA

MMA SM90

MMA SM100

....

More pre-tuned recipes

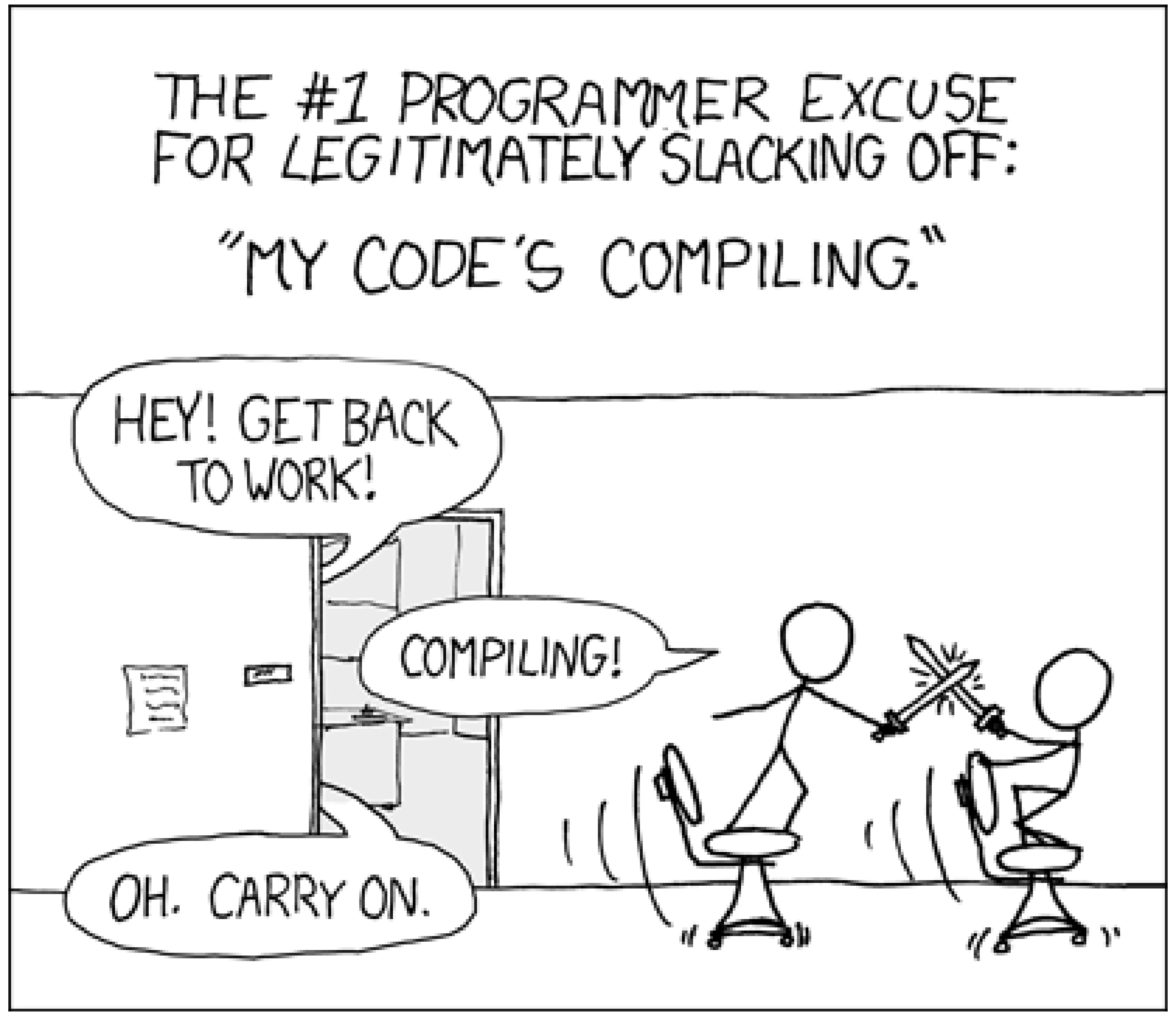
More Control

- Open source https://github.com/NVIDIA/cutlass
- Presented: GTC’18, GTC’19, GTC’20, GTC’21, GTC’22, GTC’22, GTC’23, GTC’24
- Multiple entry points depending on your needs
- More details this year in *Programming Blackwell Tensor Cores with CUTLASS* [S72720]

Major pain points with C++

C++ templates and unfortunate consequences

- C++ templates suffer from slow compilation time
  - Front-end too generic for our purposes
  - Prevents fast iteration
  - Prohibits JIT-ting at scale and brute force auto-tuning
- C++ templates are inconvenient
  - Additional mental load when writing compile-time logic
  - Error messages are longer than novels
- The DL space fully embraces the python ecosystem regardless
  - Everybody hates writing binding code
  - Dependency on nvcc
- LLMs are likely better at generating python programs



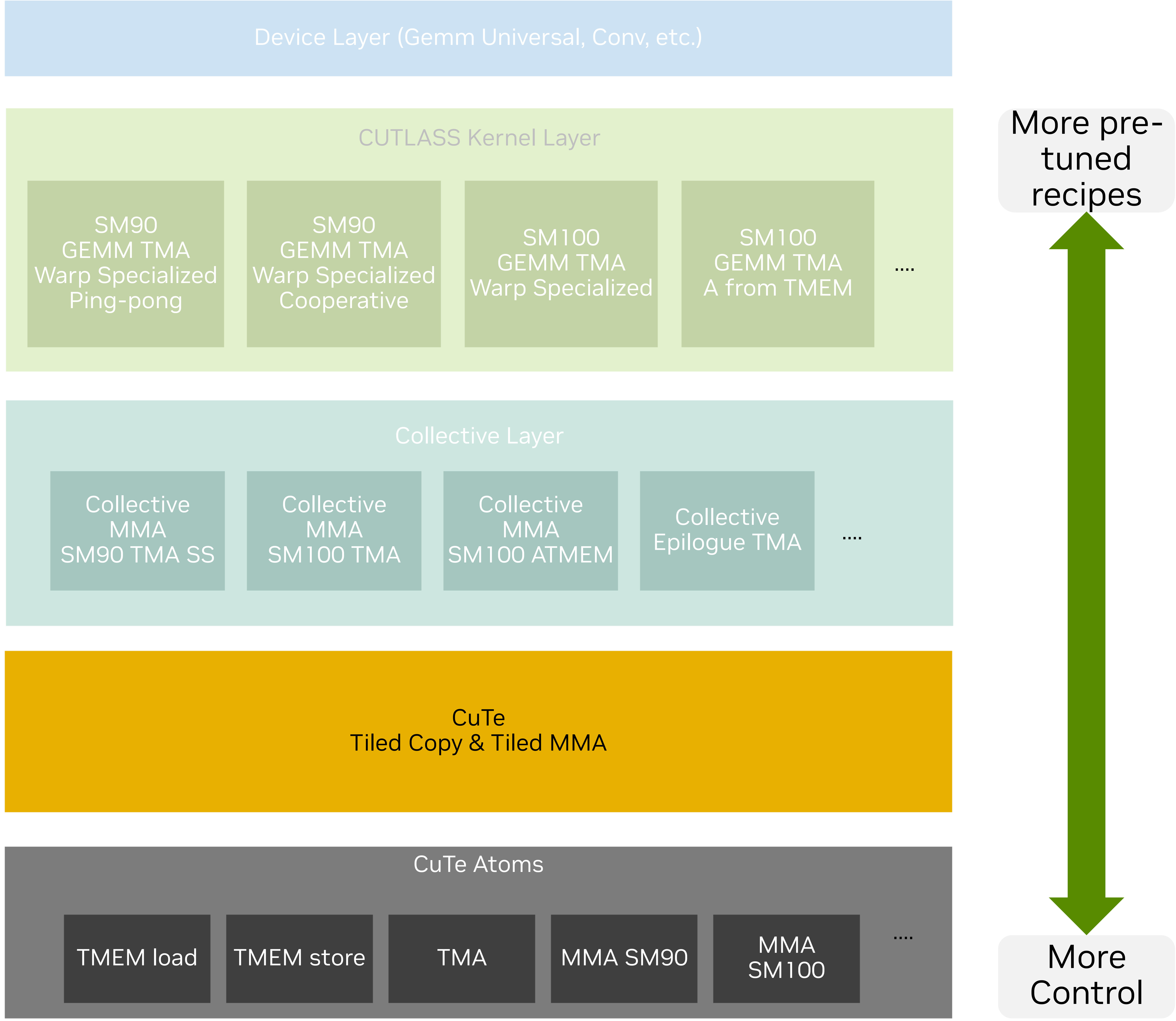
Do I have to tolerate all of this to use CUTLASS?

Introducing with CUTLASS 4.0

Tensor core programming in Python

CUTLASS in Python

Initial launch to include CuTe


Device Layer (Gemm Universal, Conv, etc.)
CUTLASS Kernel Layer
SM90 GEMM TMA Warp Specialized Ping-pong
SM90 GEMM TMA Warp Specialized Cooperative
SM100 GEMM TMA Warp Specialized
SM100 GEMM TMA A from TMEM
....
Collective Layer
Collective MMA SM90 TMA SS
Collective MMA SM100 TMA
Collective MMA SM100 ATMEM
Collective Epilogue TMA
....
CuTe
Tiled Copy & Tiled MMA
CuTe Atoms
TMEM load
TMEM store
TMA
MMA SM90
MMA SM100
....
More pre-tuned recipes
More Control

This first release makes available a ***mature*** low-level tensor programming model, giving access to tensor cores with full control.

More to come later…

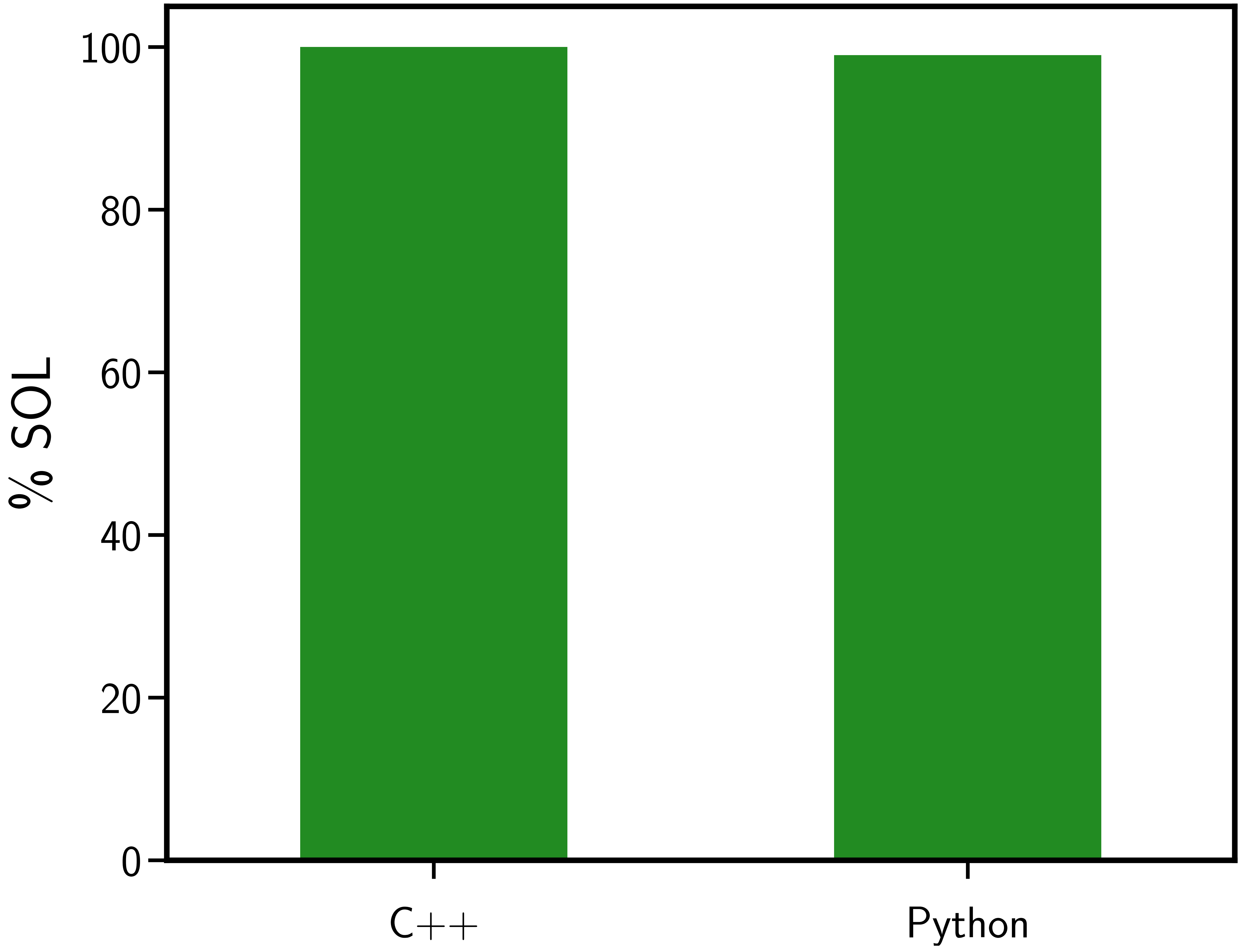
- `make_shape(Int<1>{}, Int<2>{}, x)` → `(1,2,x)`
- `make_layout`
- `make_identity_tensor`
- `zipped_divide`
- `local_tile`
- `tiled_mma.get_slice`
- `thr_mma.partition_A`
- `thr_copy.partition_S`

CUTLASS in Python

What do you get?

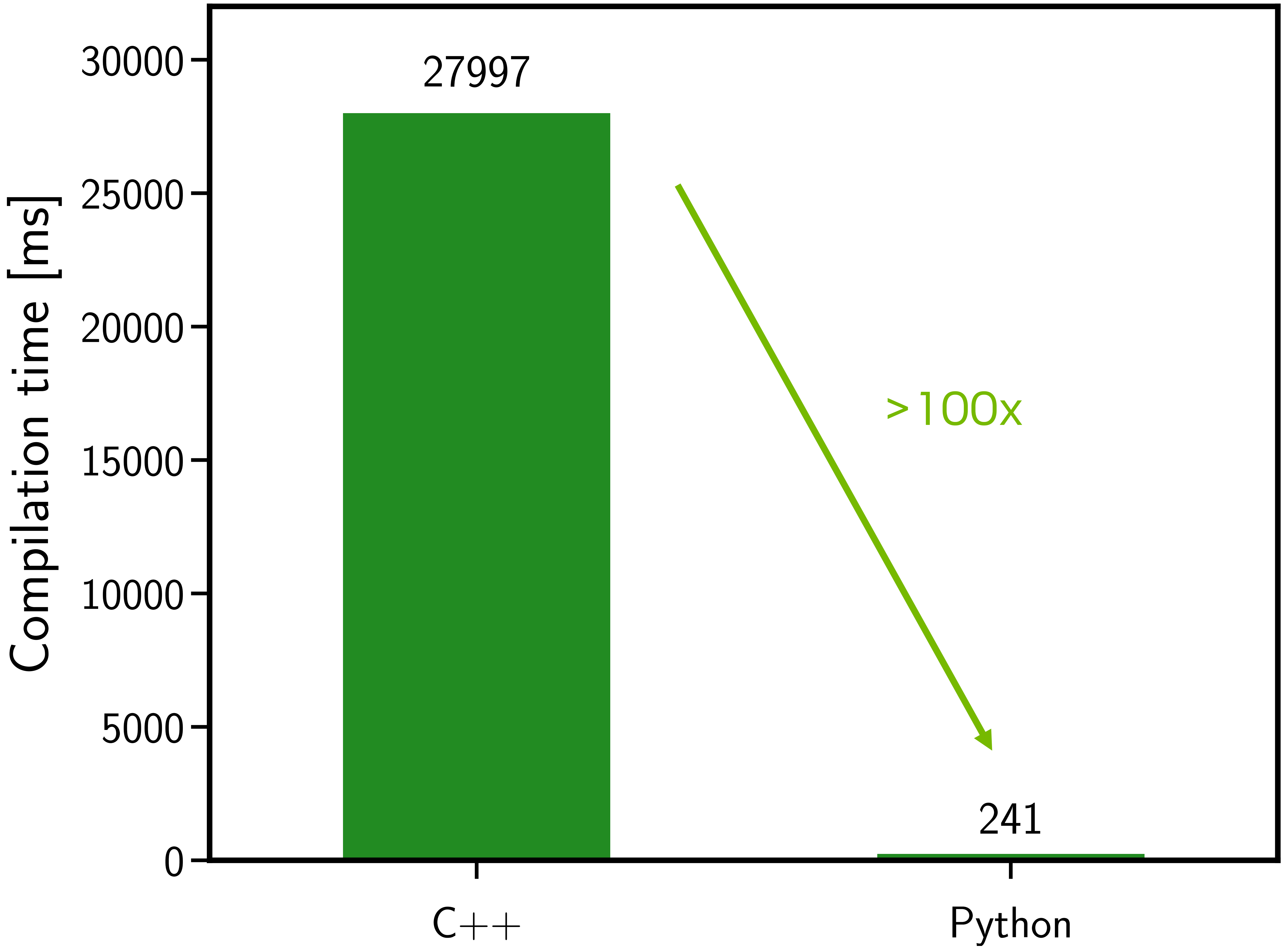
8kx8kx8k GEMM

Peak Performance!



Higher is better

Blazing Fast Compilation Time!



Lower is better

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(config=cutlass.LaunchConfig(
    grid=(1, 1, 1), block=(32, 1, 1)))

host()
```

```
python3 hello_world.py
```

Agenda

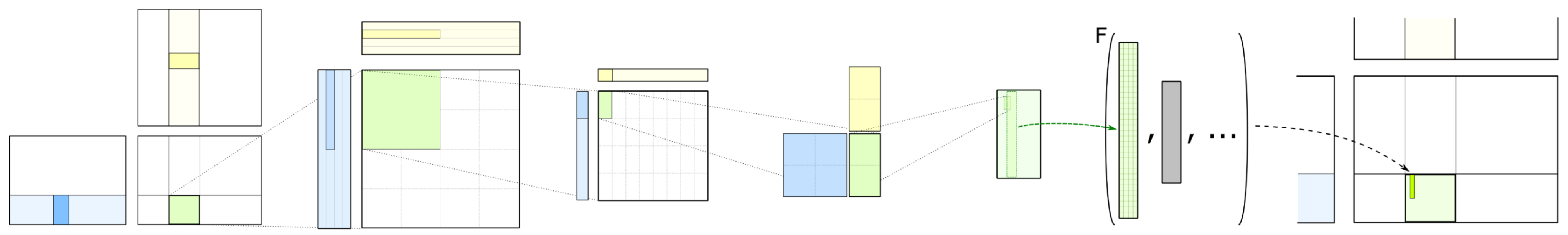
- Introduction and Motivations
- The DSL Infrastructure
- Kernel Authoring in Python with CuTe
- Runtime Performance
- Conclusion

Soul of CuTe DSL

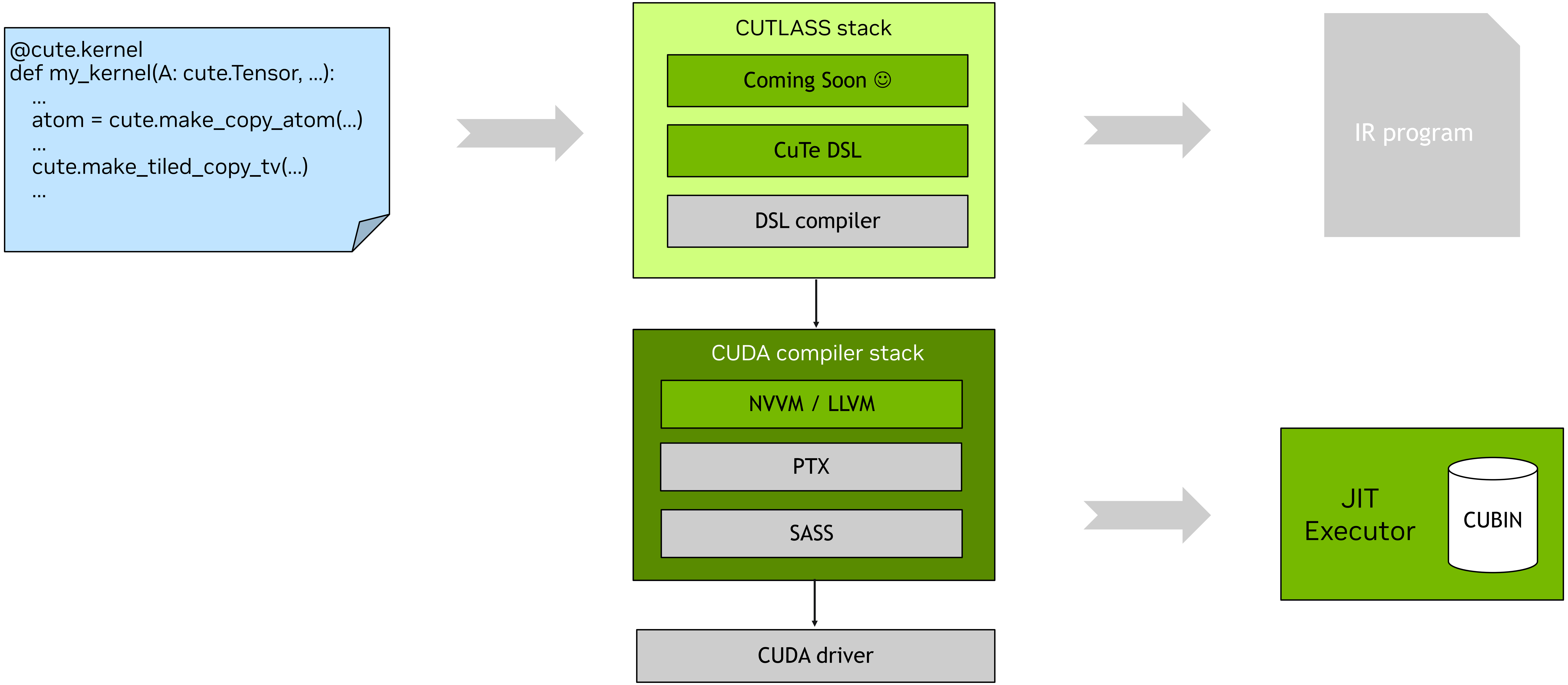
Where we start with for CUTLASS Python

- **A Python based programming language to program Tensor Cores with CuTe semantics for best possible performance**

- Enables kernel authoring in Python rather than just accessing CUTLASS kernels
- Empowered by CuTe abstractions
- Easy integration with popular Python frameworks, like Pytorch
- Model hardware accurately for full control of performance
- Based on MLIR framework to leverage the power of MLIR ecosystem

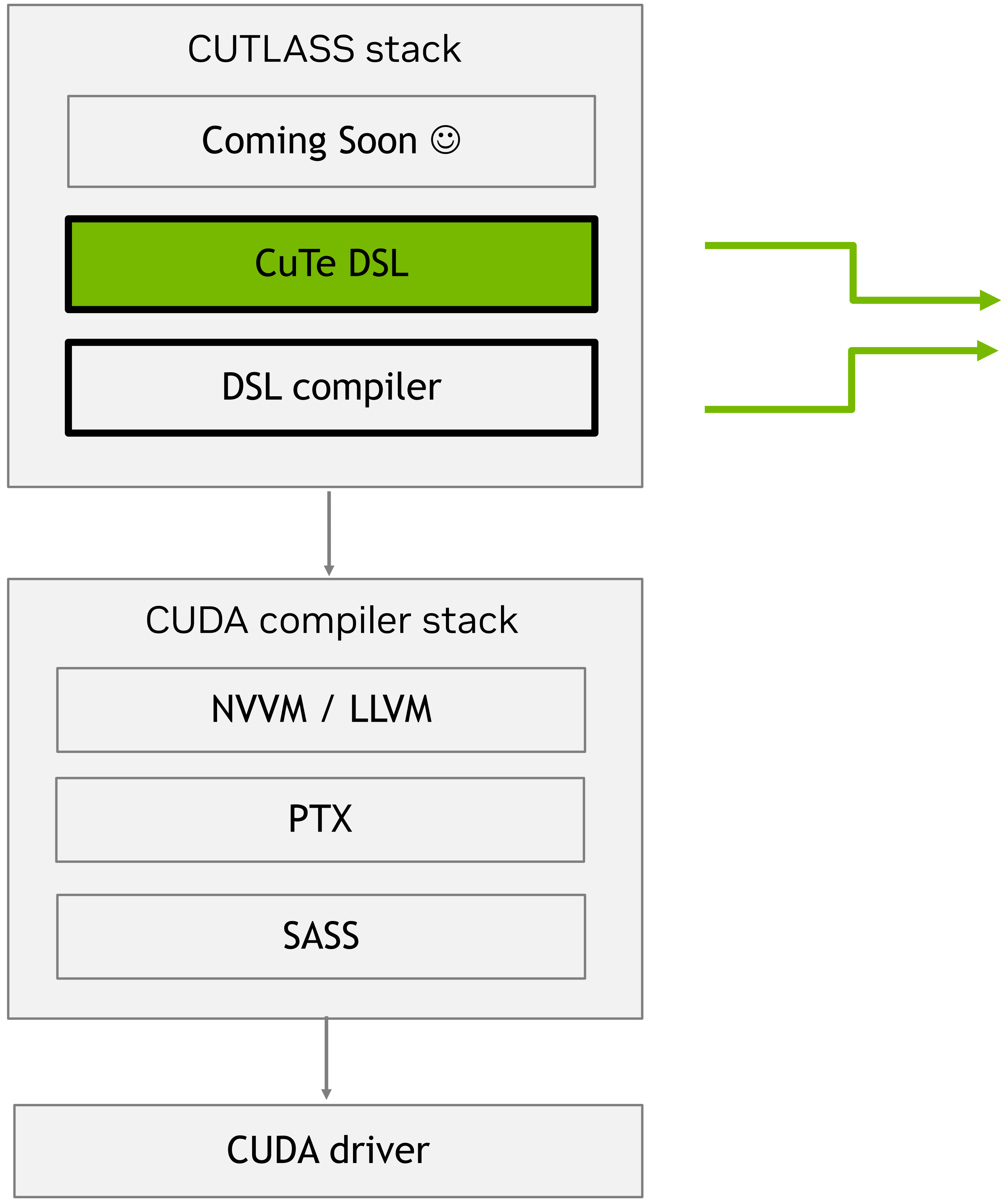


CUTLASS Python architecture



```
@cute.kernel
def my_kernel(A: cute.Tensor, ...):
    ...
    atom = cute.make_copy_atom(...)
    ...
    cute.make_tiled_copy_tv(...)
    ...
```

Tensor Core programming in Python



Initial release to include CuTe support

Writing a kernel in Python

@cute.jit / @cute.kernel

```cpp
template <class ProblemShape, class CtaTiler,
          class TA, class SmemLayoutA, class TmaA,
          class TB, class SmemLayoutB, class TmaB,
          class TC, class CStride, class TiledMma,
          class Alpha, class Beta>
__global__ static
__launch_bounds__(decltype(size(TiledMma{}))::value)
void
gemm_device(ProblemShape shape_MNK, CtaTiler cta_tiler,
            TA const* A, CUTLASS_GRID_CONSTANT TmaA const tma_a,
            TB const* B, CUTLASS_GRID_CONSTANT TmaB const tma_b,
            TC      * C, CStride dC, TiledMma mma,
            Alpha alpha, Beta beta) {
   ...

// Kernel Launch
cutlass::Status status = cutlass::launch_kernel_on_cluster(
                                      params, kernel_ptr,
                                      prob_shape, cta_tiler,
                                      A, tmaA,
                                      B, tmaB,
                                      C, dC, tiled_mma,
                                      alpha, beta);
```

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

@cute.kernel
def kernel(self, tma_atom_a: cute.CopyAtom, mA_mkl: cute.Tensor,
           tma_atom_b: cute.CopyAtom, mB_nkl: cute.Tensor,
           mC_mnl: cute.Tensor,
           cluster_layout_vmnk: cute.Layout,
           a_smem_layout_staged: cute.Layout,
           b_smem_layout_staged: cute.Layout,
           epilogue_op: cutlass.Constexpr = lambda x: x,
):
    ...

@cute.jit
def __call__(self, mA: cute.Tensor, mB: cute.Tensor, mC: cute.Tensor,
             epilogue_op=lambda x: x,
):
    ...

    # Launch the kernel
    self.kernel(...)
```

Easy integration: Interop with Pytorch

Support DLPack protocol

```python
import cutlass
import cutlass.cute as cute
from cutlass.cute.runtime import from_dlpack
import torch


@cute.kernel
def jit_kernel(A: cute.Tensor):
    ...


@cute.jit
def jit_func(A: cute.Tensor):
    jit_kernel(
        A, config=cutlass.LaunchConfig(grid=[1, 1, 1], block=[1, 1,
1])
    )
```

```python
A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(A_tensor)
# Or jit_func(from_dlpack(A_tensor).mark_layout_dynamic())
```

- Take torch.tensor as input seamlessly
- Finer grained control with explicit call

Easy integration: Interop with Pytorch

Support DLPack protocol

```
// Create instantiation for device reference gemm kernel
cutlass::reference::device::Gemm<ElementInputA,
                                 LayoutInputA,
                                 ElementInputB,
                                 LayoutInputB,
                                 ElementOutput,
                                 LayoutOutput,
                                 ElementComputeEpilogue,
                                 ElementComputeEpilogue>
    gemm_device;
// Launch device reference gemm kernel
gemm_device(problem_size,
            alpha,
            tensor_a.device_ref(),
            tensor_b.device_ref(),
            beta,
            tensor_c.device_ref(),
            tensor_ref_d.device_ref());
// Wait for kernels to finish
cudaDeviceSynchronize();
// Copy output data from CUTLASS and reference kernel to host for comparison
tensor_d.sync_host();
tensor_ref_d.sync_host();
// Check if output from CUTLASS kernel and reference kernel are equal or not
bool passed = cutlass::reference::host::TensorEquals(
  tensor_d.host_view(),
  tensor_ref_d.host_view());
std::cout << (passed ? "Passed" : "Failed") << std::endl;
```

```
ref_c = (torch.einsum("mkl,nkl->mnl", a_ref, b_ref)).cpu()
torch.testing.assert_close(gpu_c, ref_c, atol=tolerance, rtol=1e-05)
```

Easy integration: Interop with Pytorch

From static layout to dynamic layout

```
@cute.kernel
def jit_kernel(A: cute.Tensor, x: cutlass.Int32, y: cutlass.Int32):
    A[x] = y


@cute.jit
def jit_func(A: cute.Tensor):
    x = 0
    y = 3
    jit_kernel(
        A, x, y, config=cutlass.LaunchConfig(grid=[1, 1, 1],
                                             block=[1, 1, 1])
    )


A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(from_dlpack(A_tensor))
```

- cute.Tensor type would have a layout (3:1) that follows the size of input A_tensor

Easy integration: Interop with Pytorch

From static layout to dynamic layout

```python
@cute.kernel
def jit_kernel(A: cute.Tensor, x: cutlass.Int32, y: cutlass.Int32):
    A[x] = y


@cute.jit
def jit_func(A: cute.Tensor):
    x = 0
    y = 3
    jit_kernel(
        A, x, y, config=cutlass.LaunchConfig(grid=[1, 1, 1],
                                             block=[1, 1, 1])
    )


A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(from_dlpack(A_tensor))

B_tensor = torch.tensor([0, 0, 0, 0, 0], dtype=torch.int32).cuda()
jit_func(from_dlpack(B_tensor))
```

- cute.Tensor type would have a layout (5:1) that follows the size of input B_tensor

- Two sets of JIT functions are compiled:
  - One with cute.Tensor type of layout (3:1)
  - One with cute.Tensor type of layout (5:1)

- Static layout results in distinct codes

Easy integration: Interop with Pytorch

From static layout to dynamic layout

```
@cute.kernel
def jit_kernel(A: cute.Tensor, x: cutlass.Int32, y: cutlass.Int32):
    A[x] = y


@cute.jit
def jit_func(A: cute.Tensor):
    x = 0
    y = 3
    jit_kernel(
        A, x, y, config=cutlass.LaunchConfig(grid=[1, 1, 1],
                                             block=[1, 1, 1])
    )


A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(from_dlpack(A_tensor).mark_layout_dynamic(mode=[0]))

B_tensor = torch.tensor([0, 0, 0, 0, 0], dtype=torch.int32).cuda()
jit_func(B_tensor)
```

- cute.Tensor type would have a dynamic layout (?:1)
- Only one set of JIT functions compiled for both cases
- Use dynamic layout to allow generalized code generation

Easy integration: Interop with Pytorch

Integration with LLaMA 8b

- Wire up the gate/up/down projection layer with the customized linear module

```
class LlamaMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.intermediate_size
        self.gate_proj = nn.Linear(
            self.hidden_size,
            self.intermediate_size,
            bias=config.mlp_bias
        )
        self.up_proj = nn.Linear(
            self.hidden_size,
            self.intermediate_size,
            bias=config.mlp_bias
        )
        self.down_proj = nn.Linear(
            self.intermediate_size,
            self.hidden_size,
            bias=config.mlp_bias
        )
        self.act_fn = ACT2FN[config.hidden_act]
```

```
class LlamaMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.intermediate_size
        self.gate_proj = MyCutlassLinear(
            self.hidden_size,
            self.intermediate_size,
            bias=config.mlp_bias
        )
        self.up_proj = MyCutlassLinear(
            self.hidden_size,
            self.intermediate_size,
            bias=config.mlp_bias
        )
        self.down_proj = MyCutlassLinear(
            self.intermediate_size,
            self.hidden_size,
            bias=config.mlp_bias
        )
        self.act_fn = ACT2FN[config.hidden_act]
```

Source: [*https://github.com/huggingface/transformers/blob/main/src/transformers/models/llama/modeling_llama.py*](https://github.com/huggingface/transformers/blob/main/src/transformers/models/llama/modeling_llama.py)

Easy integration: Interop with Pytorch

Integration with LLaMA 8b

```
class MyCutlassLinear(nn.Module):
    def __init__(self, in_features, out_features, bias=False):
        ...
        from blackwell.gemm import MyGemmKernel
        self.gemm = MyGemmKernel(
            cutlass.Float16,
            cutlass.Float32,
            cutlass.Float16,
            False,                # 2-CTA optimization
            (128, 128),           # MMA tile shape
            (2, 1, 1),            # cluster shape
            True)                 # Use TMA

    def forward(self, input, bias=None):
        batch_size, seq_len, hidden_size = input.shape
        try:
            input = input.reshape(batch_size * seq_len, hidden_size)
            output = torch.empty(
                input.size(0), self.out_features, device=input.device,
                                                  dtype=input.dtype)
            ...

            self.gemm(
                input.detach().contiguous(),
                weight.detach().contiguous(),
                output.contiguous(),
                stream)
            ...
```

- Setup the linear module with your customized kernel implemented by CUTLASS Python APIs

- Invoke your kernel for the forward pass
- Leverage implicit from_dlpack conversion

Metaprogramming with an imperative style

Python to Python code generation: dynamic if

- Pythonic way to write “meta-kernel” that automatically fits for dynamic layout
  - Conditional execution based on dynamic expression

```
@cute.kernel
def jit_kernel(A: cute.Tensor, x: cutlass.Int32, y: cutlass.Int32):

    # Conditional on dynamic value
    if x < cute.size(A):
        A[x] = y
    else:
        ...


@cute.jit
def jit_func(A: cute.Tensor):
    x = 0
    y = 3
    jit_kernel(
        A, x, y, config=cutlass.LaunchConfig(grid=[1, 1, 1],
                                             block=[1, 1, 1])
    )

A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(A_tensor)
```

This will be converted to a dynamic if automatically which will be executed at runtime

Metaprogramming with an imperative style

Python to Python code generation: dynamic loop

- Pythonic way to write “meta-kernel” that automatically fits for dynamic layout
  - Loop on dynamic expression

```
@cute.kernel
def jit_kernel(A: cute.Tensor, x: cutlass.Int32, y: cutlass.Int32):

    # Loop on dynamic value
    for i in range(cute.size(A)):
        ...

    # Loop on dynamic value with loop unrolling
    for i in range_dynamic(cute.size(A), unroll=1):
        ...
```

Both will be converted to dynamic loop automatically which will be executed at runtime

```
@cute.jit
def jit_func(A: cute.Tensor):
    x = 0
    y = 3
    jit_kernel(
        A, x, y, config=cutlass.LaunchConfig(grid=[1, 1, 1],
                                             block=[1, 1, 1])
    )

A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(A_tensor)
```

Metaprogramming with an imperative style

Compile-time constants as Constexpr

- Pythonic way to write “meta-kernel” that automatically fits for dynamic shape
  - Use Constexpr for constants known at compile-time

```
@cute.kernel
def jit_kernel(A: cute.Tensor, x: cutlass.Int32, y: cutlass.Int32 , z: cutlass.Constexpr[int]):

    # Loop on compile-time constant
    for i in range(z):
        A[x + i] = y
```

No conversion for compile-time constants

```
@cute.jit
def jit_func(A: cute.Tensor):
    x = 0
    y = 3
    z = 3
    jit_kernel(
        A, x, y, z, config=cutlass.LaunchConfig(grid=[1, 1, 1],
                                                block=[1, 1, 1]))

A_tensor = torch.tensor([0, 0, 0], dtype=torch.int32).cuda()
jit_func(A_tensor)
```

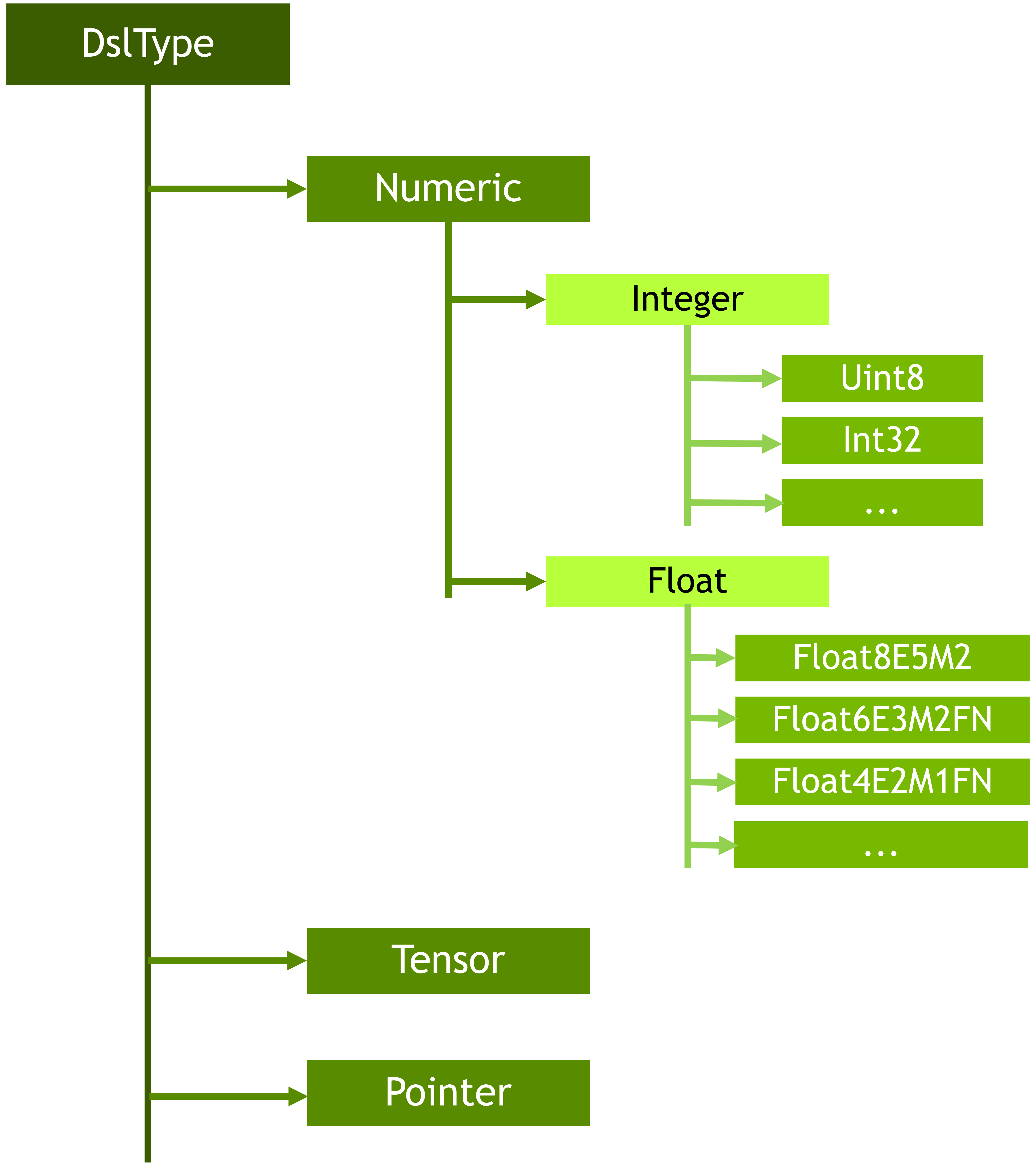
Metaprogramming with an imperative style

TiledCopy and SMEM layout as kernel parameters

```
using CollectiveOp = cutlass::gemm::collective::CollectiveMma<
      DispatchPolicy,
      TileShape_MNK,
      ElementA,
      cute::tuple<cutlass::gemm::TagToStrideA_t<GmemLayoutATag>,
                  cutlass::gemm::TagToStrideA_t<GmemLayoutSFATag>>,
      ElementB,
      cute::tuple<cutlass::gemm::TagToStrideB_t<GmemLayoutBTag>,
                  cutlass::gemm::TagToStrideB_t<GmemLayoutSFBTag>>,
      TiledMma,
      GmemTiledCopyA,
      SmemLayoutAtomA,
      void,
      cute::identity,
      GmemTiledCopyB,
      SmemLayoutAtomB,
      void,
      cute::identity
    >;
```

```
@cute.kernel
def kernel(
    self,
    mA: cute.Tensor,
    mB: cute.Tensor,
    mC: cute.Tensor,
    sA_layout: cute.Layout,
    sB_layout: cute.Layout,
    tiled_copy_A: cute.TiledCopy,
    tiled_copy_B: cute.TiledCopy,
    tiled_mma: cute.TiledMma,
    epilogue_op: cutlass.Constexpr = lambda x: x,
):
    ...
```

Data types modeling to generate fast code



- Primitive types
  - Full support of CUTLASS data types
  - One type for all contexts
    - Host/device JIT function
    - Non-JIT context
  - Easy type access and conversions with CuTe DSL type
    - torch.dtype
    - numpy.dtype
- Compound types
  - Tensor
    - Modeling CuTe tensor concept
  - Pointer
    - Modeling the raw pointer to memory location

Better code expressiveness and readability

Operator overloading for arithmetic, comparison and bitwise

```
@cute.jit
def jit_func(x: cutlass.Int32, y: cutlass.Int32):
    # basic arithmetic
    x + y
    x - y
    x * y
    x // y
    x ** y
    x % y

    x >= y
    x < y

    x & y
    x | y
    x ^ y
    x << y
    x >> y
    ~x

    ...
```

- Pythonic way to deal with DslTyped values

✗ Avoid tedious spelling like **arith.muli(a, arith.constant(a.type, 4))**

✓ Instead, simply write **a * 4** and let the DSL take care of it for you

Better code expressiveness and readability

Operator overloading with vectorization

```python
@cute.kernel
def sgemm_kernel(
    mA: cute.Tensor,
    mB: cute.Tensor,
    mC: cute.Tensor,
    ...
):
    ...
    # Activation function fusion
    tCrC = cute.make_tensor_like(tAcc)
    for i in range_dynamic(cute.size(tAcc)):
       a = tAcc[i]
       tCrC[i] = if a > 0 a else a.dtype(0)
    ...
```

```python
@cute.kernel
def sgemm_kernel(
    ...
    epilogue_op: cutlass.Constexpr = lambda x: x,
):
    ...
    # Activation function fusion
    tCrC.store(epilogue_op(tCrC.load()))
    ...

# Fused GEMM and ReLU
gemm(input.detach(),
     weight.detach(),
     output,
     epilogue_op=lambda x: cute.where(x > 0, x, cute.full_like(x, 0)),
)
```

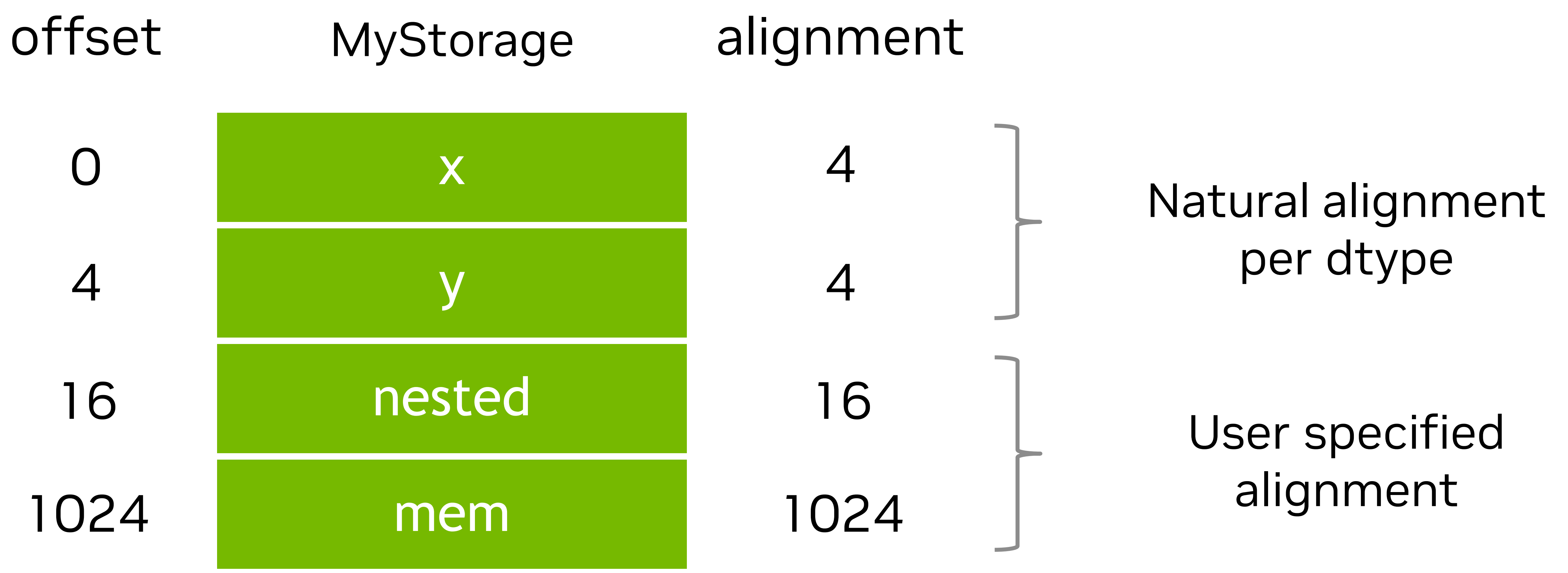
- **TensorSSA**: thread local data modeling for CuTe Tensor in value semantics and immutable
  - Vector based with nested CuTe shape support
  - Load tensor elements as vector / store vector data into tensor
  - Operator overloading for vectorized operations

Customized C struct like data types

@cute.struct

- 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

```
@cute.struct
class complex:
    real: cutlass.Float32
    imag: cutlass.Float32

@cute.struct
class MyStorage:
    x: cutlass.Float32
    y: cutlass.Int32
    nested: cute.struct.align(complex, 16)
    mem: cute.struct.align(cute.struct.MemRange(
            cutlass.Float32, cute.cosize(layout_a)), 1024)
```



offset	MyStorage	alignment
0	x	4
4	y	4
16	nested	16
1024	mem	1024

Natural alignment per dtype

User specified alignment

Kernel writing in an OOP manner

@cute.struct

```
@cute.kernel
def kernel(...):
    ...
    smem = cutlass.utils.SmemAllocator()
    ab_full_mbar_ptr = smem.allocate_array(cutlass.Int64,
                                           self.ab_stage)
    ab_empty_mbar_ptr = smem.allocate_array(cutlass.Int64,
                                            self.ab_stage)
    buffer_align_bytes = 1024
    sa = smem.allocate_tensor(
         self.a_dtype,
         a_smem_layout_staged.outer,
         buffer_align_bytes,
         swizzle=a_smem_layout_staged.inner)
    sb = ...
    ...
def _compute_smem(cta_tile_shape_mnk, a_dtype, b_dtype, ab_stage, c_dtype):
        a_shape = cute.slice_(cta_tile_shape_mnk, (None, 0, None))
        b_shape = cute.slice_(cta_tile_shape_mnk, (0, None, None))
        ab_bytes_per_stage = (
            cute.size(a_shape) * a_dtype.width // 8
            + cute.size(b_shape) * b_dtype.width // 8)
        mbar_helpers_bytes = 1024
        num_smem_bytes = ab_bytes_per_stage * ab_stage + mbar_helpers_bytes
        return num_smem_bytes

self.kernel(...,
            config=cutlass.LaunchConfig(
                ...
                smem=self._compute_smem(...),
                async_deps=[stream]))
```

```
@cute.kernel
def kernel(...):
    ...
    smem = cutlass.utils.SmemAllocator()
    storage = smem.allocate_struct(self.shared_storage)
    # access ab_full_mbar_ptr through struct data members
    ab_full_mbar_ptr = storage.ab_full_mbar_ptr.data_ptr()
    ...

@cute.struct
class SharedStorage:
    ab_full_mbar_ptr: cute.struct.MemRange(cutlass.Int64,
                                           self.ab_stage)
    ab_empty_mbar_ptr: cute.struct.MemRange(cutlass.Int64,
                                             self.ab_stage)
    sa: ...
    sb: ...

self.kernel(...,
            config=cutlass.LaunchConfig(
                ...
                smem=SharedStorage.size_in_bytes(),
                async_deps=[stream]))
```

Reduced kernel launching latency with caching

Significant overhead without caching

```python
class MyGemmKernel:
    @cute.jit
    def __call__(
        self,
        a: cute.Tensor,
        b: cute.Tensor,
        c: cute.Tensor,
        stream: cutlass.Stream,
        epilogue_op: cutlass.Constexpr = lambda x: x,
    ):
        ...

def forward(self, input, bias=None):
    batch_size, seq_len, hidden_size = input.shape
    try:
        # Reshape input to prepare for GEMM
        input = input.reshape(batch_size * seq_len, hidden_size)
        output = torch.empty(
            input.size(0), self.out_features, device=input.device,
                                              dtype=input.dtype)
        ...

        self.gemm(
            input.detach().contiguous(),
            weight.detach().contiguous(),
            output.contiguous(),
            stream)
        ...
```

- In each forward pass, the method will be called with kernel JIT compilation which would cause a significant runtime overhead

Reduced kernel launching latency with caching

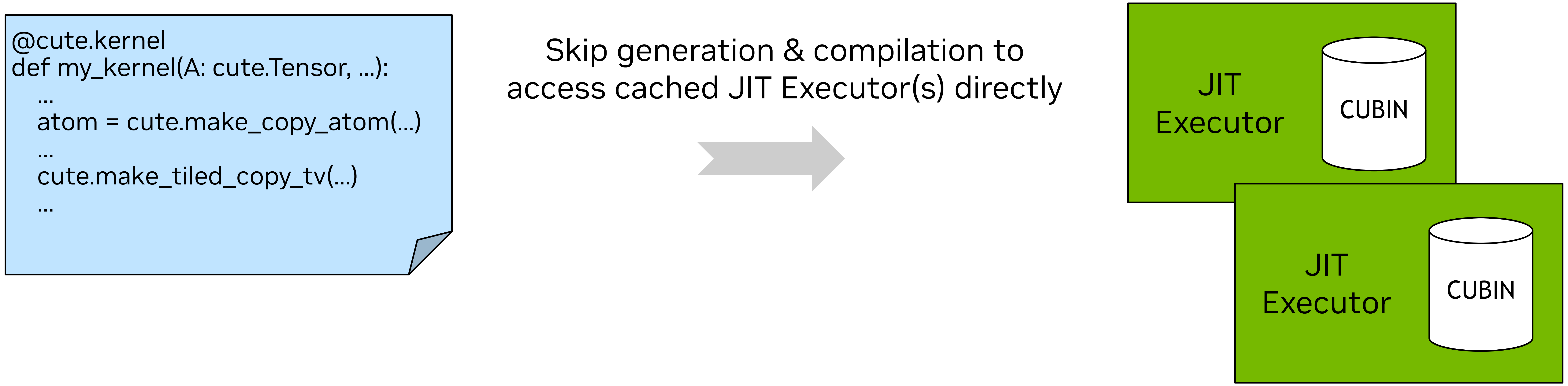
Zero Compile: JIT Executor with CUBIN cached

```python
def forward(self, input, bias=None):
    batch_size, seq_len, hidden_size = input.shape
    try:
        # Reshape input to prepare for GEMM
        input = input.reshape(batch_size * seq_len, hidden_size)
        output = torch.empty(
            input.size(0), self.out_features, device=input.device, dtype=input.dtype)
        ...

        input_tensor = from_dlpack(input.detach().contiguous()).mark_layout_dynamic()
        weight_tensor = from_dlpack(weight.detach().contiguous()).mark_layout_dynamic()
        output_tensor = from_dlpack(output.contiguous()).mark_layout_dynamic()

        key = input.shape
        if key not in self.cached_kernels:
            self.cached_kernels[key] = cute.compile(
                self.gemm,
                input_tensor,
                weight_tensor,
                output_tensor,
                cutlass_stream,
        )
        self.cached_kernels[key](
            input_tensor, weight_tensor, output_tensor, cutlass_stream
        )
        ...
```

- Customized keys for kernel caching
- JIT Executor with CUBIN cached
- JIT Executor supports serialization to file and deserialization from file

Reduced kernel launching latency with caching

Zero Compile: JIT Executor with CUBIN cached



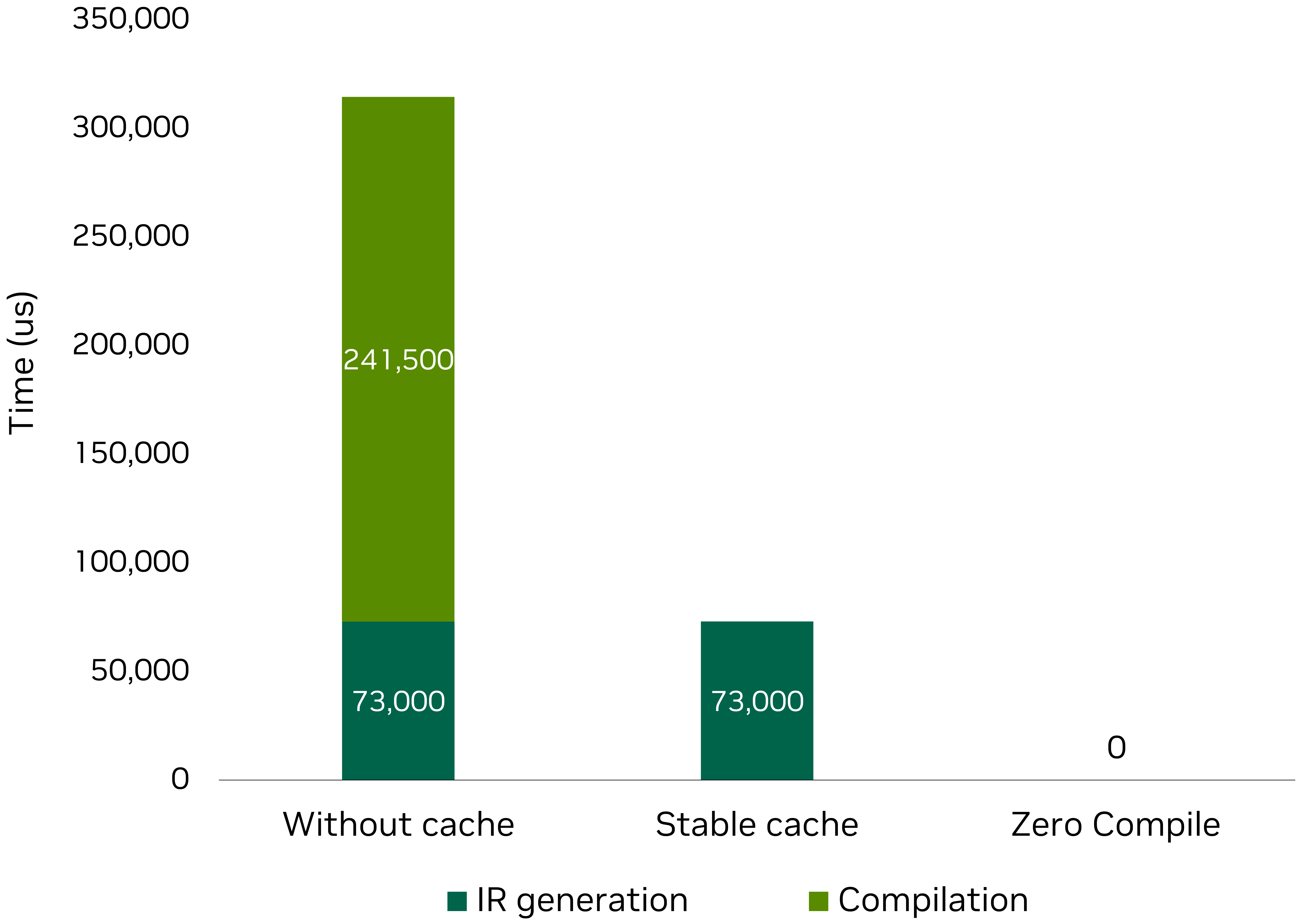
```
@cute.kernel
def my_kernel(A: cute.Tensor, …):
   …
   atom = cute.make_copy_atom(…)
   …
   cute.make_tiled_copy_tv(…)
   …
```

Skip generation & compilation to access cached JIT Executor(s) directly

Generation, Compilation, and Launch overhead

Blackwell B100 FP16 GEMM: M=N=K=8K

- **Without cache**
  - Goes all the way down to kernel launching
  - Includes IR generation, compilation and kernel launch
- **Stable cache**
  - By default ON
  - Always generates codes for functional correctness
  - Skips compilation if generated codes are identical
- **Zero Compile**
  - Launches kernel directly through JIT Executor managed by users
  - No IR generation and compilation
  - Minimized kernel launch overhead at ~4.6 us



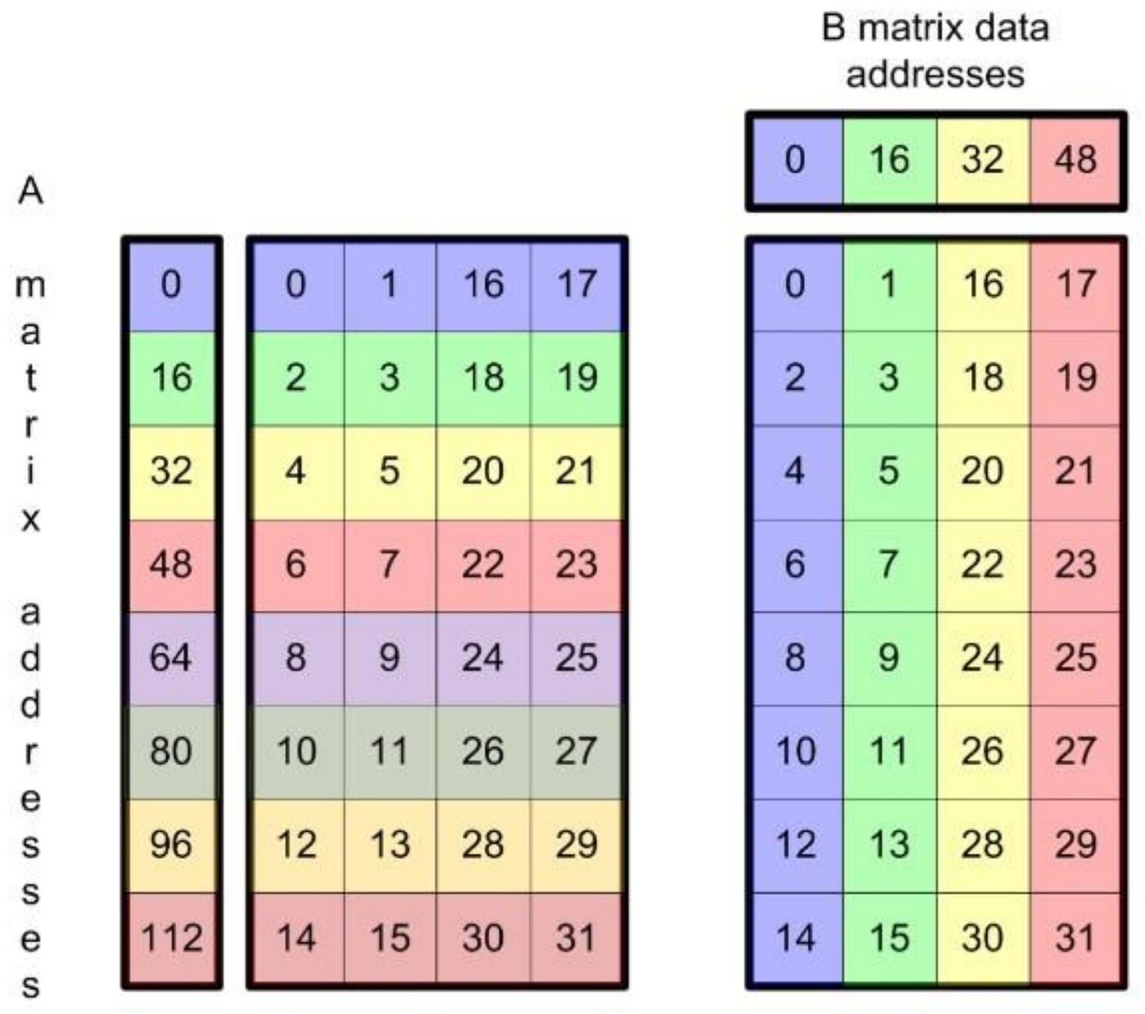
Agenda

- Introduction and Motivations
- The DSL Infrastructure
- Kernel Authoring in Python with CuTe
- Runtime Performance
- Conclusion

What is CuTe and why should you care about it

Program GPUs at peak throughout architectural generations

- What's needed for performance
  - Complex tiling and partitioning patterns
  - Arch-specific instructions with their own set of requirements
- Layout book-keeping is hard, error-prone, and takes away time
- CuTe at your rescue without taking away control!
  - A single Layout concept to express all layouts of interest and more based on a hierarchical representation
  - A formal algebra
  - A programming model maintaining logical consistency
  - A consistent set of idioms applicable throughout GPU generations
  - A safer low-level programming
    - Detect illegal patterns by inspecting layouts at compile-time



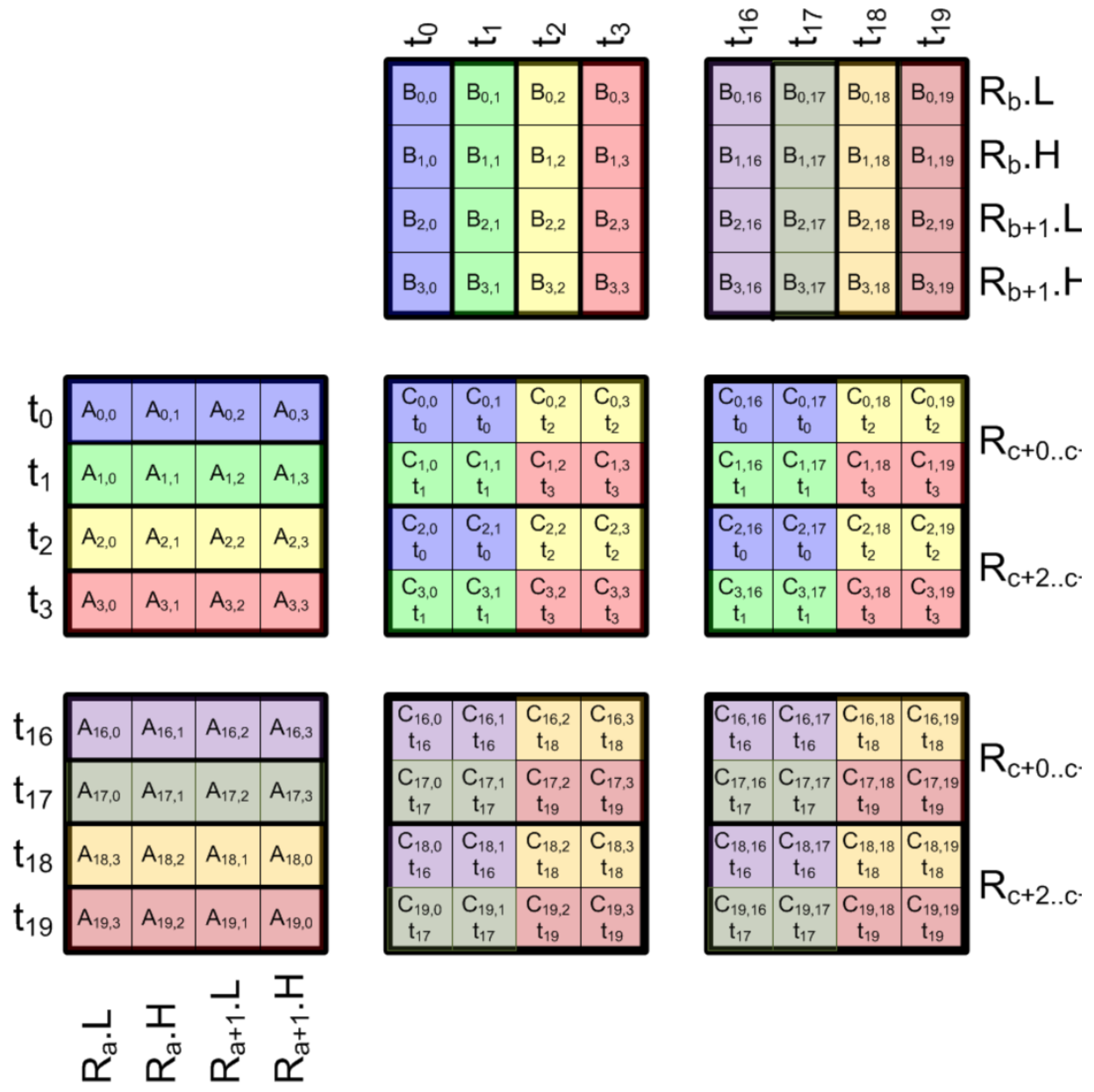
B matrix data addresses: 0 | 16 | 32 | 48

A matrix addresses	Maxwell A access				Maxwell B access			
0	0	1	16	17	0	1	16	17
16	2	3	18	19	2	3	18	19
32	4	5	20	21	4	5	20	21
48	6	7	22	23	6	7	22	23
64	8	9	24	25	8	9	24	25
80	10	11	26	27	10	11	26	27
96	12	13	28	29	12	13	28	29
112	14	15	30	31	14	15	30	31

Maxwell A access addr[tid] = addr[tid^1]

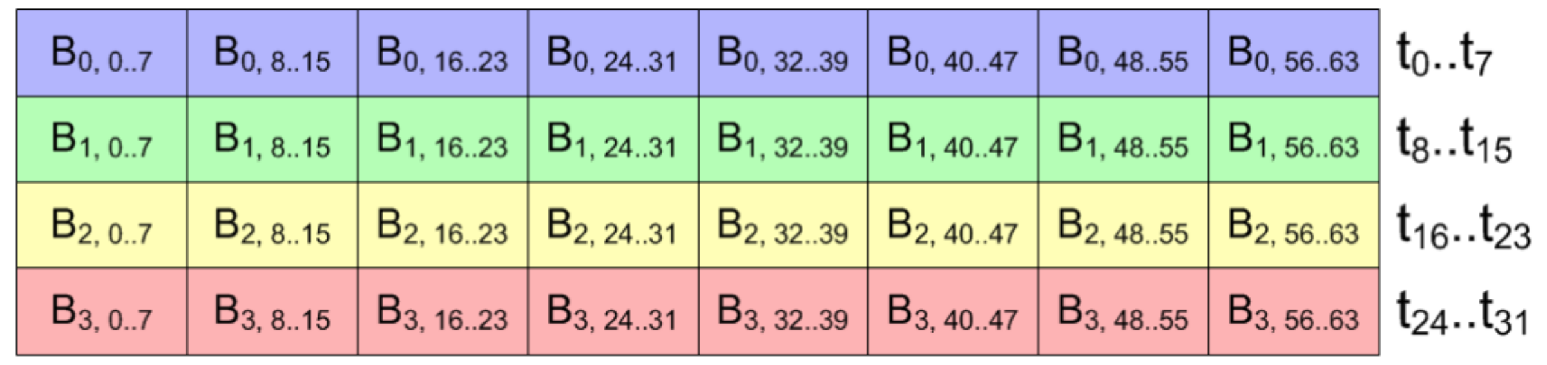
Maxwell B access addr[tid] = addr[tid^2]

Figure 3b: Maxwell thread assignments for LDS.U matching



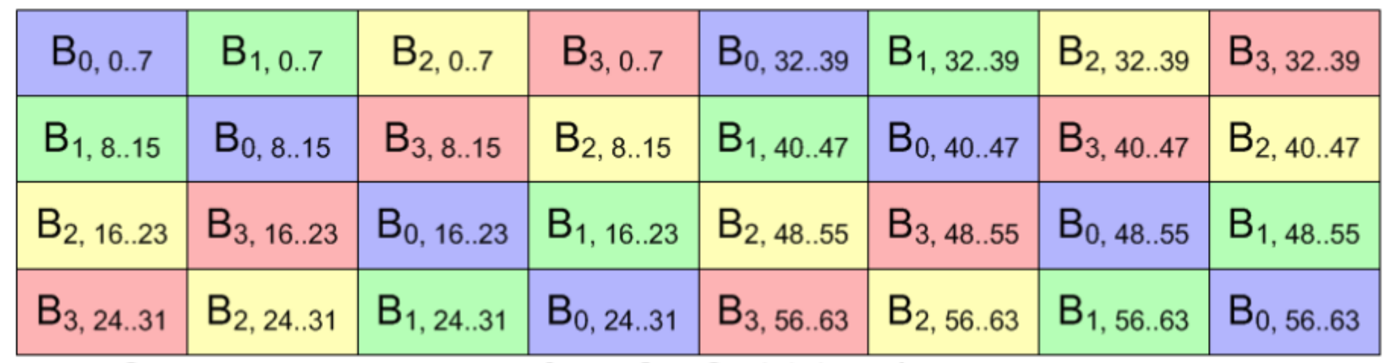
t_0	t_1	t_2	t_3	t_{16}	t_{17}	t_{18}	t_{19}	
$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{0,16}$	$B_{0,17}$	$B_{0,18}$	$B_{0,19}$	R_b.L
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{1,16}$	$B_{1,17}$	$B_{1,18}$	$B_{1,19}$	R_b.H
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{2,16}$	$B_{2,17}$	$B_{2,18}$	$B_{2,19}$	R_{b+1}.L
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$	$B_{3,16}$	$B_{3,17}$	$B_{3,18}$	$B_{3,19}$	R_{b+1}.H

	R_a.L	R_a.H	R_{a+1}.L	R_{a+1}.H									
t_0	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$C_{0,0}$ t_0	$C_{0,1}$ t_0	$C_{0,2}$ t_2	$C_{0,3}$ t_2	$C_{0,16}$ t_0	$C_{0,17}$ t_0	$C_{0,18}$ t_2	$C_{0,19}$ t_2	$R_{c+0..c}$
t_1	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$C_{1,0}$ t_1	$C_{1,1}$ t_1	$C_{1,2}$ t_3	$C_{1,3}$ t_3	$C_{1,16}$ t_1	$C_{1,17}$ t_1	$C_{1,18}$ t_3	$C_{1,19}$ t_3	
t_2	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$C_{2,0}$ t_0	$C_{2,1}$ t_0	$C_{2,2}$ t_2	$C_{2,3}$ t_2	$C_{2,16}$ t_0	$C_{2,17}$ t_0	$C_{2,18}$ t_2	$C_{2,19}$ t_2	$R_{c+2..c}$
t_3	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$C_{3,0}$ t_1	$C_{3,1}$ t_1	$C_{3,2}$ t_3	$C_{3,3}$ t_3	$C_{3,16}$ t_1	$C_{3,17}$ t_1	$C_{3,18}$ t_3	$C_{3,19}$ t_3	
t_{16}	$A_{16,0}$	$A_{16,1}$	$A_{16,2}$	$A_{16,3}$	$C_{16,0}$ t_{16}	$C_{16,1}$ t_{16}	$C_{16,2}$ t_{18}	$C_{16,3}$ t_{18}	$C_{16,16}$ t_{16}	$C_{16,17}$ t_{16}	$C_{16,18}$ t_{18}	$C_{16,19}$ t_{18}	$R_{c+0..c}$
t_{17}	$A_{17,0}$	$A_{17,1}$	$A_{17,2}$	$A_{17,3}$	$C_{17,0}$ t_{17}	$C_{17,1}$ t_{17}	$C_{17,2}$ t_{19}	$C_{17,3}$ t_{19}	$C_{17,16}$ t_{17}	$C_{17,17}$ t_{17}	$C_{17,18}$ t_{19}	$C_{17,19}$ t_{19}	
t_{18}	$A_{18,3}$	$A_{18,2}$	$A_{18,1}$	$A_{18,0}$	$C_{18,0}$ t_{16}	$C_{18,1}$ t_{16}	$C_{18,2}$ t_{18}	$C_{18,3}$ t_{18}	$C_{18,16}$ t_{16}	$C_{18,17}$ t_{16}	$C_{18,18}$ t_{18}	$C_{18,19}$ t_{18}	$R_{c+2..c}$
t_{19}	$A_{19,3}$	$A_{19,2}$	$A_{19,1}$	$A_{19,0}$	$C_{19,0}$ t_{17}	$C_{19,1}$ t_{17}	$C_{19,2}$ t_{19}	$C_{19,3}$ t_{19}	$C_{19,16}$ t_{17}	$C_{19,17}$ t_{17}	$C_{19,18}$ t_{19}	$C_{19,19}$ t_{19}	



$B_{0, 0..7}$	$B_{0, 8..15}$	$B_{0, 16..23}$	$B_{0, 24..31}$	$B_{0, 32..39}$	$B_{0, 40..47}$	$B_{0, 48..55}$	$B_{0, 56..63}$	$t_0..t_7$
$B_{1, 0..7}$	$B_{1, 8..15}$	$B_{1, 16..23}$	$B_{1, 24..31}$	$B_{1, 32..39}$	$B_{1, 40..47}$	$B_{1, 48..55}$	$B_{1, 56..63}$	$t_8..t_{15}$
$B_{2, 0..7}$	$B_{2, 8..15}$	$B_{2, 16..23}$	$B_{2, 24..31}$	$B_{2, 32..39}$	$B_{2, 40..47}$	$B_{2, 48..55}$	$B_{2, 56..63}$	$t_{16}..t_{23}$
$B_{3, 0..7}$	$B_{3, 8..15}$	$B_{3, 16..23}$	$B_{3, 24..31}$	$B_{3, 32..39}$	$B_{3, 40..47}$	$B_{3, 48..55}$	$B_{3, 56..63}$	$t_{24}..t_{31}$

LDG.128 register quad ($R_{n..n+3}$)



$B_{0, 0..7}$	$B_{1, 0..7}$	$B_{2, 0..7}$	$B_{3, 0..7}$	$B_{0, 32..39}$	$B_{1, 32..39}$	$B_{2, 32..39}$	$B_{3, 32..39}$
$B_{1, 8..15}$	$B_{0, 8..15}$	$B_{3, 8..15}$	$B_{2, 8..15}$	$B_{1, 40..47}$	$B_{0, 40..47}$	$B_{3, 40..47}$	$B_{2, 40..47}$
$B_{2, 16..23}$	$B_{3, 16..23}$	$B_{0, 16..23}$	$B_{1, 16..23}$	$B_{2, 48..55}$	$B_{3, 48..55}$	$B_{0, 48..55}$	$B_{1, 48..55}$
$B_{3, 24..31}$	$B_{2, 24..31}$	$B_{1, 24..31}$	$B_{0, 24..31}$	$B_{3, 56..63}$	$B_{2, 56..63}$	$B_{1, 56..63}$	$B_{0, 56..63}$

Shared memory after STS.128 w/swizzled thread id = (tid[1:0] << 3) | (tid & 4) | (tid[4:3] ^ tid[1:0])

CuTe exposure

If you already use CUTLASS-C++, you will feel at home

- CuTe Layouts and Tensors in all their flavors
  - The algebra is available in its entirety
  - Mixed static/dynamic Layouts are fully supported
- Robust tensor programming model that users of CUTLASS-C++ are already familiar with

MMA/Copy Atoms
Encapsulate the PTX with metadata encoded using Layouts

Tiled MMA/Copy
Robust generic partitioning interface

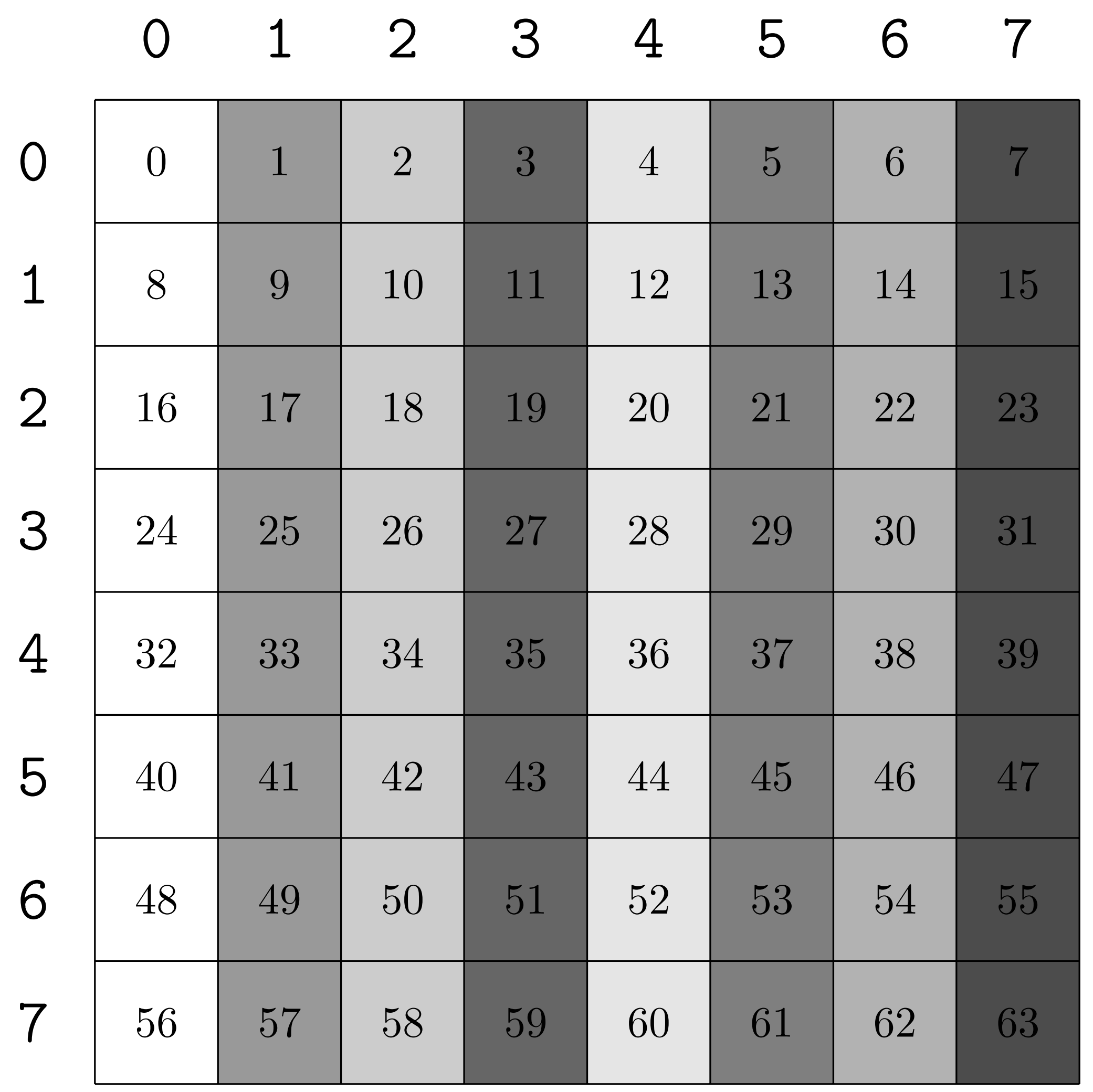
cute.{gemm|copy} algorithms
Automatically dispatch to the requested PTX instruction

CuTe Layouts and Algebra

Programming with CuTe in Python

Examples

1. Tiling MMAs
2. Ampere warp-level MMA
3. Blackwell 2CTA MMA

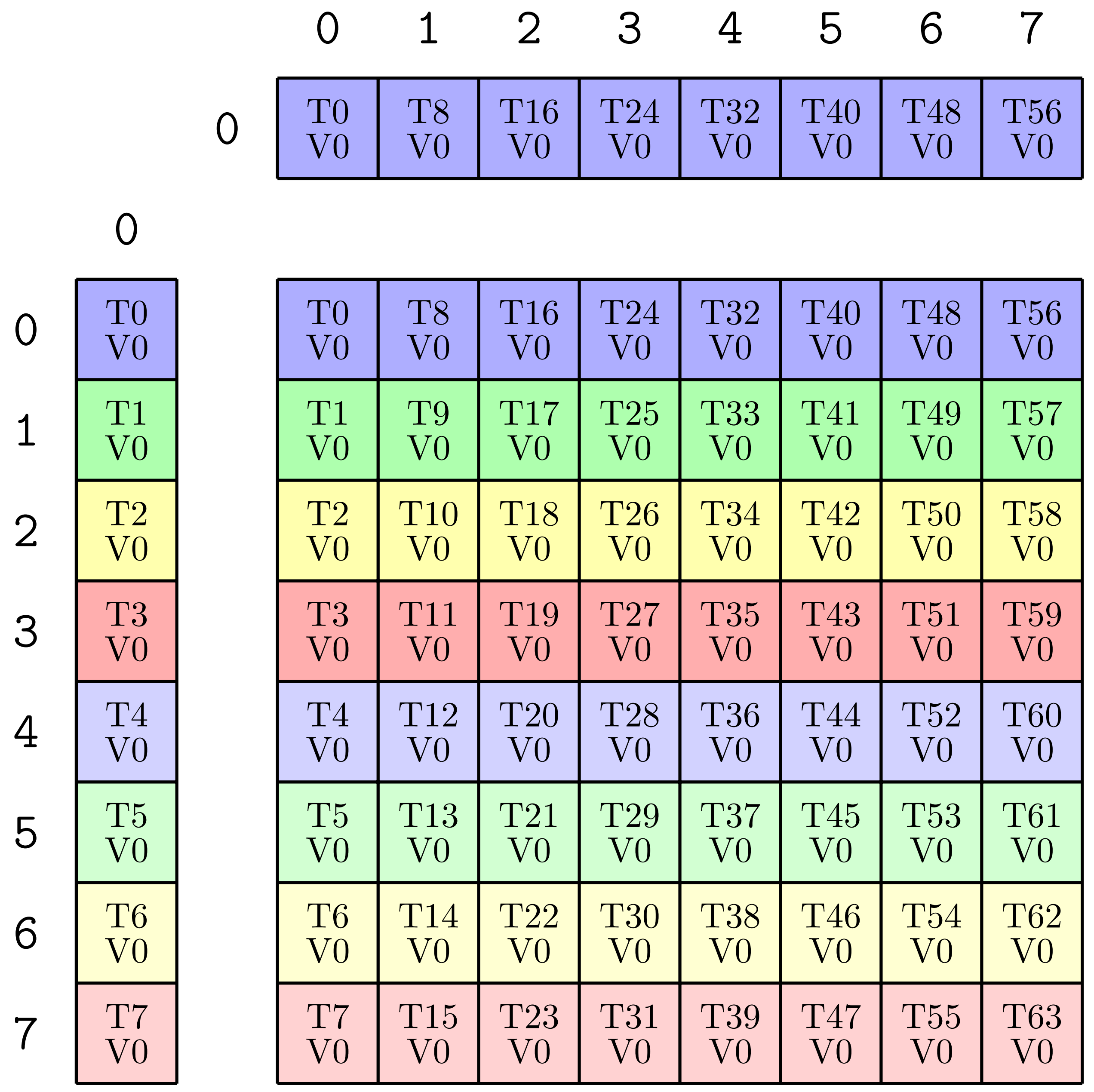


	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63

Data Layouts

`logical coord → data offset`

$$\texttt{shape:stride} = (8,8):(8,1)$$



	0	1	2	3	4	5	6	7
0	T0 V0	T8 V0	T16 V0	T24 V0	T32 V0	T40 V0	T48 V0	T56 V0

	0
0	T0 V0
1	T1 V0
2	T2 V0
3	T3 V0
4	T4 V0
5	T5 V0
6	T6 V0
7	T7 V0

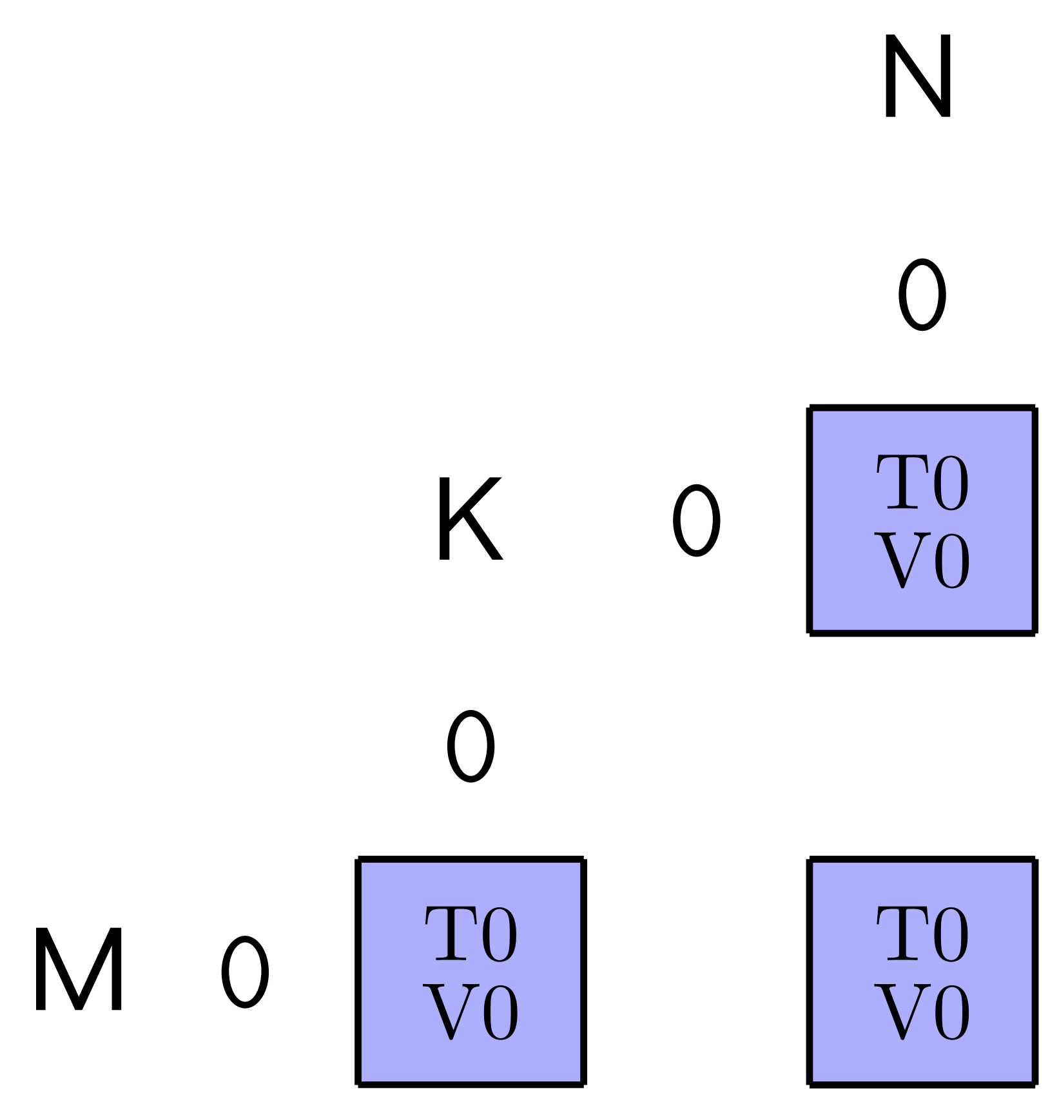
T0 V0	T8 V0	T16 V0	T24 V0	T32 V0	T40 V0	T48 V0	T56 V0
T1 V0	T9 V0	T17 V0	T25 V0	T33 V0	T41 V0	T49 V0	T57 V0
T2 V0	T10 V0	T18 V0	T26 V0	T34 V0	T42 V0	T50 V0	T58 V0
T3 V0	T11 V0	T19 V0	T27 V0	T35 V0	T43 V0	T51 V0	T59 V0
T4 V0	T12 V0	T20 V0	T28 V0	T36 V0	T44 V0	T52 V0	T60 V0
T5 V0	T13 V0	T21 V0	T29 V0	T37 V0	T45 V0	T53 V0	T61 V0
T6 V0	T14 V0	T22 V0	T30 V0	T38 V0	T46 V0	T54 V0	T62 V0
T7 V0	T15 V0	T23 V0	T31 V0	T39 V0	T47 V0	T55 V0	T63 V0

Thread-Value (TV) Layouts

`(thr,val) → logical coord`

Programming with CuTe in Python

Example 1: Building complex tiling of MMA Atoms

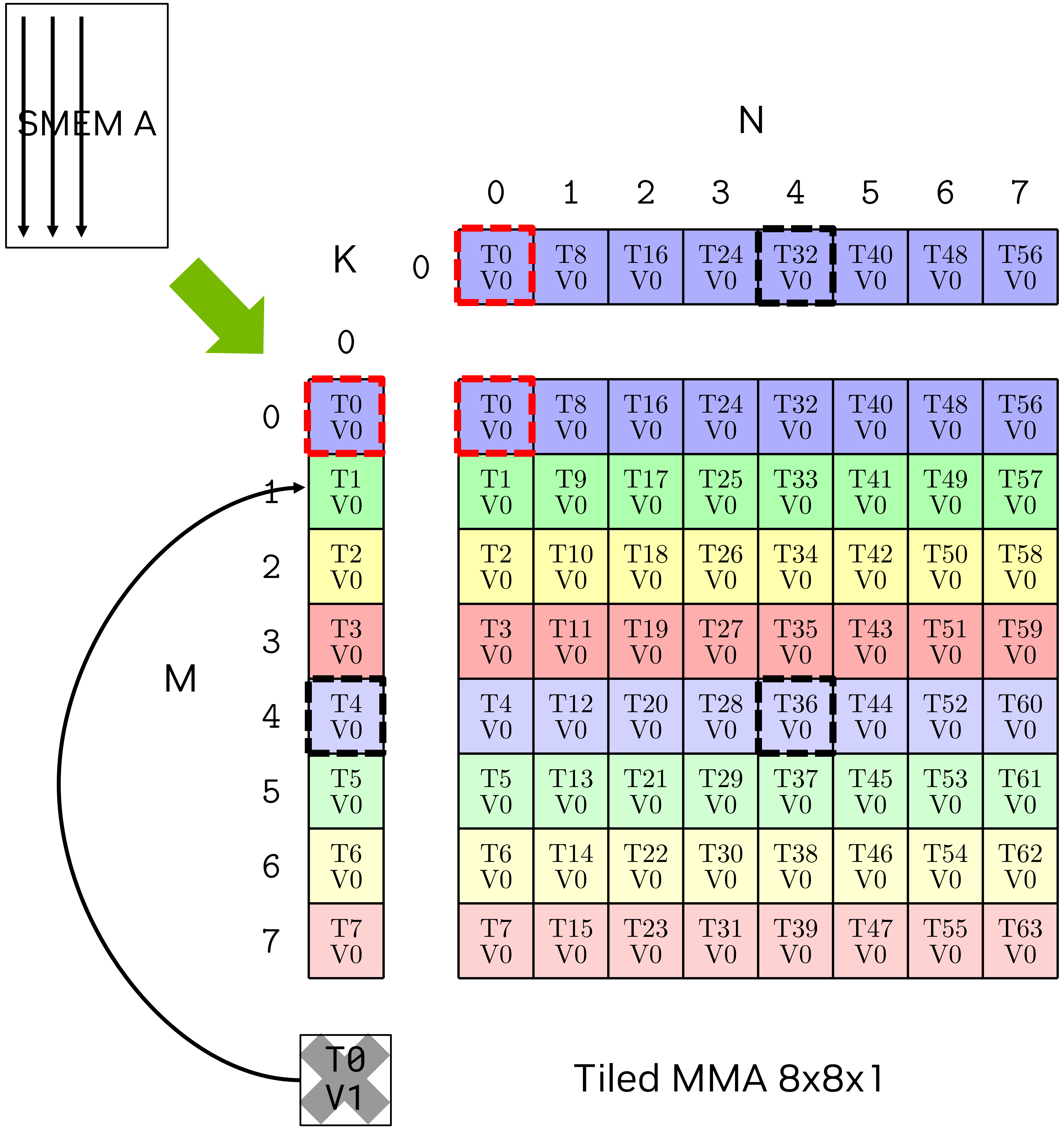


MMA Atom MxNxK=1x1x1

1x1x1 Universal MMA based on FMA

Programming with CuTe in Python

Example 1: Building complex tiling of MMA Atoms



K \ N	0	1	2	3	4	5	6	7
0	T0 V0	T8 V0	T16 V0	T24 V0	T32 V0	T40 V0	T48 V0	T56 V0

M \ K	0
0	T0 V0
1	T1 V0
2	T2 V0
3	T3 V0
4	T4 V0
5	T5 V0
6	T6 V0
7	T7 V0

M \ N	0	1	2	3	4	5	6	7
0	T0 V0	T8 V0	T16 V0	T24 V0	T32 V0	T40 V0	T48 V0	T56 V0
1	T1 V0	T9 V0	T17 V0	T25 V0	T33 V0	T41 V0	T49 V0	T57 V0
2	T2 V0	T10 V0	T18 V0	T26 V0	T34 V0	T42 V0	T50 V0	T58 V0
3	T3 V0	T11 V0	T19 V0	T27 V0	T35 V0	T43 V0	T51 V0	T59 V0
4	T4 V0	T12 V0	T20 V0	T28 V0	T36 V0	T44 V0	T52 V0	T60 V0
5	T5 V0	T13 V0	T21 V0	T29 V0	T37 V0	T45 V0	T53 V0	T61 V0
6	T6 V0	T14 V0	T22 V0	T30 V0	T38 V0	T46 V0	T54 V0	T62 V0
7	T7 V0	T15 V0	T23 V0	T31 V0	T39 V0	T47 V0	T55 V0	T63 V0

Tiled MMA 8x8x1

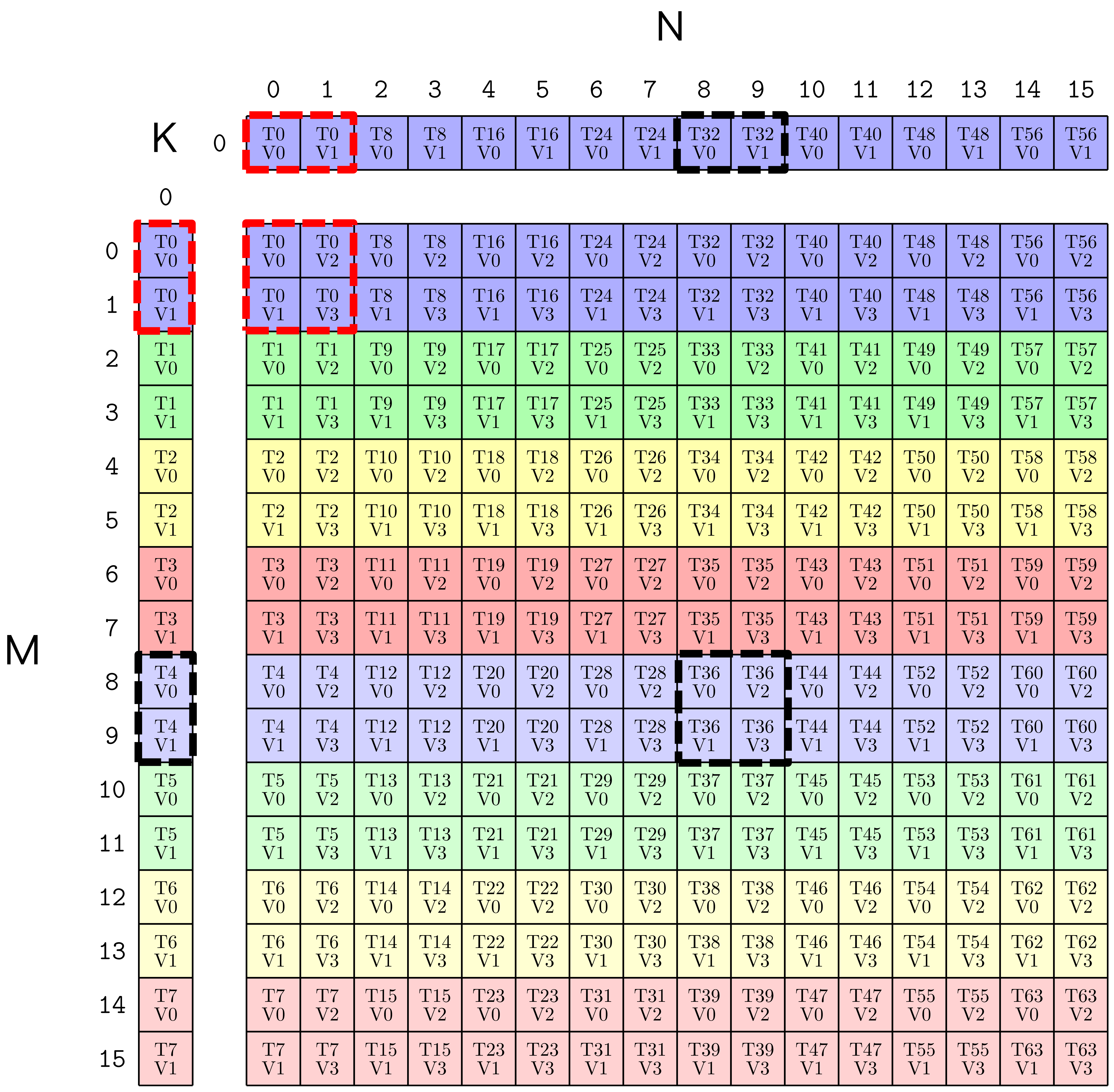
Tile this atom in an 8x8 fashion

- threads 0, 8, 16,... own in their registers the same entry of A
- threads [0,7] own in their registers the same entry of B

Description tracked by Tiled MMA with TV-Layouts

Programming with CuTe in Python

Example 1: Building complex tiling of MMA Atoms



K	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0 V0	T0 V1	T8 V0	T8 V1	T16 V0	T16 V1	T24 V0	T24 V1	T32 V0	T32 V1	T40 V0	T40 V1	T48 V0	T48 V1	T56 V0	T56 V1

M	0
0	T0 V0
1	T0 V1
2	T1 V0
3	T1 V1
4	T2 V0
5	T2 V1
6	T3 V0
7	T3 V1
8	T4 V0
9	T4 V1
10	T5 V0
11	T5 V1
12	T6 V0
13	T6 V1
14	T7 V0
15	T7 V1

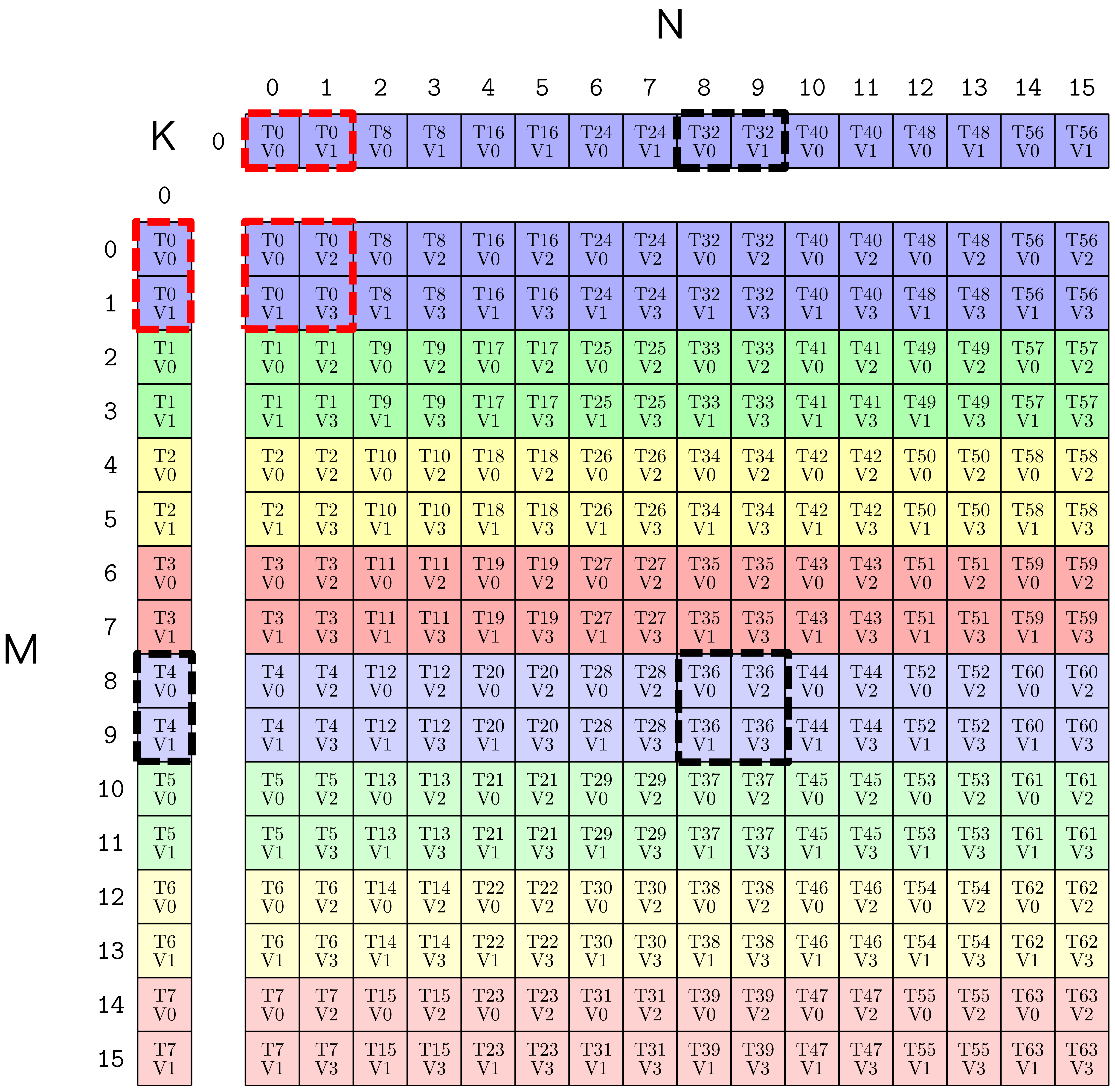
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0 V0	T0 V2	T8 V0	T8 V2	T16 V0	T16 V2	T24 V0	T24 V2	T32 V0	T32 V2	T40 V0	T40 V2	T48 V0	T48 V2	T56 V0	T56 V2
1	T0 V1	T0 V3	T8 V1	T8 V3	T16 V1	T16 V3	T24 V1	T24 V3	T32 V1	T32 V3	T40 V1	T40 V3	T48 V1	T48 V3	T56 V1	T56 V3
2	T1 V0	T1 V2	T9 V0	T9 V2	T17 V0	T17 V2	T25 V0	T25 V2	T33 V0	T33 V2	T41 V0	T41 V2	T49 V0	T49 V2	T57 V0	T57 V2
3	T1 V1	T1 V3	T9 V1	T9 V3	T17 V1	T17 V3	T25 V1	T25 V3	T33 V1	T33 V3	T41 V1	T41 V3	T49 V1	T49 V3	T57 V1	T57 V3
4	T2 V0	T2 V2	T10 V0	T10 V2	T18 V0	T18 V2	T26 V0	T26 V2	T34 V0	T34 V2	T42 V0	T42 V2	T50 V0	T50 V2	T58 V0	T58 V2
5	T2 V1	T2 V3	T10 V1	T10 V3	T18 V1	T18 V3	T26 V1	T26 V3	T34 V1	T34 V3	T42 V1	T42 V3	T50 V1	T50 V3	T58 V1	T58 V3
6	T3 V0	T3 V2	T11 V0	T11 V2	T19 V0	T19 V2	T27 V0	T27 V2	T35 V0	T35 V2	T43 V0	T43 V2	T51 V0	T51 V2	T59 V0	T59 V2
7	T3 V1	T3 V3	T11 V1	T11 V3	T19 V1	T19 V3	T27 V1	T27 V3	T35 V1	T35 V3	T43 V1	T43 V3	T51 V1	T51 V3	T59 V1	T59 V3
8	T4 V0	T4 V2	T12 V0	T12 V2	T20 V0	T20 V2	T28 V0	T28 V2	T36 V0	T36 V2	T44 V0	T44 V2	T52 V0	T52 V2	T60 V0	T60 V2
9	T4 V1	T4 V3	T12 V1	T12 V3	T20 V1	T20 V3	T28 V1	T28 V3	T36 V1	T36 V3	T44 V1	T44 V3	T52 V1	T52 V3	T60 V1	T60 V3
10	T5 V0	T5 V2	T13 V0	T13 V2	T21 V0	T21 V2	T29 V0	T29 V2	T37 V0	T37 V2	T45 V0	T45 V2	T53 V0	T53 V2	T61 V0	T61 V2
11	T5 V1	T5 V3	T13 V1	T13 V3	T21 V1	T21 V3	T29 V1	T29 V3	T37 V1	T37 V3	T45 V1	T45 V3	T53 V1	T53 V3	T61 V1	T61 V3
12	T6 V0	T6 V2	T14 V0	T14 V2	T22 V0	T22 V2	T30 V0	T30 V2	T38 V0	T38 V2	T46 V0	T46 V2	T54 V0	T54 V2	T62 V0	T62 V2
13	T6 V1	T6 V3	T14 V1	T14 V3	T22 V1	T22 V3	T30 V1	T30 V3	T38 V1	T38 V3	T46 V1	T46 V3	T54 V1	T54 V3	T62 V1	T62 V3
14	T7 V0	T7 V2	T15 V0	T15 V2	T23 V0	T23 V2	T31 V0	T31 V2	T39 V0	T39 V2	T47 V0	T47 V2	T55 V0	T55 V2	T63 V0	T63 V2
15	T7 V1	T7 V3	T15 V1	T15 V3	T23 V1	T23 V3	T31 V1	T31 V3	T39 V1	T39 V3	T47 V1	T47 V3	T55 V1	T55 V3	T63 V1	T63 V3

Tiled MMA 16x16x1

- Tile further across values
  - Each thread computes a 2x2 accumulator fragment
- Permute the tiling to enjoy vectorized loads from SMEM to RMEM for M-major A and N-major B

Programming with CuTe in Python

Example 1: Building complex tiling of MMA Atoms



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0 V0	T0 V1	T8 V0	T8 V1	T16 V0	T16 V1	T24 V0	T24 V1	T32 V0	T32 V1	T40 V0	T40 V1	T48 V0	T48 V1	T56 V0	T56 V1

	0
0	T0 V0
1	T0 V1
2	T1 V0
3	T1 V1
4	T2 V0
5	T2 V1
6	T3 V0
7	T3 V1
8	T4 V0
9	T4 V1
10	T5 V0
11	T5 V1
12	T6 V0
13	T6 V1
14	T7 V0
15	T7 V1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0 V0	T0 V2	T8 V0	T8 V2	T16 V0	T16 V2	T24 V0	T24 V2	T32 V0	T32 V2	T40 V0	T40 V2	T48 V0	T48 V2	T56 V0	T56 V2
1	T0 V1	T0 V3	T8 V1	T8 V3	T16 V1	T16 V3	T24 V1	T24 V3	T32 V1	T32 V3	T40 V1	T40 V3	T48 V1	T48 V3	T56 V1	T56 V3
2	T1 V0	T1 V2	T9 V0	T9 V2	T17 V0	T17 V2	T25 V0	T25 V2	T33 V0	T33 V2	T41 V0	T41 V2	T49 V0	T49 V2	T57 V0	T57 V2
3	T1 V1	T1 V3	T9 V1	T9 V3	T17 V1	T17 V3	T25 V1	T25 V3	T33 V1	T33 V3	T41 V1	T41 V3	T49 V1	T49 V3	T57 V1	T57 V3
4	T2 V0	T2 V2	T10 V0	T10 V2	T18 V0	T18 V2	T26 V0	T26 V2	T34 V0	T34 V2	T42 V0	T42 V2	T50 V0	T50 V2	T58 V0	T58 V2
5	T2 V1	T2 V3	T10 V1	T10 V3	T18 V1	T18 V3	T26 V1	T26 V3	T34 V1	T34 V3	T42 V1	T42 V3	T50 V1	T50 V3	T58 V1	T58 V3
6	T3 V0	T3 V2	T11 V0	T11 V2	T19 V0	T19 V2	T27 V0	T27 V2	T35 V0	T35 V2	T43 V0	T43 V2	T51 V0	T51 V2	T59 V0	T59 V2
7	T3 V1	T3 V3	T11 V1	T11 V3	T19 V1	T19 V3	T27 V1	T27 V3	T35 V1	T35 V3	T43 V1	T43 V3	T51 V1	T51 V3	T59 V1	T59 V3
8	T4 V0	T4 V2	T12 V0	T12 V2	T20 V0	T20 V2	T28 V0	T28 V2	T36 V0	T36 V2	T44 V0	T44 V2	T52 V0	T52 V2	T60 V0	T60 V2
9	T4 V1	T4 V3	T12 V1	T12 V3	T20 V1	T20 V3	T28 V1	T28 V3	T36 V1	T36 V3	T44 V1	T44 V3	T52 V1	T52 V3	T60 V1	T60 V3
10	T5 V0	T5 V2	T13 V0	T13 V2	T21 V0	T21 V2	T29 V0	T29 V2	T37 V0	T37 V2	T45 V0	T45 V2	T53 V0	T53 V2	T61 V0	T61 V2
11	T5 V1	T5 V3	T13 V1	T13 V3	T21 V1	T21 V3	T29 V1	T29 V3	T37 V1	T37 V3	T45 V1	T45 V3	T53 V1	T53 V3	T61 V1	T61 V3
12	T6 V0	T6 V2	T14 V0	T14 V2	T22 V0	T22 V2	T30 V0	T30 V2	T38 V0	T38 V2	T46 V0	T46 V2	T54 V0	T54 V2	T62 V0	T62 V2
13	T6 V1	T6 V3	T14 V1	T14 V3	T22 V1	T22 V3	T30 V1	T30 V3	T38 V1	T38 V3	T46 V1	T46 V3	T54 V1	T54 V3	T62 V1	T62 V3
14	T7 V0	T7 V2	T15 V0	T15 V2	T23 V0	T23 V2	T31 V0	T31 V2	T39 V0	T39 V2	T47 V0	T47 V2	T55 V0	T55 V2	T63 V0	T63 V2
15	T7 V1	T7 V3	T15 V1	T15 V3	T23 V1	T23 V3	T31 V1	T31 V3	T39 V1	T39 V3	T47 V1	T47 V3	T55 V1	T55 V3	T63 V1	T63 V3

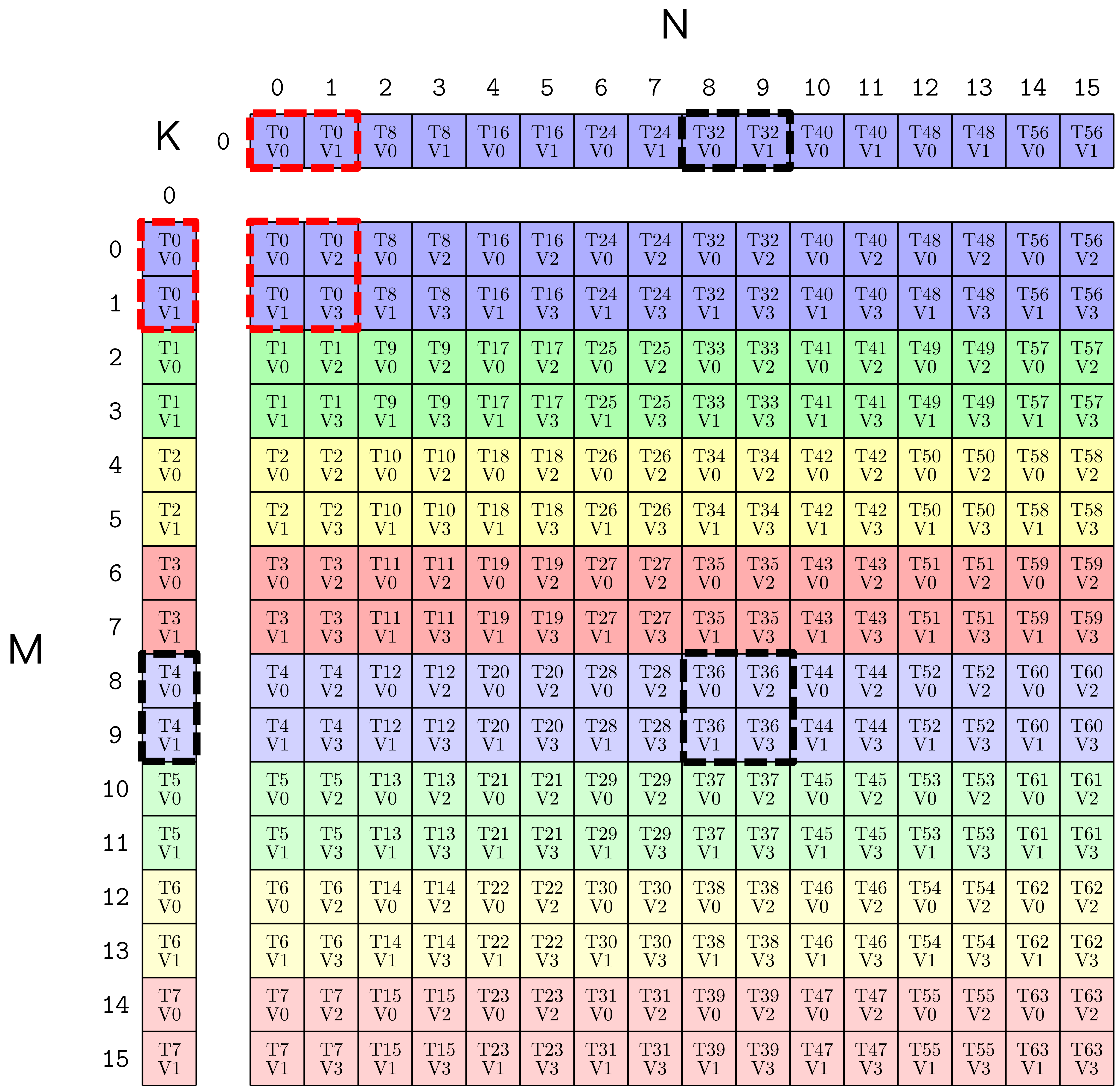
Tiled MMA 16x16x1

A programming pattern applicable to ***any*** MMA!

```
# Layout of Atoms: tiling across threads
atom_layout = cute.make_layout((8, 8, 1))
# Permutation Tiler:
# - tiling across values
# - permutation of the overall tiling
perm_tiler = (
    cute.make_layout((8, 2), stride=(2, 1)),
    cute.make_layout((8, 2), stride=(2, 1)),
    None,
)
tiled_mma = cute.make_tiled_mma(
    cute.nvgpu.MmaUniversalOp(),
    atom_layout,
    perm_tiler,
)
```

Programming with CuTe in Python

Example 1: Building complex tiling of MMA Atoms



K \ N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0 V0	T0 V1	T8 V0	T8 V1	T16 V0	T16 V1	T24 V0	T24 V1	T32 V0	T32 V1	T40 V0	T40 V1	T48 V0	T48 V1	T56 V0	T56 V1

M \ K	0
0	T0 V0
1	T0 V1
2	T1 V0
3	T1 V1
4	T2 V0
5	T2 V1
6	T3 V0
7	T3 V1
8	T4 V0
9	T4 V1
10	T5 V0
11	T5 V1
12	T6 V0
13	T6 V1
14	T7 V0
15	T7 V1

M \ N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T0 V0	T0 V2	T8 V0	T8 V2	T16 V0	T16 V2	T24 V0	T24 V2	T32 V0	T32 V2	T40 V0	T40 V2	T48 V0	T48 V2	T56 V0	T56 V2
1	T0 V1	T0 V3	T8 V1	T8 V3	T16 V1	T16 V3	T24 V1	T24 V3	T32 V1	T32 V3	T40 V1	T40 V3	T48 V1	T48 V3	T56 V1	T56 V3
2	T1 V0	T1 V2	T9 V0	T9 V2	T17 V0	T17 V2	T25 V0	T25 V2	T33 V0	T33 V2	T41 V0	T41 V2	T49 V0	T49 V2	T57 V0	T57 V2
3	T1 V1	T1 V3	T9 V1	T9 V3	T17 V1	T17 V3	T25 V1	T25 V3	T33 V1	T33 V3	T41 V1	T41 V3	T49 V1	T49 V3	T57 V1	T57 V3
4	T2 V0	T2 V2	T10 V0	T10 V2	T18 V0	T18 V2	T26 V0	T26 V2	T34 V0	T34 V2	T42 V0	T42 V2	T50 V0	T50 V2	T58 V0	T58 V2
5	T2 V1	T2 V3	T10 V1	T10 V3	T18 V1	T18 V3	T26 V1	T26 V3	T34 V1	T34 V3	T42 V1	T42 V3	T50 V1	T50 V3	T58 V1	T58 V3
6	T3 V0	T3 V2	T11 V0	T11 V2	T19 V0	T19 V2	T27 V0	T27 V2	T35 V0	T35 V2	T43 V0	T43 V2	T51 V0	T51 V2	T59 V0	T59 V2
7	T3 V1	T3 V3	T11 V1	T11 V3	T19 V1	T19 V3	T27 V1	T27 V3	T35 V1	T35 V3	T43 V1	T43 V3	T51 V1	T51 V3	T59 V1	T59 V3
8	T4 V0	T4 V2	T12 V0	T12 V2	T20 V0	T20 V2	T28 V0	T28 V2	T36 V0	T36 V2	T44 V0	T44 V2	T52 V0	T52 V2	T60 V0	T60 V2
9	T4 V1	T4 V3	T12 V1	T12 V3	T20 V1	T20 V3	T28 V1	T28 V3	T36 V1	T36 V3	T44 V1	T44 V3	T52 V1	T52 V3	T60 V1	T60 V3
10	T5 V0	T5 V2	T13 V0	T13 V2	T21 V0	T21 V2	T29 V0	T29 V2	T37 V0	T37 V2	T45 V0	T45 V2	T53 V0	T53 V2	T61 V0	T61 V2
11	T5 V1	T5 V3	T13 V1	T13 V3	T21 V1	T21 V3	T29 V1	T29 V3	T37 V1	T37 V3	T45 V1	T45 V3	T53 V1	T53 V3	T61 V1	T61 V3
12	T6 V0	T6 V2	T14 V0	T14 V2	T22 V0	T22 V2	T30 V0	T30 V2	T38 V0	T38 V2	T46 V0	T46 V2	T54 V0	T54 V2	T62 V0	T62 V2
13	T6 V1	T6 V3	T14 V1	T14 V3	T22 V1	T22 V3	T30 V1	T30 V3	T38 V1	T38 V3	T46 V1	T46 V3	T54 V1	T54 V3	T62 V1	T62 V3
14	T7 V0	T7 V2	T15 V0	T15 V2	T23 V0	T23 V2	T31 V0	T31 V2	T39 V0	T39 V2	T47 V0	T47 V2	T55 V0	T55 V2	T63 V0	T63 V2
15	T7 V1	T7 V3	T15 V1	T15 V3	T23 V1	T23 V3	T31 V1	T31 V3	T39 V1	T39 V3	T47 V1	T47 V3	T55 V1	T55 V3	T63 V1	T63 V3

Tiled MMA 16x16x1

Partitioning without ever worrying about the physical stride

```
thr_mma = tiled_mma.get_slice(tidx)
tCsA = thr_mma.partition_A(sA)
```

M-major (column-major)
MxK=32x8 SMEM A

$$\texttt{base\_ptr} \circ \boxed{(32,8):(1,32)}$$

$$\texttt{offset\_ptr} \circ (\boxed{1}, \boxed{(2,2)}, \boxed{8}):(0,(1,16),32)$$

- The MMA mode for a single atom
- # repetitions along M to cover the full 32x8 tile
- # repetitions along K to cover the full 32x8 tile

Partitioning at a high level:

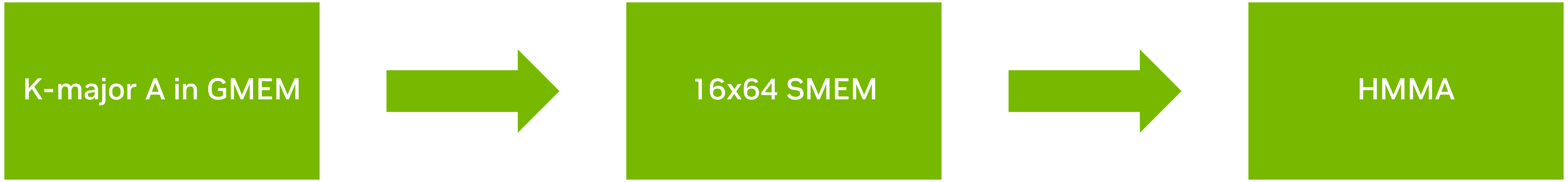
$$\texttt{t} = \text{Data} \circ \text{TV} \;:\; (\texttt{thr\_id}, \texttt{val\_id}) \rightarrow \texttt{data offset}$$

$$\texttt{thr\_part} = \texttt{t}(\texttt{my\_thr}, \_)$$

Programming with CuTe in Python

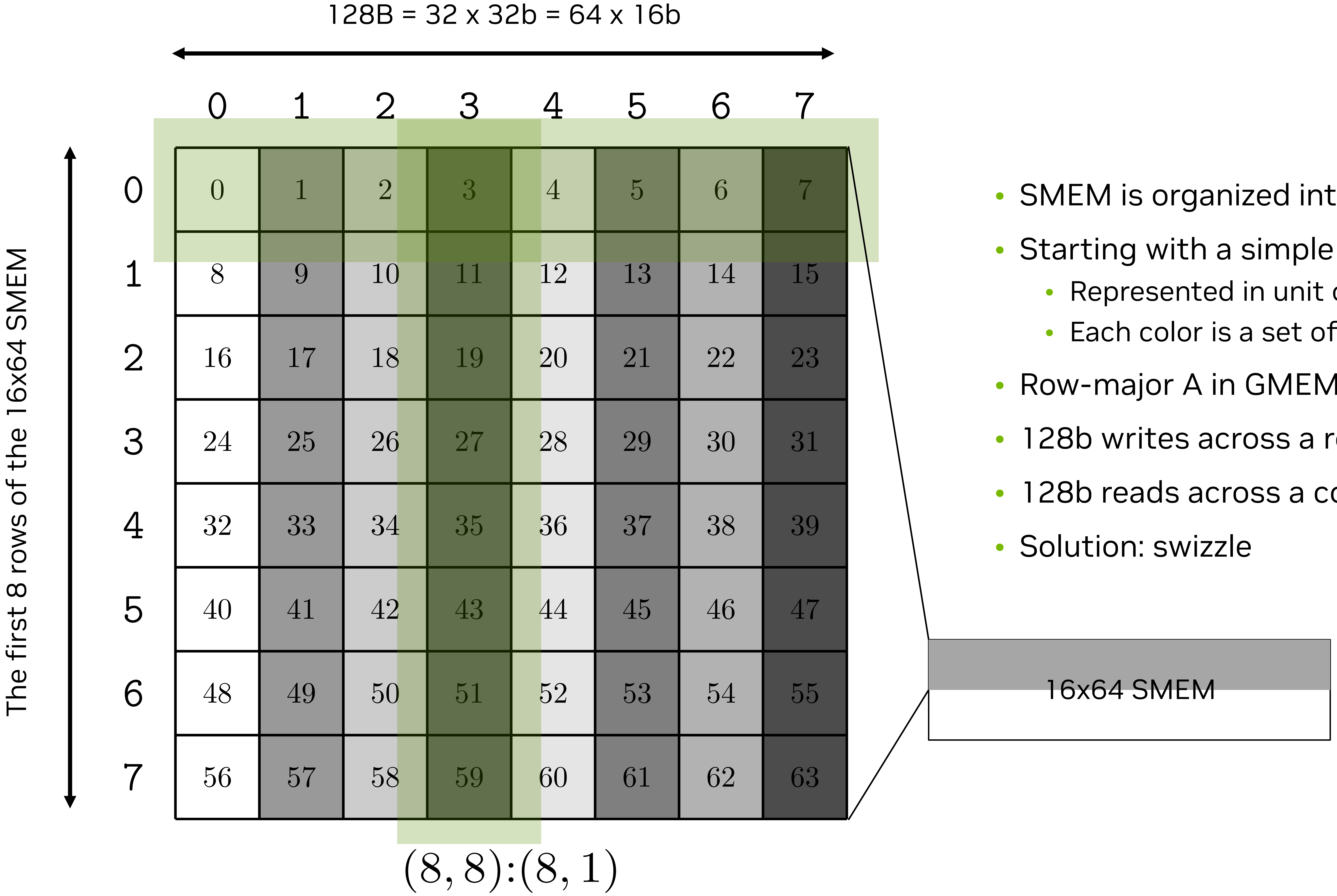
Example 2: GMEM → SMEM → HMMA

- Consider the A operand of a half-precision F16 warp-level MMA (HMMA)



Programming with CuTe in Python

Example 2: Resolving SMEM bank conflicts


128B = 32 x 32b = 64 x 16b
The first 8 rows of the 16x64 SMEM
16x64 SMEM

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63

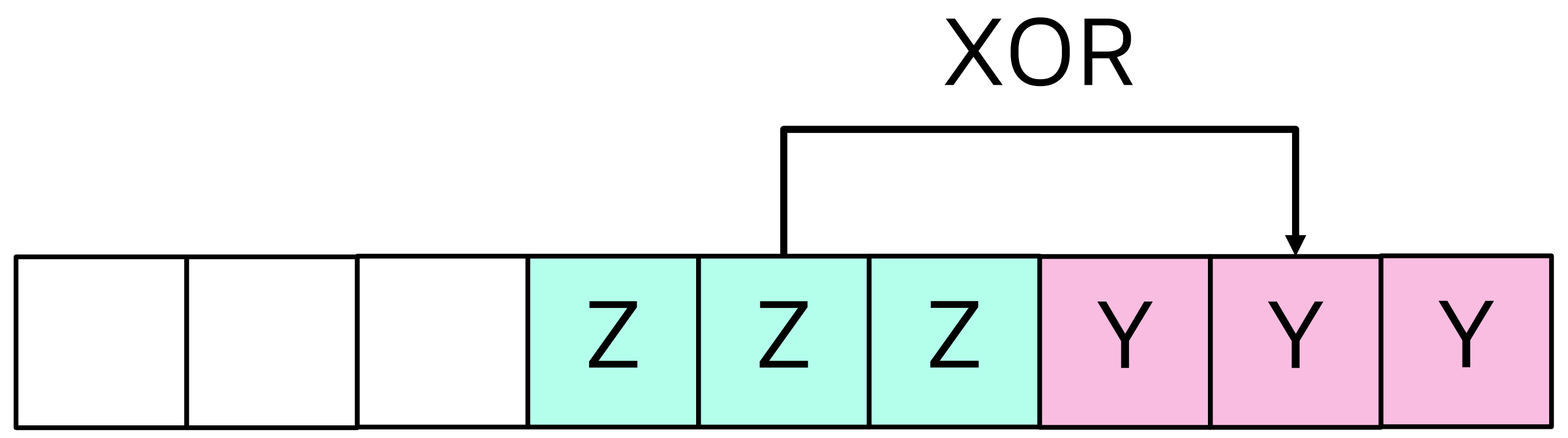
$$(8,8):(8,1)$$

- SMEM is organized into 32 x 32b banks
- Starting with a simple row-major SMEM layout
  - Represented in unit of 128b elements
  - Each color is a set of 4 banks
- Row-major A in GMEM → use LDG.128 + STS.128
- 128b writes across a row are free of bank conflicts… but
- 128b reads across a column all hit the same 4 banks
- Solution: swizzle

Programming with CuTe in Python

Example 2: Resolving SMEM bank conflicts

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	9	8	11	10	13	12	15	14
2	18	19	16	17	22	23	20	21
3	27	26	25	24	31	30	29	28
4	36	37	38	39	32	33	34	35
5	45	44	47	46	41	40	43	42
6	54	55	52	53	50	51	48	49
7	63	62	61	60	59	58	57	56



- Reads/writes across rows/columns are now free of bank conflicts!
- This layout is exactly a swizzle layout

$$S<3,0,3> \circ 0 \ \circ (8,8):(8,1)$$

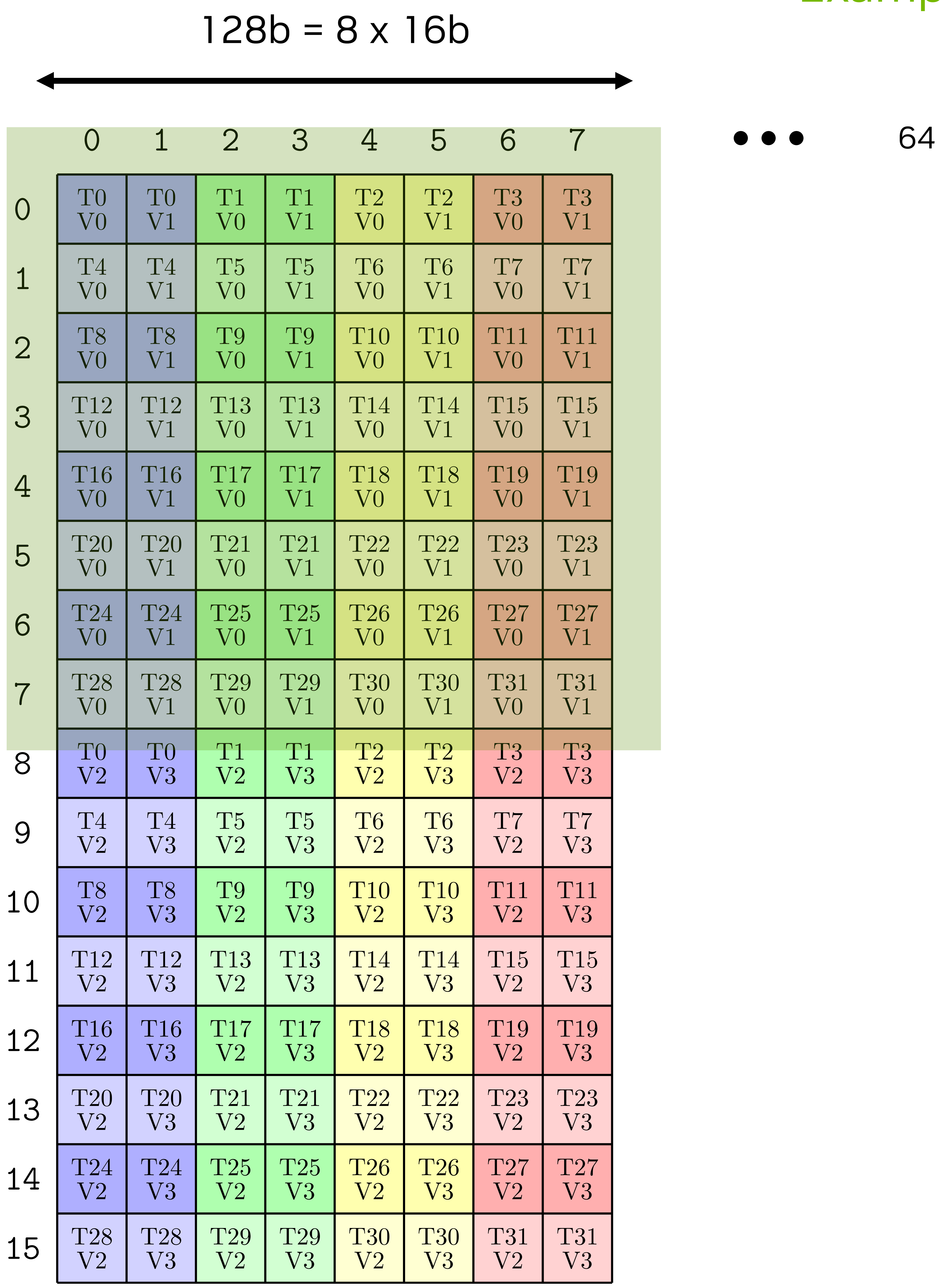
```
• l = cute.make_composed_layout(
      cute.swizzle(3,0,3),
      0,
      cute.make_layout((8,8), stride=(8,1)),
  )
 assert l( (3,6) ) == 29
```

No indexing math!

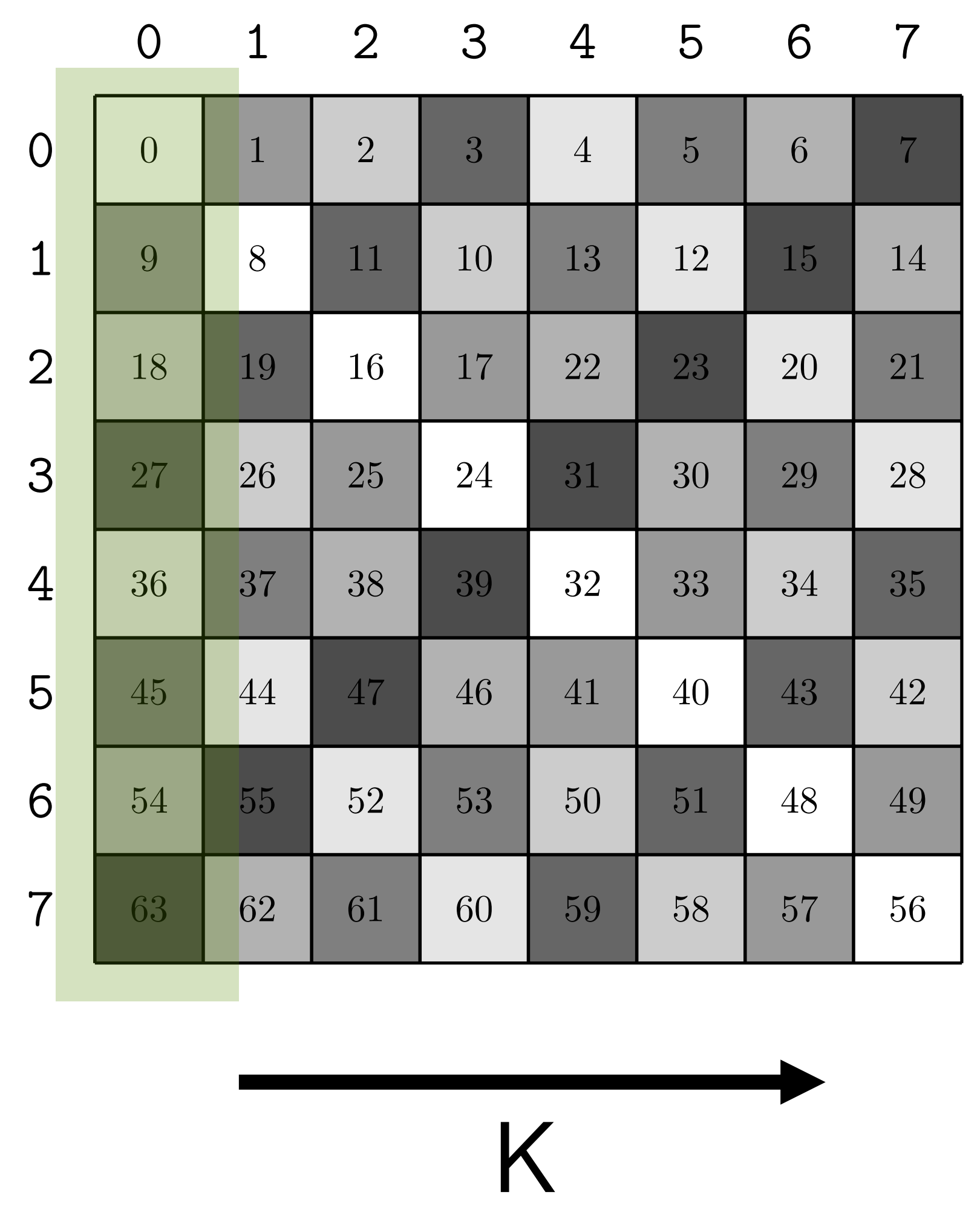
- Such layout can be partitioned just like ***any*** other layout

Programming with CuTe in Python

Example 2: SMEM → RMEM



	0	1	2	3	4	5	6	7
0	T0 V0	T0 V1	T1 V0	T1 V1	T2 V0	T2 V1	T3 V0	T3 V1
1	T4 V0	T4 V1	T5 V0	T5 V1	T6 V0	T6 V1	T7 V0	T7 V1
2	T8 V0	T8 V1	T9 V0	T9 V1	T10 V0	T10 V1	T11 V0	T11 V1
3	T12 V0	T12 V1	T13 V0	T13 V1	T14 V0	T14 V1	T15 V0	T15 V1
4	T16 V0	T16 V1	T17 V0	T17 V1	T18 V0	T18 V1	T19 V0	T19 V1
5	T20 V0	T20 V1	T21 V0	T21 V1	T22 V0	T22 V1	T23 V0	T23 V1
6	T24 V0	T24 V1	T25 V0	T25 V1	T26 V0	T26 V1	T27 V0	T27 V1
7	T28 V0	T28 V1	T29 V0	T29 V1	T30 V0	T30 V1	T31 V0	T31 V1
8	T0 V2	T0 V3	T1 V2	T1 V3	T2 V2	T2 V3	T3 V2	T3 V3
9	T4 V2	T4 V3	T5 V2	T5 V3	T6 V2	T6 V3	T7 V2	T7 V3
10	T8 V2	T8 V3	T9 V2	T9 V3	T10 V2	T10 V3	T11 V2	T11 V3
11	T12 V2	T12 V3	T13 V2	T13 V3	T14 V2	T14 V3	T15 V2	T15 V3
12	T16 V2	T16 V3	T17 V2	T17 V3	T18 V2	T18 V3	T19 V2	T19 V3
13	T20 V2	T20 V3	T21 V2	T21 V3	T22 V2	T22 V3	T23 V2	T23 V3
14	T24 V2	T24 V3	T25 V2	T25 V3	T26 V2	T26 V3	T27 V2	T27 V3
15	T28 V2	T28 V3	T29 V2	T29 V3	T30 V2	T30 V3	T31 V2	T31 V3



	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	9	8	11	10	13	12	15	14
2	18	19	16	17	22	23	20	21
3	27	26	25	24	31	30	29	28
4	36	37	38	39	32	33	34	35
5	45	44	47	46	41	40	43	42
6	54	55	52	53	50	51	48	49
7	63	62	61	60	59	58	57	56

- A's TV-layout for a 16x8x8 HMMA
- How do I prepare my register fragments ***efficiently***?
- Each `ldmatrix` instruction loads 128b from 8 rows and places the values in the registers of threads according to HMMA
- This is exactly one column in the previous figure, thus free of bank conflicts
- This pattern can be repeated along K for larger tiles and remains free of bank conflicts

PTX doc for ldmatrix
"The eight addresses required for each matrix are provided by eight threads [...] Each address corresponds to the start of a matrix row."

Threads 0–7 ???

addr0–addr7

Programming with CuTe in Python

Example 2: SMEM → RMEM

Construct the Tiled MMA and partition the (swizzled!) SMEM tensor according to the MMA

```
# Construct a 16x8x8 MMA with F16/F32 inputs/output
# Partition SMEM tensors according to the MMA
mma_op = warp.MmaF16BF16Op(
  cutlass.Float16, cutlass.Float32, (16, 8, 8))
tiled_mma = cute.make_tiled_mma(mma_op)
thr_mma = tiled_mma.get_slice(tidx)
tCsA = thr_mma.partition_A(sA)
tCrA = thr_mma.make_fragment_A(tCsA)
```

Construct a Tiled Copy of `ldmatrix` ***based off the MMA***

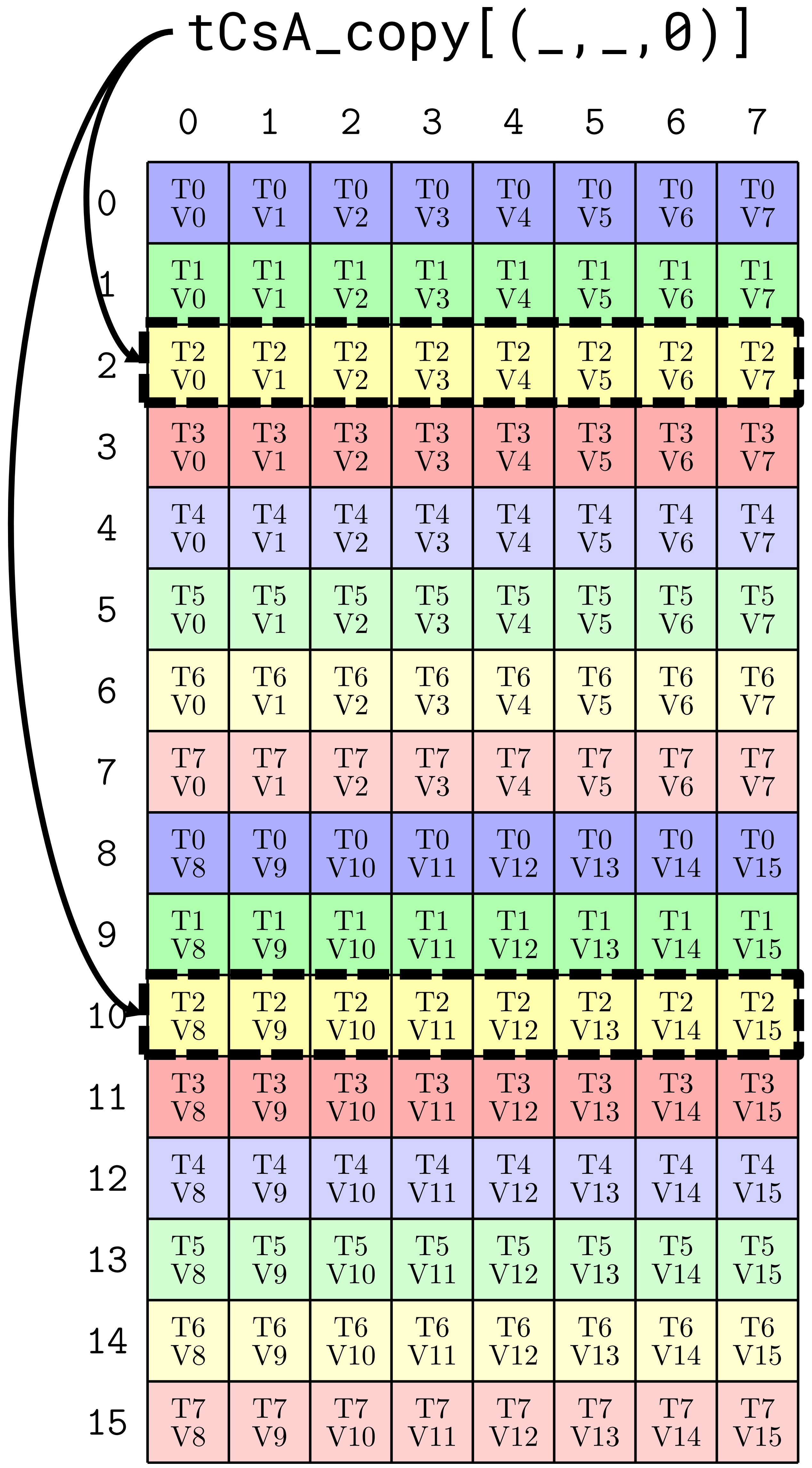
```
# Construct a tiled Copy based off the MMA
# This takes care of any repetition (x2 along M here)
copy_op = warp.LdMatrix8x8x16bOp(transpose=False)
copy_atom = cute.make_copy_atom(copy_op, cutlass.Float16)
tiled_copy = cute.make_tiled_copy_A(copy_atom, tiled_mma)
```

Repartition the source SMEM tensor now according to the Copy

```
# Repartition SMEM source tensors according to the Copy
# Retile RMEM destination tensors according to the Copy
thr_copy = tiled_copy.get_slice(tidx)
tCsA_copy = thr_copy.partition_S(sA)
tCrA_copy = thr_copy.retile(tCrA)
```

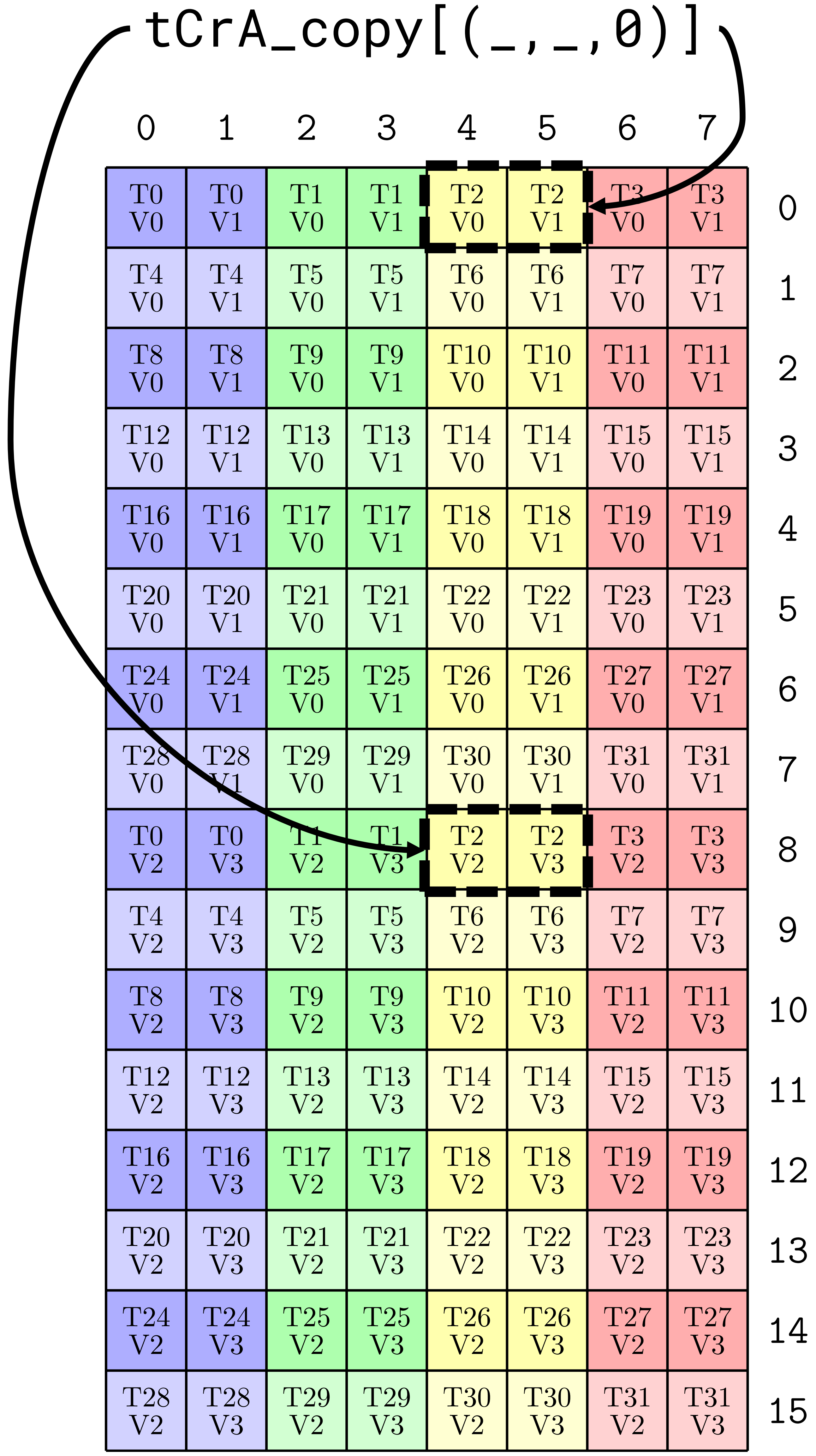
Programming with CuTe in Python

Example 2: SMEM → RMEM



	0	1	2	3	4	5	6	7
0	T0 V0	T0 V1	T0 V2	T0 V3	T0 V4	T0 V5	T0 V6	T0 V7
1	T1 V0	T1 V1	T1 V2	T1 V3	T1 V4	T1 V5	T1 V6	T1 V7
2	T2 V0	T2 V1	T2 V2	T2 V3	T2 V4	T2 V5	T2 V6	T2 V7
3	T3 V0	T3 V1	T3 V2	T3 V3	T3 V4	T3 V5	T3 V6	T3 V7
4	T4 V0	T4 V1	T4 V2	T4 V3	T4 V4	T4 V5	T4 V6	T4 V7
5	T5 V0	T5 V1	T5 V2	T5 V3	T5 V4	T5 V5	T5 V6	T5 V7
6	T6 V0	T6 V1	T6 V2	T6 V3	T6 V4	T6 V5	T6 V6	T6 V7
7	T7 V0	T7 V1	T7 V2	T7 V3	T7 V4	T7 V5	T7 V6	T7 V7
8	T0 V8	T0 V9	T0 V10	T0 V11	T0 V12	T0 V13	T0 V14	T0 V15
9	T1 V8	T1 V9	T1 V10	T1 V11	T1 V12	T1 V13	T1 V14	T1 V15
10	T2 V8	T2 V9	T2 V10	T2 V11	T2 V12	T2 V13	T2 V14	T2 V15
11	T3 V8	T3 V9	T3 V10	T3 V11	T3 V12	T3 V13	T3 V14	T3 V15
12	T4 V8	T4 V9	T4 V10	T4 V11	T4 V12	T4 V13	T4 V14	T4 V15
13	T5 V8	T5 V9	T5 V10	T5 V11	T5 V12	T5 V13	T5 V14	T5 V15
14	T6 V8	T6 V9	T6 V10	T6 V11	T6 V12	T6 V13	T6 V14	T6 V15
15	T7 V8	T7 V9	T7 V10	T7 V11	T7 V12	T7 V13	T7 V14	T7 V15

Copy's src view conforming with `ldmatrix`



0	1	2	3	4	5	6	7	
T0 V0	T0 V1	T1 V0	T1 V1	T2 V0	T2 V1	T3 V0	T3 V1	0
T4 V0	T4 V1	T5 V0	T5 V1	T6 V0	T6 V1	T7 V0	T7 V1	1
T8 V0	T8 V1	T9 V0	T9 V1	T10 V0	T10 V1	T11 V0	T11 V1	2
T12 V0	T12 V1	T13 V0	T13 V1	T14 V0	T14 V1	T15 V0	T15 V1	3
T16 V0	T16 V1	T17 V0	T17 V1	T18 V0	T18 V1	T19 V0	T19 V1	4
T20 V0	T20 V1	T21 V0	T21 V1	T22 V0	T22 V1	T23 V0	T23 V1	5
T24 V0	T24 V1	T25 V0	T25 V1	T26 V0	T26 V1	T27 V0	T27 V1	6
T28 V0	T28 V1	T29 V0	T29 V1	T30 V0	T30 V1	T31 V0	T31 V1	7
T0 V2	T0 V3	T1 V2	T1 V3	T2 V2	T2 V3	T3 V2	T3 V3	8
T4 V2	T4 V3	T5 V2	T5 V3	T6 V2	T6 V3	T7 V2	T7 V3	9
T8 V2	T8 V3	T9 V2	T9 V3	T10 V2	T10 V3	T11 V2	T11 V3	10
T12 V2	T12 V3	T13 V2	T13 V3	T14 V2	T14 V3	T15 V2	T15 V3	11
T16 V2	T16 V3	T17 V2	T17 V3	T18 V2	T18 V3	T19 V2	T19 V3	12
T20 V2	T20 V3	T21 V2	T21 V3	T22 V2	T22 V3	T23 V2	T23 V3	13
T24 V2	T24 V3	T25 V2	T25 V3	T26 V2	T26 V3	T27 V2	T27 V3	14
T28 V2	T28 V3	T29 V2	T29 V3	T30 V2	T30 V3	T31 V2	T31 V3	15

Copy's dst view = MMA view

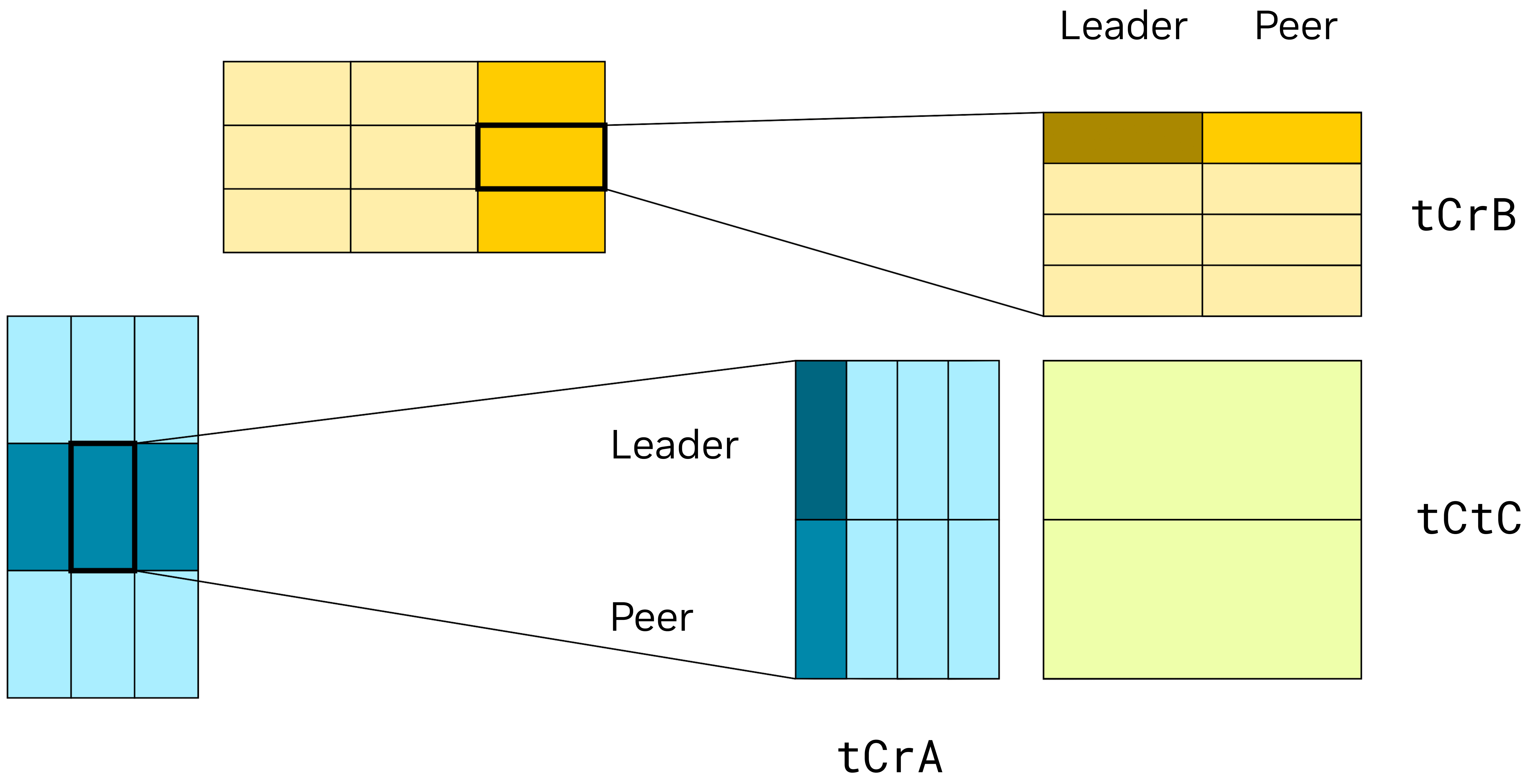
```
# Construct a 16x8x8 MMA with F16/F32 inputs/output
# Partition SMEM tensors according to the MMA
mma_op = warp.MmaF16BF16Op(
  cutlass.Float16, cutlass.Float32, (16, 8, 8))
tiled_mma = cute.make_tiled_mma(mma_op)
thr_mma = tiled_mma.get_slice(tidx)
tCsA = thr_mma.partition_A(sA)
tCrA = thr_mma.make_fragment_A(tCsA)

# Construct a tiled Copy based off the MMA
# This takes care of any repetition (x2 along M here)
copy_op = warp.LdMatrix8x8x16bOp(transpose=False)
copy_atom = cute.make_copy_atom(copy_op, cutlass.Float16)
tiled_copy = cute.make_tiled_copy_A(copy_atom, tiled_mma)

# Repartition SMEM source tensors according to the Copy
# Retile RMEM destination tensors according to the Copy
thr_copy = tiled_copy.get_slice(tidx)
tCsA_copy = thr_copy.partition_S(sA)
tCrA_copy = thr_copy.retile(tCrA)
```

Programming with CuTe in Python

Example 3: 2CTA Blackwell MMA

```
tCrA = tiled_mma.make_fragment_A(sA)   # Create a tensor of SMEM descriptors for the A operand
tCgC = thr_mma.partition_C(gC)
tCtC = tiled_mma.make_fragment_C(tCgC) # Create a TMEM tensor for the accumulator

# Issue a GEMM
cute.gemm(
    tiled_mma,
    tCtC,
    tCrA[(None, None, k_block_idx)],
    tCrB[(None, None, k_block_idx)],
    tCtC,
)
```

The 2CTA MMA consumes the SMEM tensors of both CTAs and stores half of the result in each CTA's TMEM allocation.



What if something is missing?

Seamless integration of raw Op builders

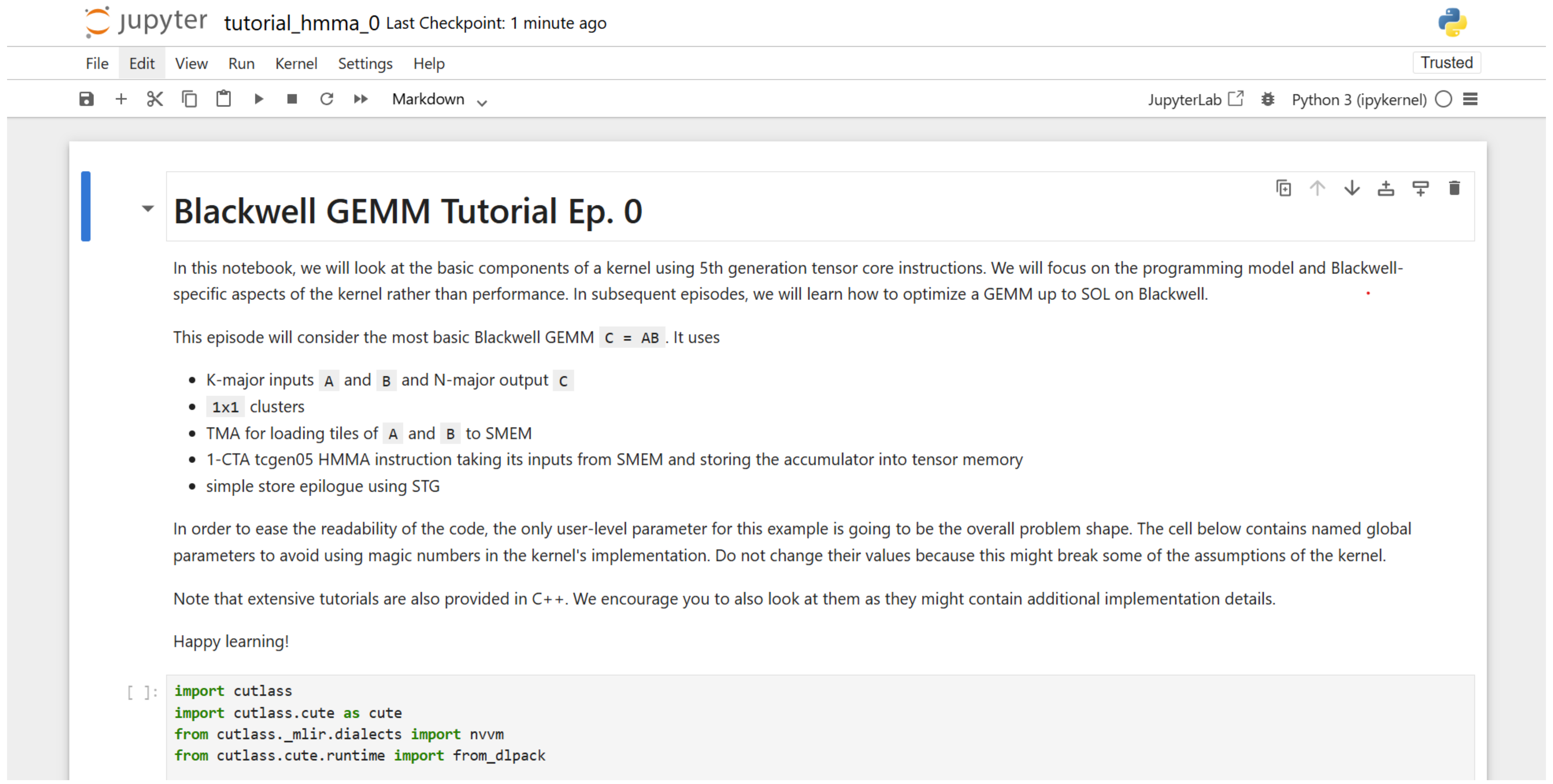
```python
from cutlass._mlir.dialects import llvm, nvvm

def thread_idx(*, loc=None, ip=None):
    return (
        nvvm.read_ptx_sreg_tid_x(T.i32(), loc=loc, ip=ip),
        nvvm.read_ptx_sreg_tid_y(T.i32(), loc=loc, ip=ip),
        nvvm.read_ptx_sreg_tid_z(T.i32(), loc=loc, ip=ip),
    )

def fence_tma_desc_acquire(tma_desc_ptr_i64, *, loc=None, ip=None):
    llvm.inline_asm(
        None,
        [tma_desc_ptr_i64],
        "fence.proxy.tensormap::generic.acquire.gpu [$0], 128;",
        "l",
        has_side_effects=True,
        is_align_stack=False,
        asm_dialect=llvm.AsmDialect.AD_ATT,
        loc=loc,
        ip=ip,
    )
```

- LLVM + NVVM Op builders exposed
- Direct access to NVVM Ops
- ***Inline PTX*** is straightforward
- Subject to upstream breaking changes

What else?

Refreshed tutorials, examples, and documentation

- Initial support for schedulers and persistency
- Initial support for pipeline abstractions
- New centralized documentation
- Educational notebooks from getting started to SOL



jupyter tutorial_hmma_0 Last Checkpoint: 1 minute ago
File Edit View Run Kernel Settings Help
Trusted
Markdown
JupyterLab
Python 3 (ipykernel)

Blackwell GEMM Tutorial Ep. 0

In this notebook, we will look at the basic components of a kernel using 5th generation tensor core instructions. We will focus on the programming model and Blackwell-specific aspects of the kernel rather than performance. In subsequent episodes, we will learn how to optimize a GEMM up to SOL on Blackwell.

This episode will consider the most basic Blackwell GEMM `C = AB`. It uses

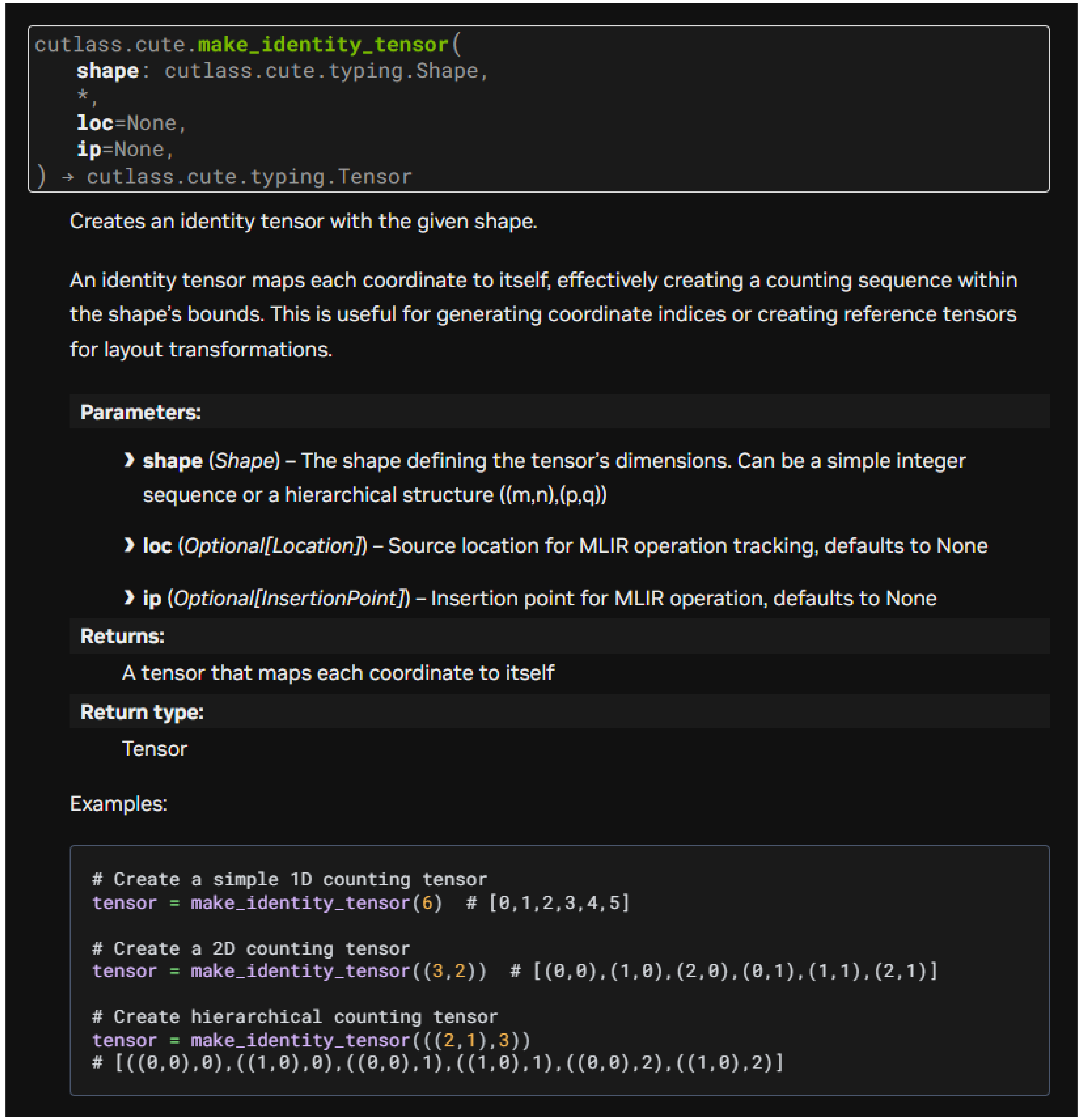
- K-major inputs `A` and `B` and N-major output `C`
- `1x1` clusters
- TMA for loading tiles of `A` and `B` to SMEM
- 1-CTA tcgen05 HMMA instruction taking its inputs from SMEM and storing the accumulator into tensor memory
- simple store epilogue using STG

In order to ease the readability of the code, the only user-level parameter for this example is going to be the overall problem shape. The cell below contains named global parameters to avoid using magic numbers in the kernel's implementation. Do not change their values because this might break some of the assumptions of the kernel.

Note that extensive tutorials are also provided in C++. We encourage you to also look at them as they might contain additional implementation details.

Happy learning!

[ ]:
```python
import cutlass
import cutlass.cute as cute
from cutlass._mlir.dialects import nvvm
from cutlass.cute.runtime import from_dlpack
```



```
cutlass.cute.make_identity_tensor(
    shape: cutlass.cute.typing.Shape,
    *,
    loc=None,
    ip=None,
) → cutlass.cute.typing.Tensor
```

Creates an identity tensor with the given shape.

An identity tensor maps each coordinate to itself, effectively creating a counting sequence within the shape's bounds. This is useful for generating coordinate indices or creating reference tensors for layout transformations.

Parameters:

- **shape** (*Shape*) – The shape defining the tensor's dimensions. Can be a simple integer sequence or a hierarchical structure ((m,n),(p,q))
- **loc** (*Optional[Location]*) – Source location for MLIR operation tracking, defaults to None
- **ip** (*Optional[InsertionPoint]*) – Insertion point for MLIR operation, defaults to None

Returns:

A tensor that maps each coordinate to itself

Return type:

Tensor

Examples:

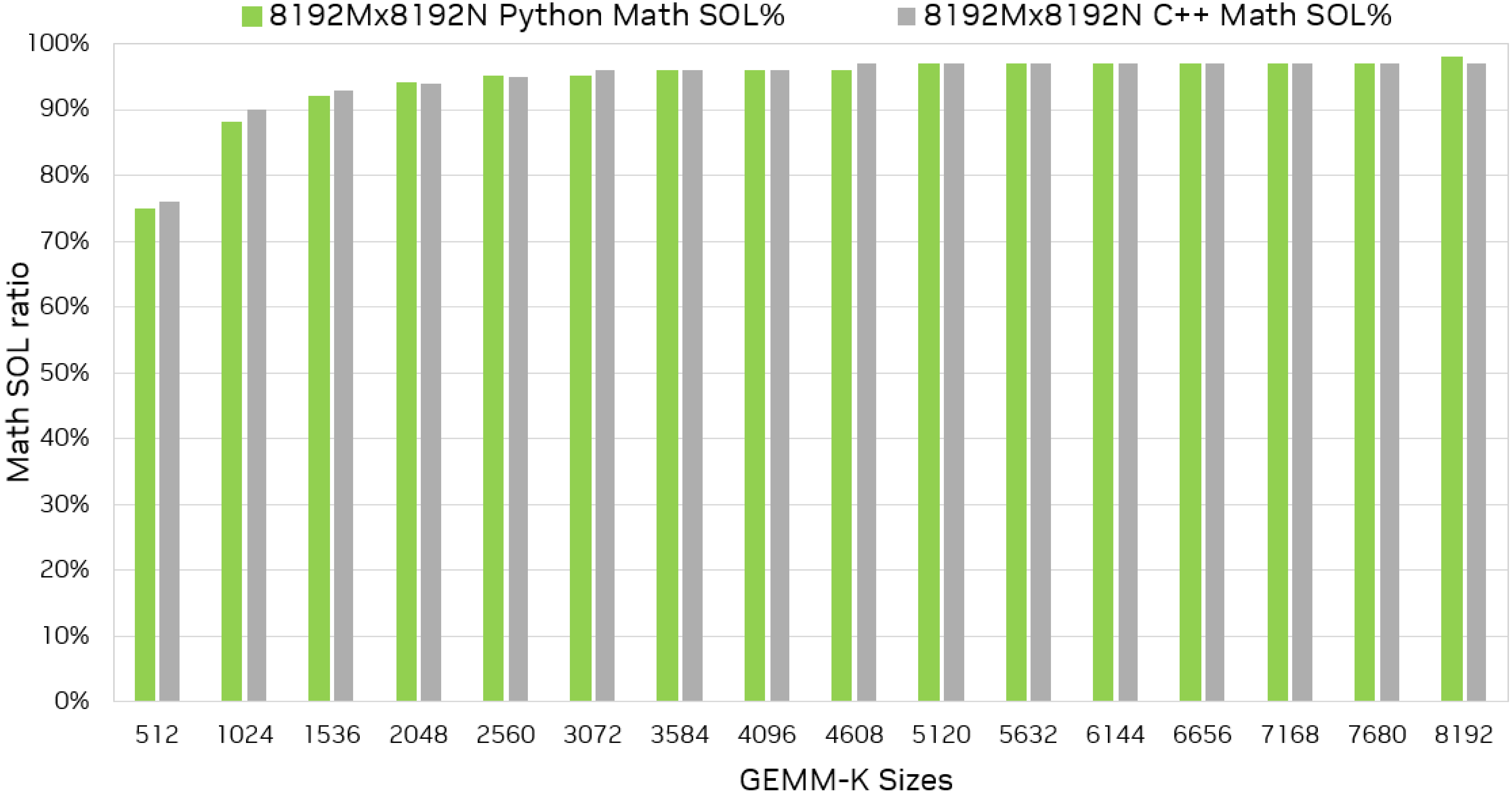
```python
# Create a simple 1D counting tensor
tensor = make_identity_tensor(6)  # [0,1,2,3,4,5]

# Create a 2D counting tensor
tensor = make_identity_tensor((3,2))  # [(0,0),(1,0),(2,0),(0,1),(1,1),(2,1)]

# Create hierarchical counting tensor
tensor = make_identity_tensor(((2,1),3))
# [((0,0),0),((1,0),0),((0,0),1),((1,0),1),((0,0),2),((1,0),2)]
```

Agenda

- Introduction and Motivations
- The DSL Infrastructure
- Kernel Authoring in Python with CuTe
- **Runtime Performance**
- Conclusion

Blackwell Performance: Python vs. C++

FP16 I/O GEMM with M=N=8192

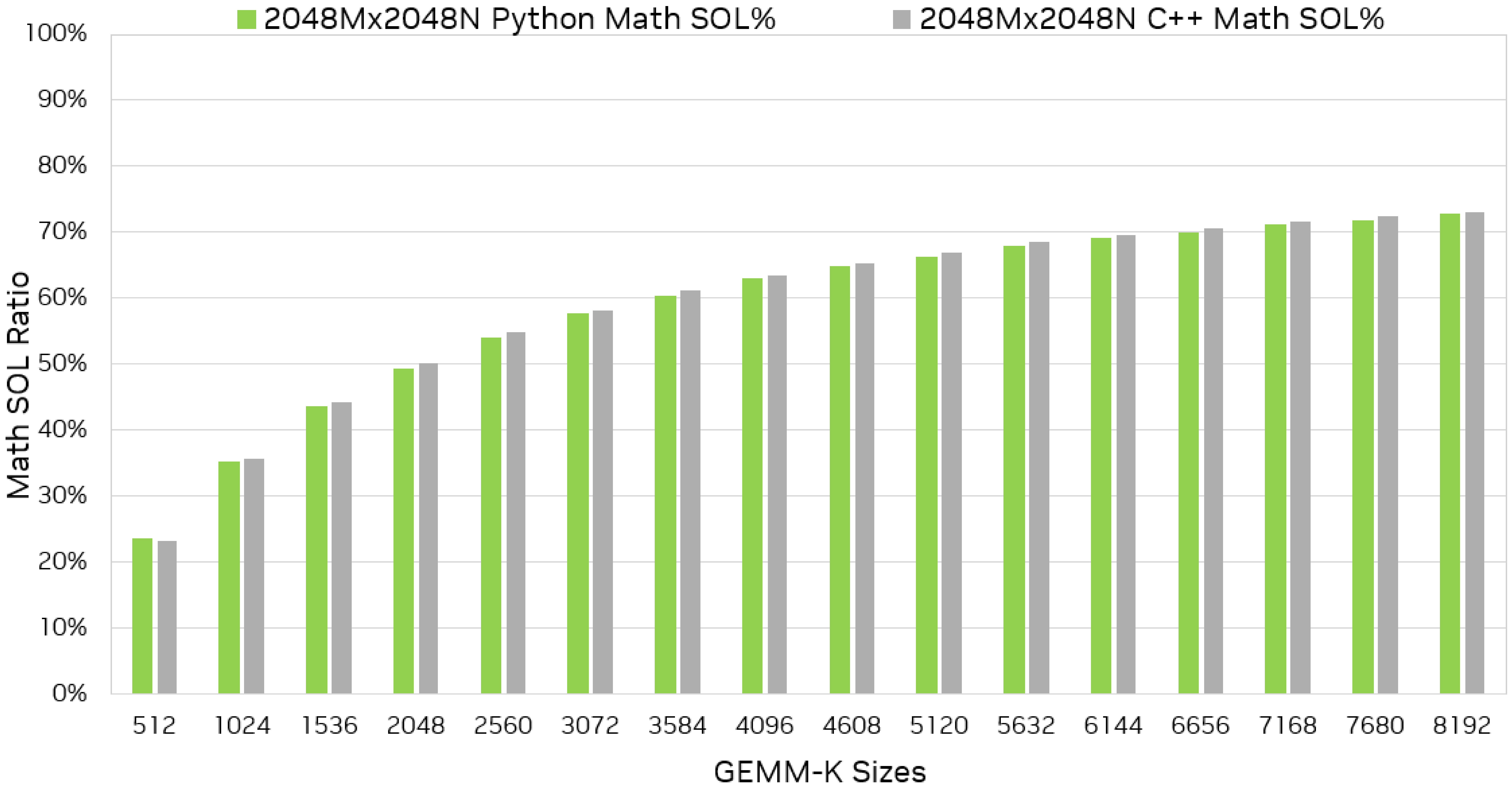


Testing spec: B100 180GB HBM3e, 148SM GPC-800MHz/DRAM 4GHz 850W

128x256x64, Warp-specialized, 2x1 Cluster shape

Blackwell Performance: Python vs. C++

FP16 I/O GEMM with M=N=2048

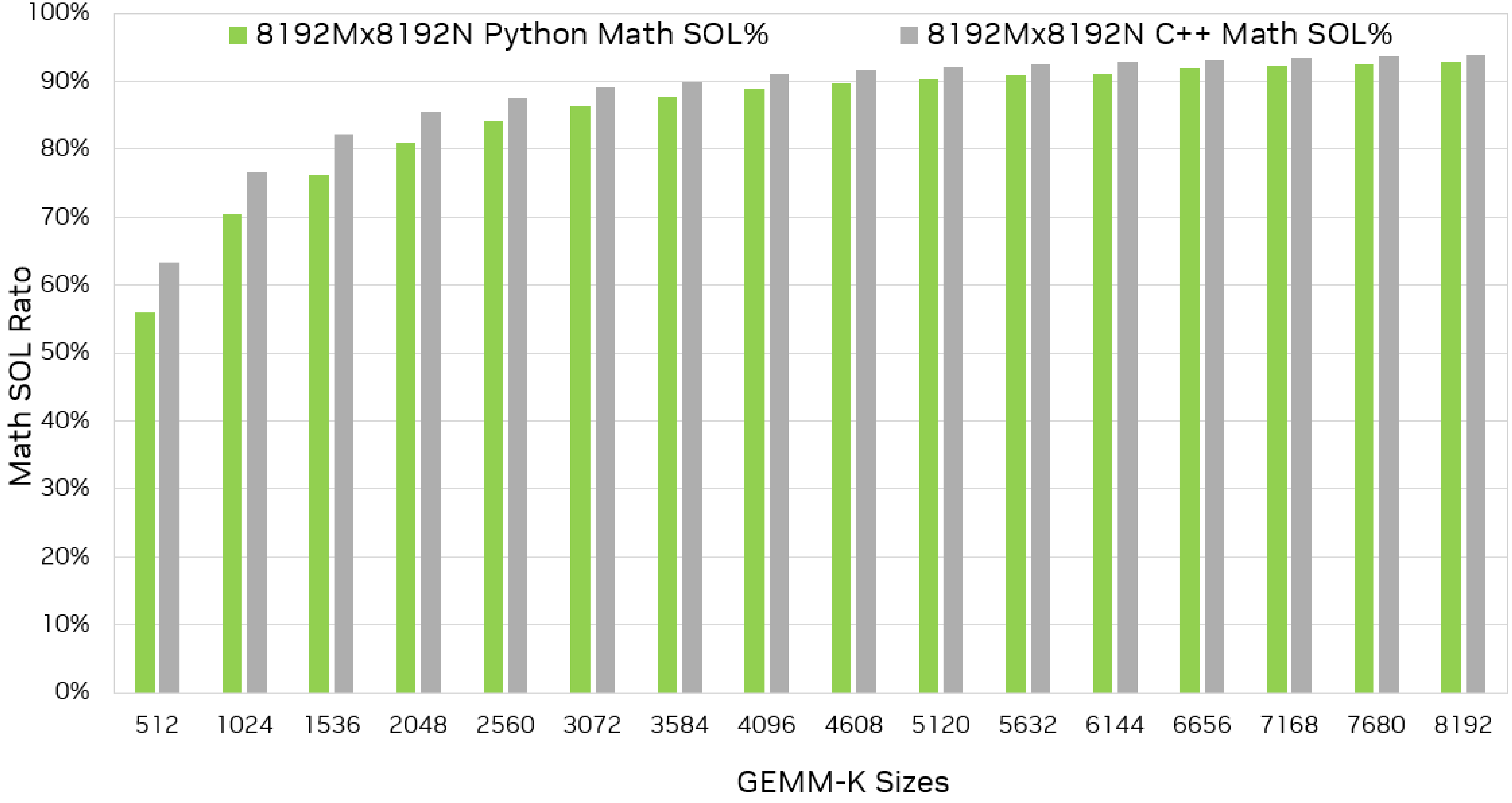


Testing spec: B100 180GB HBM3e, 148SM GPC-800MHz/DRAM 4GHz 850W

256x128x64, Warp-specialized, 2x1 Cluster shape

Hopper Performance: Python vs. C++

FP16 I/O GEMM with M=N=8192



- Perf gap

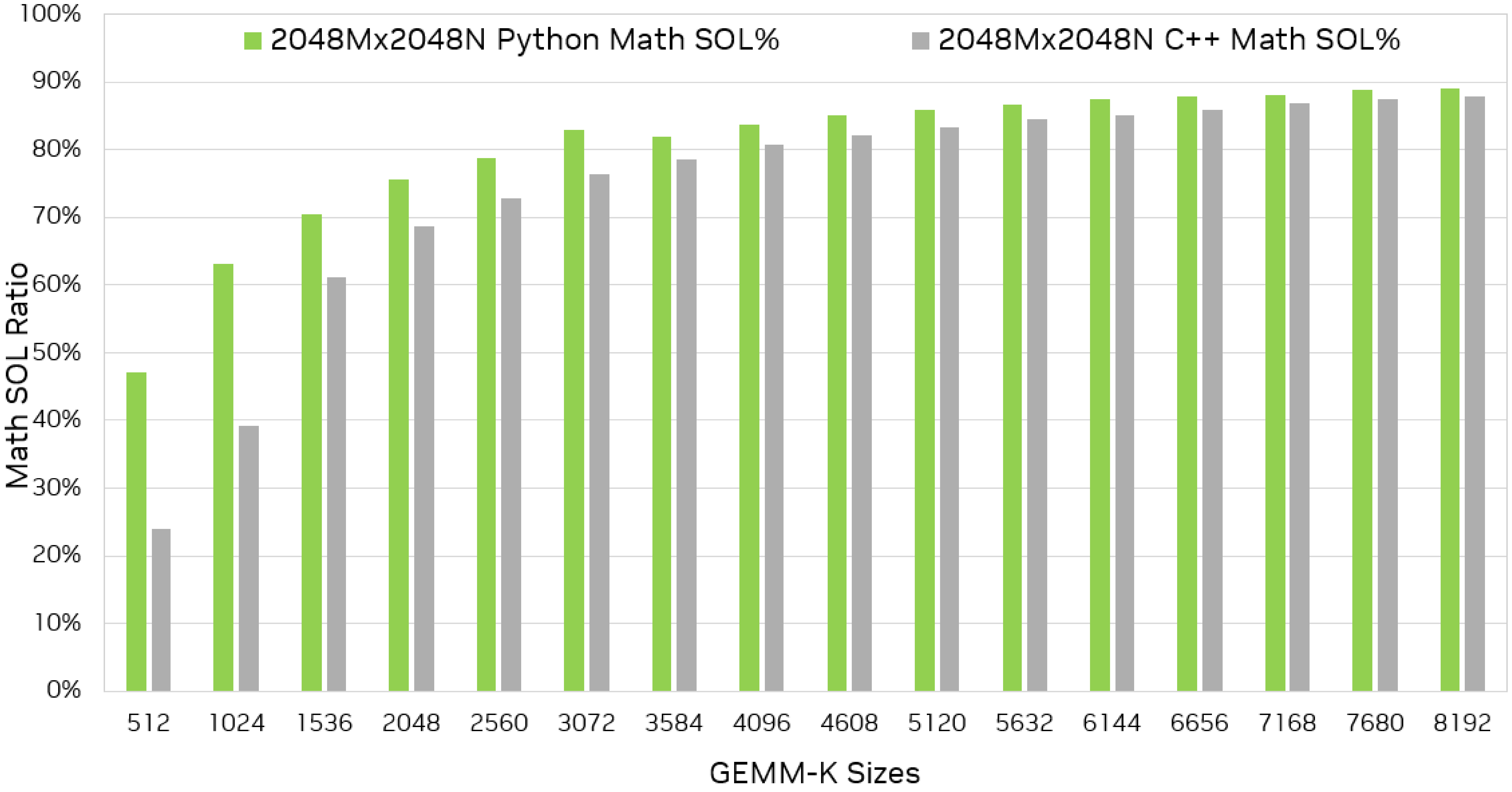
Python example does not include persistent optimization thus expose more overhead among CTA waves for small GEMM-K cases

Testing spec: H100 80GB HBM3, 132SM GPC-1500MHz/DRAM 2619MHz 700W

128x256x64 cooperative size, Swizzle size = 8

Hopper Performance: Python vs. C++

FP16 I/O GEMM with M=N=2048



- Perf gap

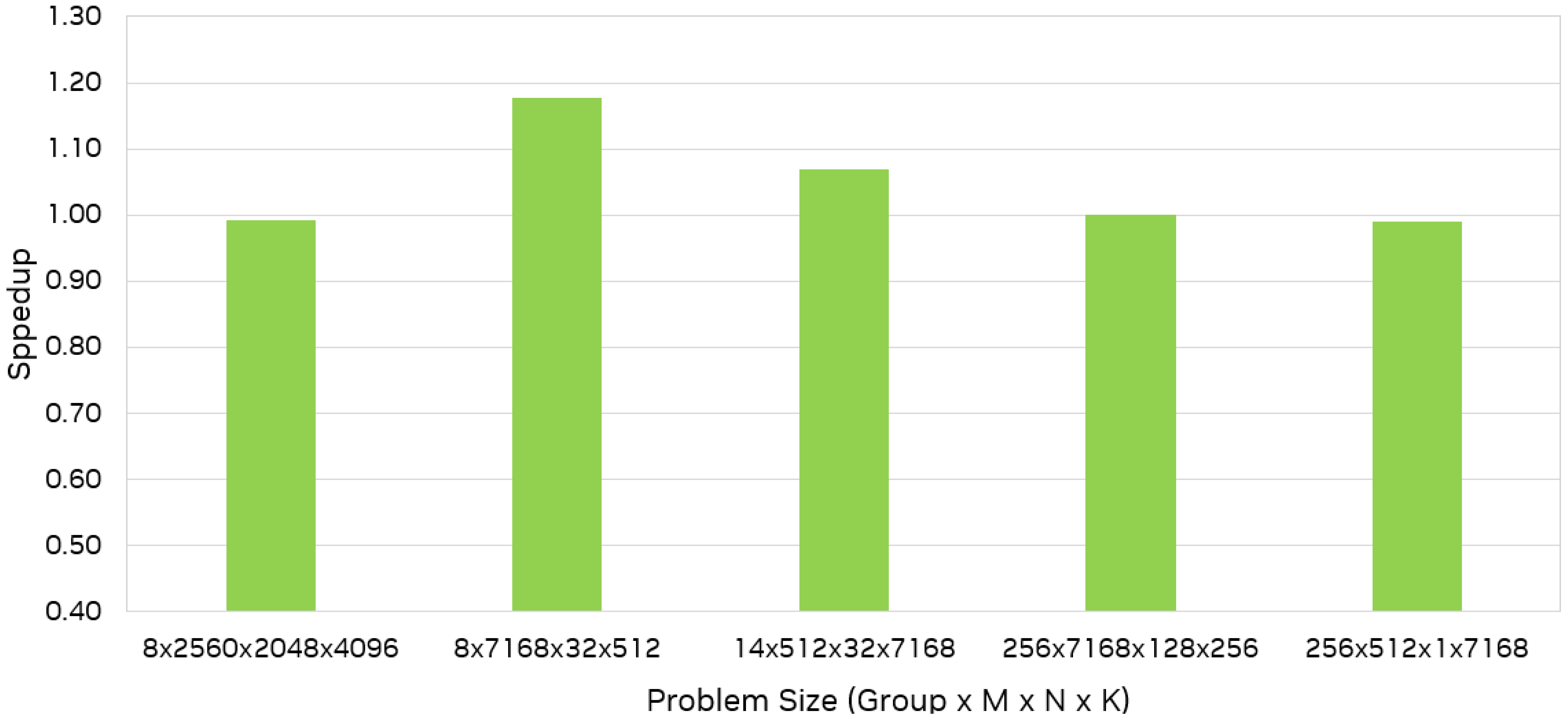
For small GEMM-K case, it’s single wave hence C++ exposed more overhead due to the tile scheduling calculation cost with its warp-specialized persistent scheduler which is not used in Python example.

Testing spec: H100 80GB HBM3, 132SM GPC-1500MHz/DRAM 2619MHz 700W

128x256x64 cooperative size, Swizzle size = 8

Blackwell Performance: Python vs. C++

FP16 I/O Group GEMM



- Perf gap

More optimized JIT codes b/c of

1) static cluster size
2) No insts related to residual if beta=0
3) Simplified prologue and epilogue

Testing spec: B100 180GB HBM3e, 148SM GPC-800MHz/DRAM 4GHz 850W

128x256/128x32/64x32/128x128/128x32 CTA Tilesize, Warp-specialized,

Cluster shape (C++) 2x1/2x1/2x1/1x1/1x1 vs. (Python)2x1/1x1/1x1/1x1/1x1

CUTLASS Python

CUTLASS comes to Python and is first class citizen with full support!

Initial beta release

- Release date: Q2’25
- Focus on GEMMs, including grouped GEMM
- Blackwell B200: support for major features
  - TMA, 2CTA MMA, TMEM
- Ada/Ampere/Hopper: experimental
- Schedulers and pipelines
- Jupyter notebook examples
  - Tutorial series on how to use the DSL
  - *Back to the basics* SGEMM tutorial series
  - Ampere TC GEMM and FA2
  - Hopper WGMMA GEMM
  - Blackwell GEMM tutorial series
  - Blackwell grouped GEMM
  - Blackwell FA2

What will come next

- GeForce RTX 50 Series
- Feature complete support for Blackwell including
  - Cluster Launch Control
  - Programming Dependent Launch
- Narrow-precision data types support and block-scaled MMAs
- EVT
- More examples of advanced fusion
- Convolutions
- Fully fledged Ahead-of-time compilation support
- Higher level abstractions & primitives
- More comprehensive documentation
- Graduate Ada, Ampere, Hopper from experimental

Acknowledgements

A great collaboration among various teams made it possible!

THANK YOU !