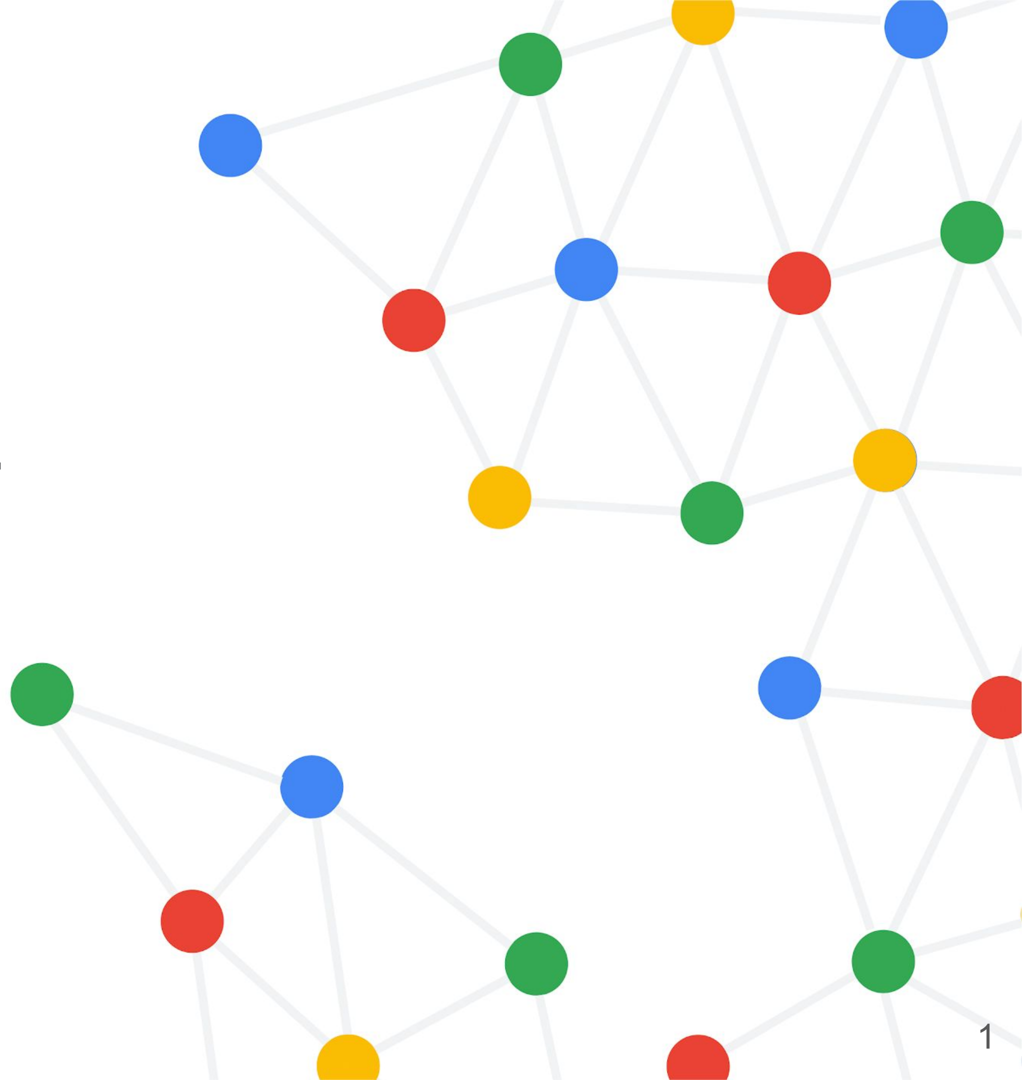
# Zero to Hero:
## Programming Nvidia Hopper with MLIR's NVGPU Dialect

**Guray Ozen**

10-11th April 24
EuroLLVM Meeting 2024

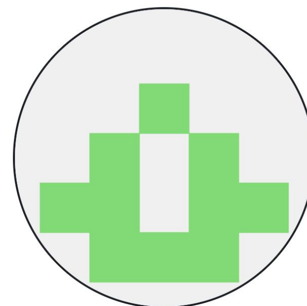
Google Research



# A big thank you to

**Quentin Colombet**
qcolombet

**Jacques Pienaar**
jpienaar · he/him

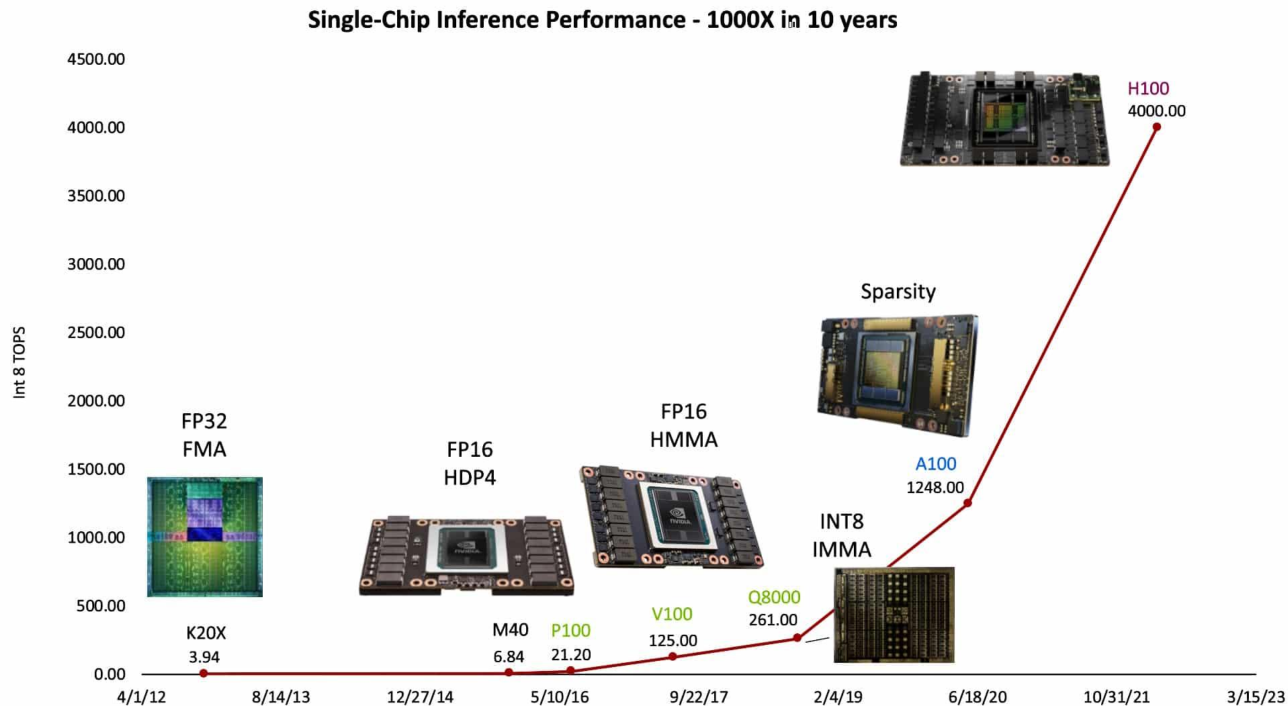


**Nicolas Vasilache**
nicolasvasilache

**Manish Gupta**
manishucsd

**Adam Paszke**
apaszke

# Huang’s Law [1, 2]




[1] https://en.wikipedia.org/wiki/Huang%27s_law
[2] Hardware for Deep Learning, Bill Dally, HotChips

# Huang's Law [1, 2]

[1] https://en.wikipedia.org/wiki/Huang%27s_law
[2] Hardware for Deep Learning, Bill Dally, HotChips

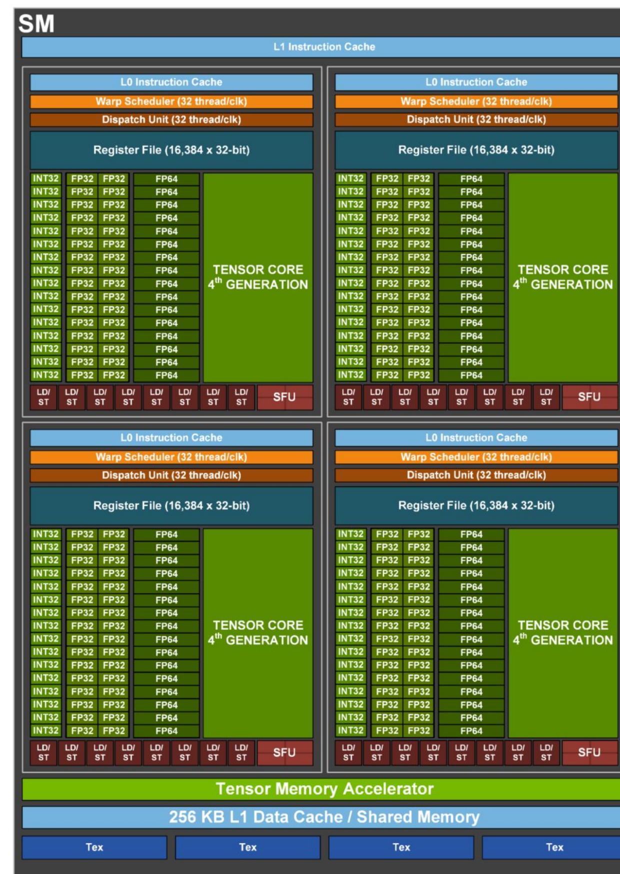
# Evolution in Hardware:
## NVIDIA Hopper Architecture

### 4th gen Tensor Core
- Warpgroup level (128 threads) PTX instructions
- Matrix A or B can be shared memory or registers
- Supports transpose for f16

### Thread Block Clusters
- Clustering helps reusing data on L2

### Tensor Memory Accelerator (TMA)
- Load a tile asynchronously
- Not wasting registers
- Swizzling 32b, 64b, 128b

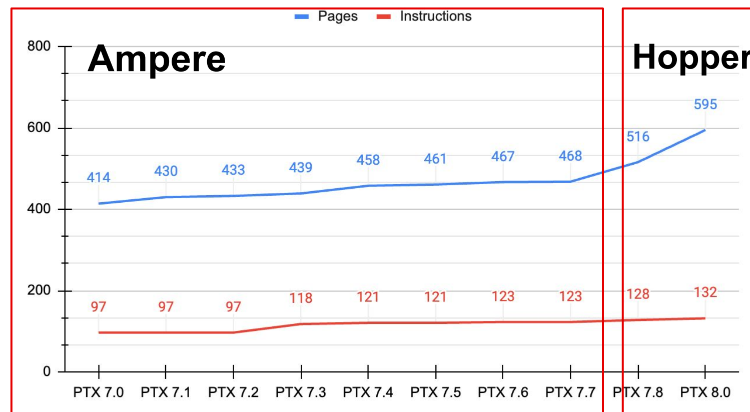
### Asynchronous Barriers
- Helps waiting TMA asynchronously

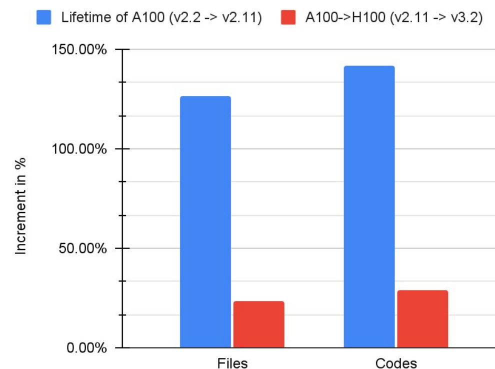


# Evolution in Software:
## PTX[1] & CUTLASS[2]

Significantly growed

1. Lifespan of Ampere (~2 years)
2. Hopper Architecture




Did MLIR & LLVM keep up?

*[1] Compared pages and table-2 in PTX pdf*
*[2] Used cloc for LoC*

# MLIR has gained Hopper Support

**NVGPU and NVVM Dialects**

🔥 Hopper GPU Support

**Performance**

🚀 MLIR has close performance to cuBLAS

**Upstream**

✅ All the work presented is fully upstreamed to MLIR

# What We Will Discover in Tutorial

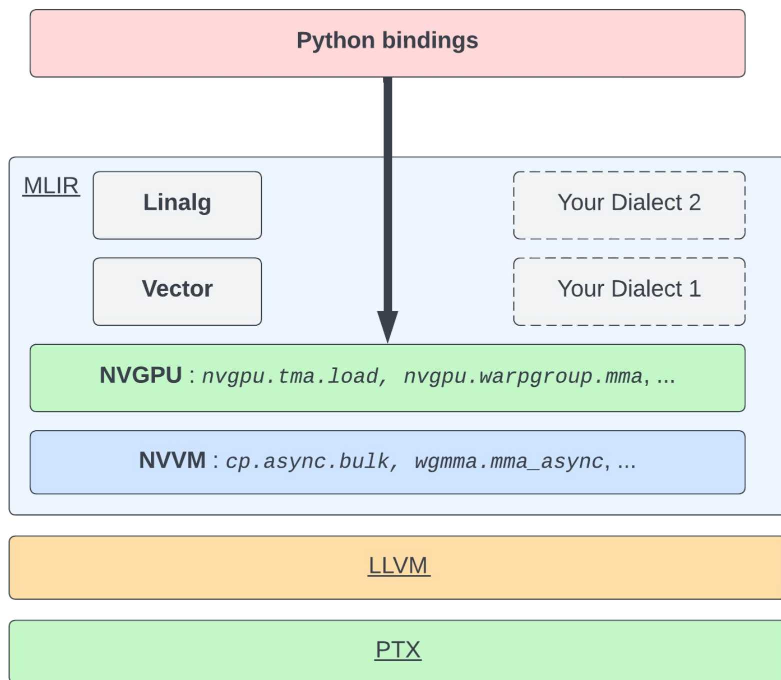
## Navigating Zero to Hero

Me no Hopper GPU

NVGPU dialect
*`Multistage`* Kernel
*`Warp Specialized`* Kernel
Get `cuBLAS` like Performance

# MLIR Upstream Dialect Layers

## Improved GPU, NVGPU, and NVVM Dialects




### NVGPU Dialect

- High level operations for Tensor Core, TMA
- NVGPU → NVVM

### NVVM Dialect

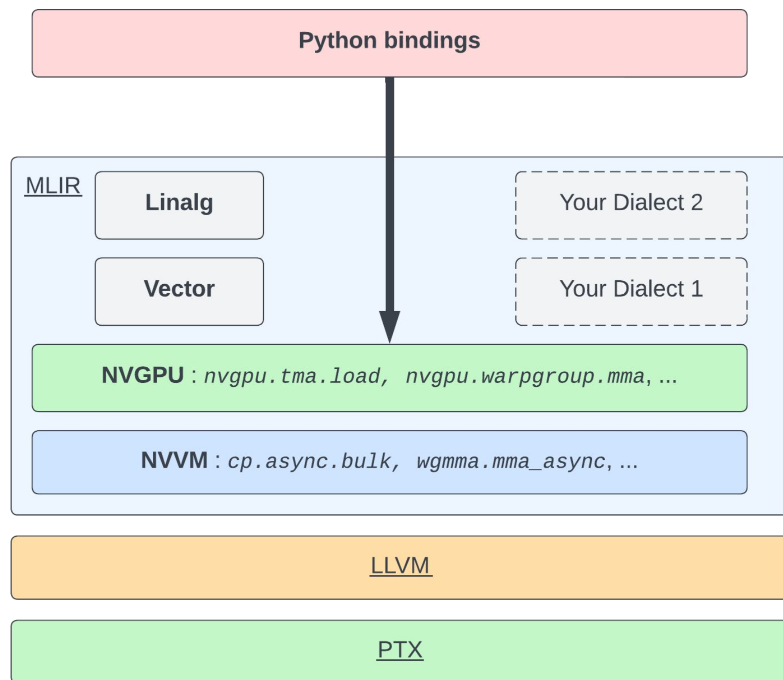
- Low level operations (closer to PTX)
- NVVM → PTX or LLVM intrinsic

### GPU Dialect

- Kernel launch, Cluster launch
- Driver communication

# MLIR Upstream Dialect Layers

## Let's program NVGPU with python bindings

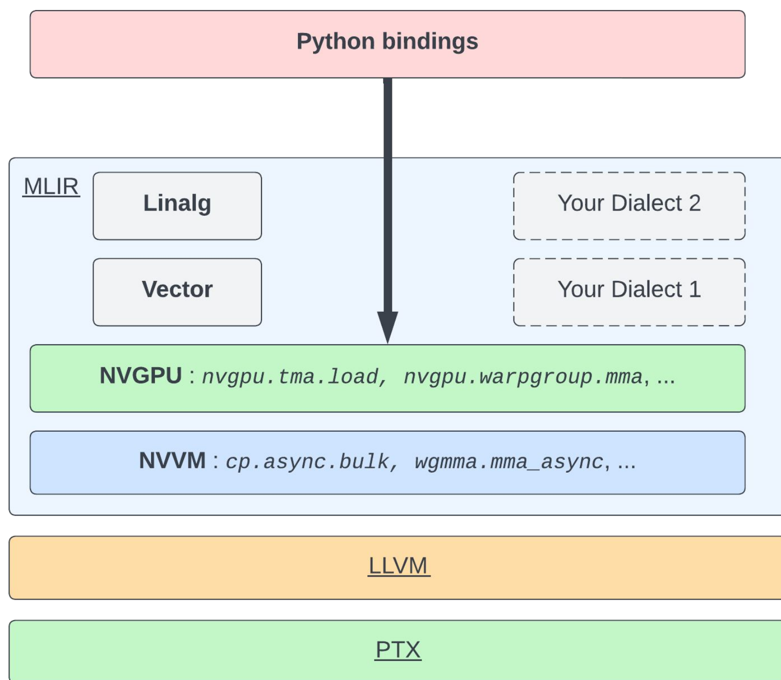



**Today we will program**

- Python → NVGPU → NVVM

# MLIR Upstream Dialect Layers

## Connect Your Dialect ⟶ NVGPU




**Today we will program**

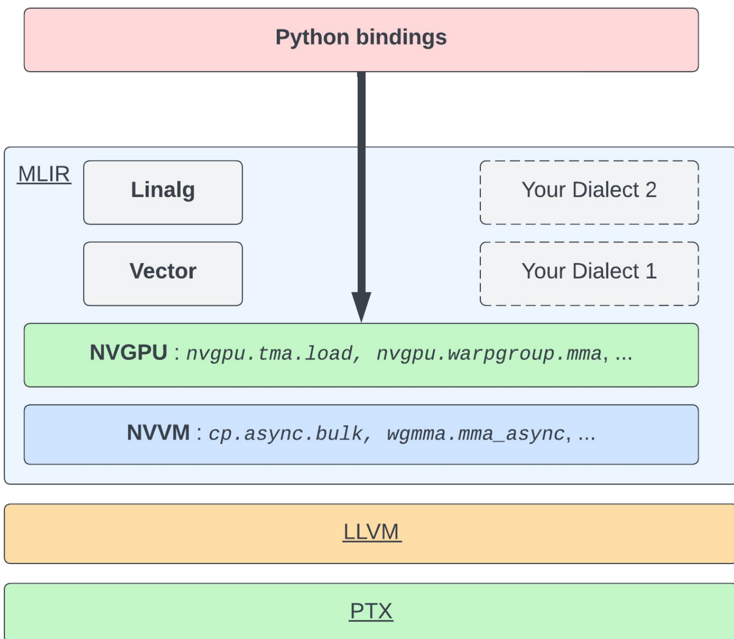
- Python ⟶ NVGPU ⟶ NVVM

One can lower other dialects into NVGPU

- Vector ⟶ NVGPU ⟶ NVVM
- Linalg ⟶ NVGPU ⟶ NVVM
- Your Dialect 1 ⟶ NVGPU ⟶ NVVM

# Py + MLIR vs CUTLASS

## Dialects vs Layer Comparison




| CUTLASS | MLIR NVGPU + Py |
|---|---|
| Device | `@NVDSL.mlir_func`<br>`def gemm(x, y, z):`<br>`  # Setups and Calls Kernel` |
| Kernel | `@NVDSL.mlir_gpu_launch(...)`<br>`def gemm_kernel()`<br>`  # Kernel Body` |
| Collective | Multistage:<br>`  def prologue() # has nvgpu OPs`<br>`  def mainloop() # has nvgpu OPs`<br>`  def epilogue() # has nvgpu OPs`<br><br>Warp Specialized:<br>`  def producer_loop() # has nvgpu OPs`<br>`  def consumer_loop() # has nvgpu OPs` |
| Atom | NVGPU Dialect |
| Thread | . |
| Intrinsic | NVVM Dialect |

→ Setup interface on Host

→ Launch kernel, calculate the grid and smem

→ Pipeline matmul, main loop, epilogue

→ Tensor core, TMA load/store

→ Numerical conversion, fast math, ...

→ PTX instruction

Google Research

# Tutorials : #87065 is about the be upstreamed

Codes are here: *{your-llvm-path}/mlir/test/Examples/nvgpu*

- **Ch0.py** ⟶ Hello World
- **Ch1.py** ⟶ 2D Saxpy
- **Ch2.py** ⟶ 2D Saxpy with TMA
- **Ch3.py** ⟶ GEMM 128x128x64 Tensor Core and TMA
- **Ch4.py** ⟶ GEMM Multistage
- **Ch5.py** ⟶ GEMM Warp Specialized
- **Ch6.py** ⟶ GEMM Warp Specialized Persistent ping-pong (WIP)

# NVDSL : Simplify NVGPU Dialect usage

We focus on Hopper Performance:

Simplifies:

- Simplify MLIR Host Function (`func.func`) IR Building
- JIT Compilation and Execution
- Operator Overloading with Arith Dialect for Readable Code
- Easy GPU IR (`gpu.launch`) Building

# **Ch1.py:** 2D SAXPY

## Single-Precision A·X Plus Y (SAXPY)

```
# Regular Python code
def saxpy(y, x, alpha):
 for i in range(256):
   for j in range(32):
       y[i, j] += alpha * x[i, j]



# Use numpy arrays
x = np.ones((256 32), np.float32)
y = np.ones((256 32), np.float32)
saxpy(x, y, 2.0)
```

# Ch1.py: 2D SAXPY

## Single-Precision A·X Plus Y (SAXPY)

```python
# Regular Python code
def saxpy(y, x, alpha):
 for i in range(256):  # -> blockIdx.x
   for j in range(32): # -> threadIdx.x
       y[i, j] += alpha * x[i, j]
```

```python
# Use numpy arrays
x = np.ones((256 32), np.float32)
y = np.ones((256 32), np.float32)
saxpy(x, y, 2.0)
```

Let's write in MLIR

Building IR with Py bindings

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
    # 1. Use MLIR GPU dialect to allocate and copy memory
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    # 2. Compute 2D SAXPY kernel
    @NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1))
    def saxpy_kernel():
        bidx = gpu.block_id(gpu.Dimension.x)
        tidx = gpu.thread_id(gpu.Dimension.x)
        x_val = memref.load(x_dev, [bidx, tidx])
        y_val = memref.load(y_dev, [bidx, tidx])
        y_val += x_val * alpha
        memref.store(y_val, y_dev, [bidx, tidx])
    saxpy_kernel()

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Ch1.py: 2D SAXPY

## Build Host IR

**This Decorator (`@NVDSL.mlir_func`) builds:**

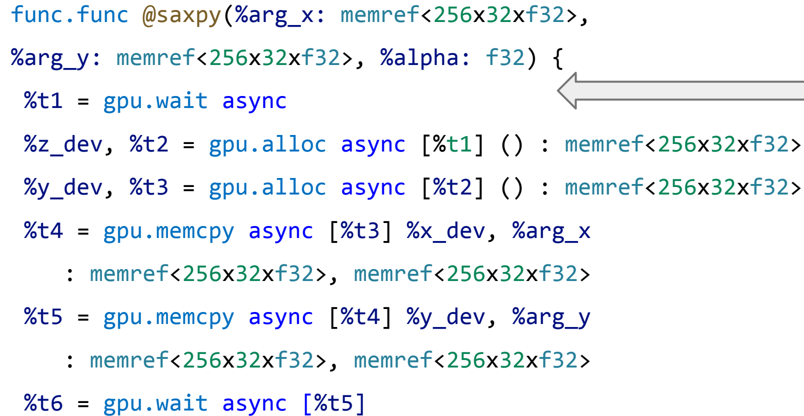
```
func.func @saxpy(%arg_x: memref<256x32xf32>,
                 %arg_y: memref<256x32xf32>,
                 %alpha: f32)
```

Building IR with Py bindings

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
    # 1. Use MLIR GPU dialect to allocate and copy memory
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    # 2. Compute 2D SAXPY kernel
    @NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1))
    def saxpy_kernel():
        bidx = gpu.block_id(gpu.Dimension.x)
        tidx = gpu.thread_id(gpu.Dimension.x)
        x_val = memref.load(x_dev, [bidx, tidx])
        y_val = memref.load(y_dev, [bidx, tidx])
        y_val += x_val * alpha
        memref.store(y_val, y_dev, [bidx, tidx])
    saxpy_kernel()

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# **Ch1.py:** 2D SAXPY

## Build Host IR



```
func.func @saxpy(%arg_x: memref<256x32xf32>,
%arg_y: memref<256x32xf32>, %alpha: f32) {
 %t1 = gpu.wait async
 %z_dev, %t2 = gpu.alloc async [%t1] () : memref<256x32xf32>
 %y_dev, %t3 = gpu.alloc async [%t2] () : memref<256x32xf32>
 %t4 = gpu.memcpy async [%t3] %x_dev, %arg_x
    : memref<256x32xf32>, memref<256x32xf32>
 %t5 = gpu.memcpy async [%t4] %y_dev, %arg_y
    : memref<256x32xf32>, memref<256x32xf32>
 %t6 = gpu.wait async [%t5]
```

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
   # 1. Use MLIR GPU dialect to allocate and copy memory
   t1 = gpu.wait(token_ty, [])
   x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
   y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
   t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
   t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
   t6 = gpu.wait(token_ty, [t5])

   # 2. Compute 2D SAXPY kernel
   @NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1))
   def saxpy_kernel():
       bidx = gpu.block_id(gpu.Dimension.x)
       tidx = gpu.thread_id(gpu.Dimension.x)
       x_val = memref.load(x_dev, [bidx, tidx])
       y_val = memref.load(y_dev, [bidx, tidx])
       y_val += x_val * alpha
       memref.store(y_val, y_dev, [bidx, tidx])
   saxpy_kernel()

   t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
   gpu.wait(token_ty, [t7])
```

# Ch1.py: 2D SAXPY

## Build Host IR

```
func.func @saxpy(%arg_x: memref<256x32xf32>,
%arg_y: memref<256x32xf32>, %alpha: f32) {
 %t1 = gpu.wait async
 %z_dev, %t2 = gpu.alloc async [%t1] () : memref<256x32xf32>
 %y_dev, %t3 = gpu.alloc async [%t2] () : memref<256x32xf32>
 %t4 = gpu.memcpy async [%t3] %x_dev, %arg_x
     : memref<256x32xf32>, memref<256x32xf32>
 %t5 = gpu.memcpy async [%t4] %y_dev, %arg_y
     : memref<256x32xf32>, memref<256x32xf32>
 %t6 = gpu.wait async [%t5]
```

```
 %t7 = gpu.memcpy async [%t6] %arg_y, %y_dev
     : memref<256x32xf32>, memref<256x32xf32>
 gpu.wait async [%t7]
```



```
@NVDSL.mlir_func
def saxpy(x, y, alpha):
   # 1. Use MLIR GPU dialect to allocate and copy memory
   t1 = gpu.wait(token_ty, [])
   x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
   y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
   t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
   t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
   t6 = gpu.wait(token_ty, [t5])

   # 2. Compute 2D SAXPY kernel
   @NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1))
   def saxpy_kernel():
       bidx = gpu.block_id(gpu.Dimension.x)
       tidx = gpu.thread_id(gpu.Dimension.x)
       x_val = memref.load(x_dev, [bidx, tidx])
       y_val = memref.load(y_dev, [bidx, tidx])
       y_val += x_val * alpha
       memref.store(y_val, y_dev, [bidx, tidx])
   saxpy_kernel()

   t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
   gpu.wait(token_ty, [t7])
```

# Ch1.py: 2D SAXPY

## Build GPU Kernel IR

**This Decorator (`@NVDSL.mlir_gpu_launch`) builds:**

```
gpu.launch blocks(256,1,1) threads(32, 1, 1) {

}
```

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
    # 1. Use MLIR GPU dialect to allocate and copy memory
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    # 2. Compute 2D SAXPY kernel
    @NVDSL.mlir_gpu_launch(grid=(M,1,1), block=(N,1,1))
    def saxpy_kernel():
        bidx = gpu.block_id(gpu.Dimension.x)
        tidx = gpu.thread_id(gpu.Dimension.x)
        x_val = memref.load(x_dev, [bidx, tidx])
        y_val = memref.load(y_dev, [bidx, tidx])
        y_val += x_val * alpha
        memref.store(y_val, y_dev, [bidx, tidx])
    saxpy_kernel()

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Ch1.py: 2D SAXPY

## Build GPU Kernel IR

**Inside the decorator function is the GPU Kernel:**

```
gpu.launch blocks(256,1,1) threads(32, 1, 1) {
 %bidx = gpu.block_id  x
 %tidx = gpu.thread_id  x
 %6 = memref.load %x_dev[%bidx, %tidx] : memref<256x32xf32>
 %7 = memref.load %y_dev[%bidx, %tidx] : memref<256x32xf32>
 %8 = arith.mulf %6, %alpha : f32
 %9 = arith.addf %7, %8 : f32
 memref.store %9, %y_dev[%bidx, %tidx] : memref<256x32xf32>
 gpu.terminator
}
```

**Building IR with Py bindings**

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
    # 1. Use MLIR GPU dialect to allocate and copy memory
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    # 2. Compute 2D SAXPY kernel
    @NVDSL.mlir_gpu_launch(grid=(M,1,1), block=(N,1,1))
    def saxpy_kernel():
        bidx = gpu.block_id(gpu.Dimension.x)
        tidx = gpu.thread_id(gpu.Dimension.x)
        x_val = memref.load(x_dev, [bidx, tidx])
        y_val = memref.load(y_dev, [bidx, tidx])
        y_val += x_val * alpha
        memref.store(y_val, y_dev, [bidx, tidx])
    saxpy_kernel()

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Ch1.py: 2D SAXPY

## Build GPU Kernel IR

**DSL overloads operators with `arith.*` dialect:**

```
gpu.launch blocks(256,1,1) threads(32, 1, 1) {
 %bidx = gpu.block_id  x
 %tidx = gpu.thread_id  x
 %6 = memref.load %x_dev[%bidx, %tidx] : memref<256x32xf32>
 %7 = memref.load %y_dev[%bidx, %tidx] : memref<256x32xf32>
 %8 = arith.mulf %6, %alpha : f32
 %9 = arith.addf %7, %8 : f32
 memref.store %9, %y_dev[%bidx, %tidx] : memref<256x32xf32>
 gpu.terminator
}
```

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
    # 1. Use MLIR GPU dialect to allocate and copy memory
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    # 2. Compute 2D SAXPY kernel
    @NVDSL.mlir_gpu_launch(grid=(256,1,1), block=(32,1,1))
    def saxpy_kernel():
        bidx = gpu.block_id(gpu.Dimension.x)
        tidx = gpu.thread_id(gpu.Dimension.x)
        x_val = memref.load(x_dev, [bidx, tidx])
        y_val = memref.load(y_dev, [bidx, tidx])
        y_val += x_val * alpha
        memref.store(y_val, y_dev, [bidx, tidx])
    saxpy_kernel()

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Ch1.py: 2D SAXPY

## Python calls MLIR func and pass parameters

Decorator (`@NVDSL.mlir_func`):

- `numpy arrays -> memref`
- JIT compiles
- `Executes`

```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
  # MLIR function body …
```

```python
# 3. Pass numpy arrays to MLIR
alpha = 2.0
x = np.ones((256, 32), np.float32)
y = np.ones((256, 32), np.float32)
ref = np.ones((256, 32), np.float32)
saxpy(x, y, alpha)
```

```python
#  4. Verify MLIR with reference computation
ref += x * alpha
np.testing.assert_allclose(y, ref, rtol=5e-03, atol=1e-01)
print("PASS")
# CHECK-NOT: Mismatched elements
```

# Ch1.py: 2D SAXPY

## Python calls MLIR func and pass parameters

```
@NVDSL.mlir_func
def saxpy(x, y, alpha):
  # MLIR function body ...
```

Decorator (`@NVDSL.mlir_func`):

- `numpy arrays -> memref`
- JIT compiles
- Executes

```
# 3. Pass numpy arrays to MLIR
alpha = 2.0
x = np.ones((256, 32), np.float32)
y = np.ones((256, 32), np.float32)
ref = np.ones((256, 32), np.float32)
saxpy(x, y, alpha)
```

Verify the Result

```
#  4. Verify MLIR with reference computation
ref += x * alpha
np.testing.assert_allclose(y, ref, rtol=5e-03, atol=1e-01)
print("PASS")
# CHECK-NOT: Mismatched elements
```

# Ch1.py: 2D SAXPY

## Execute

```
$> python Ch1.py
  PASS
```

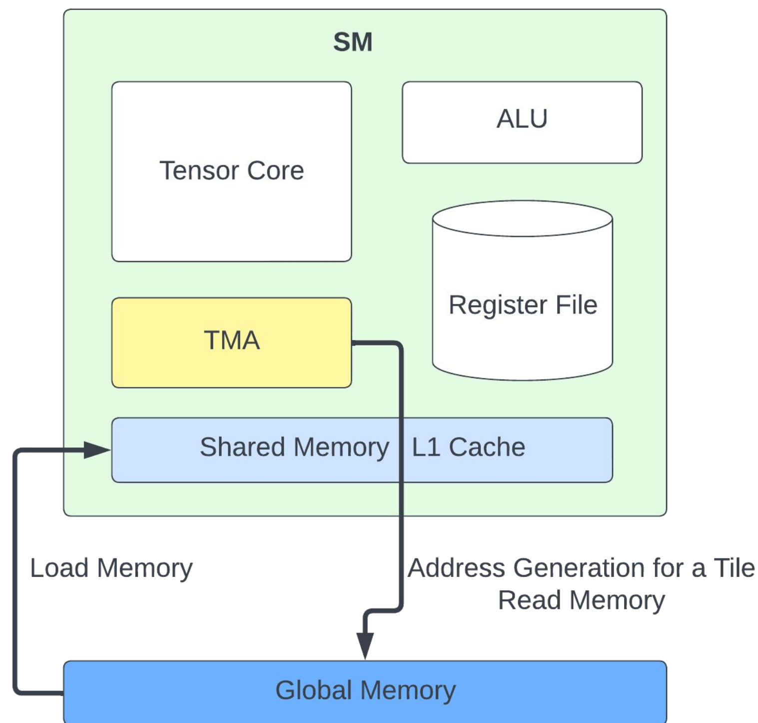
```python
@NVDSL.mlir_func
def saxpy(x, y, alpha):
  # MLIR function body ...


# 3. Pass numpy arrays to MLIR
alpha = 2.0
x = np.ones((256, 32), np.float32)
y = np.ones((256, 32), np.float32)
ref = np.ones((256, 32), np.float32)
saxpy(x, y, alpha)

#  4. Verify MLIR with reference computation
ref += x * alpha
np.testing.assert_allclose(y, ref, rtol=5e-03, atol=1e-01)
print("PASS")
# CHECK-NOT: Mismatched elements
```

# **Ch2.py:** 2D SAXPY with TMA

## Now let TMA load the data

```
# Non-Regular Python code
def saxpy(y, x, alpha):
  x_smem = # TMA loads ‘x’ <32xf32> to shared memory
  y_smem = # TMA loads ‘y’ <32xf32> to shared memory
  for i in range(256):
    for j in range(32):
        y[i, j] = y_tma[i, j] + alpha * x_smem[i, j]
```

**Let’s use NVIDIA Hopper TMA to load data**



# Ch2.py: 2D SAXPY with TMA

## TMA OPs and TMA class in NVDSL

### `nvgpu.tma.create.descriptor`

- Host calls the CUDA driver, it triggers the function cuTensorMapEncodeTiled.

### `nvgpu.tma.prefetch`

- Prefetch TMA descriptor to L1 cache

### `nvgpu.tma.async.load`

- Loads 1D - 5D tile
- Supports predicated execution
- Lowered: `nvvm.cp.async.bulk.tensor.shared.cluster.global`



```python
class TMA:
 def __init__(self, shape, memref_ty,
   swizzle=nvgpu.TensorMapSwizzleKind.SWIZZLE_NONE,
   l2promo=nvgpu.TensorMapL2PromoKind.L2PROMO_NONE,
   oob=nvgpu.TensorMapOOBKind.OOB_ZERO,
   interleave=nvgpu.TensorMapInterleaveKind.INTERLEAVE_NONE,
 ): # init ...

 @property
 def tensormap_descriptor_ty(self):
   memref_str = f"memref<{self.tma_shape[0]}x{self.tma_shape[1]}, 3>"
   parse_str = f"!nvgpu.tensormap.descriptor<tensor = {memref_str},swizzle = {self.swizzle},\
                 l2promo = {self.l2promo},oob = {self.oob},interleave = {self.interleave}>"
   return ir.Type.parse(parse_str)

 def create_descriptor(self, device_ptr):
   tma_descriptor_ty = self.tensormap_descriptor_ty
   device_unranked_memref = memref.CastOp(ir.UnrankedMemRefType.get(
                                              self.memref_ty.element_type,
                                              self.memref_ty.memory_space), device_ptr)
    self.tma_descriptor = nvgpu.TmaCreateDescriptorOp(tma_descriptor_ty,
                                device_unranked_memref,map(const, self.tma_shape))
def prefetch(self, predicate=None):
   nvgpu.tma_prefetch_descriptor(self.tma_descriptor, predicate=predicate)

def load(self, dest, mbarrier: Mbarriers, coords=[0, 0], predicate=None):
   nvgpu.TmaAsyncLoadOp(dest,mbarrier.mbar_group_op,self.tma_descriptor,
         coordinates=map(const, coords),mbarId=mbarrier.id_op, predicate=predicate,
      )
```

# Ch2.py: 2D SAXPY with TMA

## TMA OPs and TMA class in NVDSL

`nvgpu.tma.create.descriptor`

- Host calls the CUDA driver, it triggers the function cuTensorMapEncodeTiled.

`nvgpu.tma.prefetch`

- Prefetch TMA descriptor to L1 cache

`nvgpu.tma.async.load`

- Loads 1D - 5D tile
- Supports predicated execution
- Lowered: `nvvm.cp.async.bulk.tensor.shared.cluster.global`

**Building IR with Py bindings**

```
class TMA:
 def __init__(self, shape, memref_ty,
   swizzle=nvgpu.TensorMapSwizzleKind.SWIZZLE_NONE,
   l2promo=nvgpu.TensorMapL2PromoKind.L2PROMO_NONE,
   oob=nvgpu.TensorMapOOBKind.OOB_ZERO,
   interleave=nvgpu.TensorMapInterleaveKind.INTERLEAVE_NONE,
 ): # init ...

 @property
 def tensormap_descriptor_ty(self):
   memref_str = f"memref<{self.tma_shape[0]}x{self.tma_shape[1]}, 3>"
   parse_str = f"!nvgpu.tensormap.descriptor<tensor = {memref_str},swizzle = {self.swizzle},\
                 l2promo = {self.l2promo},oob = {self.oob},interleave = {self.interleave}>"
   return ir.Type.parse(parse_str)

 def create_descriptor(self, device_ptr):
   tma_descriptor_ty = self.tensormap_descriptor_ty
   device_unranked_memref = memref.CastOp(ir.UnrankedMemRefType.get(
                                             self.memref_ty.element_type,
                                             self.memref_ty.memory_space), device_ptr)
    self.tma_descriptor = nvgpu.TmaCreateDescriptorOp(tma_descriptor_ty,
                                         device_unranked_memref,map(const, self.tma_shape))
def prefetch(self, predicate=None):
   nvgpu.tma_prefetch_descriptor(self.tma_descriptor, predicate=predicate)

def load(self, dest, mbarrier: Mbarriers, coords=[0, 0], predicate=None):
   nvgpu.TmaAsyncLoadOp(dest,mbarrier.mbar_group_op,self.tma_descriptor,
            coordinates=map(const, coords),mbarId=mbarrier.id_op, predicate=predicate,
        )
```

# Ch2.py: 2D SAXPY with TMA

## TMA OPs and TMA class in NVDSL

`nvgpu.tma.create.descriptor`

- Host calls the CUDA driver, it triggers the function cuTensorMapEncodeTiled.

`nvgpu.tma.prefetch`

- Prefetch TMA descriptor to L1 cache

`nvgpu.tma.async.load`

- Loads 1D - 5D tile
- Supports predicated execution
- Lowered: `nvvm.cp.async.bulk.tensor.shared.cluster.global`

**Building IR with Py bindings**

```python
class TMA:
 def __init__(self, shape, memref_ty,
   swizzle=nvgpu.TensorMapSwizzleKind.SWIZZLE_NONE,
   l2promo=nvgpu.TensorMapL2PromoKind.L2PROMO_NONE,
   oob=nvgpu.TensorMapOOBKind.OOB_ZERO,
   interleave=nvgpu.TensorMapInterleaveKind.INTERLEAVE_NONE,
 ): # init ...

 @property
 def tensormap_descriptor_ty(self):
   memref_str = f"memref<{self.tma_shape[0]}x{self.tma_shape[1]}, 3>"
   parse_str = f"!nvgpu.tensormap.descriptor<tensor = {memref_str},swizzle = {self.swizzle},\
                l2promo = {self.l2promo},oob = {self.oob},interleave = {self.interleave}>"
   return ir.Type.parse(parse_str)

 def create_descriptor(self, device_ptr):
   tma_descriptor_ty = self.tensormap_descriptor_ty
   device_unranked_memref = memref.CastOp(ir.UnrankedMemRefType.get(
                                               self.memref_ty.element_type,
                                               self.memref_ty.memory_space), device_ptr)
    self.tma_descriptor = nvgpu.TmaCreateDescriptorOp(tma_descriptor_ty,
                                   device_unranked_memref,map(const, self.tma_shape))
def prefetch(self, predicate=None):
   nvgpu.tma_prefetch_descriptor(self.tma_descriptor, predicate=predicate)

def load(self, dest, mbarrier: Mbarriers, coords=[0, 0], predicate=None):
   nvgpu.TmaAsyncLoadOp(dest,mbarrier.mbar_group_op,self.tma_descriptor,
            coordinates=map(const, coords),mbarId=mbarrier.id_op, predicate=predicate,
       )
```

# Ch2.py: 2D SAXPY with TMA

## TMA OPs and TMA class in NVDSL

`nvgpu.tma.create.descriptor`

- Host calls the CUDA driver, it triggers the function cuTensorMapEncodeTiled.

`nvgpu.tma.prefetch`

- Prefetch TMA descriptor to L1 cache

`nvgpu.tma.async.load`

- Loads 1D - 5D tile
- Supports predicated execution
- Lowered: `nvvm.cp.async.bulk.tensor.shared.cluster.global`

**Building IR with Py bindings**

```python
class TMA:
 def __init__(self, shape, memref_ty,
   swizzle=nvgpu.TensorMapSwizzleKind.SWIZZLE_NONE,
   l2promo=nvgpu.TensorMapL2PromoKind.L2PROMO_NONE,
   oob=nvgpu.TensorMapOOBKind.OOB_ZERO,
   interleave=nvgpu.TensorMapInterleaveKind.INTERLEAVE_NONE,
 ): # init ...

 @property
 def tensormap_descriptor_ty(self):
   memref_str = f"memref<{self.tma_shape[0]}x{self.tma_shape[1]}, 3>"
   parse_str = f"!nvgpu.tensormap.descriptor<tensor = {memref_str},swizzle = {self.swizzle},\
                 l2promo = {self.l2promo},oob = {self.oob},interleave = {self.interleave}>"
   return ir.Type.parse(parse_str)

 def create_descriptor(self, device_ptr):
   tma_descriptor_ty = self.tensormap_descriptor_ty
   device_unranked_memref = memref.CastOp(ir.UnrankedMemRefType.get(
                                            self.memref_ty.element_type,
                                            self.memref_ty.memory_space), device_ptr)
    self.tma_descriptor = nvgpu.TmaCreateDescriptorOp(tma_descriptor_ty,
                                device_unranked_memref,map(const, self.tma_shape))
def prefetch(self, predicate=None):
   nvgpu.tma_prefetch_descriptor(self.tma_descriptor, predicate=predicate)

def load(self, dest, mbarrier: Mbarriers, coords=[0, 0], predicate=None):
   nvgpu.TmaAsyncLoadOp(dest,mbarrier.mbar_group_op,self.tma_descriptor,
            coordinates=map(const, coords),mbarId=mbarrier.id_op, predicate=predicate,
      )
```

# Ch2.py: 2D SAXPY with TMA

## Mbarrier OPs and Mbarrier class in NVDSL

### `nvgpu.mbarrier.create`

- Allows creating multiple mbarriers
  - `%mbarGroup = nvgpu.mbarrier.create <...,`
    `num_barriers = 7>`

### `nvgpu.mbarrier.init | arrive | try_wait`

- Convenient access to mbarriers with SSA index. Ideal for handling multiple barriers within a loop
  - `nvgpu.mbarrier.init %mbarGroup[%mbar_id]`
- Provides support for predication
  - `nvgpu.mbarrier.expect_tx`
    `%mbarGroup[%mbar_id] predicate = %tidx0`

```python
class Mbarriers:
 def __init__(self, number_of_barriers=1):
   self.mbar_ty = ...
   self.number_of_barriers = number_of_barriers
   self.mbar_group_op = nvgpu.mbarrier_create(self.mbar_ty)
 def __getitem__(self, key):
   self.id_op = const(key)
   return self
 def init(self, count: int, predicate=None):
   count_op = const(count)
   nvgpu.mbarrier_init(self.mbar_group_op, count_op, self.id_op...)

 def arrive(self, txcount: int = 0, predicate=None):
   if txcount != 0:
     txcount_op = const(txcount)
     nvgpu.mbarrier_arrive_expect_tx(
       self.mbar_group_op, txcount_op, self.id_op, predicate=predicate
     )

 def try_wait(self, phase: bool = False, ticks: int = 10000000):
   nvgpu.MBarrierTryWaitParityOp(...)
```

# Ch2.py: 2D SAXPY with TMA

## Mbarrier OPs and Mbarrier class in NVDSL

### nvgpu.mbarrier.create

- Allows creating multiple mbarriers
  - `%mbarGroup = nvgpu.mbarrier.create <...,` `num_barriers = 7>`

### nvgpu.mbarrier.init | arrive | try_wait

- Convenient access to mbarriers with SSA index. Ideal for handling multiple barriers within a loop
  - `nvgpu.mbarrier.init %mbarGroup[%mbar_id]`
- Provides support for predication
  - `nvgpu.mbarrier.expect_tx %mbarGroup[%mbar_id] predicate = %tidx0`

```python
class Mbarriers:
 def __init__(self, number_of_barriers=1):
   self.mbar_ty = ...
   self.number_of_barriers = number_of_barriers
   self.mbar_group_op = nvgpu.mbarrier_create(self.mbar_ty)
 def __getitem__(self, key):
   self.id_op = const(key)
   return self
 def init(self, count: int, predicate=None):
   count_op = const(count)
   nvgpu.mbarrier_init(self.mbar_group_op, count_op, self.id_op...)

 def arrive(self, txcount: int = 0, predicate=None):
   if txcount != 0:
     txcount_op = const(txcount)
     nvgpu.mbarrier_arrive_expect_tx(
       self.mbar_group_op, txcount_op, self.id_op, predicate=predicate
     )

 def try_wait(self, phase: bool = False, ticks: int = 10000000):
   nvgpu.MBarrierTryWaitParityOp(...)
```

# Ch2.py: 2D SAXPY with TMA

## Mbarrier OPs and Mbarrier class in NVDSL

`nvgpu.mbarrier.create`

- Allows creating multiple mbarriers
    - `%mbarGroup = nvgpu.mbarrier.create <...,`
      `num_barriers = 7>`

`nvgpu.mbarrier.init | arrive | try_wait`

- Convenient access to mbarriers with SSA index. Ideal for handling multiple barriers within a loop
    - `nvgpu.mbarrier.init %mbarGroup[%mbar_id]`
- Provides support for predication
    - `nvgpu.mbarrier.expect_tx`
      `%mbarGroup[%mbar_id] predicate = %tidx0`

```python
class Mbarriers:
  def __init__(self, number_of_barriers=1):
    self.mbar_ty = ...
    self.number_of_barriers = number_of_barriers
    self.mbar_group_op = nvgpu.mbarrier_create(self.mbar_ty)
  def __getitem__(self, key):
    self.id_op = const(key)
    return self
  def init(self, count: int, predicate=None):
    count_op = const(count)
    nvgpu.mbarrier_init(self.mbar_group_op, count_op, self.id_op…)

  def arrive(self, txcount: int = 0, predicate=None):
    if txcount != 0:
      txcount_op = const(txcount)
      nvgpu.mbarrier_arrive_expect_tx(
        self.mbar_group_op, txcount_op, self.id_op, predicate=predicate
      )

  def try_wait(self, phase: bool = False, ticks: int = 10000000):
    nvgpu.MBarrierTryWaitParityOp(...)
```

# **Ch2.py:** 2D SAXPY with TMA

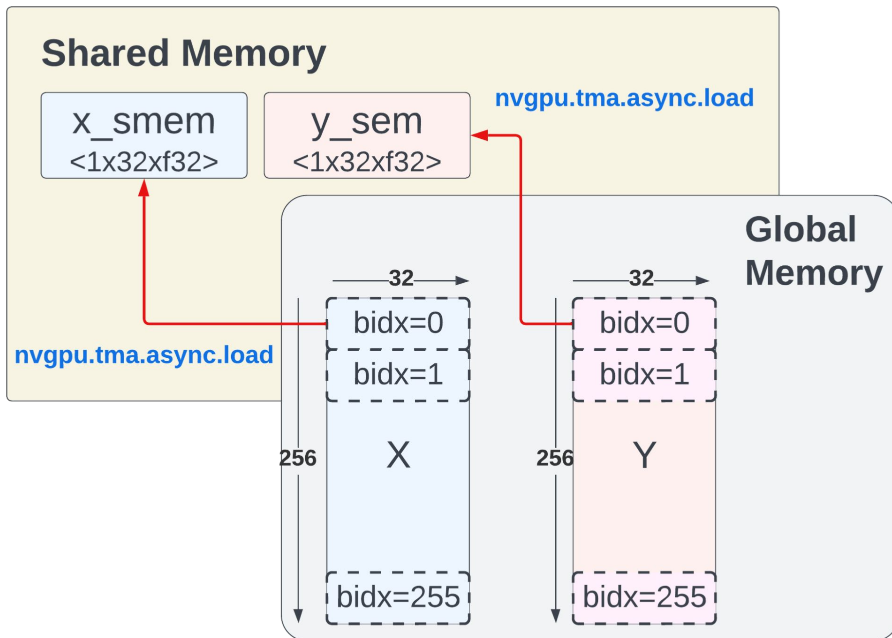


Start Building Host IR

```
@NVDSL.mlir_func
def saxpy_tma(x, y, alpha):
    token_ty = ir.Type.parse("!gpu.async.token")
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    x_tma = TMA((1,32), x.type)
    y_tma = TMA((1,32), y.type)
    x_tma.create_descriptor(x_dev)
    y_tma.create_descriptor(y_dev)

   @NVDSL.mlir_gpu_launch(grid=(M,1,1),block=(N,1,1),smem=256)
   def saxpy_tma_kernel():
     # Kernel Body (next slides)

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Ch2.py: 2D SAXPY with TMA

## Generated IR




```python
@NVDSL.mlir_func
def saxpy_tma(x, y, alpha):
    token_ty = ir.Type.parse("!gpu.async.token")
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    x_tma = TMA((1,32), x.type)
    y_tma = TMA((1,32), y.type)
    x_tma.create_descriptor(x_dev)
    y_tma.create_descriptor(y_dev)

   @NVDSL.mlir_gpu_launch(grid=(M,1,1),block=(N,1,1),smem=256)
   def saxpy_tma_kernel():
     # Kernel Body (next slides)

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Ch2.py: 2D SAXPY with TMA

## Generated IR

```
%3 = nvgpu.tma.create.descriptor %x box[%c1, %c32]
    : memref<*xf32>
    -> <tensor = memref<1x32xf32, 3>,
        swizzle = none,
        l2promo = none,
        oob = zero,
        interleave = none>
```

```
%4 = nvgpu.tma.create.descriptor %y box[%c1, %c32]
    : memref<*xf32>
    -> <tensor = memref<1x32xf32, 3>,
        swizzle = none,
        l2promo = none,
        oob = zero,
        interleave = none>
```

```
@NVDSL.mlir_func
def saxpy_tma(x, y, alpha):
    token_ty = ir.Type.parse("!gpu.async.token")
    t1 = gpu.wait(token_ty, [])
    x_dev, t2 = gpu.alloc(x.type, token_ty, [t1], [], [])
    y_dev, t3 = gpu.alloc(y.type, token_ty, [t2], [], [])
    t4 = gpu.memcpy(token_ty, [t3], x_dev, x)
    t5 = gpu.memcpy(token_ty, [t4], y_dev, y)
    t6 = gpu.wait(token_ty, [t5])

    x_tma = TMA((1,32), x.type)
    y_tma = TMA((1,32), y.type)
    x_tma.create_descriptor(x_dev)
    y_tma.create_descriptor(y_dev)

  @NVDSL.mlir_gpu_launch(grid=(M,1,1),block=(N,1,1),smem=256)
  def saxpy_tma_kernel():
     # Kernel Body (next slides)

    t7 = gpu.memcpy(token_ty, [t6], y, y_dev)
    gpu.wait(token_ty, [t7])
```

# Lowering `nvgpu.tma.create.descriptor`

## Calls MLIR Runtime `@mgpuTensorMapEncodeTiledMemref`

```
%3 = nvgpu.tma.create.descriptor %x box[%c1, %c32]
    : memref<*xf32>
    -> <tensor = memref<1x32xf32, 3>,
        swizzle = none,
        l2promo = none,
        oob = zero,
        interleave = none>
```

--convert-nvgpu-to-nvvm Pass

```
%memref, %asyncToken = gpu.alloc async [%6] () : memref<256x32xf32>
%7 = gpu.memcpy async [%asyncToken_1] %memref, %arg0 : memref<256x32xf32>, memref<256x32xf32>

%cast = memref.cast %memref : memref<256x32xf32> to memref<*xf32>
%9 = builtin.unrealized_conversion_cast %cast : memref<*xf32> to !llvm.struct<(i64, ptr)>
%10 = llvm.mlir.constant(7 : i32) : i64
%11 = llvm.extractvalue %9[0] : !llvm.struct<(i64, ptr)>
%12 = llvm.extractvalue %9[1] : !llvm.struct<(i64, ptr)>
%13 = llvm.mlir.constant(5 : i32) : i64
%14 = llvm.alloca %13 x i64 : (i64) -> !llvm.ptr
%15 = llvm.mlir.constant(0 : i32) : i64
%16 = llvm.getelementptr %14[%15] : (!llvm.ptr, i64) -> !llvm.ptr, !llvm.ptr
llvm.store %5, %16 : i64, !llvm.ptr
%17 = llvm.mlir.constant(1 : i32) : i64
%18 = llvm.getelementptr %14[%17] : (!llvm.ptr, i64) -> !llvm.ptr, !llvm.ptr
llvm.store %4, %18 : i64, !llvm.ptr
%19 = llvm.mlir.constant(0 : i32) : i64
%20 = llvm.mlir.constant(0 : i32) : i64
%21 = llvm.mlir.constant(0 : i32) : i64
%22 = llvm.mlir.constant(0 : i32) : i64
%23 = llvm.call @mgpuTensorMapEncodeTiledMemref(%11, %12, %10, %19, %20, %21, %22, %14) : (i64,
!llvm.ptr, i64, i64, i64, i64, i64, !llvm.ptr) -> !llvm.ptr
```

# Inside `@mgpuTensorMapEncodeTiledMemref`

## Calls CUDA driver *cuTensorMapEncodeTiled*

```
extern "C" MLIR_CUDA_WRAPPERS_EXPORT void mgpuTensorMapEncodeTiled(...) {
 CUDA_REPORT_IF_ERROR(cuTensorMapEncodeTiled(tensorMap, tensorDataType, tensorRank, globalAddress,
   globalDim, globalStrides, boxDim, elementStrides, interleave, swizzle, l2Promotion, oobFill));
 }

extern "C" MLIR_CUDA_WRAPPERS_EXPORT void *mgpuTensorMapEncodeTiledMemref(...
) {
 CUtensorMap tensorMap;
 ...
 mgpuTensorMapEncodeTiled(&tensorMap, tensorDataType, tensorRank32,globalAddress, globalDim,
globalStrides, boxDim, elementStrides, interleave, swizzle, l2Promotion, oobFill);
 // Copy created tensor map to device
 CUdeviceptr dTensorMap;
 CUDA_REPORT_IF_ERROR(cuMemAlloc(&dTensorMap, sizeof(CUtensorMap)));
 CUDA_REPORT_IF_ERROR(cuMemcpy(dTensorMap,
                               reinterpret_cast<CUdeviceptr>(&tensorMap),
                               sizeof(CUtensorMap)));
 return reinterpret_cast<void *>(dTensorMap);
}
```

CUDA driver call generates TMA descriptor (aka *CUtensorMap **)

cuMemAlloc & cuMemcpy

# **Ch2.py:** 2D SAXPY with TMA

Start Building GPU IR

Building IR with Py bindings

```
@NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1), smem=sz_x_y)
def saxpy_tma_kernel():
    bidx = gpu.block_id(gpu.Dimension.x)
    tidx = gpu.thread_id(gpu.Dimension.x)
    isThread0 = tidx == 0

    # 1. Create and initialize asynchronous transactional barrier (mbarrier)
    mbar_group = Mbarriers(number_of_barriers=1)
    mbar_group[0].init(1, predicate=isThread0)

    # 2. Execute Tensor Memory Accelerator (TMA) Load
    x_smem = get_dynamic_shared_memory((1, N), T.f32())
    y_smem = get_dynamic_shared_memory((1, N), T.f32(), offset=M * N * 2)
    x_tma.load(x_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    y_tma.load(y_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    mbar_group[0].arrive(txcount=size_x + size_y, predicate=isThread0)

    # 3. Wait for completion of TMA load with mbarrier
    mbar_group[0].try_wait()

    x_val = memref.load(x_smem, [0, tidx])
    y_val = memref.load(y_smem, [0, tidx])
    # SAXPY: y[i] += a * x[i];
    y_val += x_val * alpha
    memref.store(y_val, y_dev, [bidx, tidx])
```

# Ch2.py: 2D SAXPY with TMA

## Generated IR

```
gpu.launch blocks(256,1,1) threads(32,1,1) … {
  %bdimx = gpu.block_id  x
  %tidx = gpu.thread_id  x
  %isThread0 = arith.cmpi eq, %thread_id_x, %c0 : index

  // 1. Create and initialize mbarrier
  %bar = nvgpu.mbarrier.create -> <memorySpace = #gpu.address_space<workgroup>>
  nvgpu.mbarrier.init %bar[%c0], %c1, predicate = %isThread0
                      : <memorySpace = #gpu.address_space<workgroup>>
```

```python
@NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1), smem=sz_x_y)
def saxpy_tma_kernel():
    bidx = gpu.block_id(gpu.Dimension.x)
    tidx = gpu.thread_id(gpu.Dimension.x)
    isThread0 = tidx == 0

    # 1. Create and initialize asynchronous transactional barrier (mbarrier)
    mbar_group = Mbarriers(number_of_barriers=1)
    mbar_group[0].init(1, predicate=isThread0)

    # 2. Execute Tensor Memory Accelerator (TMA) Load
    x_smem = get_dynamic_shared_memory((1, N), T.f32())
    y_smem = get_dynamic_shared_memory((1, N), T.f32(), offset=M * N * 2)
    x_tma.load(x_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    y_tma.load(y_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    mbar_group[0].arrive(txcount=size_x + size_y, predicate=isThread0)

    # 3. Wait for completion of TMA load with mbarrier
    mbar_group[0].try_wait()

    x_val = memref.load(x_smem, [0, tidx])
    y_val = memref.load(y_smem, [0, tidx])
    # SAXPY: y[i] += a * x[i];
    y_val += x_val * alpha
    memref.store(y_val, y_dev, [bidx, tidx])
```

# Ch2.py: 2D SAXPY with TMA

## Generated IR

```
// 2. Execute Tensor Memory Accelerator (TMA) Load
%8 = gpu.dynamic_shared_memory : memref<?xi8, #gpu.address_space<workgroup>>
%x_smem = memref.view %8[%c0][] : memref<?xi8, #gpu.address_space<workgroup>> to memref<256x32xf32, #gpu.address_space<workgroup>>
%y_smem = memref.view %8[%c128][] : memref<?xi8, #gpu.address_space<workgroup>> to memref<256x32xf32, #gpu.address_space<workgroup>>

nvgpu.tma.async.load %desc_x[%c0, %bidx], %bar[%c0] to %x_smem,
                predicate = %isThread0
: <tensor = memref<256x32xf32, 3>, swizzle = none, l2promo = none, oob = zero, interleave = none>, <memorySpace = #gpu.address_space<workgroup>> -> memref<256x32xf32, #gpu.address_space<workgroup>>

nvgpu.tma.async.load %desc_y[%c0, %bidx], %bar[%c0] to %y_smem,
                predicate = %isThread0
: <tensor = memref<256x32xf32, 3>, swizzle = none, l2promo = none, oob = zero, interleave = none>, <memorySpace = #gpu.address_space<workgroup>> -> memref<256x32xf32, #gpu.address_space<workgroup>>

nvgpu.mbarrier.arrive.expect_tx %bar[%c0], %c256,
                predicate = %isThread0
: <memorySpace = #gpu.address_space<workgroup>>
```

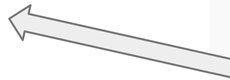
*Bytes expected to be loaded by TMA*
*txcount = sizeof(x + y) = 256byte*



```python
@NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1), smem=sz_x_y)
def saxpy_tma_kernel():
    bidx = gpu.block_id(gpu.Dimension.x)
    tidx = gpu.thread_id(gpu.Dimension.x)
    isThread0 = tidx == 0

    # 1. Create and initialize asynchronous transactional barrier (mbarrier)
    mbar_group = Mbarriers(number_of_barriers=1)
    mbar_group[0].init(1, predicate=isThread0)

    # 2. Execute Tensor Memory Accelerator (TMA) Load
    x_smem = get_dynamic_shared_memory((1, N), T.f32())
    y_smem = get_dynamic_shared_memory((1, N), T.f32(), offset=M * N * 2)
    x_tma.load(x_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    y_tma.load(y_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    mbar_group[0].arrive(txcount=size_x + size_y, predicate=isThread0)

    # 3. Wait for completion of TMA load with mbarrier
    mbar_group[0].try_wait()

    x_val = memref.load(x_smem, [0, tidx])
    y_val = memref.load(y_smem, [0, tidx])
    # SAXPY: y[i] += a * x[i];
    y_val += x_val * alpha
    memref.store(y_val, y_dev, [bidx, tidx])
```

# Ch2.py: 2D SAXPY with TMA

```
nvgpu.mbarrier.try_wait.parity %bar[%c0], %false,
%c10000000 : <memorySpace = #gpu.address_space<workgroup>>
```



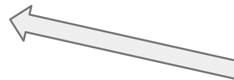
**Building IR with Py bindings**

```python
@NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1), smem=sz_x_y)
def saxpy_tma_kernel():
    bidx = gpu.block_id(gpu.Dimension.x)
    tidx = gpu.thread_id(gpu.Dimension.x)
    isThread0 = tidx == 0

    # 1. Create and initialize asynchronous transactional barrier (mbarrier)
    mbar_group = Mbarriers(number_of_barriers=1)
    mbar_group[0].init(1, predicate=isThread0)

    # 2. Execute Tensor Memory Accelerator (TMA) Load
    x_smem = get_dynamic_shared_memory((1, N), T.f32())
    y_smem = get_dynamic_shared_memory((1, N), T.f32(), offset=M * N * 2)
    x_tma.load(x_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    y_tma.load(y_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    mbar_group[0].arrive(txcount=size_x + size_y, predicate=isThread0)

    # 3. All threads in CTA wait for completion of TMA load with mbarrier
    mbar_group[0].try_wait()

    x_val = memref.load(x_smem, [0, tidx])
    y_val = memref.load(y_smem, [0, tidx])
    # SAXPY: y[i] += a * x[i];
    y_val += x_val * alpha
    memref.store(y_val, y_dev, [bidx, tidx])
```

# Ch2.py: 2D SAXPY with TMA

```
@NVDSL.mlir_gpu_launch(grid=(M, 1, 1), block=(N, 1, 1), smem=sz_x_y)
def saxpy_tma_kernel():
    bidx = gpu.block_id(gpu.Dimension.x)
    tidx = gpu.thread_id(gpu.Dimension.x)
    isThread0 = tidx == 0

    # 1. Create and initialize asynchronous transactional barrier (mbarrier)
    mbar_group = Mbarriers(number_of_barriers=1)
    mbar_group[0].init(1, predicate=isThread0)

    # 2. Execute Tensor Memory Accelerator (TMA) Load
    x_smem = get_dynamic_shared_memory((1, N), T.f32())
    y_smem = get_dynamic_shared_memory((1, N), T.f32(), offset=M * N * 2)
    x_tma.load(x_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    y_tma.load(y_smem, mbar_group[0], coords=[0, bidx], predicate=isThread0)
    mbar_group[0].arrive(txcount=size_x + size_y, predicate=isThread0)

    # 3. Wait for completion of TMA load with mbarrier
    mbar_group[0].try_wait()

    x_val = memref.load(x_smem, [0, tidx])
    y_val = memref.load(y_smem, [0, tidx])
    # SAXPY: y[i] += a * x[i];
    y_val += x_val * alpha
    memref.store(y_val, y_dev, [bidx, tidx])
```

Computation is similar to Ch1.py



# **Ch3.py:** GEMM 128x128x64 with Tensor Core

```
def gemm_128x128x64(a, b, d):
 a_smem, b_smem = tma_load()
 for i in range(128):
  for j in range(128):
    for k in range(64):
      d[i, j] += a_smem[i,k] * b_smem[k,j]
```

# **Ch3.py:** GEMM 128x128x64 with Tensor Core

```
def gemm_128x128x64(a, b, d):
 a_smem, b_smem = tma_load()
 for i in range(128):
  for j in range(128):
    for k in range(64):
      d[i, j] += a_smem[i,k] * b_smem[k,j]
```

Launch 1 Thread Block (CTA)

Offload 128x128x64 GEMM to Tensor Core

# Ch3.py: GEMM 128x128x64

## Tensor Core OPs and class WGMMAMatrix in NVDSL

### nvgpu.warpgroup.mma.init.accumulator

- Create and initialize registers (no need for a new op in nvvm)

### nvgpu.warpgroup.generate.descriptor

- Generates 64-bit descriptor that keeps: Start Address, leading dimension, stride, swizzle (no need for a new op in nvvm)

### nvgpu.warpgroup.mma

- Use Tensor Core using following new ops in nvvm
  - nvvm.wgmma.fence.aligned
  - nvvm.wgmma.mma_async
  - nvvm.wgmma.commit.group.sync.aligned
  - nvvm.wgmma.wait.group.sync.aligned

### nvgpu.warpgroup.mma.store

- Store fragmented registers to shared or global memory using following nvvm operations
  - nvvm.stmatrix
  - vector.store

```python
class WGMMAType(Enum):
 Accumulator = 1
 Descriptor = 2

class WGMMAMatrix:
 def __init__( self, matrix_type: WGMMAType, shape: list = None, ):...
 def update_smem(self, smem):
   self.smem = smem

 def update_accumulator(self, acc_op):
   self.acc_op = acc_op

 def __matmul__(self, rhs):
   lhs = nvgpu.warpgroup_generate_descriptor(...)
   rhs = nvgpu.warpgroup_generate_descriptor(...)
   return [lhs, rhs]

 def __iadd__(self, matmulResult):
   ...
   return nvgpu.warpgroup_mma(self.acc_op.type, lhs, rhs, self.acc_op, …)

 def store_accumulator(self):
     nvgpu.warpgroup_mma_store(...)
```

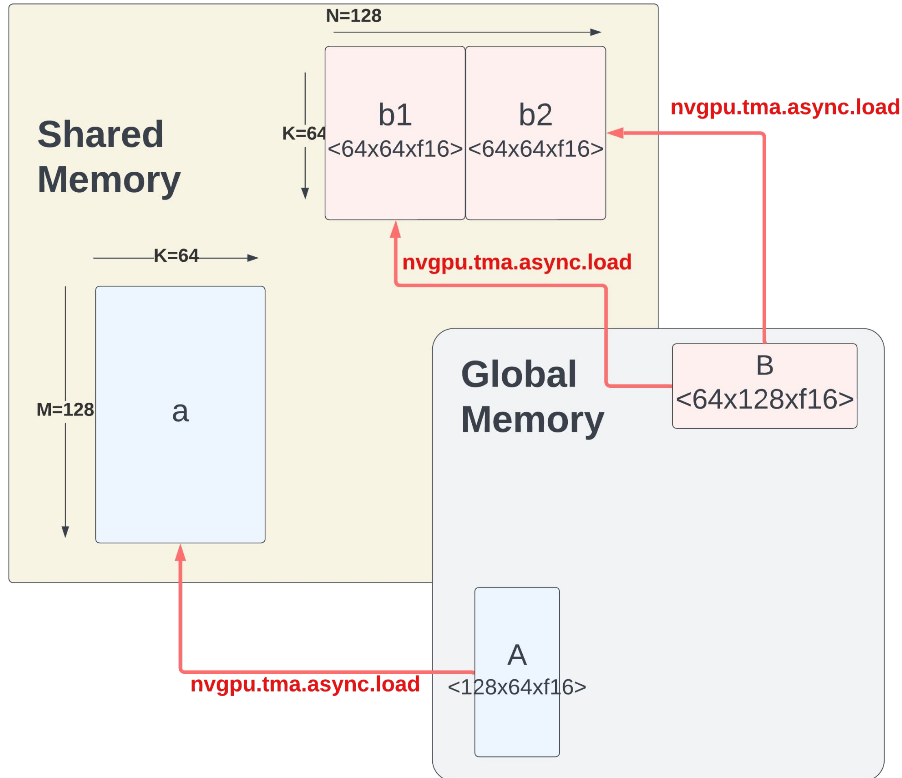
# Ch3.py: GEMM 128x128x64

Start Building Host IR

Building IR with Py bindings

```
@NVDSL.mlir_func
def gemm_128_128_64(a, b, d):

 t1 = gpu.wait(token_ty, [])
 a_dev, t2 = gpu.alloc(a.type, token_ty, [t1], [], [])
 b_dev, t3 = gpu.alloc(b.type, token_ty, [t2], [], [])
 d_dev, t4 = gpu.alloc(d.type, token_ty, [t3], [], [])
 t5 = gpu.memcpy(token_ty, [t4], a_dev, a)
 t6 = gpu.memcpy(token_ty, [t5], b_dev, b)
 t7 = gpu.wait(token_ty, [t6])

 sw = nvgpu.TensorMapSwizzleKind.SWIZZLE_128B
 a_tma = TMA([128, 64], a.type, swizzle=sw)
 b_tma = TMA([64,  64], b.type, swizzle=sw)
 a_tma.create_descriptor(a_dev)
 b_tma.create_descriptor(b_dev)
 sz = get_type_size(a.type) + get_type_size(b.type)

 @NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=sz)
 def gemm_tma_kernel():
   # Kernel Body
 gemm_tma_kernel()

 t8 = gpu.memcpy(token_ty, [t7], d, d_dev)
 gpu.wait(None, [t8])
```

# Ch3.py: GEMM 128x128x64




```
@NVDSL.mlir_func
def gemm_128_128_64(a, b, d):

  t1 = gpu.wait(token_ty, [])
  a_dev, t2 = gpu.alloc(a.type, token_ty, [t1], [], [])
  b_dev, t3 = gpu.alloc(b.type, token_ty, [t2], [], [])
  d_dev, t4 = gpu.alloc(d.type, token_ty, [t3], [], [])
  t5 = gpu.memcpy(token_ty, [t4], a_dev, a)
  t6 = gpu.memcpy(token_ty, [t5], b_dev, b)
  t7 = gpu.wait(token_ty, [t6])

  sw = nvgpu.TensorMapSwizzleKind.SWIZZLE_128B
  a_tma = TMA([128, 64], a.type, swizzle=sw)
  b_tma = TMA([64,  64], b.type, swizzle=sw)
  a_tma.create_descriptor(a_dev)
  b_tma.create_descriptor(b_dev)
  sz = get_type_size(a.type) + get_type_size(b.type)

  @NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=sz)
  def gemm_tma_kernel():
    # Kernel Body
  gemm_tma_kernel()

  t8 = gpu.memcpy(token_ty, [t7], d, d_dev)
  gpu.wait(None, [t8])
```

# Ch3.py: GEMM 128x128x64

Start Building GPU IR

Building IR with Py bindings

```
@NVDSL.mlir_gpu_launch(grid=(1,1,1),
                       block=(128,1,1),smem=sz)
def gemm_tma_kernel():
  tidx = gpu.thread_id(gpu.Dimension.x)
  isThread0 = tidx == 0
  mbar_group = Mbarriers(number_of_barriers=1)
  mbar_group[0].init(1, predicate=isThread0)

  # 1. TMA Load for two input matrices
  tma_load(mbar_group, a_tma, b_tma, isThread0)

  # 2. All threads wait TMA load completion
  mbar_group[0].try_wait()
  a_smem = get_dynamic_shared_memory((128, 64), T.f16())
  b_smem = get_dynamic_shared_memory((64, 128), T.f16(), offset=off_b)
  # 3. Performs Tensor Core GEMM 128x128x64 by warpgroup
  A = WGMMAMatrix(WGMMAType.Descriptor, [128, 64], desc=a_tma, a_smem)
  B = WGMMAMatrix(WGMMAType.Descriptor, [64, 128], desc=b_tma, b_smem)
  C = WGMMAMatrix(WGMMAType.Accumulator, shape=[128, 128], ty=T.f32())
  # Matrix Multiply
  C += A @ B

  # 4. Stores fragmented registers to global memory by warpgroup
  C.store_accumulator(d_dev)
```

# Ch3.py: GEMM 128x128x64

```
gpu.launch blocks(1,1,1) threads(128,1,1) {
  %tidx = gpu.thread_id  x
  %isThread0 = arith.cmpi eq, %thread_id_x, %c0 : index

  %bar = nvgpu.mbarrier.create -> <memorySpace = #gpu.address_space<workgroup>>
  nvgpu.mbarrier.init %bar[%c0], %c1, predicate = %isThread0
                                  : <memorySpace = #gpu.address_space<workgroup>>
```

Building IR with Py bindings

```python
@NVDSL.mlir_gpu_launch(grid=(1,1,1),
                       block=(128,1,1),smem=sz)
def gemm_tma_kernel():
  tidx = gpu.thread_id(gpu.Dimension.x)
  isThread0 = tidx == 0
  mbar_group = Mbarriers(number_of_barriers=1)
  mbar_group[0].init(1, predicate=isThread0)

  # 1. TMA Load for two input matrices
  tma_load(mbar_group, a_tma, b_tma, 0, 0, isThread0)

  # 2. All threads wait TMA load completion
  mbar_group[0].try_wait()
  a_smem = get_dynamic_shared_memory((128, 64), T.f16())
  b_smem = get_dynamic_shared_memory((64, 128), T.f16(), offset=off_b)
  # 3. Performs Tensor Core GEMM 128x128x64 by warpgroup
  A = WGMMAMatrix(WGMMAType.Descriptor, [128, 64], desc=a_tma, a_smem)
  B = WGMMAMatrix(WGMMAType.Descriptor, [64, 128], desc=b_tma, b_smem)
  C = WGMMAMatrix(WGMMAType.Accumulator, shape=[128, 128], ty=T.f32())
  # Matrix Multiply
  C += A @ B

  # 4. Stores fragmented registers to global memory by warpgroup
  C.store_accumulator(d_dev)
```

# Ch3.py: GEMM 128x128x64

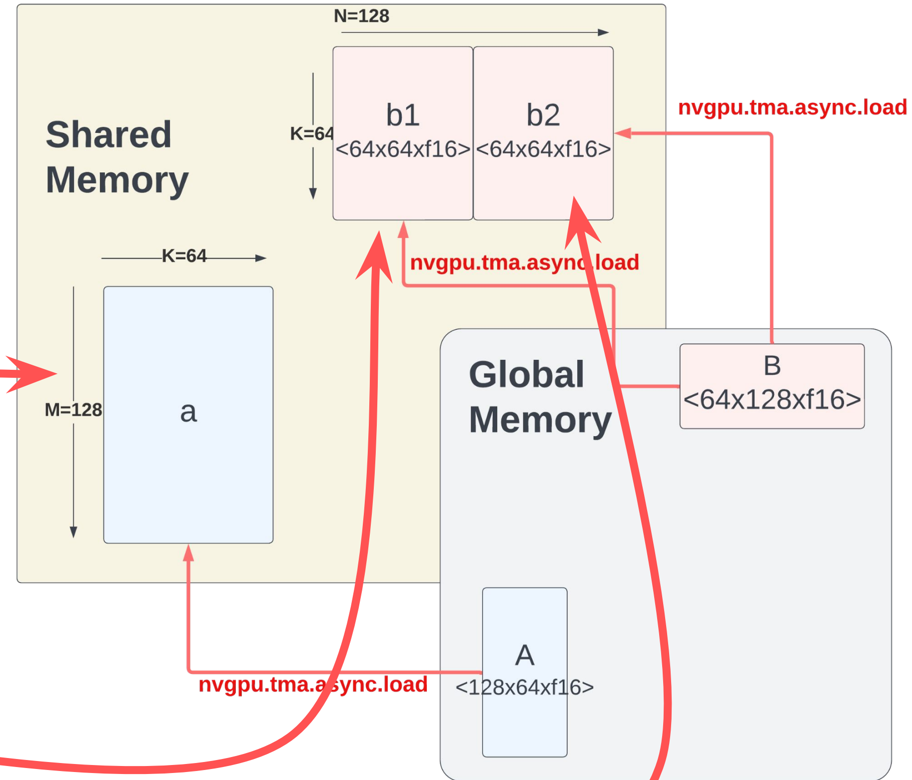
```python
def tma_load(mbar_group:Mbarriers, a_tma:TMA, b_tma:TMA, slot, stage, pred):
    size_tma_a = get_type_size(a_tma.tma_memref)
    size_tma_b = get_type_size(b_tma.tma_memref)
    ta_count = size_tma_a + (size_tma_b * 2)

    off_b = size_tma_a
    off_b2 = off_b + size_tma_b
    a_elem_ty = a_tma.tma_memref.element_type
    b_elem_ty = b_tma.tma_memref.element_type
    a = get_dynamic_shared_memory(a_tma.tma_memref.shape, a_elem_ty)
    b1 = get_dynamic_shared_memory(b_tma.tma_memref.shape, b_elem_ty, off_b)
    b2 = get_dynamic_shared_memory(b_tma.tma_memref.shape, b_elem_ty, off_b2)

    mbar_group[slot].arrive(ta_count, predicate=pred)
    dimN, dimM = partition_shape()
    a_tma.load(a,  mbar_group[slot], coords=[dimK     , dimM], predicate=pred)
    b_tma.load(b1, mbar_group[slot], coords=[dimN     , dimK], predicate=pred)
    b_tma.load(b2, mbar_group[slot], coords=[dimN + 64, dimK], predicate=pred)
```

**Building IR with Py bindings**

```python
@NVDSL.mlir_gpu_launch(grid=(1,1,1),
                       block=(128,1,1),smem=sz)
def gemm_tma_kernel():
    tidx = gpu.thread_id(gpu.Dimension.x)
    isThread0 = tidx == 0
    mbar_group = Mbarriers(number_of_barriers=1)
    mbar_group[0].init(1, predicate=isThread0)

    # 1. TMA Load for two input matrices
    tma_load(mbar_group, a_tma, b_tma, 0, 0, isThread0)

    # 2. All threads wait TMA load completion
    mbar_group[0].try_wait()
    a_smem = get_dynamic_shared_memory((128, 64), T.f16())
    b_smem = get_dynamic_shared_memory((64, 128), T.f16(), offset=off_b)
    # 3. Performs Tensor Core GEMM 128x128x64 by warpgroup
    A = WGMMAMatrix(WGMMAType.Descriptor, [128, 64], desc=a_tma, a_smem)
    B = WGMMAMatrix(WGMMAType.Descriptor, [64, 128], desc=b_tma, b_smem)
    C = WGMMAMatrix(WGMMAType.Accumulator, shape=[128, 128], ty=T.f32())
    # Matrix Multiply
    C += A @ B

    # 4. Stores fragmented registers to global memory by warpgroup
    C.store_accumulator(d_dev)
```

# **Ch3.py:** GEMM 128x128x64

```python
def tma_load(mbar_group:Mbarriers, a_tma:TMA, b_tma:TMA, slot, stage, pred):
    size_tma_a = get_type_size(a_tma.tma_memref)
    size_tma_b = get_type_size(b_tma.tma_memref)
    ta_count = size_tma_a + (size_tma_b * 2)

    off_b = size_tma_a
    off_b2 = off_b + size_tma_b
    a_elem_ty = a_tma.tma_memref.element_type
    b_elem_ty = b_tma.tma_memref.element_type
    a = get_dynamic_shared_memory(a_tma.tma_memref.shape, a_elem_ty)
    b1 = get_dynamic_shared_memory(b_tma.tma_memref.shape, b_elem_ty, off_b)
    b2 = get_dynamic_shared_memory(b_tma.tma_memref.shape, b_elem_ty, off_b2)

    mbar_group[slot].arrive(ta_count, predicate=pred)
    dimN, dimM = partition_shape()
    a_tma.load(a,  mbar_group[slot], coords=[dimK     , dimM], predicate=pred)
    b_tma.load(b1, mbar_group[slot], coords=[dimN     , dimK], predicate=pred)
    b_tma.load(b2, mbar_group[slot], coords=[dimN + 64, dimK], predicate=pred)
```



# Ch3.py: GEMM 128x128x64

```
%A = nvgpu.warpgroup.generate.descriptor %view, %3
       : memref<128x64xf16, ...>,  …

%B = nvgpu.warpgroup.generate.descriptor %view_5, %4
       : memref<64x128xf16, ...>,  …

%C = nvgpu.warpgroup.mma.init.accumulator
       -> <fragmented = vector<128x128xf32>>

%D = nvgpu.warpgroup.mma %10, %11, %9 {transposeB}
       : <tensor = memref<128x64xf16, ...>>,
         <tensor = memref<64x128xf16, ...>>,
         <fragmented = vector<128x128xf32>>
       -> <fragmented = vector<128x128xf32>>
```

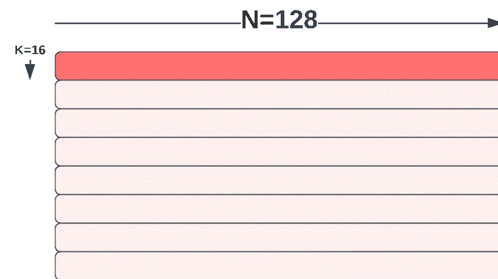
Building IR with Py bindings

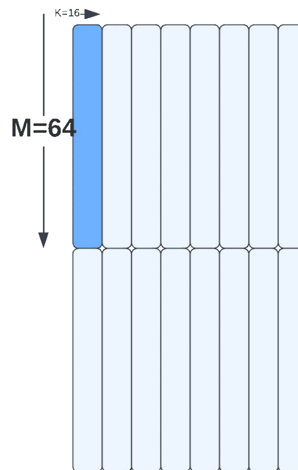
```python
@NVDSL.mlir_gpu_launch(grid=(1,1,1),
                       block=(128,1,1),smem=sz)
def gemm_tma_kernel():
  tidx = gpu.thread_id(gpu.Dimension.x)
  isThread0 = tidx == 0
  mbar_group = Mbarriers(number_of_barriers=1)
  mbar_group[0].init(1, predicate=isThread0)

  # 1. TMA Load for two input matrices
  tma_load(mbar_group, a_tma, b_tma, isThread0)

  # 2. All threads wait TMA load completion
  mbar_group[0].try_wait()
  a_smem = get_dynamic_shared_memory((128, 64), T.f16())
  b_smem = get_dynamic_shared_memory((64, 128), T.f16(), offset=off_b)
  # 3. Initialize 2 Input Matrices and Accumulator
  A = WGMMAMatrix(WGMMAType.Descriptor, [128 ,64], desc=a_tma, a_smem)
  B = WGMMAMatrix(WGMMAType.Descriptor, [64, 128], desc=b_tma, b_smem)
  C = WGMMAMatrix(WGMMAType.Accumulator, shape=[128, 128], ty=T.f32())
  # Matrix Multiply
  C += A @ B

  # 4. Stores fragmented registers to global memory by warpgroup
  C.store_accumulator(d_dev)
```

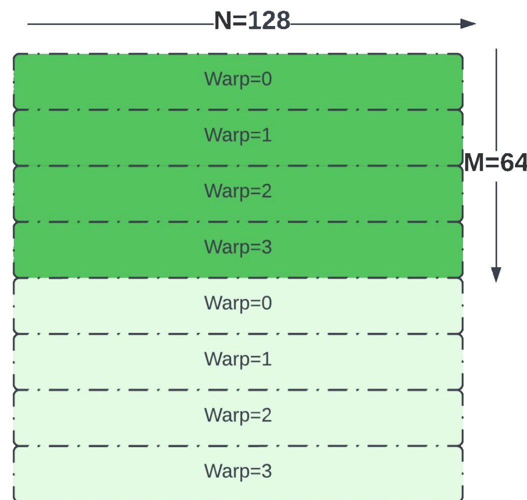
# Go Deeper `nvgpu.warpgroup.mma → nvvm/PTX`

128x128x64 → 8 times 64x128x16 (supported tensor core shape)

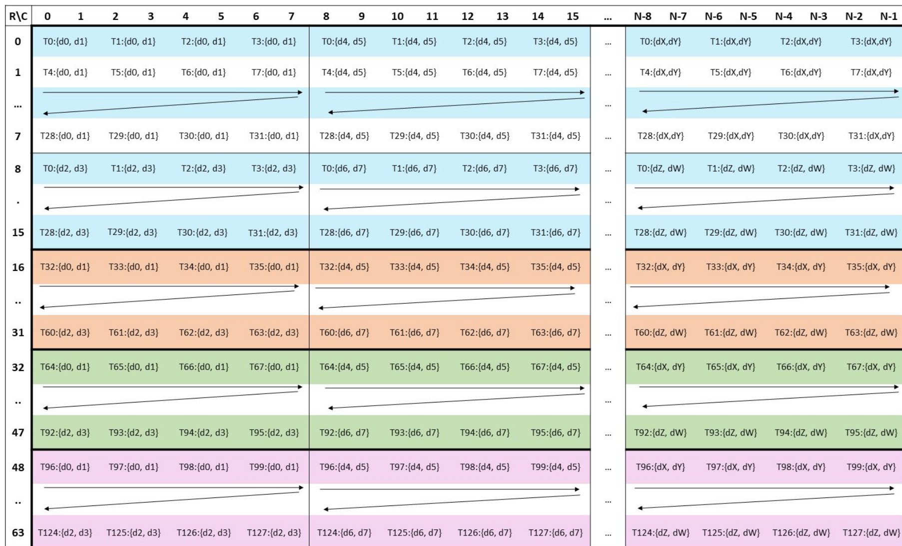
```
// Initialize input matrix: 2x64xf32 Registers
%r = 0 : !llvm.struct<(...)>

// 8 x wgmma.mma_async.m64n128k16 PTX instruction
nvvm.wgmma.fence.aligned

%w1 = nvvm.wgmma.mma_async %dA,      %dB,      %r[0],   <m=64, n=128, k=16>
%w2 = nvvm.wgmma.mma_async %dA+2,    %dB+128,  %w1,     <m=64, n=128, k=16>
%w3 = nvvm.wgmma.mma_async %dA+4,    %dB+256,  %w2,     <m=64, n=128, k=16>
%w4 = nvvm.wgmma.mma_async %dA+6,    %dB+384,  %w3,     <m=64, n=128, k=16>
%w5 = nvvm.wgmma.mma_async %dA+512,  %dB,    , %r[1],   <m=64, n=128, k=16>
%w6 = nvvm.wgmma.mma_async %dA+514,  %dB+128,  %w5,     <m=64, n=128, k=16>
%w7 = nvvm.wgmma.mma_async %dA+516,  %dB+256,  %w6,     <m=64, n=128, k=16>
%w8 = nvvm.wgmma.mma_async %dA+518,  %dB+384,  %w7,     <m=64, n=128, k=16>

nvvm.wgmma.commit.group.sync.aligned
nvvm.wgmma.wait.group.sync.aligned 1
```



# Ch3.py: GEMM 128x128x64



| R\C | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | ... | N-8 | N-7 | N-6 | N-5 | N-4 | N-3 | N-2 | N-1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | T0:{d0, d1} | | T1:{d0, d1} | | T2:{d0, d1} | | T3:{d0, d1} | | T0:{d4, d5} | | T1:{d4, d5} | | T2:{d4, d5} | | T3:{d4, d5} | | ... | T0:{dX,dY} | | T1:{dX,dY} | | T2:{dX,dY} | | T3:{dX,dY} | |
| 1 | T4:{d0, d1} | | T5:{d0, d1} | | T6:{d0, d1} | | T7:{d0, d1} | | T4:{d4, d5} | | T5:{d4, d5} | | T6:{d4, d5} | | T7:{d4, d5} | | ... | T4:{dX,dY} | | T5:{dX,dY} | | T6:{dX,dY} | | T7:{dX,dY} | |
| ... | | | | | | | | | | | | | | | | | ... | | | | | | | | |
| 7 | T28:{d0, d1} | | T29:{d0, d1} | | T30:{d0, d1} | | T31:{d0, d1} | | T28:{d4, d5} | | T29:{d4, d5} | | T30:{d4, d5} | | T31:{d4, d5} | | ... | T28:{dX,dY} | | T29:{dX,dY} | | T30:{dX,dY} | | T31:{dX,dY} | |
| 8 | T0:{d2, d3} | | T1:{d2, d3} | | T2:{d2, d3} | | T3:{d2, d3} | | T0:{d6, d7} | | T1:{d6, d7} | | T2:{d6, d7} | | T3:{d6, d7} | | ... | T0:{dZ, dW} | | T1:{dZ, dW} | | T2:{dZ, dW} | | T3:{dZ, dW} | |
| . | | | | | | | | | | | | | | | | | ... | | | | | | | | |
| 15 | T28:{d2, d3} | | T29:{d2, d3} | | T30:{d2, d3} | | T31:{d2, d3} | | T28:{d6, d7} | | T29:{d6, d7} | | T30:{d6, d7} | | T31:{d6, d7} | | ... | T28:{dZ, dW} | | T29:{dZ, dW} | | T30:{dZ, dW} | | T31:{dZ, dW} | |
| 16 | T32:{d0, d1} | | T33:{d0, d1} | | T34:{d0, d1} | | T35:{d0, d1} | | T32:{d4, d5} | | T33:{d4, d5} | | T34:{d4, d5} | | T35:{d4, d5} | | ... | T32:{dX, dY} | | T33:{dX, dY} | | T34:{dX, dY} | | T35:{dX, dY} | |
| .. | | | | | | | | | | | | | | | | | ... | | | | | | | | |
| 31 | T60:{d2, d3} | | T61:{d2, d3} | | T62:{d2, d3} | | T63:{d2, d3} | | T60:{d6, d7} | | T61:{d6, d7} | | T62:{d6, d7} | | T63:{d6, d7} | | ... | T60:{dZ, dW} | | T61:{dZ, dW} | | T62:{dZ, dW} | | T63:{dZ, dW} | |
| 32 | T64:{d0, d1} | | T65:{d0, d1} | | T66:{d0, d1} | | T67:{d0, d1} | | T64:{d4, d5} | | T65:{d4, d5} | | T66:{d4, d5} | | T67:{d4, d5} | | ... | T64:{dX, dY} | | T65:{dX, dY} | | T66:{dX, dY} | | T67:{dX, dY} | |
| .. | | | | | | | | | | | | | | | | | ... | | | | | | | | |
| 47 | T92:{d2, d3} | | T93:{d2, d3} | | T94:{d2, d3} | | T95:{d2, d3} | | T92:{d6, d7} | | T93:{d6, d7} | | T94:{d6, d7} | | T95:{d6, d7} | | ... | T92:{dZ, dW} | | T93:{dZ, dW} | | T94:{dZ, dW} | | T95:{dZ, dW} | |
| 48 | T96:{d0, d1} | | T97:{d0, d1} | | T98:{d0, d1} | | T99:{d0, d1} | | T96:{d4, d5} | | T97:{d4, d5} | | T98:{d4, d5} | | T99:{d4, d5} | | ... | T96:{dX, dY} | | T97:{dX, dY} | | T98:{dX, dY} | | T99:{dX, dY} | |
| .. | | | | | | | | | | | | | | | | | ... | | | | | | | | |
| 63 | T124:{d2, d3} | | T125:{d2, d3} | | T126:{d2, d3} | | T127:{d2, d3} | | T124:{d6, d7} | | T125:{d6, d7} | | T126:{d6, d7} | | T127:{d6, d7} | | ... | T124:{dZ, dW} | | T125:{dZ, dW} | | T126:{dZ, dW} | | T127:{dZ, dW} | |

(tid % 128) : fragments

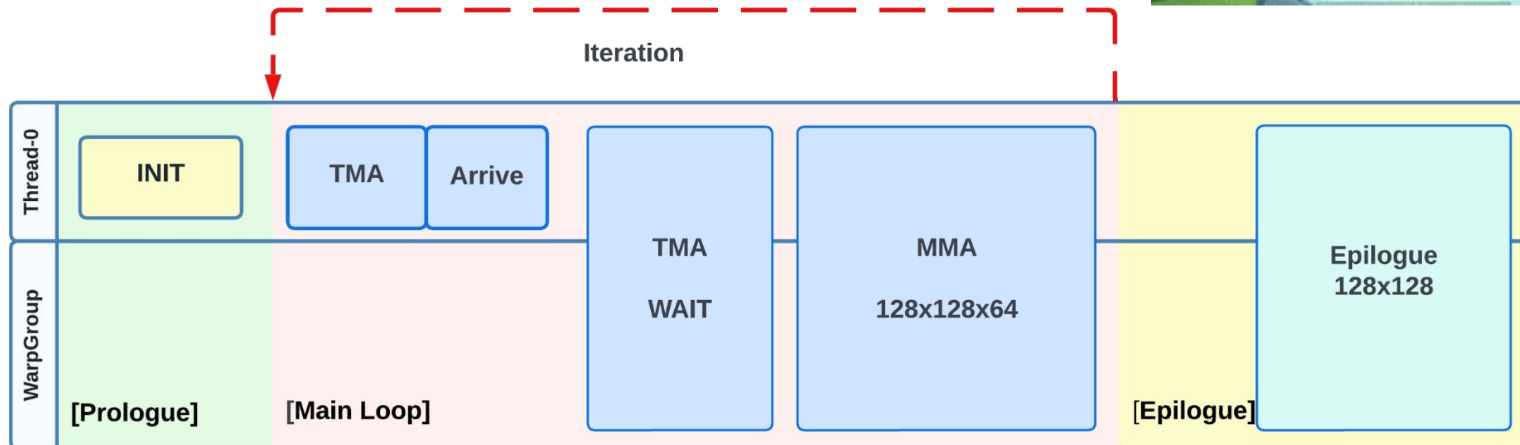
```
nvgpu.warpgroup.mma.store %C, %memref :
          <fragmented = vector<64x64xf32>>
          to memref<64x64xf32>
```

Building IR with Py bindings

```python
@NVDSL.mlir_gpu_launch(grid=(1,1,1),
                       block=(128,1,1),smem=sz)
def gemm_tma_kernel():
  tidx = gpu.thread_id(gpu.Dimension.x)
  isThread0 = tidx == 0
  mbar_group = Mbarriers(number_of_barriers=1)
  mbar_group[0].init(1, predicate=isThread0)

  # 1. TMA Load for two input matrices
  tma_load(mbar_group, a_tma, b_tma, isThread0)

  # 2. All threads wait TMA load completion
  mbar_group[0].try_wait()
  a_smem = get_dynamic_shared_memory((M, K), T.f16())
  b_smem = get_dynamic_shared_memory((K, N), T.f16(), offset=off_b)
  # 3. Initialize 2 Input Matrices and Accumulator
  A = WGMMAMatrix(WGMMAType.Descriptor, [M,K], desc=a_tma, a_smem)
  B = WGMMAMatrix(WGMMAType.Descriptor, [K,N], desc=b_tma, b_smem)
  C = WGMMAMatrix(WGMMAType.Accumulator, shape=[M,N], ty=T.f32())
  # Matrix Multiply
  C += A @ B

  # 4. Stores fragmented registers to global memory by warpgroup
  C.store_accumulator(d_dev)
```

# **Ch3.py:** GEMM 128x128x64

What about the performance?

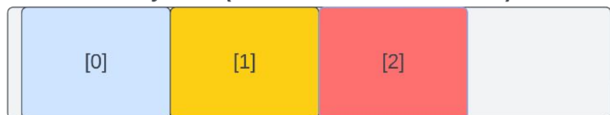


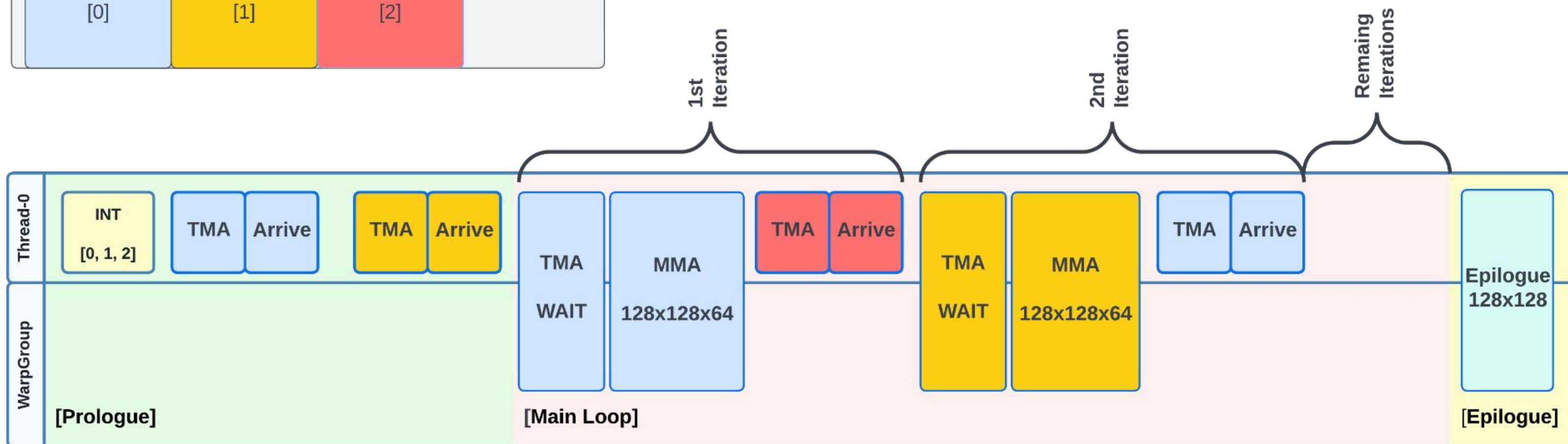
# Ch4.py: Multistage GEMM

Shape = MxNxK, Tile = 128x128x64

Overlap Tensor Core and Data Load (TMA)

Shared Memory Slots (Size of slot = Tile-A + Tile-B )

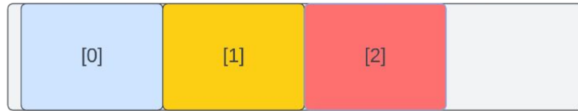


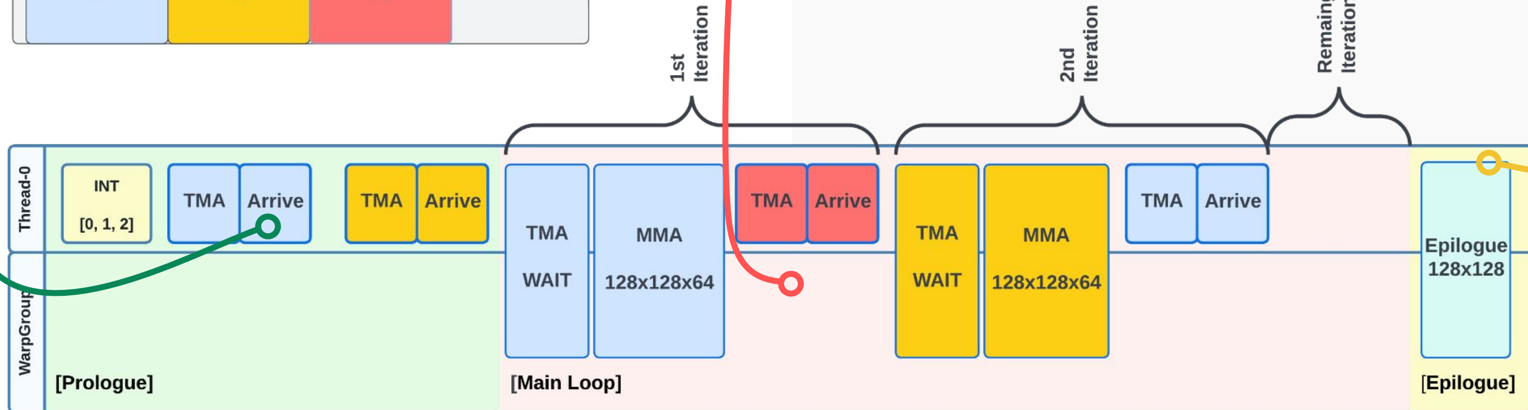



# Ch4.py: Multistage GEMM

Shape = MxNxK, Tile = 128x128x64

```
@NVDSL.mlir_gpu_launch(grid=grid,block=block,smem=...)
def gemm_multistage_kernel():
 mbar_group = init(x_tma, y_tma)

 prologue(mbar_group, x_tma, y_tma)

 D = mainloop(mbar_group, x_tma, y_tma)

 epilogue(D, z_dev)
```



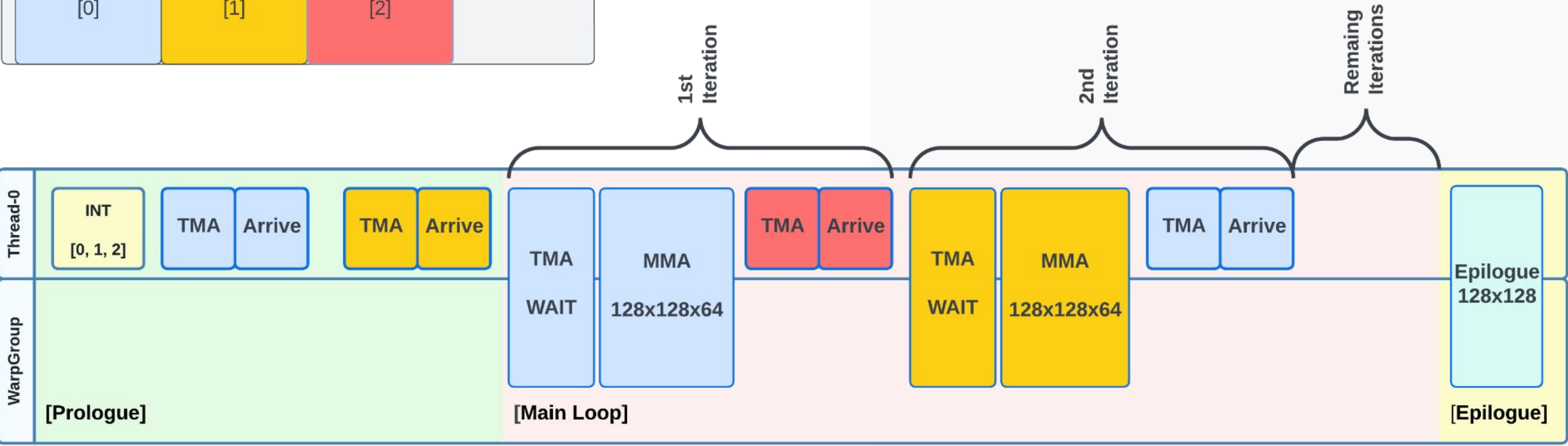



# Ch4.py: Multistage GEMM

## Prologue

```
def prologue(mbar_group: Mbarriers,
             a_tma: TMA, b_tma: TMA):

  for iv in scf.for_(0, NUM_STAGES-1, 1):
    tma_load(mbar_group, a_tma, b_tma, iv, iv)
    scf.yield_([])
```

Shared Memory Slots (Size of slot = Tile-A + Tile-B )



# **Ch4.py:** Multistage GEMM

## Mainloop

```
def mainloop(x,y,z):
for ti in range(M//128):  # -> blockIdx.x
 for tj in range(N//128): # -> blockIdx.y
  D = 0
  for tk in range(K//64):
   for i in range(128):
    for j in range(128):
     for k in range(64):
      D += # mma

  z(ti:ti:128, tj:tj:128) = D
```

# Ch4.py: Multistage GEMM

## Mainloop

```
def mainloop(x,y,z):
for ti in range(M//128):  # -> blockIdx.x
 for tj in range(N//128): # -> blockIdx.y
  D = 0
  for tk in range(K//64):
   for i in range(128):
    for j in range(128):
     for k in range(64):
      D += # mma

  z(ti:ti:128, tj:tj:128) = D
```

Need a Loop

Tensor Core
128x128x64

# Ch4.py: Multistage GEMM

## Mainloop

```
def mainloop(x,y,z):
for ti in range(M//128):  # -> blockIdx.x
 for tj in range(N//128): # -> blockIdx.y
  D = 0
  for tk in range(K//64):
   for i in range(128):
    for j in range(128):
     for k in range(64):
      D += # mma

  z(ti:ti:128, tj:tj:128) = D
```

```
def mainloop(mbar_group: Mbarriers, x_tma: TMA, y_tma: TMA):
 ns = NUM_STAGES if NUM_STAGES == 1 else NUM_STAGES - 1
 tidx = gpu.thread_id(gpu.Dimension.x)
 begin_y = NUM_STAGES * get_type_size(x_tma.tma_memref)
 size_x = TILE_M * TILE_K * get_type_size(T.f16())

 pp = const(False, ty=T.bool())
 A = WGMMAMatrix(WGMMAType.Descriptor, [TILE_M, TILE_K], desc=a_tma)
 B = WGMMAMatrix(WGMMAType.Descriptor, [TILE_K, TILE_N], desc=b_tma)
 D = WGMMAMatrix(WGMMAType.Accumulator, [TILE_M, TILE_N], ty=T.f32())

 # Main Loop
 for_op = scf.ForOp(const(0), const(K // TILE_K), const(1),
                    [D.acc_op, pp])
 with ir.InsertionPoint(for_op.body):
   # Main Loop BODY
   scf.yield_([D.acc_op, newPP])

 nvvm.WgmmaWaitGroupSyncOp(0)

 return D
```

# Ch4.py: Multistage GEMM

## Mainloop

**Start Building Mainloop IR**

**Building IR with Py bindings**

```
with ir.InsertionPoint(for_op.body):
    phase = for_op.inner_iter_args[1]
    iv = for_op.induction_variable
    stage = iv % NUM_STAGES
    # Wait for the current stage
    mbar_group[stage].try_wait(phase=phase)
    offX = stage * size_a
    offY = offset_a + begin_b
    a_smem = get_dynamic_shared_memory([TILE_M, TILE_K], T.f16(), offX)
    b_smem = get_dynamic_shared_memory([TILE_K, TILE_N], T.f16(), offY)
    # Iterate input matrices, update accumulator
    A.update_smem(a_smem)
    B.update_smem(b_smem)
    D.update_accumulator(for_op.inner_iter_args[0])
    # Matrix Multiply
    D += A @ B
    # Load next stage
    pred = ((iv + ns) < const(K // TILE_K)) & (tidx == 0)
    nextSlot = (iv + ns) % NUM_STAGES
    tma_load(mbar_group, a_tma, b_tma, nextSlot, (iv + ns), pred)
    # Switch phase parity for the mbarrier
    newPhase = arith.select( stage == (NUM_STAGES - 1),
                (phase ^ const(True, ty=T.bool())), phase, )
    scf.yield_([D.acc_op, newPhase])
```

# **Ch4.py:** Multistage GEMM

## Mainloop

```
%13 = arith.remui %arg15, %c3 : index
nvgpu.mbarrier.try_wait.parity %6[%13], %arg17, %ticks
```

**Building IR with Py bindings**

```
with ir.InsertionPoint(for_op.body):
    phase = for_op.inner_iter_args[1]
    iv = for_op.induction_variable
    stage = iv % NUM_STAGES
    # Wait for the current stage
    mbar_group[stage].try_wait(phase=phase)
    offX = stage * size_a
    offY = offset_a + begin_b
    a_smem = get_dynamic_shared_memory([TILE_M, TILE_K], T.f16(), offX)
    b_smem = get_dynamic_shared_memory([TILE_K, TILE_N], T.f16(), offY)
    # Iterate input matrices, update accumulator
    A.update_smem(a_smem)
    B.update_smem(b_smem)
    D.update_accumulator(for_op.inner_iter_args[0])
    # Matrix Multiply
    D += A @ B
    # Load next stage
    pred = ((iv + ns) < const(K // TILE_K)) & (tidx == 0)
    nextSlot = (iv + ns) % NUM_STAGES
    tma_load(mbar_group, a_tma, b_tma, nextSlot, (iv + ns), pred)
    # Switch phase parity for the mbarrier
    newPhase = arith.select( stage == (NUM_STAGES - 1),
                 (phase ^ const(True, ty=T.bool())), phase, )
    scf.yield_([D.acc_op, newPhase])
```

# Ch4.py: Multistage GEMM

## Mainloop

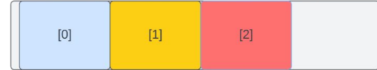
```
%A = nvgpu.warpgroup.generate.descriptor %x, %3
       : memref<128x64xf16,>
%B = nvgpu.warpgroup.generate.descriptor %y, %4
       : memref<64x128xf16,>

%D = nvgpu.warpgroup.mma %A, %B, %C {transposeB}
   : <tensor = memref<128x64xf16  ,...>>,
     <tensor = memref<64x128xf16  ,...>>,
     <fragmented = vector<128x128xf32>>
   -> <fragmented = vector<128x128xf32>>
```

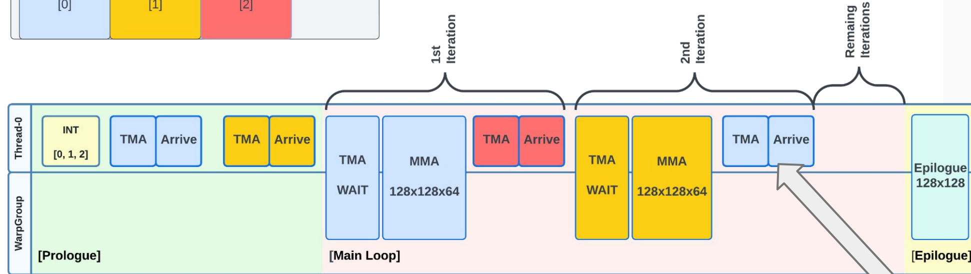
**Building IR with Py bindings**

```
with ir.InsertionPoint(for_op.body):
    phase = for_op.inner_iter_args[1]
    iv = for_op.induction_variable
    stage = iv % NUM_STAGES
    # Wait for the current stage
    mbar_group[stage].try_wait(phase=phase)
    offX = stage * size_a
    offY = offset_a + begin_b
    a_smem = get_dynamic_shared_memory([TILE_M, TILE_K], T.f16(), offX)
    b_smem = get_dynamic_shared_memory([TILE_K, TILE_N], T.f16(), offY)
    # Iterate input matrices, update accumulator
    A.update_smem(a_smem)
    B.update_smem(b_smem)
    D.update_accumulator(for_op.inner_iter_args[0])
    # Matrix Multiply
    D += A @ B
    # Load next stage
    pred = ((iv + ns) < const(K // TILE_K)) & (tidx == 0)
    nextSlot = (iv + ns) % NUM_STAGES
    tma_load(mbar_group, a_tma, b_tma, nextSlot, (iv + ns), pred)
    # Switch phase parity for the mbarrier
    newPhase = arith.select( stage == (NUM_STAGES - 1),
                   (phase ^ const(True, ty=T.bool())), phase, )
    scf.yield_([D.acc_op, newPhase])
```

# Ch4.py: Multistage GEMM

## Mainloop




Building IR with Py bindings

```
with ir.InsertionPoint(for_op.body):
    phase = for_op.inner_iter_args[1]
    iv = for_op.induction_variable
    stage = iv % NUM_STAGES
    # Wait for the current stage
    mbar_group[stage].try_wait(phase=phase)
    offX = stage * size_a
    offY = offset_a + begin_b
    a_smem = get_dynamic_shared_memory([TILE_M, TILE_K], T.f16(), offX)
    b_smem = get_dynamic_shared_memory([TILE_K, TILE_N], T.f16(), offY)
    # Iterate input matrices, update accumulator
    A.update_smem(a_smem)
    B.update_smem(b_smem)
    D.update_accumulator(for_op.inner_iter_args[0])
    # Matrix Multiply
    D += A @ B
    # Load next stage
    pred = ((iv + ns) < const(K // TILE_K)) & (tidx == 0)
    nextSlot = (iv + ns) % NUM_STAGES
    tma_load(mbar_group, a_tma, b_tma, nextSlot, (iv + ns), pred)
    # Switch phase parity for the mbarrier
    newPhase = arith.select( stage == (NUM_STAGES - 1),
                (phase ^ const(True, ty=T.bool())), phase, )
    scf.yield_([D.acc_op, newPhase])
```

# **Ch4.py:** Multistage GEMM

## Epilogue

Steps:

1. Stores registers -> shared memory

2. Store shared memory tile -> global memory

**Building IR with Py bindings**

```
def epilogue(D: WGMMAMatrix, d_dev):
    tidx = gpu.thread_id(gpu.Dimension.x)
    dimX, dimY = partition_shape()
    d_smem = get_dynamic_shared_memory([TILE_M, TILE_N], T.f32())
    d_gmem = memref.subview(d_dev,[dimX, dimY],[TILE_M,TILE_N],[1, 1])

    # Store (registers -> shared memory)
    D.store_accumulator(d_smem)
    gpu.barrier()

    # Store (shared memory --> global memory)
    for i in scf.for_(0, TILE_M, 1):
        val = memref.load(d_smem, [i, tidx])
        memref.store(val, d_gmem, [i, tidx])
        scf.yield_([])
```

Google Research

# Single Stage vs Multi Stage

# Benchmark

## Effect of Multistage vs Single Stage

Shape: (K varies)

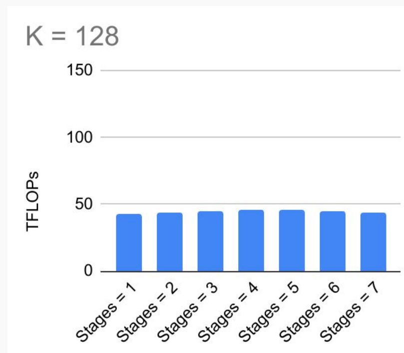
- **7296 x 256 x K**

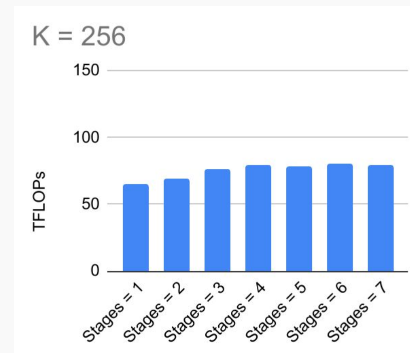
Operation:

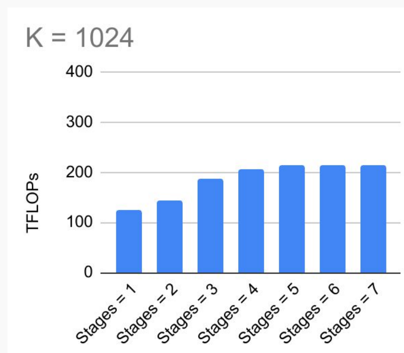
- F32 += F16 * F16

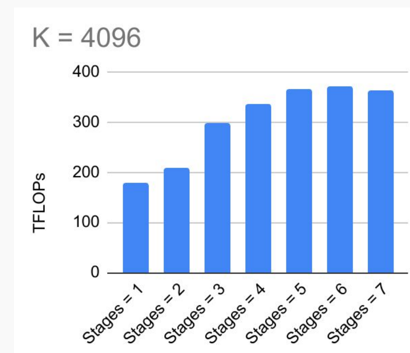
Tile Size:

- 128 x 128 x 64



# Ch4.py vs NVIDIA cuBLAS

# cuBLAS vs MLIR

**Operation:**

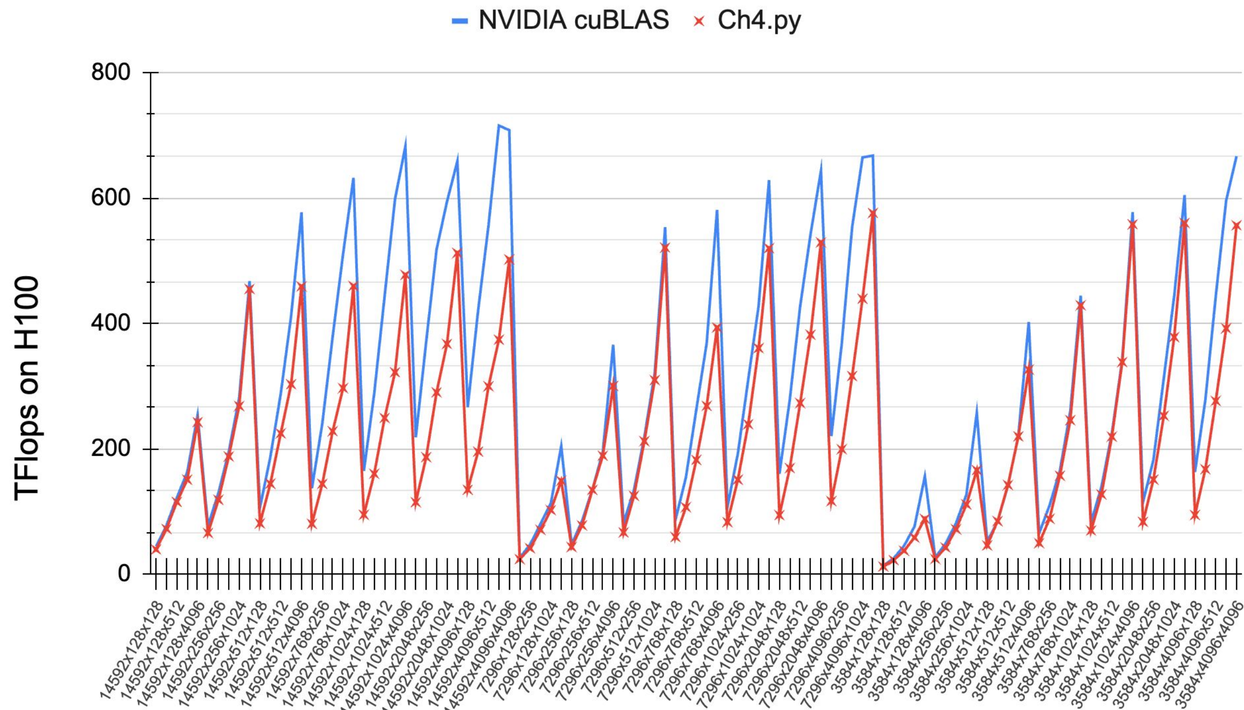
- F32 += F16 * F16

**Tile Size:**

- 128 x 128 x 64

**Ch4.py**

- Multistage Kernel

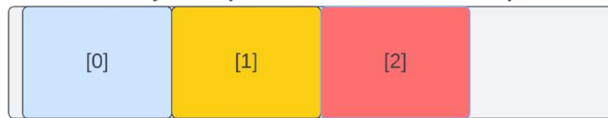


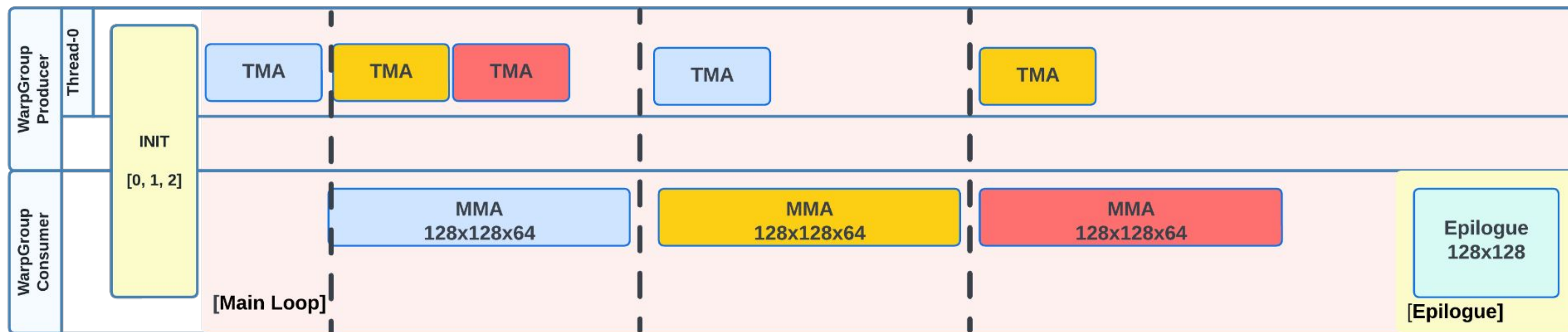
# Ch5.py: Warp Specialized GEMM

1 Producer + 1 Consumer Warpgroups

## Thread Block has 2 Warpgroups (256 threads):

1. Producer Warpgroup → Performs TMA
2. Consumer Warpgroup → Performs Tensor Core



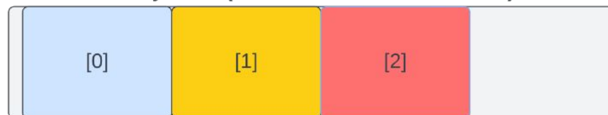
# Ch5.py: Warp Specialized GEMM

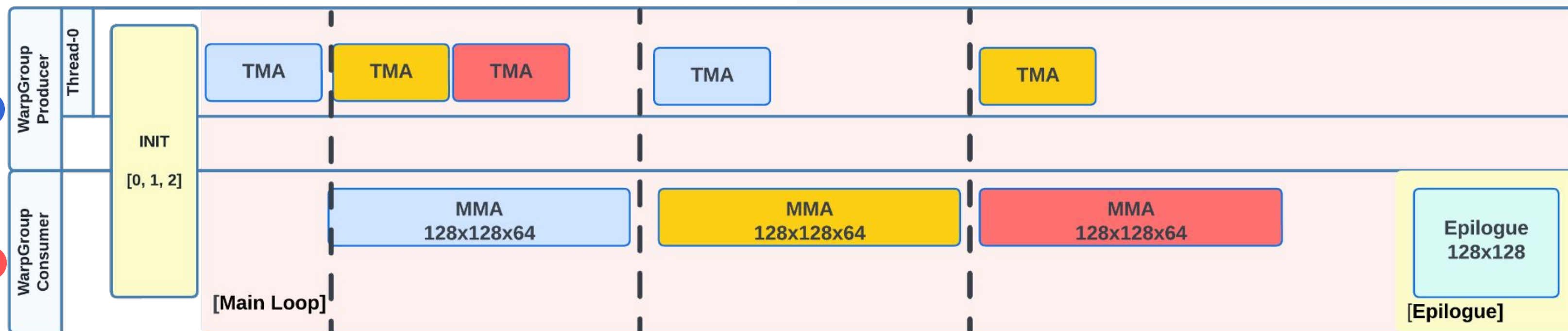
## 1 Producer + 1 Consumer Warpgroups

Building IR with Py bindings

```
def gemm_warp_specialized_kernel():
    wg_producer = Warpgroup(primaryThread = 128, regSize = 40)
    wg_consumer = Warpgroup(primaryThread = 0,   regSize = 232)
    mbar_group_mma, mbar_group_tma = bootstrap(a_tma, b_tma)
    # Producer performs TMA
    with wg_producer:
        producer_loop(mbar_group_tma, mbar_group_mma,
                      a_tma,b_tma, wg_producer)

    # Consumer performs MMA/Tensor Core
    with wg_consumer:
        D = consumer_loop(mbar_group_tma, mbar_group_mma,
                          a_tma,b_tma, wg_consumer)
        epilogue(D, d_dev)
```

Shared Memory Slots (Size of slot = Tile-A + Tile-B )



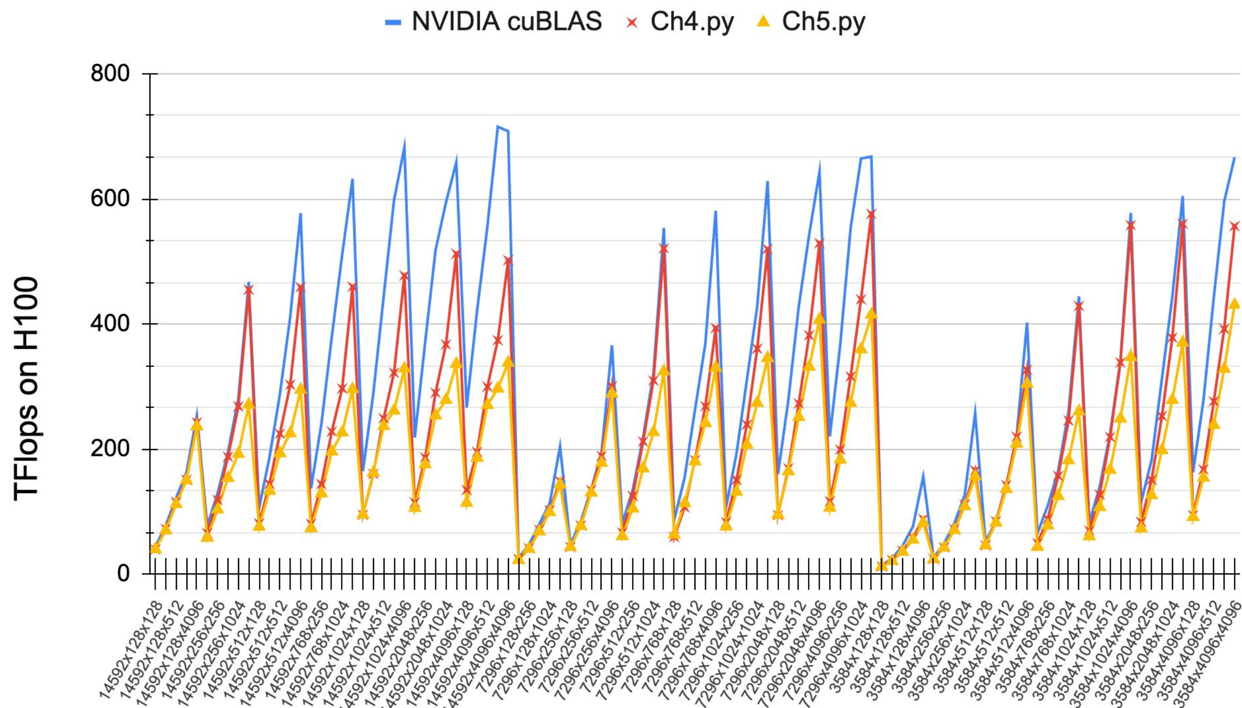
# Ch4.py vs Ch5.py

**Operation:**
- F32 += F16 * F16

**Tile Size:**
- 128 x 128 x 64

**Ch4.py**
- Multistage Kernel

**Ch5.py**
- Warp Specialized



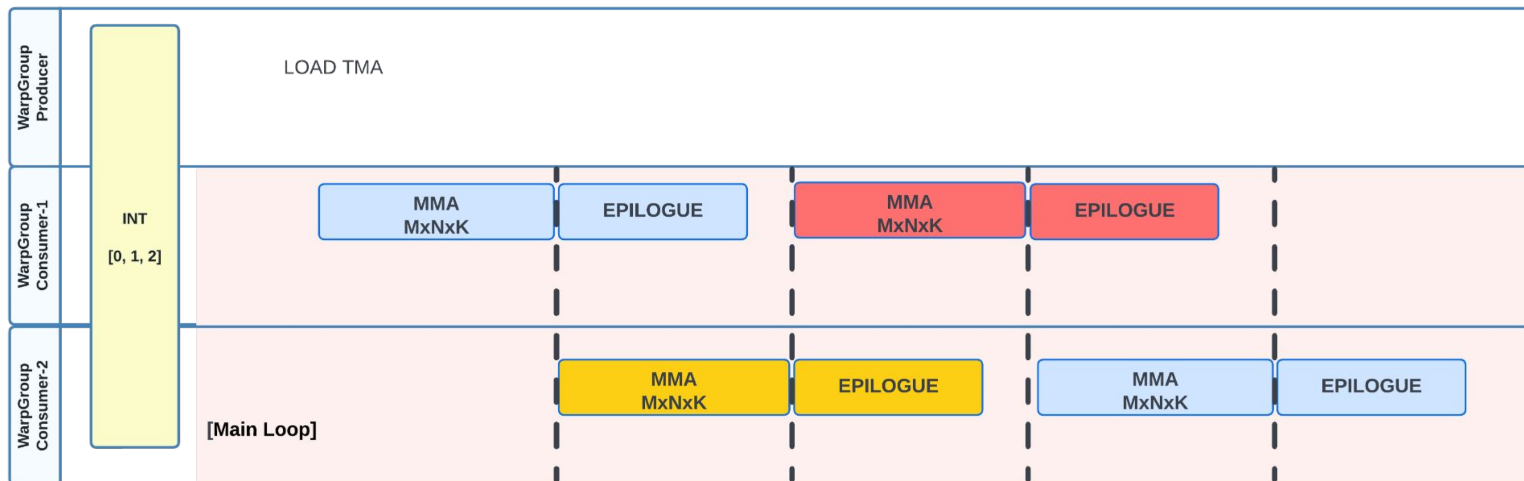
# What is next?

# **Ch6.py:** Warp Specialized Persistent Ping-Pong GEMM (WIP)

1 Producer and 2 Consumers Warpgroups

## Thread Block has 3 Warpgroups (384 threads):

Consumers Warpgroups MMA ←→ Epilogue



# Use MLIR’s NVGPU Dialect with Python

**NVGPU and NVVM Dialects**

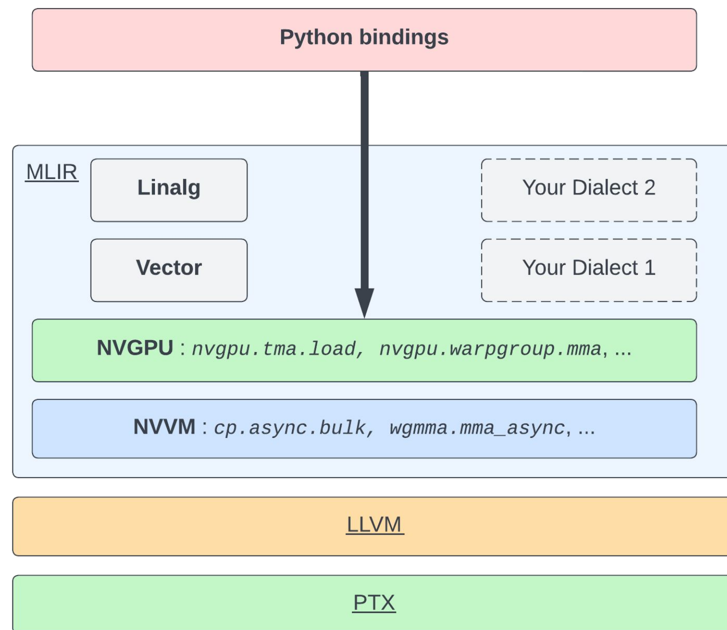
🔥 Hopper GPU Support

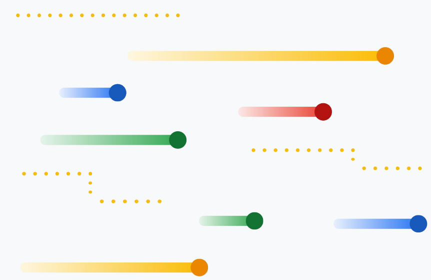
**Unlocking Hopper's Power: MLIR's Python Binding**

🔀 Seamlessly Express Multistage and Warp Specialization!

**Peak performance**

🚀 Achieve cuBLAS-Level performance





Google Research

# MLIR Upstream Dialect Layers

## What happens after NVVM Dialect?

Python bindings

MLIR

Linalg

Vector

Your Dialect 2

Your Dialect 1

NVGPU : `nvgpu.tma.load, nvgpu.warpgroup.mma`, ...

NVVM : `cp.async.bulk, wgmma.mma_async`, ...

LLVM

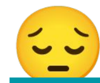
PTX

**BasicPtxBuilder**

**Today we will program**

- Python → NVGPU → NVVM

**LLVM doesn't have Hopper intrinsics**



**BasicPtxBuilder generates** `inline assembly`

# New Interface: BasicPtxBuilder

Builds PTX automatically (no C++ need)

Generates register constraints:

```
"h" = .u16 reg
"r" = .u32 reg
"l" = .u64 reg
etc.
```

Generates read/write

```
"r"(y)      read
"+r"(y)     readwrite
"=r"(y)     write
```

Supports predicates

```
@%p opcode
```

```
def NVVM_MBarrierArriveExpectTxOp : NVVM_Op<"mbarrier.arrive.expect_tx",
                      [DeclareOpInterfaceMethods<BasicPtxBuilderOpInterface>]>

Arguments<(ins LLVM_i64ptr_any:$addr, I32:$txcount, PtxPredicate:$predicate)> {

let assemblyFormat =
    "$addr `,` $txcount (`,` `predicate` `=` $predicate^)? attr-dict `:` type(operands)"   ;

let extraClassDefinition = [{
    std::string $cppClass::getPtx() {
      return std::string("mbarrier.arrive.expect_tx.b64 _, [%0], %1;"); }
    }];
}
```

Predicate is automatically placed

PTX instruction

Arguments are placed automatically