Bringing NVIDIA Blackwell GPU support to LLVM and MLIR

Durgadoss Ramanathan
Guray Ozen
Pradeep Kumar

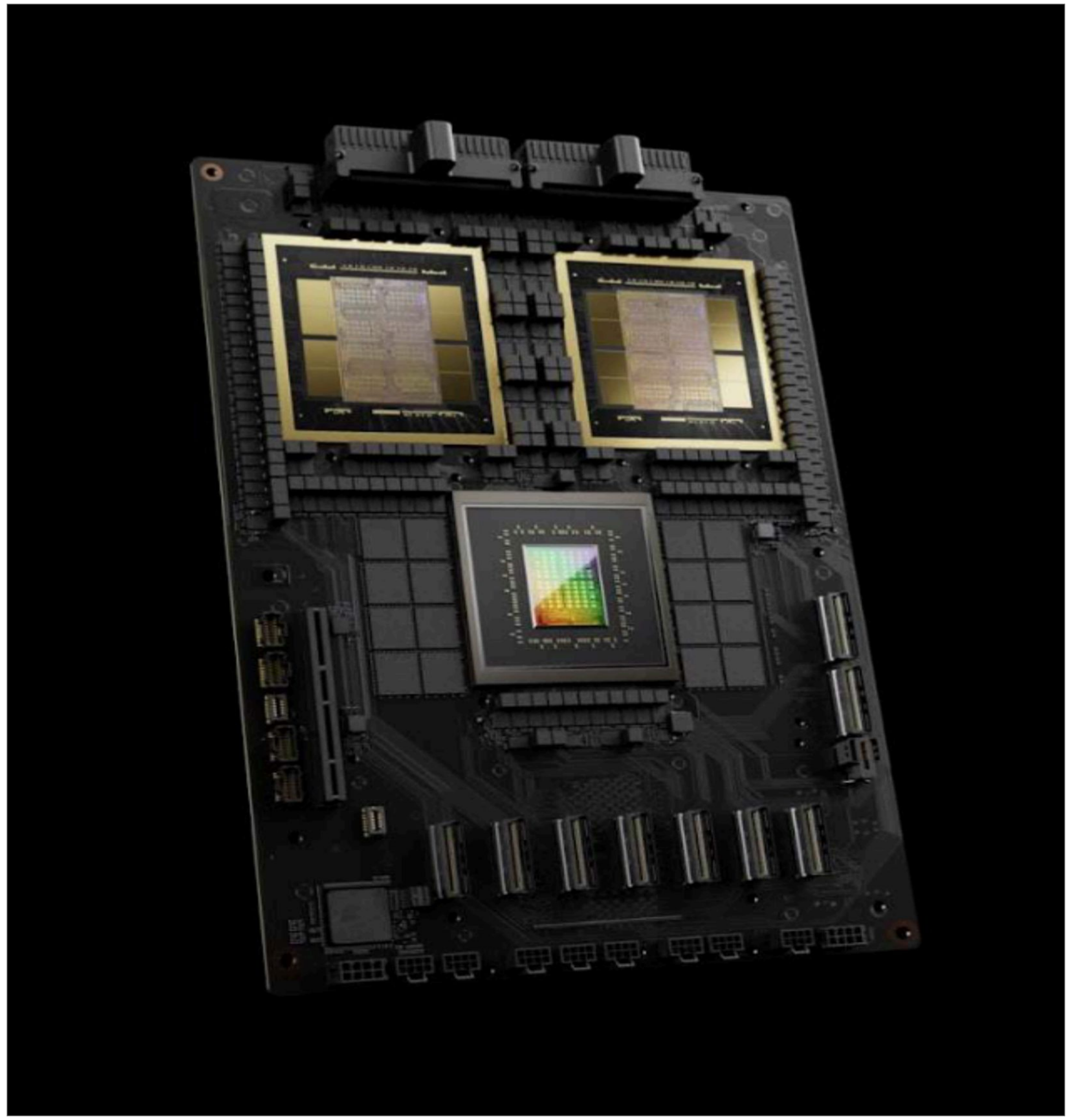
EuroLLVM 2025, Berlin, 15th April

Agenda

- Brief Intro on Blackwell GPU Features
- LLVM: NVPTX Support, APFloat additions
- MLIR: NVVM Dialect support
- Running GEMM with NVDSL Python

Introduction to Blackwell Features

- Enhancements to accelerate AI Compute
- On MMAs
  - Ampere(Warp) → Hopper(Warp Group) → Blackwell(a pair of CTAs)
  - New Tensor Memory (TMEM) per SM
    - Dedicated alloc/dealloc and load/store instructions
    - Accumulators/Operands in TMEM
  - Support for Block-scaled types
  - Exposed as the ‘tcgen05’ instruction family
- On TMAs (Tensor Memory Accelerator)
  - Newer modes like im2col_w/scatter/gather
  - Masked Copy support
- Conversions, Arithmetics etc.



Blackwell - GTC 2025

Compiler Lowering Flow

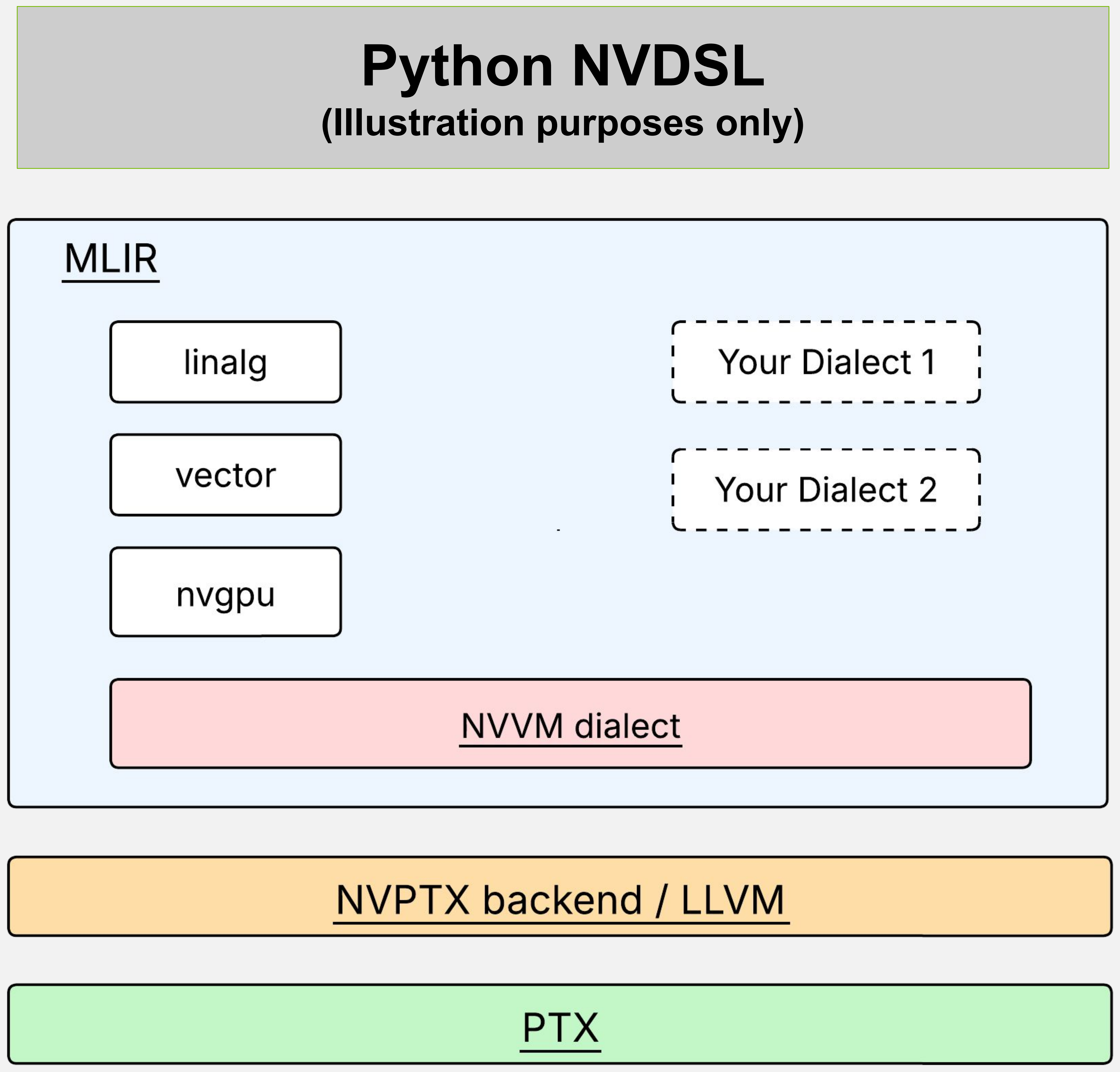
MLIR → LLVM → PTX

NVVM Dialect

- Low level operations (closer to PTX)
- NVVM Ops to NVVM/LLVM intrinsics

NVPTX Backend

- NVVM Intrinsics → PTX

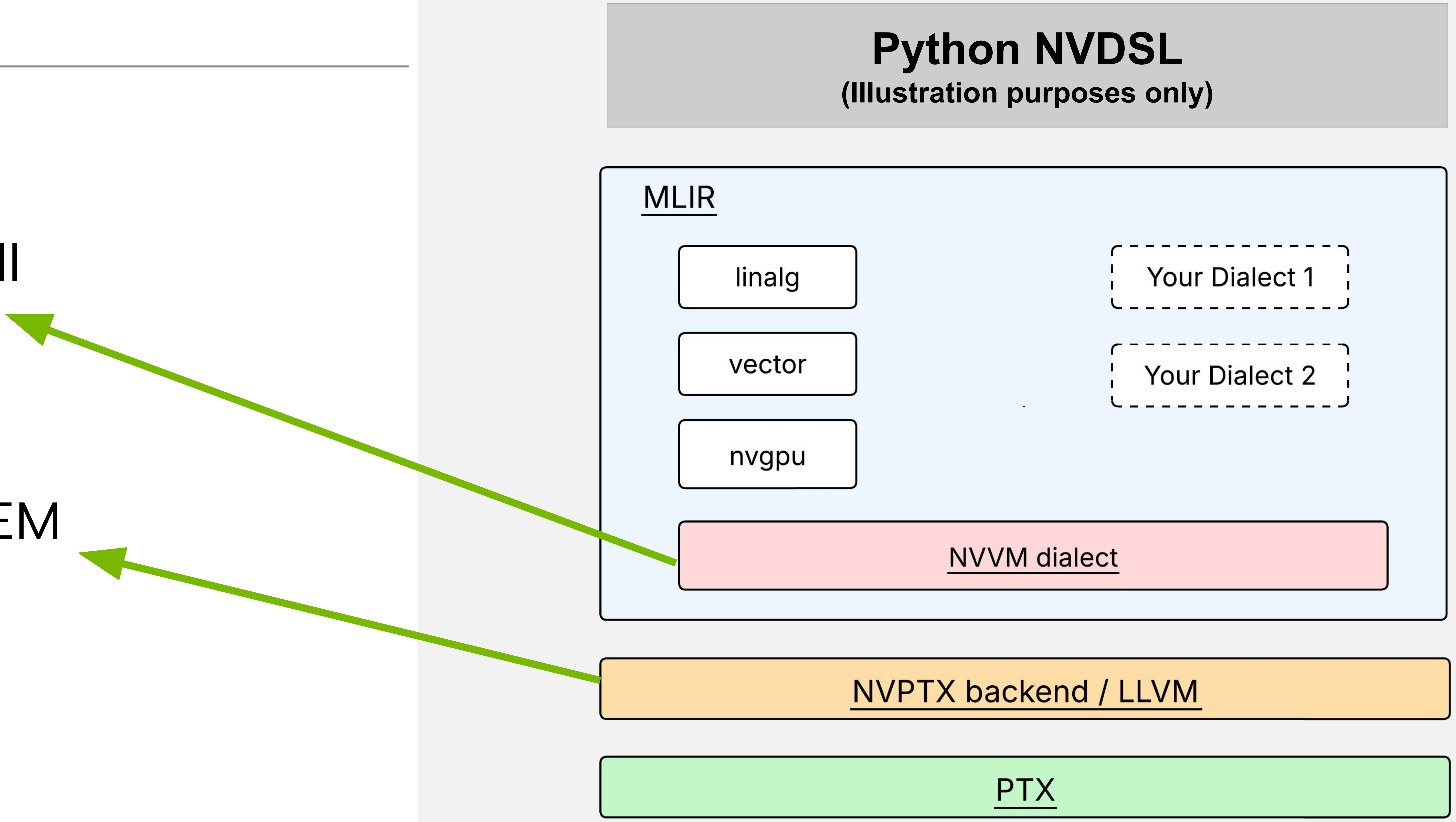


Blackwell additions

MLIR → LLVM → PTX

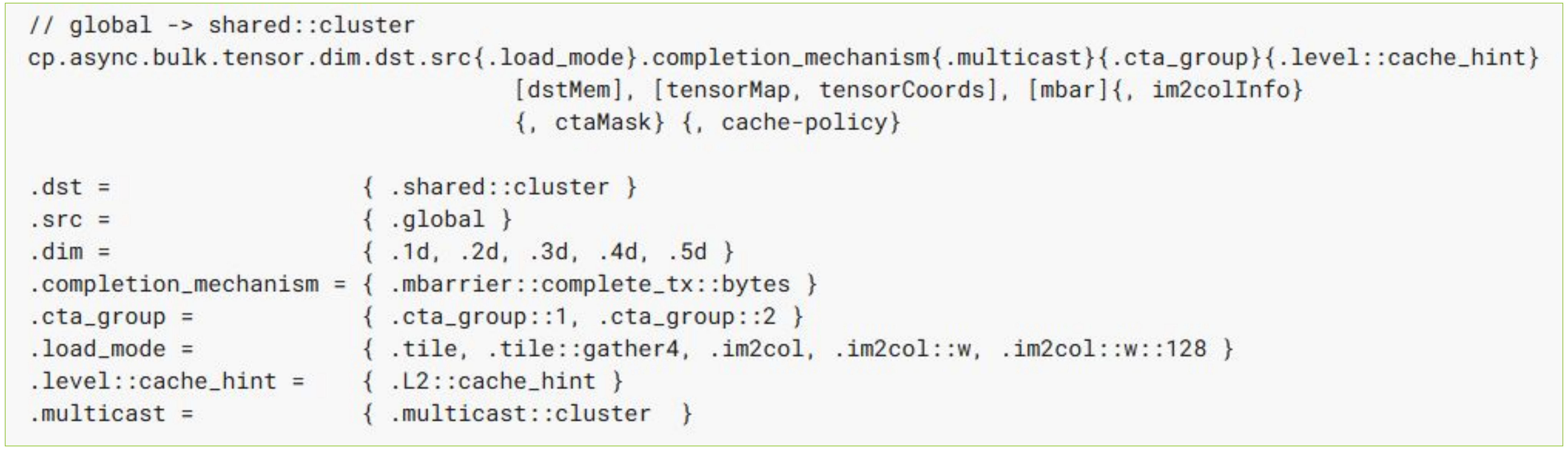
+ ~30 ops for Blackwell

+ Addrspace 6 for TMEM
+ ~1k intrinsics



Modelling as Intrinsics

One PTX instruction → Many Intrinsics



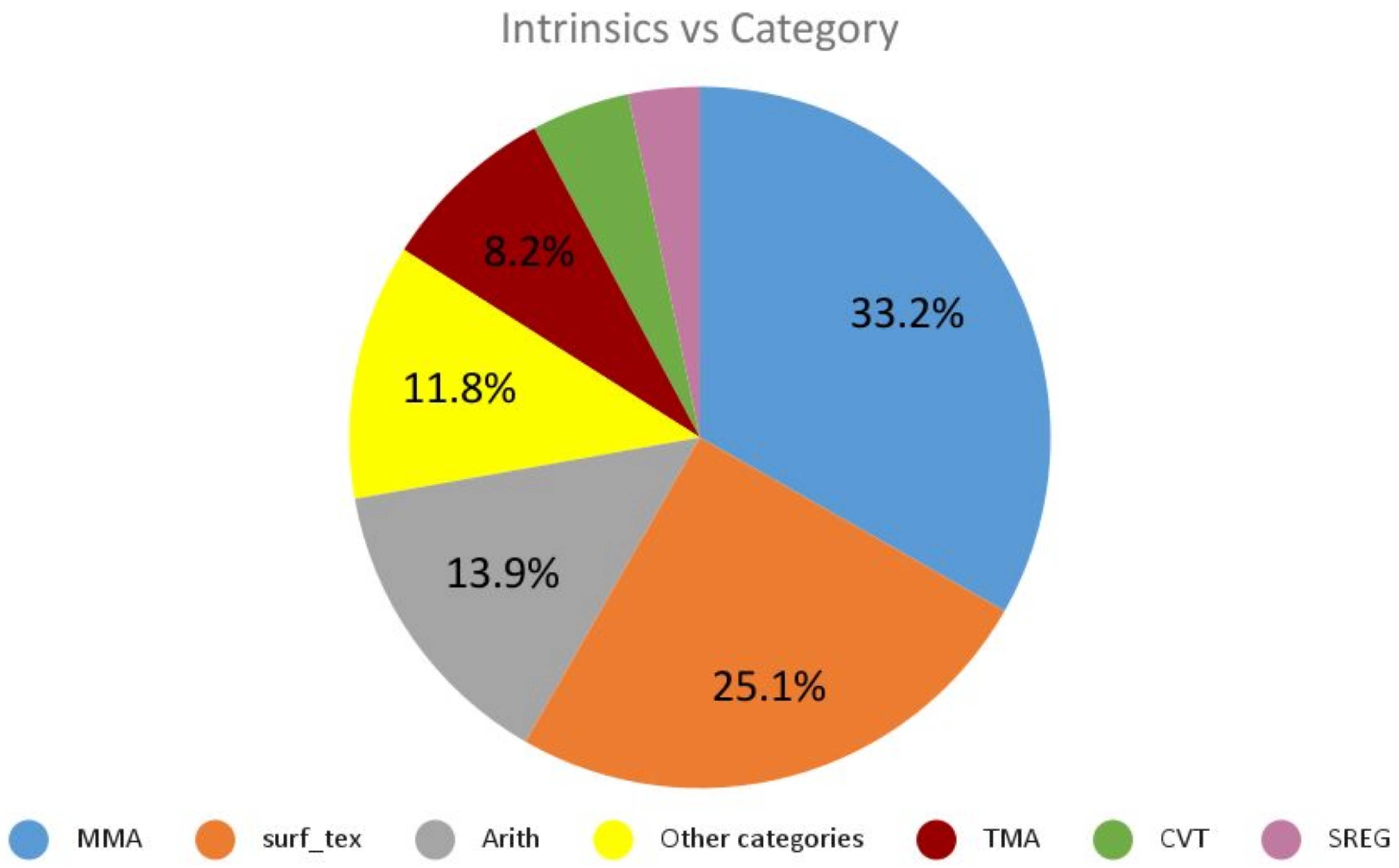
```
// global -> shared::cluster
cp.async.bulk.tensor.dim.dst.src{.load_mode}.completion_mechanism{.multicast}{.cta_group}{.level::cache_hint}
                                   [dstMem], [tensorMap, tensorCoords], [mbar]{, im2colInfo}
                                   {, ctaMask} {, cache-policy}

.dst =                    { .shared::cluster }
.src =                    { .global }
.dim =                    { .1d, .2d, .3d, .4d, .5d }
.completion_mechanism = { .mbarrier::complete_tx::bytes }
.cta_group =              { .cta_group::1, .cta_group::2 }
.load_mode =              { .tile, .tile::gather4, .im2col, .im2col::w, .im2col::w::128 }
.level::cache_hint =     { .L2::cache_hint }
.multicast =              { .multicast::cluster  }
```

An example TMA Instruction

- A single TMA variant → hundreds of intrinsics (dims x modes x cta_group x cache-hint x multicast)
- The Tcgen05-MMA family alone can expand up to ~750 individual intrinsics
  - Types (int/float), re-use hints, metadata scaling/sparsity, masks etc.

Distribution of Intrinsics

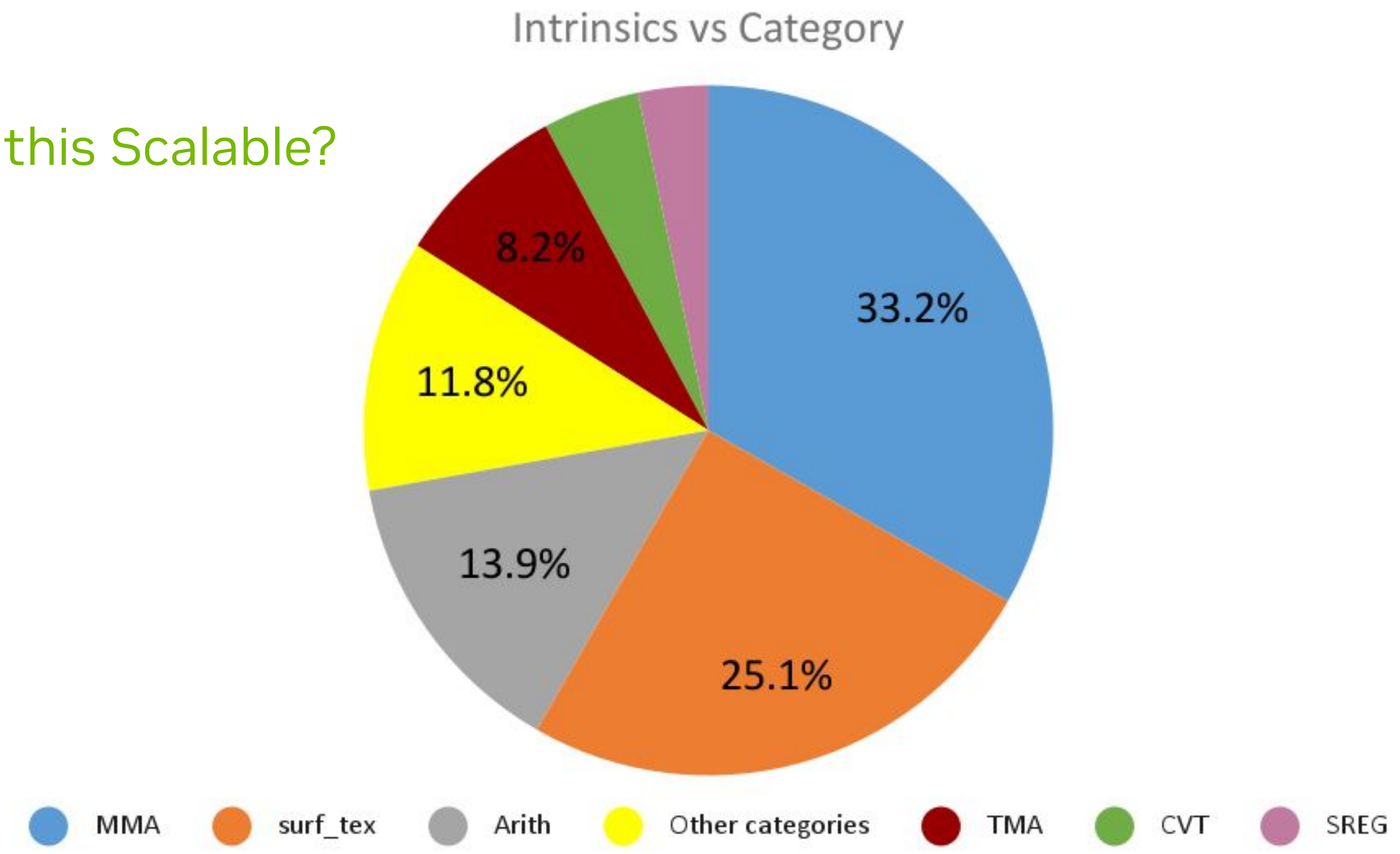
MMA/TMAs are ~40% of NVVM IR Intrinsics
Total: ~3k pre-Blackwell



Distribution of Intrinsics

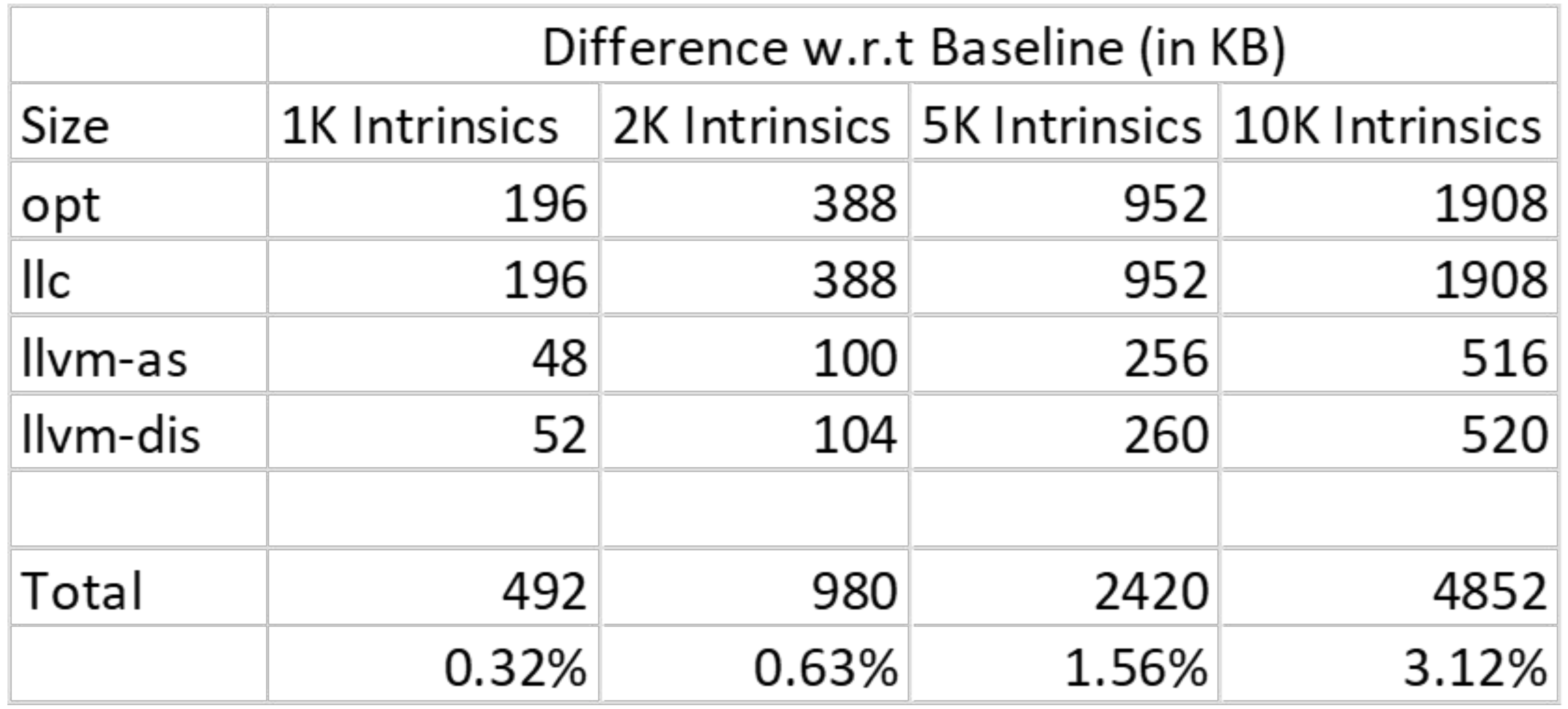
MMA/TMAs are ~40% of NVVM IR Intrinsics
Total: ~3k pre-Blackwell

Is this Scalable?



Adding Intrinsics - Binary Size

- Empirical observations on a set of {1k, 2k, 5k, 10k} Intrinsics
  - Modelled on top of a TMA intrinsic for the basic structure
  - Includes NVPTX Codegen (from td → td)

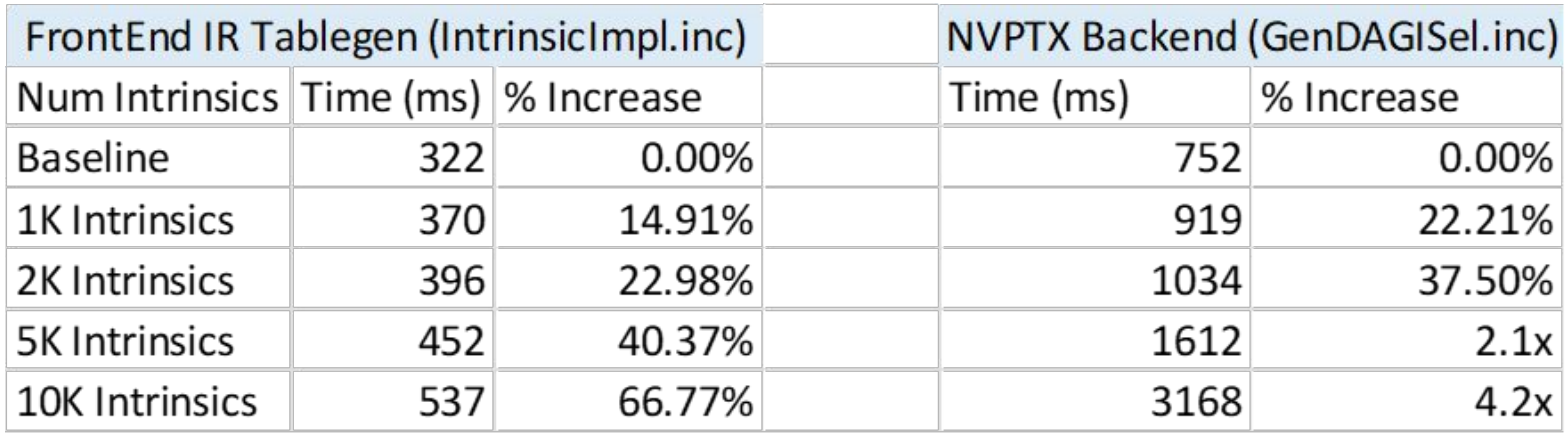


	Difference w.r.t Baseline (in KB)			
Size	1K Intrinsics	2K Intrinsics	5K Intrinsics	10K Intrinsics
opt	196	388	952	1908
llc	196	388	952	1908
llvm-as	48	100	256	516
llvm-dis	52	104	260	520
Total	492	980	2420	4852
	0.32%	0.63%	1.56%	3.12%

Impact on Binary Size (RELEASE)

Adding Intrinsics - LLVM Build Time

- Building the necessary include files from Tablegen



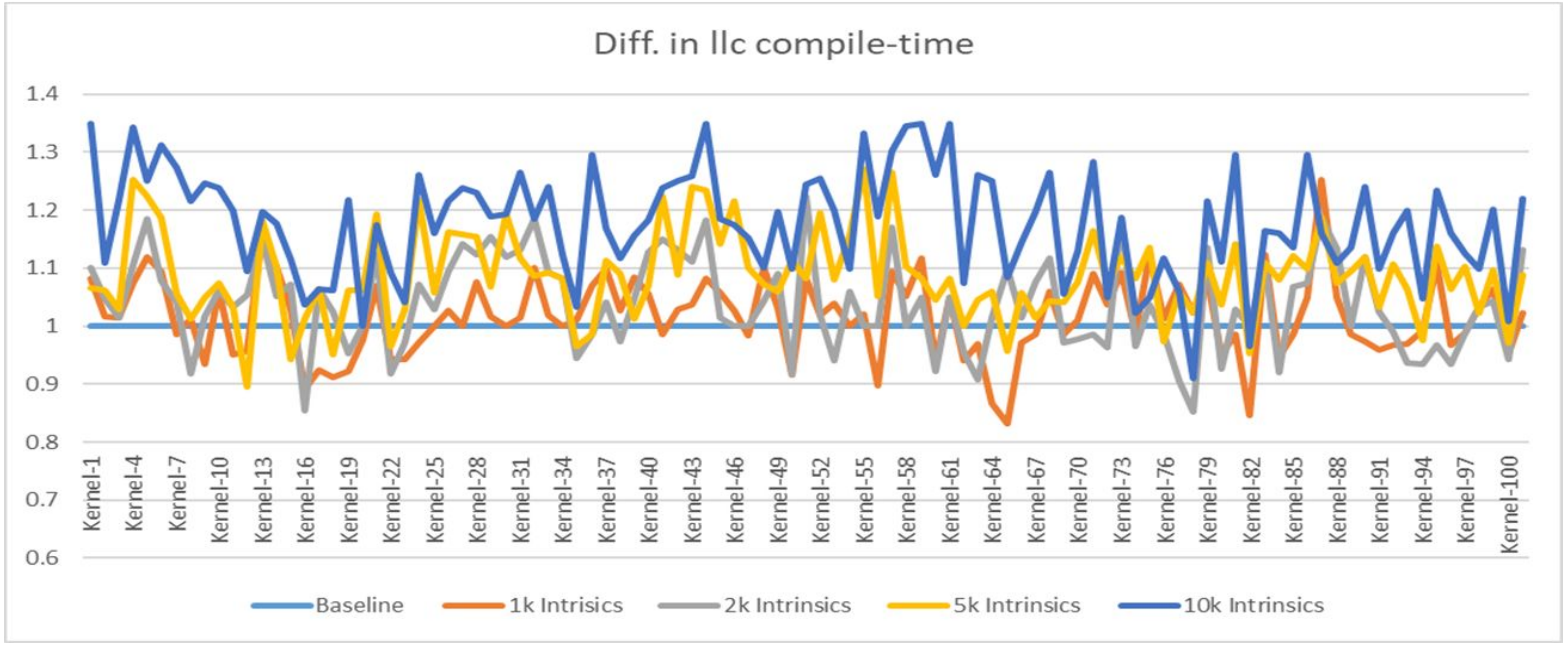
FrontEnd IR Tablegen (IntrinsicImpl.inc)				NVPTX Backend (GenDAGISel.inc)	
Num Intrinsics	Time (ms)	% Increase		Time (ms)	% Increase
Baseline	322	0.00%		752	0.00%
1K Intrinsics	370	14.91%		919	22.21%
2K Intrinsics	396	22.98%		1034	37.50%
5K Intrinsics	452	40.37%		1612	2.1x
10K Intrinsics	537	66.77%		3168	4.2x

Impact on building the necessary include files

- But the impact on overall build time is None… (make -j8 with Release=1)

Adding Intrinsics - Compile Time

- Study over a set of 100 blackwell kernels, using the intrinsics discussed so far..
  - Impact on Compile time
    - Measured by 'time-passes' option (average of 20 runs)
    - Separately for opt and llc phases with NVCC(cicc) release builds
    - No impact on the opt phase



Num Intrinsics	GeoMean
1k Intrinsics	1.01
2k Intrinsics	1.04
5k Intrinsics	1.08
10k Intrinsics	1.17

Consolidating Intrinsics - Packed flags

- Using 'flag' operands to encode the modifiers (PR 96083 - v1)
  - Compile time integer constants with appropriate Docs!
  - A single 'flag' with many bit fields representing the modifiers
  - Backend uses the flag to generate the correct variant
  - cp.async.bulk.tensor.3d.***tile.multicast.cache_hint***(arg1, arg2, ...) → cp.async.bulk.tensor.3d(**i32 %flag**, arg1, arg2, ...)

```
typedef union {
    int V;
    struct {
        unsigned CacheHint: 1;
        unsigned Multicast: 1;
        unsigned LoadMode: 3; // CpAsyncBulkTensorLoadMode
        unsigned NumCTAs: 1;
        unsinged reserved: 28;
    } U;
} CpAsyncBulkTensorFlags;
```

```
enum class CpAsyncBulkTensorLoadMode {
    TILE = 0,
    IM2COL = 1,
    IM2COL_W = 2,
    IM2COL_W_128 = 3
};
```

Consolidating Intrinsics - Packed flags

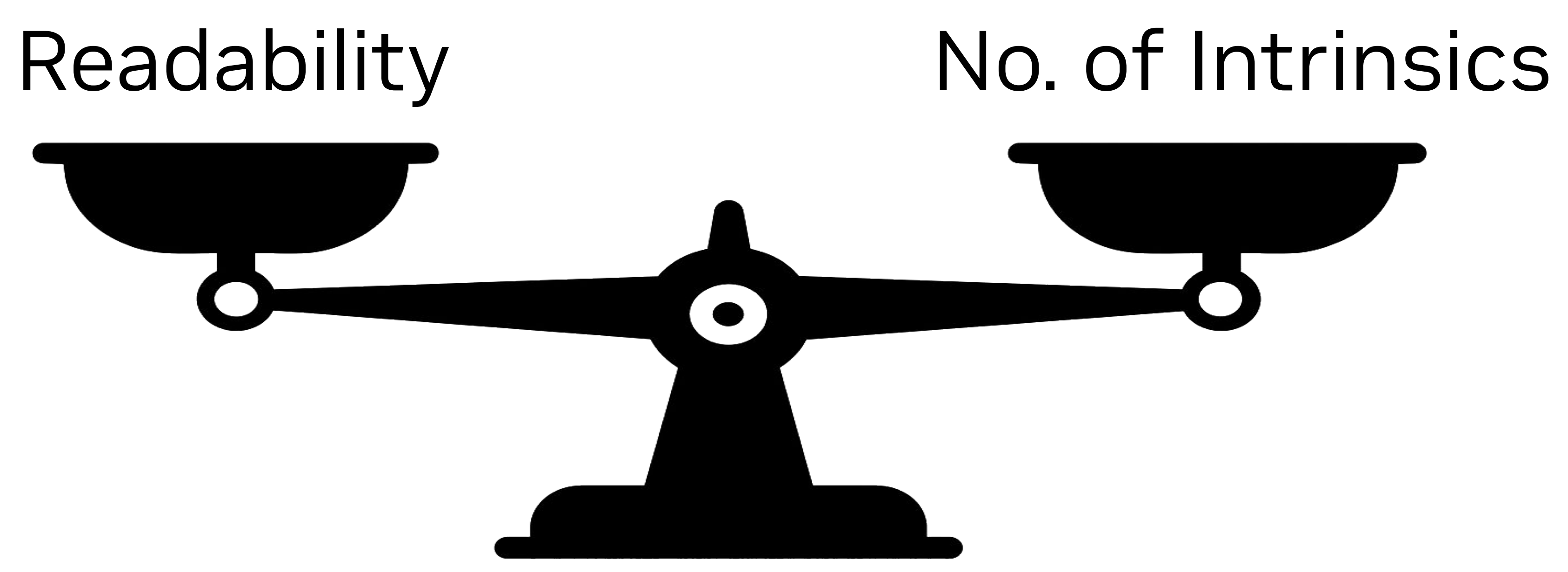
- Worked well for maintainability (w.r.t numbers) and compatibility
- However...
  - Handling a particular field when it exceeds bit-width.
  - As the intrinsic becomes more complicated,
    - very hard to decipher the flag field(s)
      - Unpack and then map!
      - `call void @llvm.nvvm.test(i64 u0x3C220000030000, arg1, arg2, ..)`

Consolidating Intrinsics - Separate flags

- Instead of one flag for *the entire intrinsic* (PR 96083)
    - Use a 'flag' field for *one set* of instruction modifiers
    - cp_async_bulk_tensor_var1/var2/...varN(arg1, arg2, arg3 ...) →
    - cp_async_bulk_tensor(**i1 %flag_mc, i1 %flag_ch, i32 %flag_mode,** arg1, arg2, ...)
- (Practically) Never run out of bit-width
- Much easier for IR debug/hacking! (flag-value == enum value)
- Impact on Bit-code size: Up to a max of 3% for a few kernels
    - But the benefit of readability is much more!

In summary…

- Balance between readability and the aspect of proliferation!



- Few guidelines to help design intrinsics for NVPTX - PR 133136

APFloat additions

- APFloat Infra support for FP6/FP4 types from OCP-MX Specification
  - FP6 types (e2m3, e3m2) PR 94735
  - FP4 type (e2m1) PR 95392
- The scale-format type e8m0 presented many challenges: PR 107127
  - Key learning: Teach APFloat to handle “precision=0”
  - Corner-cases revealed by “Float8E8M0FNU*ExhaustivePair*” (APFloatTest.cpp)
- All these have type-definition support in MLIR now!

NVVM Dialect support

- Except MMA, all the tcgen05 operations are supported in NVVM Dialect!
    - These are lowered through NVPTX Intrinsics!
    - Ops for pre-Blackwell features are also being actively added
- Support for inline-ptx as well as Intrinsics lowering
    - NVVM Ops can lower to inline-ptx using the PTXBuilder
    - One Op can be lowered partially to either of the flows too!
- Actively migrating the lowering to intrinsics-based
    - ~14/90 Ops remain to be migrated.

The Road Ahead...

- The ability to pretty-print Imm intrinsic args (flags)
  - RFC: discourse 82629
- From tcgen05 intrinsics enabling (PR 124961)
  - Ability to attach addr-space specific RW properties to Intrinsics

- On the NVVM Dialect
  - Doing SM version checks for NVVM Ops (under review PR 126886)
  - Continue adding support for the remaining instructions/intrinsics

Running GEMM example with NVDSL Python

NVDSL: Simple NVGPU/NVVM Dialect usage

We focus on the Performance

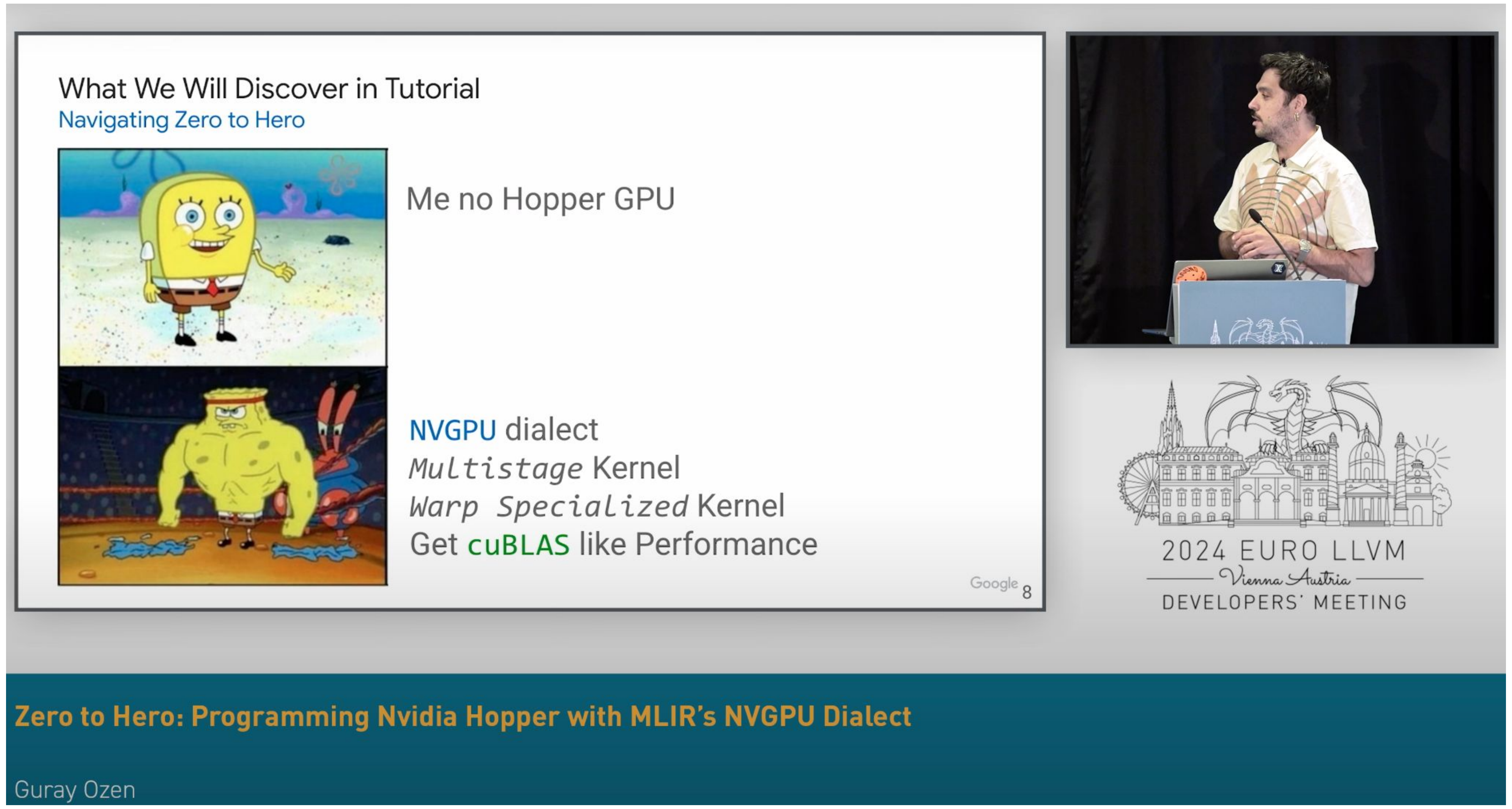
Introduced in EuroLLVM24

- Zero to Hero: Programming Nvidia Hopper Tensor Core

Testing DSL

Simplifies:

- MLIR Function building via decorators
- JIT Compilation and Execution
- Operator Overloading with Arith Dialect
- NumPy -> memref translation



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

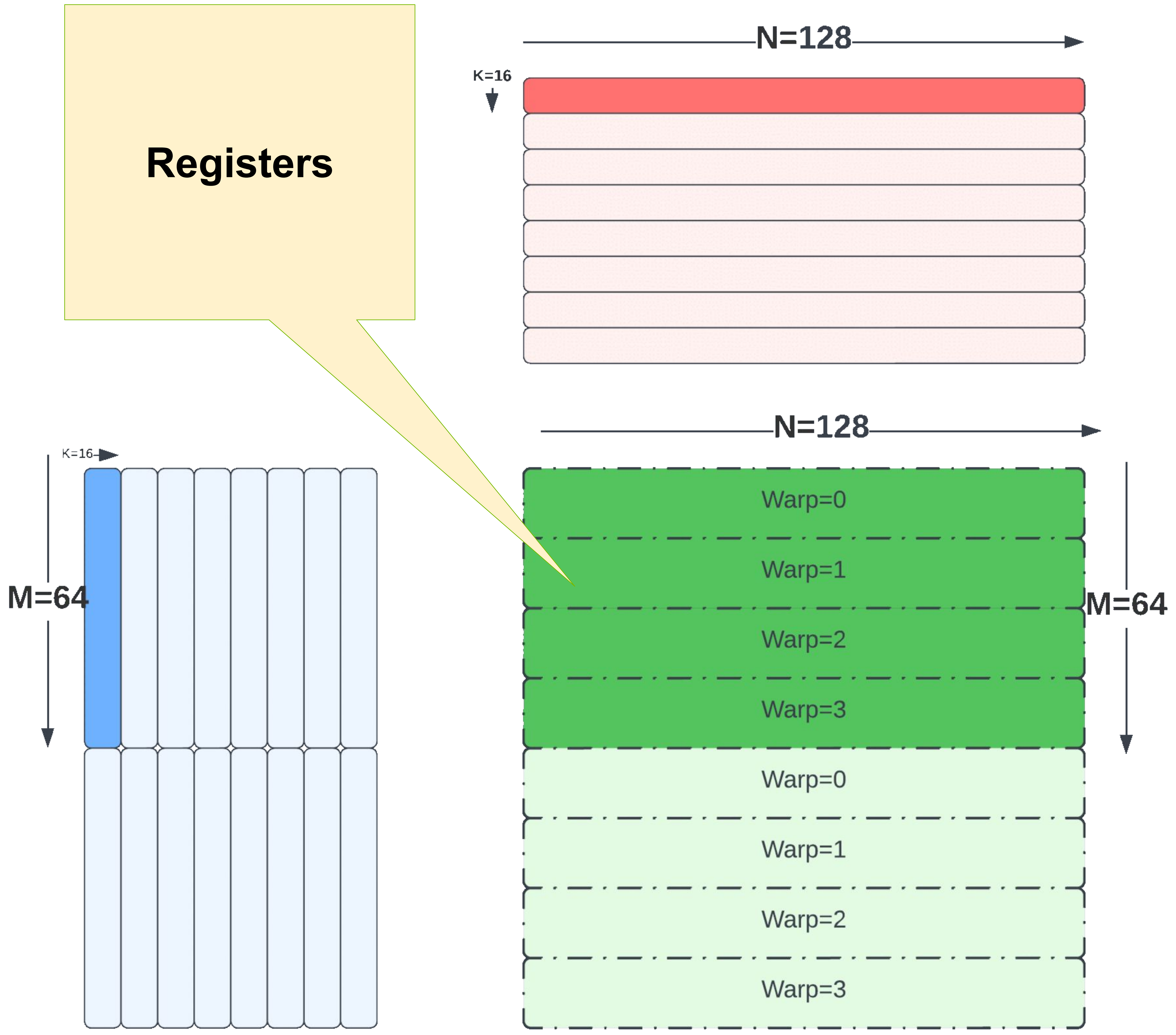
Zero to Hero: Programming Nvidia Hopper with MLIR's NVGPU Dialect

Guray Ozen

Recap: How NVIDIA Hopper Tensor Core works?

Accumulate and Store Results in **Registers**

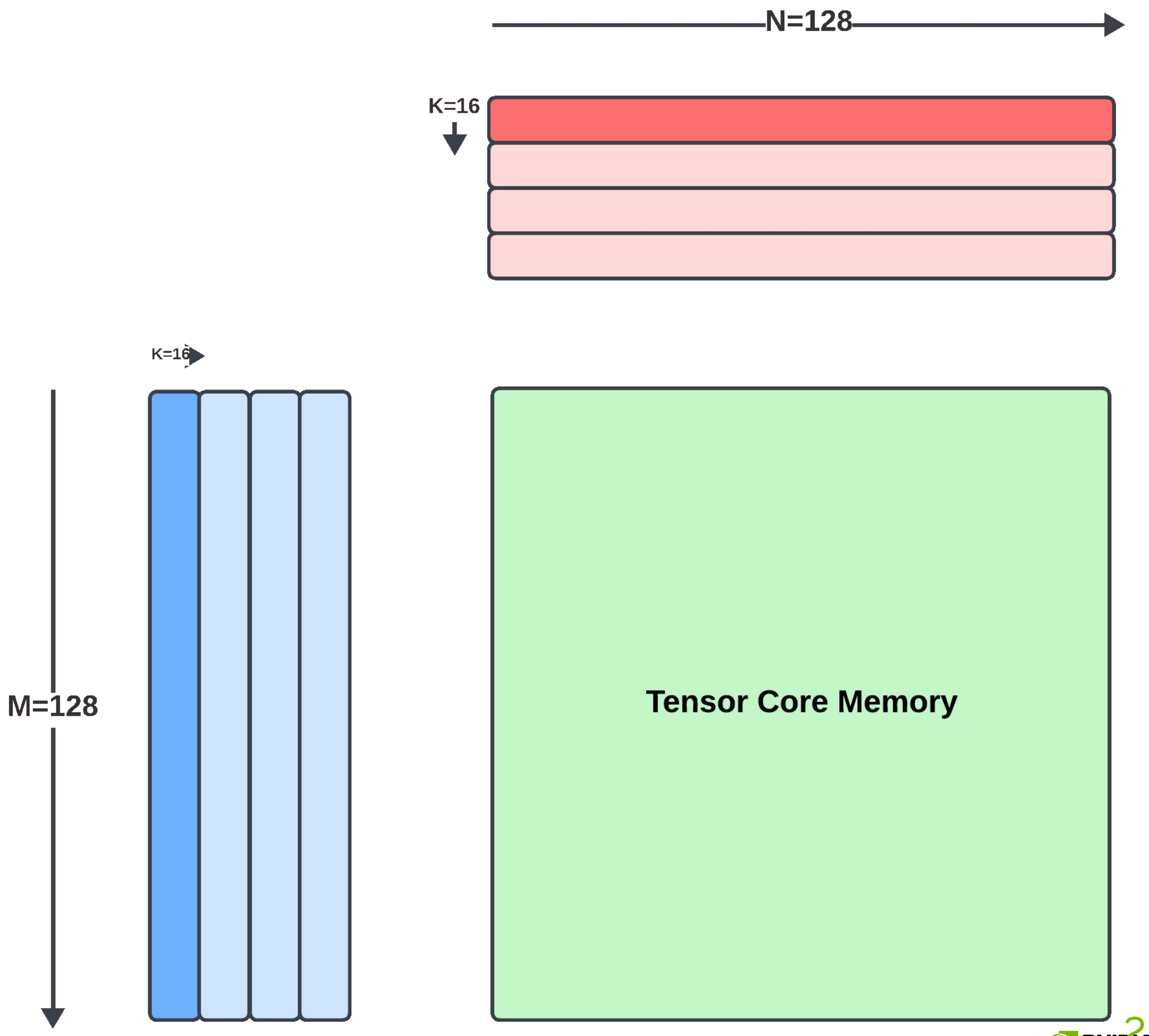
```
// Initialize input matrix: 2x64xf32 Registers
%r = 0 : !llvm.struct<(...)>
//              m     n    k
//     GEMM:  128 x 128 x 128
//    WGMMA:   64 x 128 x 16
//   factor:    2 x   1 x  4
nvvm.wgmma.fence.aligned

%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
```



What is new in Blackwell in tcgen05?

Accumulate and Store Result in TMEM (Visible to Other Warps)

```
// Load tmem
%0 = llvm.load %ptr : !llvm.ptr<3> -> i32
%tmem = llvm.inttoptr %0 : i32 to !llvm.ptr<6>

//               m     n    k
//     GEMM:   128 x 128 x 64
//    WGMMA:   128 x 128 x 16
//   factor:     1 x   1 x  4

%w1 = nvvm.tcgen05.mma %tmem, %dA,      %dB,     %idesc, …
%w2 = nvvm.tcgen05.mma %tmem, %dA+2,    %dB+128, %idesc, …
%w3 = nvvm.tcgen05.mma %tmem, %dA+4,    %dB+256, %idesc, …
%w4 = nvvm.tcgen05.mma %tmem, %dA+6,    %dB+384, %idesc, …

// Commit arrive with mbarrier
nvvm.tcgen05.commit.arrive %mbarrier : !llvm.ptr<3>
```



Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to 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]
```

Mission:

- Launch a kernel with a thread block
- Offload the GEMM to tensor core

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core

```python
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]
```

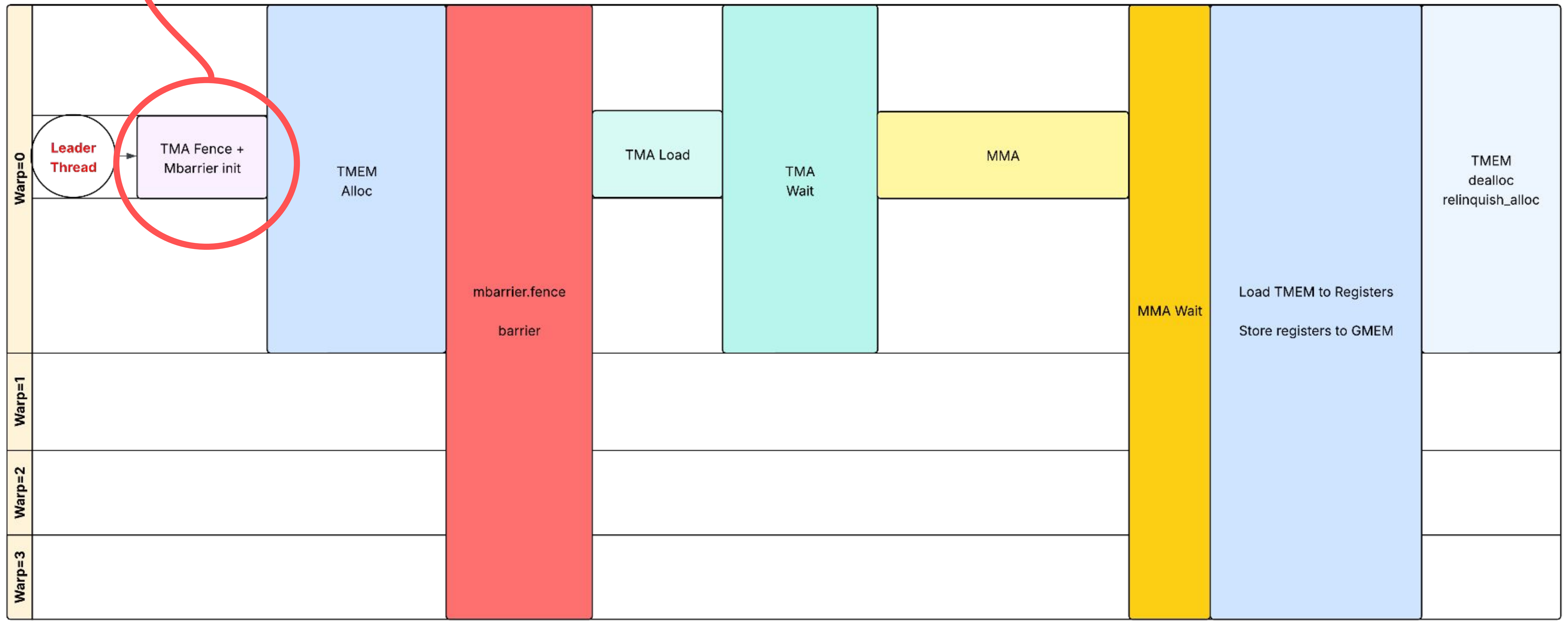
in NVDSL

```python
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16, CTA_1, tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Mission:
- Launch a kernel with a thread block
- Offload the GEMM to tensor core

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core



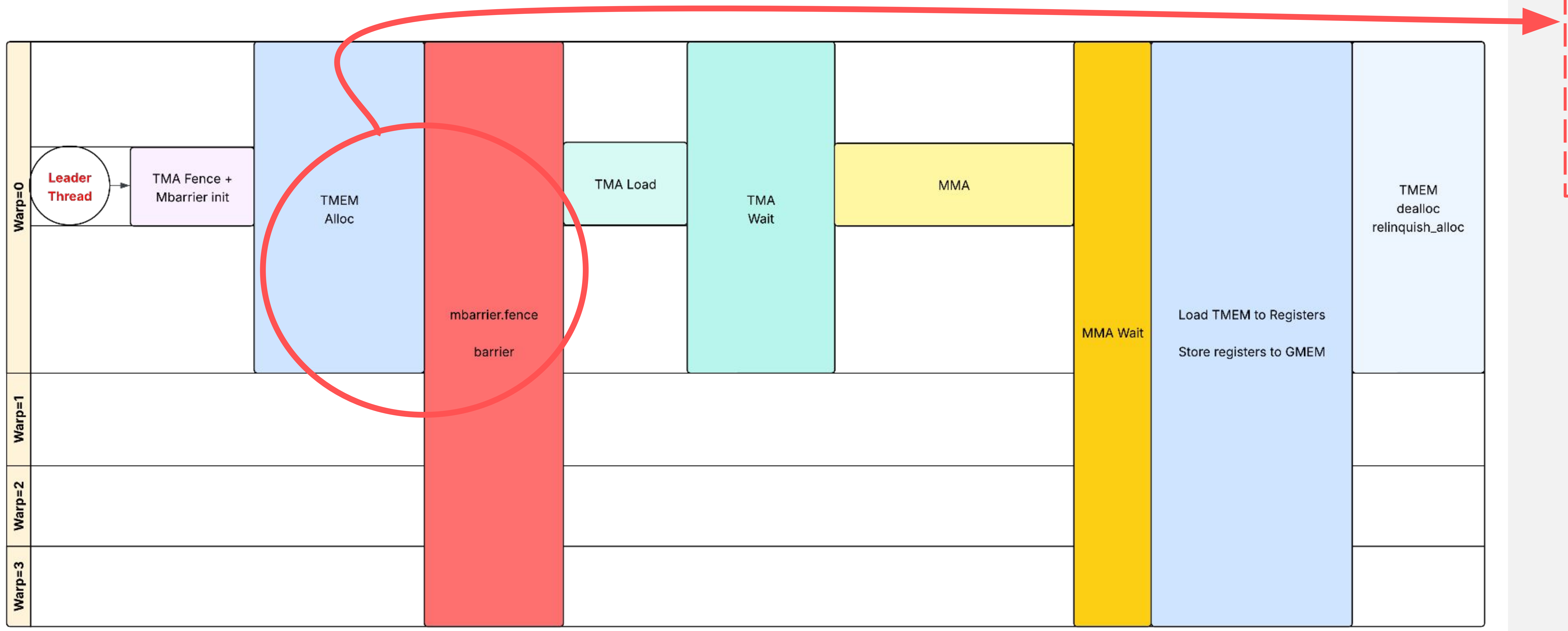
Step 1:

- TMA fence for TMA descriptors
- Create 2 MBarriers and initialize them

```
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16,CTA_1,tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core



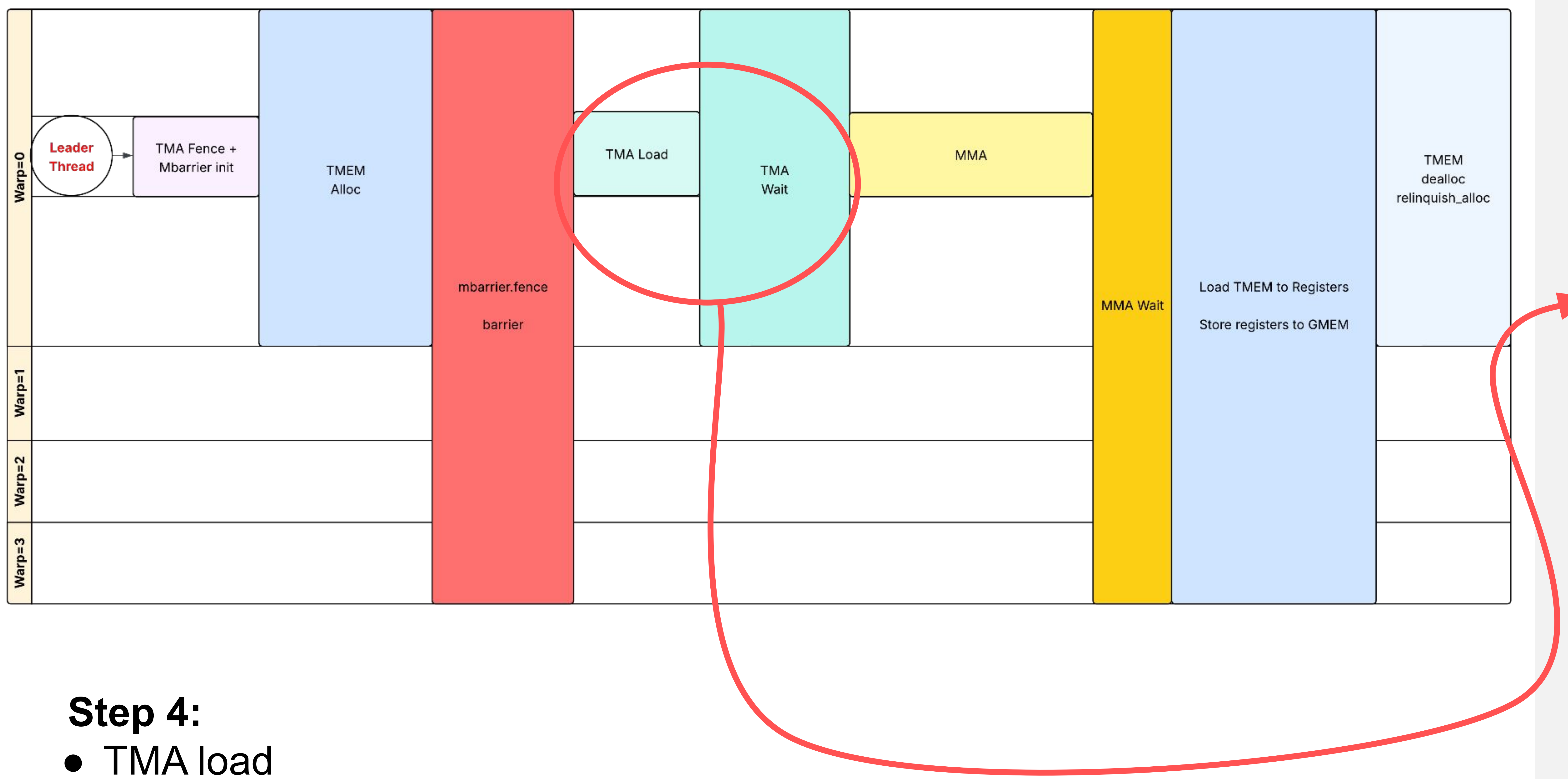
Step 2:
- Allocate TMEM

Step 3:
- Barrier

```python
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16,CTA_1,tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core



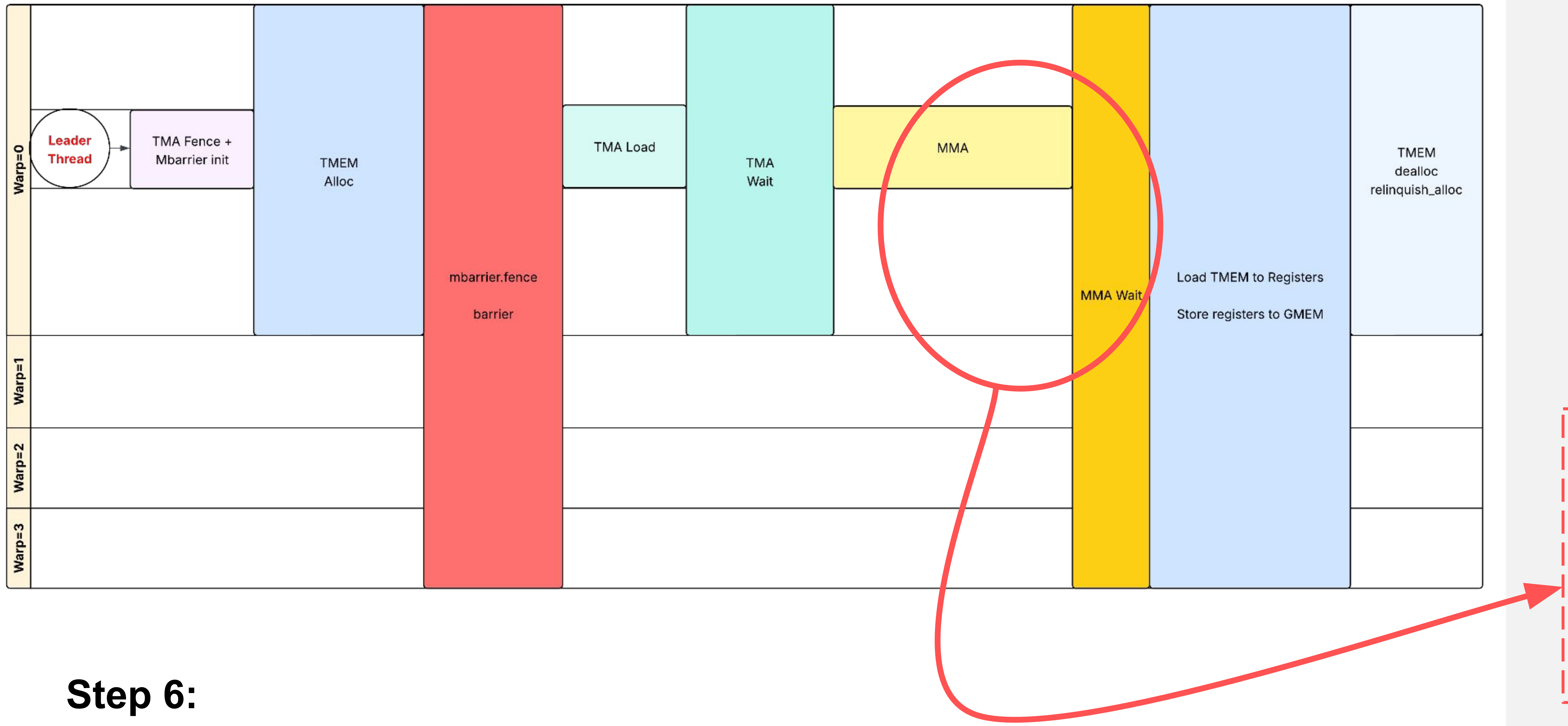
Step 4:
- TMA load

Step 5:
- Wait the TMA Load

```
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16,CTA_1,tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core



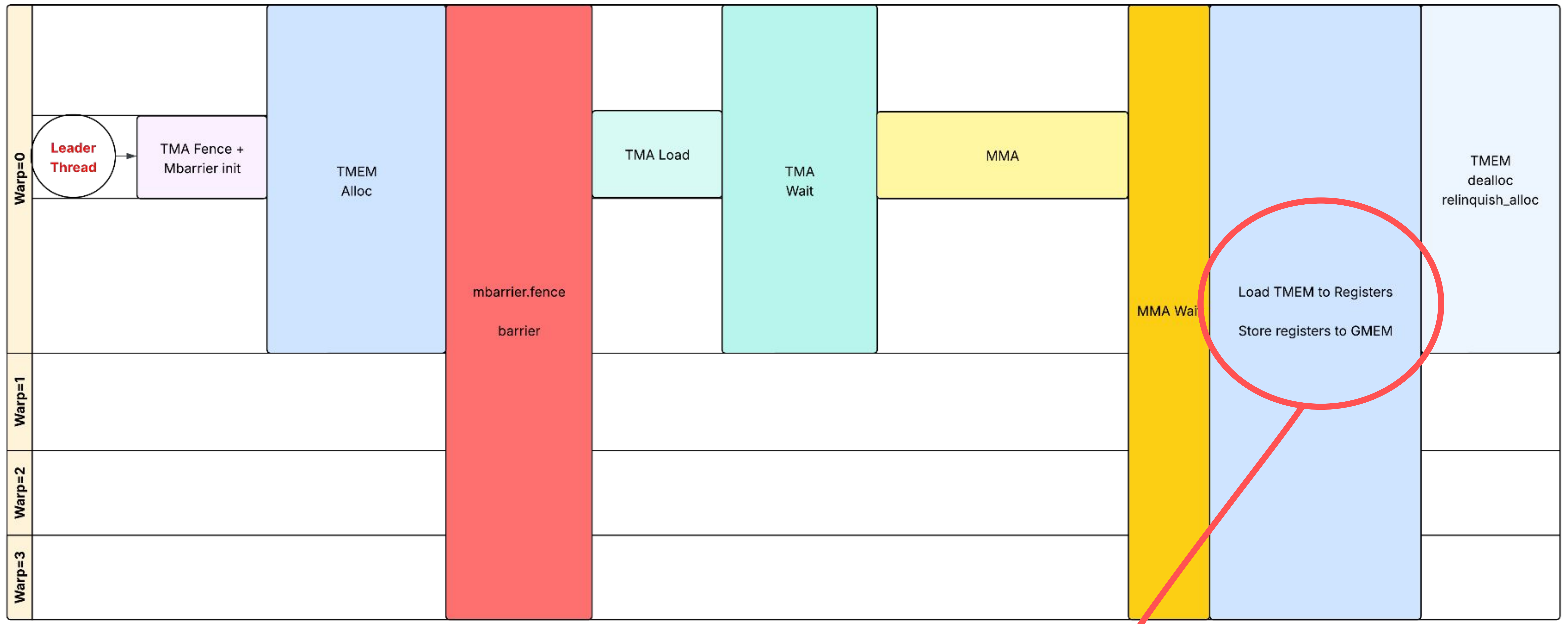
Step 6:
- MMA Tensor Core

Step 7:
- Wait the MMA Tensor Core

```
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16,CTA_1,tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core



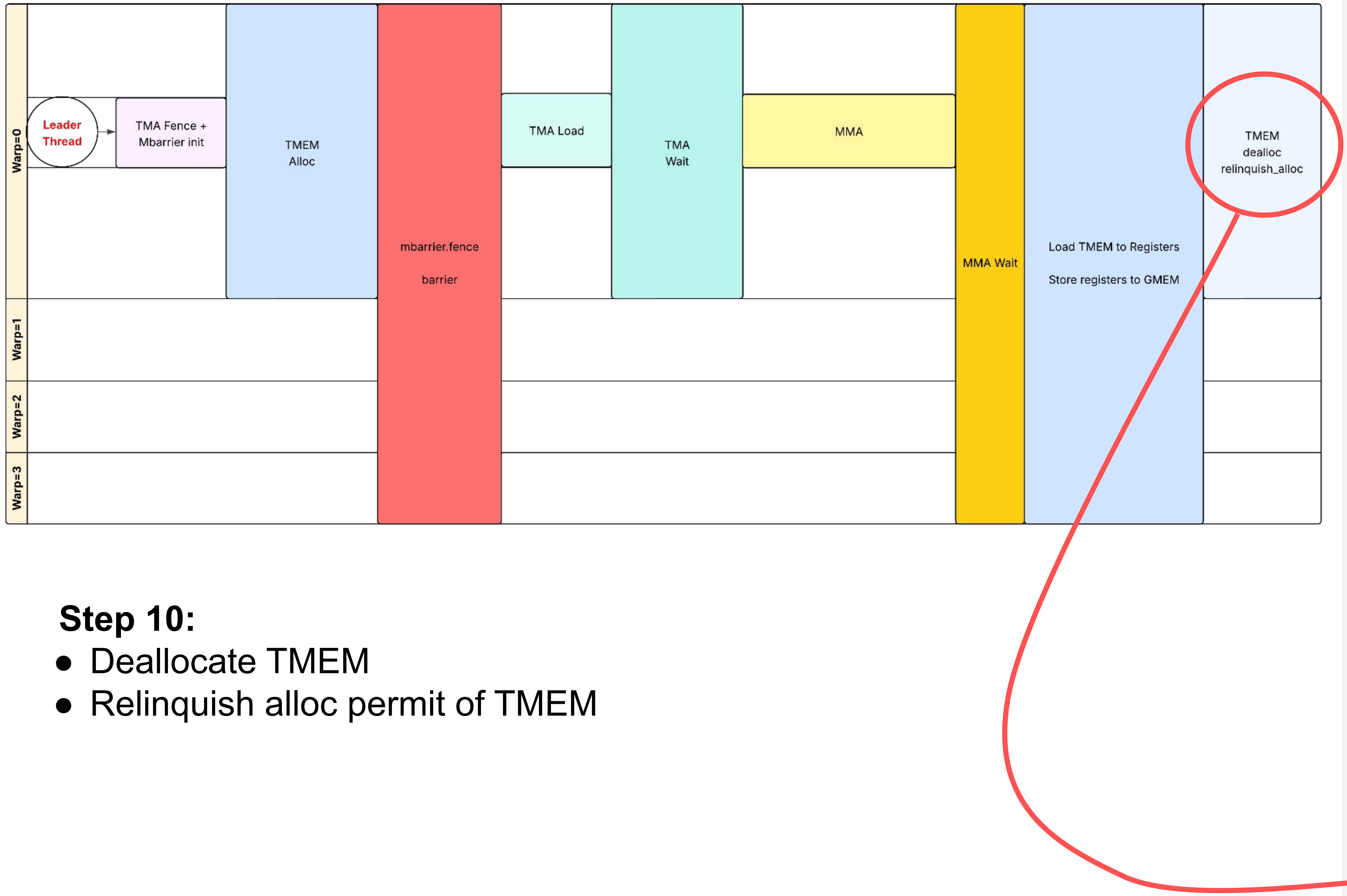
Step 8:
- Load TMEM to registers

Step 9:
- Store registers to global memory

```
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16,CTA_1,tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Ch3_sm100.py: GEMM 128x128x64

Launch 1 CTA, offload GEMM to Tensor Core



Step 10:

- Deallocate TMEM
- Relinquish alloc permit of TMEM

```
@NVDSL.mlir_gpu_launch(grid=(1, 1, 1), block=(128, 1, 1), smem=bytes)
def gemm_kernel():
    mbar_group = Mbarriers(2)
    # Step 1. TMA fence + Initialize mbarrier
    with if_generate(warpIdx == 0 and elect_one()):
        a_tma.fence()
        b_tma.fence()
        mbar_group[0].init(1)
        mbar_group[1].init(1)
    # Step 2: Allocate 128 col TMEM resources.
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_alloc(c_ptr, 128)
    # Step 3. This fence orders the mbar init operations with respect to the barrier below
    nvvm.fence_mbarrier_init()
    nvvm.barrier()
    # Step 4: Performed TMA load only by the leader thread
    with if_generate(warpIdx == 0 and elect_one()):
        tma_load(mbar_group, a_tma, b_tma)
    # Step 5: Wait TMA Load
    with if_generate(warpIdx == 0):
        mbar_group[0].try_wait()
    # Step 6: Tensor Core MMA
    with if_generate(warpIdx == 0 and elect_one()):
        for i in range(4):
            nvvm.tcgen05_mma(F16,CTA_1,tmem,dA + i * 2,dB + i * 128,idesc)
        nvvm.tcgen05_commit_arrive(mbar_group[1].get())
    # Step 7: Wait Tensor Core MMA
    mbar_group[1].try_wait()
    # Step 8: Load TMEM to registers
    acc = nvvm.tcgen05_ld(nvvm.Tcgen05LdStShape.SHAPE_32X32B, tmem, vType128)
    # Step 9: Store registers to GMEM
    for iv in scf.for_(128):
        memref.store(...)
    # Step 10: TMEM dealloc, relinquish permit
    with if_generate(warpIdx == 0):
        nvvm.tcgen05_dealloc(tmem, 128)
        nvvm.tcgen05_relinquish_alloc_permit()
```

Conclusion

Bringing NVIDIA Blackwell support to LLVM and MLIR

- **MLIR and LLVM Support for Blackwell:**

  NVIDIA is working towards supporting Blackwell in upstream MLIR and LLVM

  ✓ Hopper-Baseline work is complete in both NVPTX and NVVM Dialect.

  ✓ Tcgen05 family base modelling is complete.

  ✓ APFloat type support for newer types in Blackwell is complete.

- Blackwell updates for remaining TMA, MMA etc. are in progress
    - Thanks to all the reviewers and Maintainers!

- **Providing Runnable Examples via NVDSL:**

  ✓ Provide Blackwell Runnable example via NVDSL using NVGPU/NVVM dialect.

  ✓ One can build own compiler based on example in NVDSL.