Google Research

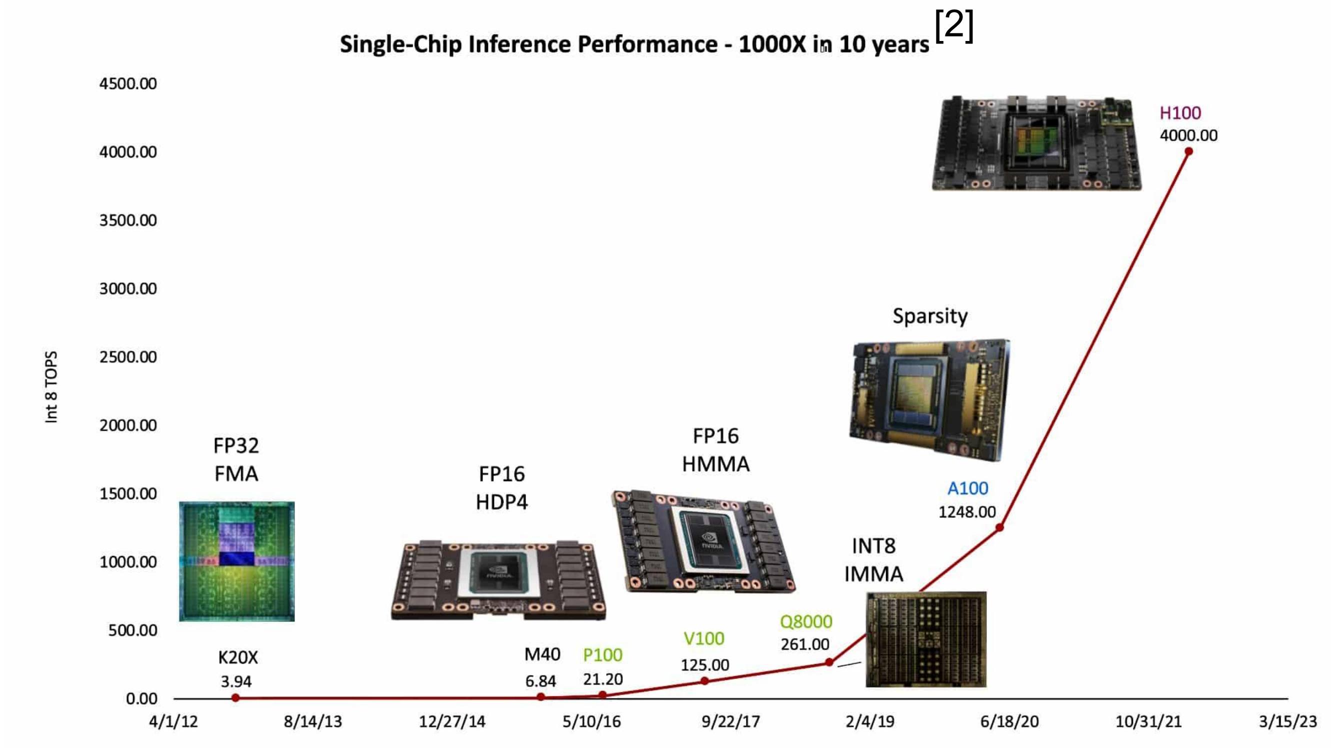
Targeting H100 in MLIR

Guray Ozen

November 2023, MLIR Open Design Meeting

Google Research

Huang's Law [1]



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

Google Research

Evolution

Lifespan of A100 (~2 years)

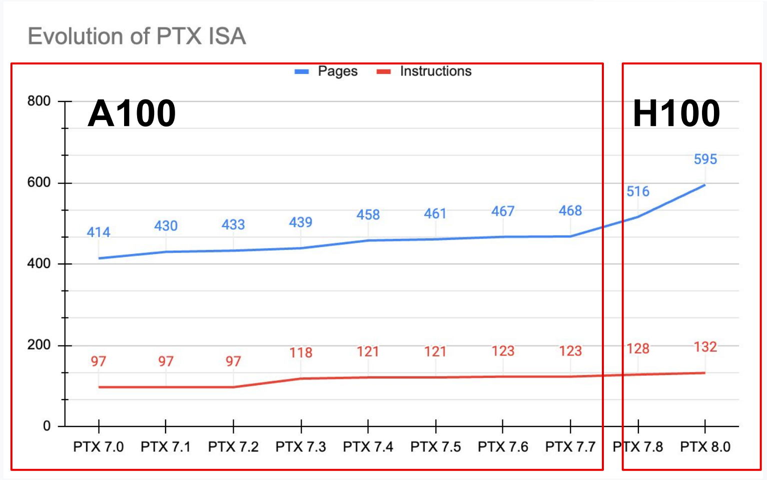
PTX*[1]* CUTLASS*[2]* significantly increased

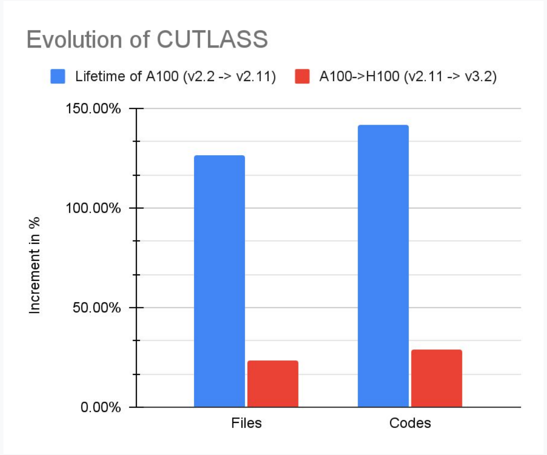
H100

Added even more complexity

Did MLIR & LLVM keep up?

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





Anatomy of Simple Fused ML Kernel
Where is the challenge?

Google Research

```
GPU kernel(matrixA, matrixB, otherData) {
 for ivM = blockIdx.x to M step TileM {
   for ivN = blockIdx.y to N step TileN {

    // GEMM
    for ivMM = 0 to TileM step 1
     for ivNN = 0 to TileN step 1
      for ivK = 0 to K step 1
       matrix-D += matrix-A * matrix-B

    // Element-wise Op (consumer of GEMM)
    for ivMM = 0 to Z step 1
     sum += matrix-D[ivMM]
   }
 }
}
```

Loop Tiling [introduced in `87]
- Any compiler can transform

GEMM [Hopper came in 2022]
- No perfect compiler solution
- How to handle warp-specialization

Element-wise GPU codegen [introduced in 2010]
- Any compiler can generate

Target H100 in MLIR

Focusing on Tensor Core of H100

Code generation for Tensor Core of H100

MLIR Compiler

MLIR is community-driven

Implement in `NVGPU` and `NVVM` Dialects

These dialects serve as building blocks across multiple compilers (e.g., Triton, IREE, etc.)

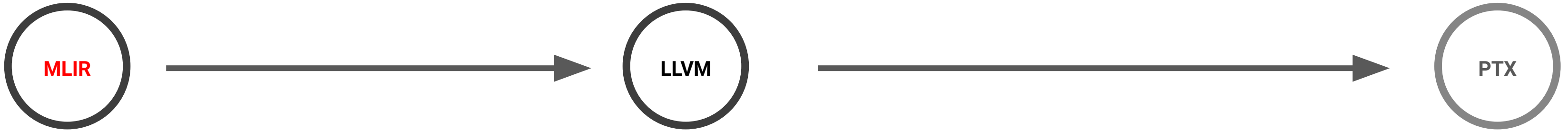
Upstream

All the work presented in this slide has been upstreamed to MLIR

NVVM Dialect

What is NVVM Dialect?

A dialect that maps to NVPTX intrinsic of LLVM



NVVM Dialect

```
%0 = nvvm.vote.ballot.sync
     %arg0, %arg1 : i32
```

NVPTX intrinsic

```
%0 =
call i32 @llvm.nvvm.vote.ballot.sync(i32 %mask, i1 %pred)
```

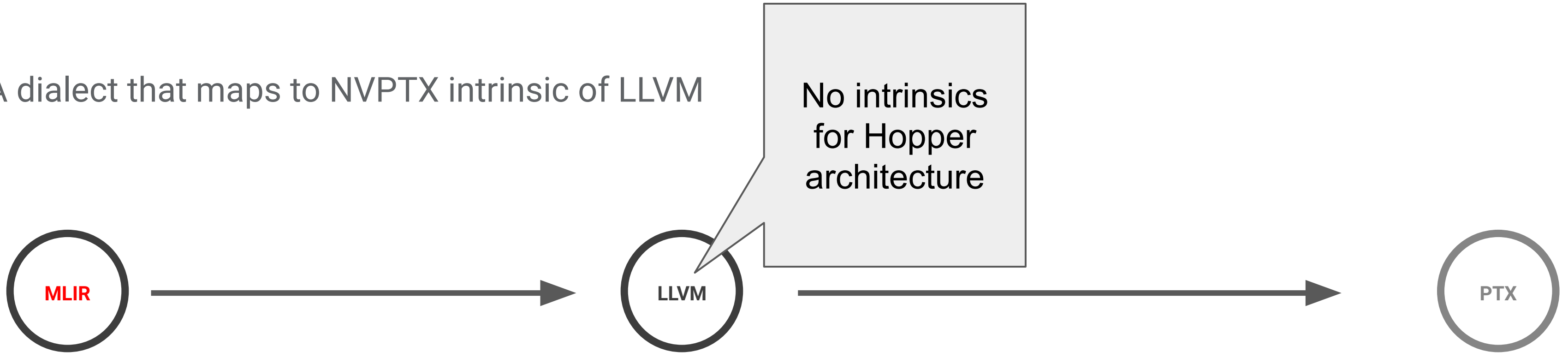
PTX Instruction

```
vote.sync.ballot.b32
%r5, %p1, %r3;
```

Google Research

What is NVVM Dialect?

A dialect that maps to NVPTX intrinsic of LLVM



NVVM Dialect | **NVPTX intrinsic** | **PTX Instruction**

```
nvvm.mbarrier.arrive.expect_tx.
shared %1, %2, predicate = %p :
!llvm.ptr<3>, i32, i1
```

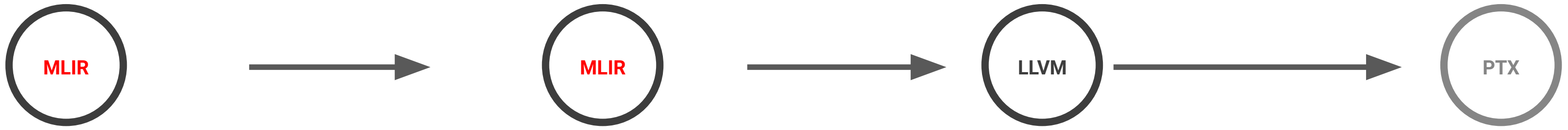
```
????
```

```
@%p1
mbarrier.init.shared.b64
[%r272], 0x1;
```

Google Research

What is NVVM Dialect?

A dialect that maps to NVPTX intrinsic of LLVM



NVVM Dialect

```
nvvm.mbarrier.arrive.expect_tx.
shared %1, %2, predicate = %p :
!llvm.ptr<3>, i32, i1
```

BasicPtxBuilderInterface

```
llvm.inline_asm has_side_effects
asm_dialect = att
"@$2
mbarrier.arrive.expect_tx.shared.
b64 _, [$0], $1;",
"r,r,b"
%arg0, %arg1, %3
: (!llvm.ptr<3>, i32, i1) -> ()
```

NVPTX intrinsic

```
tail call void asm
sideeffect "@$2
mbarrier.arrive.expect_tx
.shared.b64 _, [$0],
$1;", ""() #3
```

PTX Instruction

```
@%p1
mbarrier.arrive.expect_tx.sh
ared.b64 _, [%r411], %r414;
```

Google Research

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
```

NVVM Op with BasicPtxBuilder interface in MLIR:

```
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;"); }
    }];
}
```

How is implemented in CUTLASS:

```
asm volatile ("mbarrier.arrive.expect_tx.shared.b64 _, [%0], %1;\n"
                          :: "r"(smem_int_ptr),
                             "r" (bytes));
```

Google Research

Example: `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
```

MLIR Snippet

```
%tidx = gpu.thread_id x
%p = arith.cmpi eq, %tidx, %c0 : index
nvvm.mbarrier.arrive.expect_tx.shared %1, %2, predicate = %p : !llvm.ptr<3>, i32, i1
```

Generated LLVM Dialect (convert-nvvm-to-llvm)

```
llvm.inline_asm has_side_effects asm_dialect = att
"@$2 mbarrier.arrive.expect_tx.shared.b64 _, [$0], $1;",
"r,r,b"
%arg0, %arg1, %3 : (!llvm.ptr<3>, i32, i1) -> ()
```

Predicate automatically is generated

Register types are automatically generated

Generated PTX

```
mov.u32 %r125, %tid.x;
cvt.s64.s32 %rd2, %r125;
setp.eq.s64 %p1, %rd2, %rd222;

mov.u32 %r127, __mbarrier;
add.s32 %r272, %r127, %r126;
// begin inline asm
@%p1 mbarrier.init.shared.b64 [%r272], 0x1;
// end inline asm
```

Use-Case: `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
```

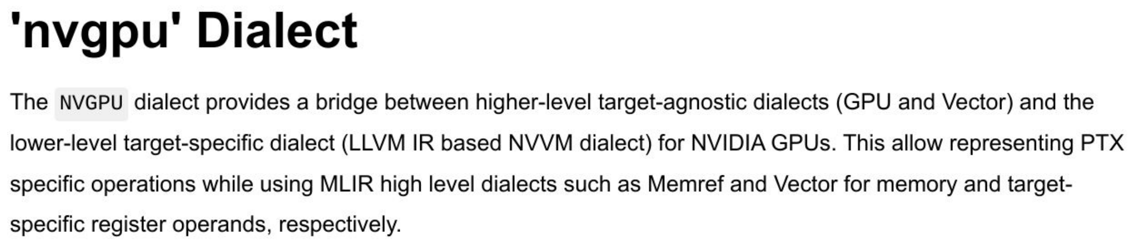
Supports predicates

```
@%p opcode
```

```
Split arrive/wait Barriers instructions
        nvvm.mbarrier.arrive
        nvvm.mbarrier.arrive.expect_tx
        nvvm.mbarrier.arrive.expect_tx.shared
        nvvm.mbarrier.arrive.nocomplete
        nvvm.mbarrier.arrive.nocomplete.shared
        nvvm.mbarrier.arrive.shared
        nvvm.mbarrier.init
        nvvm.mbarrier.init.shared
        nvvm.mbarrier.inval
        nvvm.mbarrier.inval.shared
        nvvm.mbarrier.test.wait
        nvvm.mbarrier.test.wait.shared
        nvvm.mbarrier.try_wait.parity
        nvvm.mbarrier.try_wait.parity.shared

Elect Leader
        nvvm.elect.sync

New warp-group level tensor core instructions
        nvvm.wgmma.commit.group.sync.aligned
        nvvm.wgmma.fence.aligned
        nvvm.wgmma.mma_async
        nvvm.wgmma.wait.group.sync.aligned


TMA load/store instructions
        nvvm.cp.async.bulk.tensor.global.shared.cta
        nvvm.cp.async.bulk.tensor.shared.cluster.global

Prefetch
        nvvm.prefetch.tensormap

Warp-level matrix store instruction
        nvvm.stmatrix
```

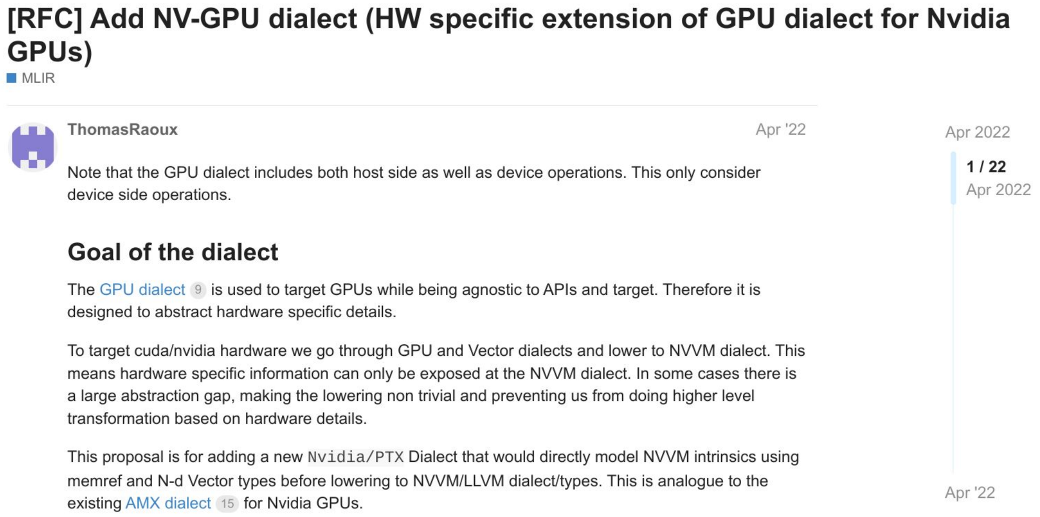
NVGPU Dialect

What is NVGPU Dialect?



'nvgpu' Dialect

The `NVGPU` dialect provides a bridge between higher-level target-agnostic dialects (GPU and Vector) and the lower-level target-specific dialect (LLVM IR based NVVM dialect) for NVIDIA GPUs. This allow representing PTX specific operations while using MLIR high level dialects such as Memref and Vector for memory and target-specific register operands, respectively.



[RFC] Add NV-GPU dialect (HW specific extension of GPU dialect for Nvidia GPUs)

MLIR

ThomasRaoux Apr '22

Note that the GPU dialect includes both host side as well as device operations. This only consider device side operations.

Goal of the dialect

The GPU dialect 9 is used to target GPUs while being agnostic to APIs and target. Therefore it is designed to abstract hardware specific details.

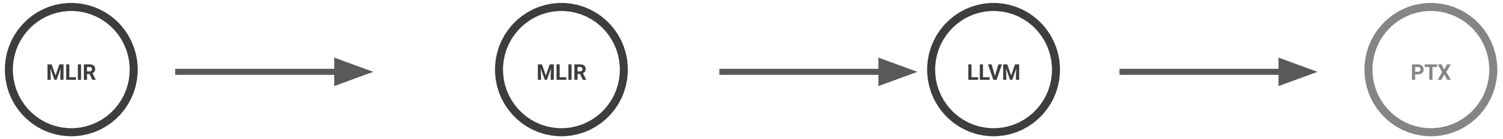
To target cuda/nvidia hardware we go through GPU and Vector dialects and lower to NVVM dialect. This means hardware specific information can only be exposed at the NVVM dialect. In some cases there is a large abstraction gap, making the lowering non trivial and preventing us from doing higher level transformation based on hardware details.

This proposal is for adding a new `Nvidia/PTX` Dialect that would directly model NVVM intrinsics using memref and N-d Vector types before lowering to NVVM/LLVM dialect/types. This is analogue to the existing AMX dialect 15 for Nvidia GPUs.

Google Research

What is NVGPU Dialect?

Bridge between high-level dialects to NVVM/LLVM dialect



NVGPU Dialect

NVVM Dialect

```
%0 = nvgpu.wargroup.mma
  %matrixA, %matrixB, %matrixACC
  : <tensor = memref<128x64xf16,3>>,
  <tensor = memref<64x128xf16,3>>,
  <fragmented = vector<128x128xf32,3>> ->
  <fragmented = vector<128x128xf32,3>>
```

```
// Generated PTX Code
wgmma.fence.sync.aligned;
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f1, %f2,...}, %dA, %dB, p, 1, 1, 0, 1;
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f1, %f2,...}, %dA+2, %dB+128, p, 1, 1, 0, 1;
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f1, %f2,...}, %dA+4, %dB+256, p, 1, 1, 0, 1;
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f1, %f2,...}, %dA+8, %dB+348, p, 1, 1, 0, 1;

wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f500,%f501,...} %dA+512, %dB, p, 1, 1, 0, 1;
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f500,%f501,...} %dA+514, %dB+128, p, 1, 1, 0, 1
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f500,%f501,...} %dA+516, %dB+256, p, 1, 1, 0, 1
wgmma.mma_async.sync.aligned.m64n128k16.f32.f16.f16 {%f500,%f501,...} %dA+518, %dB+348, p, 1, 1, 0, 1
wgmma.commit_group.sync.aligned;
wgmma.wait_group.sync.aligned 1;
```

What is NVGPU Dialect?

Implemented following features to hide architectural details:

- High-level `nvgpu.mbarrier` OPs
- High-level `nvgpu.tma` OPs
- High-level `nvgpu.warpgrop` MMA OPs

Feature: Arrive/Wait Barriers

7 new Ops:

1) **nvgpu.mbarrier.create**
2) nvgpu.mbarrier.init
3) nvgpu.mbarrier.try_wait.parity
4) nvgpu.mbarrier.arrive.expect_tx
5) nvgpu.mbarrier.arrive
6) nvgpu.mbarrier.arrive.nocomplete
7) nvgpu.mbarrier.test.wait

```
// Step 1. Create a barrier group with 3 barriers
%barrier = nvgpu.mbarrier.create -> <num_barriers = 3>
```

Feature:
Arrive/Wait Barriers

7 new Ops:

1) nvgpu.mbarrier.create
2) **nvgpu.mbarrier.init**
3) nvgpu.mbarrier.try_wait.parity
4) nvgpu.mbarrier.arrive.expect_tx
5) nvgpu.mbarrier.arrive
6) nvgpu.mbarrier.arrive.nocomplete
7) nvgpu.mbarrier.test.wait

```
// Step 1. Create a barrier group with 3 barriers
%barrier = nvgpu.mbarrier.create -> <num_barriers = 3>

// Step 2. Initialize barriers in the group
nvgpu.mbarrier.init %barrier[0], 1, predicate = %thread0 : <num_barriers = 3>
nvgpu.mbarrier.init %barrier[1], 1, predicate = %thread0 : <num_barriers = 3>
nvgpu.mbarrier.init %barrier[2], 1, predicate = %thread0 : <num_barriers = 3>
```

Feature: Arrive/Wait Barriers

7 new Ops:

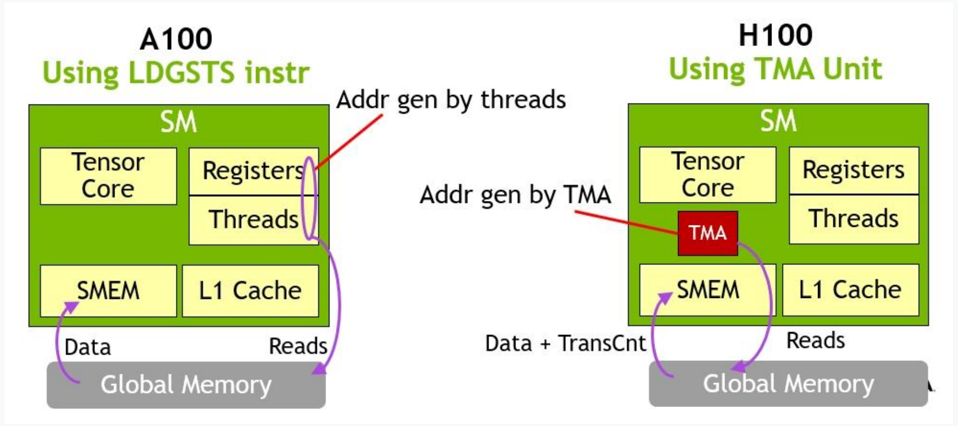
1) nvgpu.mbarrier.create
2) nvgpu.mbarrier.init
3) **nvgpu.mbarrier.try_wait.parity**
4) **nvgpu.mbarrier.arrive.expect_tx**
5) nvgpu.mbarrier.arrive
6) nvgpu.mbarrier.arrive.nocomplete
7) nvgpu.mbarrier.test.wait

```
// Step 1. Create 3 barriers
%barrier = nvgpu.mbarrier.create -> <num_barriers = 3>

// Step 2. Initialize barriers in the group
nvgpu.mbarrier.init %barrier[0], 1, predicate = %thread0 : <num_barriers = 3>
nvgpu.mbarrier.init %barrier[1], 1, predicate = %thread0 : <num_barriers = 3>
nvgpu.mbarrier.init %barrier[2], 1, predicate = %thread0 : <num_barriers = 3>

// Step 3. Use the barrier group in a loop
scf.for %i = %c0 to %N step %c1 {
   %barId = arith.remui %i, 3 : index
    nvgpu.mbarrier.try_wait.parity %barrier[%barId], %c0, %ticks : <num_barriers = 3>
    // do work
    nvgpu.mbarrier.arrive.expect_tx %barrier[%barId], %txcount, predicate = %thread0
                                    : <num_barriers = 3>
}
```

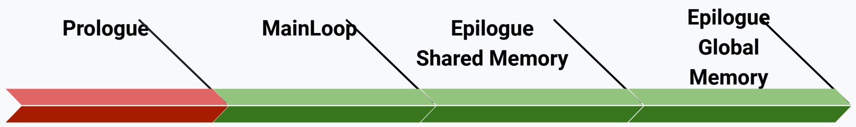
Feature:
`Tensor Memory Accelerator (TMA)`

TMA has two parts

1) Host creates descriptor `cuTensorMapEncodeTiled`
2) GPU submit request to TMA



Feature: Tensor Memory Accelerator (TMA)



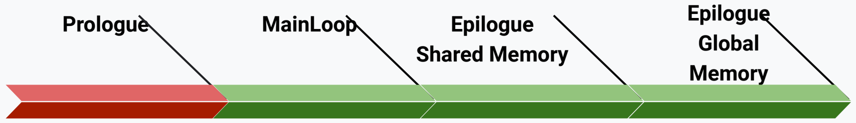
Google Research

3 new Ops for TMA:

1) **nvgpu.tma.create.descriptor**
2) nvgpu.tma.prefetch.descriptor
3) nvgpu.tma.async.load

```
!dtype = !nvgpu.tensormap.descriptor<tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b, l2promo = l2promo_128b,
                                     oob = zero,  interleave = none>
func @main() {
   // Step 1. [HOST] Create TensorMap descriptor (calls cuTensorMapEncodeTiled from cuda driver)
   %desc = nvgpu.tma.create.descriptor %memref box[%c128, %c64] : memref<*xf16> -> !dtype

   gpu.launch … {
     // Step 2. [GPU] Create and initialize mbarrier
     %mbar = nvgpu.mbarrier.create -> !btype
     nvgpu.mbarrier.init %mbar, %c1, predicate = %cnd : !btype

     // Step 3. [GPU] Prefetch TensorMap Descriptor to L1 cache
     nvgpu.tma.prefetch.descriptor %desc, predicate = %cnd : !dtype

     // Step 4. [GPU] Request to load data tile 128x64 using TMA
     nvgpu.tma.async.load %desc[%c0, %c0], %mbar to %buffer, predicate = %cnd : !dtype, !btype

     // Step 5. [GPU] Mbarrier is assigned to wait 16384 byte data from TMA
     %txcount = arith.constant 16384
     nvgpu.mbarrier.arrive.expect_tx %mbar, %txcount, predicate = %cnd : !btype

     // Step 6. [GPU] All Threads wait the mbarrier (until the data ready)
     nvgpu.mbarrier.try_wait.parity %mbar, %phase, %ticks : !btype
     … use the data …
} }
```

Calls *cuTensorMapEncodeTiled*



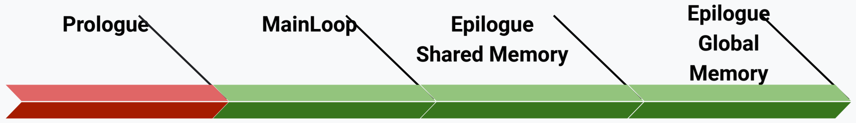
Google Research

Feature: Tensor Memory Accelerator (TMA)

3 new Ops for TMA:

1) nvgpu.tma.create.descriptor
2) **nvgpu.tma.prefetch.descriptor**
3) nvgpu.tma.async.load

```
!dtype = !nvgpu.tensormap.descriptor<tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b, l2promo = l2promo_128b,
                                     oob = zero,  interleave = none>
func @main() {
   // Step 1. [HOST] Create TensorMap descriptor (calls cuTensorMapEncodeTiled from cuda driver)
   %desc = nvgpu.tma.create.descriptor %memref box[%c128, %c64] : memref<*xf16> -> !dtype

   gpu.launch … {
     // Step 2. [GPU] Create and initialize mbarrier
     %mbar = nvgpu.mbarrier.create -> !btype
     nvgpu.mbarrier.init %mbar, %c1, predicate = %cnd : !btype

     // Step 3. [GPU] Prefetch TensorMap Descriptor to L1 cache
     nvgpu.tma.prefetch.descriptor %desc, predicate = %cnd : !dtype

     // Step 4. [GPU] Request to load data tile 128x64 using TMA
     nvgpu.tma.async.load %desc[%c0, %c0], %mbar to %buffer, predicate = %cnd : !dtype, !btype

     // Step 5. [GPU] Mbarrier is assigned to wait 16384 byte data from TMA
     %txcount = arith.constant 16384
     nvgpu.mbarrier.arrive.expect_tx %mbar, %txcount, predicate = %cnd : !btype

     // Step 6. [GPU] All Threads wait the mbarrier (until the data ready)
     nvgpu.mbarrier.try_wait.parity %mbar, %phase, %ticks : !btype
     … use the data …
} }
```



Google Research

Feature: Tensor Memory Accelerator (TMA)

3 new Ops for TMA:

1) `nvgpu.tma.create.descriptor`
2) `nvgpu.tma.prefetch.descriptor`
3) `nvgpu.tma.async.load`

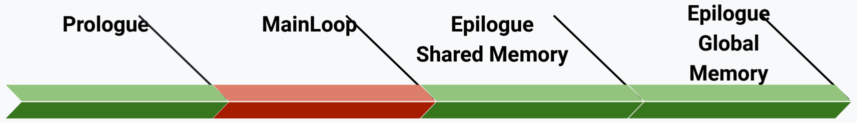
```
!dtype = !nvgpu.tensormap.descriptor<tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b, l2promo = l2promo_128b,
                                     oob = zero,  interleave = none>
func @main() {
   // Step 1. [HOST] Create TensorMap descriptor (calls cuTensorMapEncodeTiled from cuda driver)
   %desc = nvgpu.tma.create.descriptor %memref box[%c128, %c64] : memref<*xf16> -> !dtype

   gpu.launch … {
     // Step 2. [GPU] Create and initialize mbarrier
     %mbar = nvgpu.mbarrier.create -> !btype
     nvgpu.mbarrier.init %mbar, %c1, predicate = %cnd : !btype

     // Step 3. [GPU] Prefetch TensorMap Descriptor to L1 cache
     nvgpu.tma.prefetch.descriptor %desc, predicate = %cnd : !dtype

     // Step 4. [GPU] Request to load data tile 128x64 using TMA
     nvgpu.tma.async.load %desc[%c0, %c0], %mbar to %buffer, predicate = %cnd : !dtype, !btype

     // Step 5. [GPU] Mbarrier is assigned to wait 16384 byte data from TMA
     %txcount = arith.constant 16384
     nvgpu.mbarrier.arrive.expect_tx %mbar, %txcount, predicate = %cnd : !btype

     // Step 6. [GPU] All Threads wait the mbarrier (until the data ready)
     nvgpu.mbarrier.try_wait.parity %mbar, %phase, %ticks : !btype
     … use the data …
} }
```

Feature: Tensor Memory Accelerator (TMA)

Google Research

```
!dtype = !nvgpu.tensormap.descriptor<tensor = memref<128x64xf16, 3>,
       swizzle = swizzle_128b, l2promo = l2promo_128b, ob = zero,  interleave = none>
func @main() {
   // Step 1. [HOST] Create TensorMap descriptor. It calls cuTensorMapEncodeTiled
   %dsc = nvgpu.tma.create.descriptor %mem box[%c128, %c64] : memref<*xf16> -> !dtype

   gpu.launch … {
     // Step 2. [GPU] Create and initialize mbarrier
     %b = nvgpu.mbarrier.create -> !btype
     nvgpu.mbarrier.init %b, %c1, predicate = %cnd : !btype

     // Step 3. [GPU] Prefetch TensorMap Descriptor to L1 cache
     nvgpu.tma.prefetch.descriptor %dsc, predicate = %cnd : !dtype

     // Step 4. [GPU] Request to load data tile 128x64 using TMA
     nvgpu.tma.async.load %desc[%c0, %c0], %b to %shmem, predicate = %p

     // Step 5. [GPU] Mbarrier is assigned to wait 16384 byte data from TMA
     %txcount = arith.constant 16384
     nvgpu.mbarrier.arrive.expect_tx %b, %txcount, predicate = %p : !btype

     // Step 6. [GPU] All Threads wait the mbarrier (until the data ready)
     nvgpu.mbarrier.try_wait.parity %b, %phase, %ticks : !btype
     … use the data …
```

```
.visible .entry main_kernel(.param .u64 main_kernel_param_0)
{
 .reg .pred      %p<5>;
 .reg .b32       %r<13>;
 .reg .b64       %rd<3>;
 .shared .align 8 .b8 __mbarrier[8];
 .shared .align 2 .b8 bufferLhsGlobal[16384];
 ld.param.u64    %rd1, [main_kernel_param_0];
 mov.u32         %r12, %tid.x;
 setp.eq.s32     %p1, %r12, 0;
 mov.u32         %r4, __mbarrier;
 mov.b32         %r2, 1;
 @%p1 mbarrier.init.shared.b64 [%r4], %r2;
 @%p1 prefetch.tensormap [%rd1];
 mov.u32         %r3, bufferLhsGlobal;
 mov.b32         %r5, 0;

 @%p1 cp.async.bulk.tensor.2d.shared::cluster.global.mbarrier::complete_tx::bytes
                                             [%r3], [%rd1, {%r5, %r5} ], [%r4];

 @%p1 mbarrier.arrive.expect_tx.shared.b64 _, [%r4], 16384;
 {
         .reg .pred       P1;
         LAB_WAIT:
         mbarrier.try_wait.parity.shared.b64 P1, [%r4], %r5, 10000000;
         @P1 bra.uni DONE;
         bra.uni     LAB_WAIT;
         DONE:
 }
 ret; }
```



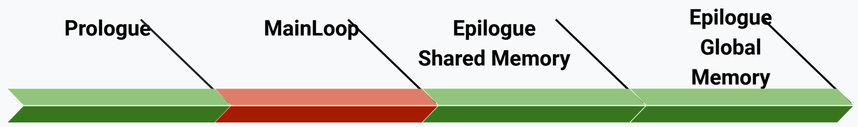
Google Research

Feature: Warp-group MMA

4 new Ops:

1) nvgpu.wargroup.mma.init.accumulator
2) nvgpu.wargroup.generate.descriptor
3) nvgpu.wargroup.mma
4) nvgpu.wargroup.mma.store

```
// Step 1. Intialize input matrix
%m = nvgpu.wargroup.mma.init.accumulator -> !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>

%mD = scf.for %i = %ic0 to %N step %ic1 iter_args(%mC = %m) -> (!nvgpu.wargroup.accumulator<fragmented =
vector<128x128xf32>>) {
  // Step 3. Wait data from TMA …

  // Step 4. Generate WGMMA Descriptor
  %mA = nvgpu.wgmma.generate.descriptor %lhs, %descA : memref<64x128xf16,3>, !lhsTensorMap
                                                        -> !nvgpu.wgmma.descriptor<tensor=memref<128x64xf16, 3>>
  %mB = nvgpu.wgmma.generate.descriptor %rhs, %descB : memref<1x128x64xf16,3>, !rhsTensorMap
                                                        -> !nvgpu.wgmma.descriptor<tensor=memref<64x128xf16, 3>>

  // Step 5. Perform MMA 128x128x64, 8 x wgmma [shape 64 128 16]
  %mD = nvgpu.wargroup.mma %mA, %mB, %mC : <tensor = memref<128x64xf16, 3>>, <tensor = memref<64x128xf16, 3>>
                                          -> <fragmented = vector<128x128xf32>>
  // Step 6. Keep TMA loading …

  // Step 7. Yield matrixD
  scf.yield %mD : !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>
}

// Step 8. Epilogue registers -> shared memory
nvgpu.wargroup.mma.store %mD to %shmem : <fragmented = vector<128x128xf32>> into memref<128x128xf32,3>
```

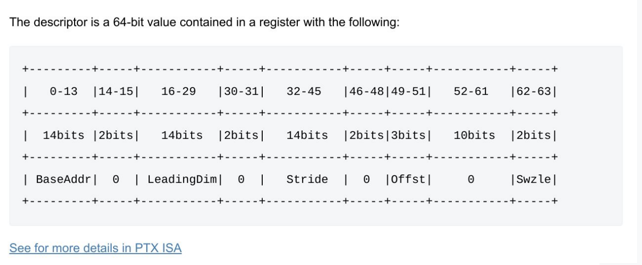


Google Research

Feature:
`Warp-group MMA`

4 new Ops:

1) `nvgpu.wargroup.mma.init.accumulator`
2) [`nvgpu.wargroup.generate.descriptor`]
3) `nvgpu.wargroup.mma`
4) `nvgpu.wargroup.mma.store`

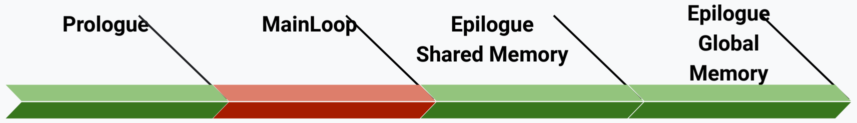


The descriptor is a 64-bit value contained in a register with the following:

```
+---------+-----+-----------+-----+-----------+-----+-----+-----------+-----+
|   0-13  |14-15|   16-29   |30-31|   32-45   |46-48|49-51|   52-61   |62-63|
+---------+-----+-----------+-----+-----------+-----+-----+-----------+-----+
|  14bits |2bits|   14bits  |2bits|   14bits  |2bits|3bits|   10bits  |2bits|
+---------+-----+-----------+-----+-----------+-----+-----+-----------+-----+
| BaseAddr|  0  | LeadingDim|  0  |   Stride  |  0  |Offst|     0     |Swzle|
+---------+-----+-----------+-----+-----------+-----+-----+-----------+-----+
```

See for more details in PTX ISA

```
// Step 1. Intialize input matrix
%m = nvgpu.wargroup.mma.init.accumulator -> !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>

%mD = scf.for %i = %ic0 to %N step %ic1 iter_args(%mC = %m) -> (!nvgpu.wargroup.accumulator<fragmented =
vector<128x128xf32>>) {
  // Step 3. Wait data from TMA …

  // Step 4. Generate WGMMA Descriptor
  %mA = nvgpu.wgmma.generate.descriptor %lhs, %descA : memref<64x128xf16,3>, !lhsTensorMap
                                                      -> !nvgpu.wgmma.descriptor<tensor=memref<128x64xf16, 3>>
  %mB = nvgpu.wgmma.generate.descriptor %rhs, %descB : memref<1x128x64xf16,3>, !rhsTensorMap
                                                      -> !nvgpu.wgmma.descriptor<tensor=memref<64x128xf16, 3>>

  // Step 5. Perform MMA 128x128x64, 8 x wgmma [shape 64 128 16]
  %mD = nvgpu.wargroup.mma %mA, %mB, %mC : <tensor = memref<128x64xf16, 3>>, <tensor = memref<64x128xf16, 3>>
                                        -> <fragmented = vector<128x128xf32>>
  // Step 6. Keep TMA loading …

  // Step 7. Yield matrixD
  scf.yield %mD : !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>
}

// Step 8. Epilogue registers -> shared memory
nvgpu.wargroup.mma.store %mD to %shmem : <fragmented = vector<128x128xf32>> into memref<128x128xf32,3>
```

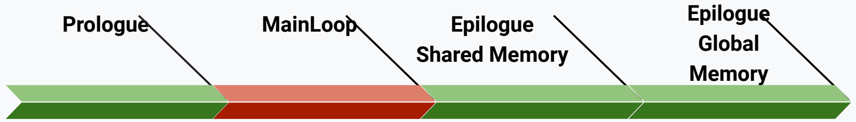


Feature: Warp-group MMA

4 new Ops:

1) nvgpu.wargroup.mma.init.accumulator
2) nvgpu.wargroup.generate.descriptor
3) **nvgpu.wargroup.mma**
4) nvgpu.wargroup.mma.store

```
// Step 1. Intialize input matrix
%m = nvgpu.wargroup.mma.init.accumulator -> !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>

%mD = scf.for %i = %ic0 to %N step %ic1 iter_args(%mC = %m) -> (!nvgpu.wargroup.accumulator<fragmented =
vector<128x128xf32>>) {
  // Step 3. Wait data from TMA …

  // Step 4. Generate WGMMA Descriptor
  %mA = nvgpu.wgmma.generate.descriptor %lhs, %descA : memref<64x128xf16,3>, !lhsTensorMap
                                                      -> !nvgpu.wgmma.descriptor<tensor=memref<128x64xf16, 3>>
  %mB = nvgpu.wgmma.generate.descriptor %rhs, %descB : memref<1x128x64xf16,3>, !rhsTensorMap
                                                      -> !nvgpu.wgmma.descriptor<tensor=memref<64x128xf16, 3>>

  // Step 5. Perform MMA 128x128x64, 8 x wgmma [shape 64 128 16]
  %mD = nvgpu.wargroup.mma %mA, %mB, %mC : <tensor = memref<128x64xf16, 3>>, <tensor = memref<64x128xf16, 3>>
                                          -> <fragmented = vector<128x128xf32>>
  // Step 6. Keep TMA loading …

  // Step 7. Yield matrixD
  scf.yield %mD : !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>
}

// Step 8. Epilogue registers -> shared memory
nvgpu.wargroup.mma.store %mD to %shmem : <fragmented = vector<128x128xf32>> into memref<128x128xf32,3>
```



Google Research

Feature: Warp-group MMA

4 new Ops:

1) nvgpu.wargroup.mma.init.accumulator
2) nvgpu.wargroup.generate.descriptor
3) nvgpu.wargroup.mma
4) **nvgpu.wargroup.mma.store**

```
// Step 1. Intialize input matrix
%m = nvgpu.wargroup.mma.init.accumulator -> !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>

%mD = scf.for %i = %ic0 to %N step %ic1 iter_args(%mC = %m) -> (!nvgpu.wargroup.accumulator<fragmented =
vector<128x128xf32>>) {
  // Step 3. Wait data from TMA …

  // Step 4. Generate WGMMA Descriptor
  %mA = nvgpu.wgmma.generate.descriptor %lhs, %descA : memref<64x128xf16,3>, !lhsTensorMap
                                                        -> !nvgpu.wgmma.descriptor<tensor=memref<128x64xf16, 3>>
  %mB = nvgpu.wgmma.generate.descriptor %rhs, %descB : memref<1x128x64xf16,3>, !rhsTensorMap
                                                        -> !nvgpu.wgmma.descriptor<tensor=memref<64x128xf16, 3>>

  // Step 5. Perform MMA 128x128x64, 8 x wgmma [shape 64 128 16]
  %mD = nvgpu.wargroup.mma %mA, %mB, %mC : <tensor = memref<128x64xf16, 3>>, <tensor = memref<64x128xf16, 3>>
                                          -> <fragmented = vector<128x128xf32>>
  // Step 6. Keep TMA loading …

  // Step 7. Yield matrixD
  scf.yield %mD : !nvgpu.wargroup.accumulator<fragmented = vector<128x128xf32>>
}

// Step 8. Epilogue registers -> shared memory
nvgpu.wargroup.mma.store %mD to %shmem : <fragmented = vector<128x128xf32>> into memref<128x128xf32,3>
```

Benchmarks

Nvidia H100 PCIe

GEMM 4096x4096x4096 [f32 += f16 * f16]

Google Research

Baseline

- TMA Load (`cp.async.bulk.tensor`)
- 7 software pipeline stages (`mbarriers`)
- Warp-group level GEMM (`wgmma.mma_async`)
- 128B swizzling

Opt 1: 32 bit address space for shared memory (*-nvptx-short-ptr*)

Opt 2: L2 Cache hint for TMA (`cp.async.bulk.tensor…L2::cache_hint`)

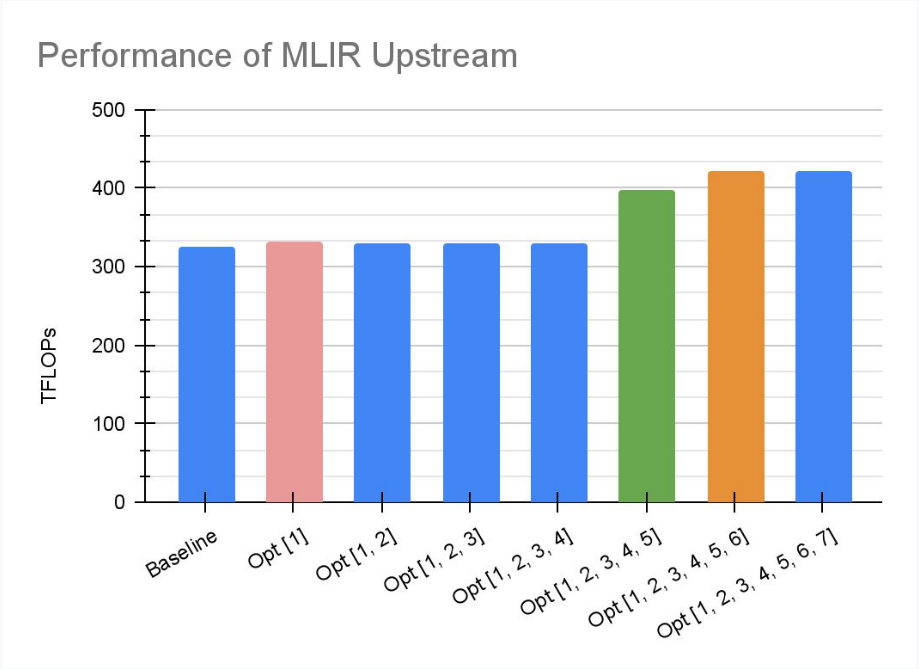
Opt 3: Better register allocations (`maxntid`)

Opt 4: `llvm.noalias` on kernel parameters

Opt 5: Use predicate

Opt 6: Interleaved epilogue

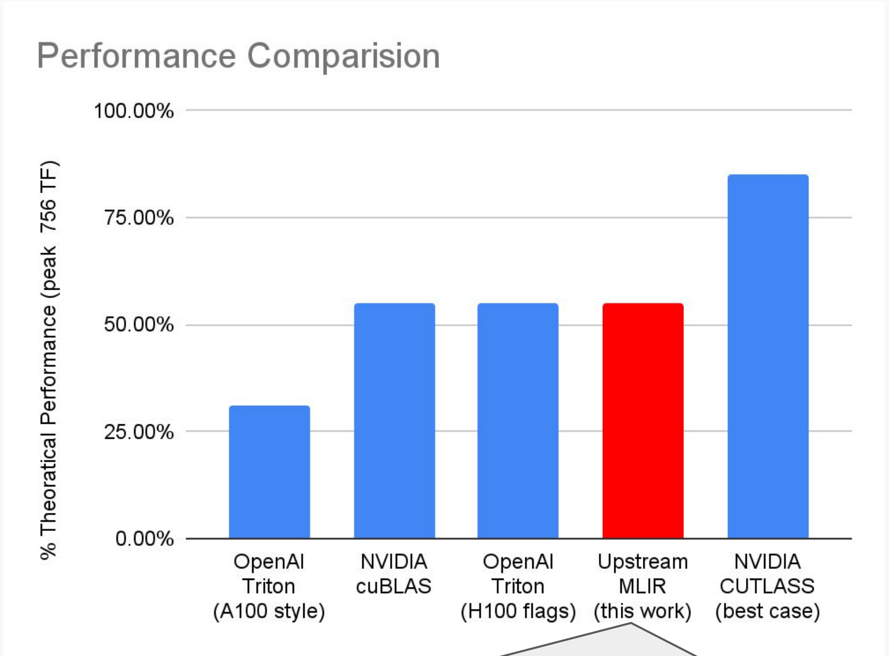
Opt 7: Locate TMA descriptors to L1 (`prefetch.tensormap`)



GEMM 4096x4096x4096 [f32 += f16 * f16]

OpenAI Triton:	A100 style code generation
NVIDIA cuBLAS:	NVIDIA's Closed source state-of-art GEMM library
OpenAI Triton(H100):	Flags for TMA + wgmma
Upstream MLIR:	This work
NVIDIA CUTLASS*:	NVIDIA's open-source state-of-art GEMM library

* selected the best performing GEMM algorithm (pinpong, epi_tma)



Does MLIR generates slow/bad PTX:

- No

Why do we off from the peak?

- Need to different GEMM algorithm for this shape

GEMM Algorithms for H100 in CUTLASS

GEMM Algorithm	Required features
KernelMultistage	wgmma
KernelTma	TMA, mbarriers, wgmma $\rightarrow$ MLIR implemented
KernelTmaWarpSpecialized	TMA, mbarriers, wgmma, warp sp
KernelTmaWarpSpecializedCooperative	TMA, mbarriers, wgmma, warp sp, cta cluster
KernelTmaWarpSpecializedPingpong	TMA, mbarriers, wgmma, warp sp, cta cluster

Future Plan

- Study and implement these GEMM algorithms in MLIR.
- Profile their performance with/without fused kernels.

Google Research

Target H100 in MLIR

Focusing on Tensor Core of H100

🎯 Code generation for Tensor Core of H100

MLIR Compiler

🤝 MLIR is community-driven, and vendors have a history of active contributions.

Implement in NVGPU and NVVM Dialects

🔀 These dialects serve as building blocks across multiple compilers (e.g., Triton, IREE, etc.)

Upstream

🚀 All the work presented in this slide has been upstreamed to MLIR