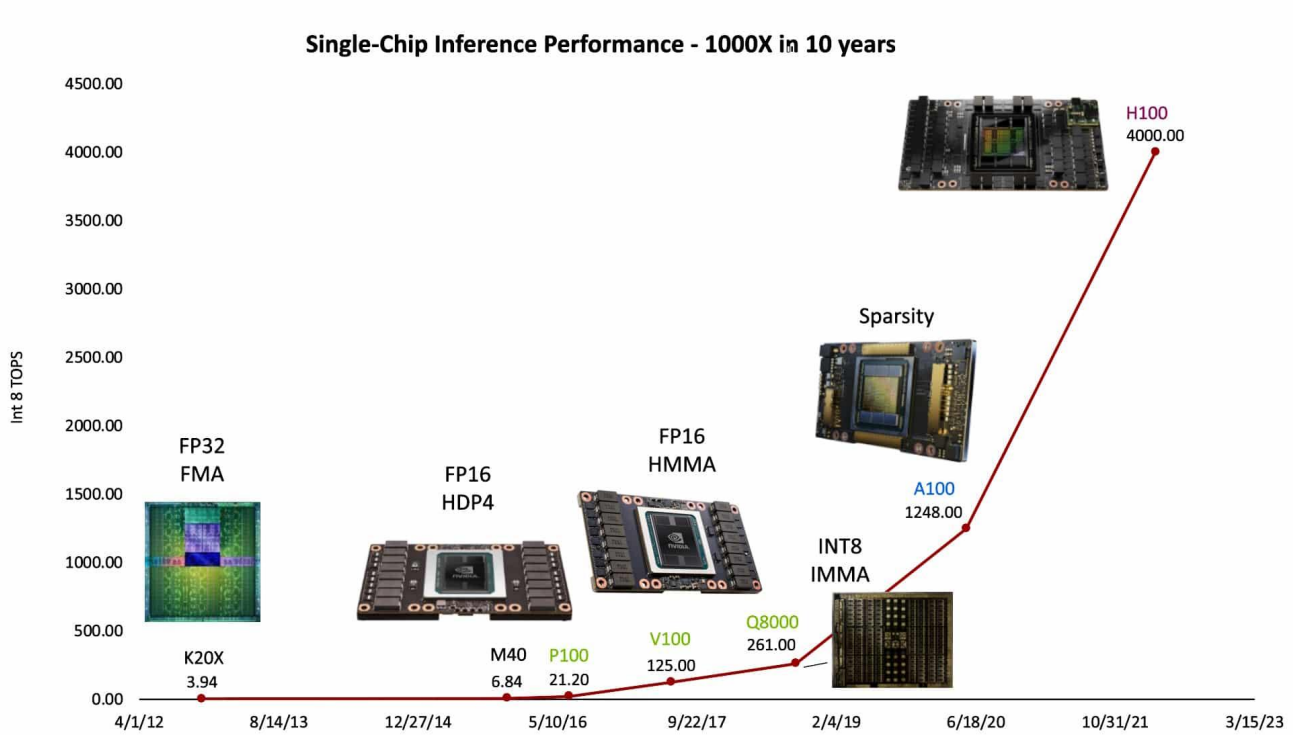
Targeting NVIDIA Hopper in MLIR

Guray Ozen

2th March 24 -
8th LLVM Performance Workshop at CGO 24

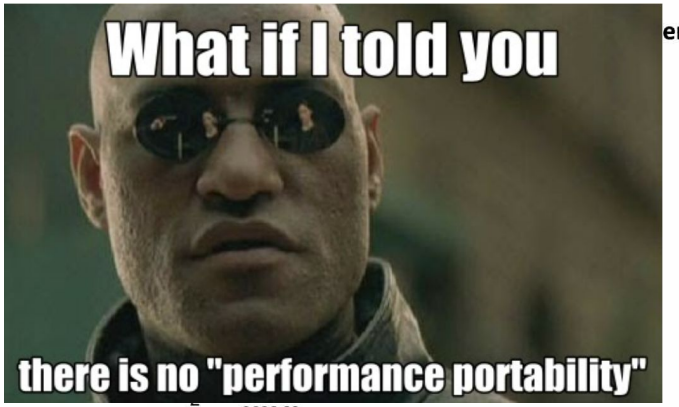
Google Research

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

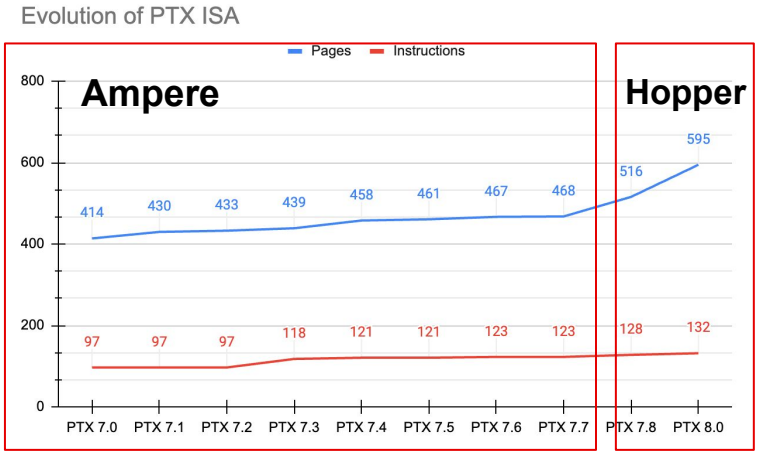
Asynchronous Barriers
- Helps waiting TMA asynchronously

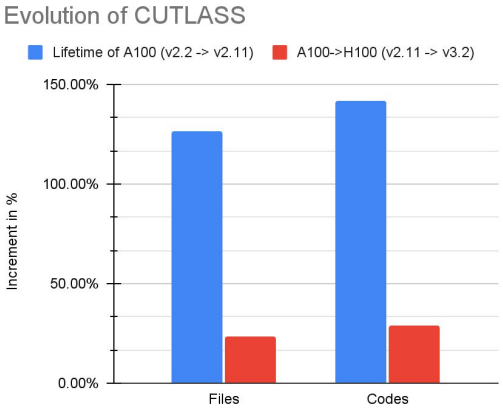
Evolution in Software:
PTX[1] & CUTLASS[2]

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

Motivation

Exploit NVIDIA Hopper architecture using MLIR

Target NVIDIA Hopper in MLIR

MLIR Compiler

🤝 Community-driven with significant vendor contributions.

Focusing on Hopper Architecture

🎯 Advanced code generation for Tensor Core, TMA, etc.

NVGPU and NVVM Dialects

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

Performance

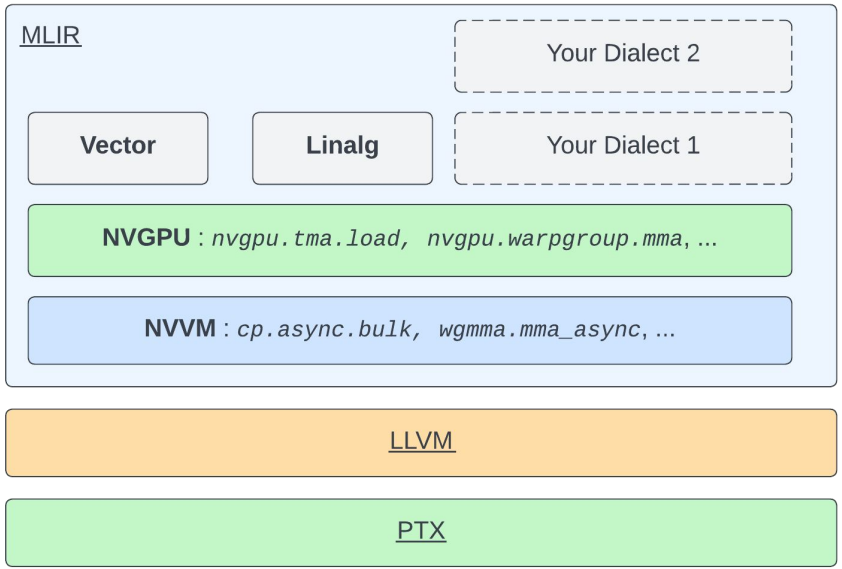
🚀 MLIR has close performance to cuBLAS

Upstream

✅ All the work presented is fully upstreamed to MLIR

MLIR Upstream Dialect Layers

Improved GPU, NVGPU, and NVVM Dialects



NVGPU Dialect

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

NVVM Dialect

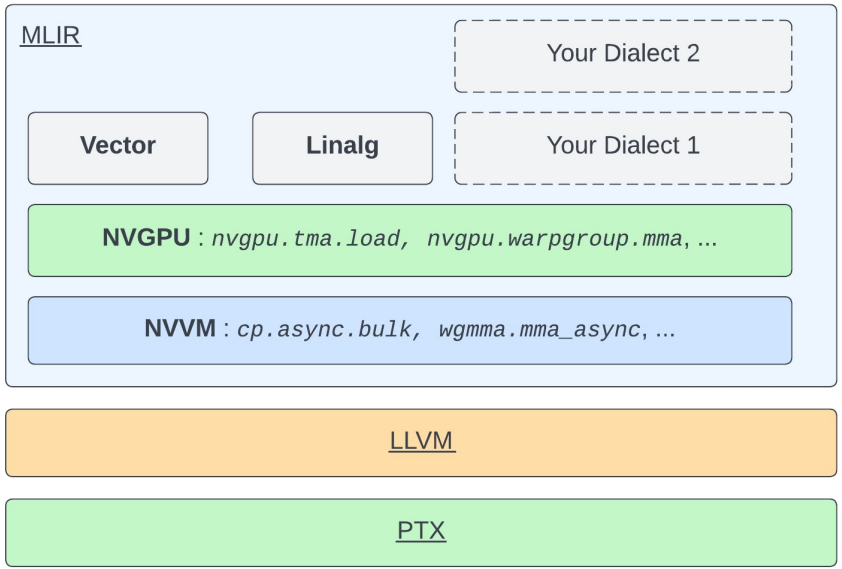
- Low level operations (closer to PTX)
- NVVM → PTX or LLVM intrinsic

GPU Dialect

- Kernel launch, Cluster launch
- Driver communication

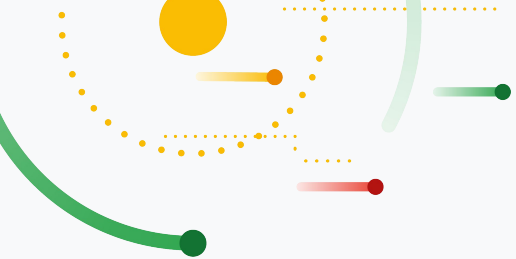
MLIR Upstream Dialect Layers

Connect Your Dialect → NVGPU



One can lower other dialects into NVGPU

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



Google Research

NVGPU & NVVM Dialects

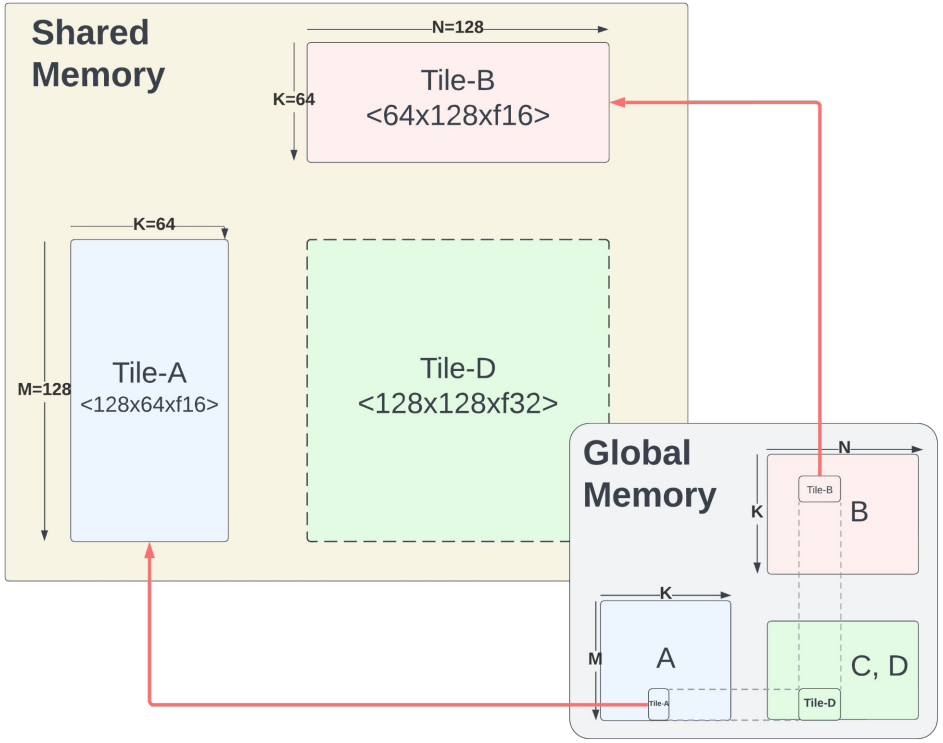
Tensor Core

Tensor Core

- Warp group wide (128 threads)
  - `wgmma.mma_async`
- Work asynchronously
- Matrix-A and B can be in shared memory
  - **Tile-D += Tile-A * Tile-B**
- Supported Shapes:
  - M=64, N = [8, 256], K=[8, 16, 32, 256]

Memory Locations

- A, B, D @ global memory
- Tile-A & Tile-B @ shared memory
- Tile-D @ registers | shared memory



Tensor Core: `nvgpu Dialect`

New Abstractions : Expected to run by a Warpgroup (128 threads)

`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 (via `wgmma.mma_async` PTX)

`nvgpu.warpgroup.mma.store`

- Store fragmented registers to shared or global memory (via vectorized copy or `stmatrix` PTX)

Tensor Core Example: `nvgpu Dialect`

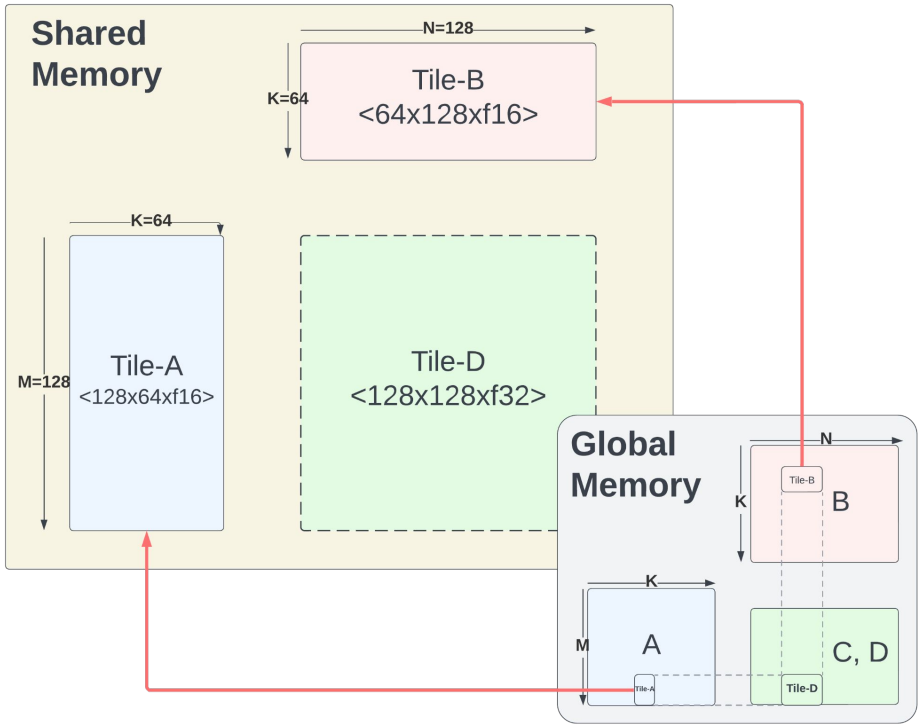
Shape = 128x128x64

Let's take this sequential GEMM

Run on Tensor Core using **`nvgpu dialect`**

By Warpgroup (128 threads)

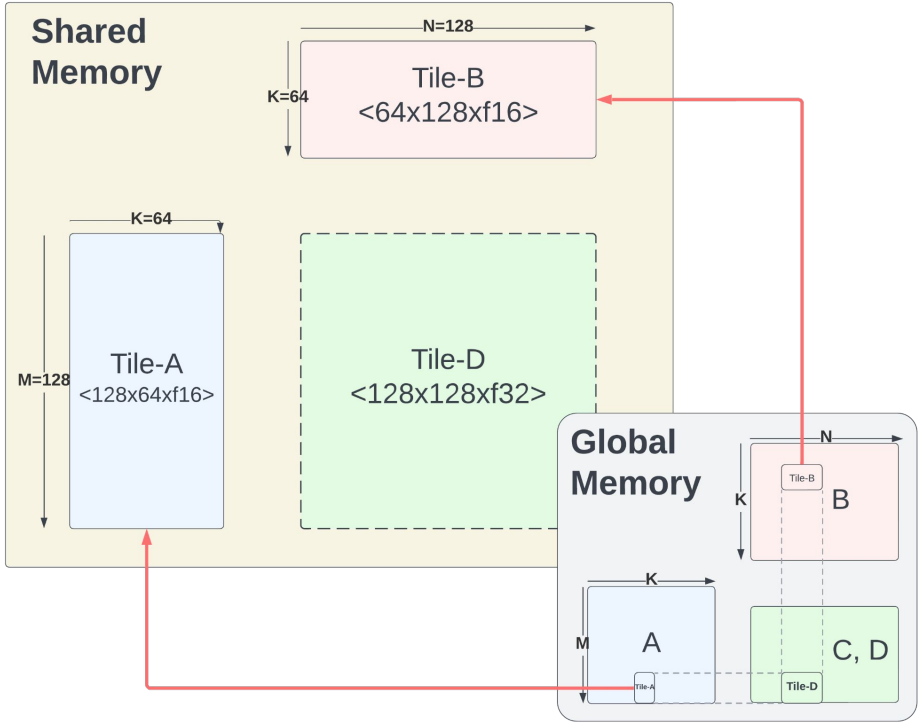
```
Tile-D += Tile-A * Tile-B
for(i=0; i < 128; ++i)
 for(j=0; i < 128; ++j)
   for(k=0; i < 64; ++k)
     FMA(...)
```



Tensor Core Example: nvgpu Dialect

Shape = 128x128x64

```
%regC = nvgpu.warpgroup.mma.init.accumulator
                -> !<fragmented = vector<128x128xf32>>


%dA = nvgpu.warpgroup.generate.descriptor %tileA : ...
                -> !<tensor=memref<128x64xf16, 3>>
%dB = nvgpu.warpgroup.generate.descriptor %tileB : ...
                -> !<tensor=memref<64x128xf32, 3>>


%regD = nvgpu.warpgroup.mma %dA, %dB, %regC :
                            <tensor = memref<128x64xf16, 3>>,
                            <tensor = memref<64x128xf32, 3>>
                            ->
                            <fragmented = vector<128x128xf32>>


nvgpu.warpgroup.mma.store %regD to %tileD
```



Tensor Core Example: nvgpu Dialect

Shape = 128x128x64

```
%regC = nvgpu.warpgroup.mma.init.accumulator
                -> !<fragmented = vector<128x128xf32>>


%dA = nvgpu.warpgroup.generate.descriptor %tileA : ...
                -> !<tensor=memref<128x64xf16, 3>>
%dB = nvgpu.warpgroup.generate.descriptor %tileB : ...
                -> !<tensor=memref<64x128xf32, 3>>


%regD = nvgpu.warpgroup.mma %dA, %dB, %regC :
                            <tensor = memref<128x64xf16, 3>>,
                            <tensor = memref<64x128xf32, 3>>
                            ->
                            <fragmented = vector<128x128xf32>>


nvgpu.warpgroup.mma.store %regD to %tileD
```

Create and Initialize the accumulator registers

Tensor Core Example: nvgpu Dialect

Shape = 128x128x64

```
%regC = nvgpu.warpgroup.mma.init.accumulator
                -> !<fragmented = vector<128x128xf32>>
```

Create and Initialize the accumulator registers

```
%dA = nvgpu.warpgroup.generate.descriptor %tileA : ...
                -> !<tensor=memref<128x64xf16, 3>>
%dB = nvgpu.warpgroup.generate.descriptor %tileB : ...
                -> !<tensor=memref<64x128xf32, 3>>
```

Generates 64-bit wgmma descriptor that keeps:

1. **Start Address,**
2. **leading dimension,**
3. **stride,**
4. **swizzle**

```
%regD = nvgpu.warpgroup.mma %dA, %dB, %regC :
                            <tensor = memref<128x64xf16, 3>>,
                            <tensor = memref<64x128xf32, 3>>
                            ->
                            <fragmented = vector<128x128xf32>>
```

```
nvgpu.warpgroup.mma.store %regD to %tileD
```

Tensor Core Example: nvgpu Dialect

Shape = 128x128x64

```
%regC = nvgpu.warpgroup.mma.init.accumulator
                -> !<fragmented = vector<128x128xf32>>
```

Create and Initialize the accumulator registers

```
%dA = nvgpu.warpgroup.generate.descriptor %tileA : ...
                -> !<tensor=memref<128x64xf16, 3>>
%dB = nvgpu.warpgroup.generate.descriptor %tileB : ...
                -> !<tensor=memref<64x128xf32, 3>>
```

Generates 64-bit wgmma descriptor that keeps:

1. Start Address,
2. leading dimension,
3. stride,
4. swizzle

```
%regD = nvgpu.warpgroup.mma %dA, %dB, %regC :
                            <tensor = memref<128x64xf16, 3>>,
                            <tensor = memref<64x128xf32, 3>>
                            ->
                            <fragmented = vector<128x128xf32>>
```

Use Tensor Core

```
nvgpu.warpgroup.mma.store %regD to %tileD
```

Tensor Core Example: nvgpu Dialect

Shape = 128x128x64

```
%regC = nvgpu.warpgroup.mma.init.accumulator
                -> !<fragmented = vector<128x128xf32>>
```

Create and Initialize the accumulator registers

```
%dA = nvgpu.warpgroup.generate.descriptor %tileA : ...
                -> !<tensor=memref<128x64xf16, 3>>
%dB = nvgpu.warpgroup.generate.descriptor %tileB : ...
                -> !<tensor=memref<64x128xf32, 3>>
```

Generates 64-bit wgmma descriptor that keeps:

1. Start Address,
2. leading dimension,
3. stride,
4. swizzle

```
%regD = nvgpu.warpgroup.mma %dA, %dB, %regC :
                            <tensor = memref<128x64xf16, 3>>,
                            <tensor = memref<64x128xf32, 3>>
                            ->
                            <fragmented = vector<128x128xf32>>
```

Use Tensor Core

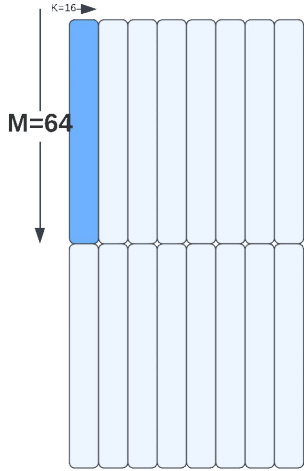
```
nvgpu.warpgroup.mma.store %regD to %tileD
```

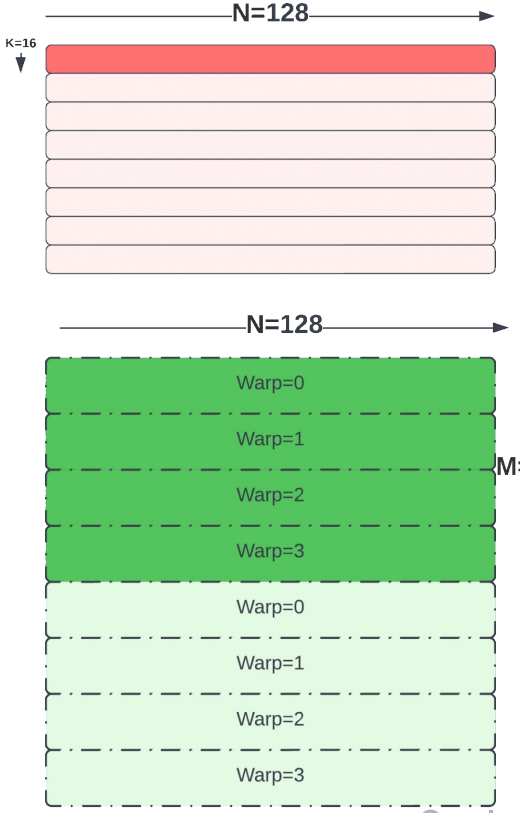
Store fragmented registers

Lowering nvgpu.warpgroup.mma → nvvm.*

NVGPU Shape = 128x128x64 (*nvgpu.wargroup.mma*)
PTX Shape = 64x128x16 (*nvvm.wgmma.mma_async*)

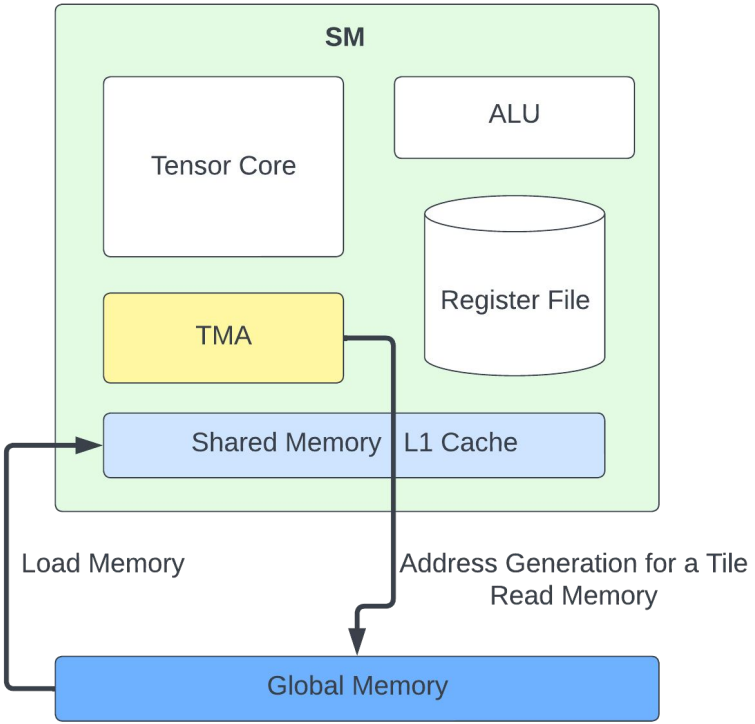
```
%r = 0 : !llvm.struct<(...)>

// 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
```





TMA: Tensor Memory Accelerator

- Loads 1D-5D tile global ⟶ shared memory (vice versa)
- Asynchronous
- Do not waste registers
- Does address calculation
- Used with Asynchronous Transaction Barriers



TMA: `nvgpu dialect`

Abstractions for TMA `(nvgpu.tma.*)`

`nvgpu.tma.create.descriptor`

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

`nvgpu.tma.async.load` *(Lowered:* ***`nvvm.cp.async.bulk.tensor.shared.cluster.global`****)*

- Automatically calculates Tma dimension (1D … 5D)

`nvgpu.tma.prefetch.descriptor` *(Lowered:* ***`nvvm.prefetch.tensor`****)*

- Prefetch tensor descriptor to L1

TMA: nvgpu dialect

Abstractions for Asynchronous Barriers (nvgpu.mbarrier.*)

nvgpu.mbarrier.create

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

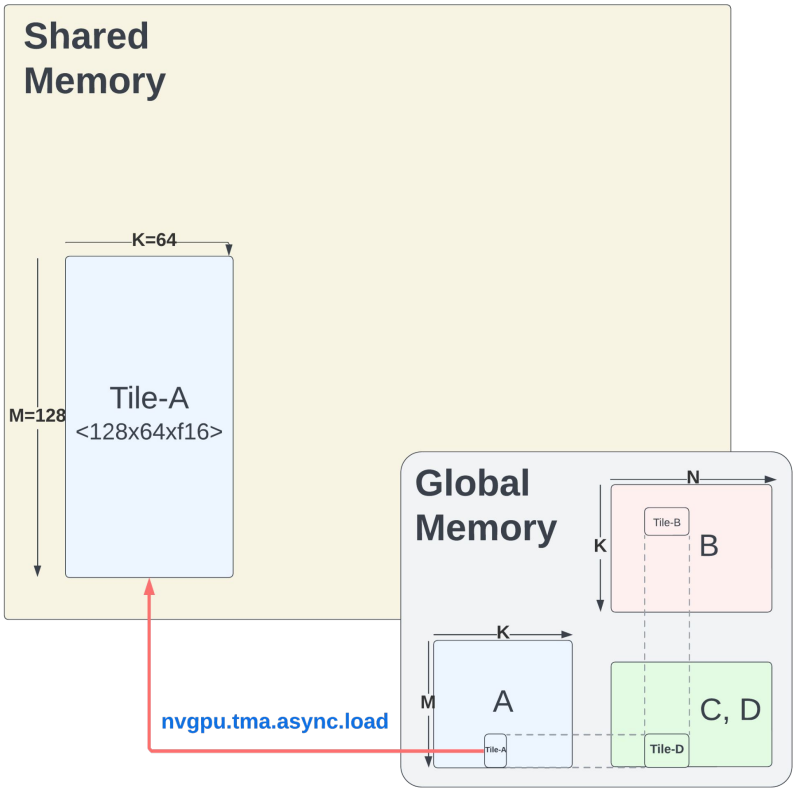
nvgpu.mbarrier.init | expect_tx | try_wait | test_wait | ...

- Access with SSA value. Ideal for handling multiple barriers within a loop
  - `nvgpu.mbarrier.init %mbarGroup[%mbar_id]`
- Predicated
  - `nvgpu.mbarrier.expect_tx  %mbarGroup[%mbar_id] predicate = %tidx0`

TMA Example: nvgpu dialect

Shape Tile-A: 128x64 from Global ⟶ Shared Memory

```
!descType = !nvgpu.tensormap.descriptor
        <tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b,
         l2promo = l2promo_128b, oob = zero,  interleave = none>

func @main() {

  %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType

  gpu.launch ... {

    %mbar = nvgpu.mbarrier.create -> <num_barriers = 1>
    nvgpu.mbarrier.init %mbar[0], %c1, predicate = %tidx0


    scf.if(%tidx0 == 0) {
      nvgpu.tma.async.load %tmaDesc[i, j], %mbar[0] to %tileA : !descType
      nvgpu.mbarrier.arrive.expect_tx %mbar[0], 16384
    }


    nvgpu.mbarrier.try_wait.parity %mbar[0], %phase, %ticks
} }
```



TMA Example: nvgpu dialect

Shape Tile-A: 128x64 from Global → Shared Memory

```
!descType = !nvgpu.tensormap.descriptor
        <tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b,
         l2promo = l2promo_128b, oob = zero,  interleave = none>

func @main() {

  %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType

  gpu.launch … {

    %mbar = nvgpu.mbarrier.create -> <num_barriers = 1>
    nvgpu.mbarrier.init %mbar[0], %c1, predicate = %tidx0


    scf.if(%tidx0 == 0) {
      nvgpu.tma.async.load %tmaDesc[0, 0], %mbar[0] to %tileA : !descType
      nvgpu.mbarrier.arrive.expect_tx %mbar[0], 16384
    }


    nvgpu.mbarrier.try_wait.parity %mbar[0], %phase, %ticks
} }
```

Type keeps TMA information

TMA Example: nvgpu dialect

Shape Tile-A: 128x64 from Global → Shared Memory

```
!descType = !nvgpu.tensormap.descriptor
        <tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b,
         l2promo = l2promo_128b, oob = zero,  interleave = none>

func @main() {

  %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType

  gpu.launch … {

    %mbar = nvgpu.mbarrier.create -> <num_barriers = 1>
    nvgpu.mbarrier.init %mbar[0], %c1, predicate = %tidx0


    scf.if(%tidx0 == 0) {
      nvgpu.tma.async.load %tmaDesc[0, 0], %mbar[0] to %tileA : !descType
      nvgpu.mbarrier.arrive.expect_tx %mbar[0], 16384
    }


    nvgpu.mbarrier.try_wait.parity %mbar[0], %phase, %ticks
} }
```

Type keeps TMA information

Host calls *cuTensorMapEncodeTiled*

TMA Example: nvgpu dialect

Shape Tile-A: 128x64 from Global → Shared Memory

```
!descType = !nvgpu.tensormap.descriptor
        <tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b,
         l2promo = l2promo_128b, oob = zero,  interleave = none>

func @main() {

  %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType

  gpu.launch ... {

    %mbar = nvgpu.mbarrier.create -> <num_barriers = 1>
    nvgpu.mbarrier.init %mbar[0], %c1, predicate = %tidx0


    scf.if(%tidx0 == 0) {
      nvgpu.tma.async.load %tmaDesc[0, 0], %mbar[0] to %tileA : !descType
      nvgpu.mbarrier.arrive.expect_tx %mbar[0], 16384
    }


    nvgpu.mbarrier.try_wait.parity %mbar[0], %phase, %ticks
} }
```

Type keeps TMA information

Host calls *cuTensorMapEncodeTiled*

Create & Initialize Async Transaction Barrier

TMA Example: nvgpu dialect

Shape Tile-A: 128x64 from Global → Shared Memory

```
!descType = !nvgpu.tensormap.descriptor
        <tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b,
         l2promo = l2promo_128b, oob = zero,  interleave = none>

func @main() {

  %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType

  gpu.launch ... {

    %mbar = nvgpu.mbarrier.create -> <num_barriers = 1>
    nvgpu.mbarrier.init %mbar[0], %c1, predicate = %tidx0


    scf.if(%tidx0 == 0) {
      nvgpu.tma.async.load %tmaDesc[0, 0], %mbar[0] to %tileA : !descType
      nvgpu.mbarrier.arrive.expect_tx %mbar[0], 16384
    }


    nvgpu.mbarrier.try_wait.parity %mbar[0], %phase, %ticks
} }
```

Type keeps TMA information

Host calls *cuTensorMapEncodeTiled*

Create & Initialize Async Transaction Barrier

A thread:

- **Send request to TMA to load Tile-A**
- **Barrier arrives with 16k bytes**

TMA Example: nvgpu dialect

Shape Tile-A: 128x64 from Global → Shared Memory

```
!descType = !nvgpu.tensormap.descriptor
        <tensor = memref<128x64xf16, 3>,  swizzle = swizzle_128b,
         l2promo = l2promo_128b, oob = zero,  interleave = none>

func @main() {

  %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType

  gpu.launch ... {

    %mbar = nvgpu.mbarrier.create -> <num_barriers = 1>
    nvgpu.mbarrier.init %mbar[0], %c1, predicate = %tidx0


    scf.if(%tidx0 == 0) {
      nvgpu.tma.async.load %tmaDesc[0, 0], %mbar[0] to %tileA : !descType
      nvgpu.mbarrier.arrive.expect_tx %mbar[0], 16384
    }


    nvgpu.mbarrier.try_wait.parity %mbar[0], %phase, %ticks
} }
```

Type keeps TMA information

Host calls *cuTensorMapEncodeTiled*

Create & Initialize Async Transaction Barrier

A thread:

- Send request to TMA to load Tile-A
- Barrier arrives with 16k bytes

All Threads wait until the data is in shared memory

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, 0"(y)`	`readwrite`
`"=r"(y)`	`write`

Supports predicates

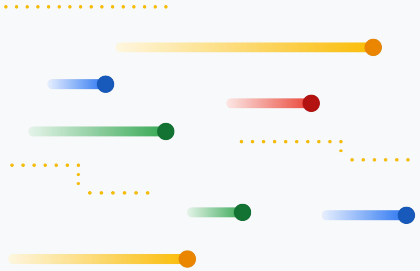
```
@%p opcode
```

```
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



Google Research

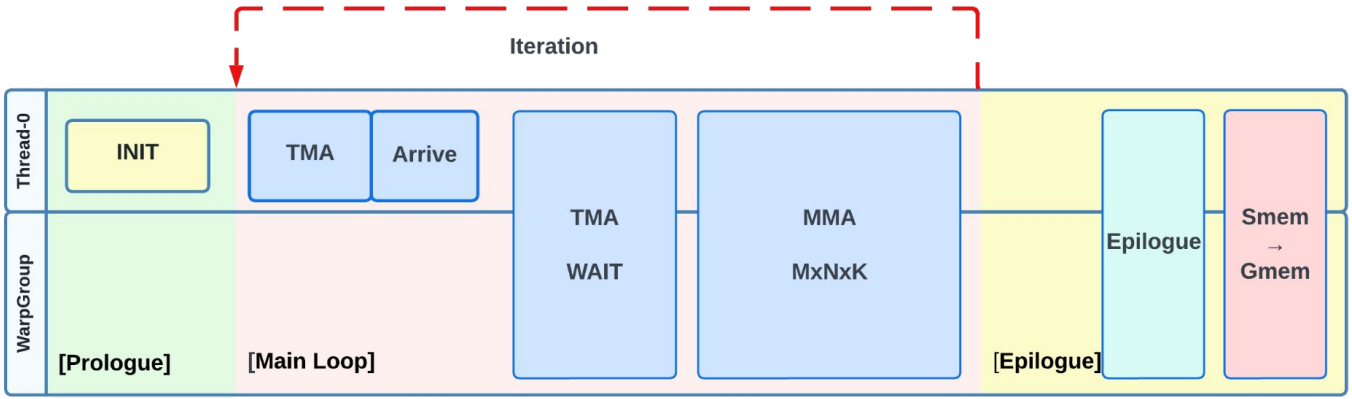
Use `NVGPU Dialect` to implement GEMM

GEMM: Single Stage (no pipelining)

Shared Memory Size = sizeof(Tile-A) + sizeof(Tile-B)

Each iteration:

- TMA copies Tile-A & Tile-B → Shared Memory
- Tensor Core does MMA on Tile-A & Tile-B



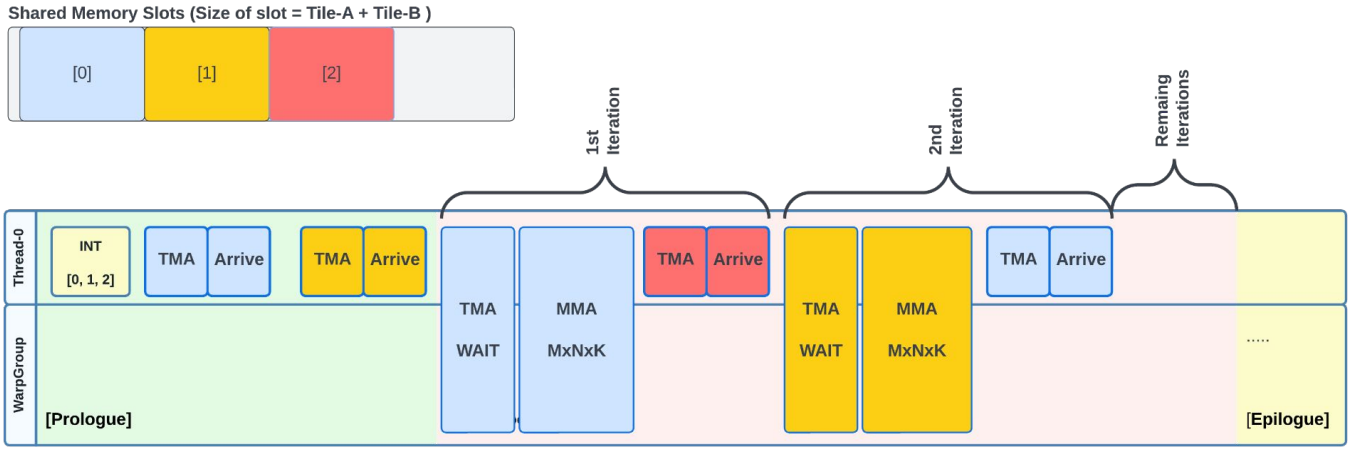
GEMM: Multi Stage (3 stages)

Prologue

- Copy TMA[Slot=0] and TMA[Slot=1]

Main Loop:

- **1st Iteration :** Tensor Core [Slot=0], TMA[Slot=2]
- **2nd Iteration :** Tensor Core [Slot=1], TMA[Slot=0]
- ...



Benchmark:
Analyzing Impact of Multistage

7296x256xK **[F32 += F16 * F16]**

Tile = 128x128x64

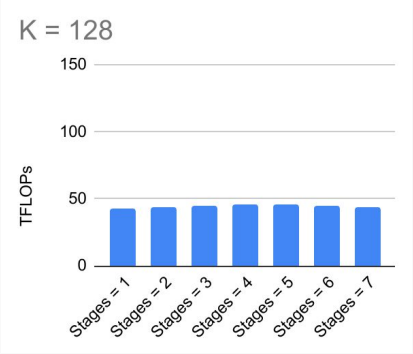
- Max **7 stages** fits shared memory

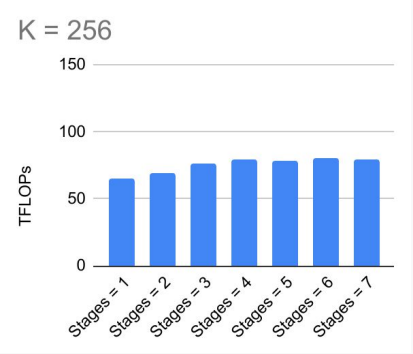
Larger-K has larger Main Loop

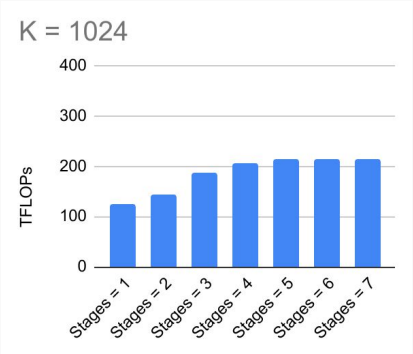
- Larger Main Loop
- Yields performance up to 3x

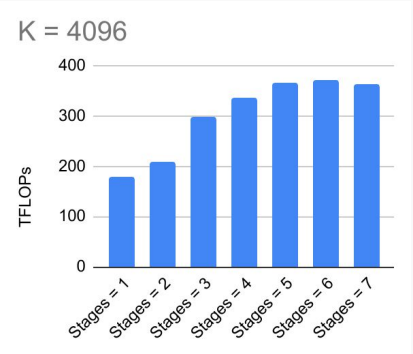
Sweet spot

- 3 or 4 stages









Benchmark:
cuBLAS vs MLIR

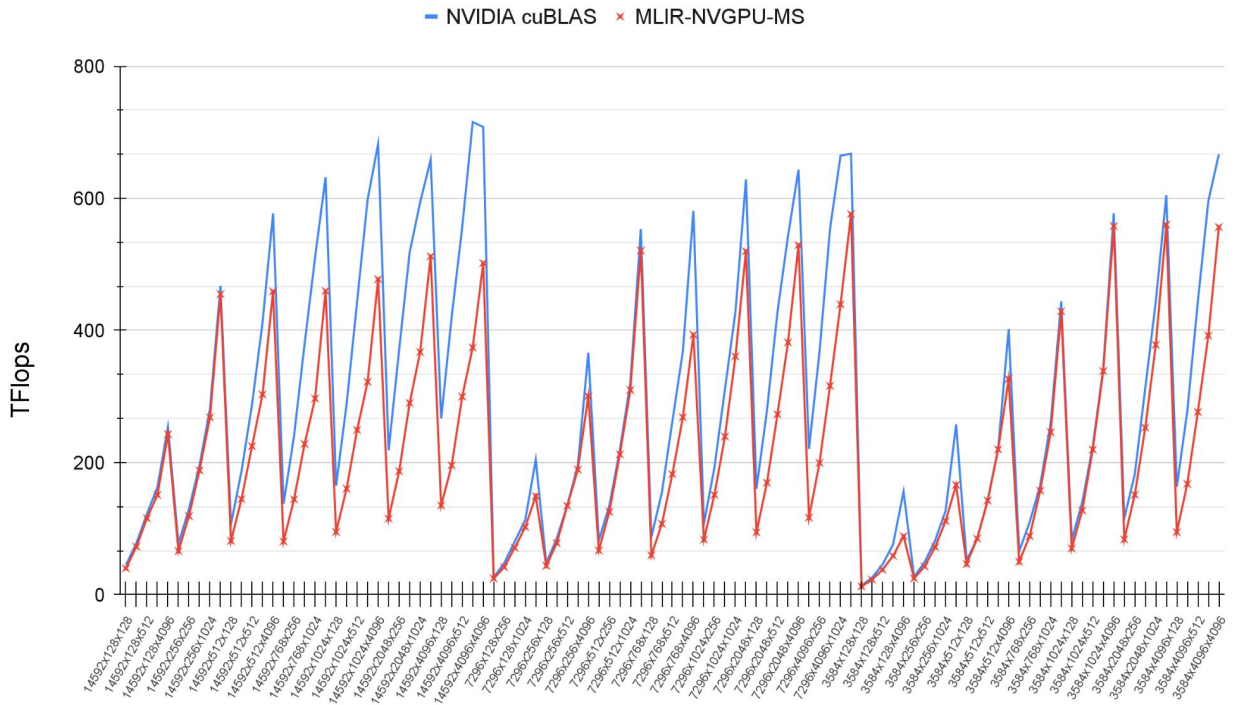
MLIR:

- Tile = 128x128x64
- 3 stages

What MLIR needs more:

- Warp Specialization
- Cluster
- Split-k
- ...

[F32 += F16 * F16]



Target NVIDIA Hopper in MLIR

MLIR Compiler

🤝 Community-driven with significant vendor contributions.

Focusing on Hopper Architecture

🎯 Advanced code generation for Tensor Core, TMA, etc.

NVGPU and NVVM Dialects

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

Performance

🚀 MLIR has close performance to cuBLAS

Upstream

✅ All the work presented is fully upstreamed to MLIR