rocMLIR: High-Performance ML Compilation for AMD GPUs with MLIR

Pablo Antonio Martinez
pablo.martinezsanchez@amd.com

EuroLLVM 2026
April 14, 2026

Agenda

1. Introduction and motivation
2. Design
3. Fusions
4. Optimizations
    - XCDs remapping
    - DirectToLDS
    - SplitK
5. Evaluation
6. Conclusions and future work

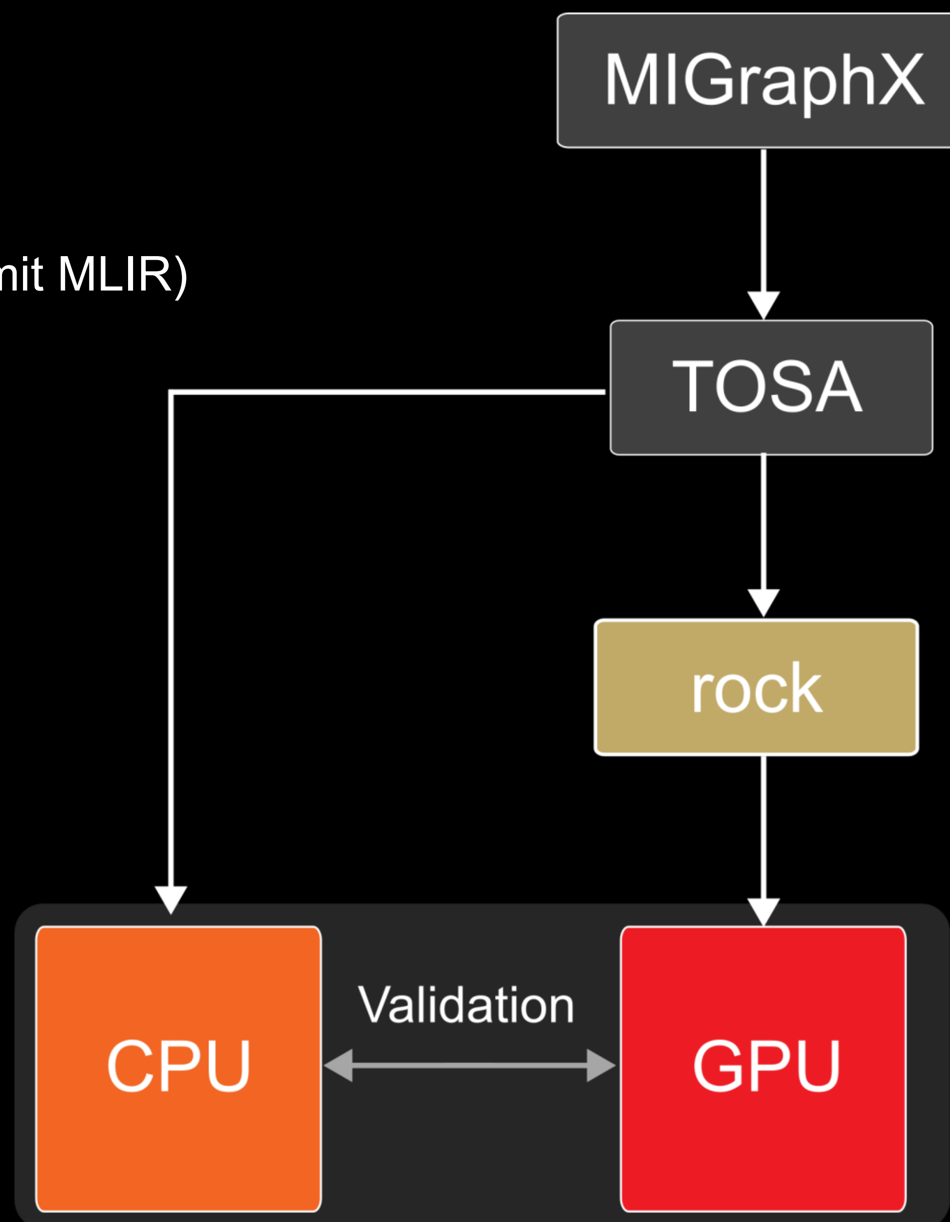
AMD
together we advance_

[Public]

1. Introduction

The compilation flow

- MIGraphX: AMD's graph compiler (it may emit MLIR)
- TOSA: Intermediate IR
- **rock**: The rocMLIR dialect



See more about MIGraphX here:

https://github.com/ROCm/AMDMIGraphX

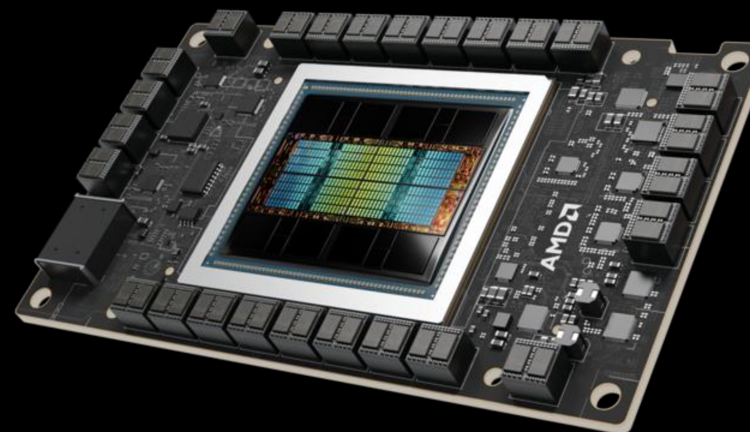
AMD
together we advance_

1. Introduction

Hardware targets

Gaming GPUs (RDNA)

Server GPUs (CDNA)



We support all GPUs that ROCm support

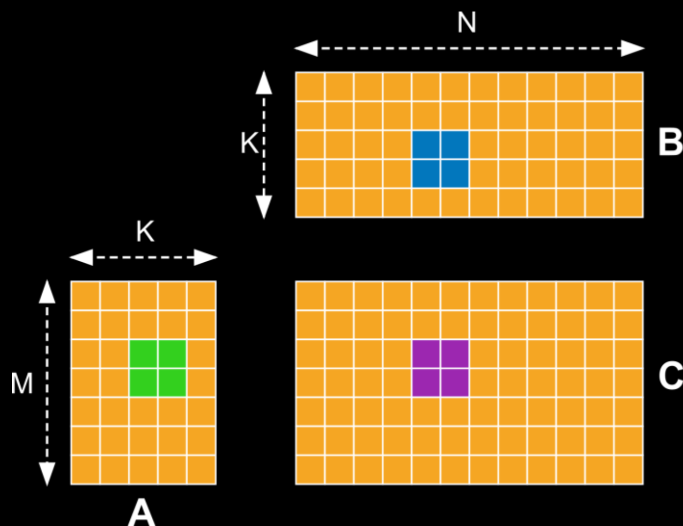
AMD
together we advance_

[Public]

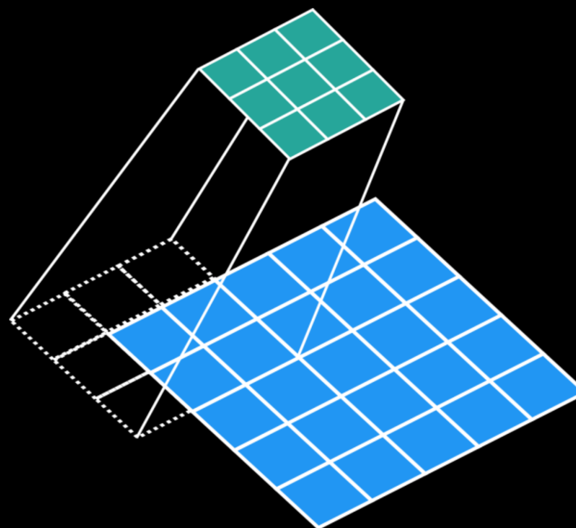
1. Introduction

What kernels do we support

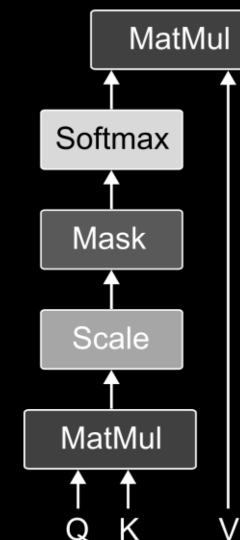
Matrix multiplication



Convolutions



Attention



Also:
- gemm+gemm
- conv+gemm
- Backward convolution

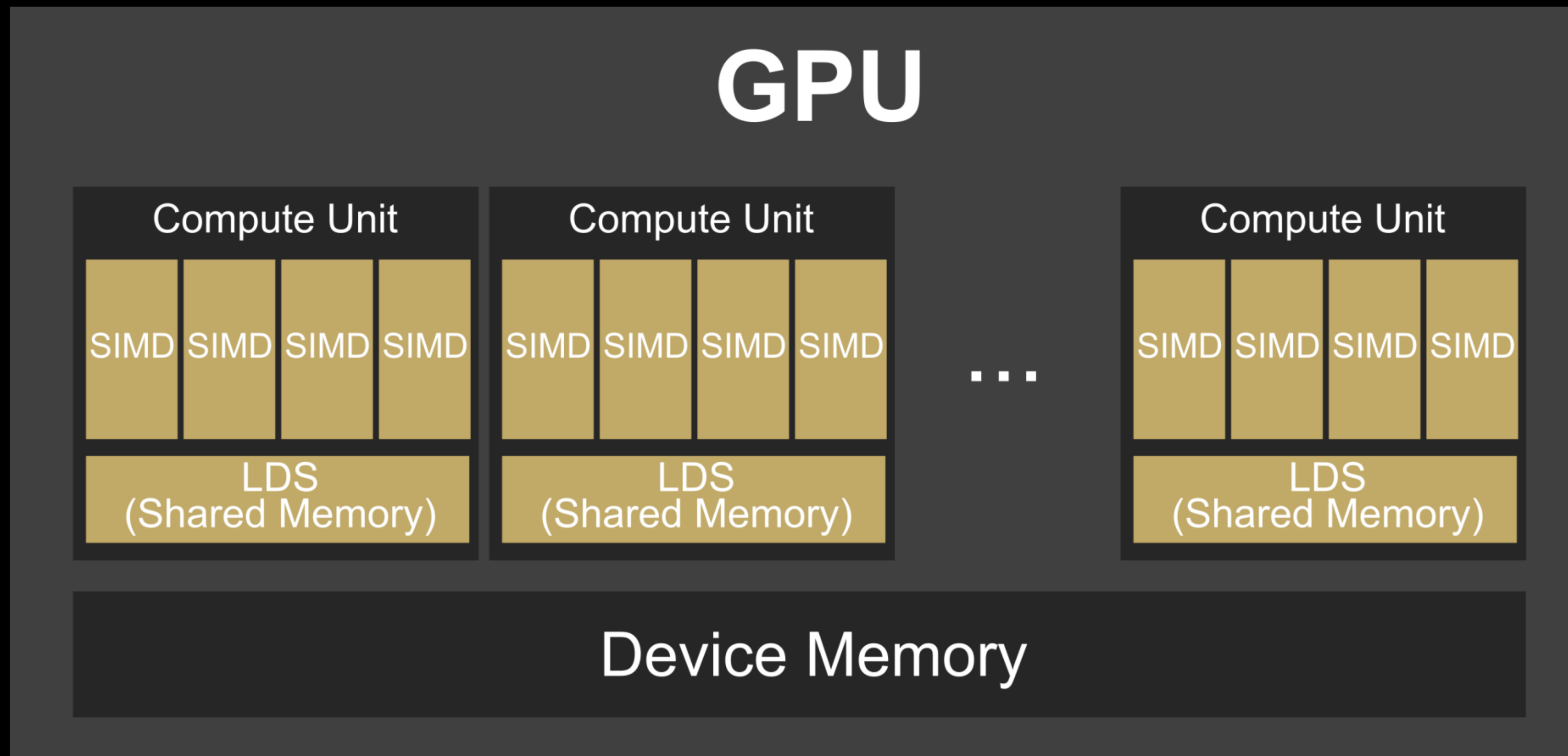
Also different variants (not exhaustive list):
- KV-Cache
- Grouped convolutions
- Scaled GEMMs
- …

AMD
together we advance_

[Public]

1. Introduction

Quick background on GPUs



AMD
together we advance_

[Public]

2. Design

MIGraphX IR kernel example

```
func.func private @mlir_attention(
  %arg0: !migraphx.shaped<1x64x64xf32, 4096x64x1>,
  %arg1: !migraphx.shaped<1x64x64xf32, 4096x64x1>,
  %arg2: !migraphx.shaped<1x64x64xf32, 4096x64x1>) ->
  (!migraphx.shaped<1x64x64xf32, 4096x64x1>)
{
  %0 = migraphx.dot %arg0, %arg1:
    !migraphx.shaped<1x64x64xf32, 4096x64x1>,
    !migraphx.shaped<1x64x64xf32, 4096x64x1>  ->
    !migraphx.shaped<1x64x64xf32, 4096x64x1>

  %1 = migraphx.softmax %0 {axis = 2 : i64} :
    !migraphx.shaped<1x64x64xf32, 4096x64x1> ->
    !migraphx.shaped<1x64x64xf32, 4096x64x1>

  %2 = migraphx.dot %1, %arg2:
    !migraphx.shaped<1x64x64xf32, 4096x64x1>,
    !migraphx.shaped<1x64x64xf32, 4096x64x1> ->
    !migraphx.shaped<1x64x64xf32, 4096x64x1>

  return %2 : !migraphx.shaped<1x64x64xf32, 4096x64x1>
}
```


AMD
together we advance_

[Public]

2. Design
IR Design

```
$ rocmlir-gen --arch gfx950 --num_cu 256 --operation gemm -t f32 -g 1 -m 1024 -n 1024 -k 1024
```


AMD
together we advance_

[Public]

2. Design

IR Design

```
$ rocmlir-gen --arch gfx950 --num_cu 256 --operation gemm -t f32 -g 1 -m 1024 -n 1024 -k 1024

module attributes {
  func.func @rock_gemm(
    %arg0: memref<1048576xf32>,
    %arg1: memref<1048576xf32>,
    %arg2: memref<1048576xf32>)
  attributes
  {
    enable_splitk_for_tuning,
    kernel,
    mhal.arch = "amdgcn-amd-amdhsa:gfx950",
    num_chiplets = 8 : i64,
    num_cu = 256 : i64
  }
  {
    %0 = rock.transform %arg0 by #transform1 : memref<1048576xf32> to memref<1x1024x1024xf32>
    %1 = rock.transform %arg1 by #transform2 : memref<1048576xf32> to memref<1x1024x1024xf32>
    %2 = rock.transform %arg2 by #transform2 : memref<1048576xf32> to memref<1x1024x1024xf32>
    rock.gemm %2 = %0 * %1 features = #gemm_features storeMethod = set :
      memref<1x1024x1024xf32> = memref<1x1024x1024xf32> * memref<1x1024x1024xf32>
    return
  }
}
```


AMD
together we advance_

[Public]

2. Design

IR Design

```
$ rocmlir-gen --arch gfx950 --num_cu 256 --operation gemm -t f32 -g 1 -m 1024 -n 1024 -k 1024

module attributes {
  func.func @rock_gemm(
    %arg0: memref<1048576xf32>,
    %arg1: memref<1048576xf32>,
    %arg2: memref<1048576xf32>)
  attributes
  {
    enable_splitk_for_tuning,
    kernel,
    mhal.arch = "amdgcn-amd-amdhsa:gfx950",
    num_chiplets = 8 : i64,
    num_cu = 256 : i64
  }
  {
    %0 = rock.transform %arg0 by #transform1 : memref<1048576xf32> to memref<1x1024x1024xf32>
    %1 = rock.transform %arg1 by #transform2 : memref<1048576xf32> to memref<1x1024x1024xf32>
    %2 = rock.transform %arg2 by #transform2 : memref<1048576xf32> to memref<1x1024x1024xf32>
    rock.gemm %2 = %0 * %1 features = #gemm_features storeMethod = set :
      memref<1x1024x1024xf32> = memref<1x1024x1024xf32> * memref<1x1024x1024xf32>
    return
  }
}
```

AMD
together we advance_

[Public]

2. Design

IR Design

```
#map = affine_map<(d0, d1, d2) -> (d1 * 1024 + d2)>
#transform1 = #rock.transform_map<#map by
[
  <Unmerge{1024, 1024} ["m", "k"] at [1, 2] -> ["raw"] at [0]>,
  <AddDim{1} ["g"] at [0] -> [] at []>
] bounds = [1, 1024, 1024] -> [1048576]>
#transform2 = #rock.transform_map<#map by
[
  <Unmerge{1024, 1024} ["k", "n"] at [1, 2] -> ["raw"] at [0]>,
  <AddDim{1} ["g"] at [0] -> [] at []>
] bounds = [1, 1024, 1024] -> [1048576]>
#transform3 = #rock.transform_map<#map by
[
  <Unmerge{1024, 1024} ["m", "n"] at [1, 2] -> ["raw"] at [0]>,
  <AddDim{1} ["g"] at [0] -> [] at []>
] bounds = [1, 1024, 1024] -> [1048576]>

module attributes {
  func.func @rock_gemm(...) {
    %0 = rock.transform %arg0 by #transform1 : memref<1048576xf32> to memref<1x1024x1024xf32>
    %1 = rock.transform %arg1 by #transform2 : memref<1048576xf32> to memref<1x1024x1024xf32>
    %2 = rock.transform %arg2 by #transform2 : memref<1048576xf32> to memref<1x1024x1024xf32>
    rock.gemm %2 = %0 * %1 features = #gemm_features storeMethod = set …
    return
  }
}
```

See Krzysztof presentation for more details:

https://youtu.be/xrMJis3Rak0

Title: *Coordinate Transformations in AMD's rocMLIR*

AMD
together we advance_

[Public]

2. Design

Lowering pipeline

Our GPU pipeline looks like this:

```
funcPm.addPass(rock::createRockAffixTuningParametersPass(
        rock::RockAffixTuningParametersPassOptions{options.tuningFallback}));
funcPm.addPass(rock::createRockConvToGemmPass());
funcPm.addPass(rock::createRockGemmLinalgSplitkNormalizationPass());
funcPm.addPass(rock::createRockGemmToGridwisePass());
funcPm.addPass(rock::createRockRegularizePass());
funcPm.addPass(rock::createRockShuffleGemmForReductions());
funcPm.addPass(rock::createRockGridwiseGemmToBlockwisePass());
funcPm.addPass(rock::createRockBlockwiseLoadTileToThreadwisePass());

funcPm.addPass(rock::createRockLinalgAlignPass());
funcPm.addPass(rock::createRockBlockwiseGemmToThreadwisePass());
funcPm.addPass(rock::createRockPipelinePass());
funcPm.addPass(createCanonicalizerPass());
funcPm.addPass(createConvertLinalgToAffineLoopsPass());
funcPm.addPass(rock::createRockVectorizeFusionsPass());
funcPm.addPass(rock::createRockAddAsyncWaitPass());

funcPm.addPass(rock::createRockAnnotateLivenessPass());
funcPm.addPass(rock::createRockReuseLDSPass());
funcPm.addPass(rock::createRockOutputSwizzlePass());
funcPm.addPass(rock::createRockAnnotateLivenessPass());
funcPm.addPass(rock::createRockReuseLDSPass());
```


AMD
together we advance_

[Public]

2. Design

Lowering pipeline

IR after `RockAffixTuningParametersPass`

```
func.func @rock_gemm(...) {
  ...
  rock.gemm %2 = %0 * %1 features =  #gemm_features storeMethod = set
  {
    derivedBlockSize = 256 : i32,
    params = #rock.accel_gemm_params<
      kpackPerBlock = 8,
      mPerBlock = 32,
      nPerBlock = 64,
      kpack = 8,
      mPerWave = 32,
      nPerWave = 16,
      mnPerXdl = 16,
      splitKFactor = 1,
      scheduleVersion = 1,
      outputSwizzle = 2,
      wavesPerEU = 0,
      gridGroupSize = 0,
      forceUnroll = true
    >
  } : memref<1x1024x1024xf32> = memref<1x1024x1024xf32> * memref<1x1024x1024xf32>

  return
}
```


AMD
together we advance_

[Public]

2. Design

Lowering pipeline

IR after `RockGridwiseGemmToBlockwisePass` (part 1)

```
%4 = rock.workgroup_id : index
%5 = rock.workitem_id : index

...

%c4 = arith.constant 4 : index
%c128 = arith.constant 128 : index
%6 = arith.remui %4, %c4 : index
%7 = arith.divui %4, %c4 : index
%8 = arith.muli %6, %c128 : index
%9 = arith.addi %7, %8 : index
%c511 = arith.constant 511 : index
%10 = arith.cmpi sgt, %4, %c511 : index
%11 = arith.select %10, %4, %9 : index

...

%22 = rock.alloc() : memref<8192xi8, #gpu.address_space<workgroup>>
%23 = rock.alloc() : memref<16384xi8, #gpu.address_space<workgroup>>

...

%24 = rock.alloc() : memref<16xf32, #gpu.address_space<private>>
%25 = rock.alloc() : memref<16xf32, #gpu.address_space<private>>
```


AMD
together we advance_

2. Design

Lowering pipeline

IR after `RockGridwiseGemmToBlockwisePass` (part 2)

```
scf.for %arg3 = %c0_3 to %c16_2 step %c1 {
  rock.blockwise_load_tile %1[%arg3, %12, %19, %21, %5] LDS -> %23 -> %25
  rock.blockwise_load_tile %3[%arg3, %12, %19, %21, %5] LDS -> %22 -> %24
  rock.lds_barrier
  rock.stage {
    rock.blockwise_gemm_accel %26 += %24 from %view * %25 from %view_1
      features =  {#gemm_features} : ...
      rock.yield
    } {name = "MMA"}
    rock.lds_barrier
  } {pipeline = #rock.pipeline<2>}
  %27 = rock.alloc() : memref<8xf32, #regs>
  rock.transforming_for {forceUnroll, useIndexDiffs}
    (%arg3) = [](%c0), (%arg4) = [#transform_map4](%c0_4)
    (%arg5, %arg6) = validity bounds [2] strides [1] {
    %28 = memref.load %26[%arg3] : memref<2xvector<4xf32>, #regs>>
    rock.in_bounds_store %28 -> %27[%arg4] : vector<4xf32> -> memref<8xf32, #lds>>,
index
    rock.yield
  }
}
rock.threadwise_write_all {forceUnroll, useIndexDiffs} %27 ... :
  memref<8xf32, #regs> -> memref<1x1024x1024xf32>
```


AMD
together we advance_

[Public]

2. Design

Lowering pipeline

IR after `RockBlockwiseLoadTileToThreadwisePass` (GlobalRead)

```
scf.for %arg3 = %c0_3 to %c16_2 step %c1 {
  rock.stage {
    %33 = rock.transform %1 by #transform_map5 :
      memref<1x1024x1024xf32> to memref<16x1x32x16x256x8xf32>
    %34 = rock.threadwise_read_into [](%33) [%arg3, %12, %19, %21, %5] -> %29 :
      memref<16x1x32x16x256x8xf32> -> memref<8xf32, #regs>, vector<8xi1>

    %36 = rock.transform %2 by #transform_map7 : memref<1x1024x1024xf32> to
       memref<16x1x32x16x256x16xf32>
    %37 = rock.threadwise_read_into [](%36) [%arg3, %12, %19, %21, %5] -> %27 :
       memref<16x1x32x16x256x16xf32> -> memref<16xf32, #regs>, vector<16xi1>
    rock.yield
  } {name = "GlobalRead"}

  rock.stage { ... } {name = "LDSWrite"}

  rock.lds_barrier

  rock.stage { ... } {name = "MMA"}

  rock.lds_barrier
}
{pipeline = #rock.pipeline<2>}
```

AMD
together we advance_

[Public]

2. Design

Lowering pipeline

IR after `RockBlockwiseLoadTileToThreadwisePass (LDSWrite)`

```
scf.for %arg3 = %c0_3 to %c16_2 step %c1 {
  rock.stage { ... } {name = "GlobalRead"}

  rock.stage {
    %32 = rock.workitem_id : index
    %34 = rock.transform %29 by #transform_map9 : memref<8xf32, #regs> to
        memref<4x2xf32, #regs>
    %36 = rock.transform %30 by #transform_map11 : memref<8xf32, #regs> to
        memref<4x2xf32, #regs>

    %42 = rock.transform ...
    rock.threadwise_copy %34 -> %36 : memref<4x2xf32, #regs> -> memref<4x2xf32, #regs>
    rock.threadwise_write_all {forceUnroll, useIndexDiffs} %30 -> [](%42) [%32]
        by  set : memref<8xf32, #regs> -> memref<256x8xvector<4xf32>, #lds>

    // Same chain of transforms for B tile
    rock.yield
  } {name = "LDSWrite"}
  rock.lds_barrier

  rock.stage { ... } {name = "MMA"}

  rock.lds_barrier

} {pipeline = #rock.pipeline<2>}
```


AMD
together we advance_

[Public]

2. Design

Lowering pipeline

IR after `RockBlockwiseLoadTileToThreadwisePass` (MMA)

```
scf.for %arg3 = %c0_3 to %c16_2 step %c1 {
  rock.stage { ... } {name = "GlobalRead"}
  rock.stage { ... } {name = "LDSWrite"}

  rock.lds_barrier

  rock.stage {
    rock.blockwise_gemm_accel %26 += %24 from %view * %25 from %view_1
    features =  #gemm_features
  : memref<2xvector<4xf32>, #regs> +=
    memref<16xf32, #regs> from
    memref<256xvector<8xf32>, #lds> *
    memref<16xf32, #regs> from memref<512xvector<8xf32>, #lds>
    rock.yield
  } {name = "MMA"}
  rock.lds_barrier

} {pipeline = #rock.pipeline<2>}
```


AMD
together we advance_

2. Design

Lowering pipeline

IR after we reach amdgpu (upstream dialect)

```
amdgpu.gather_to_lds %memspacecast[%227], %view_2[%228] : f128,
        memref<1048576xf32, #gpu.address_space<global>>,
        memref<8192xf32, #gpu.address_space<workgroup>>
```

NOTE: `gather_to_lds` is generated only if DirectToLDS optimization is enabled

```
affine.for %arg5 = 0 to 16 {
  %174 = memref.load %18[%arg5] : memref<16xf32, #regs>
  %175 = memref.load %19[%arg5] : memref<16xf32, #regs>
  %176 = memref.load %20[%arg4] : memref<2xvector<4xf32>, #regs>
  %177 = amdgpu.mfma 16x16x4 %174 * %175 + %176 blgp =  none : f32, f32, vector<4xf32>
  memref.store %177, %20[%arg4] : memref<2xvector<4xf32>, #regs>
}
```

```
amdgpu.lds_barrier
```


AMD
together we advance_

2. Design

Lowering pipeline

IR after we reach rocdl (upstream dialect)

```
rocdl.mfma.f32.16x16x4f32 %494, %497, %500, %501, %502, %503 :
        (f32, f32, vector<4xf32>, i32, i32, i32) -> vector<4xf32>
```

```
rocdl.s.waitcnt -7937
rocdl.s.barrier
```

The rest of the IR is in llvm dialect, so we can lower all to LLVM…

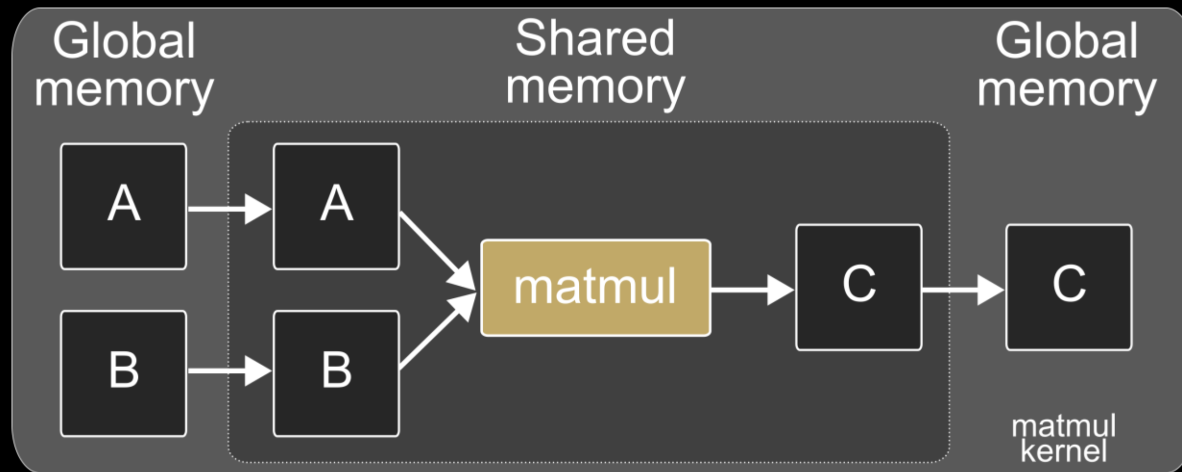
… and then, to assembly!

AMD
together we advance_

[Public]

3. Fusions

3.1 Fusions

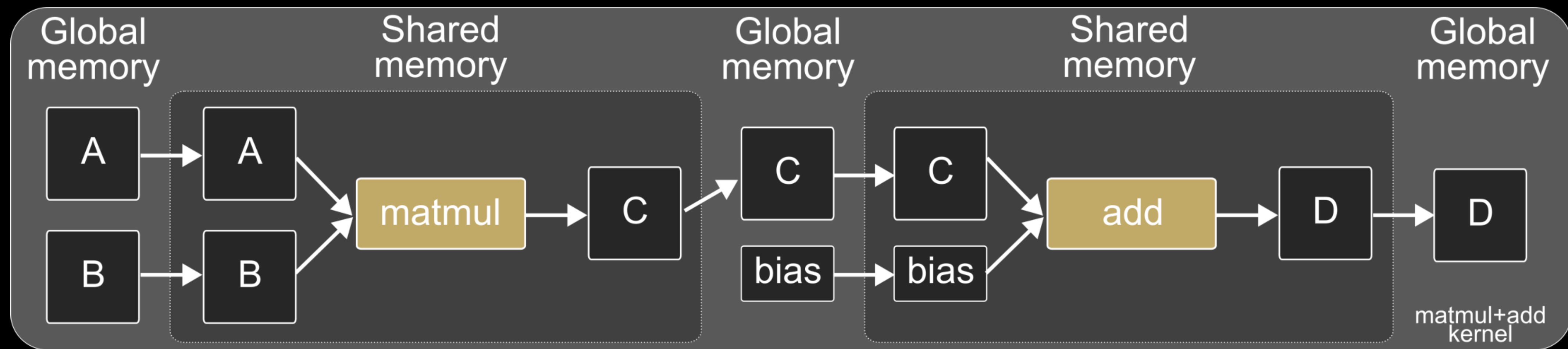


AMD
together we advance_

[Public]

3. Fusions

3.1 Fusions

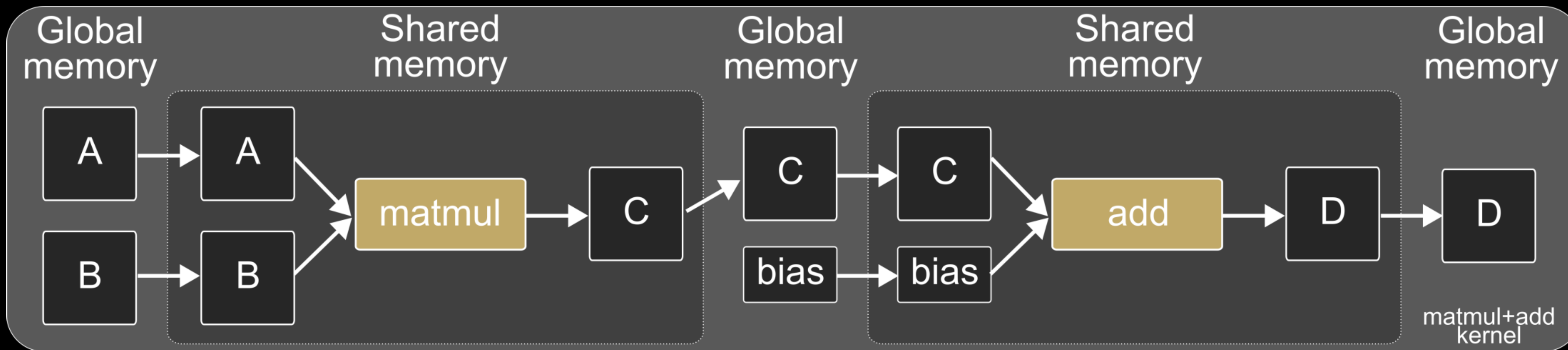


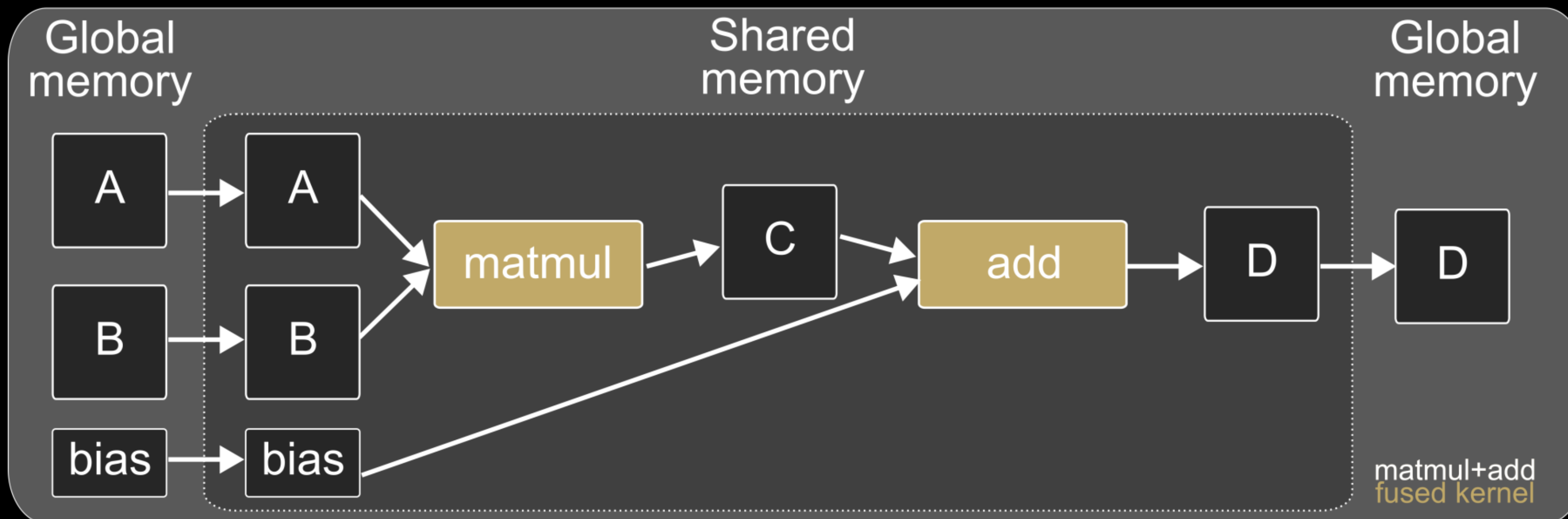
AMD
together we advance_

[Public]

3. Fusions

3.1 Fusions



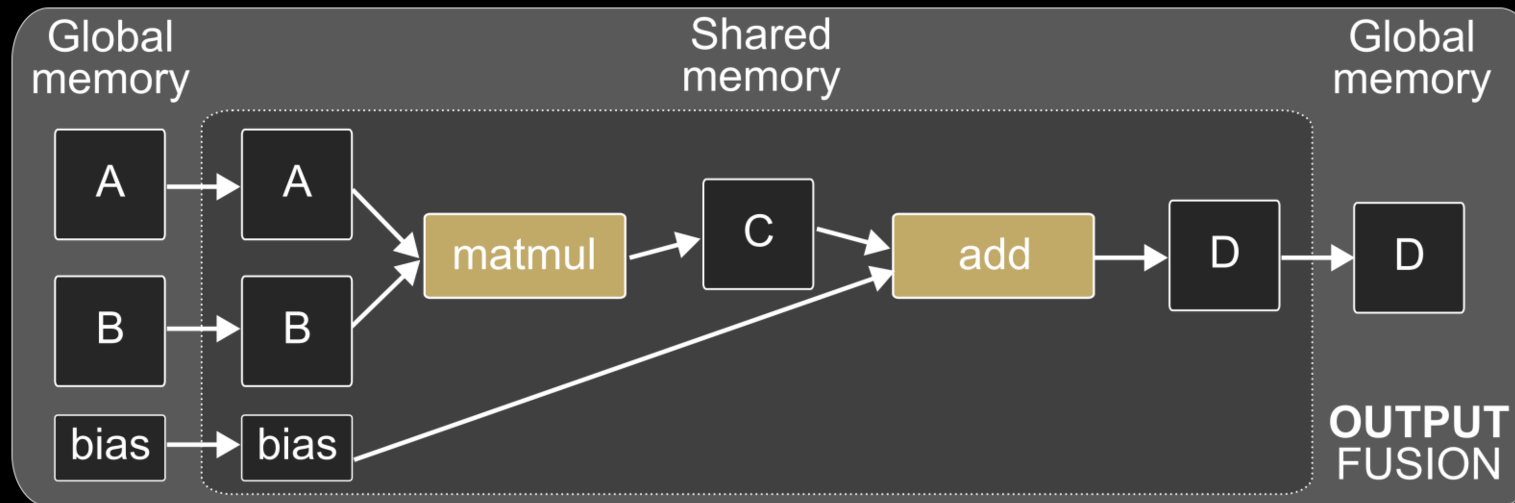


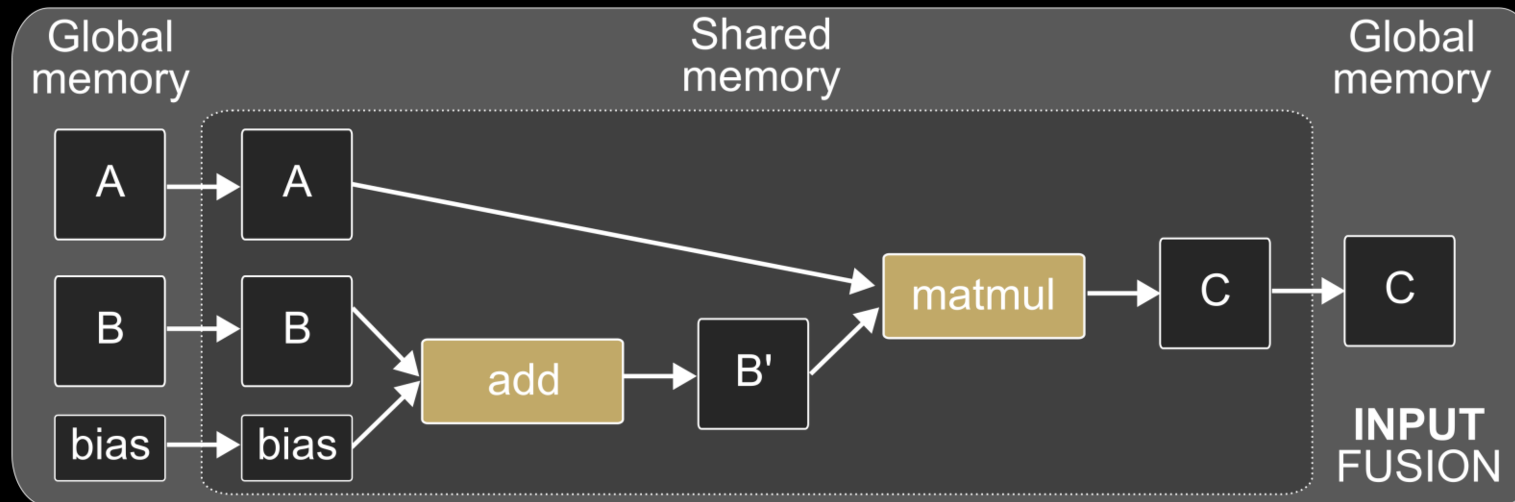
AMD
together we advance_

[Public]

3. Fusions

Input vs Output fusions





AMD
together we advance_

3. Fusions

Input fusion

```
func.func @input_fusion(
  %arg0: memref<1x1024x1024xf32>,
  %arg1: memref<1x1024x1024xf32>,
  %arg2: memref<1x1024x1024xf32>)
{
  %cst = arith.constant 4.000000e+00 : f32
  %alloc = memref.alloc() : memref<1x1024x1024xf32>

  linalg.generic {
    indexing_maps = [#map, #map],
    iterator_types = ["parallel", "parallel", "parallel"]}
    ins(%arg0 : memref<1x1024x1024xf32>) outs(%alloc : memref<1x1024x1024xf32>) {
  ^bb0(%in: f32, %out: f32):
    %0 = arith.addf %in, %cst : f32
    linalg.yield %0 : f32
  }

  rock.gemm %arg2 = %alloc * %arg1 {arch = "gfx950", params = #general_gemm_params} :
    memref<1x1024x1024xf32> =
    memref<1x1024x1024xf32> * memref<1x1024x1024xf32>

  return
}
```


AMD
together we advance_

[Public]

3. Fusions

3.1 Input fusion

Fusion happens after ToBlockwise pass, in a pass called RockLinalgAlignPass

This is actually **loop fusion** because the add is fused within the kernel loop

```
scf.for %arg3 = %c0_11 to %c64_10 step %c1 {
  rock.stage {
    ...
    %53 = rock.threadwise_read_into [](%52) [%arg3, %19, %26, %28, %6] -> %29 :
        memref<64x1x8x8x128x16xf32> -> memref<16xf32, #regs>, vector<16xi1>
    %54 = rock.alloc() : memref<16xf32, #regs>
    linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]}
      ins(%29 : memref<16xf32, #regs>) outs(%54 : memref<16xf32, #regs>) {
    ^bb0(%in: f32, %out: f32):
      %57 = arith.addf %in, %cst_1 : f32
      linalg.yield %57 : f32
    }
    ...
  } {name = "GlobalRead"}
  rock.stage {
    ...
    rock.threadwise_write_all %54 -> %43[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
    rock.threadwise_write_all %55 -> %47[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
  } {name = "LDSWrite"}
  rock.stage {
    rock.blockwise_gemm %4 += %43 * %47 {params = #rock.general_gemm_params} :
      memref<8x16xf32, #regs> += memref<16x128x1xf32, #lds> * memref<16x128x1xf32, #lds>
  } {name = "MMA"}
} {pipeline = #rock.pipeline<2>}
```


AMD
together we advance_

[Public]

3. Fusions

3.1 Input fusion

```
scf.for %arg3 = %c0_11 to %c64_10 step %c1 {
  rock.stage {
    ...
    %53 = rock.threadwise_read_into [](%52) [%arg3, %19, %26, %28, %6] -> %29 :
      memref<64x1x8x8x128x16xf32> -> memref<16xf32, #regs>, vector<16xi1>
    %54 = rock.alloc() : memref<16xf32, #regs>
    linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]}
      ins(%29 : memref<16xf32, #regs>) outs(%54 : memref<16xf32, #regs>) {
    ^bb0(%in: f32, %out: f32):
      %57 = arith.addf %in, %cst_1 : f32
      linalg.yield %57 : f32
    }
    ...
  } {name = "GlobalRead"}
  rock.stage {
    ...
    rock.threadwise_write_all %54 -> %43[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
    rock.threadwise_write_all %55 -> %47[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
  } {name = "LDSWrite"}
  rock.stage {
    rock.blockwise_gemm %4 += %43 * %47 {params = #rock.general_gemm_params} :
      memref<8x16xf32, #regs> += memref<16x128x1xf32, #lds> * memref<16x128x1xf32, #lds>
  } {name = "MMA"}
} {pipeline = #rock.pipeline<2>}
```

Load A tile from global memory to registers

AMD
together we advance_

3. Fusions

3.1 Input fusion

```
scf.for %arg3 = %c0_11 to %c64_10 step %c1 {
  rock.stage {
    ...
    %53 = rock.threadwise_read_into [](%52) [%arg3, %19, %26, %28, %6] -> %29 :
      memref<64x1x8x8x128x16xf32> -> memref<16xf32, #regs>, vector<16xi1>
    %54 = rock.alloc() : memref<16xf32, #regs>
    linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]}
      ins(%29 : memref<16xf32, #regs>) outs(%54 : memref<16xf32, #regs>) {
    ^bb0(%in: f32, %out: f32):
      %57 = arith.addf %in, %cst_1 : f32
      linalg.yield %57 : f32
    }
    ...
  } {name = "GlobalRead"}
  rock.stage {
    ...
    rock.threadwise_write_all %54 -> %43[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
    rock.threadwise_write_all %55 -> %47[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
  } {name = "LDSWrite"}
  rock.stage {
    rock.blockwise_gemm %4 += %43 * %47 {params = #rock.general_gemm_params} :
      memref<8x16xf32, #regs> += memref<16x128x1xf32, #lds> * memref<16x128x1xf32, #lds>
  } {name = "MMA"}
} {pipeline = #rock.pipeline<2>}
```

Add happens in registers (fused within the kernel loop)

AMD
together we advance_

[Public]

3. Fusions

3.1 Input fusion

```
scf.for %arg3 = %c0_11 to %c64_10 step %c1 {
  rock.stage {
    ...
    %53 = rock.threadwise_read_into [](%52) [%arg3, %19, %26, %28, %6] -> %29 :
        memref<64x1x8x8x128x16xf32> -> memref<16xf32, #regs>, vector<16xi1>
    %54 = rock.alloc() : memref<16xf32, #regs>
    linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]}
      ins(%29 : memref<16xf32, #regs>) outs(%54 : memref<16xf32, #regs>) {
    ^bb0(%in: f32, %out: f32):
      %57 = arith.addf %in, %cst_1 : f32
      linalg.yield %57 : f32
    }
    ...
  } {name = "GlobalRead"}
  rock.stage {
    ...
    rock.threadwise_write_all %54 -> %43[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
    rock.threadwise_write_all %55 -> %47[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
  } {name = "LDSWrite"}
  rock.stage {
    rock.blockwise_gemm %4 += %43 * %47 {params = #rock.general_gemm_params} :
      memref<8x16xf32, #regs> += memref<16x128x1xf32, #lds> * memref<16x128x1xf32, #lds>
  } {name = "MMA"}
} {pipeline = #rock.pipeline<2>}
```

A' and B tiles (%54 and %55) are moved from registers into LDS memory

AMD
together we advance_

[Public]

3. Fusions

3.1 Input fusion

```
scf.for %arg3 = %c0_11 to %c64_10 step %c1 {
  rock.stage {
    ...
    %53 = rock.threadwise_read_into [](%52) [%arg3, %19, %26, %28, %6] -> %29 :
        memref<64x1x8x8x128x16xf32> -> memref<16xf32, #regs>, vector<16xi1>
    %54 = rock.alloc() : memref<16xf32, #regs>
    linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]}
      ins(%29 : memref<16xf32, #regs>) outs(%54 : memref<16xf32, #regs>) {
    ^bb0(%in: f32, %out: f32):
      %57 = arith.addf %in, %cst_1 : f32
      linalg.yield %57 : f32
    }
    ...
  } {name = "GlobalRead"}
  rock.stage {
    ...
    rock.threadwise_write_all %54 -> %43[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
    rock.threadwise_write_all %55 -> %47[%6]: memref<16xf32,#regs> -> memref<128x16xf32,#lds>
  } {name = "LDSWrite"}
  rock.stage {
    rock.blockwise_gemm %4 += %43 * %47 {params = #rock.general_gemm_params} :
      memref<8x16xf32, #regs> += memref<16x128x1xf32, #lds> * memref<16x128x1xf32, #lds>
  } {name = "MMA"}
} {pipeline = #rock.pipeline<2>}
```

A' tile (%43) has the addf computed

AMD together we advance_

3. Fusions

3.2 Output fusion

```
func.func @output_fusion(
  %arg0: memref<1x2x320xf32>,
  %arg1: memref<1x2x1280xf32>,
  %arg2: memref<1x1280x320xf32>,
  %arg3: memref<1x2x320xf32>)
{
  %alloc = memref.alloc() {alignment = 64 : i64} : memref<1x2x320xf32>
  rock.gemm %alloc = %arg1 * %arg2 features = #gemm_features :
    memref<1x2x320xf32> = memref<1x2x1280xf32> * memref<1x1280x320xf32>
  %0 = rock.transform %alloc by #transform1 : memref<1x2x320xf32> to memref<2x320xf32>
  %1 = rock.transform %arg0 by #transform1 : memref<1x2x320xf32> to memref<2x320xf32>
  %alloc_0 = memref.alloc() {alignment = 64 : i64} : memref<2x320xf32>
  linalg.generic {indexing_maps = [#map1, #map1, #map1], iterator_types = ["parallel",
"parallel"]} ins(%0, %1 : memref<2x320xf32>, memref<2x320xf32>) outs(%alloc_0 :
memref<2x320xf32>) {
  ^bb0(%in: f32, %in_1: f32, %out: f32):
    %3 = arith.addf %in, %in_1 : f32
    linalg.yield %3 : f32
  }
  %2 = rock.transform %alloc_0 by #transform2 : memref<2x320xf32> to memref<1x2x320xf32>
  memref.copy %2, %arg3 : memref<1x2x320xf32> to memref<1x2x320xf32>
  return
}
```


AMD
together we advance_

[Public]

3. Fusions

3.2 Output fusion

```
scf.for %arg4 = %c0_3 to %c20 step %c1_2 {
  rock.stage { ... } {name = "GlobalRead"}
  rock.stage { ... } {name = "LDSWrite"}
  rock.lds_barrier
  rock.stage { ... } {name = "MMA"}
  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}
...

rock.threadwise_read_into [...](%31) [%6, %13, %15, %5] -> %29 :
  memref<1x16x320xf32> -> memref<4xf32, #regs>

...

linalg.generic {
  indexing_maps = [#map30, #map30, #map30],
  iterator_types = ["parallel"]}
  ins(%25, %29 : memref<4xf32, #regs>, memref<4xf32, #regs>)
  outs(%28 : memref<4xf32, #regs>) {
^bb0(%in: f32, %in_6: f32, %out: f32):
  %37 = arith.addf %in, %in_6 : f32
  linalg.yield %37 : f32
}
...
```

This is the matmul kernel. It produces the output tile in %25, which resides in registers

AMD
together we advance_

3. Fusions

3.2 Output fusion

```
scf.for %arg4 = %c0_3 to %c20 step %c1_2 {
  rock.stage { ... } {name = "GlobalRead"}
  rock.stage { ... } {name = "LDSWrite"}
  rock.lds_barrier
  rock.stage { ... } {name = "MMA"}
  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}
...
```

```
rock.threadwise_read_into [...](%31) [%6, %13, %15, %5] -> %29 :
  memref<1x16x320xf32> -> memref<4xf32, #regs>
```

The bias tile is loaded from global into registers

```
...

linalg.generic {
  indexing_maps = [#map30, #map30, #map30],
  iterator_types = ["parallel"]}
  ins(%25, %29 : memref<4xf32, #regs>, memref<4xf32, #regs>)
  outs(%28 : memref<4xf32, #regs>) {
^bb0(%in: f32, %in_6: f32, %out: f32):
  %37 = arith.addf %in, %in_6 : f32
  linalg.yield %37 : f32
}
...
```


AMD
together we advance_

[Public]

3. Fusions

3.2 Output fusion

```
scf.for %arg4 = %c0_3 to %c20 step %c1_2 {
  rock.stage { ... } {name = "GlobalRead"}
  rock.stage { ... } {name = "LDSWrite"}
  rock.lds_barrier
  rock.stage { ... } {name = "MMA"}
  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}
...

rock.threadwise_read_into [...](%31) [%6, %13, %15, %5] -> %29 :
  memref<1x16x320xf32> -> memref<4xf32, #regs>

...

linalg.generic {
  indexing_maps = [#map30, #map30, #map30],
  iterator_types = ["parallel"]}
  ins(%25, %29 : memref<4xf32, #regs>, memref<4xf32, #regs>)
  outs(%28 : memref<4xf32, #regs>) {
^bb0(%in: f32, %in_6: f32, %out: f32):
  %37 = arith.addf %in, %in_6 : f32
  linalg.yield %37 : f32
}
...
```

After the bias we'll write %28 (the output tile) to global memory

The bias happens outside of the kernel loop, yet it's computed in registers, so no trip to global memory is performed here

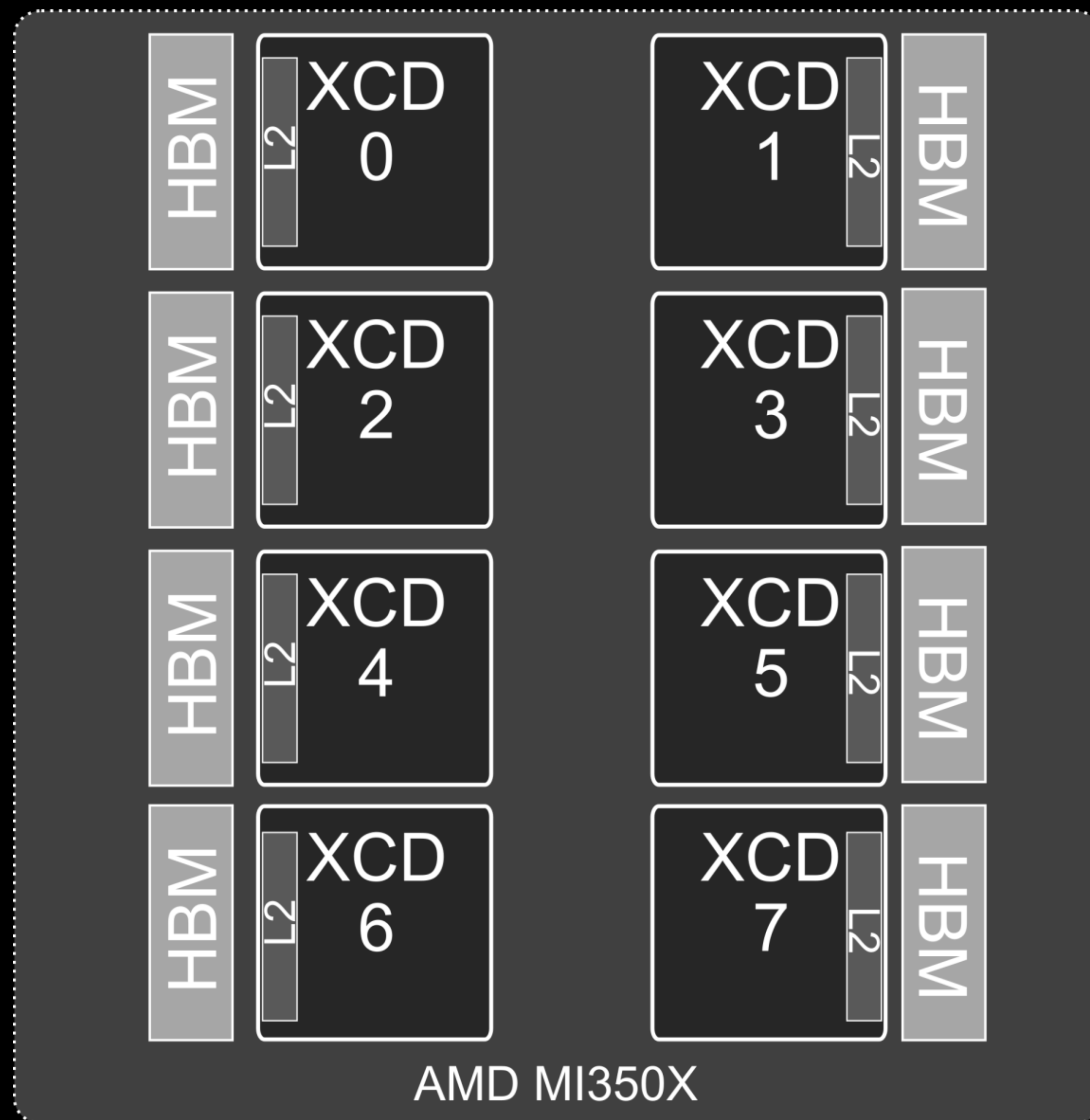
AMD
together we advance_

[Public]

4. Optimizations

XCDs remapping

- MI350X is divided into 8 XCDs
- Each XCD has:
  - Its own HBM slice
  - Its own L2 cache (4MB)
  - 32 CUs



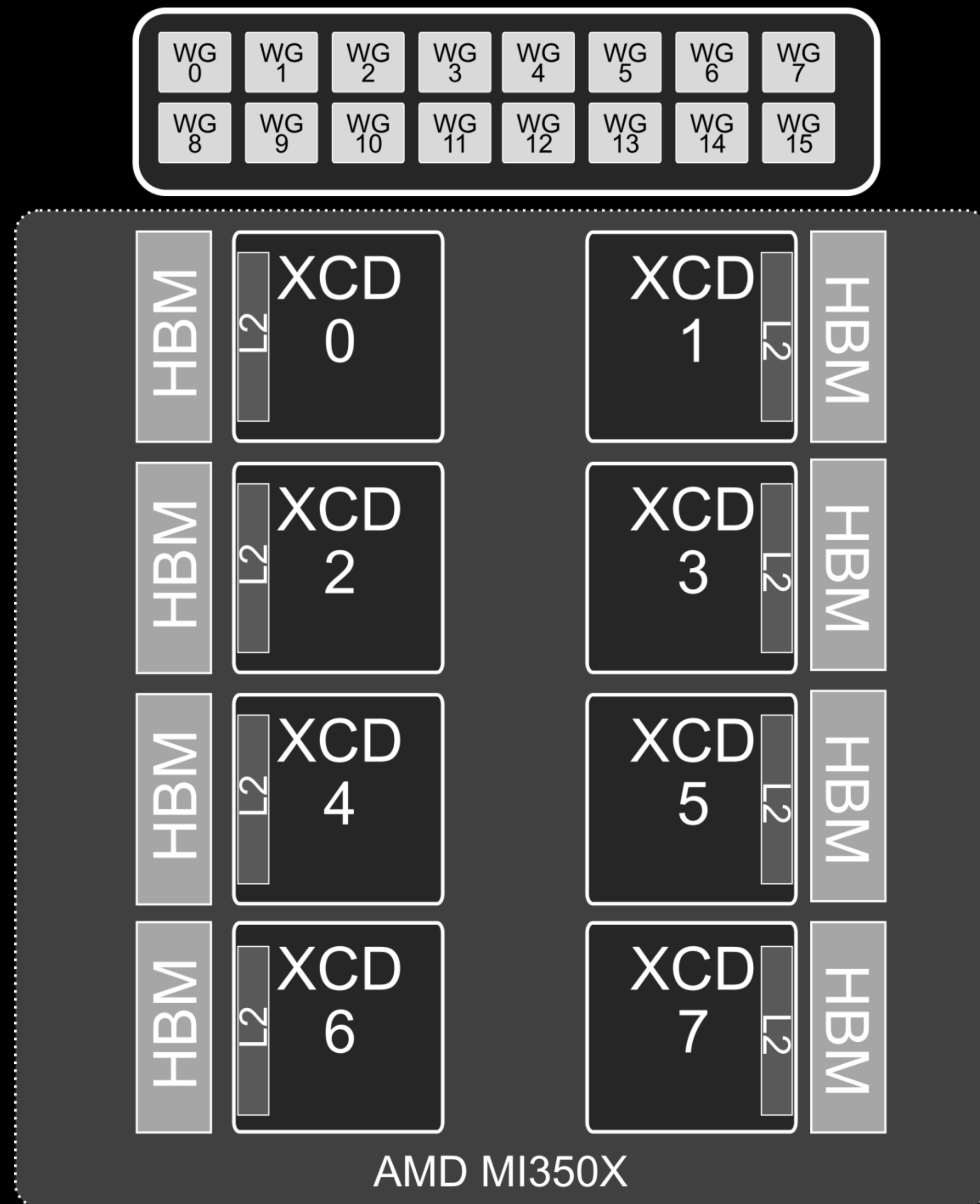
AMD
together we advance_

[Public]

4. Optimizations

XCDs remapping

- Let's imagine we launch a kernel with 16 blocks.
- Each of them will be scheduled to a different XCD in a round-robin fashion

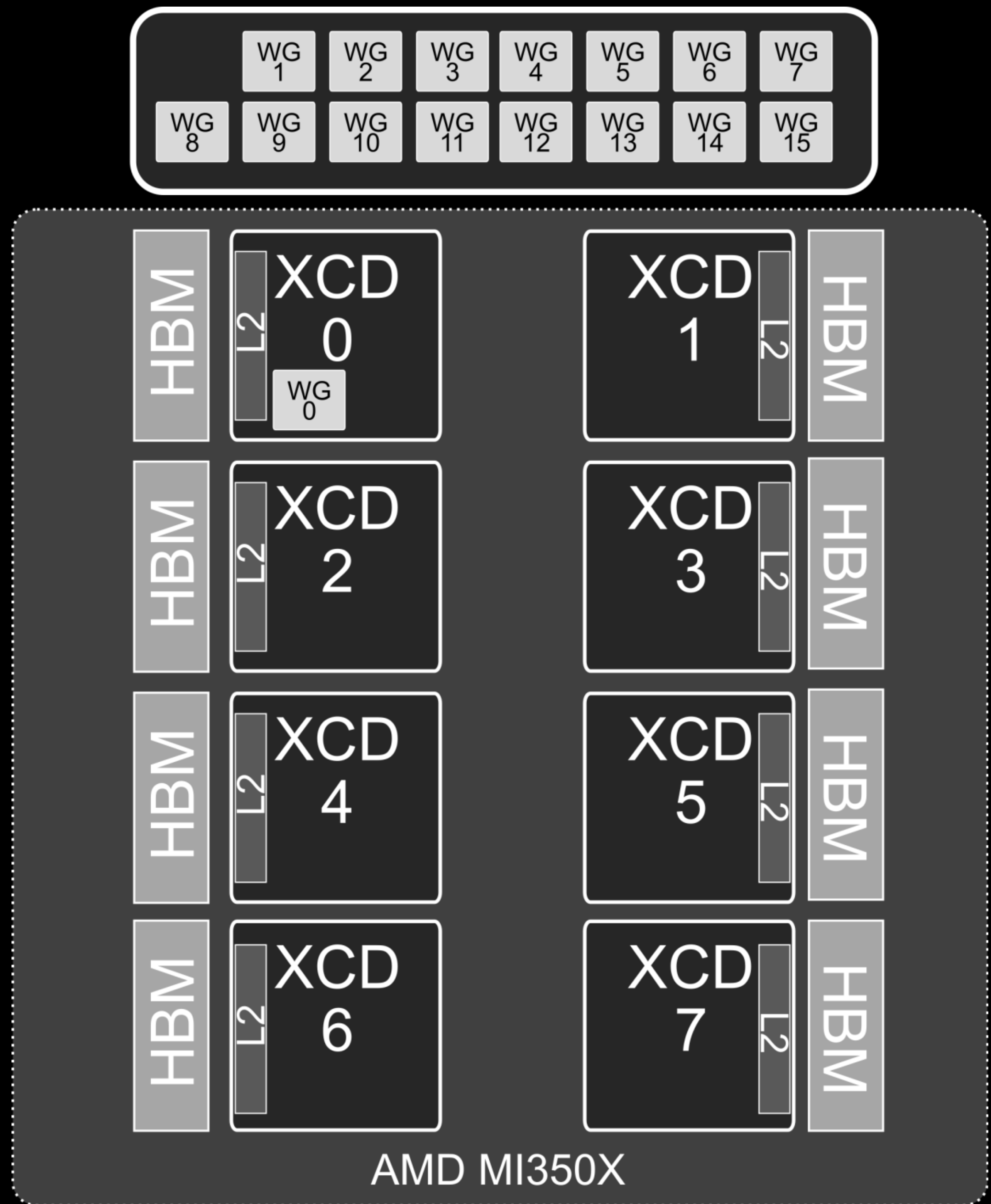


AMD
together we advance_

[Public]

4. Optimizations

XCDs remapping

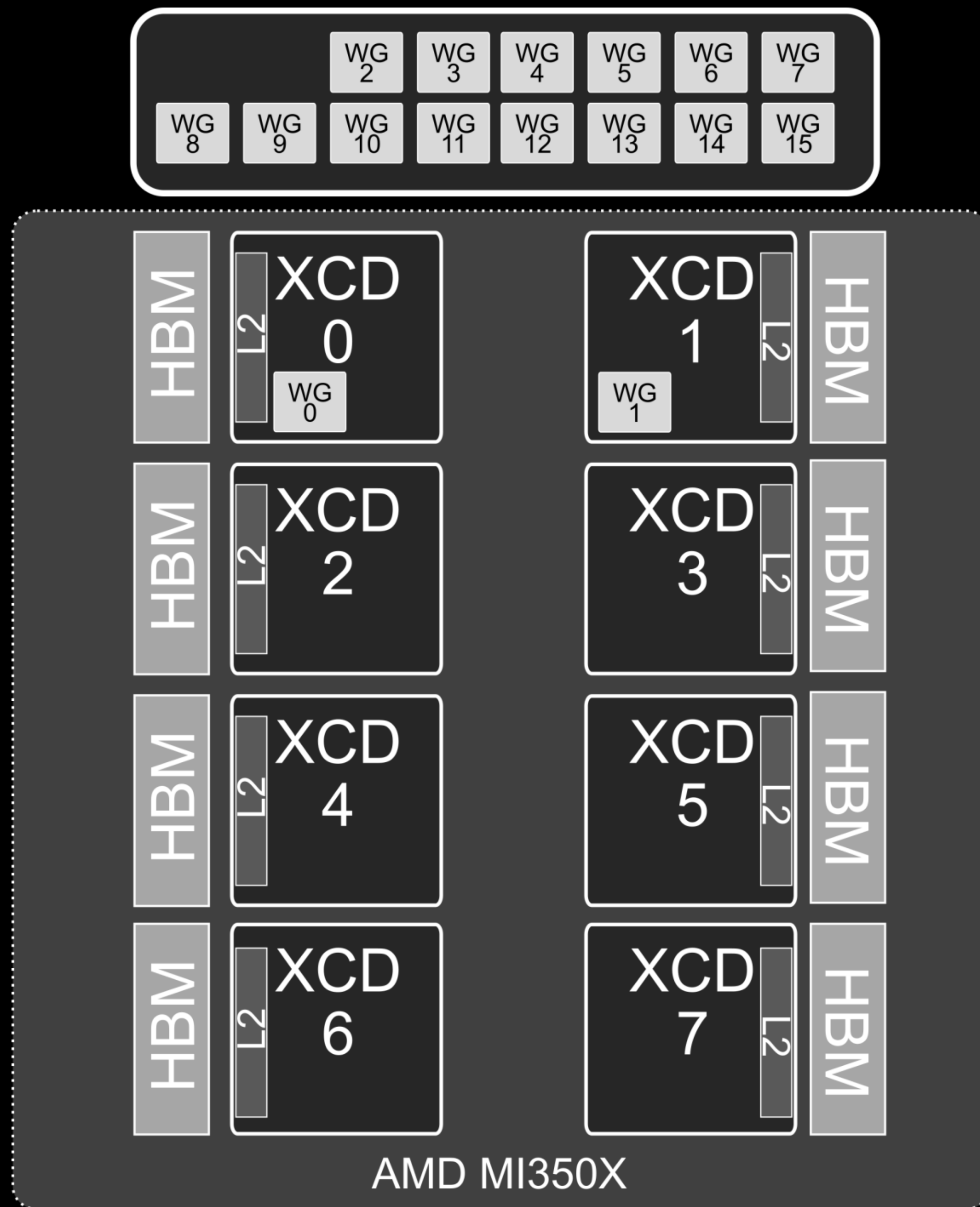


AMD
together we advance_

[Public]

4. Optimizations

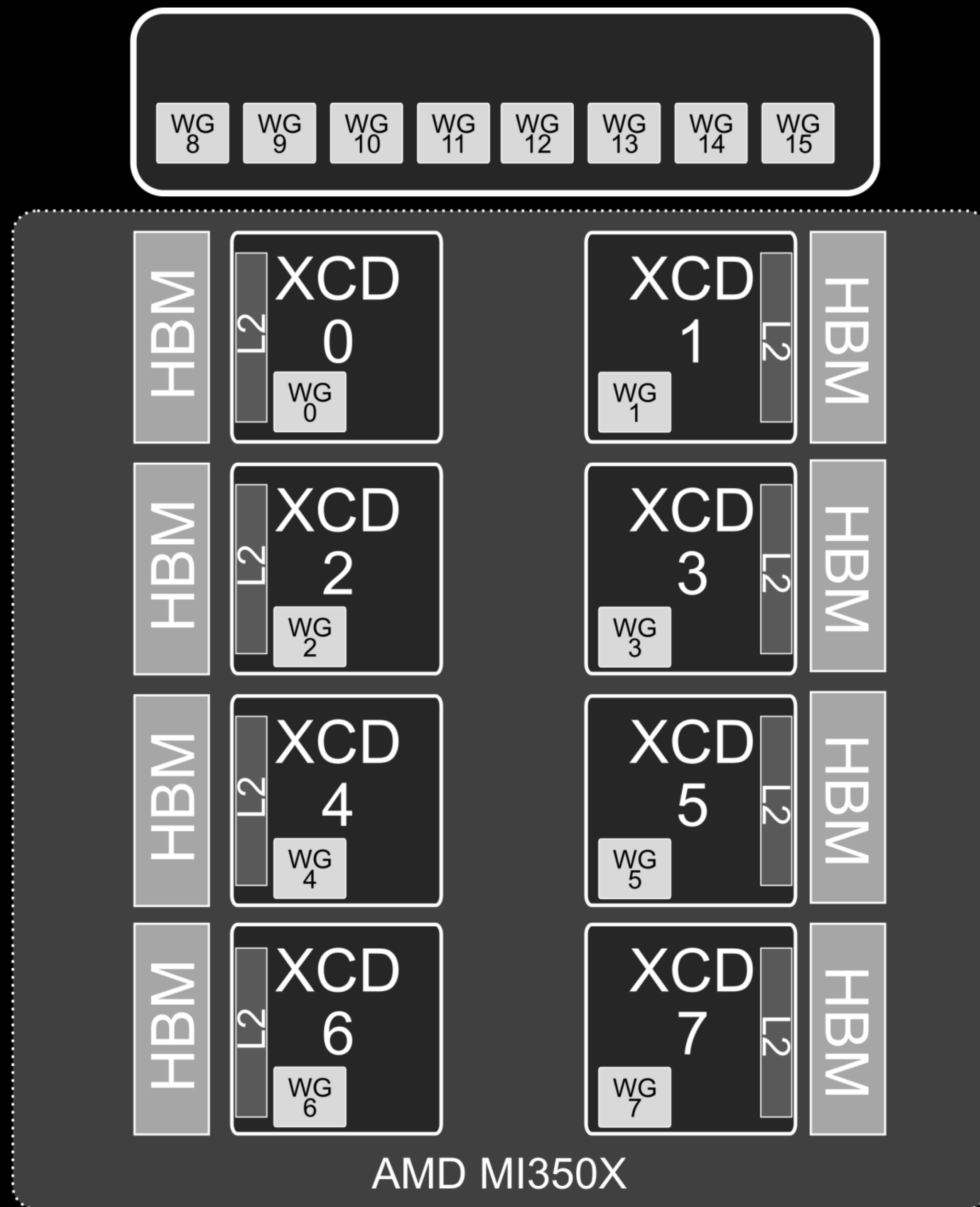
XCDs remapping



AMD together we advance_

[Public]

4. Optimizations
XCDs remapping



AMD together we advance_

[Public]

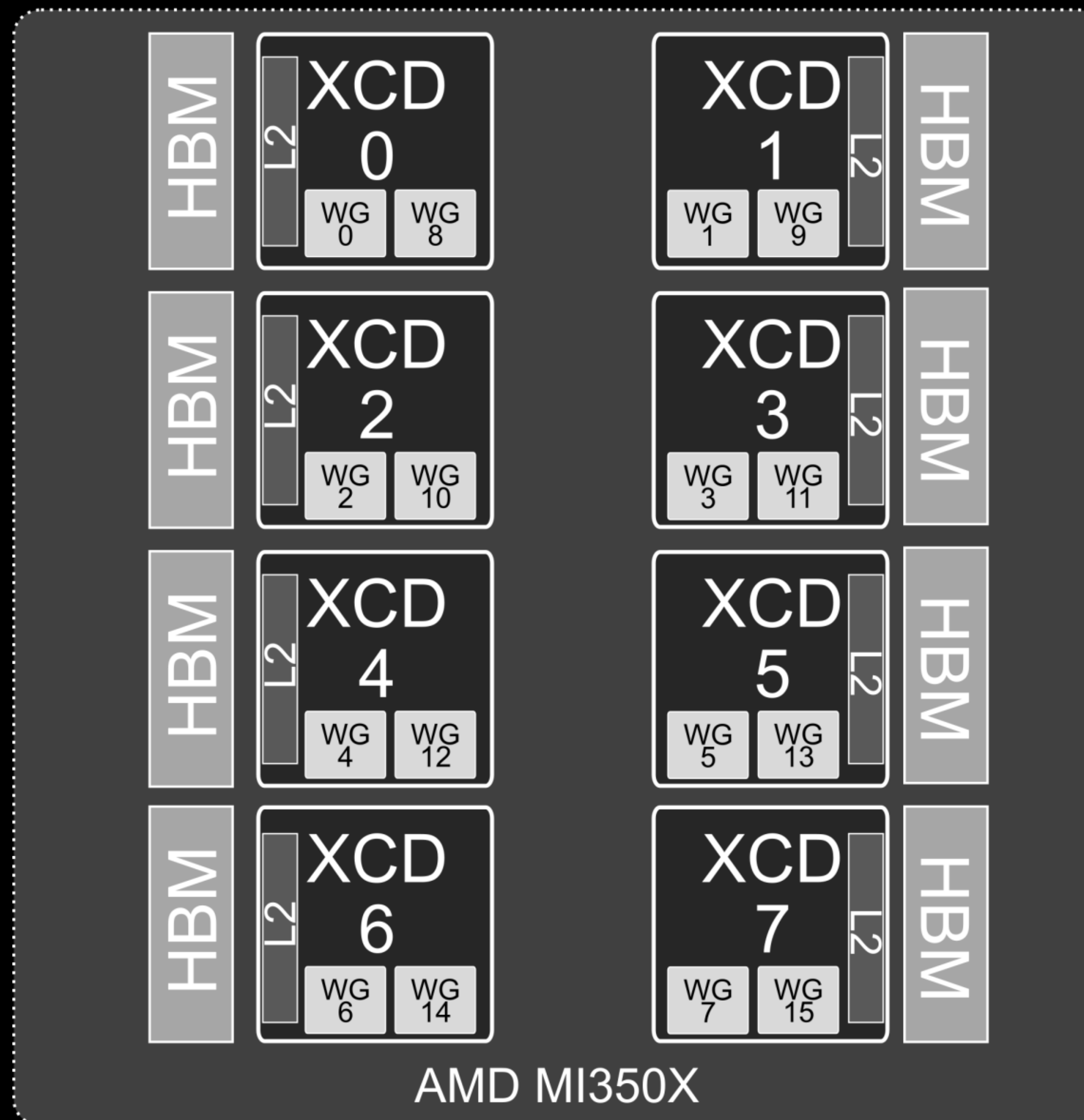
4. Optimizations

XCDs remapping

- Consecutive workgroups usually work on consecutive data.

  However, they are scheduled far away (for instance, WG0 and WG1 are in different XCDs)

- This is very inefficient due to:
  - Inter XCD communication
  - L2 trashing



AMD
together we advance_

[Public]

4. Optimizations

XCDs remapping

- We can’t change how the workgroups are mapped to the XCDs
- But can remap our workgroup id, so that each workgroup works with another part of the data, such that the data locality is maximized

Let’s assume:

- GPU with 4 XCD and 32 CUs
- A matrix that is subdivided in 6x6 sub-tiles
- No remapping would look like this:

0	2	0	2	0	2
1	3	1	3	1	3
2	0	2	0	2	0
3	1	3	1	3	1
0	2	0	2	0	2
1	3	1	3	1	3

(Each number means that such sub-tile is assigned to the XCD with that ID)

AMD
together we advance_

[Public]

4. Optimizations

XCDs remapping

- Our remapping consist on creating groups
- The group size is determined by a heuristic:

$$groupSize = \sqrt{\frac{numCU}{numXCD} * \frac{bitWidthOut}{bitWidthIn}} = \sqrt{\frac{32}{4} * \frac{32}{32}} = 2$$

0	0	3	3	2	2
0	0	3	3	2	2
1	1	0	0	3	3
1	1	0	0	3	3
2	2	1	1	0	1
2	2	1	1	2	3

AMD
together we advance_

[Public]

4. Optimizations

XCDs remapping

- Our remapping consist on creating groups
- The group size is determined by an heuristic:

$$groupSize = \sqrt{\frac{numCU}{numXCD} * \frac{bitWidthOut}{bitWidthIn}} = \sqrt{\frac{32}{4} * \frac{32}{32}} = 2$$

0	0	3	3	2	2
0	0	3	3	2	2
1	1	0	0	3	3
1	1	0	0	3	3
2	2	1	1	0	1
2	2	1	1	2	3

- NOTE: This mapping is used for all kernels. We could specialize remapping for specific kernels or use cases (e.g., KV cache)

AMD
together we advance_

[Public]

4. Optimizations

DirectToLDS

How does the picture look like? (without DirectToLDS optimization)

```
scf.for {
  rock.stage {
    rock.threadwise_read_into
    rock.threadwise_read_into
  } {name = "GlobalRead"}

  rock.stage {
    rock.threadwise_write_all
    rock.threadwise_write_all
  } {name = "LDSWrite"}

  rock.lds_barrier

  rock.stage {
    rock.blockwise_gemm_accel
  } {name = "MMA"}

  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}

rock.threadwise_write_all
```

Reads A and B tile from global memory into registers

AMD
together we advance_

4. Optimizations

DirectToLDS

How does the picture look like? (without DirectToLDS optimization)

```
scf.for {
  rock.stage {
    rock.threadwise_read_into
    rock.threadwise_read_into
  } {name = "GlobalRead"}

  rock.stage {
    rock.threadwise_write_all
    rock.threadwise_write_all
  } {name = "LDSWrite"}

  rock.lds_barrier

  rock.stage {
    rock.blockwise_gemm_accel
  } {name = "MMA"}

  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}

rock.threadwise_write_all
```

Copies A and B tile from tiles into LDS

AMD
together we advance_

[Public]

4. Optimizations

DirectToLDS

How does the picture look like? (without DirectToLDS optimization)

```
scf.for {
  rock.stage {
    rock.threadwise_read_into
    rock.threadwise_read_into
  } {name = "GlobalRead"}

  rock.stage {
    rock.threadwise_write_all
    rock.threadwise_write_all
  } {name = "LDSWrite"}

  rock.lds_barrier

  rock.stage {
    rock.blockwise_gemm_accel
  } {name = "MMA"}
  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}

rock.threadwise_write_all
```

Computes matmul by reading from LDS and writing into registers

AMD
together we advance_

[Public]

4. Optimizations

DirectToLDS

How does the picture look like? (without DirectToLDS optimization)

```
scf.for {
  rock.stage {
    rock.threadwise_read_into
    rock.threadwise_read_into
  } {name = "GlobalRead"}

  rock.stage {
    rock.threadwise_write_all
    rock.threadwise_write_all
  } {name = "LDSWrite"}

  rock.lds_barrier

  rock.stage {
    rock.blockwise_gemm_accel
  } {name = "MMA"}

  rock.lds_barrier
} {pipeline = #rock.pipeline<2>}
```

`rock.threadwise_write_all` Reads from registers and writes the output back into global memory

AMD
together we advance_

4. Optimizations

DirectToLDS

Modern AMD GPUs have instructions that allows you to load from global memory to LDS directly, so we can have something like:

```
scf.for {
  rock.stage {
    rock.threadwise_read_into
    rock.threadwise_read_into
    rock.yield
  } {name = "GlobalRead"}

  rock.lds_barrier

  rock.stage {
    rock.blockwise_gemm_accel
    rock.yield
  } {name = "MMA"}

  rock.lds_barrier

} {pipeline = #rock.pipeline<2>}
rock.threadwise_write_all
```

This now loads from global memory into LDS directly (DirectToLDS)

This is the same, load from LDS and store to registers

This is the same, loads from registers and stores to global memory

AMD
together we advance_

4. Optimizations

DirectToLDS

Modern AMD GPUs have instructions that allows you to load from global memory to LDS directly, so we can have something like:

`rock.threadwise_read_into`

`amdgpu.gather_to_lds`

`rocdl.load.to.lds`

global_load_lds_dword
global_load_lds_dwordx3
global_load_lds_dwordx4

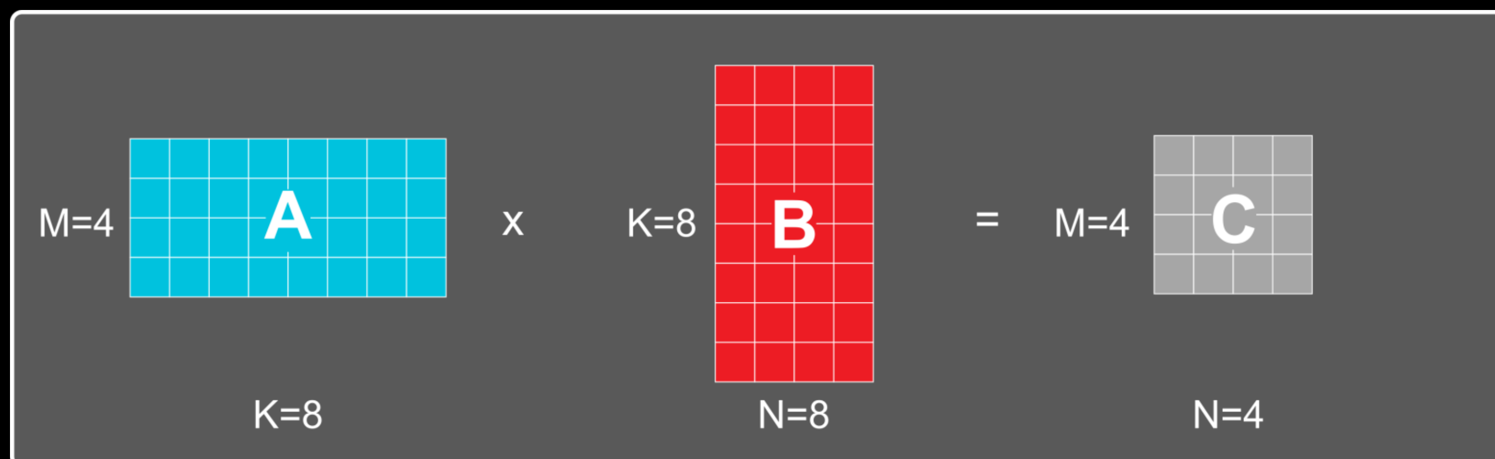
The rock op lowers to amdgpu, rocdl and later to the DirectToLDS assembly load instruction

AMD
together we advance_

[Public]

4. Optimizations

GEMM specific: SplitK



Let's imagine a GPU with 8 CUs and a matmul kernel with:
M=N=4 and K=8

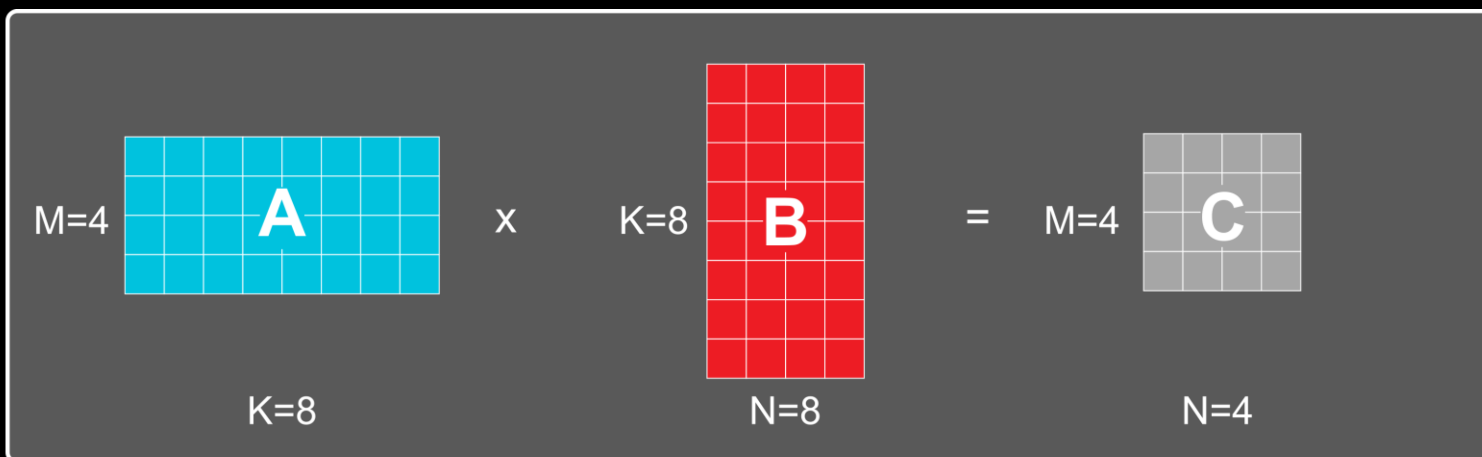
Let's assume:
Tile sizes for M and N is 4.

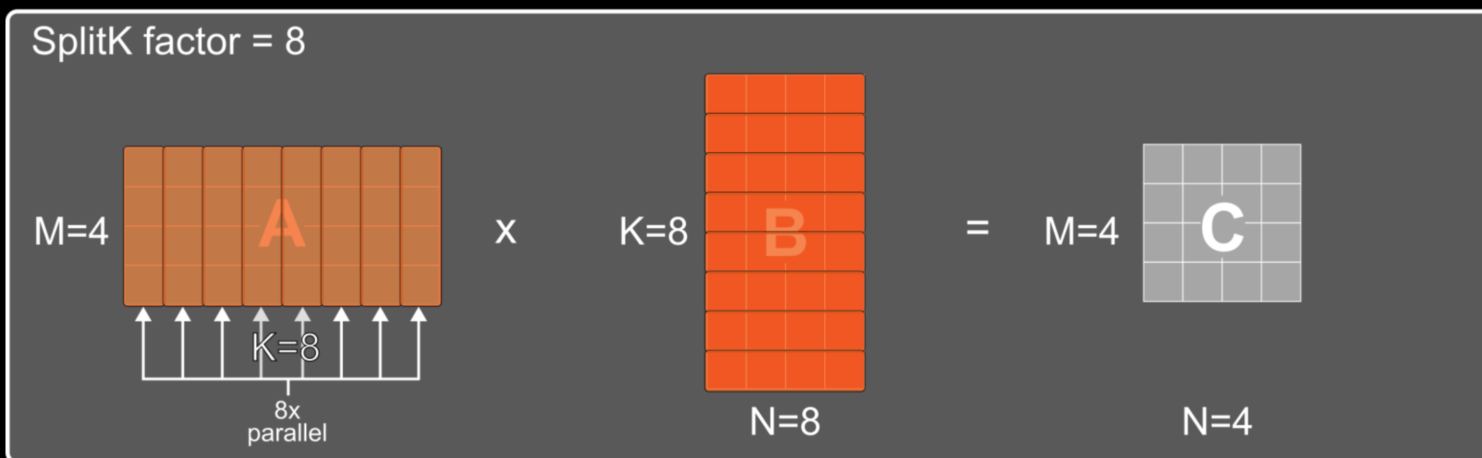
Therefore, we can only use 1 CU (out of 8) because M and N dimensions are small

[Public]

4. Optimizations

GEMM specific: SplitK





Let’s imagine a GPU with 8 CUs and a matmul kernel with: M=N=4 and K=8

With SplitK=8 we split the K dimension 8 times, thus allowing 8x more parallelism, using the 8 CUs effectively

AMD
together we advance_

[Public]

5. Evaluation

Software:
- rocMLIR
- CKTile

We used ROCm 7.2
We show the average over 3 independent runs

We evaluate different Attention kernels

Exhaustive tuning mode was used in rocMLIR

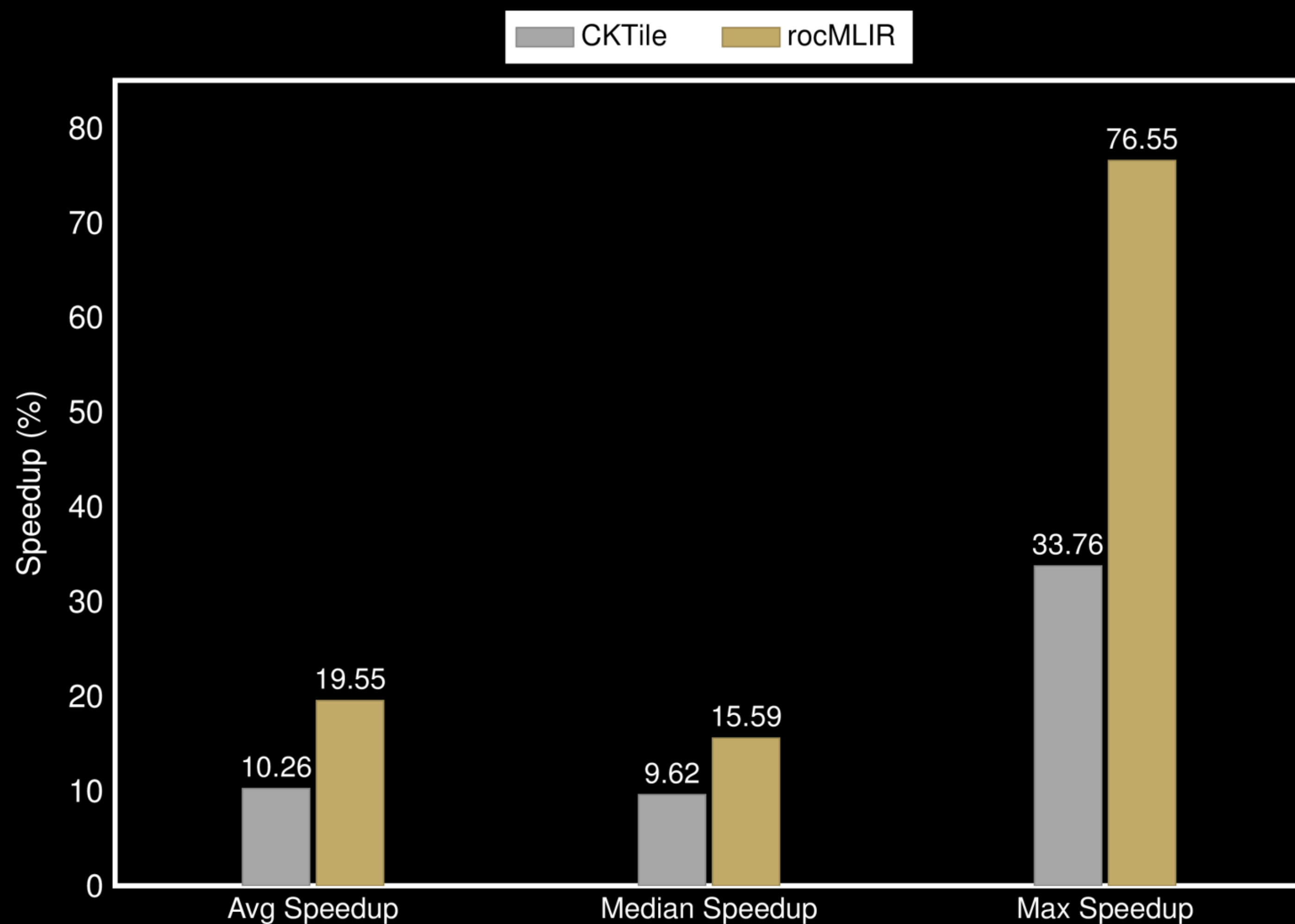
Hardware: MI300X (gfx942)

AMD
together we advance_

[Public]

5. Evaluation (Attention)

MI300X



397 configs tested:

- CKTile is faster 56% of the configs
- rocMLIR is faster 30% of the configs

AMD
together we advance_

[Public]

6. Conclusions and future work

Conclusions

- rocMLIR combines upstream MLIR dialects with downstream GPU-focused dialects:
- Our work upstream:
  - We have fixed several bugs in different parts of MLIR
  - We have added support for different `amdgpu` and `rocdl` ops
- Upstreaming candidates:
  - Transforms: The main benefit is exploited in the rock dialect, so upstreaming does not seem trivial
  - Fusions: Too much dependent on our internal IR design, upstreaming not appropriate

- rocMLIR is open source, check it out: https://github.com/ROCm/rocMLIR

AMD
together we advance_

[Public]

6. Conclusions and future work

Conclusions

- rocMLIR combines upstream MLIR dialects with downstream GPU-focused dialects
- Our work upstream:
  - We have fixed several bugs in different parts of MLIR
  - We have added support for different `amdgpu` and `rocdl` ops
- Upstreaming candidates:
  - Transforms: The main benefit is exploited in the rock dialect, so upstreaming does not seem trivial
  - Fusions: Too much dependent on our internal IR design, upstreaming not appropriate

- rocMLIR is open source, check it out: https://github.com/ROCm/rocMLIR

We are hiring!

If you are interested, make sure to visit AMD's stand during the conference for more information

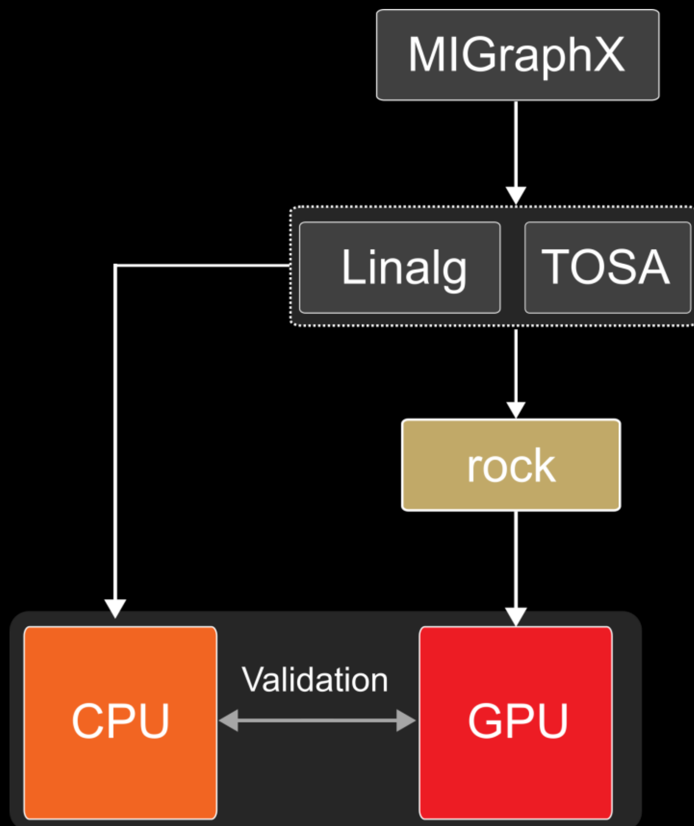
AMD
together we advance_

[Public]

6. Conclusions and future work

Future work

Integrating linalg as middle layer between graph IRs and `rock`



Using Triton as the backend of rocMLIR

AMD
together we advance_

AMD