



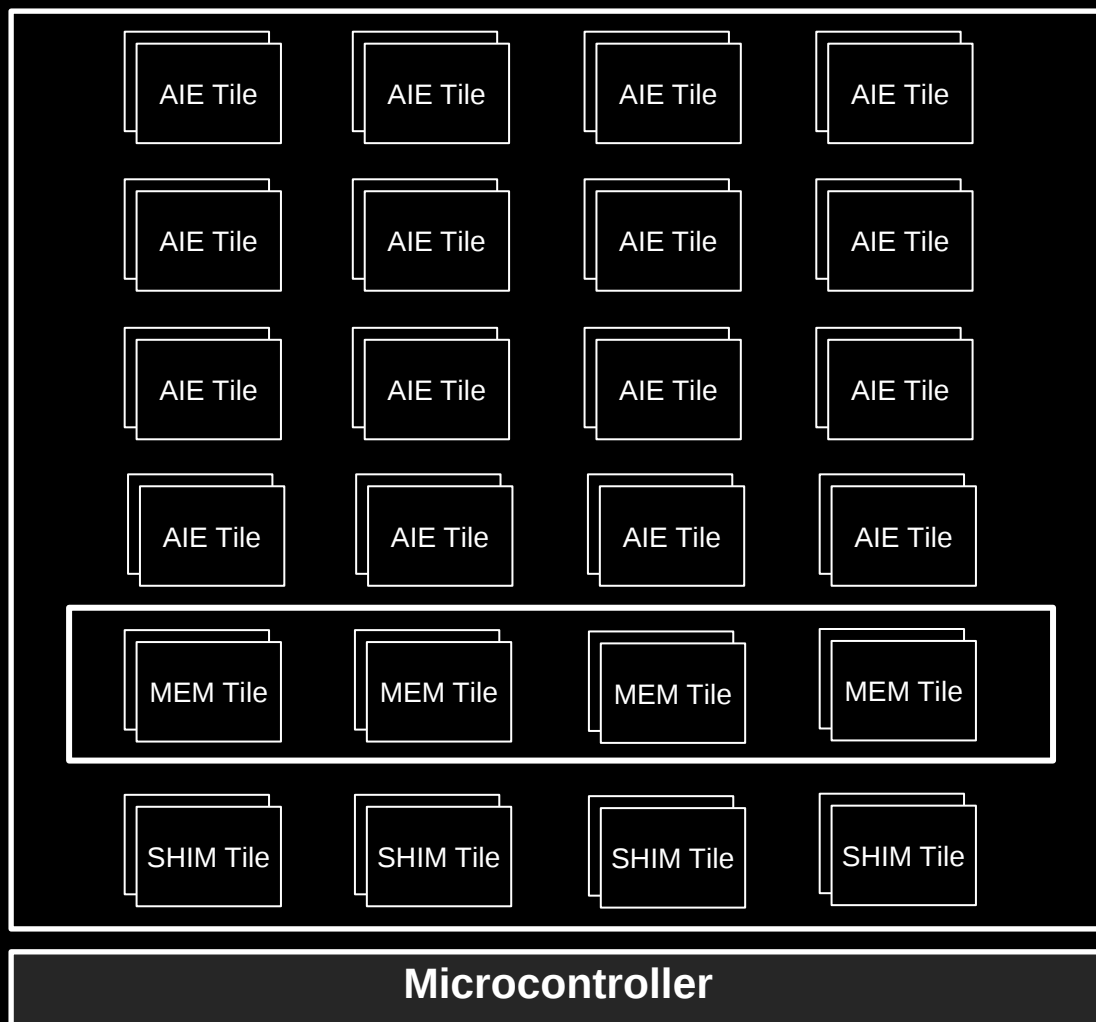
MLIR-based Data Tiling Design for Ryzen AI

Jorn Tuyls, Vivian Zhang, Abhishek Varma, James Newling, Maksim Levant, Nirvedh Meshram, Mahesh Ravishankar

MLIR-based Data Tiling Design for AI Engines

-
1. AI Engine Overview
 2. Data Packing through DMAs
 3. Mapping GEMM to the AI Engine
 4. Parallelization and Data Broadcasting Examples

XDNA AI Engine Overview



An array of VLIW SIMD high-performance processors that deliver up to 8X silicon compute density at 50% the power consumption of traditional programmable logic solutions.

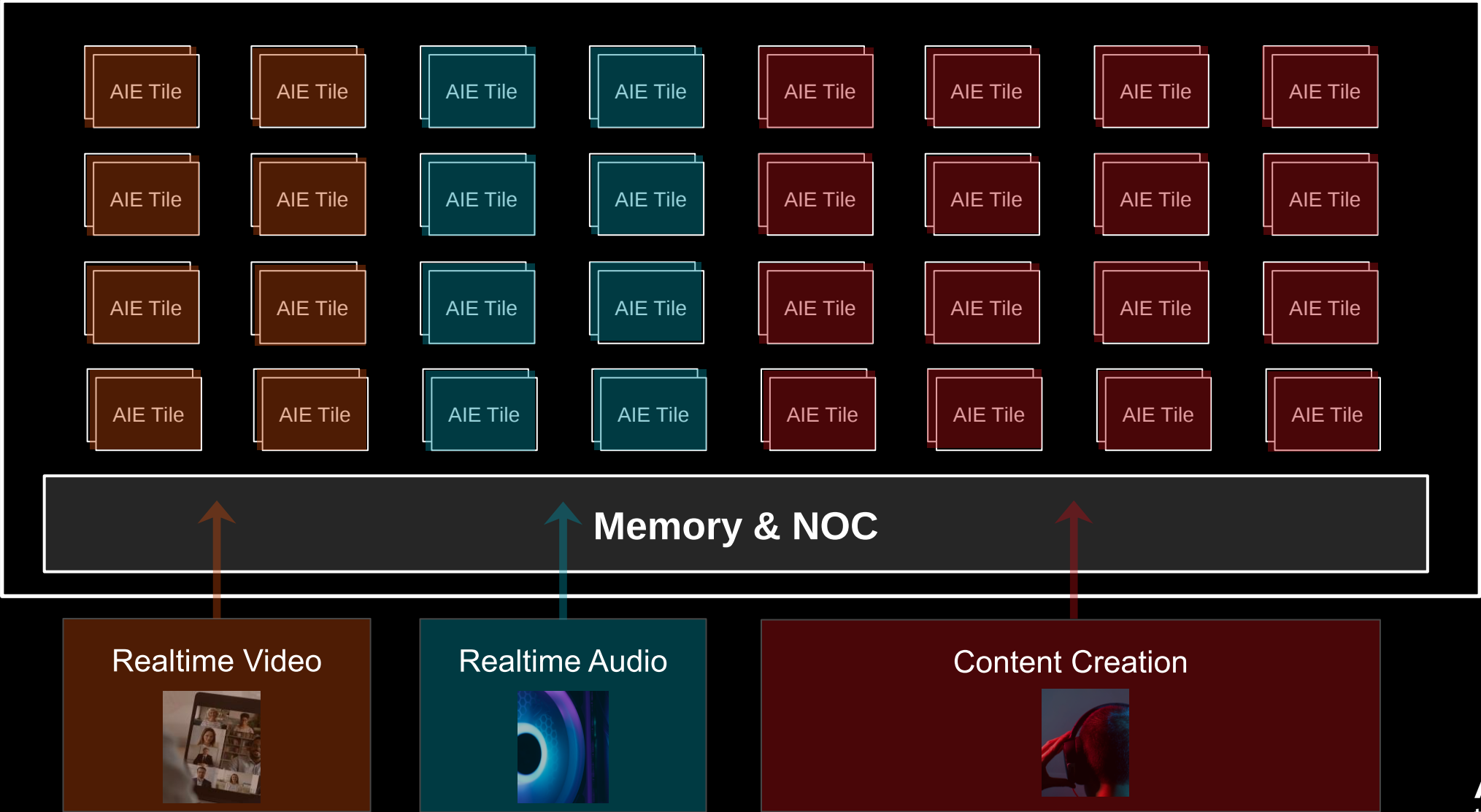
XDNA 1

- 5x4 array of AIE tiles
- 256 INT8 MACs / AIE tile
- 512 KB / MEM tile
- 10 TOPS on array (INT8, 1 GHz)

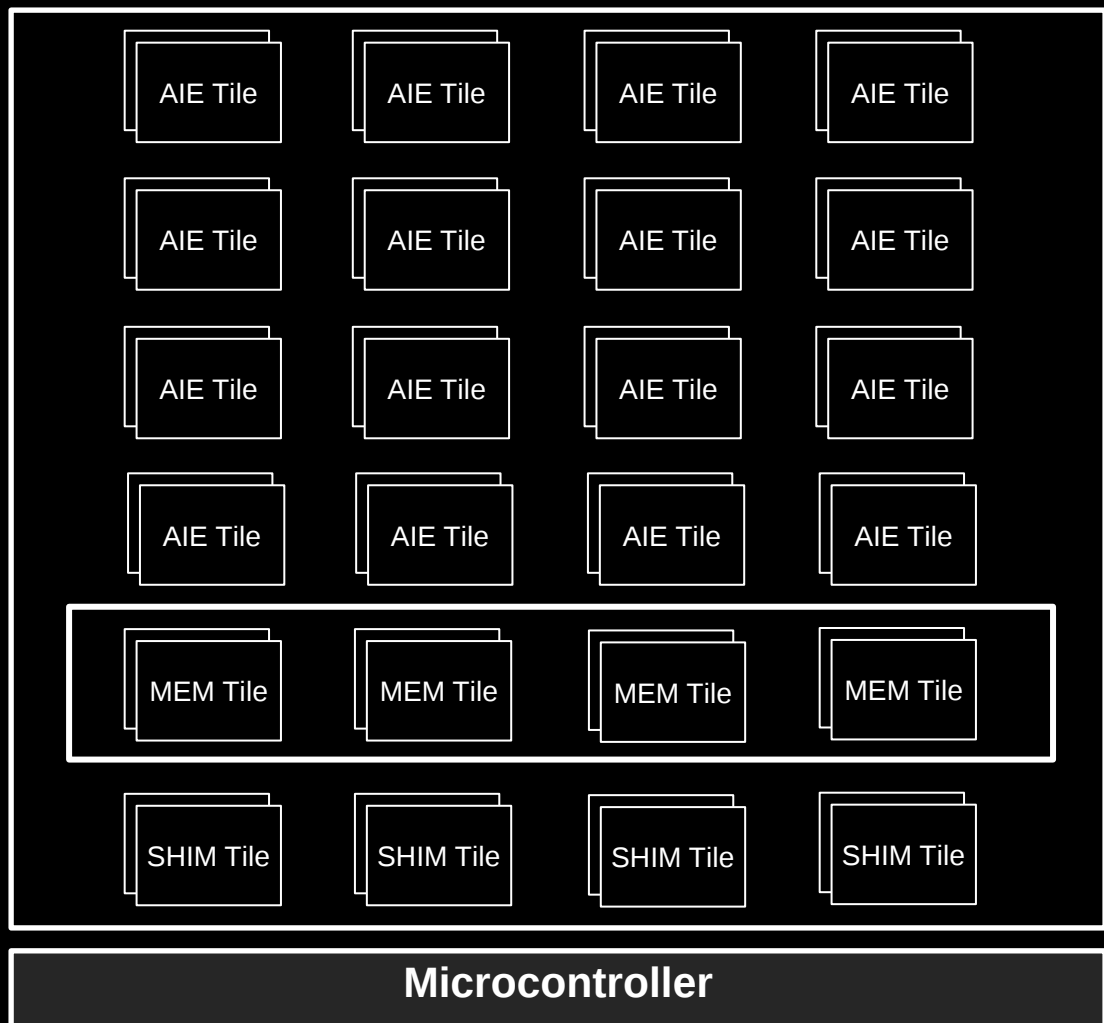
XDNA 2

- 8x4 array of AIE tiles
- 512 INT8 MACs / AIE tile
- 512 KB / MEM tile
- 32 TOPS on array (INT8, 1 GHz), up to 50+ TOPS

Partitioning



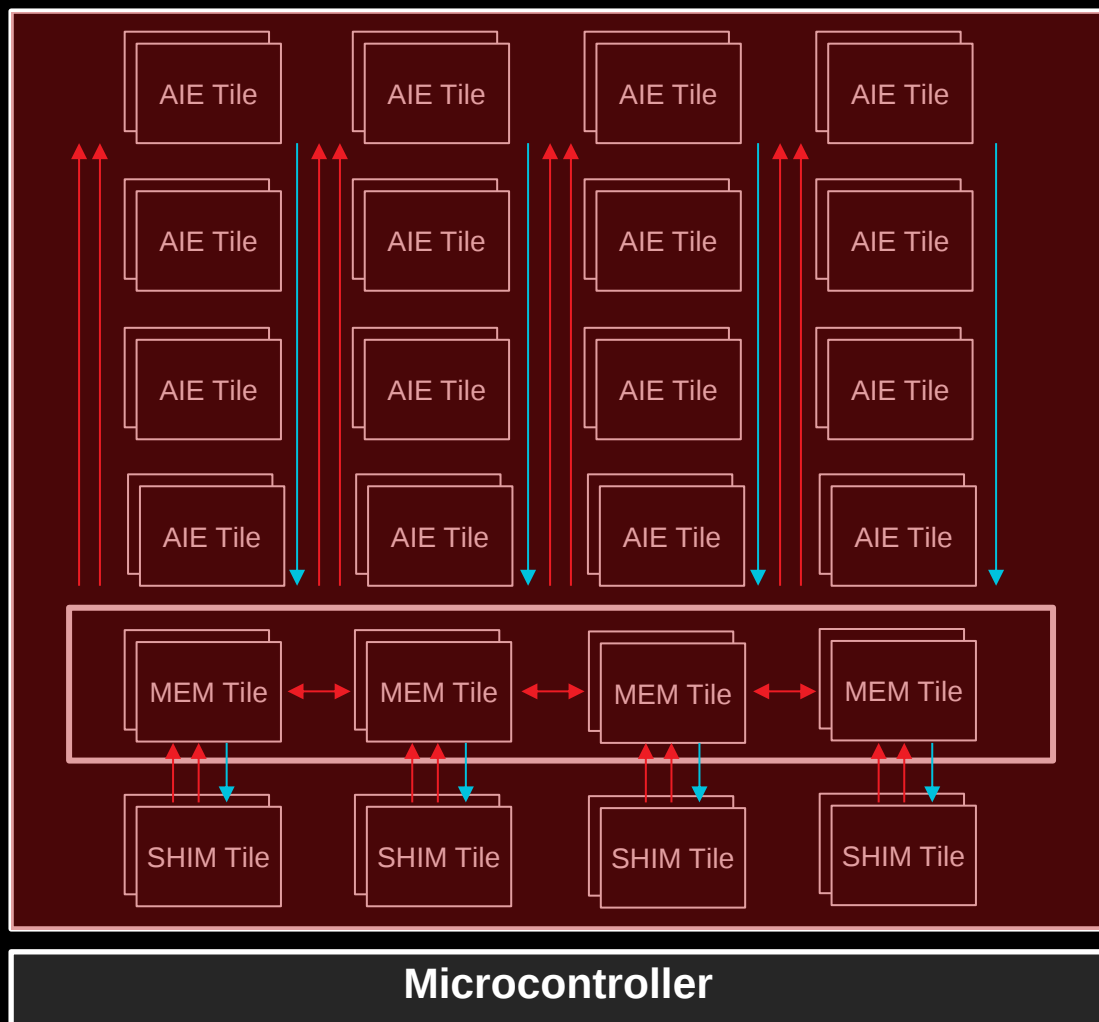
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

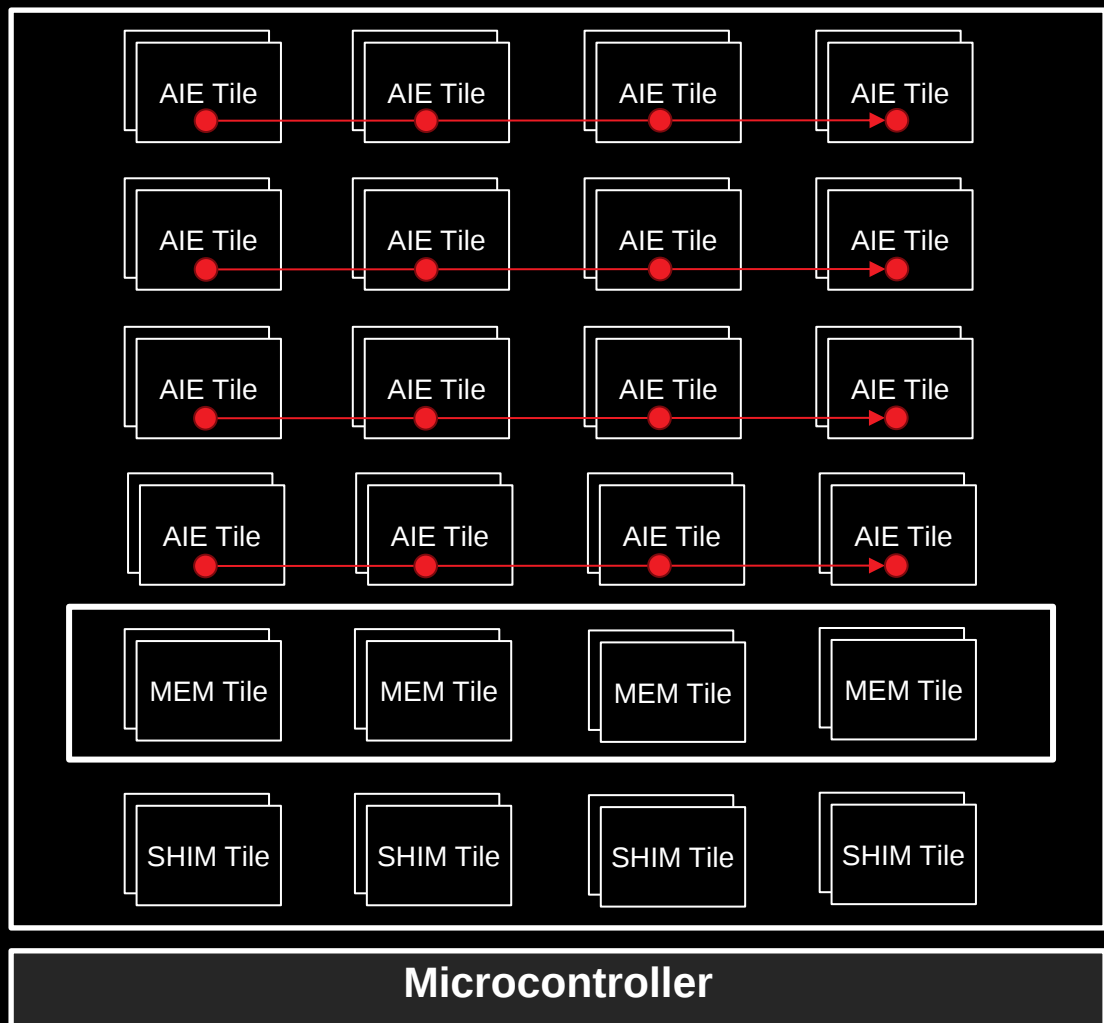
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

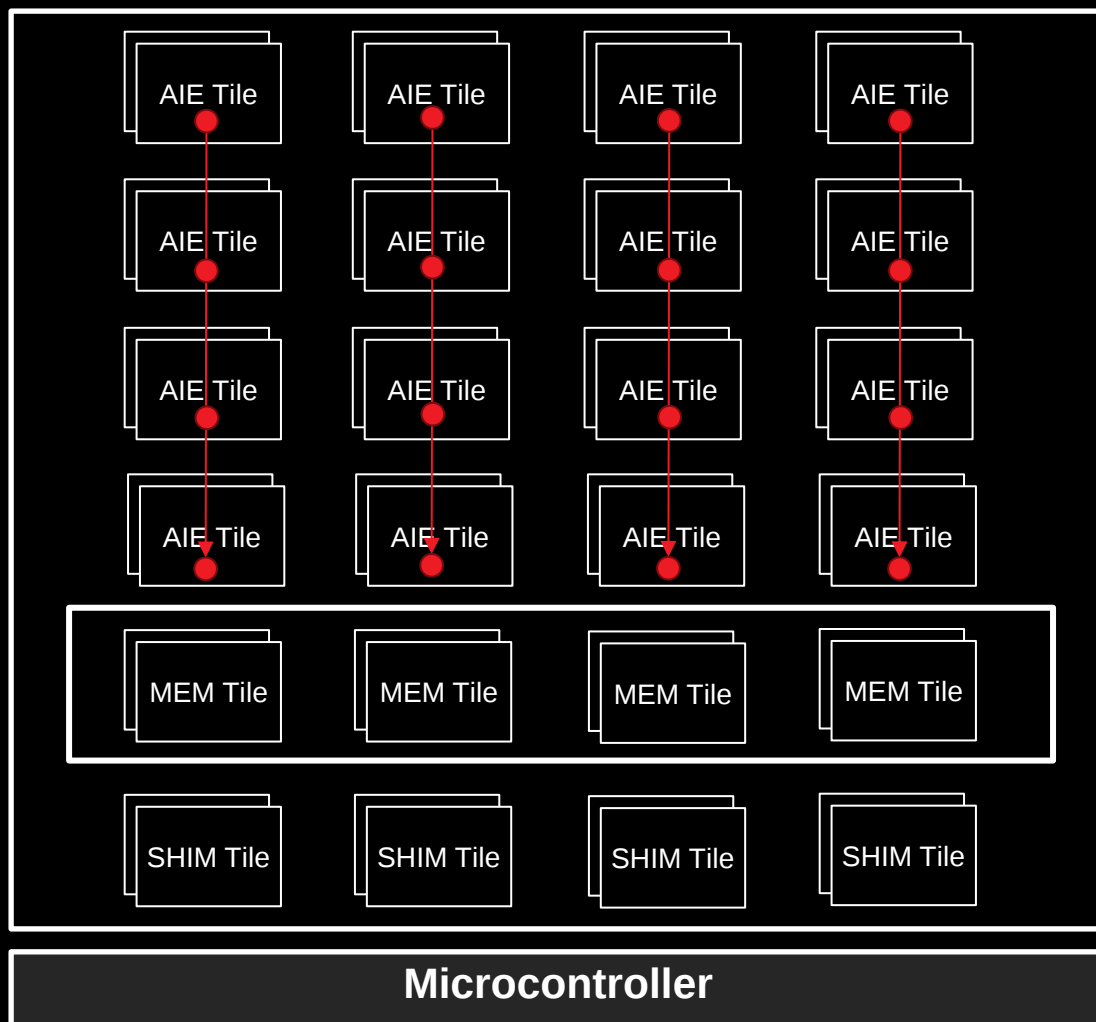
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

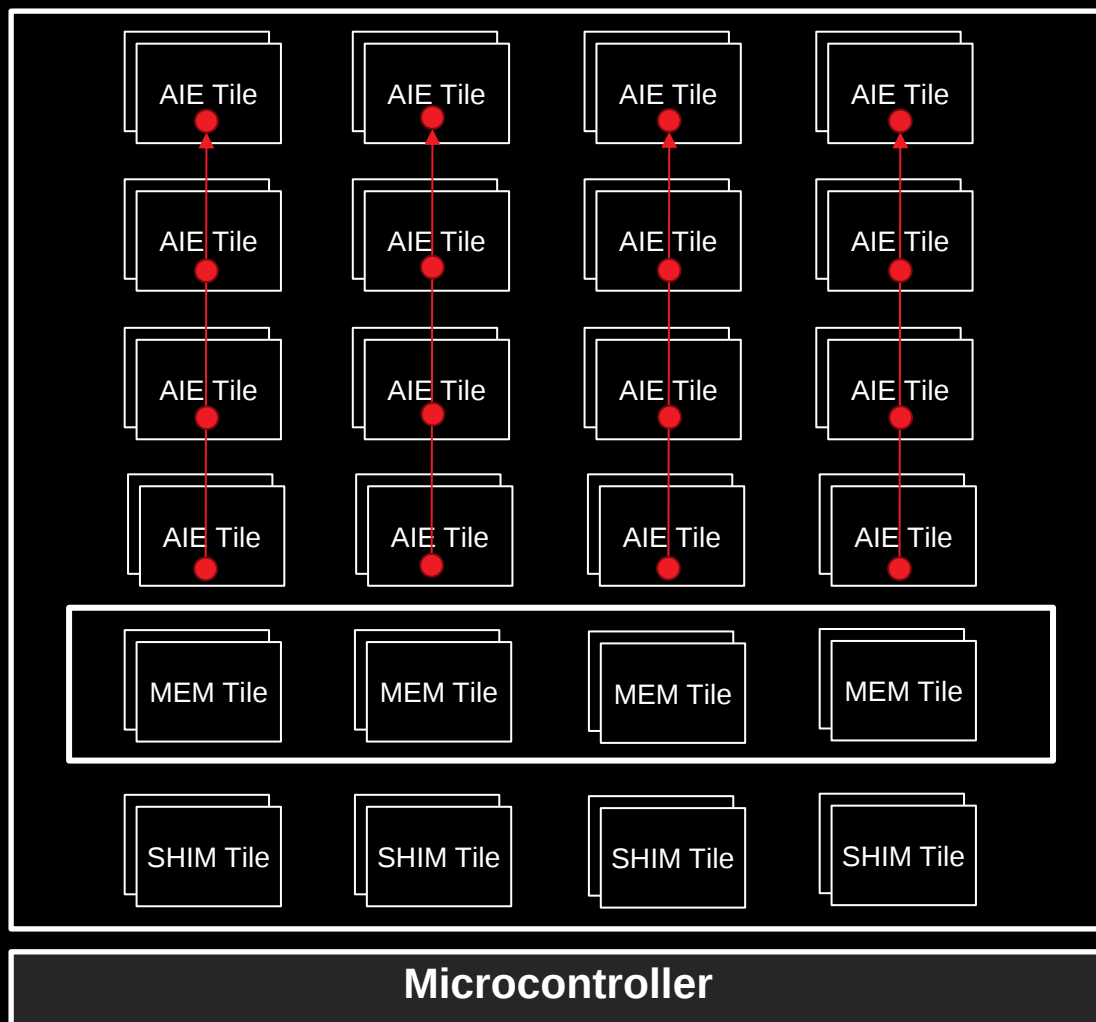
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

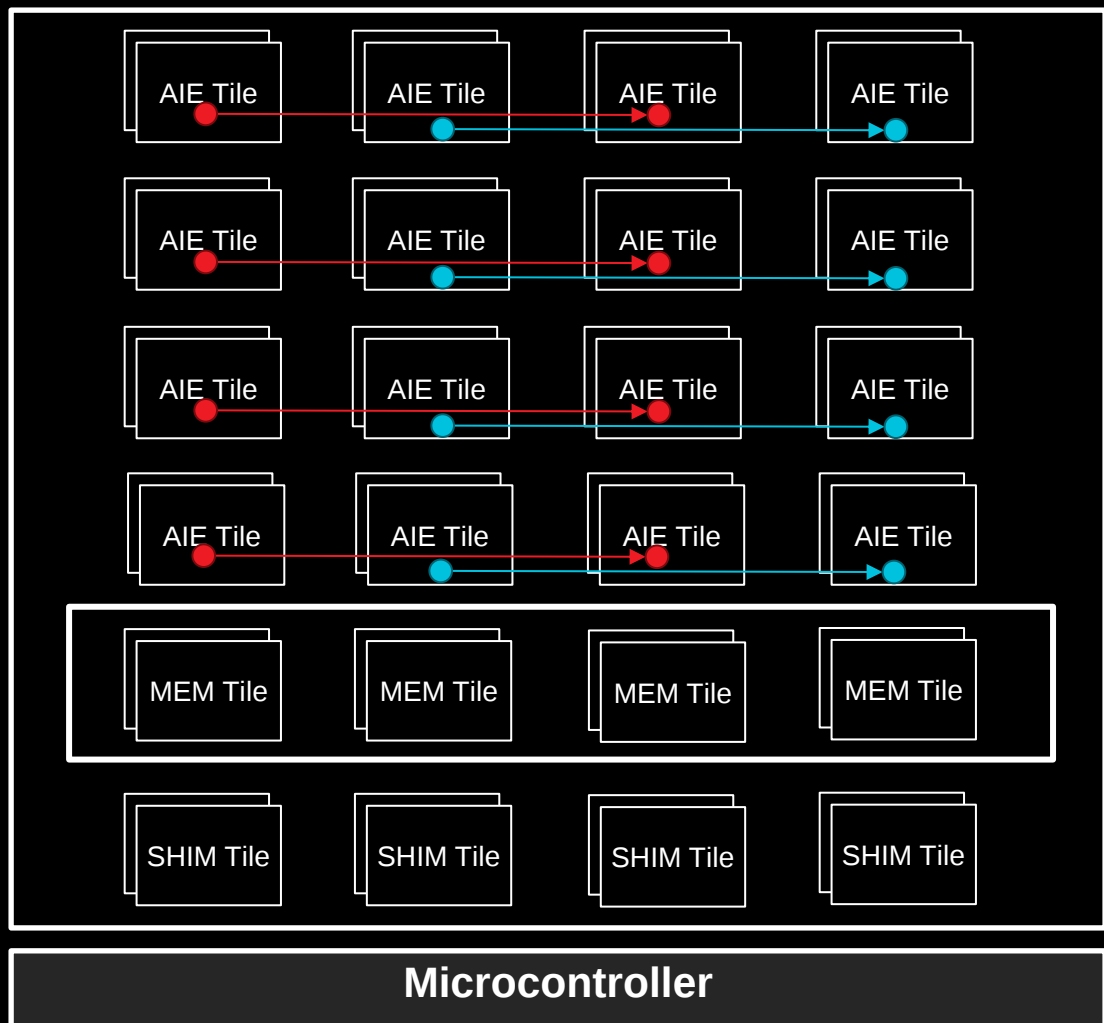
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

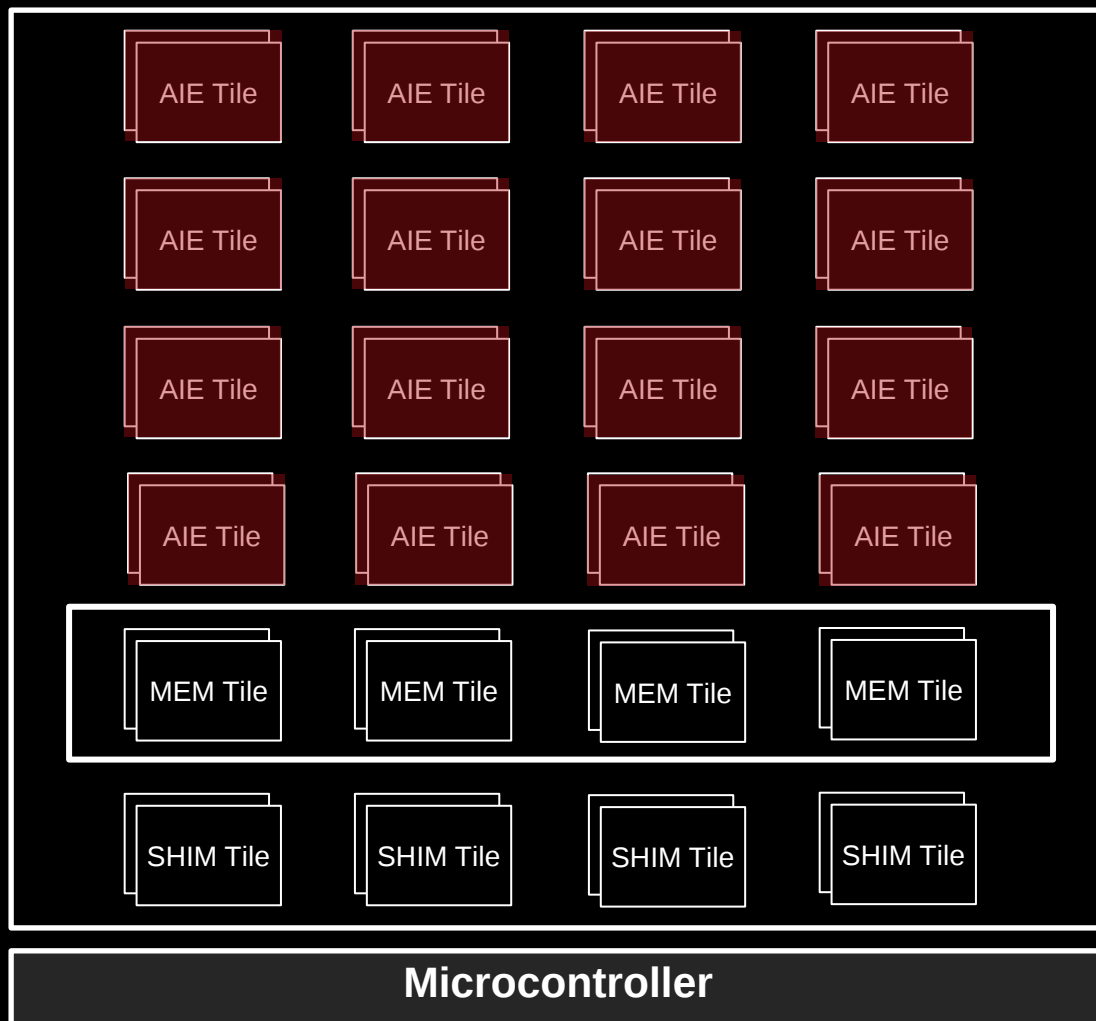
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

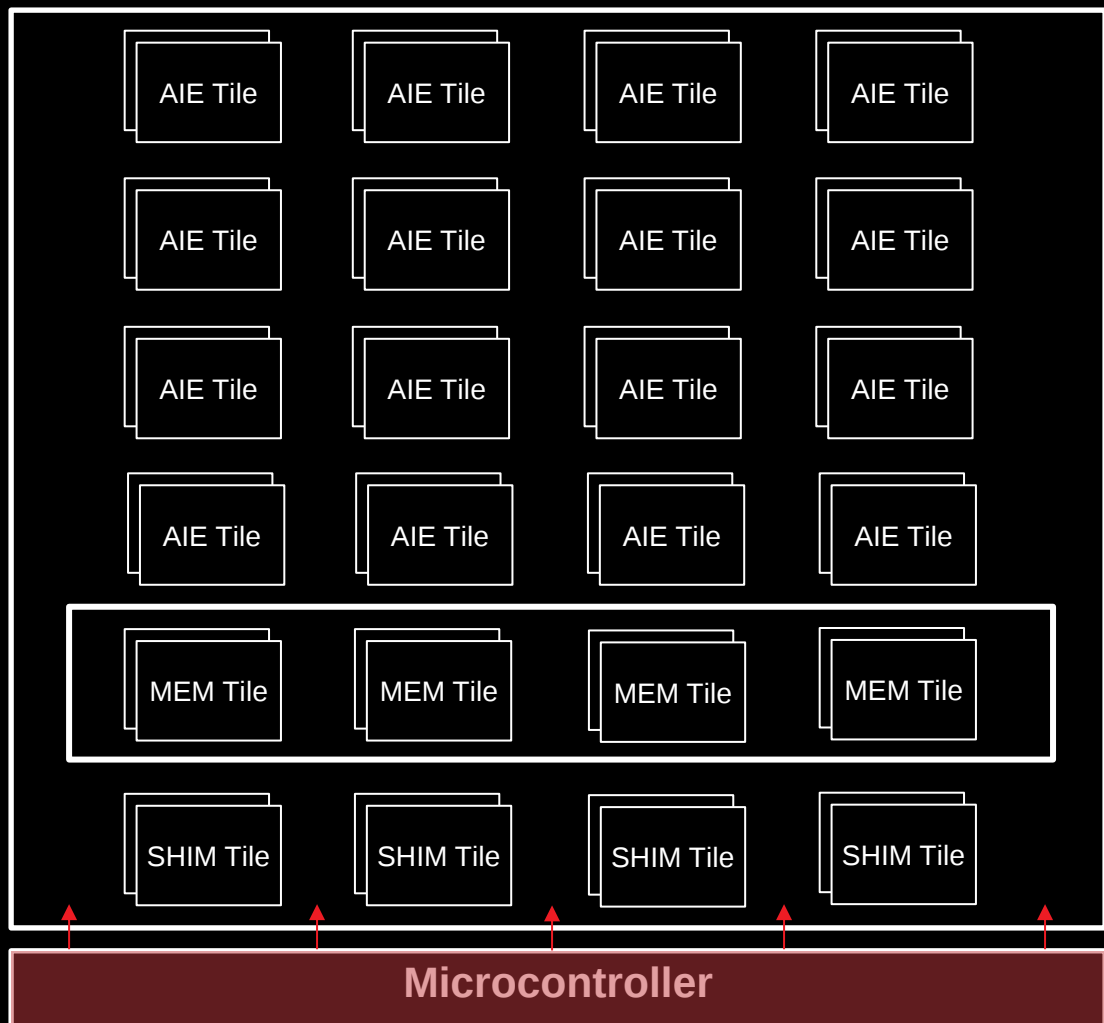
XDNA AI Engine Overview



How to program?

- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

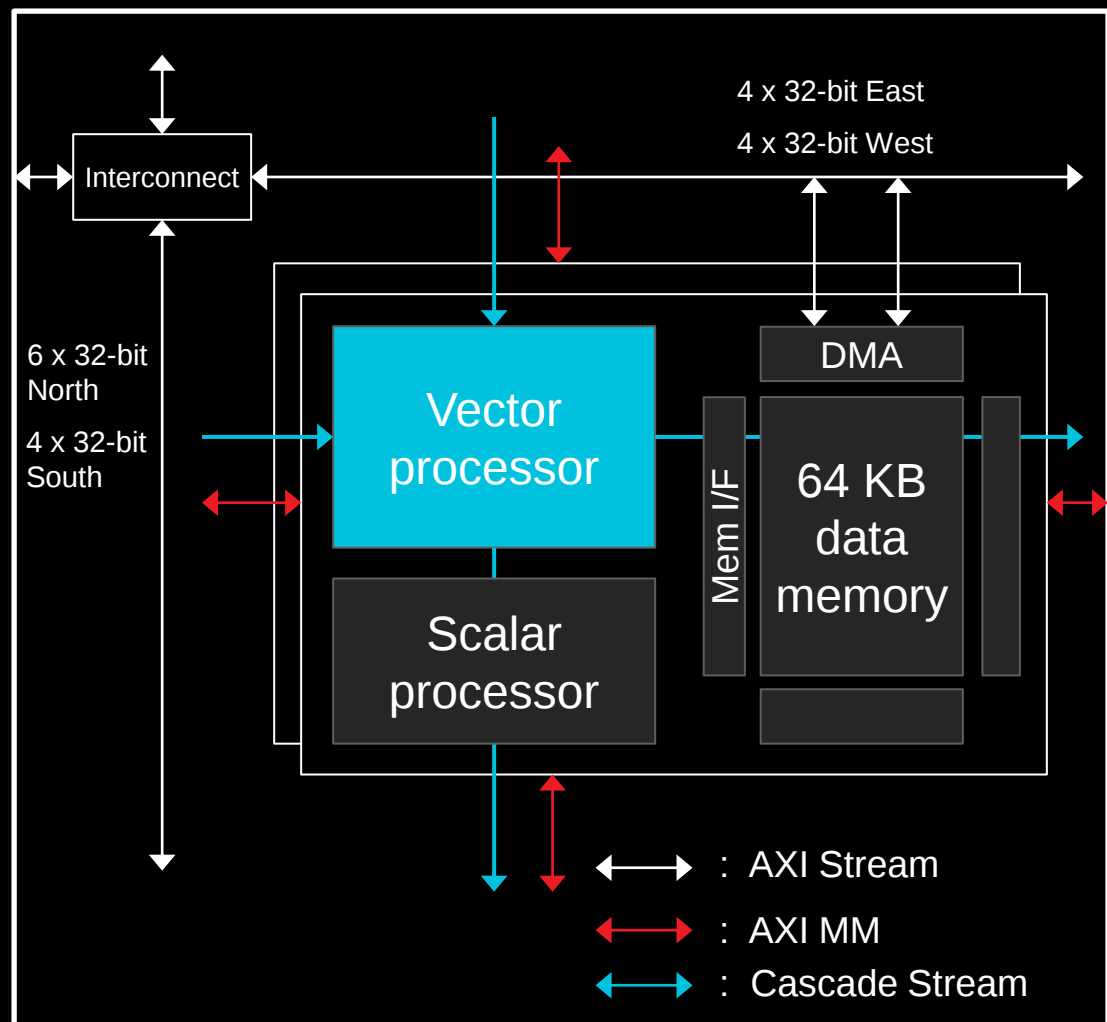
XDNA AI Engine Overview



How to program?

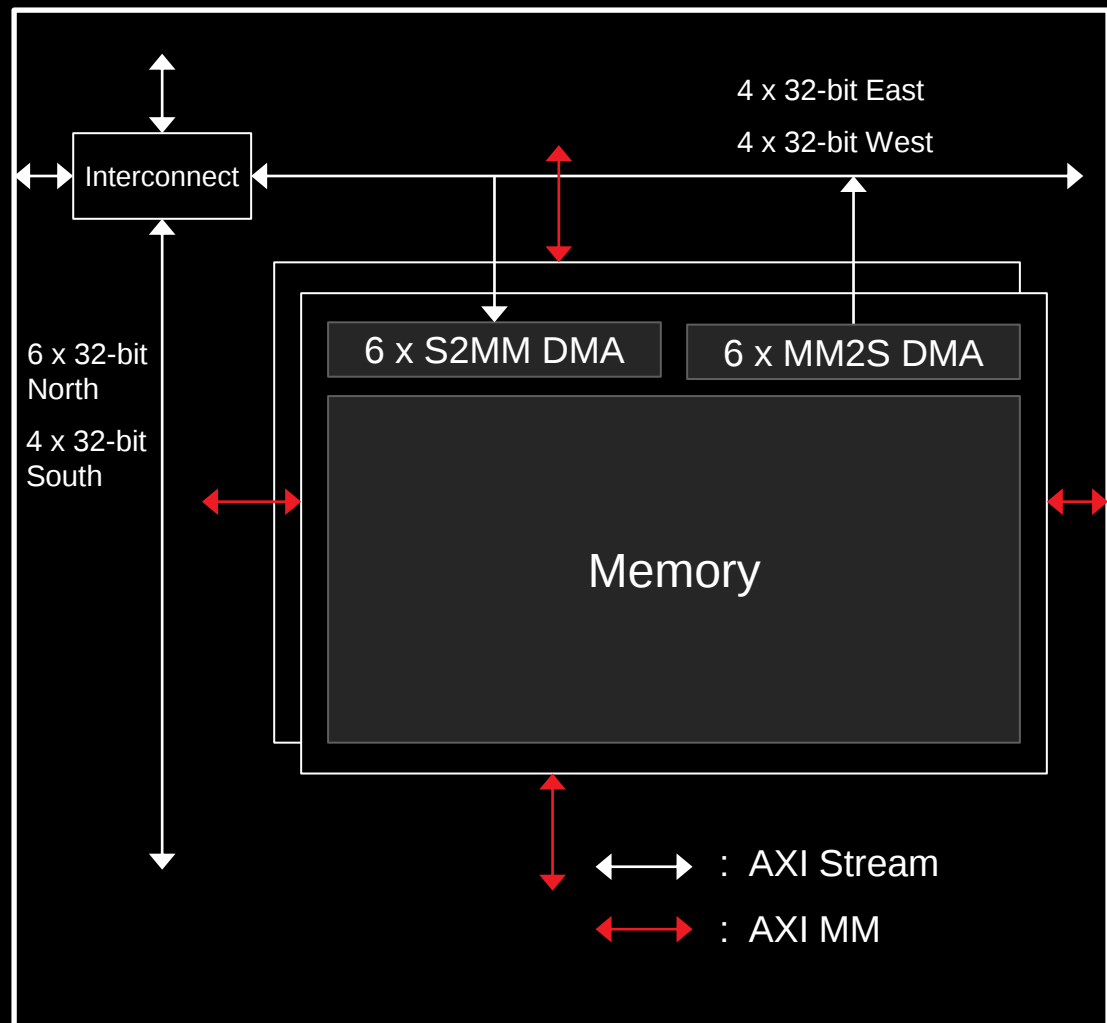
- Needs an initial boot PDI (Programmable Device Image)
 - Can contain initial configuration (routes, locks...), which can be generated
- Generate AIE Core code to perform the desired computation on that tile (ELF)
- Generate control code for the microcontroller to drive the AIE array data movement

AI Engine Core



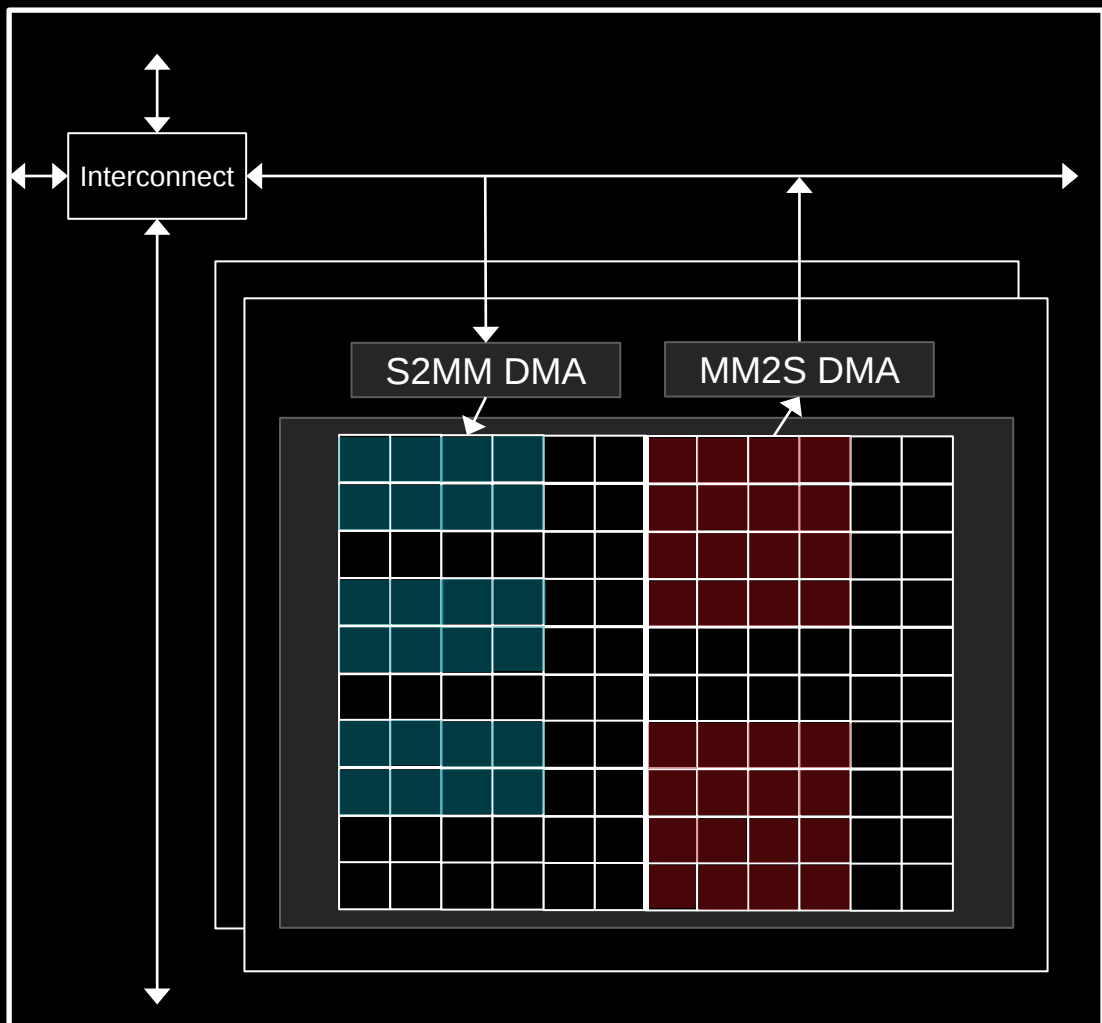
- A VLIW SIMD processor
- A scalar RISC processor
- 64KB of data memory
- 16KB of program memory
- 2 MM2S and 2 S2MM DMAs
- 32-bit stream and memory AXI interconnect
- Stream switches with broadcasting capability

AI Engine Memory Tile



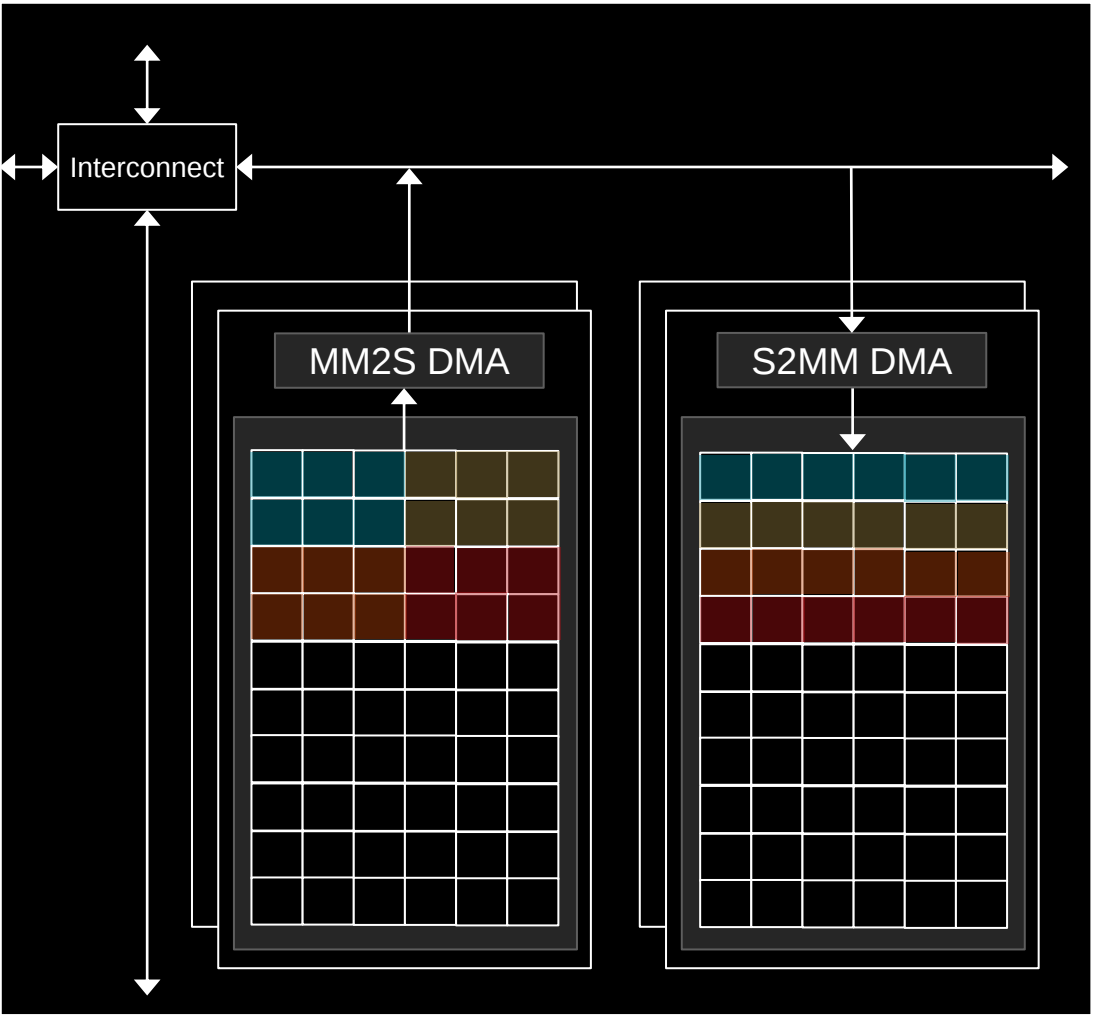
- 512KB data memory
- 6 stream-to-memory and 6 memory-to-stream DMAs
- Each DMA has a 128-bit memory and 32-bit stream interface
- 32-bit stream and memory AXI interconnect

AI Engine DMA



- Stream to memory and memory to stream
- Usually, 32-bit stream interface and 128-bit memory interface
- Supports multi-dimensional data addressing
- S2MM Example:
 - Strides = [18, 6, 1]
 - Sizes = [3, 2, 4]
- MM2S Example:
 - Strides = [36, 6, 1]
 - Sizes = [2, 4, 4]

Data Packing through DMA

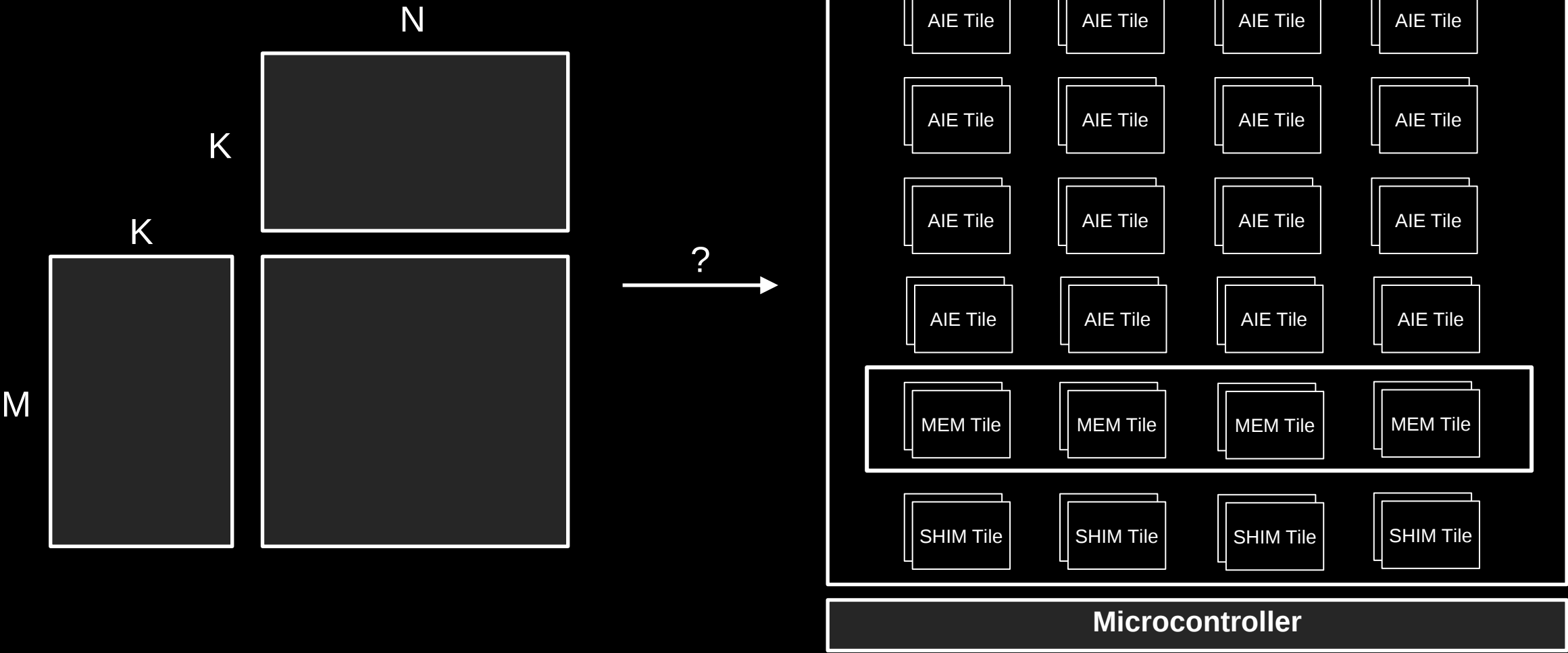


```
%0 = memref.alloc() : memref<4x6xi32>
%1 = memref.alloc() : memref<2x2x2x3xi32, 1>
iree_linalg_ext.pack %0
  inner_dims_pos = [0, 1] inner_tiles = [2, 3]
  into %1 : (memref<4x6xi32, strided<[6, 1],
offset: ?>> memref<2x2x2x3xi32, 1>)
```

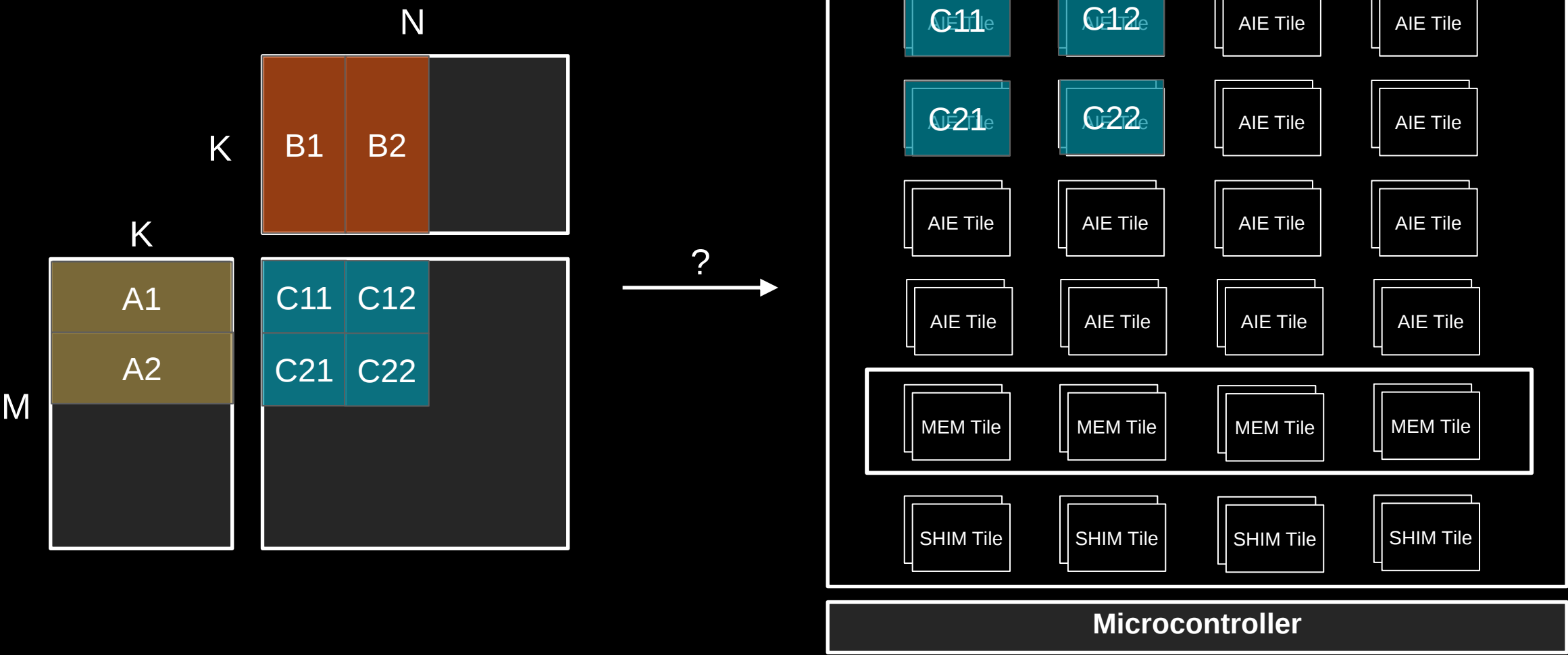
```
%0 = memref.alloc() : memref<4x6xi32>
%1 = memref.alloc() : memref<2x2x2x3xi32, 1>
amdaie.dma_cpy_nd(
  %1[0] [24] [1]
  %0[0, 0, 0, 0] [2, 2, 2, 3] [12, 3, 6, 1]
)
```

```
%0 = memref.alloc() : memref<4x6xi32>
%1 = memref.alloc() : memref<2x2x2x3xi32, 1>
amdaie.dma_cpy_nd(
  %1[0, 0, 0, 0] [2, 2, 2, 3] [12, 3, 6, 1]
  %0[0] [24] [1]
)
```

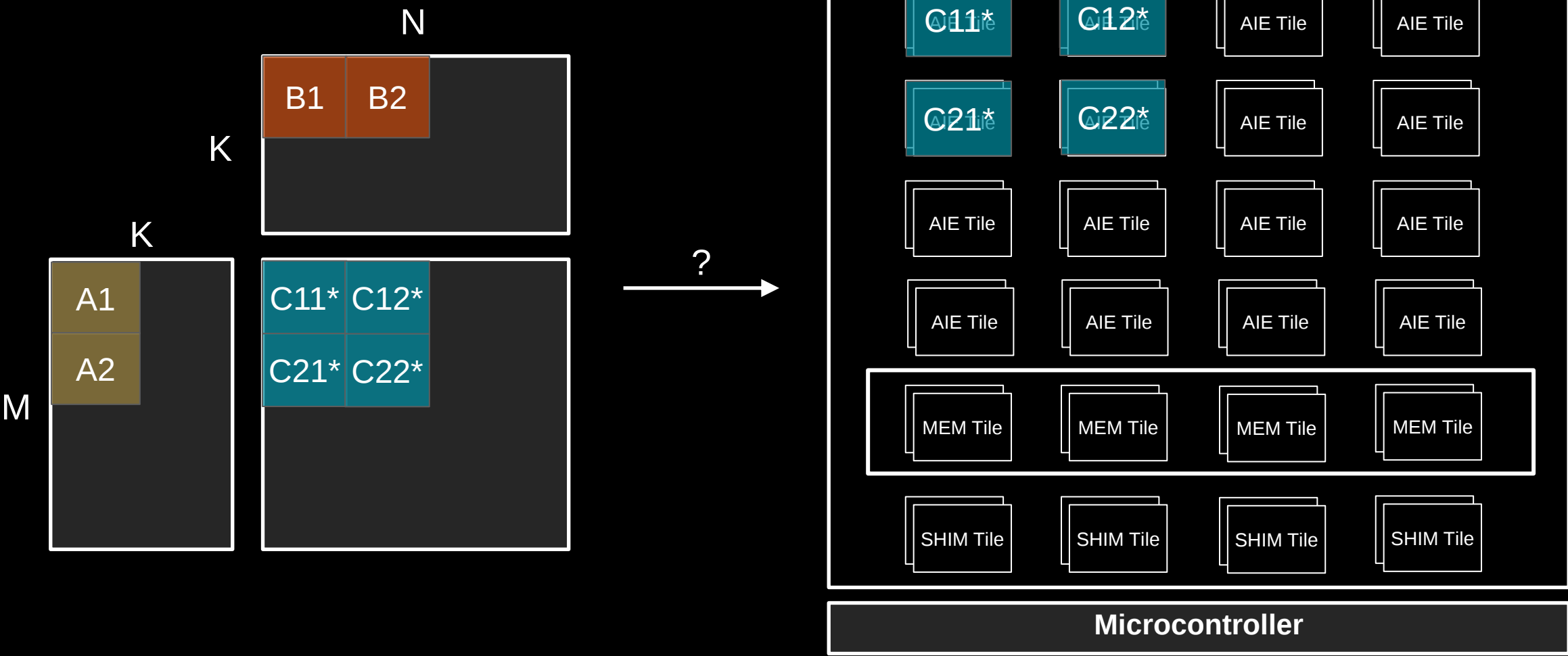
MxKxN GEMM Tiling Problem



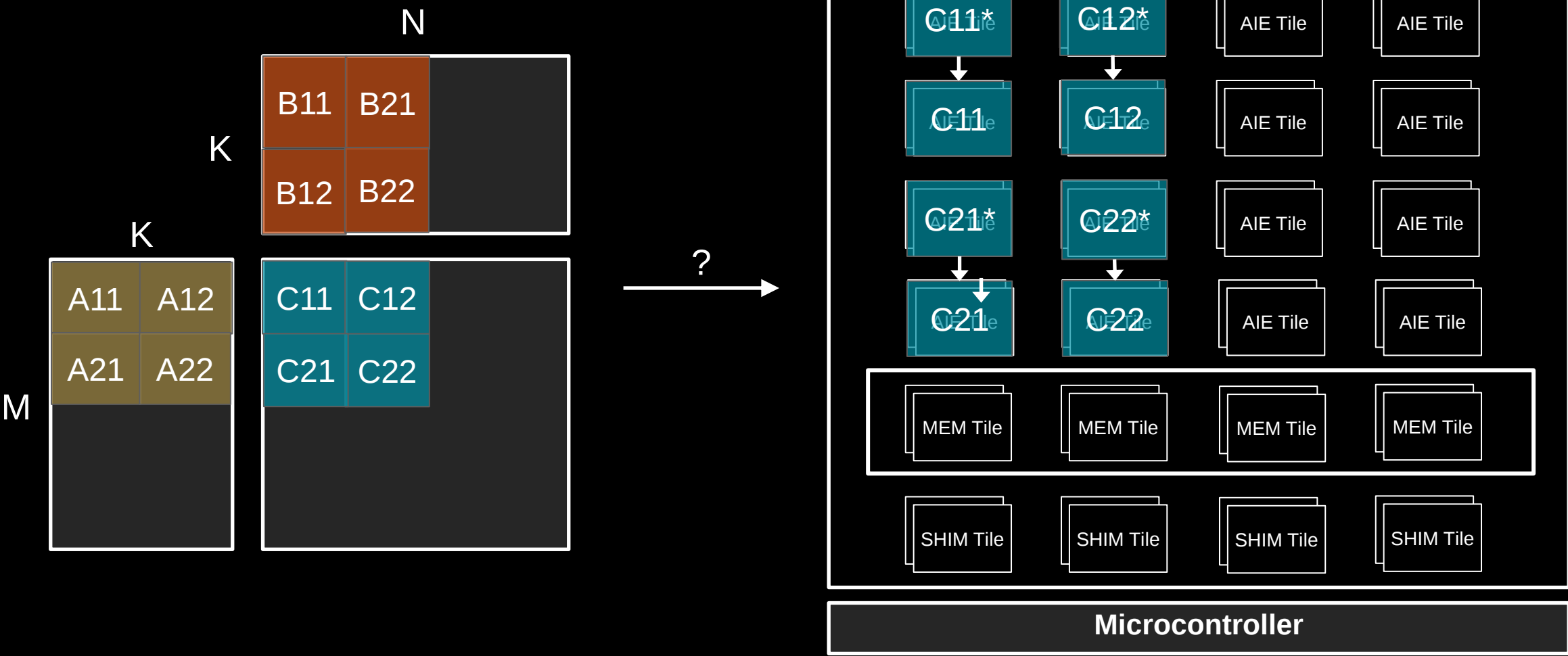
MxKxN GEMM Tiling Problem



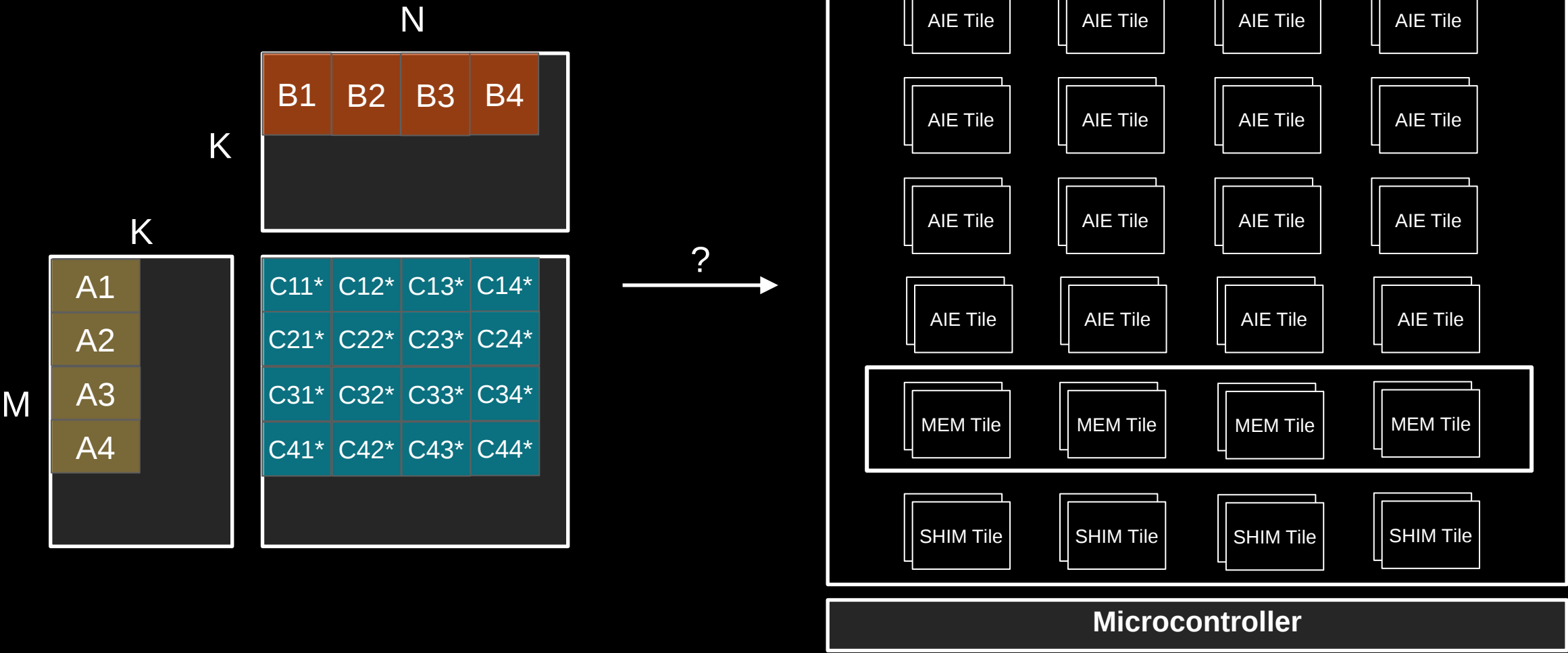
MxKxN GEMM Tiling Problem



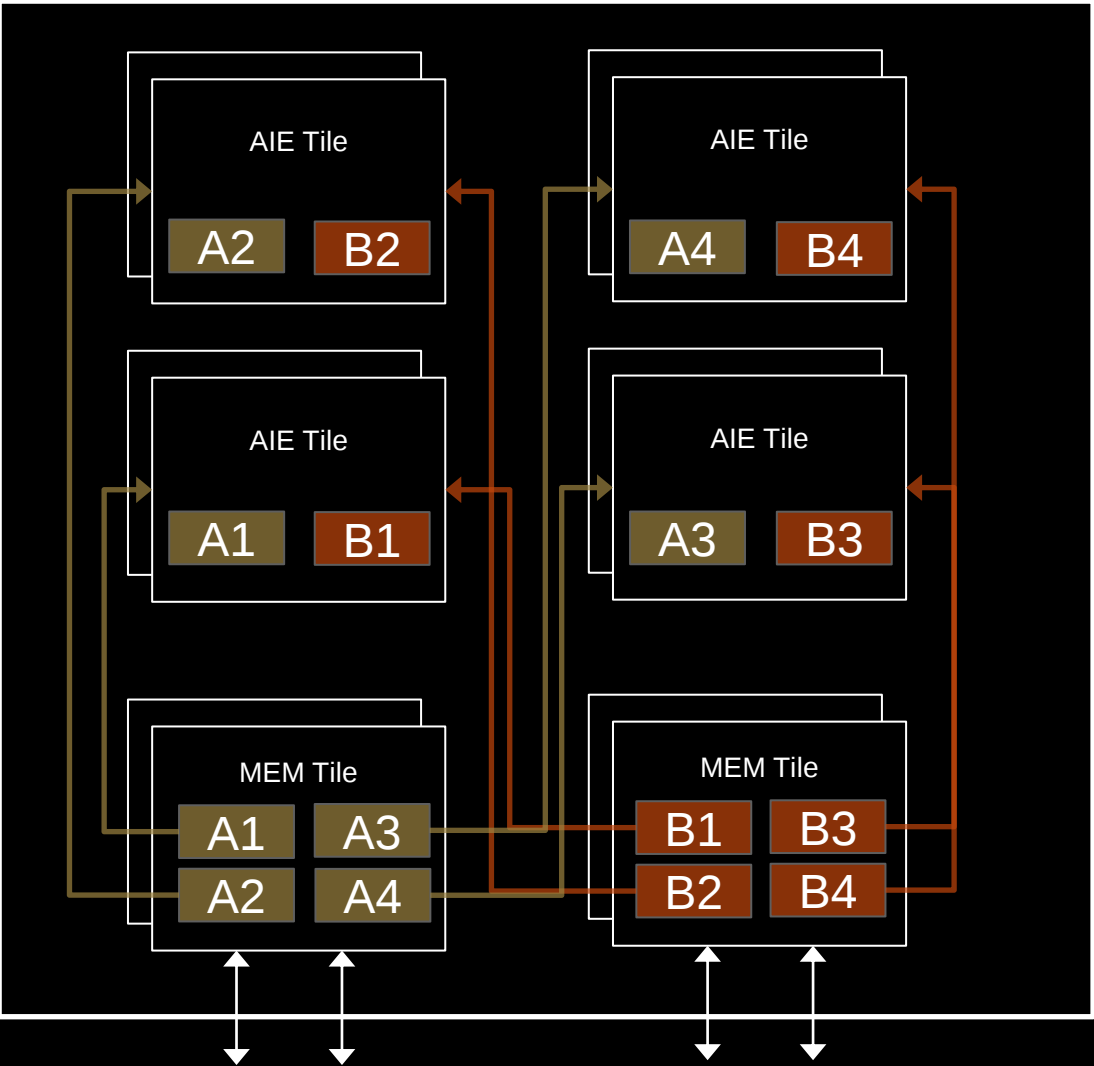
MxKxN GEMM Tiling Problem



MxKxN GEMM Tiling Problem



Parallelization: Matmul Unicast

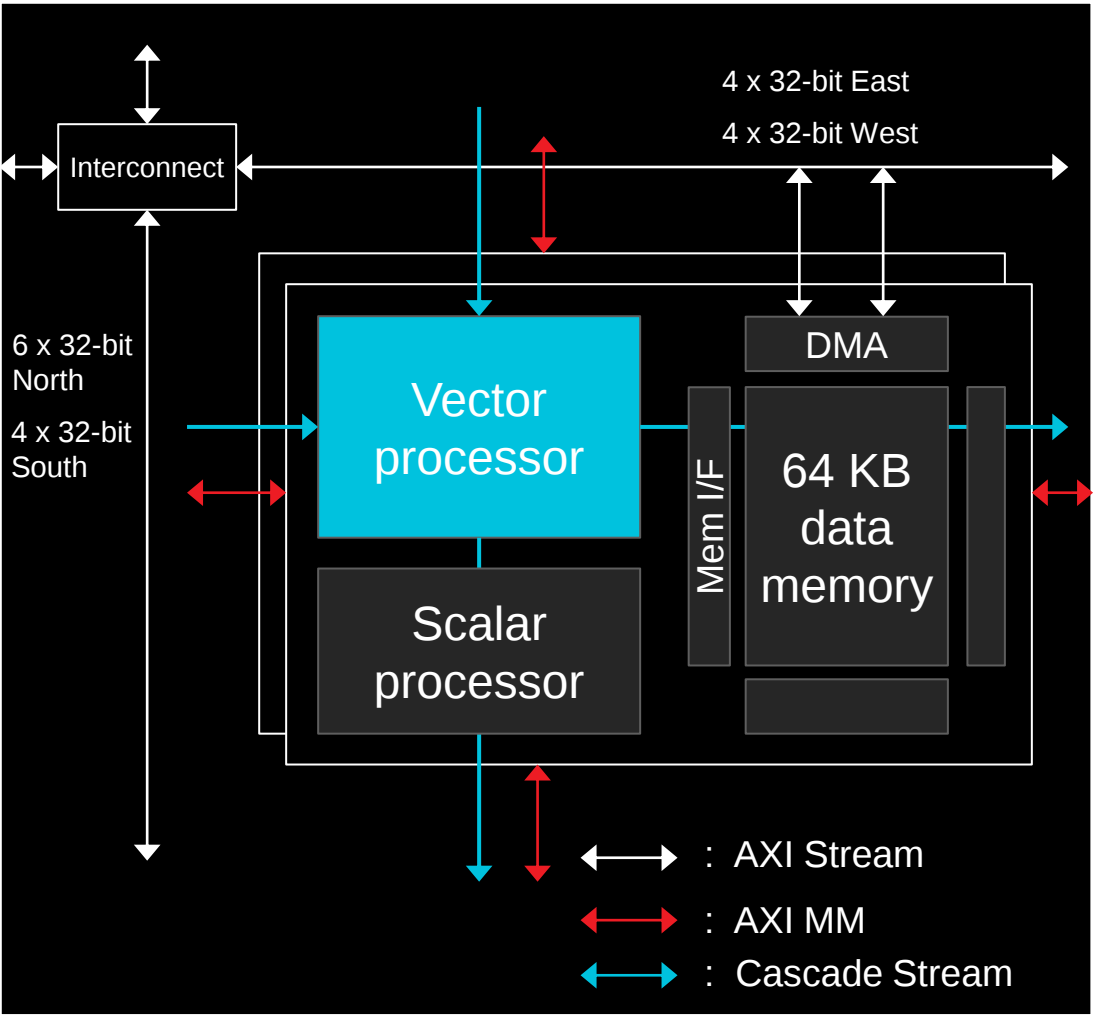


```

%0 = memref.alloc() : memref<4x1x32x64xi8, 1>
%1 = memref.alloc() : memref<1x4x64x32xi8, 1>
%2 = memref.alloc() : memref<1x1x32x64xi8, 2>
%3 = memref.alloc() : memref<1x1x64x32xi8, 2>
scf.forall (%arg0, %arg1) in (2, 2) {
    %4 = affine.apply affine_map<(d0, d1) ->
    (d0 + 2 * d1)>(%arg0, arg1)
    dma_cpy_nd(%2[0, 0...]..., %0[%4, 0...]...)
    dma_cpy_nd(%3[0, 0...]..., %1[0, %4...]...)
    linalg.matmul(%2, %3, ...)
} {mapping = [#core<y>, #core<x>]}
    
```

- 16 GB/s bandwidth needed into each column (1GHz)

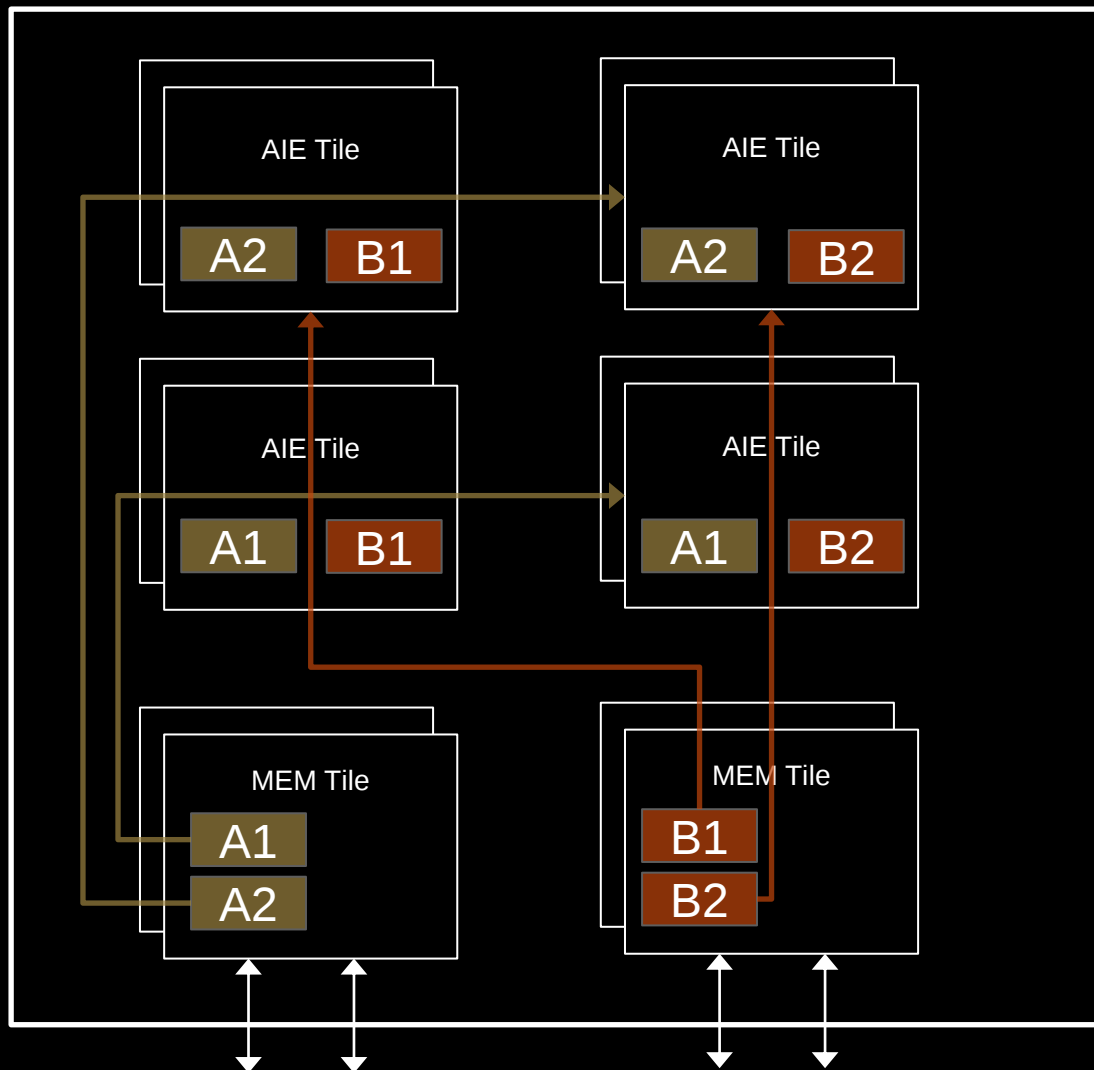
Remember the bandwidth available:



Assuming 1Ghz:

- 24GB/s North
- 16GB/s South

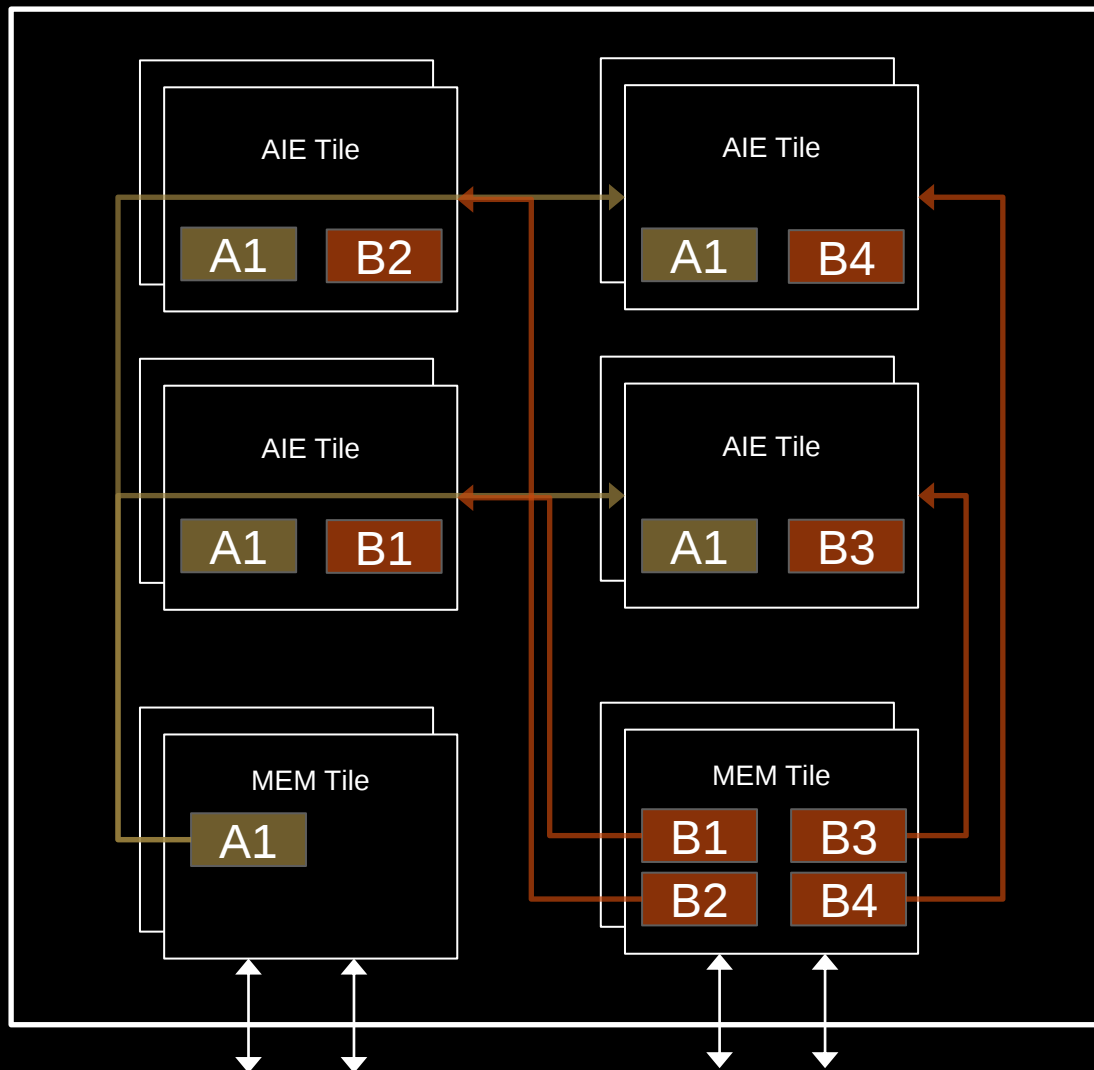
Parallelization: Matmul Double Broadcast



```
%0 = memref.alloc() : memref<2x1x32x64xi8, 1>
%1 = memref.alloc() : memref<1x2x64x32xi8, 1>
%2 = memref.alloc() : memref<1x1x32x64xi8, 2>
%3 = memref.alloc() : memref<1x1x64x32xi8, >
scf.forall (%arg0, %arg1) in (2, 2) {
  dma_cpy_nd(%2[0, 0...]..., %0[%arg0, 0...]...)
  dma_cpy_nd(%3[0, 0...]..., %1[0, %arg1...]...)
  linalg.matmul(%2, %3, ...)
} {mapping = [#core<y>, #core<x>]}
```

- 8 GB/s bandwidth needed into each column (1GHz)

Parallelization: Matmul Full A Broadcast



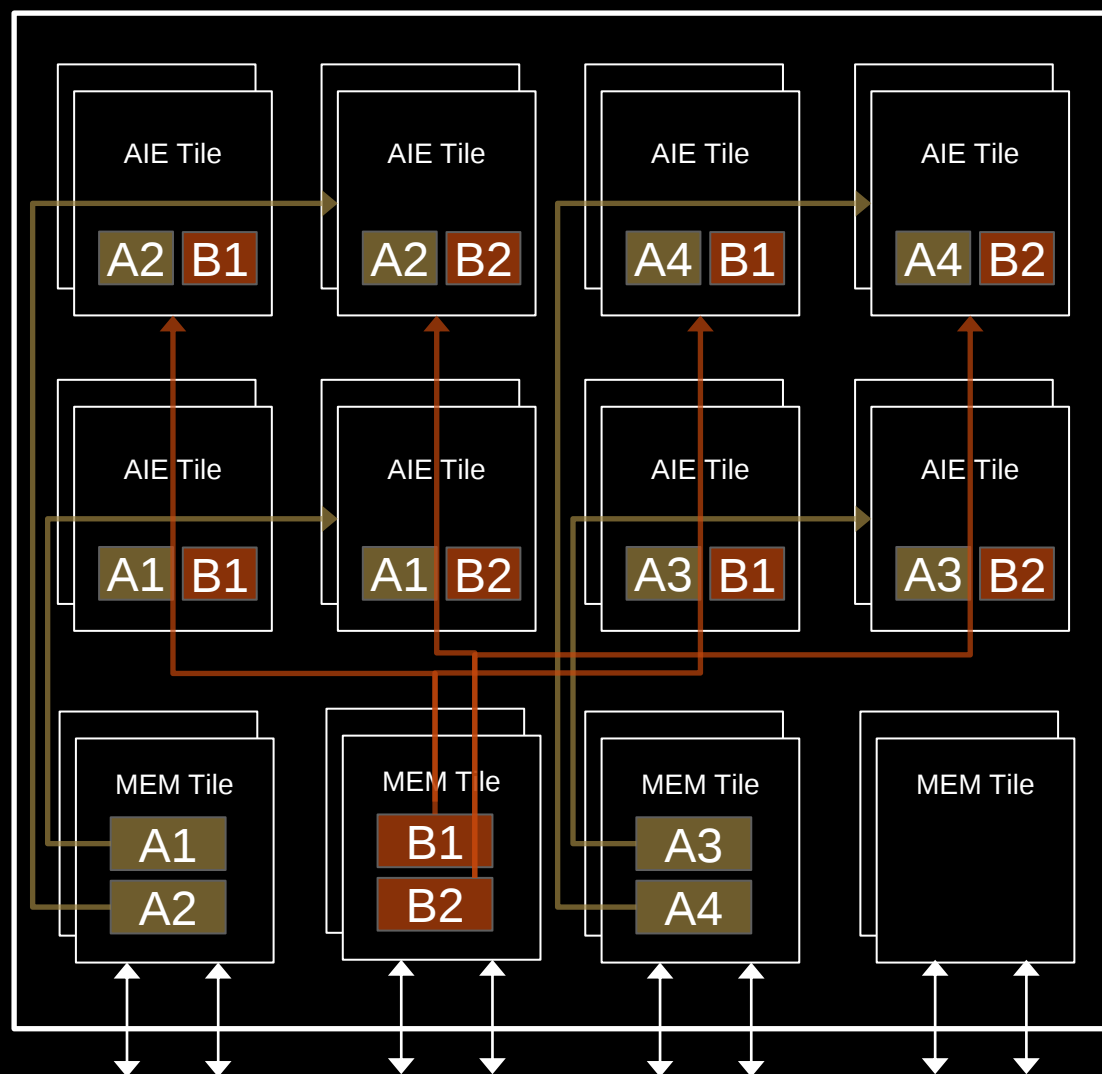
```

%0 = memref.alloc() : memref<4x1x32x64xi8, 1>
%1 = memref.alloc() : memref<1x4x64x32xi8, 1>
%2 = memref.alloc() : memref<1x1x32x64xi8, 2>
%3 = memref.alloc() : memref<1x1x64x32xi8, 2>
scf.forall (%arg0, %arg1) in (2, 2) {
  %4 = affine.apply affine_map<(d0, d1) ->
    (d0 + 2 * d1)>(%arg0, %arg1)
  dma_cpy_nd(%2[0, 0...]..., %0[0, 0...]...)
  dma_cpy_nd(%3[0, 0...]..., %1[0, %4...]...)
  linalg.matmul(%2, %3, ...)
} {mapping = [#core<y>, #core<x>]}

```

- 10 GB/s bandwidth needed into each column on average (1GHz)

Adding Dimensions



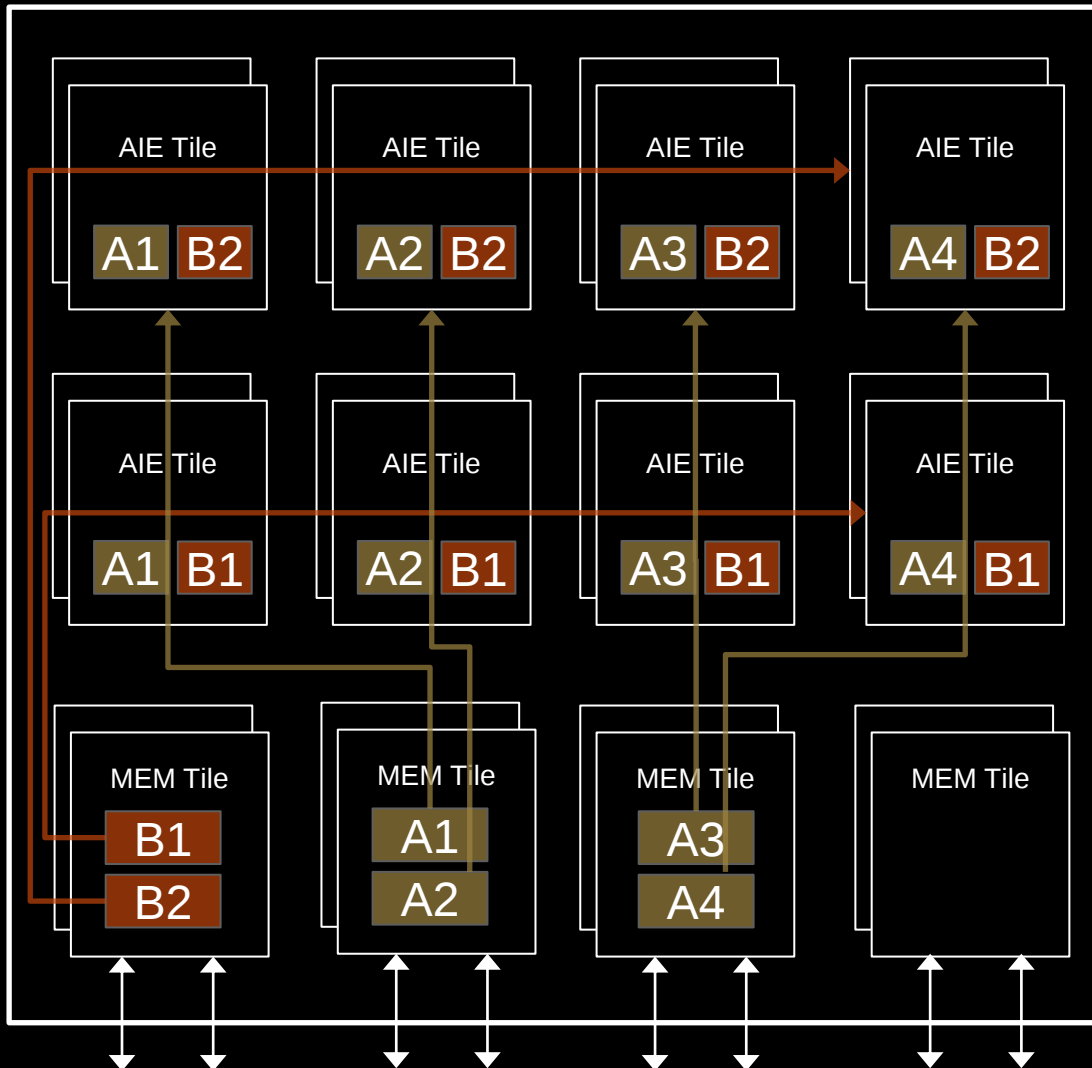
```

%0 = memref.alloc() : memref<4x1x32x64xi8, 1>
%1 = memref.alloc() : memref<1x2x64x32xi8, 1>
%2 = memref.alloc() : memref<1x1x32x64xi8, 2>
%3 = memref.alloc() : memref<1x1x64x32xi8, 2>
scf.forall (%arg0, %arg1, %arg2) in (2, 2, 2)
{
  %4 = affine.apply affine_map<(d0, d1) -> (2
* d0 + d1)>(%arg0, %arg1)
  dma_cpy_nd(%2[0, 0...]..., %0[%4, 0...]...)
  dma_cpy_nd(%3[0, 0...]..., %1[0, %arg2...]...)
  linalg.matmul(%2, %3, ...)
} {mapping = [#core<y>, #core<x1>,
#core<x2>]}

```

- 6GB/s bandwidth needed into each column on average (1GHz), but higher horizontal bandwidth

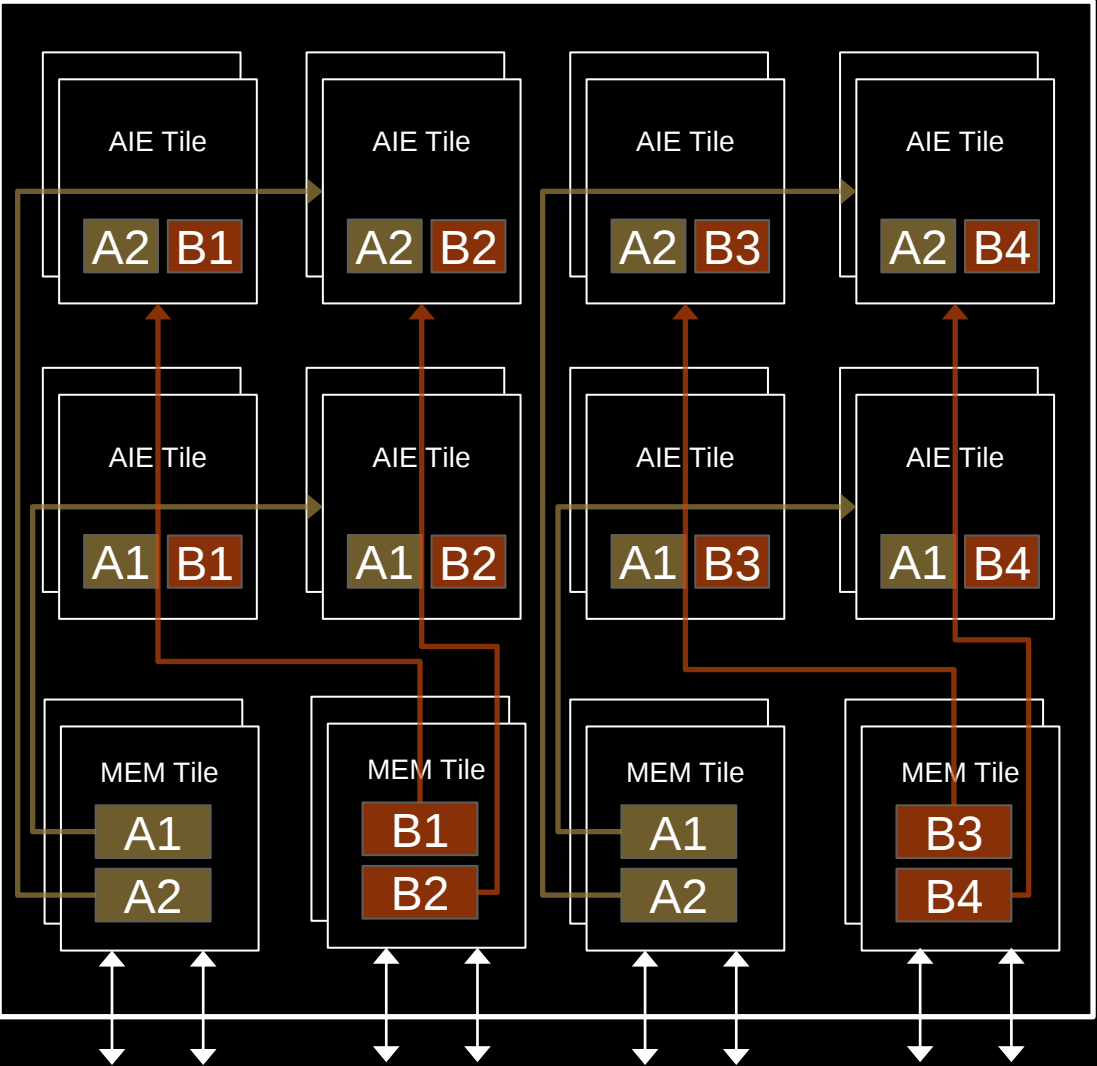
Achieving the same BW needs without adding Dimensions



```
%0 = memref.alloc() : memref<4x1x32x64xi8, 1>
%1 = memref.alloc() : memref<1x2x64x32xi8, 1>
%2 = memref.alloc() : memref<1x1x32x64xi8, 2>
%3 = memref.alloc() : memref<1x1x64x32xi8, 2>
scf.forall (%arg0, %arg1) in (2, 4) {
  dma_cpy_nd(%2[0, 0...]..., %0[%arg1, 0...]...)
  dma_cpy_nd(%3[0, 0...]..., %1[0, %arg0...]...)
  linalg.matmul(%2, %3, ...)
} {mapping = [#core<y>, #core<x>]}
```

- 6GB/s bandwidth needed into each column on average (1GHz)

Parallelization: Adding Blocks



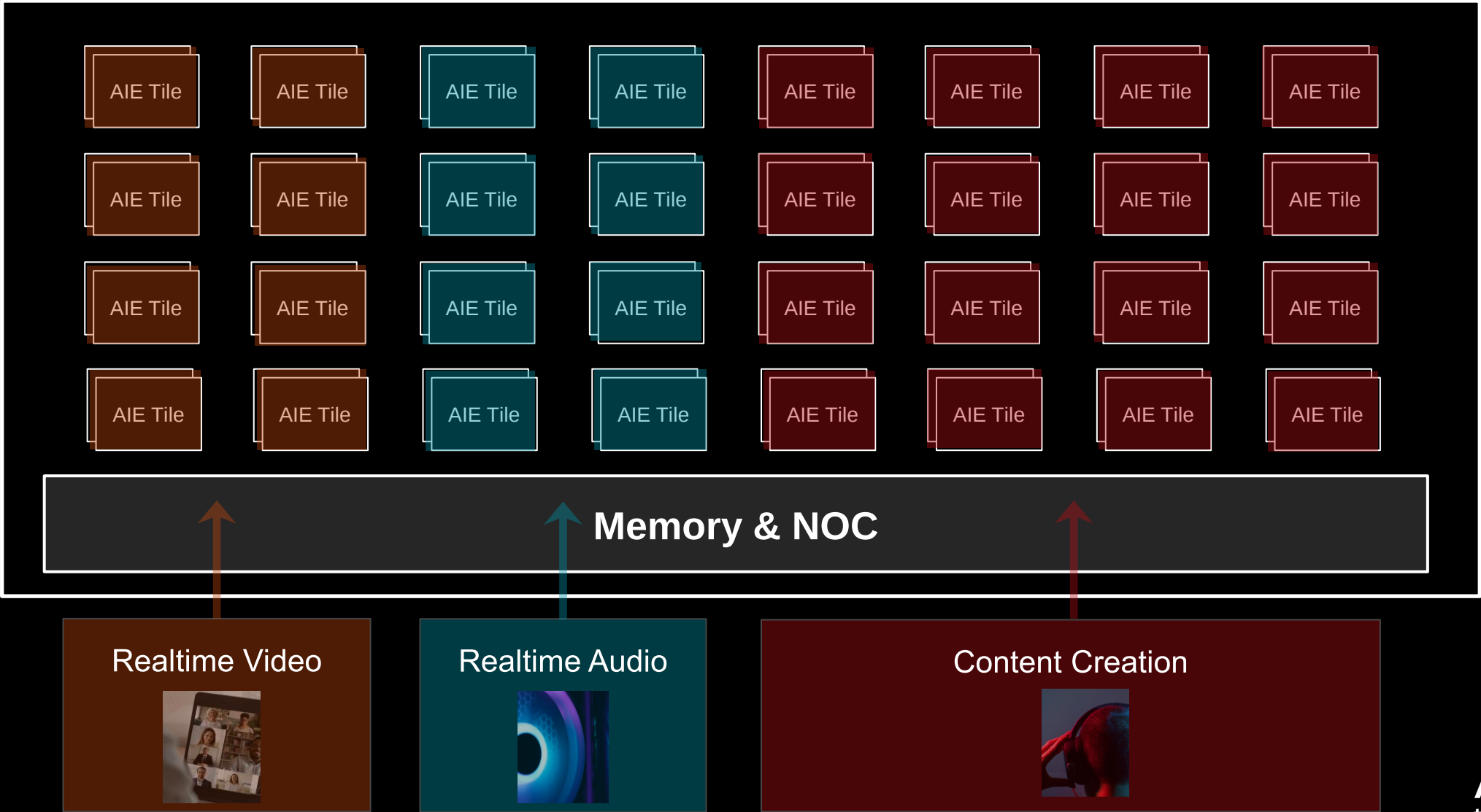
```

%0 = memref.alloc() : memref<2x1x32x64xi8, 1>
%1 = memref.alloc() : memref<1x4x64x32xi8, 1>
%2 = memref.alloc() : memref<1x1x32x64xi8, 2>
%3 = memref.alloc() : memref<1x1x64x32xi8, 2>
scf.forall (%arg0) in (2) {
  scf.forall (%arg1, %arg2) in (2, 2) {
    %4 = affine.apply affine_map<(d0, d1) ->
    (2 * d0 + d1)>(%arg0, %arg2)
    dma_cpy_nd(%2[0, 0...]..., %0[%arg1, 0...]...)
    dma_cpy_nd(%3[0, 0...]..., %1[0, %4...]...)
    linalg.matmul(%2, %3, ...)
  } {mapping = [#core<y>, #core<x>]}
} {mapping = [#block<x>]}

```

- 8GB/s bandwidth needed into each column (1GHz)

Partitioning



Recap

1. We're targeting an array of VLIW SIMD processors, generating core code, routing and DMA configurations for a (fused) computation.
2. We use ``iree_linalg_ext.pack`` to describe data packing while moving/copying data through DMAs
3. We use 'core' mapping to describe parallelization AND to discover data broadcasting opportunities
4. All open source at: <https://github.com/nod-ai/iree-amd-aie>

