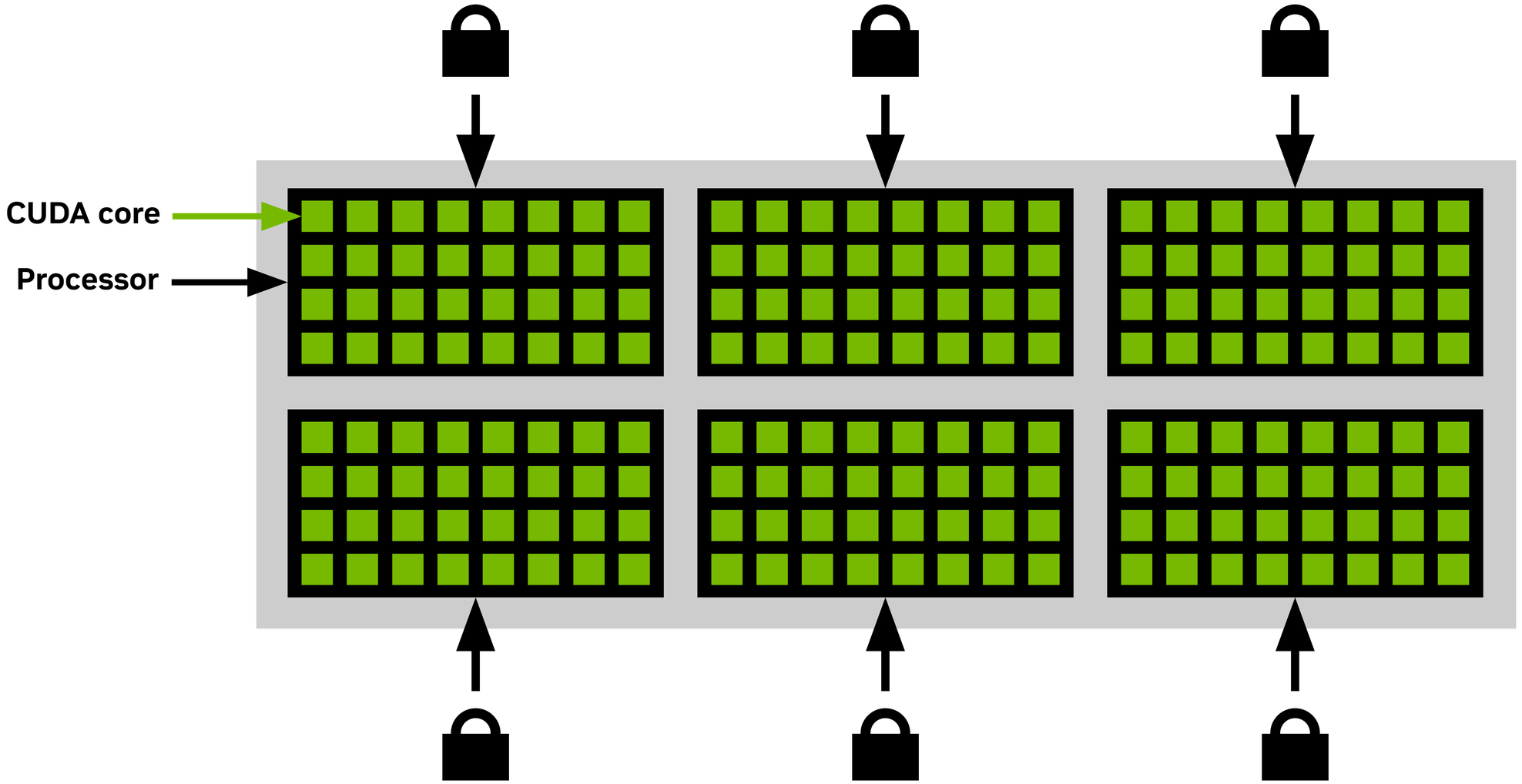
Introduction to CUDA Tile IR

Théo Degioanni

The Structure of a GPU

Extremely simplified



Hierarchy of Execution Models

How do we program GPUs?

Grid-Level Programming Model

System maps data to blocks and threads

```
import cupy as cp

a = cp.random.rand(3,3).astype(cp.float32)
b = cp.random.rand(3,3).astype(cp.float32)

c = cp.matmul(a, b) # use cuBLAS
print(c)
```

cuBLAS

cuDNN

cuSPARSE

Thrust

Hierarchy of Execution Models

How do we program GPUs?

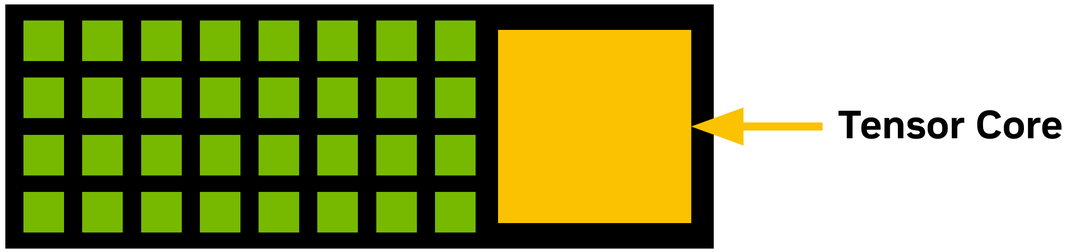
```
__global__ void hello() {
  if (threadId.x == 0 && blockId.x == 0)
    printf(“Hello from thread0!”)

int main() {
  hello<<<1, 256>>>();
  cudaDeviceSynchronize();
  return 0
}
```

Thread-Level Programming Model

User maps data
to blocks and threads

But the Problem is...

We have tensor cores!



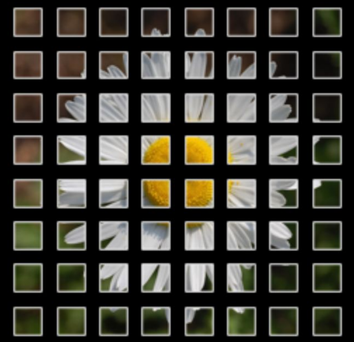
Block-Level Programming Model

A middle ground

Grid-Level Programming Model

System maps data to blocks and threads

Block-Level Programming Model



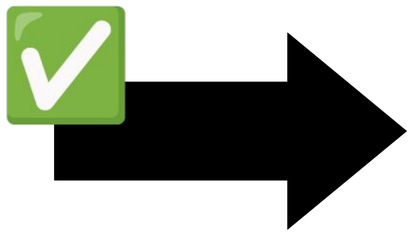
User maps data to blocks, **system** maps data to threads

Thread-Level Programming Model

User maps data to blocks and threads

Block-Level Programming in CUDA: TileIR

The PTX of block-level programming



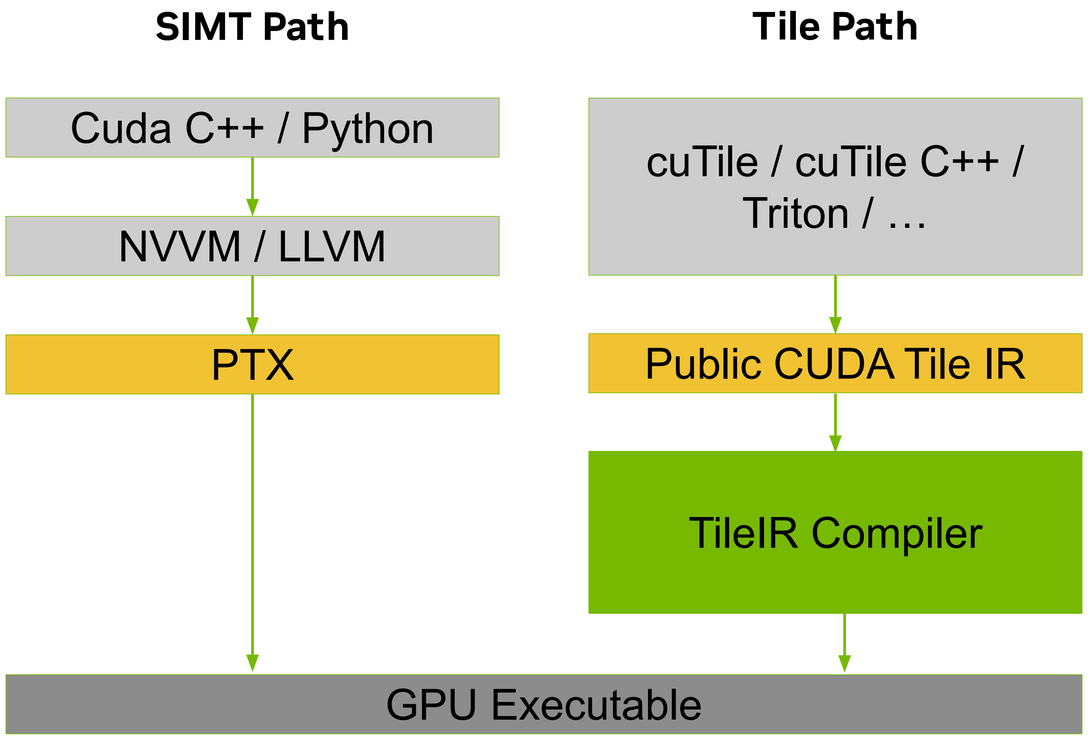
Forward-compatible

MLIR-based

Integrated with CUDA

CUDA Tile IR in the CUDA Platform

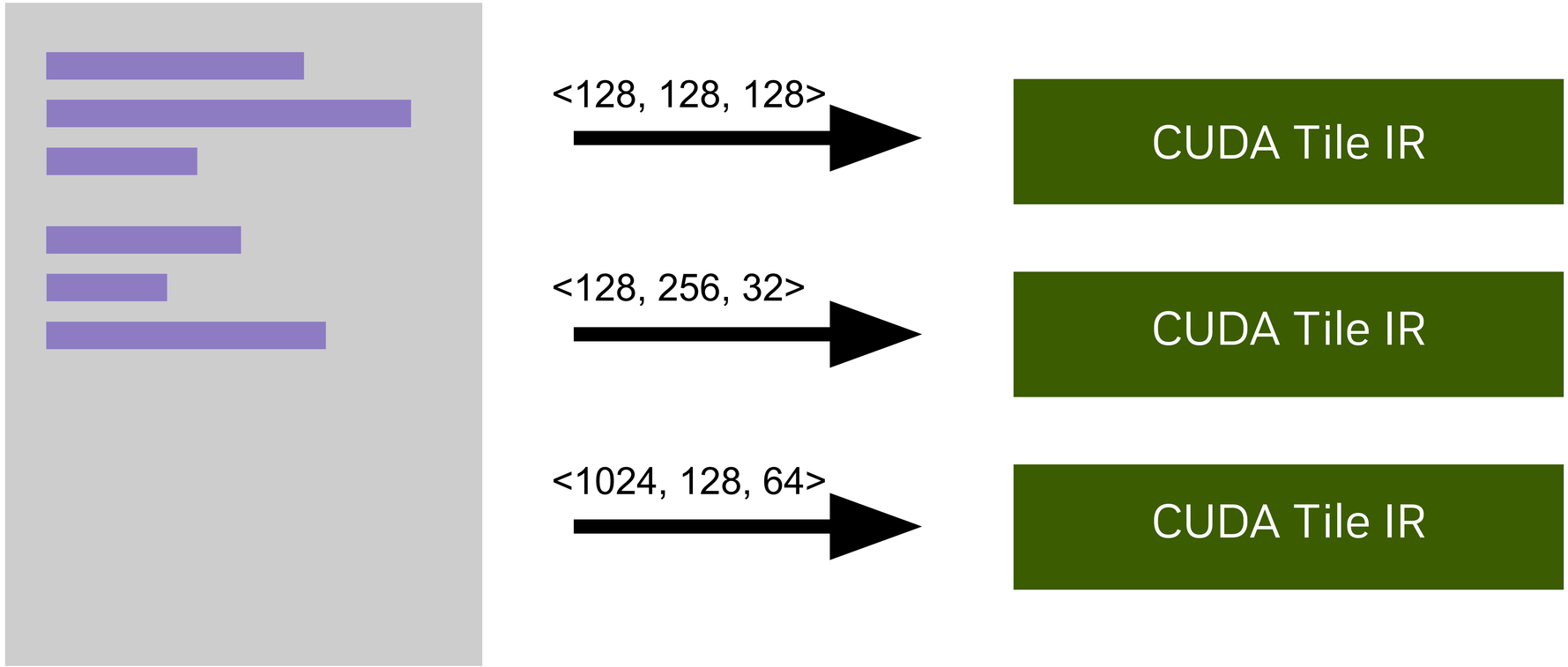
CUDA Tile IR is a new core component of the CUDA platform



How to use CUDA TileIR?

A GPU Requires a Host

Running GPU programs requires a distribution of work



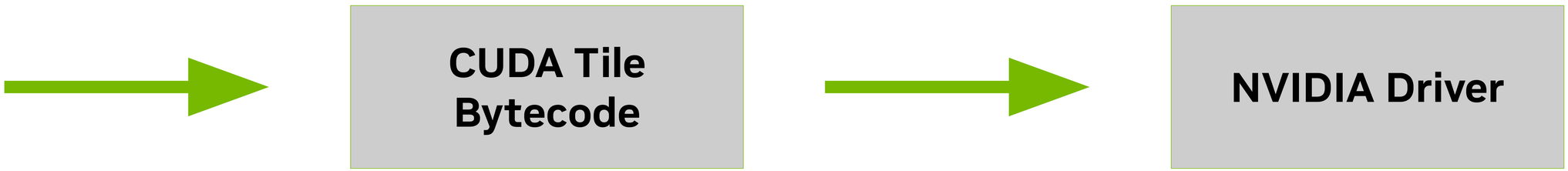
CPU Host Program

GPU Kernels

An Example CUDA Tile IR Kernel

Expressed in MLIR textual format

```
cuda_tile.module @kernels {
    entry @my_kernel() {
        print_tko "hello_world" -> !cuda_tile.token
    }
}
```



The Tile Type

CUDA Tile's main data type

!cuda_tile.tile<i32> *(or tile<i32>)*

!cuda_tile.tile<16xf32> *(or tile<16xf32>)*

!cuda_tile.tile<32x32xf4E2M1FN> *(or tile<32x32xf4E2M1FN>)*

```
%c24 = constant <f16: [24.0, 42.0]> : tile<2xf16>
%t42 = addi %c42, %c24 : tile<2xf16>

%cmp = cmpi equal %t42, %t42, signed
                  : tile<2xf16> -> tile<2xi1>
```

An Example CUDA Tile IR Kernel

Expressed in MLIR textual format

```
cuda_tile.module @kernels {
    entry @my_kernel() {
        %x, %y, %z = get_tile_block_id : tile<i32>

        print_tko "(%i, %i, %i)\n", %x, %y, %z
            : tile<i32>, tile<i32>, tile<i32>
            -> !cuda_tile.token
    }
}
```

Example output for grid 2x1x2:

(0, 0, 1)
(1, 0, 1)
(0, 0, 0)
(1, 0, 0)

How Control-Flow Works in CUDA Tile IR

Region-based control-flow is first class

```
%if_result = if %cond -> (tile<i1>) {
  yield %x : tile<i32>
} else {
  yield %y : tile<i32>
}


for %iv_u in (%zero to %ten, step %one) : tile<i32> {
    continue
}
```

How Control-Flow Works in CUDA Tile IR

Early-exit is supported

```
%for_result = for %iv in
        (%zero to %ten, step %one) : tile<i32>
        iter_values(%acc = %zero) -> (tile<i32>) {
  if %cond {
    continue %zero : tile<i32>
  }

  continue %iv : tile<i32>
}
```

How Control-Flow Works in CUDA Tile IR

Early-exit is supported

```
%res = loop iter_values(%acc = %zero) : tile<i32> {
  if %cond {
    continue %acc : tile<i32>
  }

  if %cond2 {
    break %acc : tile<i32>
  }

  %acc_next = addi %acc, %one : tile<i32>
  continue %acc_next : tile<i32>
}
```

How Memory Access Works

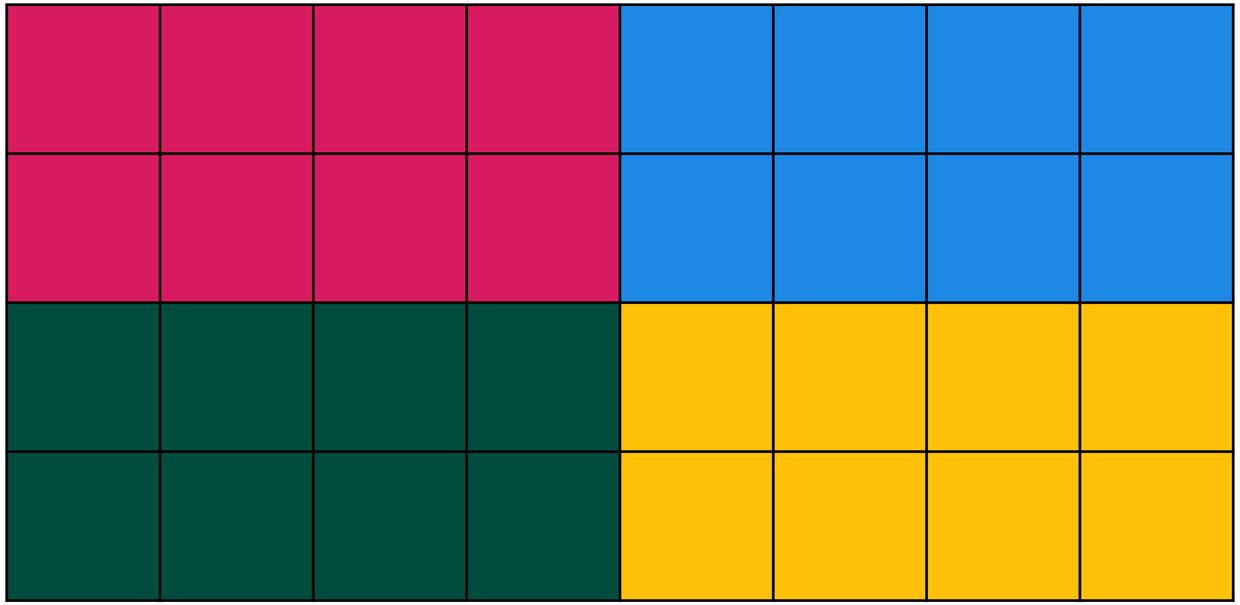
Tiles are loaded from global memory

```
cuda_tile.module @kernels {
    entry @my_kernel(%ptr: !cuda_tile.tile<ptr<f32>>) {
        %tensor_view = make_tensor_view
            %ptr, shape = [8,4], strides = [4,1]
            : tensor_view<8x4xf32, strides=[4,1]>

        // …
    }
}
```

How Memory Access Works

Tiles are loaded from global memory

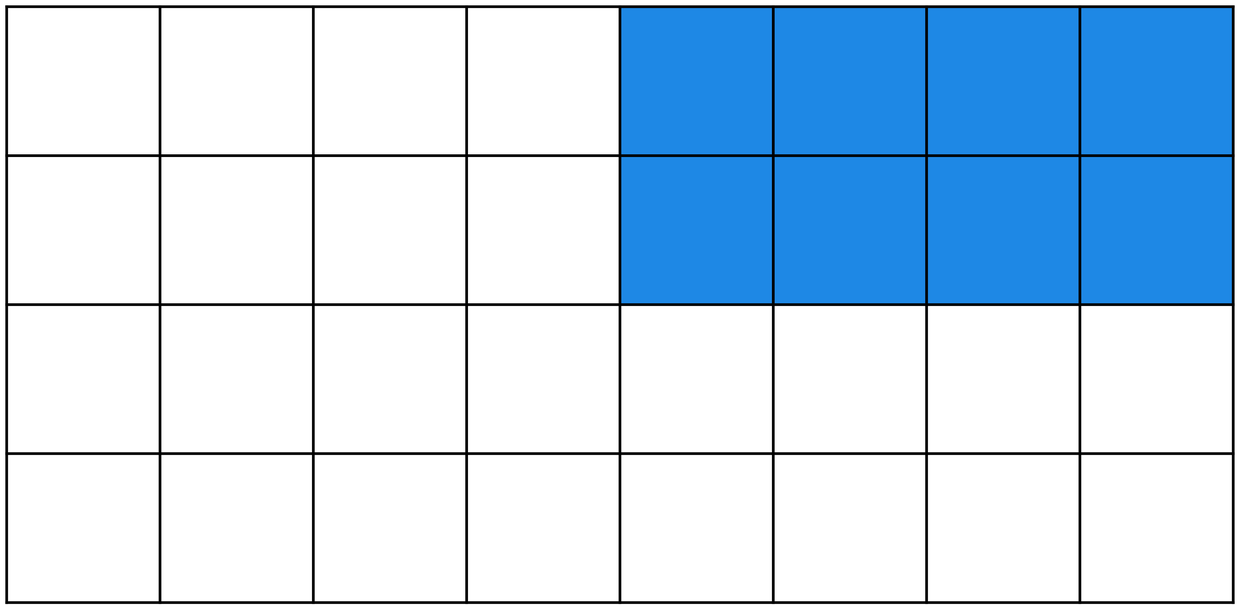
```
%partition_view = make_partition_view
    %tensor_view
     : partition_view<
         tile=(4x2),
         tensor_view<8x4xf32, strides=[4,1]>
       >
```



How Memory Access Works

Tiles are loaded from global memory

```
%tile, %token = load_view_tko weak
    %partition_view[%one, %zero]
    : partition_view<tile=(4x2),
                     tensor_view<8x4xf32, strides=[4,1]>>,
      tile<i64>
      -> tile<4x2xf32>, token
```



Token Ordering

Informs the compiler about in-tile dependencies

```
// Partition view contains 0

store_view_tko weak %one, %partition_view[%zero]

%loaded = load_view_tko weak %partition_view[%zero]


// What is the value of %loaded?
```

It is ambiguous! 🤯

Token Ordering

Informs the compiler about in-tile dependencies

```
// Partition view contains 0

%token = store_view_tko weak %one, %partition_view[%zero]

%loaded = load_view_tko weak token = %token
                %partition_view[%zero]

// What is the value of %loaded?
```

Now it is always one!

Intermission: Free GPU!!

(for the duration of this session only, sorry)

1. Create an account on Brev with this QR code (or link your existing account).

2. Once you see “Environment”, download the Brev CLI: **https://docs.nvidia.com/brev/cli/getting-started#installation**

3. `brev login && brev set mlir-summer-school-2026`

4. `brev shell summer-school`

5. `cp -r /cuda-tile-project ~/<your-name>`

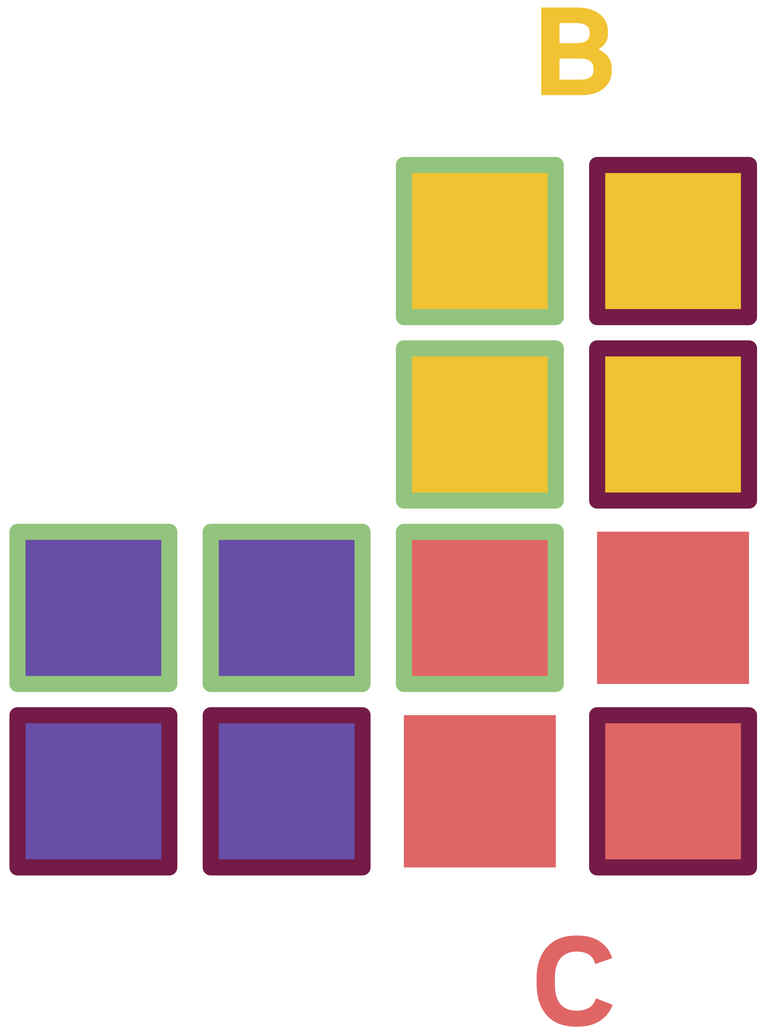
Let's Build a MatMul

I've heard this is popular

for each tile of C:

- **create** an accumulator tile set to zero
- **for** k in the K dimension:
  - **multiply** the kth-tile of the corresponding column of B and the corresponding row of A
  - **accumulate** the result in an accumulator tile
- **write** the accum[illegible] to the tile of C

In CUDA Tile IR, this is one operation: **mmaf**



Let’s Build a MatMul

I’ve heard this is popular

Complete **matmul.mlir** in the project folder.

To reopen the environment:

```
$ brev shell summer-school
$ cd <your-name>
```

To open the environment in VS Code:

```
$ brev open summer-school code
```

To copy files to the environment:

```
$ brev copy $local_path summer-school:$remote_path
```

Cheat sheet for operations in **matmul-cheat-sheet.md**
Execution instructions can be found in the matmul.mlir file.