CUDA Tile IR: Lessons from a Tile-Centric CUDA Dialect for MLIR

Matthias Springer and Lorenzo Chelini

MLIR Workshop @ EuroLLVM 2026, April 13, 2026

Agenda

- What is CUDA Tile IR?
- GEMM by Example
- Dialect Design
- Lessons Learned and Summary

Motivation: The Problem with PTX + SIMT

PTX follows the SIMT (Single-Instruction Multiple-Threads) model: each thread executes scalar instructions independently. Modern GPU hardware has outgrown this model:

Challenges with PTX and SIMT

Tensor Cores — multiple threads must cooperate on a tile; complex synchronization (barriers, fences) is needed.

TMA (Tensor Memory Accelerator) — a dedicated HW unit for indexed loads/stores using a pre-defined pattern. Requires special instructions.

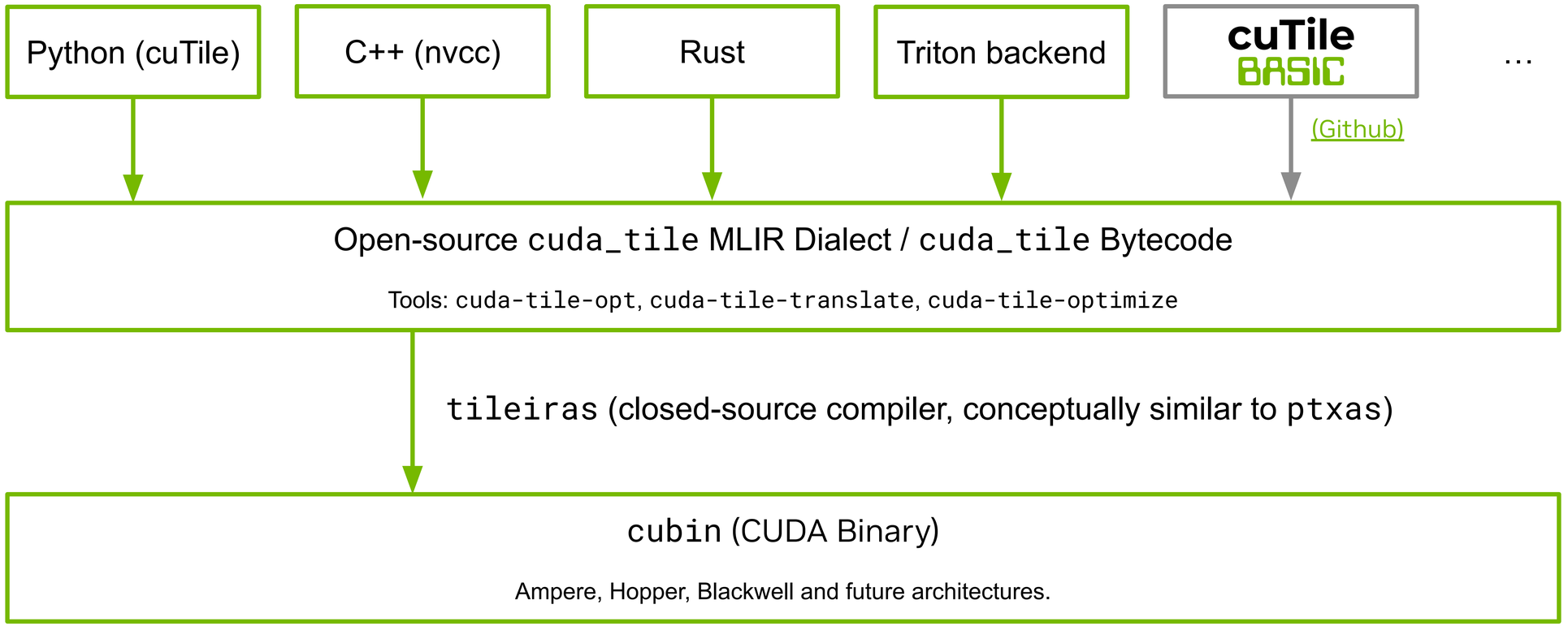
Architecture-specific instructions — the "a" in SM version names (e.g., `sm_100a`) indicates arch-specific features. Code for one arch may not work on the next.

CUDA Tile IR Goals

Ease of use — raise the abstraction above PTX.

Forward compatibility — same program runs on future architectures.

What is CUDA Tile IR?



GEMM by Example

GEMM by Example

```
cuda_tile.module @simple_gemm {
entry @gemm_64x64(
     %a_ptr: tile<ptr<f16>>, %b_ptr: tile<ptr<f16>>, %d_ptr: tile<ptr<f32>>,
     %M: tile<i32>, %N: tile<i32>, %K: tile<i32>) {

   // Inject alignment assumptions (needed for TMA eligibility).
   %a = assume #cuda_tile.div_by<16>, %a_ptr : tile<ptr<f16>>
   %b = assume #cuda_tile.div_by<16>, %b_ptr : tile<ptr<f16>>
   %d = assume #cuda_tile.div_by<16>, %d_ptr : tile<ptr<f32>>
   %Ma = assume #cuda_tile.div_by<128>, %M : tile<i32>
   %Na = assume #cuda_tile.div_by<128>, %N : tile<i32>
   %Ka = assume #cuda_tile.div_by<128>, %K : tile<i32>

   // Create tensor views: pointer + shape + strides
   %a_view = make_tensor_view %a, shape = [%Ma, %Ka], strides = [%Ka, 1]
       : tile<i32> -> tensor_view<?x?xf16, strides=[?,1]>
   %b_view = make_tensor_view %b, shape = [%Ka, %Na], strides = [1, %Ka]
       : tile<i32> -> tensor_view<?x?xf16, strides=[1,?]>
   %d_view = make_tensor_view %d, shape = [%Ma, %Na], strides = [%Na, 1]
       : tile<i32> -> tensor_view<?x?xf32, strides=[?,1]>

   // Partition views define how tiles access the data
   %a_part = make_partition_view %a_view
       : partition_view<tile=(64x64), tensor_view<?x?xf16, strides=[?,1]>>
   %b_part = make_partition_view %b_view
       : partition_view<tile=(64x64), tensor_view<?x?xf16, strides=[1,?]>>
   %d_part = make_partition_view %d_view
       : partition_view<tile=(64x64), tensor_view<?x?xf32, strides=[?,1]>>

   // Load tiles (TMA used due to alignment + partition metadata)
   %c0 = constant <i32: 0> : tile<i32>
   %a_tile, %tok_a = load_view_tko weak %a_part[%c0, %c0]
      : … -> tile<64x64xf16>, token
   %b_tile, %tok_b = load_view_tko weak %b_part[%c0, %c0]
      : … -> tile<64x64xf16>, token

   // Matrix multiply-accumulate
   %zero = constant <f32: 0.0> : tile<64x64xf32>
   %result = mmaf %a_tile, %b_tile, %zero
       : tile<64x64xf16>, tile<64x64xf16>, tile<64x64xf32>

   // Store result
   %tok_d = store_view_tko weak %result, %d_part[%c0, %c0] : … -> token
}
}
```

GEMM by Example

dialect prefix is optional (for ops and types)

cuda_tile dialect is "closed": int/float types are the only types that we reuse from MLIR.

```
cuda_tile.module @simple_gemm {
entry @gemm_64x64(
    %a_ptr: tile<ptr<f16>>, %b_ptr: tile<ptr<f16>>, %d_ptr: tile<ptr<f32>>,
    %M: tile<i32>, %N: tile<i32>, %K: tile<i32>) {

  // Inject alignment assumptions (needed for TMA eligibility).
  %a = assume #cuda_tile.div_by<16>, %a_ptr : tile<ptr<f16>>
  %b = assume #cuda_tile.div_by<16>, %b_ptr : tile<ptr<f16>>
  %d = assume #cuda_tile.div_by<16>, %d_ptr : tile<ptr<f32>>
  %Ma = assume #cuda_tile.div_by<128>, %M : tile<i32>
  %Na = assume #cuda_tile.div_by<128>, %N : tile<i32>
  %Ka = assume #cuda_tile.div_by<128>, %K : tile<i32>

  // Create tensor views: pointer + shape + strides
  %a_view = make_tensor_view %a, shape = [%Ma, %Ka], strides = [%Ka, 1]
      : tile<i32> -> tensor_view<?x?xf16, strides=[?,1]>
  %b_view = make_tensor_view %b, shape = [%Ka, %Na], strides = [1, %Ka]
      : tile<i32> -> tensor_view<?x?xf16, strides=[1,?]>
  %d_view = make_tensor_view %d, shape = [%Ma, %Na], strides = [%Na, 1]
      : tile<i32> -> tensor_view<?x?xf32, strides=[?,1]>

  // Partition views define how tiles access the data
  %a_part = make_partition_view %a_view
      : partition_view<tile=(64x64), tensor_view<?x?xf16, strides=[?,1]>>
  %b_part = make_partition_view %b_view
      : partition_view<tile=(64x64), tensor_view<?x?xf16, strides=[1,?]>>
  %d_part = make_partition_view %d_view
      : partition_view<tile=(64x64), tensor_view<?x?xf32, strides=[?,1]>>

  // Load tiles (TMA used due to alignment + partition metadata)
  %c0 = constant <i32: 0> : tile<i32>
  %a_tile, %tok_a = load_view_tko weak %a_part[%c0, %c0]
     : … -> tile<64x64xf16>, token
  %b_tile, %tok_b = load_view_tko weak %b_part[%c0, %c0]
     : … -> tile<64x64xf16>, token

  // Matrix multiply-accumulate
  %zero = constant <f32: 0.0> : tile<64x64xf32>
  %result = mmaf %a_tile, %b_tile, %zero
      : tile<64x64xf16>, tile<64x64xf16>, tile<64x64xf32>

  // Store result
  %tok_d = store_view_tko weak %result, %d_part[%c0, %c0] : … -> token
}
}
```

tile can be backed by any memory except gmem

architecture independent ops, the exact instruction is chosen by the compiler

tensor view is always in global memory

No stability guarantees for the textual IR format. We only guarantee forward compatibility for the bytecode!

Compiling a cuda_tile Program

```
$ cuda-tile-translate matmul_test.mlir -mlir-to-cudatilebc -o matmul_test.bc
$ tileiras --gpu-name=sm_100 matmul_test.bc -o matmul_test.o

$ cuobjdump -sass matmul_test.o

code for sm_100a
.target sm_100a

   Function : gemm_64x64
…
      /*0920*/   UTMALDG.2D [UR12], [UR4], desc[URZ] ;
…
      /*0be0*/   UTCHMMA gdesc[UR22], gdesc[UR24], tmem[UR17], tmem[UR14], idesc[UR15], !UPT ;
…
      /*0de0*/   UTMASTG.2D [UR8], [UR4], desc[URZ] ;
```

Blackwell-specific SASS instructions. These are undocumented.
PTX is documented, but `tileiras` will generate SASS only.

Dialect Design

Type System Overview

Every SSA value is a Tile, View or Token

cuda_tile type	**MLIR analogue**	**Details**
Number types (ints, floats). E.g., `i8`, `f32`, `f4E2M1FN`.	*(same types)*	Only a subset of integer / float types are supported.
`ptr<f8E5M2>`	`!ptr.ptr`	Pointee type must be a number type.
`tile<128x8xf32>`	`tensor<128x8xf32>`	Fully static shape. Can reside in registers, shared memory, distributed shared memory, tensor memory, etc. (You cannot control the type of memory and we won't tell you.)
`tensor_view<128x8xf32, strides=[...]>`	`memref<128x8xf32, strided<[...]>>`	Pointer into **global memory** with shape and strides. Minimal API surface. Sub-byte types are supported (dense packing).
`partition_view<tile=(2x4), tensor_view<...>>`	*(no direct analogue)*	Adds tiling metadata to a `tensor_view`. Encodes tile sizes needed for TMA descriptor generation.
`token`	`!async.token`	**Compile-time only**, no runtime representation. Used for ordering, not for waiting. (This is different from `!async.token`, which encodes the completion of an operation.)

Type System: `tile`

- **Like MLIR `tensor`**: no aliasing, no side effects on the value itself.
- Shape is always **fully static** (no `?` dimensions on tiles).
- Supports pointer element type: `tile<16x32xptr<f32>>`.
- Resides in an **unspecified memory space**: could be registers, shared, distributed shared, tensor memory, etc. The different types of memory could change with each architecture. The choice of memory space is a compiler decision.
- May not be materialized at all if the SSA value is dead.
- May be materialized in parts / pieces only, if the compiler chooses to sub-tile the program.

Type System: `tensor_view`

A Pointer into Global Memory with Shape + Stride

```
tensor_view<8192x?xf16, strides=[?, 1]>          memref<8192x?xf16, strided<[?, 1], offset: 0>
```

	`cuda_tile tensor_view`	**MLIR `memref`**
Dynamic sizes / strides	yes	yes
Offset	no	yes
Sub-byte packing	yes	? (index/size computatations lower to GEP, elements are padded to full byte)
Create operation	`make_tensor_view`	`ptr.from_ptr`
Query dimension size operation	`get_tensor_shape`	`memref.dim`
Query dimension stride operation	n/a	`memref.extract_strided_metadata`
Query pointer operation	n/a	`ptr.to_ptr` / `memref.extract_aligned_pointer_as_index`
Tiling operation	`make_partition_view`, `make_strided_view`, …	n/a (possible with `memref.expand_shape` + `memref.transpose`)

The `tensor_view` API is intentionally smaller API than memref. No slicing / subview / etc. We may extend the API as needed.

Type System: `tensor_view`

A Pointer into Global Memory with Shape + Stride

`tensor_view<8192x?xf16, strides=[?, 1]>` `memref<8192x?xf16, strided<[?, 1], offset: 0>`

	`cuda_tile tensor_view`	**MLIR `memref`**
Dynamic sizes / strides	yes	yes
Offset	no	yes
Sub-byte packing	yes	? (index/size computatations lower to GEP, elements are padded to full byte)
Create operation	`make_tensor_view`	`ptr.from_ptr`
Query dimension size operation	`get_tensor_shape`	`memref.dim`
Query dimension stride operation	n/a	`_strided_metadata`
Query pointer operation	n/a	`_aligned_pointer_as_index`
Tiling operation	`make_partition_view`, `make_strided_view`, …	n/a (possible with `memref.expand_shape` + `memref.transpose`)

Keep information readily available for easy TMA descriptor generation.

The `tensor_view` API is intentionally smaller API than memref. No slicing / subview / etc. We may extend the API as needed.

Load / Store: Two Flavors

View-based Load vs. Pointer-based Load

```
%a_view = make_tensor_view %ptr, shape = [128, 64], strides = [%S, 1]
    : tile<i32> -> tensor_view<128x64xf16, strides=[?,1]>
%a_part = make_partition_view %a_view
    : partition_view<tile=(64x64),
                     tensor_view<128x64xf16, strides=[?,1]>>
%a_tile, %tok_a = load_view_tko weak %a_part[%c0, %c0]
      : … -> tile<64x64xf16>, token
```

View-based (Structured) Load

equivalent

```
// Row indices: 0..63 broadcast across columns
%row_1d = iota : tile<64xi32>
%row_col = reshape %row_1d : tile<64xi32> -> tile<64x1xi32>
%row_idx = broadcast %row_col : tile<64x1xi32> -> tile<64x64xi32>
// Column indices: 0..63 broadcast across rows
%col_1d = iota : tile<64xi32>
%col_row = reshape %col_1d : tile<64xi32> -> tile<1x64xi32>
%col_idx = broadcast %col_row : tile<1x64xi32> -> tile<64x64xi32>
// Broadcast stride to 64x64
%S_2d = reshape %S : tile<i32> -> tile<1x1xi32>
%S_bcast = broadcast %S_2d : tile<1x1xi32> -> tile<64x64xi32>
// Compute element offsets: row * S + col
%row_offset = muli %row_idx, %S_bcast : tile<64x64xi32>
%offsets = addi %row_offset, %col_idx : tile<64x64xi32>
// Build pointer tile and apply offsets
%p_2d = reshape %ptr : tile<ptr<f16>> -> tile<1x1xptr<f16>>
%p_bcast = broadcast %p_2d : tile<1x1xptr<f16>> -> tile<64x64xptr<f16>>
%ptrs = offset %p_bcast, %offsets :
    tile<64x64xptr<f16>>, tile<64x64xi32> -> tile<64x64xptr<f16>>
// Load tile
%a_tile, %tok_a = load_ptr_tko weak %ptrs :
    tile<64x64xptr<f16>> -> tile<64x64xf16>, token
```

Pointer-based (Gather-style) Load
(Triton style)

Load / Store: Two Flavors

View-based Load vs. Pointer-based Load

View-based Load	Pointer-based Load
`load_view_tko`	`load_ptr_tko`
token-based	token-based
TMA-friendly by construction	full flexibility
View type encodes tile sizes, strides, dim mappings.	Recovering structure from pointer arithmetics is brittle.
Assumes on sizes, strides, base pointer give the compiler everything it needs.	Alignment + contiguity is difficult to prove from arithmetics and difficult to explain to the users via remarks (when missing).
Sub-byte types are supported.	Sub-byte types with dense packing are not supported. (Pointers always point to the beginning of a byte.)
Short, concise IR	Verbose IR

View-style loads/stores with assume-based alignment are the recommended path for performance-critical code.

The compiler can decide whether to use TMA or fall back to thread-level loads based on the available metadata.

CUDA Tile IR Memory Model

tile<64xptr<f32>>

Tile-thread execution

```
%cst = constant <f32: [0.0, 1.0, 2.0, …]> : tile<64xf32>
%token_result = store_ptr_tko weak %ptr, %cst -> token
%data, %token_result_1 = load_ptr_tko weak %ptr -> tile<64xf32>, token
```

CUDA Tile IR Memory Model

A Weak Memory Model

Tile-thread execution

```
%cst = constant <f32: [0.0, 1.0, 2.0, …]> : tile<64xf32>
%token_result = store_ptr_tko weak %ptr, %cst -> token
%data, %token_result_1 = load_ptr_tko weak %ptr -> tile<64xf32>, token
```

One possible lowering:

Thread ID	T0	…	T31	T32	…	T63
	`st.global.f32 [63]`		`st.global.f32 [32]`	`st.global.f32 [31]`		`st.global.f32 [0]`
	`ld.global.f32 [0]`		`ld.global.f32 [31]`	`ld.global.f32 [32]`		`ld.global.f32 [63]`

`ld.global` and `st.global` are SIMT-level instructions.

CUDA Tile IR Memory Model

A Weak Memory Model

Tile-thread execution

```
%cst = constant <f32: [0.0, 1.0, 2.0, ...]> : tile<64xf32>
%token_result = store_ptr_tko weak %ptr, %cst -> token
%data, %token_result_1 = load_ptr_tko weak %ptr -> tile<64xf32>, token
```

Absence of token dependency allows the compiler to parallelize `load` / `store`.

One possible lowering:

Thread ID	T0	...	T31	T32	...	T63
	`ld.global.f32 [0]`		`ld.global.f32 [31]`	`st.global.f32 [0]`		`st.global.f32 [31]`
	`ld.global.f32 [32]`		`ld.global.f32 [63]`	`st.global.f32 [32]`		`st.global.f32 [63]`

`ld.global` and `st.global` are SIMT-level instructions.

CUDA Tile IR Memory Model

A Weak Memory Model

Tile-thread execution

```
%cst = constant <f32: [0.0, 1.0, 2.0, …]> : tile<64xf32>
%token_result = store_ptr_tko weak %ptr, %cst -> token
%data, %token_result_1 = load_ptr_tko weak %ptr -> tile<64xf32>, token
```

One possible lowering:

Thread ID	T0	…	T31	T32	…	T63
	`st.global.f32 [0]`		`st.global.f32 [31]`	`st.global.f32 [32]`		`st.global.f32 [63]`
	`ld.global.f32 [0]`		`ld.global.f32 [31]`	`ld.global.f32 [32]`		`ld.global.f32 [63]`

`ld.global` and `st.global` are SIMT-level instructions.

CUDA Tile IR Memory Model

A Weak Memory Model

Tile-thread execution

```
%cst = constant <f32: [0.0, 1.0, 2.0, …]> : tile<63xf32>
%token_result = store_ptr_tko weak %ptr, %cst -> token
%data, %token_result_1 = load_ptr_tko weak %ptr -> tile<64xf32>, token
```

One possible lowering:

Thread ID	T0	…	T31	T32	…	T63
	`cp.async`					
	`cp.async`					

could be a TMA load into shared memory

`cp.async.bulk.tensor` is asynchronous and issued by a single thread.

CUDA Tile IR Memory Model

A Weak Memory Model

Tile-thread execution

```
%cst = constant <f32: [0.0, 1.0, 2.0, …]> : tile<64xf32>
%token_result = store_ptr_tko weak %ptr, %cst -> token
%data, %token_result_1 = load_ptr_tko weak %ptr token=%token_result -> tile<64xf32>, token
```

Token chain op1→op2: Within a tile block, as if all effects of op1 are complete before op2 starts.

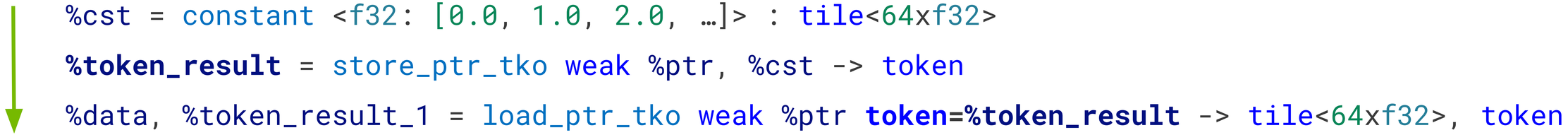
Thread ID	T0	…	T31	T32	…	T63
	`st.global.f32 [0]`		`st.global.f32 [31]`	`st.global.f32 [32]`		`st.global.f32 [63]`
	`ld.global.f32 [0]`		`ld.global.f32 [31]`	`ld.global.f32 [32]`		`ld.global.f32 [63]`

Key insight: Tokens control the degree to which the compiler can parallelize and/or reorder the program within a tile thread. For performance, we insert barriers/fences only when tokens require it. The programmer models dependencies explicitly.

CUDA Tile IR Memory Model

A Weak Memory Model

Tile-thread execution



Thread ID	T0	…	T31	T32	…	T63
	st.global.f32 [0]		st.global.f32 [31]	st.global.f32 [32]		st.global.f32 [63]
	bar.sync		bar.sync	bar.sync		bar.sync
	ld.global.f32 [63]		ld.global.f32 [32]	ld.global.f32 [31]		ld.global.f32 [0]

Key insight: Tokens control the degree to which the compiler can parallelize and/or reorder the program within a tile thread. For performance, we insert barriers/fences only when tokens require it. The programmer models dependencies explicitly.

`token` vs. `!async.token`

	`!cuda_tile.token`	`!async.token`
Runtime representation	**None** (purely compile-time)	Runtime library calls
Purpose	Constrain compiler reordering	Wait for async completion
Can you "wait" on it?	**No**	Yes (`async.await`)
Combining tokens	`join_token`	`async.create_groups`, async.add_to_group

Note: GPUs have asynchronous instructions (e.g., `cp.async`), but we don't expose them on the cuda_tile level. There are no asynchronous operations in cuda_tile. Overlapping loads/stores/arithmetics/tensor core computation is the compiler's job.

Load / Store Semantics

Weak, Acquire and Release

Weak: No concurrent access to the source/destination location.

Acquire: No reads or writes in the current thread can be reordered before the load (if connected by token).

Release: No reads or writes in the current thread can be reordered after the store (if connected by token).

```
%cst = constant <f32: 13.1>
%one = constant <i32: 1>
```

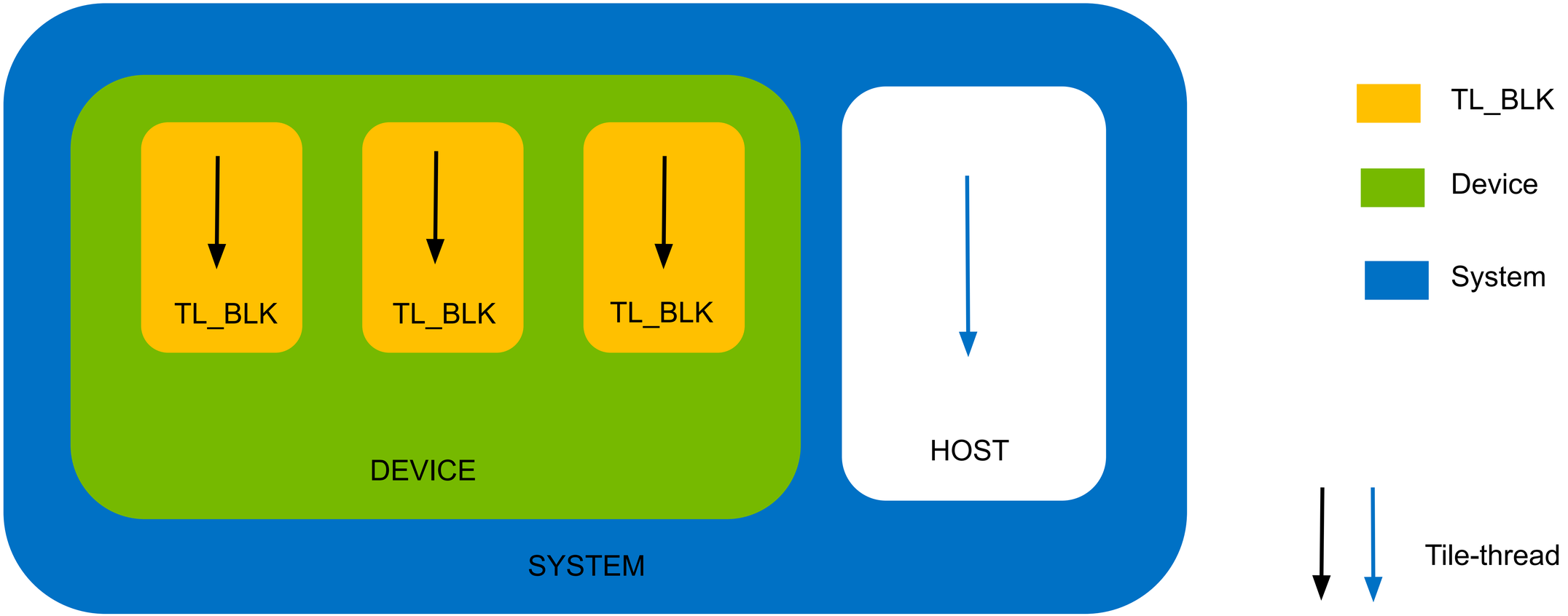
```
// Tile block 0: producer
%tk = store_ptr_tko weak %ptr_data, %cst -> token
%tk1 = store_ptr_tko release device %ptr_flag, %one token=%tk -> token
```

```
// Tile block 1: consumer
%flag, %tk2 = load_ptr_tko acquire device %ptr_flag -> tile<i32>, token
%data, %tk3 = load_ptr_tko weak %ptr_data token=%tk2 -> tile<f32>, token
```

Key insight: Semantics and Scope allow to establish a casualty between tile-threads that establish a correct synchronization.

Load / Store Scope

Set of tile-threads that may interact directly with an operation



assume: Metadata Injection

`cuda_tile.assume` injects metadata about SSA values. The compiler uses it to enable fast-path optimizations. Similar in spirit to `llvm.intr.assume` and `memref.assume_alignment`. Remarks notify programmers of missed optimizations and the reason.

```
// Pointer alignment (e.g., enables TMA — requires 16-byte aligned pointers)
%ptr_a = assume #cuda_tile.div_by<16>, %ptr : tile<ptr<f16>>

// Shaped divisibility (e.g., enables vectorized loads/stores)
%stride = assume #cuda_tile.div_by<32>, %s : tile<2xi16>
%p = assume #cuda_tile.div_by<4, every 4 along 1>, %ptrs : tile<1x8x8xptr<f32>>

// Value range bounds (e.g., required to prove that TMA indexing does not overflow)
%idx = assume #cuda_tile.bounded<0, 42>, %i : tile<i16>

// Uniform values across tile dimensions
%u = assume #cuda_tile.same_elements<[1, 4, 2]>, %ptrs : tile<1x8x8xptr<f32>>
```

`assume`: Design Trade-off

Where should metadata live? In the type system or attached via `assume` + dataflow analysis?

Metadata in Types

- Always available at the use site
- No analysis needed
- Type system becomes complex
- Custom types needed (e.g., cannot reuse `i32`)
- Type conversions / casts needed (e.g., yielding integers with different divisibility from `if` branches)
- Transformations must keep metadata correct
- Hard to compose different metadata

`assume` + Data Flow Analysis

- Types remain simple
- Composable: multiple `assume`s on the same value, back to back
- Metadata can flow through operations
- Requires dataflow analysis
- Analysis results may be incomplete

cuda_tile chose `assume` + dataflow analysis: simpler types, more flexible metadata, at the cost of requiring the compiler to propagate information.

Arithmetic Operations: Vendor-Specific Semantics

cuda_tile arithmetic looks similar to the `arith` dialect on the surface...

```
%sum = addi %a, %b overflow<no_signed_wrap> : tile<i16>
%sum = addf %a, %b rounding<zero> flush_to_zero : tile<f32>
%prod = mulf %x, %y : tile<128x64xf16>
```

...but details differ: rounding modes, denormal flushing, and fast-math flags can be specified per-op and are sometimes vendor-specific (example).

This prevented us from using `arith` dialect ops (which are platform-independent) internally in some places.

The unsolved problem: canonicalization reuse. We'd like to use `arith` ops internally in the compiler. The `arith` dialect has a large set of canonicalizations and foldings. Reusing these for cuda_tile arithmetic with vendor-specific semantics is an open problem. **We'd love to hear ideas from the community.**

Matrix Multiply: `mmaf`

`cuda_tile.mmaf` accepts large tile shapes. The programmer can write a single large MMA. The compiler breaks it into HW-supported tensor core applications:

1. Selects the best HW-supported (sub-)tile size for the target architecture.
2. Decomposes the large MMA into multiple tensor core applications.
3. Handles memory space copies (e.g., copy to TMEM) and data distribution (layout).

```
// Simple MMA
%result = mmaf %a_tile, %b_tile, %zero : tile<64x64xf32>, tile<64x64xf32>, tile<64x64xf32>

// Block-scaled MMA (Blackwell) — microscaling with FP8/FP4 types
%result = mmaf_scaled %lhs, %rhs, %acc, %lhs_scale, %rhs_scale
    : tile<128x128xf8E5M2>, tile<128x128xf8E5M2>, tile<128x128xf32>, tile<128x4xf8E8M0FNU>,
tile<4x128xf8E8M0FNU>

// Also: mmai (integer), mixed types, FP4 (f4E2M1FN), ...
```

Optimization Remarks

```
$ tileiras  --gpu-name=sm_100 matmul_test.bc -o matmul_test.o --remarks-passed=all --remarks-failed=all --remark-format=command-line
in function: gemm_64x64:

  category: Memory
    remark[passed]: Load operation successfully optimized to use TMA
      --> loc(unknown)
       |
       = name: RemarkMemoryLoadOptimized
       = note: Instruction = TMA Load instruction
       = note: Shape = [64, 64]

    remark[passed]: Load operation successfully optimized to use TMA
      --> loc(unknown)
       |
       = name: RemarkMemoryLoadOptimized
       = note: Instruction = TMA Load instruction
       = note: Shape = [64, 64]

    remark[passed]: Store operation successfully optimized to use TMA
      --> loc(unknown)
       |
       = name: RemarkMemoryStoreOptimized
       = note: Instruction = TMA Store instruction
       = note: Shape = [32, 64]
```

Various Combinations						Shapes Supported		
.kind::*	Has .ws	CTA Group	Sparsity	dtype	atype/btype			
kind::f16	No .ws	1	Dense	.f16	.f16	64xNxK 128xNxK	N = {8, 16, 24, ... 256} steps of 8	K = 16
				.f32	.f16, .bf16			
			Sparse	.f16	.f16			K = 32
				.f32	.f16, .bf16			

```
%result = mmaf %a_tile, %b_tile, %zero
     : tile<64x64xf32>, tile<64x64xf32>, tile<64x64xf32>
```

```
category: Tensor-core
  remark[failed]: MMA operation failed to optimize to use Tensor Cores, it is using FMA instructions instead
    --> loc(unknown)
     |
     = name: RemarkTensorCoreMMA
     = note: Instruction = FMA
     = note: Shape = [1, 1, 1]
```

Optimization Remarks

```
$ tileiras  --gpu-name=sm_100 matmul_test.bc -o matmul_test.o --remarks-passed=all --remarks-failed=all --remark-format=command-line
in function: gemm_64x64:

  category: Memory
    remark[passed]: Load operation successfully optimized to use TMA
      --> loc(unknown)
       |
       = name: RemarkMemoryLoadOptimized
       = note: Instruction = TMA Load instruction
       = note: Shape = [64, 64]

    remark[passed]: Load operation successfully optimized to use TMA
      --> loc(unknown)
       |
       = name: RemarkMemoryLoadOptimized
       = note: Instruction = TMA Load instruction
       = note: Shape = [64, 64]

    remark[passed]: Store operation successfully optimized to use TMA
      --> loc(unknown)
       |
       = name: RemarkMemoryStoreOptimized
       = note: Instruction = TMA Store instruction
       = note: Shape = [32, 64]
```

```
%result = mmaf %a_tile, %b_tile, %zero
     : tile<64x64xf16>, tile<64x64xf16>, tile<64x64xf32>
```

```
category: Tensor-core
  remark[passed]: MMA operation successfully optimized to use Tensor Cores
    --> loc(unknown)
     |
     = name: RemarkTensorCoreMMA
     = note: Instruction = Tensor-core SM100
     = note: Shape = [64, 64, 16]
     = note: NumCTAs = 1
```

Breaking Exit from Loops

- Language frontends (e.g., Python) have early exit and we need a way to map those to cuda_tile. Only `cuda_tile.loop` supports early exit. `cuda_tile.for` does not support early exit.
- Example: Work stealing / persistent kernels

```
loop {
  work_id = try_get_new_work_id()
  if work_id == NULL: break
  do_work(work_id)
}
```

- Customers have asked for this feature.
- Early exit can make optimizations are complicated. E.g., software pipelining would be non-trivial (if we were to allow early exit on `cuda_tile.for`).

Breaking Exit from Loops

infinite loop

```
entry @loop_with_breaking_exit(%lb: tile<i32>, %ub: tile<i32>) {
 %result = loop iter_values(%x = %lb) : tile<i32> -> tile<i32> {
   %done = cmpi equal %x, %ub, signed : tile<i32> -> tile<i1>
   if %done {
     break %x : tile<i32>
   }
   %one = constant <i32: 1> : tile<i32>
   %x1 = addi %x, %one : tile<i32>
   continue %x1 : tile<i32>
 }
 return
}
```

exit the closest `cuda_tile.loop` with the given results

run another iteration of the closest `cuda_tile.loop` with the given loop-carried variables

- `cuda_tile.loop` op is conceptually similar to `scf.while`, but with a single region.
- MLIR does not support breaking exit from regions. Our dialect design violates the MLIR LangRef.
- Help is on the way: [RFC] Region-based control-flow with early exits in MLIR and a prototype (#166688).
  If you are interested in breaking exit from regions: **Come to the roundtable on Wednesday 10:30-11:00 @ Ulster.**
- Until then: `cuda_tile.loop` cannot implement `RegionBranchOpInterface`. No support for dataflow analysis etc.
- Note: Our MLIR dialect is mainly "MLIR/C++ bindings for the Tile IR bytecode". But we aim for more in the future...

Breaking Exit from Loops

```
entry @loop_with_breaking_exit(%lb: tile<i32>, %ub: tile<i32>) {
 %result = loop iter_values(%x = %lb) : tile<i32> -> tile<i32> {
   %done = cmpi equal %x, %ub, signed : tile<i32> -> tile<i1>
   if %done {
     break %x : tile<i32>
   }
   %one = constant <i32: 1> : tile<i32>
   %x1 = addi %x, %one : tile<i32>
   continue %x1 : tile<i32>
 }
 return
}
```

one `iter_arg` for the "break" case and one for the "continue" case

```
%tmp, result = scf.while (%x = %lb) : (i32) -> (i32, i32) {
  %done = arith.cmpi eq, %x, %ub : i32
  %x1 = scf.if %done -> (i32) {
    scf.yield %x : i32
  } else {
    %one = arith.constant dense<1> : i32
    %x2 = arith.addi %x, %one : i32
    scf.yield %x2 : i32
  }
  %not_done = arith.xor %done, %true : i1
  scf.condition(%not_done) %x1, %x : i32
} do {
^bb0(%arg0: i32, %arg1: i32):
  scf.yield %arg0 : i32
}
```

IR that follows after a "break" branch goes into the other "if" branch

pass-through "after" region

In practice, this transformation is more complicated. We just want to show that it's always possible to rewrite breaking exit with existing `scf` dialect ops.

Bytecode: Versioned and Forward-Compatible

cuda_tile has its own versioned bytecode. Why not MLIR bytecode?

- **We only have a single dialect**. The full generality of MLIR bytecode isn't needed.
- Denser, simpler format. First reader/writer was implemented in a few days.
- **Versioning in TableGen.** The bytecode version that introduced each op/attribute is encoded directly in TableGen.
- **Automatic migration rules**, generated from `DefaultValuedAttr` etc. when new attributes are added to existing ops.
- Being able to read/write bytecode **without an MLIR dependency.** (We have a Python reader/writer.)

cuda_tile bytecode compiled for Ampere can be recompiled for Hopper or Blackwell by `tileiras`. Forward compatibility by design.

```
# Text → bytecode → text round-trip
cuda-tile-translate -mlir-to-cudatilebc -bytecode-version=13.1 input.mlir -o out.bc
cuda-tile-translate -cudatilebc-to-mlir out.bc -o output.mlir
```

cuda-tile-optimize

Experimental Optimizations for cuda_tile IR

- FMA Fusion Pass: `(a * b) + c ⇒ FMA(a, b, c)`
  - Does not preserve numerics (rounding modes).
- Standard optimizations: Canonicalizer, Common Subexpression Elimination (CSE), Loop-invariant Code Motion (LICM)
  - With support for cuda_tile.loop, which has early exit.
- Loop Splitting Pass: Split single `cuda_tile.for` loop in two loops when there's an if-comparison on the IV.
- Redundant Token Elimination: Accesses to the same view with non-overlapping indices don't need a token.
- Integer Simplification Pass: E.g., Python's `mod` is different from `cuda_tile.remf` and requires us to insert select ops. These can be eliminated when all operands are positive.

Lessons Learned and Summary

Optimization Hints: Tunable and Discardable

The compiler can't always get it right. Expose knobs for auto-tuning.

Performance-critical compiler decisions (buffer assignment, second-level tiling, warp specialization, software pipelining) sometimes need extra knowledge. Key properties:

- Hints may not be performant on other architectures and can always be discarded.
- Hints **can be auto-tuned** alongside tile sizes by the frontend.
- Architecture-specific keys (`sm_100`, `sm_120`) allow per-target tuning.

```
// Entry-level hints (per-SM architecture)
entry @kernel(%arg0: tile<ptr<f32>>)
   optimization_hints=<sm_100 = {num_cta_in_cga = 2},
                       sm_120 = {num_cta_in_cga = 2, occupancy = 2}> { ... }

// Op-level hints (e.g., disable TMA, specify latency)
%t = store_view_tko weak %tile, %view[%i, %j]
   optimization_hints=<sm_100={allow_tma = false, latency = 5}> -> token
```

Open-Sourcing an MLIR Dialect Outside `llvm-project`

The cuda_tile dialect lives outside the LLVM monorepo. Main concern: LLVM API stability.

- MLIR C++ APIs can change. Will the dialect build with only a specific LLVM commit?
- We track a pinned LLVM commit and regularly update.
- Reuse as many MLIR abstractions as possible internally, but keep full control externally. The external dialect is versioned and follows a specification.

	External IR	**Internal IR**
Versioned?	**Yes – specification + bytecode**	No
MLIR dialects used	`cuda_tile` only	Whatever makes our work easier. (`arith`, `scf`, `vector`, `llvm`, …)
Stability	Full backward compatibility	No guarantees – IR design can be refactored freely and may change between releases
Compatibility with MLIR versions	Compatible as long as C++ APIs, ODS, builtin types do not change.	Bumping the LLVM version requires a "full" LLVM integrate. (Not the user's concern.)

Lessons Learned

1. **Own your external interface.** Reuse MLIR internally; version and control the external dialect.
2. **View-style loads > pattern matching.** Structured access patterns (partition views) beat recovering tiling info from pointer arithmetic.
3. **Tokens > side effects for GPUs.** Compile-time–only tokens give fine-grained control over ordering and enable the compiler to insert minimal synchronization.
4. **`assume` op > complex types**. Metadata injection via `cuda_tile.assume` + dataflow analysis keeps types simple while enabling HW-specific fast paths (TMA, etc.).
5. **Custom bytecode can be simple.** A single-dialect bytecode with TableGen-driven versioning is easier than adapting the general MLIR bytecode format.
6. **Canonicalization reuse is unsolved.** Vendor-specific arithmetic semantics make it hard to reuse `arith` foldings.

Questions?

cuda_tile, `tileiras`
PTX, `ptxas`
`cuda-tile-opt`, `cuda-tile-translate`
`cuda-tile-optimize`
cuTile, nvcc, Rust, Triton backend
SASS
SIMT
TMA (Tensor Memory Accelerator)
GEMM, `mmaf`, `mmaf_scaled`
Tensor Cores
Forward Compatibility
Bytecode Versioning
`tensor_view`
`partition_view`
`token`
`!async.token`
Integer / Float Types
Side Effects
Pointer-based Load vs. View-based Load
Sub-byte Packing
`tile`
Memory Space

Weak Memory Model
Acquire-Release Semantics
Assume Metadata vs. Rich Type System
`arith` Dialect
Canonicalization
Optimization Remarks
Optimization Hints
Breaking Exit from Loops
Open-Sourcing an MLIR Dialect
Pinned LLVM Version
Token Elimination
Buffer Aliasing
Loop Splitting
FMA Fusion
Loop-Invariant Code Motion (LICM)

The CUDA Tile IR team (and other compiler teams at NVIDIA) are hiring.