Introduction to CUDA Tile

January 2026

Agenda

- Why evolve the programming model?
  - What are we trying to abstract?
- Introducing CUDA Tile
  - TileIR
  - Programming Model
  - Evolving a GEMM
  - Extending the CUDA Platform
- TileGym
- Performance Analysis
- Next Steps & Questions

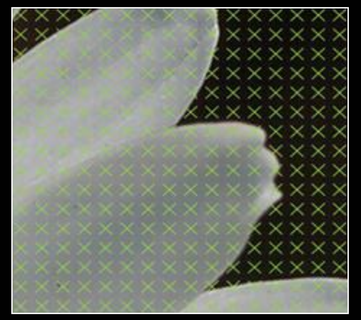
Grid-Level

Easy acceleration of bulk data operations

System splits work into blocks, divides data into tiles, & maps both onto threads

Thread-Level

Full control for all parallel workloads



User splits work into blocks, divides data into tiles, & maps both onto threads

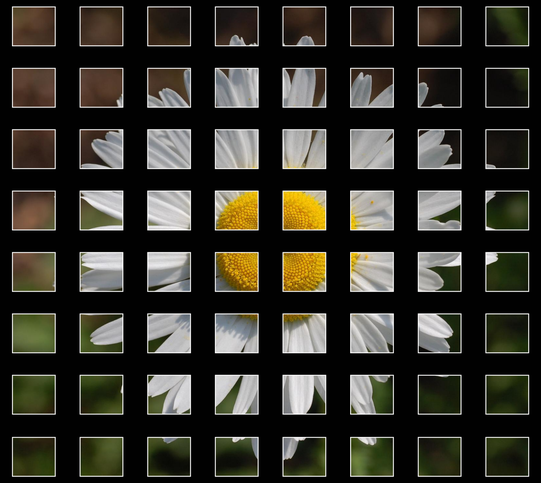
Grid-Level

Easy acceleration of bulk data operations

System splits work into blocks, divides data into tiles, & maps both onto threads

Tile-Level

Productive, portable, & performant array ops

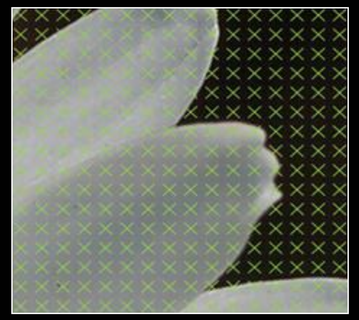


User splits work into blocks & divides data into tiles

System maps both onto threads

Thread-Level

Full control for all parallel workloads



User splits work into blocks, divides data into tiles, & maps both onto threads

Tile based programming for GPU was first brought by Philippe Tillet & al. in
Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations

Grid-Level

cuBLAS

Tile-Level

Thread-Level

NVIDIA CUDA C++

Grid-Level

cuBLAS

Tile-Level

CUDA Tile

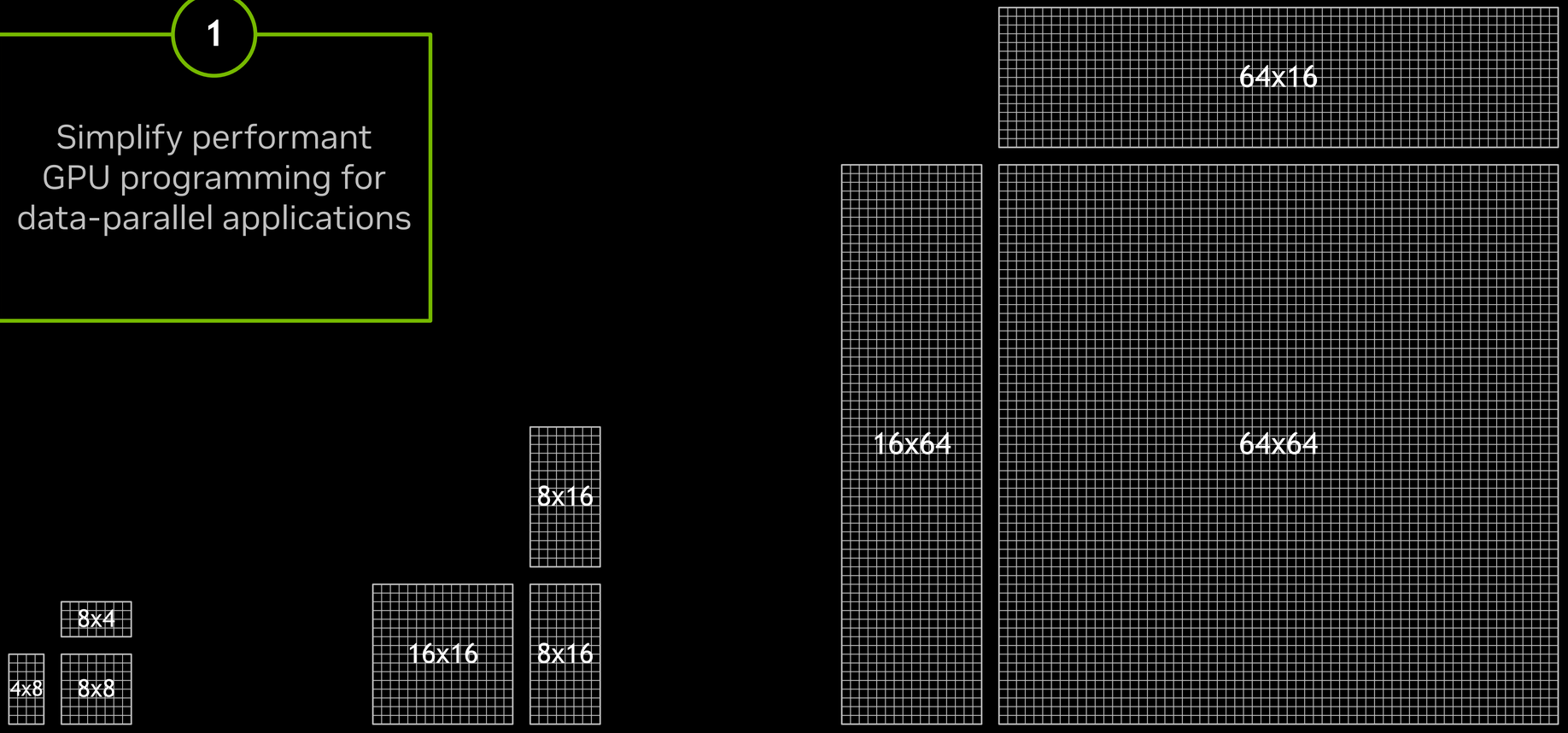
Thread-Level

NVIDIA CUDA C++

Why evolve again?

1

Simplify performant GPU programming for data-parallel applications



Volta Tensor Core
¼ warp (8 threads)

Ampere Tensor Core
1 warp (32 threads)

Hopper Tensor Core
4 warps (128 threads)

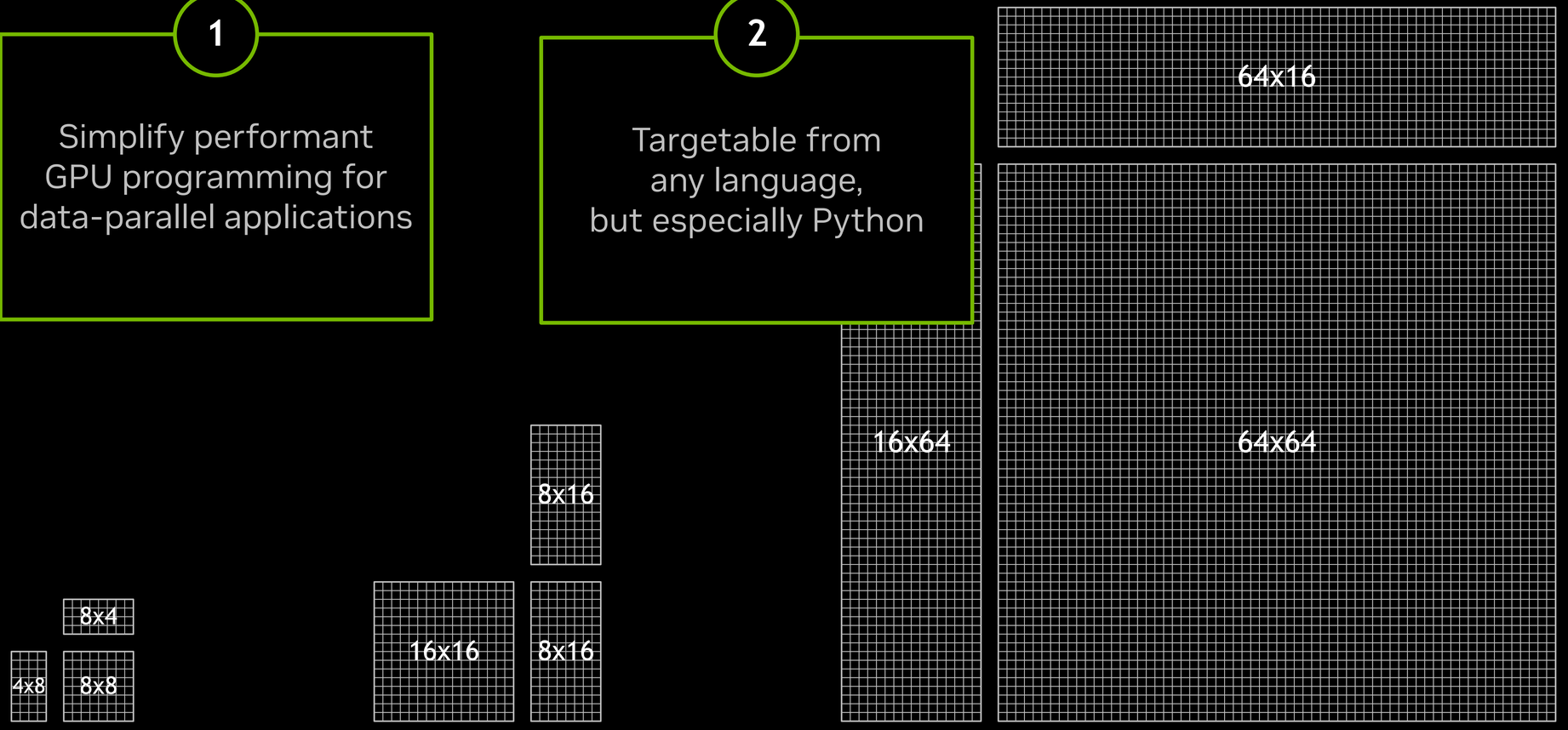
Why evolve again?

1

Simplify performant GPU programming for data-parallel applications

2

Targetable from any language, but especially Python



Volta Tensor Core
¼ warp (8 threads)

Ampere Tensor Core
1 warp (32 threads)

Hopper Tensor Core
4 warps (128 threads)

Why evolve again?

1

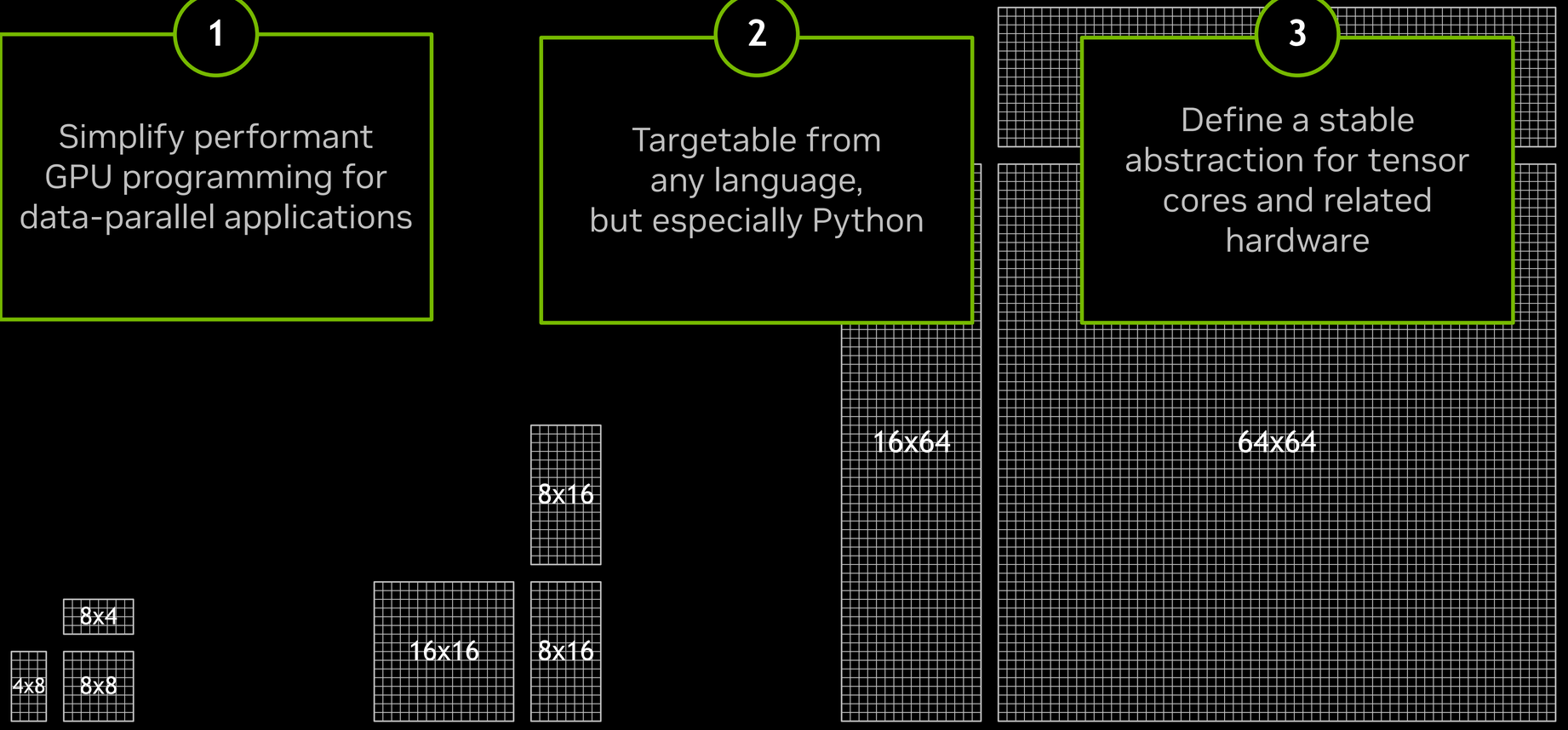
Simplify performant GPU programming for data-parallel applications

2

Targetable from any language, but especially Python

3

Define a stable abstraction for tensor cores and related hardware



Volta Tensor Core
¼ warp (8 threads)

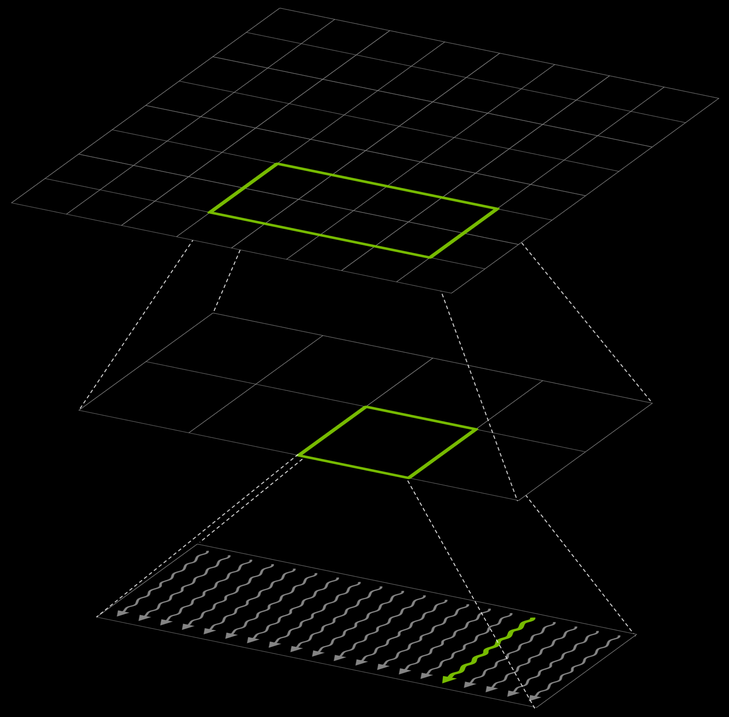
Ampere Tensor Core
1 warp (32 threads)

Hopper Tensor Core
4 warps (128 threads)

What do we abstract?

Unit of	SIMT
Execution	Thread

Thread Grid



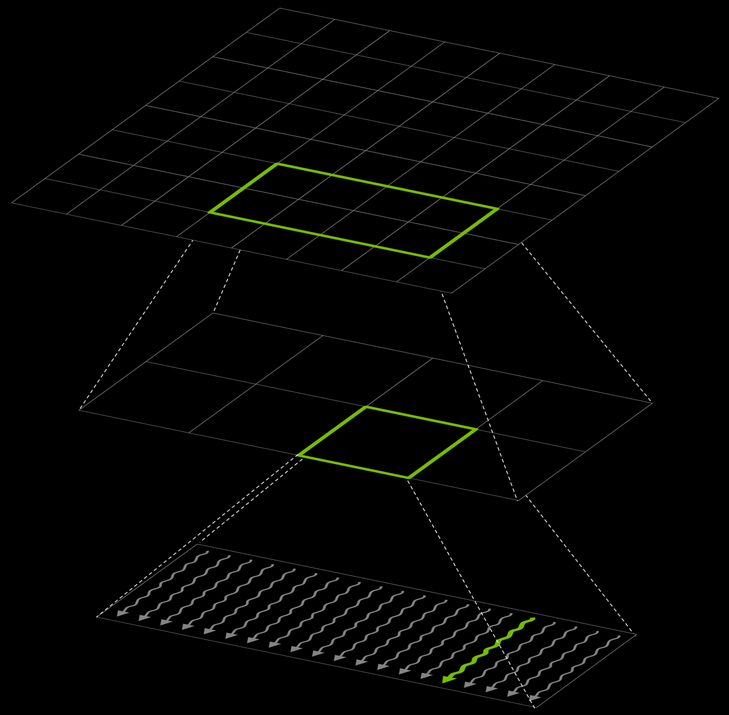
Blocks of Warps

Warps of Threads

Threads

Unit of	SIMT	Tile
Execution	Thread	Block

Thread Grid

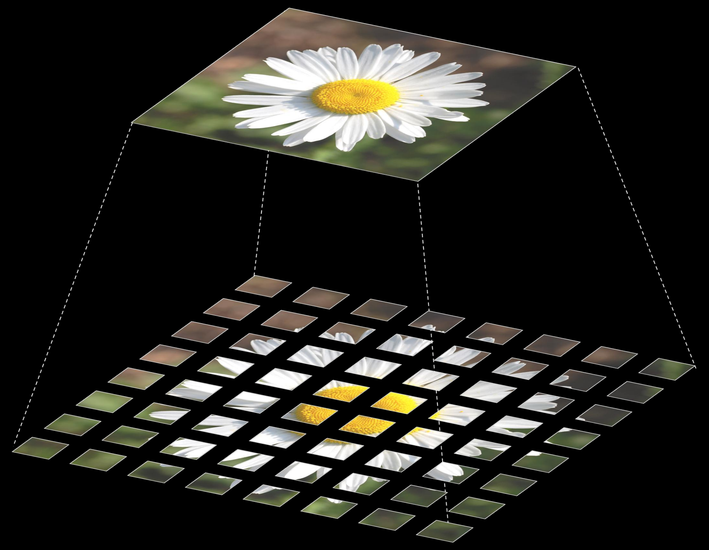


Blocks of Warps

Warps of Threads

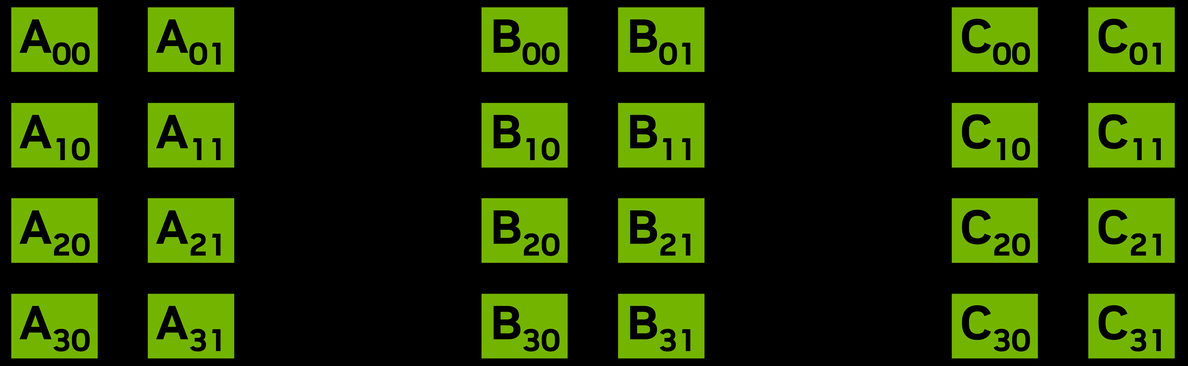
Threads

Tile Grid

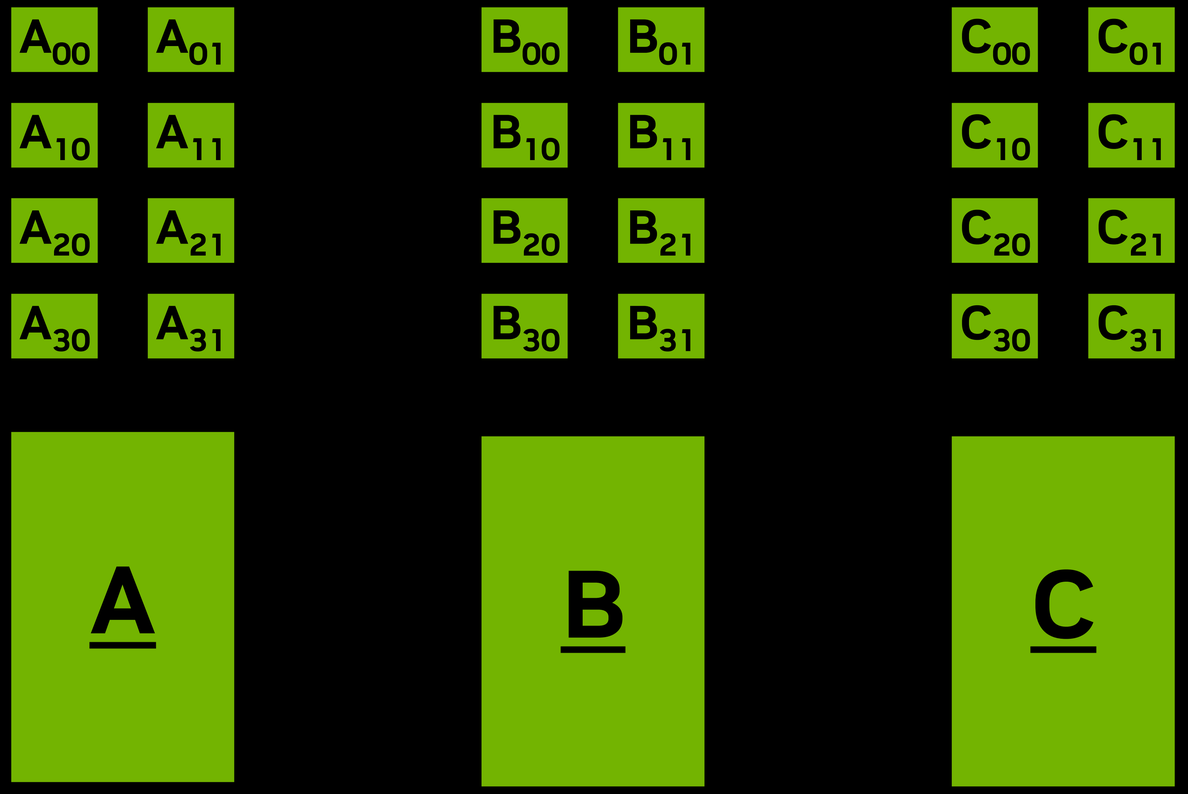


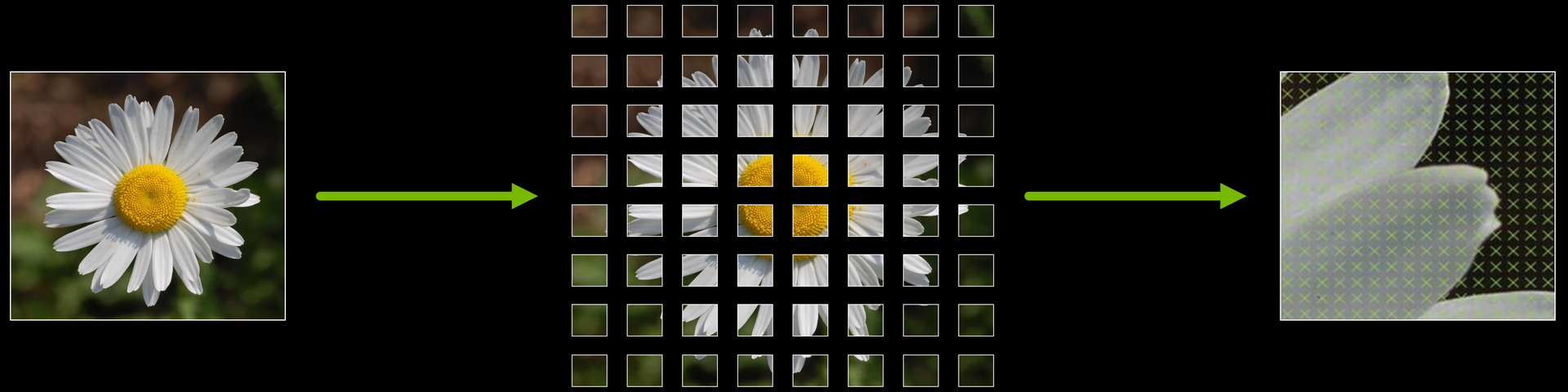
Tile Blocks

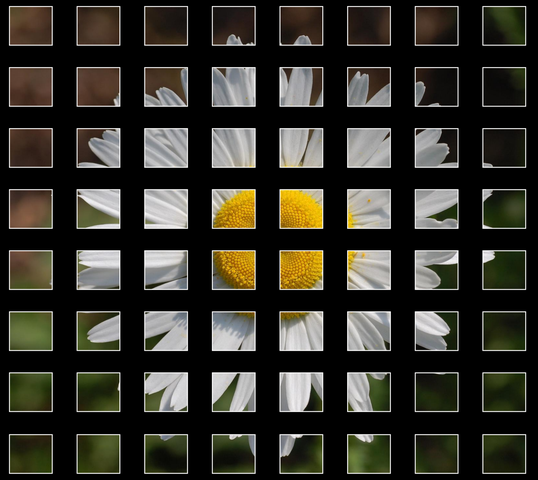
Unit of	SIMT	Tile
Execution	Thread	Block
Data	Element	

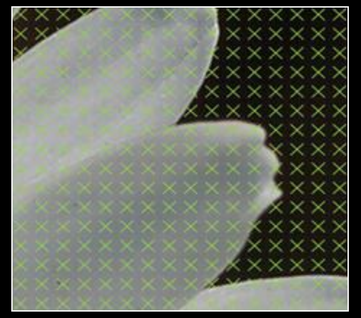


Unit of	SIMT	Tile
Execution	Thread	Block
Data	Element	Tile

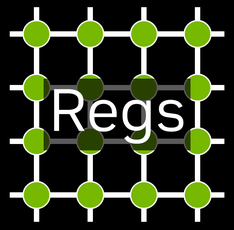


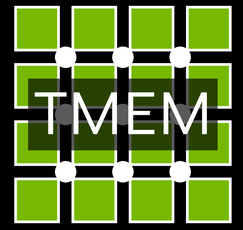


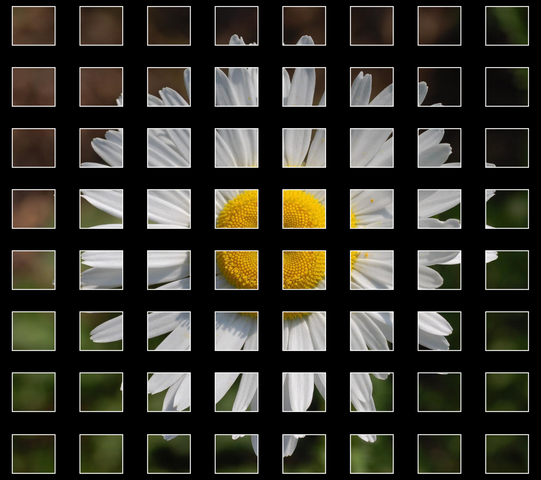


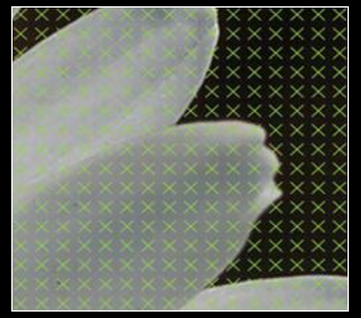


Tile storage?

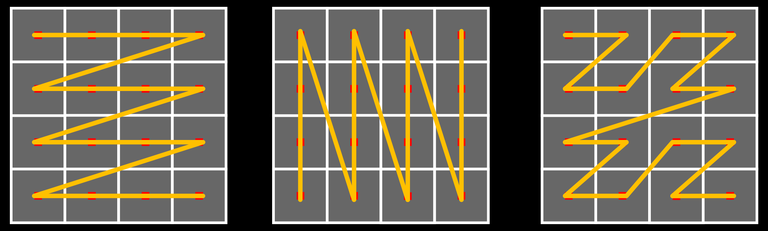






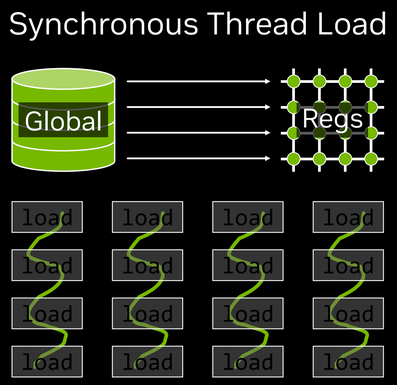


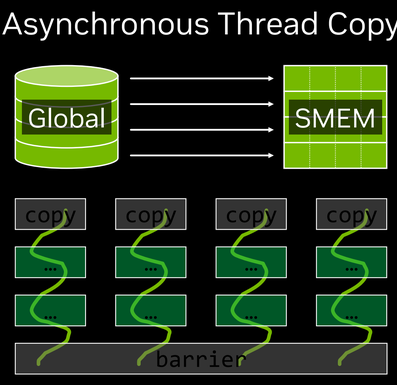
Tile memory layout?

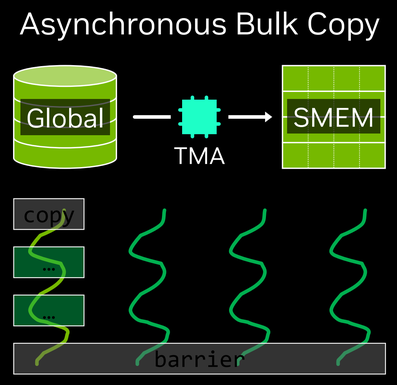


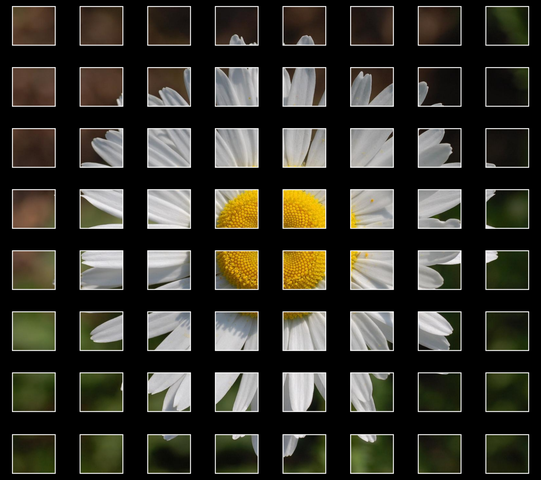
Tile storage?

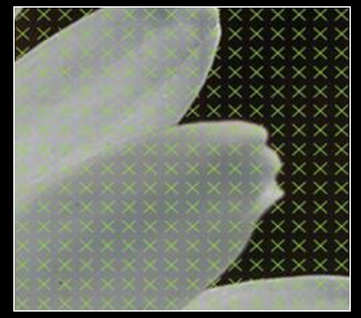
Memory movement strategy?



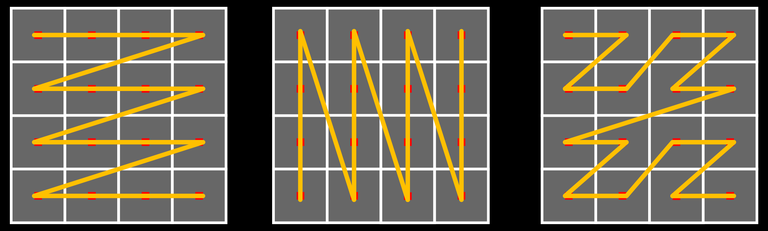








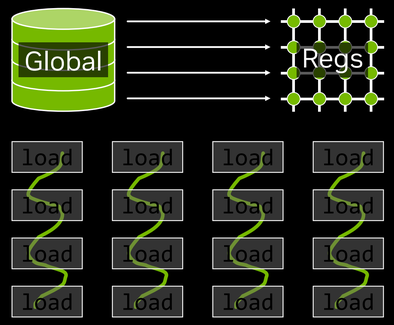
Tile memory layout?



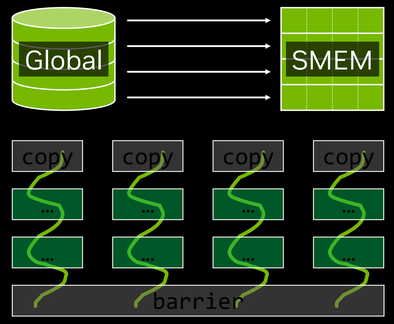
Tile storage?

Memory movement strategy?

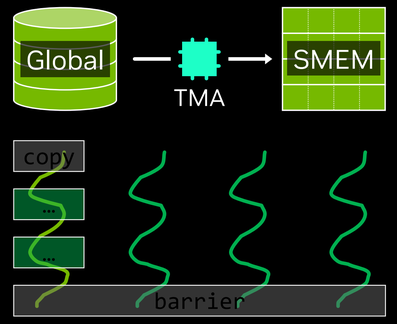
Synchronous Thread Load



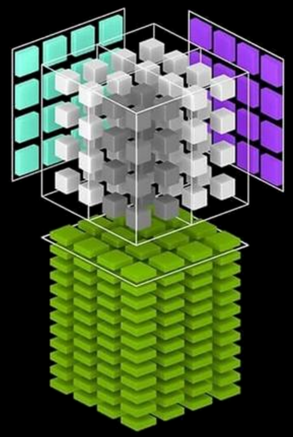
Asynchronous Thread Copy

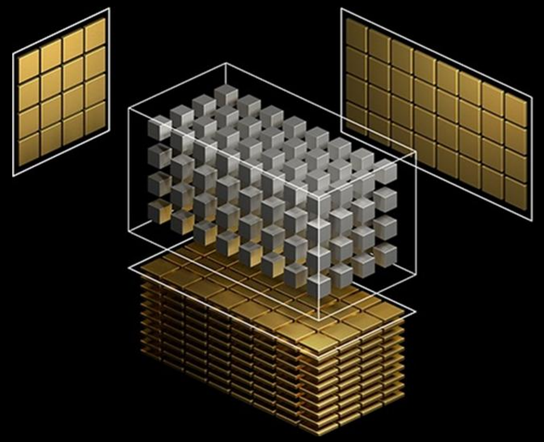


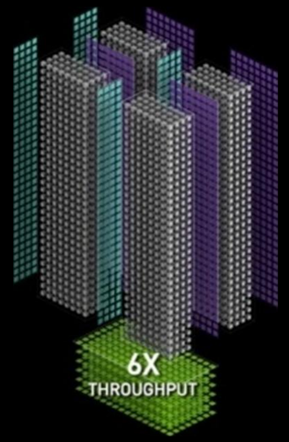
Asynchronous Bulk Copy

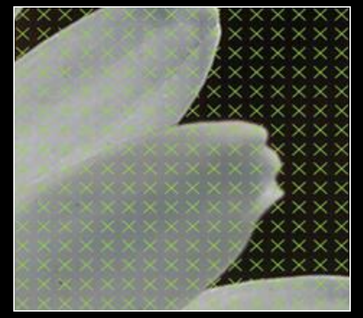


Compute strategy?

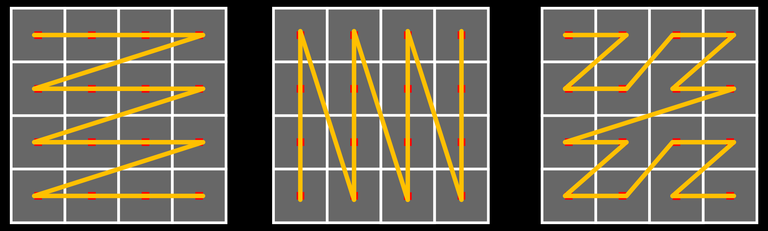








Tile memory layout?

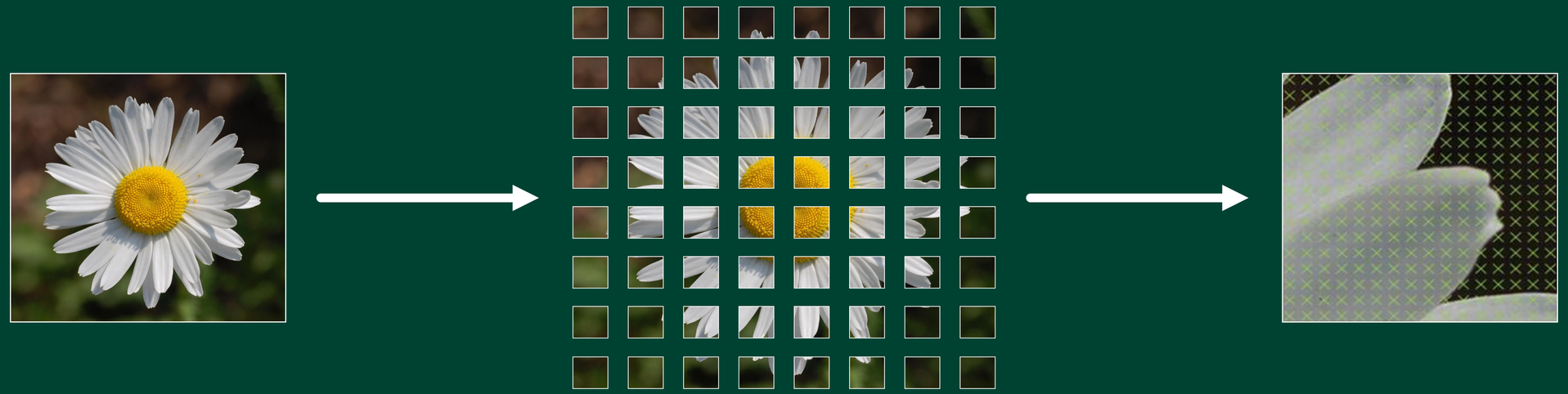


Tile storage?

CUDA SIMT User Responsibility

Split work into blocks
Divide data into tiles

Map blocks & tiles onto threads



Memory movement strategy?

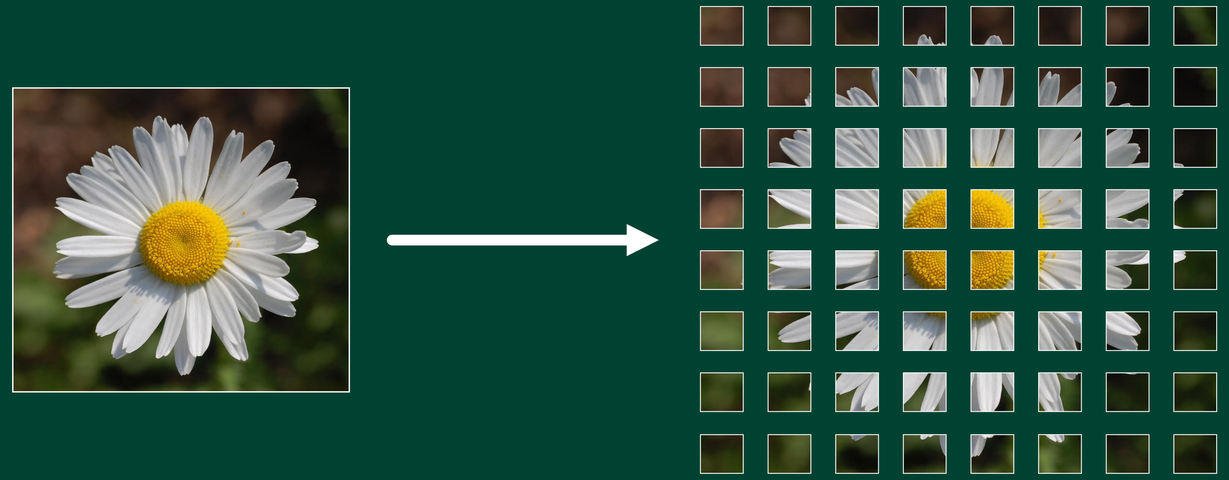
Compute strategy?

Tile memory layout?

Tile storage?

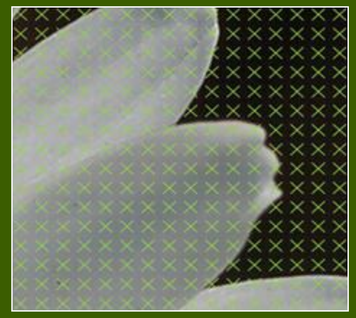
CUDA Tile User Responsibility

Split work into blocks
Divide data into tiles



System Responsibility

Map blocks & tiles onto threads



Memory movement strategy?

Compute strategy?

Tile memory layout?

Tile storage?

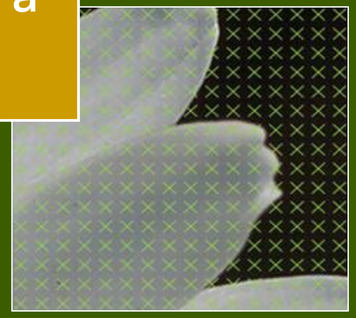
CUDA Tile User Responsibility

Split work into blocks
Divide data into tiles

User guides the system through a pallet of hints

System Responsibility

Map blocks & tiles onto threads



Memory movement strategy?

Compute strategy?

Tile memory layout?

Tile storage?

CUDA Tile is Simpler CUDA

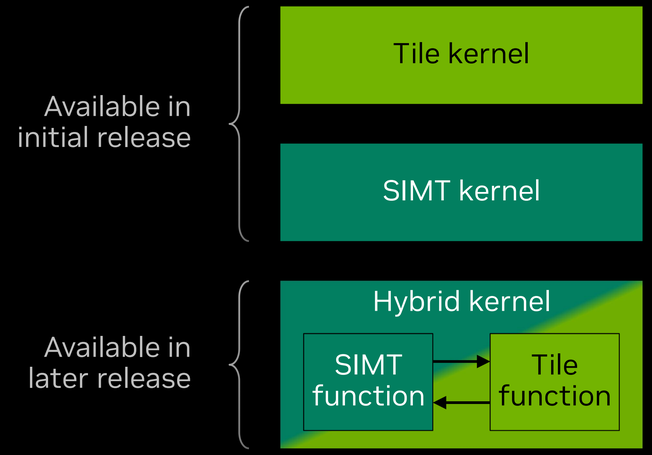
Tile Programming simplifies a wide range of applications

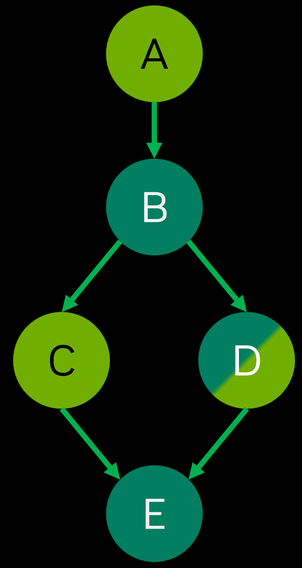
Applies to any program operating on regular arrays or data-parallel problems

Easier to achieve peak performance

Provides a stable, portable way to program tensor cores

Just another part of CUDA – works with everything





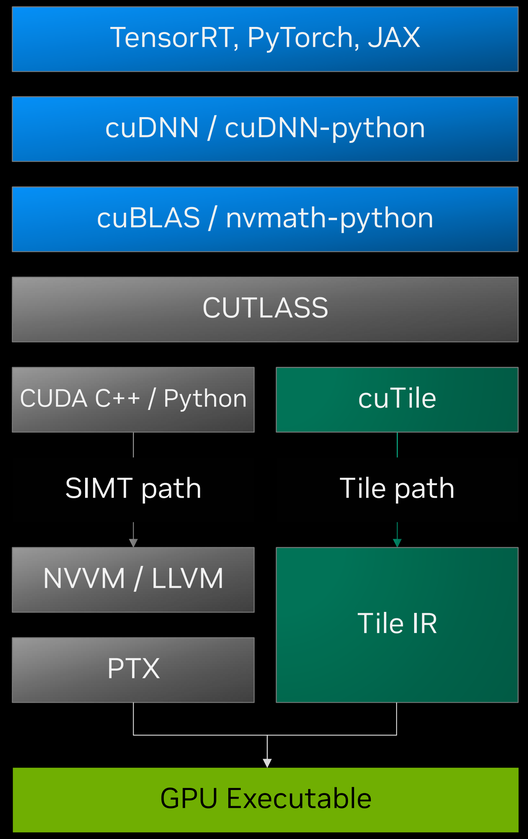
cuTile kernels are CUDA kernels

Use Tile and SIMT kernels together in any workflow

Agenda

- Why evolve the programming model?
  - What are we trying to abstract?
- Introducing CUDA Tile
  - TileIR
  - Programming Model
  - Evolving a GEMM
  - Extending the CUDA Platform
- TileGym
- Performance Analysis
- Next Steps & Questions

CUDA Tile IR: A new virtual ISA for NVIDIA GPUs



Tile IR is a new abstract machine model for GPUs

Can express any program operating on regular arrays or data-parallel problems

Easier to achieve peak performance

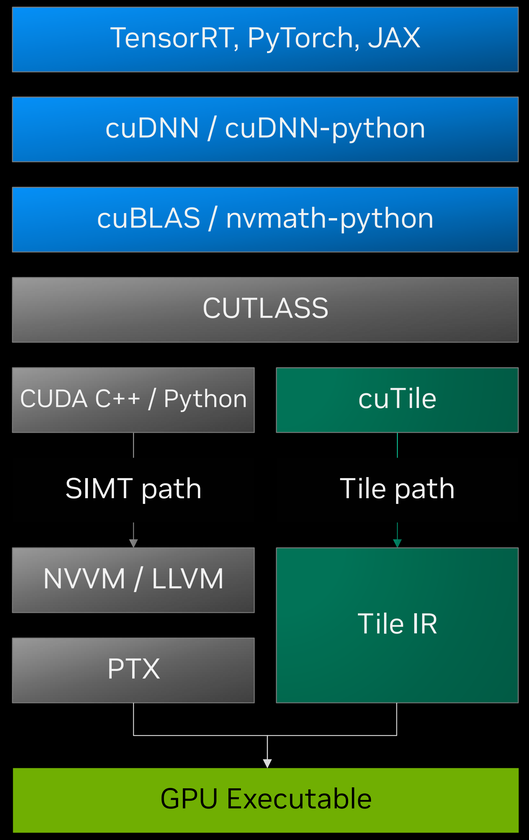
Provided as an open source MLIR dialect making it a natural target for other compilers or DSLs.

Provides a stable, portable way to program tensor cores and other GPU co-processors.

Positioned just as PTX: just another part of CUDA – works with rest of the platform including debugging and profiling.

Includes a comprehensive spec

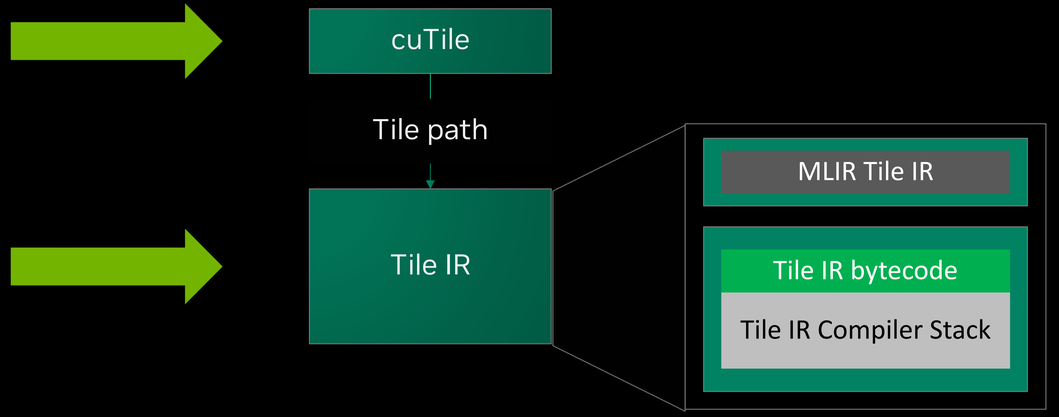
CUDA Tile IR: A new virtual ISA for NVIDIA GPUs



Two Different Compiler & Developer Targets

Python & C++ source-level target for application developers and source-to-source DSL compilers

IR-level target for compilers & DSLs
Includes bindings for Python & C++



Tile IR includes an MLIR dialect for applications emitting MLIR

Applications can emit Tile IR bytecode directly or via bindings in Python & C++

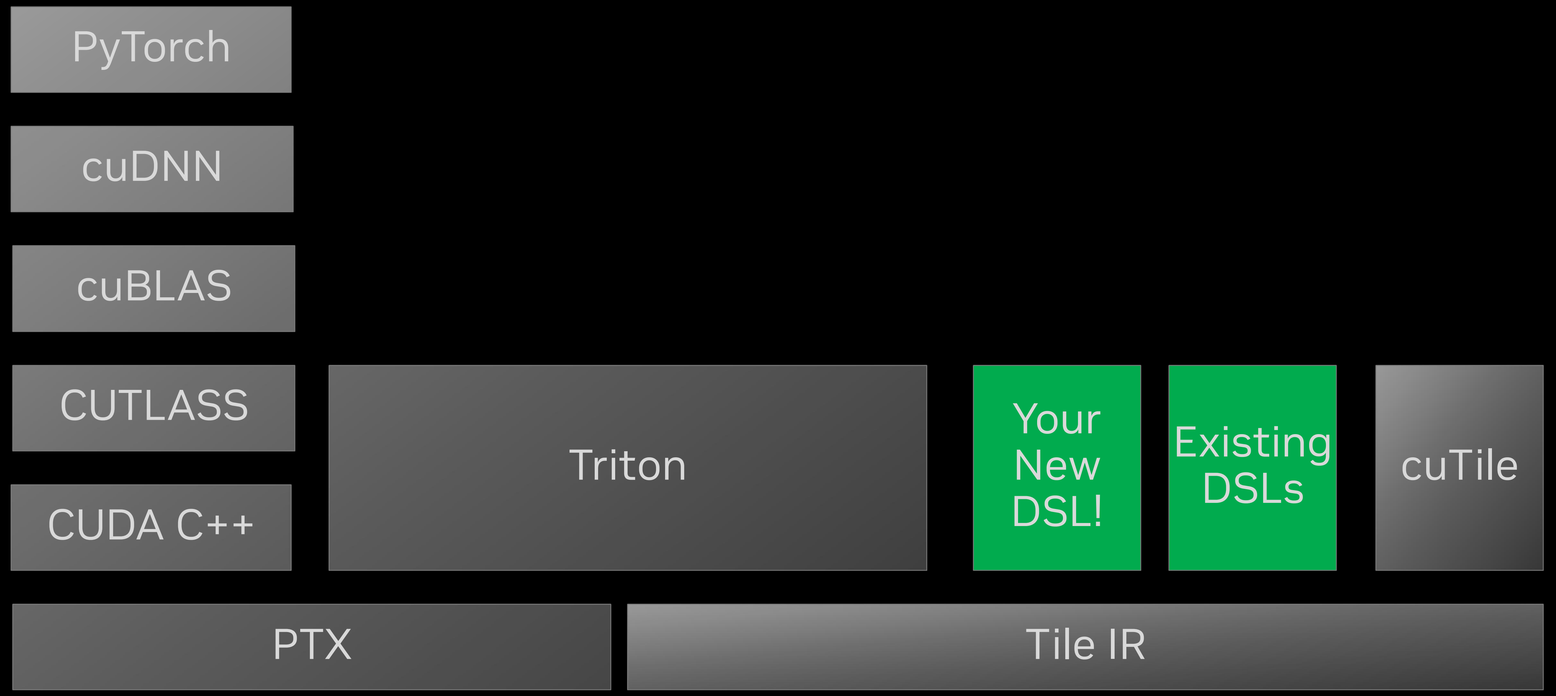
CUDA Tile IR: A new virtual ISA for NVIDIA GPUs

CUDA Tile IR: A new virtual ISA for NVIDIA GPUs

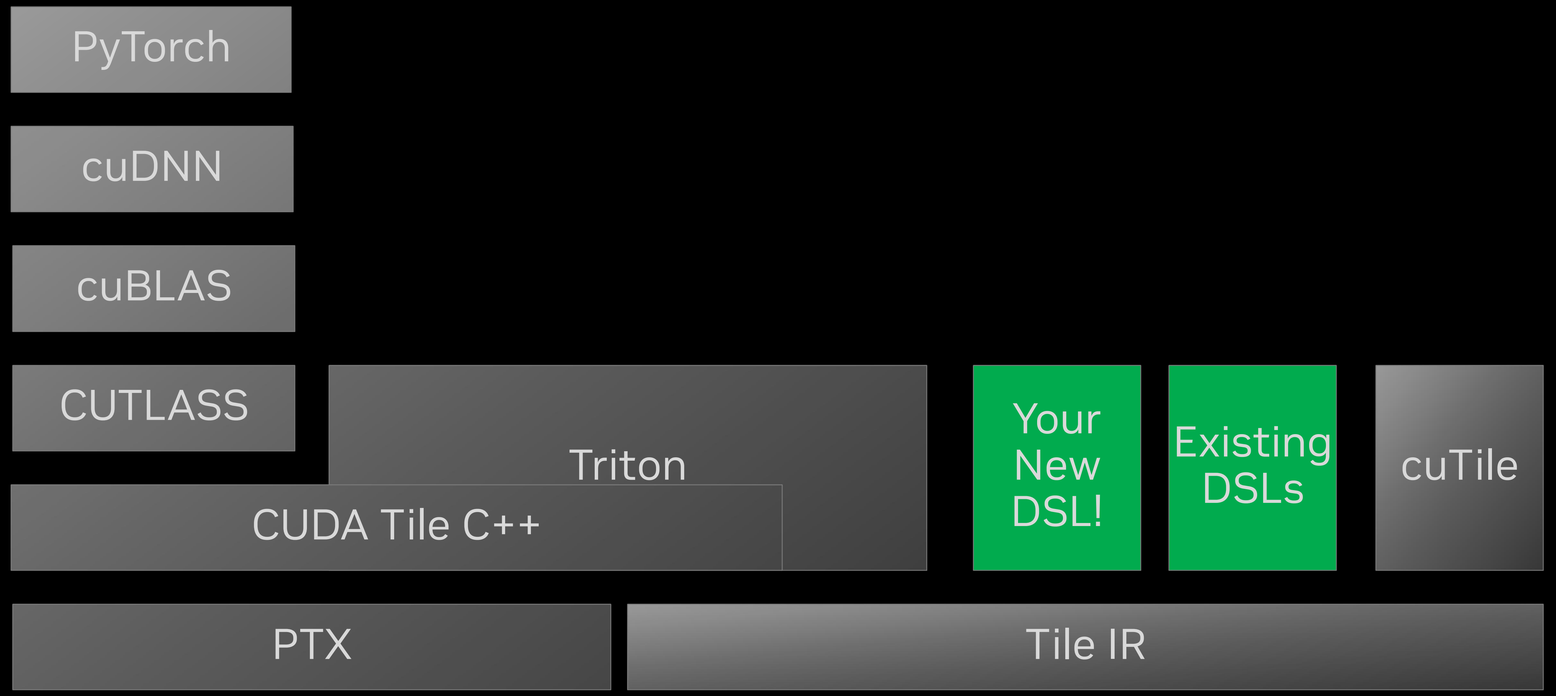
CUDA Tile IR: A new virtual ISA for NVIDIA GPUs

CUDA Tile IR: A new virtual ISA for NVIDIA GPUs

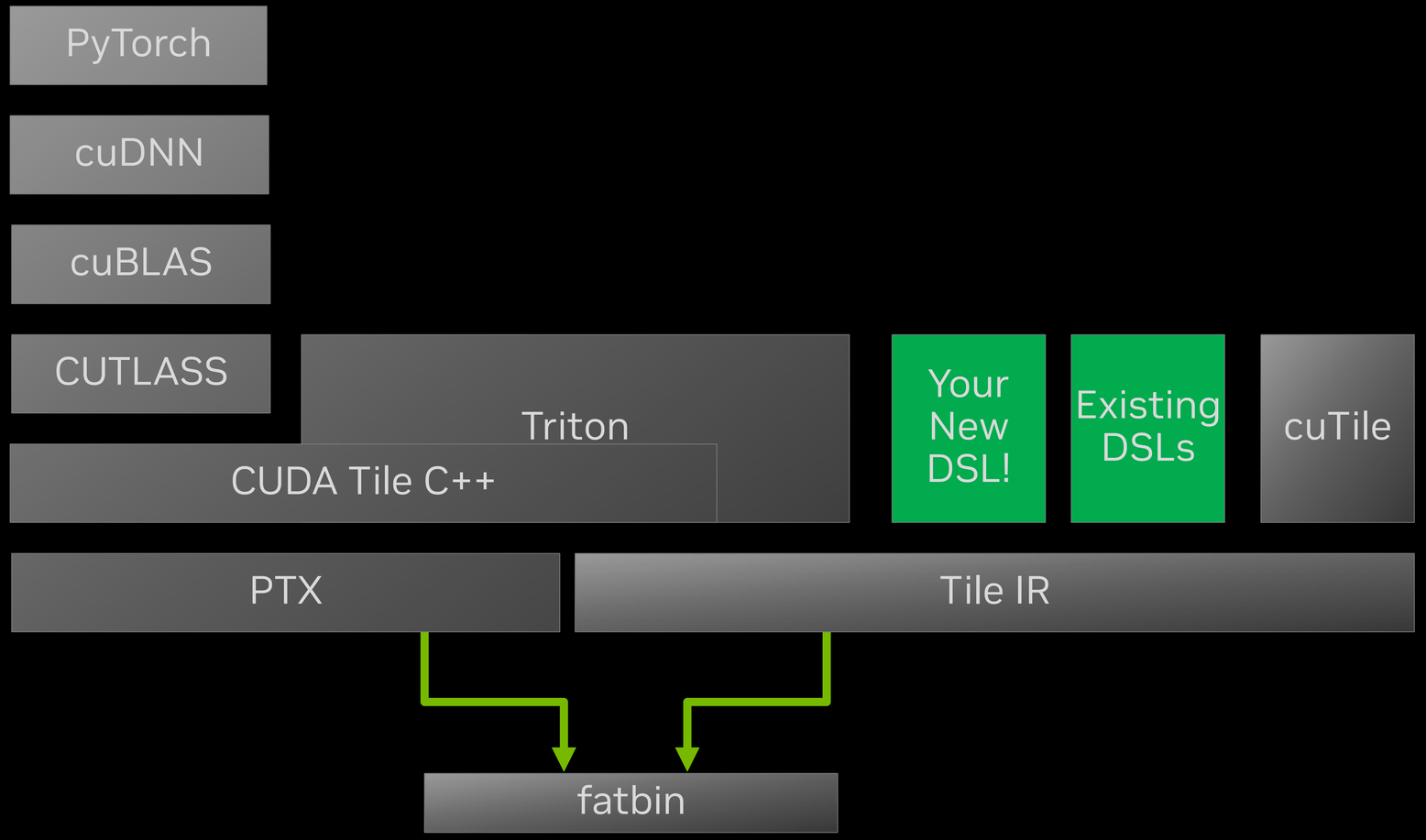
CUDA Tile IR: A new virtual ISA for NVIDIA GPUs



CUDA Tile IR: A new virtual ISA for NVIDIA GPUs



CUDA Tile IR: A new virtual ISA for NVIDIA GPUs



Just like PTX, TileIR bytecode can be embedded in a fatbin.
The CUDA driver can load it and JIT it for the current GPU!

→ You can write a kernel in any source language and embed it in a CUDA application in another language!

Agenda

- Why evolve the programming model?
  - What are we trying to abstract?
- Introducing CUDA Tile
  - TileIR
  - Programming Model
  - Evolving a GEMM
  - Extending the CUDA Platform
- TileGym
- Performance Analysis
- Next Steps & Questions

cuTile: A Python front-end for CUDA Tile

```python
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / den


    ct.store(O, (0, ct.bid(0)), smax)
```

Exposes CUDA Tile as a NumPy-like array programming model.

Naïve CUDA Python SIMT

```
@cuda.device.kernel
def matmul(A, B, C):
  TPB = cuda.blockDim.x
  BPG = cuda.gridDim.x
  x, y = cuda.grid(2)
  tx = cuda.threadIdx.x
  ty = cuda.threadIdx.y

  sA = cuda.shared.array((TPB, TPB), float32)
  sB = cuda.shared.array((TPB, TPB), float32)

  tmp = float32(0.)
  for i in range(BPG):
    sA[ty, tx] = 0
    sB[ty, tx] = 0
    if y < A.shape[0] and (tx+I*TPB) < A.shape[1]:
      sA[ty, tx] = A[y, tx + i * TPB]
    if x < B.shape[1] and (ty+i*TPB) < B.shape[0]:
      sB[ty, tx] = B[ty + i * TPB, x]

    cuda.syncthreads()
    for j in range(TPB):
      tmp += sA[ty, j] * sB[j, tx]
    cuda.syncthreads()

  if y < C.shape[0] and x < C.shape[1]:
    C[y, x] = tmp
```

cuTile Python

```
import cuda.tile as ct

@ct.kernel
def matmul(A: ct.Array,
           B: ct.Array,
           C: ct.Array,
           ts: ct.Constant[ct.Shape]):
  sum = ct.zeros(ts[0:1], A.dtype)
  (x, y) = (ct.bid(0), ct.bid(1))

  pA = ct.view.partition(A, (ts[0], ts[2]))
  pB = ct.view.partition(B, (ts[2], ts[1]))
  for k in range(pA.shape[1]):
    sum = ct.mma(pA[x, k], pB[k, y], sum)

  ct.store(C, (x, y), sum)
```

Speed-Of-Light Non-Portable CUDA C++ SIMT

```
using namespace cute;

template <class SharedStorage,
          class ATensor, class BTensor, class CTensor, class DTensor,
          class MmaTiler_MNK, class TiledMMA, class ClusterShape_MNK,
          class TmaAtomA, class TmaAtomB,
          class Alpha, class Beta>
__global__ static
void
matmul(ATensor mA,
           BTensor mB,
           CTensor mC,
           DTensor mD,
           MmaTiler_MNK mma_tiler,
           TiledMMA tiled_mma,
           ClusterShape_MNK cluster_shape,
           CUTE_GRID_CONSTANT TmaAtomA const tma_atom_A,
           CUTE_GRID_CONSTANT TmaAtomB const tma_atom_B,
           Alpha alpha, Beta beta)
{
  Layout cluster_layout_vmnk = tiled_divide(make_layout(cluster_shape),
    make_tile(typename TiledMMA::AtomThrID{}));

  auto mma_coord_vmnk = make_coord(blockIdx.x % size<0>(cluster_layout_vmnk),
                                   blockIdx.x / size<0>(cluster_layout_vmnk),
                                   blockIdx.y,
                                   _);

  auto mma_coord = select<1,2,3>(mma_coord_vmnk);
  Tensor gA = local_tile(mA, mma_tiler, mma_coord, Step<_1, X,_1>{});
  Tensor gB = local_tile(mB, mma_tiler, mma_coord, Step< X,_1,_1>{});
  Tensor gC = local_tile(mC, mma_tiler, mma_coord, Step<_1,_1, X>{});
  Tensor gD = local_tile(mD, mma_tiler, mma_coord, Step<_1,_1, X>{});
  __syncthreads();

  extern __shared__ char shared_memory[];
  SharedStorage& shared_storage =
    *reinterpret_cast<SharedStorage*>(shared_memory);

  Tensor tCsA = shared_storage.tensor_sA();
  Tensor tCsB = shared_storage.tensor_sB();

  auto mma_v = get<0>(mma_coord_vmnk);
  ThrMMA cta_mma = tiled_mma.get_slice(mma_v);
  Tensor tCgA = cta_mma.partition_A(gA);
  Tensor tCgB = cta_mma.partition_B(gB);
  Tensor tCgC = cta_mma.partition_C(gC);
  Tensor tCgD = cta_mma.partition_C(gD);
  __syncthreads();

  Tensor tCrA = cta_mma.make_fragment_A(tCsA);
  Tensor tCrB = cta_mma.make_fragment_B(tCsB);

  Tensor tCtAcc = cta_mma.make_fragment_C(tCgC);

  uint32_t elect_one_thr  = cute::elect_one_sync();
  uint32_t elect_one_warp = (threadIdx.x / 32 == 0);

  using TmemAllocator = cute::TMEM::Allocator1Sm;
  TmemAllocator tmem_allocator{};

  if (elect_one_warp) {
    tmem_allocator.allocate(TmemAllocator::Sm100TmemCapacityColumns,
      &shared_storage.tmem_base_ptr);
  }

  __syncthreads();
  tCtAcc.data() = shared_storage.tmem_base_ptr;
  __syncthreads();

  auto [tAgA, tAsA] = tma_partition(tma_atom_A, Int<0>{}, Layout<_1>{},
    group_modes<0,3>(tCsA), group_modes<0,3>(tCgA));

  auto [tBgB, tBsB] = tma_partition(tma_atom_B, Int<0>{}, Layout<_1>{},
    group_modes<0,3>(tCsB), group_modes<0,3>(tCgB));

  __syncthreads();

  if (elect_one_warp && elect_one_thr) {
    cute::initialize_barrier(shared_storage.mma_barrier, /* num_ctas */ 1);
    cute::initialize_barrier(shared_storage.tma_barrier, /* num_threads */ 1);
  }

  int mma_barrier_phase_bit = 0;
  int tma_barrier_phase_bit = 0;
  __syncthreads();

  tiled_mma.accumulate_ = UMMA::ScaleOut::Zero;

  for (int k_tile = 0; k_tile < size<3>(tCgA); ++k_tile)
  {
    if (elect_one_warp && elect_one_thr) {
      int tma_transaction_bytes = sizeof(make_tensor_like(tAsA))
                                + sizeof(make_tensor_like(tBsB));
      cute::set_barrier_transaction_bytes(shared_storage.tma_barrier,
                                          tma_transaction_bytes);
      copy(tma_atom_A.with(shared_storage.tma_barrier), tAgA(_,k_tile), tAsA);
      copy(tma_atom_B.with(shared_storage.tma_barrier), tBgB(_,k_tile), tBsB);
    }

    cute::wait_barrier(shared_storage.tma_barrier, tma_barrier_phase_bit);
    tma_barrier_phase_bit ^= 1;

    if (elect_one_warp) {
      for (int k_block = 0; k_block < size<2>(tCrA); ++k_block) {
        gemm(tiled_mma, tCrA(_,_,k_block), tCrB(_,_,k_block), tCtAcc);
        tiled_mma.accumulate_ = UMMA::ScaleOut::One;
      }
      cutlass::arch::umma_arrive(&shared_storage.mma_barrier);
    }
    cute::wait_barrier(shared_storage.mma_barrier, mma_barrier_phase_bit);
    mma_barrier_phase_bit ^= 1;
  }

  TiledCopy tiled_t2r_copy = make_tmem_copy(SM100_TMEM_LOAD_32dp32b1x{}, tCtAcc);
  ThrCopy   thr_t2r_copy   = tiled_t2r_copy.get_slice(threadIdx.x);

  Tensor tDgC = thr_t2r_copy.partition_D(tCgC);
  Tensor tDrC = make_fragment_like(tDgC);
  copy(tDgC, tDrC);

  Tensor tDtAcc = thr_t2r_copy.partition_S(tCtAcc);
  Tensor tDgD   = thr_t2r_copy.partition_D(tCgD);

  using AccType = typename decltype(tCtAcc)::value_type;
  Tensor tDrAcc = make_tensor<AccType>(shape(tDgD));

  copy(tiled_t2r_copy, tDtAcc, tDrAcc);

  axpby(alpha, tDrAcc, beta, tDrC);
  copy(tDrC, tDgD);

  __syncthreads();

  if (elect_one_warp) {
    tmem_allocator.release_allocation_lock();
    tmem_allocator.free(shared_storage.tmem_base_ptr,
      TmemAllocator::Sm100TmemCapacityColumns);
  }
}
```

cuTile Python

```
import cuda.tile as ct

@ct.kernel
def matmul(A: ct.Array,
           B: ct.Array,
           C: ct.Array,
           ts: ct.Constant[ct.Shape]):
  sum = ct.zeros(ts[0:1], A.dtype)
  (x, y) = (ct.bid(0), ct.bid(1))

  pA = ct.view.partition(A, (ts[0], ts[2]))
  pB = ct.view.partition(B, (ts[2], ts[1]))
  for k in range(pA.shape[1]):
    sum = ct.mma(pA[x, k], pB[k, y], sum)

  ct.store(C, (x, y), sum)
```

cuTile By Example

```python
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / den


    ct.store(O, (0, ct.bid(0)), smax)
```

cuTile is Uniform & Array-based

```python
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / den


    ct.store(O, (0, ct.bid(0)), smax)
```

Global arrays are mutable and accessible by all logical blocks.

cuTile is Uniform & Array-based

```python
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))

    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / den

    ct.store(O, (0, ct.bid(0)), smax)
```

Global arrays are mutable and accessible by all logical blocks.

cuTile is Uniform & Array-based

```python
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))

    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / den

    ct.store(O, (0, ct.bid(0)), smax)
```

Global arrays are mutable and accessible by all logical blocks.

Globally Mutable Arrays & Locally Immutable Tiles

```
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))

    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / den

    ct.store(O, (0, ct.bid(0)), smax)
```

Tile arrays are **immutable** and local to a logical block.

Global arrays are mutable and accessible by all logical blocks.

Globally Mutable Arrays & Locally Immutable Tiles

```
import cuda.tile as ct


@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))

    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    max = num / den

    ct.store(O, (0, ct.bid(0)), max)
```

Tile arrays are **immutable** and local to a logical block.

Global arrays are mutable and accessible by all logical blocks.

Uniform Launch APIs

```
import cuda.tile as ct


# Device
@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / de


    ct.store(O, (0, ct.bid(0)), smax)


# Host
I = cuda.to_device(np.random.rand(512, 128))
O = cuda.device_array_like(I)
ct.launch(ct.Config(grid=64), softmax, I, O, 8)
result = O.copy_to_host()
```

Tile kernels are launched with the same underlying APIs as SIMT kernels.

Uniform Launch APIs

```
import cuda.tile as ct


# Device
@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / de


    ct.store(O, (0, ct.bid(0)), smax)


# Host
I = cuda.to_device(np.random.rand(512, 128))
O = cuda.device_array_like(I)
ct.launch(ct.Config(grid=64), softmax, I, O, 8)
result = O.copy_to_host()
```

The device side code is self contained, and free of complexity like TMA descriptors.

Uniform Launch APIs

```
import cuda.tile as ct


# Device
@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / de


    ct.store(O, (0, ct.bid(0)), smax)


# Host
I = cuda.to_device(np.random.rand(512, 128))
O = cuda.device_array_like(I)
ct.launch(ct.Config(grid=64), softmax, I, O, 8)
result = O.copy_to_host()
```

Host side only copies arrays and launches the kernel.

Uniform Launch APIs

```
import cuda.tile as ct


# Device
@ct.func
def softmax(I, O, r):
    c = ct.load(I, (0, ct.bid(0)), (I.shape[0], r))


    max = ct.max(c, axis=0, keepdims=True)
    num = ct.exp(c - max)
    den = ct.sum(num, axis=0, keepdims=True)
    smax = num / de


    ct.store(O, (0, ct.bid(0)), smax)


# Host
I = cuda.to_device(np.random.rand(512, 128))
O = cuda.device_array_like(I)
ct.launch(ct.Config(grid=64), softmax, I, O, 8)
result = O.copy_to_host()
```

Any CUDA-Array-interface-compliant **array** can be passed as global array kernel parameter.

Interoperate With Other CUDA Code

```
@cuda.jit
def vector_add(x, y, z, tile_shape):
 pid = cuda.grid(1)
 for i in range(tile_shape / cuda.blockDim.x):
   item = pid + i * cuda.blockDim.x
   z[item] = x[item] + y[item]


@cuda.tile.func
def vector_add(x, y, z, ts):
 tx = cuda.tile.load(x, cuda.tile.bid(0), ts)
 ty = cuda.tile.load(y, cuda.tile.bid(0), ts)
 tz = tx + ty
 cuda.tile.store(z, cuda.tile.bid(0), ts)
```

You can write and use CUDA Python SIMT and Tile kernels in the same source file and launch on the same stream.

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Tile IR Memory Model

- Tile IR has a formally defined memory consistency model.
- For any pair of memory operations their order is undefined, in Tile IR you can use tokens to order any pair of instructions with respect to each other.
- In the examples above the loads and stores do not alias so ordering is not required.
- The frontends handle token passing today as we will see in Python, C++, and Rust.
- See the specification for more details.

Optimizing GEMM: CTA Rasterization

```python
@ct.kernel(num_ctas=ct.ByTarget(sm_100=2))
def matmul_kernel(
    A, B, C,
    TILE_SIZE_M: ConstInt, TILE_SIZE_N: ConstInt, TILE_SIZE_K: ConstInt,
):
    GROUP_SIZE_M = 8
    M = A.shape[0]
    N = B.shape[1]
    bidx, bidy = swizzle_2d(M, N, TILE_SIZE_M, TILE_SIZE_N, GROUP_SIZE_M)

     num_tiles_k = ct.num_tiles(A, axis=1, shape=(TILE_SIZE_M, TILE_SIZE_K))

    accumulator = ct.full((TILE_SIZE_M, TILE_SIZE_N), 0, dtype=ct.float32)
    zero_pad = ct.PaddingMode.ZERO
    dtype = ct.tfloat32 if A.dtype == ct.float32 else A.dtype


    for k in range(num_tiles_k):
        a = ct.load(A, index=(bidx, k), shape=(TILE_SIZE_M, TILE_SIZE_K), padding_mode=zero_pad).astype(dtype)
        b = ct.load(B, index=(k, bidy), shape=(TILE_SIZE_K, TILE_SIZE_N), padding_mode=zero_pad).astype(dtype)
        accumulator = ct.mma(a, b, accumulator)


    accumulator = ct.astype(accumulator, C.dtype)

    ct.store(C, index=(bidx, bidy), tile=accumulator)
```

https://github.com/NVIDIA/TileGym/blob/main/src/tilegym/ops/cutile/matmul.py#L37

Optimizing GEMM: CTA Rasterization

```python
@ct.kernel(num_ctas=ct.ByTarget(sm_100=2))
def matmul_kernel(
    A, B, C,
    TILE_SIZE_M: ConstInt, TILE_SIZE_N: ConstInt, TILE_SIZE_K: ConstInt,
):
    GROUP_SIZE_M = 8
    M = A.shape[0]
    N = B.shape[1]
    bidx, bidy = swizzle_2d(M, N, TILE_SIZE_M, TILE_SIZE_N, GROUP_SIZE_M)

     num_tiles_k = ct.num_tiles(A, axis=1, shape=(TILE_SIZE_M, TILE_SIZE_K))

    accumulator = ct.full((TILE_SIZE_M, TILE_SIZE_N), 0, dtype=ct.float32)
    zero_pad = ct.PaddingMode.ZERO
    dtype = ct.tfloat32 if A.dtype == ct.float32 else A.dtype


    for k in range(num_tiles_k):
        a = ct.load(A, index=(bidx, k), shape=(TILE_SIZE_M, TILE_SIZE_K), padding_mode=zero_pad).astype(dtype)
        b = ct.load(B, index=(k, bidy), shape=(TILE_SIZE_K, TILE_SIZE_N), padding_mode=zero_pad).astype(dtype)
        accumulator = ct.mma(a, b, accumulator)


    accumulator = ct.astype(accumulator, C.dtype)

    ct.store(C, index=(bidx, bidy), tile=accumulator)
```

Optimizing GEMM: CTA Rasterization

```
def swizzle_2d(M, N, TILE_SIZE_M, TILE_SIZE_N, GROUP_SIZE_M):
    # Get the global IDs of the current CUDA block (CTA) in a 1D grid.
    bid = ct.bid(0)
    num_bid_m = ct.cdiv(M, TILE_SIZE_M)
    num_bid_n = ct.cdiv(N, TILE_SIZE_N)
    num_bid_in_group = GROUP_SIZE_M * num_bid_n
    group_id = bid // num_bid_in_group
    first_bid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_bid_m - first_bid_m, GROUP_SIZE_M)
    bid_m = first_bid_m + (bid % group_size_m)
    bid_n = (bid % num_bid_in_group) // group_size_m
    return bid_m, bid_n
```

This improves performance simply by changing grid coordinates to better exploit locality.

Optimizing GEMM: Static Persistence

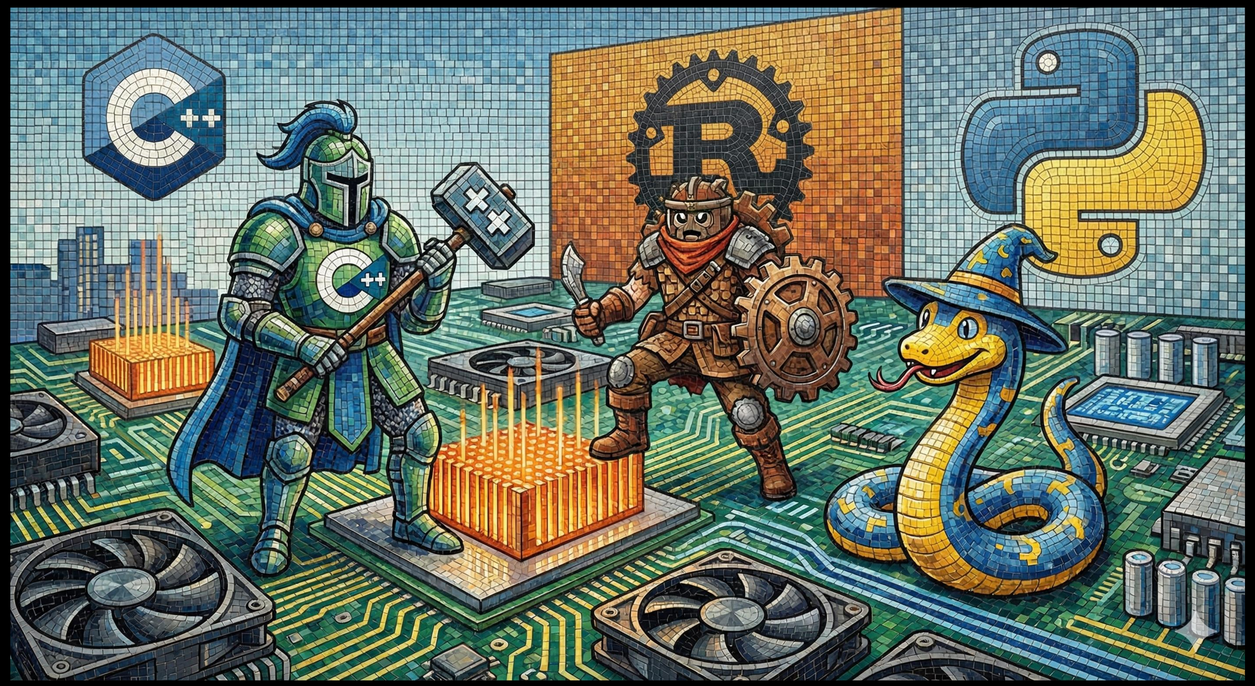
```python
@ct.kernel(num_ctas=2)
def static_persistent_matmul_kernel(
    A, B, C, M: int, N: int, K: int,
    TILE_SIZE_M: ct.Constant[int], TILE_SIZE_N: ct.Constant[int], TILE_SIZE_K: ct.Constant[int],
    GROUP_SIZE_M: ct.Constant[int],
):
    start_bid = ct.bid(0)

    num_bid_m = ct.cdiv(M, TILE_SIZE_M)
    num_bid_n = ct.cdiv(N, TILE_SIZE_N)
    k_tiles = ct.cdiv(K, TILE_SIZE_K)
    num_tiles = num_bid_m * num_bid_n
    num_programs = ct.num_blocks(0)

    # Static persistent scheduling loop
    for tile_id in range(start_bid, num_tiles, num_programs):
        num_bid_in_group = GROUP_SIZE_M * num_bid_n
        bid_m, bid_n = _compute_bid(tile_id, num_bid_in_group, num_bid_m, GROUP_SIZE_M)
        accumulator = ct.full((TILE_SIZE_M, TILE_SIZE_N), 0.0, dtype=ct.float32)

        for k_tile in range(k_tiles):
                a = ct.load(A, index=(bid_m, k_tile), shape=(TILE_SIZE_M, TILE_SIZE_K))
                b = ct.load(B, index=(k_tile, bid_n), shape=(TILE_SIZE_K, TILE_SIZE_N))

            dtype = ct.tfloat32 if A.dtype == ct.float32 else A.dtype
            a = ct.astype(a, dtype)
            b = ct.astype(b, dtype)

            accumulator = ct.mma(a, b, acc=accumulator)

        result = ct.astype(accumulator, C.dtype)
        ct.store(C, index=(bid_m, bid_n), tile=result)
```

https://github.com/NVIDIA/TileGym/blob/main/src/tilegym/ops/cutile/matmul.py#L120

Optimizing GEMM: Static Persistence

```python
@ct.kernel(num_ctas=2)
def static_persistent_matmul_kernel(
    A, B, C, M: int, N: int, K: int,
    TILE_SIZE_M: ct.Constant[int], TILE_SIZE_N: ct.Constant[int], TILE_SIZE_K: ct.Constant[int],
    GROUP_SIZE_M: ct.Constant[int],
):
    start_bid = ct.bid(0)

    num_bid_m = ct.cdiv(M, TILE_SIZE_M)
    num_bid_n = ct.cdiv(N, TILE_SIZE_N)
    k_tiles = ct.cdiv(K, TILE_SIZE_K)
    num_tiles = num_bid_m * num_bid_n
    num_programs = ct.num_blocks(0)

    # Static persistent scheduling loop
    for tile_id in range(start_bid, num_tiles, num_programs):
        num_bid_in_group = GROUP_SIZE_M * num_bid_n
        bid_m, bid_n = _compute_bid(tile_id, num_bid_in_group, num_bid_m, GROUP_SIZE_M)
        accumulator = ct.full((TILE_SIZE_M, TILE_SIZE_N), 0.0, dtype=ct.float32)

        …
```

cuTile X



- cuTile is a family of language exposures of the CUDA Tile programming model.
- cuTile Python was released as part of CUDA 13. 1.
- But wait there is more!
  - Tile programming to C++ with more information to follow at GTC.
  - We will be open sourcing an experimental safe, tile-based programming model in Rust as well early next year.
- We expect more languages to be able to easily adopt Tile for native kernel authoring / GPU offload.

cuTile C++

```
__tile_global__ void kernel(float* lhs, float* rhs, std::size_t length, float* out) {
    auto lhsTensor = ct::tensor_span{lhs, ct::shape{8_ic, length}};
    auto rhsTensor = ct::tensor_span{rhs, ct::shape{length, 16_ic}};


    auto lhsView = ct::partition_view{lhsTensor, ct::shape{4_ic, 8_ic}};
    auto rhsView = ct::partition_view{rhsTensor, ct::shape{8_ic, 4_ic}};


    auto [xBlock, yBlock, dummy] = ct::bid();


    auto accTile = ct::full<ct::tile<float, ct::shape<4, 4>>>(0);


    for (auto idx : ct::irange(0, int(length / 8))) {
        auto lhsTile = lhsView.load(xBlock, idx);
        auto rhsTile = rhsView.load(idx, yBlock);


        accTile = ct::mma(lhsTile, rhsTile, accTile);
    }


    auto outTensor = ct::tensor_span{out, ct::shape{8_ic, 16_ic}};
    auto outView = ct::partition_view{outTensor, ct::shape{4_ic, 4_ic}};


    outView.store(accTile, xBlock, yBlock);
}
```

cuTile C++

```
__tile_global__ void kernel(float* lhs, float* rhs, std::size_t length, float* out) {
    auto lhsTensor = ct::tensor_span{lhs, ct::shape{8_ic, length}};
    auto rhsTensor = ct::tensor_span{rhs, ct::shape{length, 16_ic}};


    auto lhsView = ct::partition_view{lhsTensor, ct::shape{4_ic, 8_ic}};
    auto rhsView = ct::partition_view{rhsTensor, ct::shape{8_ic, 4_ic}};


    auto [xBlock, yBlock, dummy] = ct::bid();


    auto accTile = ct::full<ct::tile<float, ct::shape<4, 4>>>(0);


    for (auto idx : ct::irange(0, int(length / 8))) {
        auto lhsTile = lhsView.load(xBlock, idx);
        auto rhsTile = rhsView.load(idx, yBlock);


        accTile = ct::mma(lhsTile, rhsTile, accTile);
    }


    auto outTensor = ct::tensor_span{out, ct::shape{8_ic, 16_ic}};
    auto outView = ct::partition_view{outTensor, ct::shape{4_ic, 4_ic}};


    outView.store(accTile, xBlock, yBlock);
}
```

Similar level of abstraction to Python, kernel structure is nearly identical.

cuTile C++

```
__tile_global__ void kernel(float* lhs, float* rhs, std::size_t length, float* out) {
    auto lhsTensor = ct::tensor_span{lhs, ct::shape{8_ic, length}};
    auto rhsTensor = ct::tensor_span{rhs, ct::shape{length, 16_ic}};


    auto lhsView = ct::partition_view{lhsTensor, ct::shape{4_ic, 8_ic}};
    auto rhsView = ct::partition_view{rhsTensor, ct::shape{8_ic, 4_ic}};

    auto [xBlock, yBlock, dummy] = ct::bid();


    auto accTile = ct::full<ct::tile<float, ct::shape<4, 4>>>(0);


    for (auto idx : ct::irange(0, int(length / 8))) {
        auto lhsTile = lhsView.load(xBlock, idx);
        auto rhsTile = rhsView.load(idx, yBlock);


        accTile = ct::mma(lhsTile, rhsTile, accTile);
    }

    auto outTensor = ct::tensor_span{out, ct::shape{8_ic, 16_ic}};
    auto outView = ct::partition_view{outTensor, ct::shape{4_ic, 4_ic}};


    outView.store(accTile, xBlock, yBlock);
}
```

The “raw” API that NVCC provide is limited to scalar/pointer types, so users must create views themselves for now.

cuTile C++

```
__tile_global__ void kernel(float* lhs, float* rhs, std::size_t length, float* out) {
    auto lhsTensor = ct::tensor_span{lhs, ct::shape{8_ic, length}};
    auto rhsTensor = ct::tensor_span{rhs, ct::shape{length, 16_ic}};


    auto lhsView = ct::partition_view{lhsTensor, ct::shape{4_ic, 8_ic}};
    auto rhsView = ct::partition_view{rhsTensor, ct::shape{8_ic, 4_ic}};

    auto [xBlock, yBlock, dummy] = ct::bid();


    auto accTile = ct::full<ct::tile<float, ct::shape<4, 4>>>(0);


    for (auto idx : ct::irange(0, int(length / 8))) {
        auto lhsTile = lhsView.load(xBlock, idx);
        auto rhsTile = rhsView.load(idx, yBlock);


        accTile = ct::mma(lhsTile, rhsTile, accTile);
    }

    auto outTensor = ct::tensor_span{out, ct::shape{8_ic, 16_ic}};
    auto outView = ct::partition_view{outTensor, ct::shape{4_ic, 4_ic}};

    outView.store(accTile, xBlock, yBlock);
}
```

The core algorithm is otherwise unchanged.

cuTile Rust

```
#[tile_rust::entry]
fn gemm<E: ElementType, const BM: i32,
        const BN: i32, const BK: i32, const K: i32
>(z: &mut Tensor<E, { [BM, BN] }>,
  x: &Tensor<E, { [-1, K] }>,
  y: &Tensor<E, { [K, -1] }>,
) {
    let part_x = x.partition(const_shape![BM, BK]);
    let part_y = y.partition(const_shape![BK, BN]);
    let pid: (i32, i32, i32) = get_tile_block_id();
    let mut tile_z = load_tile(z);
    for i in 0i32..(K / BK) {
        let tile_x = part_x.load([pid.0, i]);
        let tile_y = part_y.load([i, pid.1]);
        tile_z = mma(tile_x, tile_y, tile_z);
    }
    z.store(tile_z);
}
```

We also have an experimental support in Rust. One big difference is memory safe partitioning API. Notice the store here is to a slice of the output tensor.

Exporting CUDA Tile IR

```
import cuda.tile as ct
from cuda.tile._compiler_options import CompilerOptions
from cuda.tile._compile import compile_tile


@ct.kernel
def vector_add(a, b, c, tile_size: ct.Constant[int]):
    # Get the 1D pid
    pid = ct.bid(0)

    ...


...
# Compile kernel to TileIR bytecode and cubin binary file
binary = compile_tile(vector_add._pyfunc, (a, b, c, tile_size), CompilerOptions())
# Copy TileIR and cubin file to current directory
shutil.copy2(binary.fname_cubin, ”kernel.cubin")
with open("kernel.bytecode", "wb") as f:
      f.write(binary.bytecode)
```

TileIR bytecode can be exported and included in fatbin files, for loading and launching in the driver from any language, compatible with any future GPU!

Agenda

- Why evolve the programming model?
  - What are we trying to abstract?
- Introducing CUDA Tile
  - TileIR
  - Programming Model
  - Evolving a GEMM
  - Extending the CUDA Platform
- TileGym
- Performance Analysis
- Next Steps & Questions

TileGym

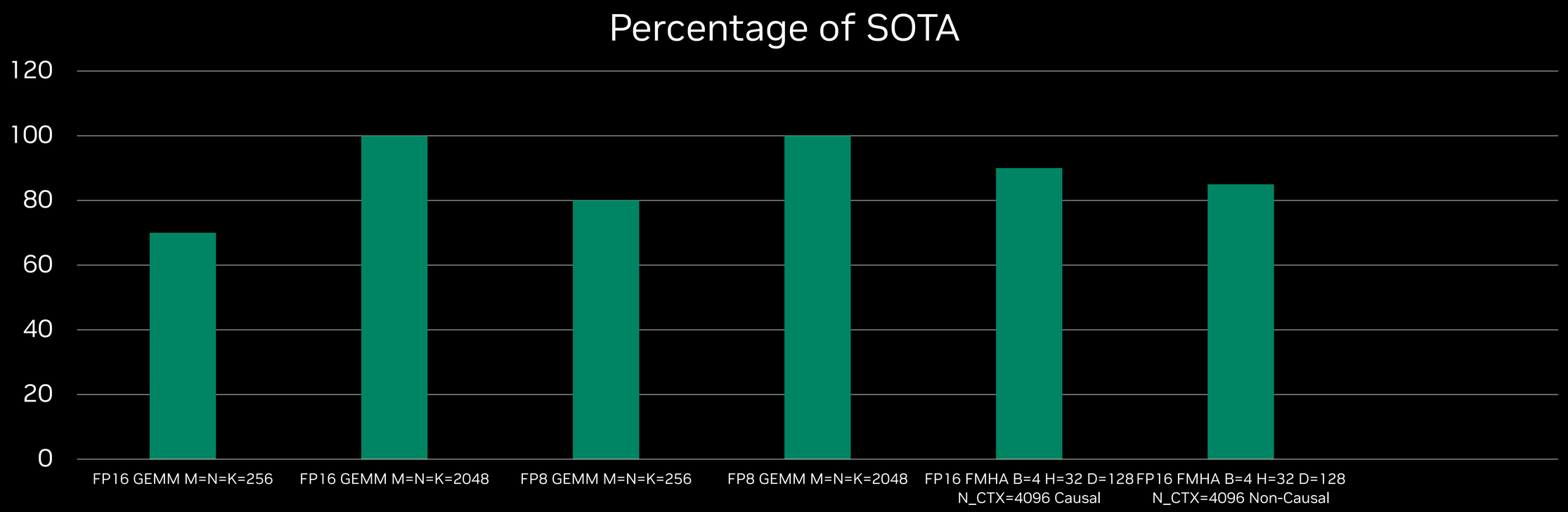
a collection of kernel tutorials and examples for tile-based GPU programming.

- Check it out today on GitHub (NVIDIA/TileGym)
- The home of cuTile examples and integrations.
- Contains an initial auto-tuner for cuTile kernels, with an official API to land in a future release.

Autotuning

```
@autotune(search_space=_matmul_autotune_configs())
def cutile_autotune_matmul(a, b, c, autotuner: Autotuner | None = None):
    M, N = c.shape
    tuned_result = autotuner(
        torch.cuda.current_stream(),
        grid_fn=lambda named_args, cfg: (
            ceil(M / cfg.TILE_SIZE_M) * ceil(N / cfg.TILE_SIZE_N),
            1,
            1,
        ),
        kernel=matmul_kernel,
        args_fn=lambda cfg: (a, b, c, cfg.TILE_SIZE_M, cfg.TILE_SIZE_N, cfg.TILE_SIZE_K),
    )
    return c
```

A taste of performance on B200



Performance dependent on the kernel implementation as well as the Tile IR compiler.

Roadmap

CUDA Tile IR

Blackwell support with other families to follow.

JIT & offline compilation

Supports standard data types down to FP8 today.

CUDA forward and minor version compatibility

Detailed semi-formal specification

Public MLIR Dialect

Linux & Windows support

cuTile

cuTile-Python as a pip installable wheel

API is NumPy-like with support for many operations

Supports dlpack, and integration with existing frameworks

Open-Source on GitHub

...

Roadmap

Ongoing performance improvements

Expanded GPU Family Support (Ampere and above)

cuTile C++ for tile programing in CUDA C++

Function-level interoperability for calling existing code

Tile-based developer tooling

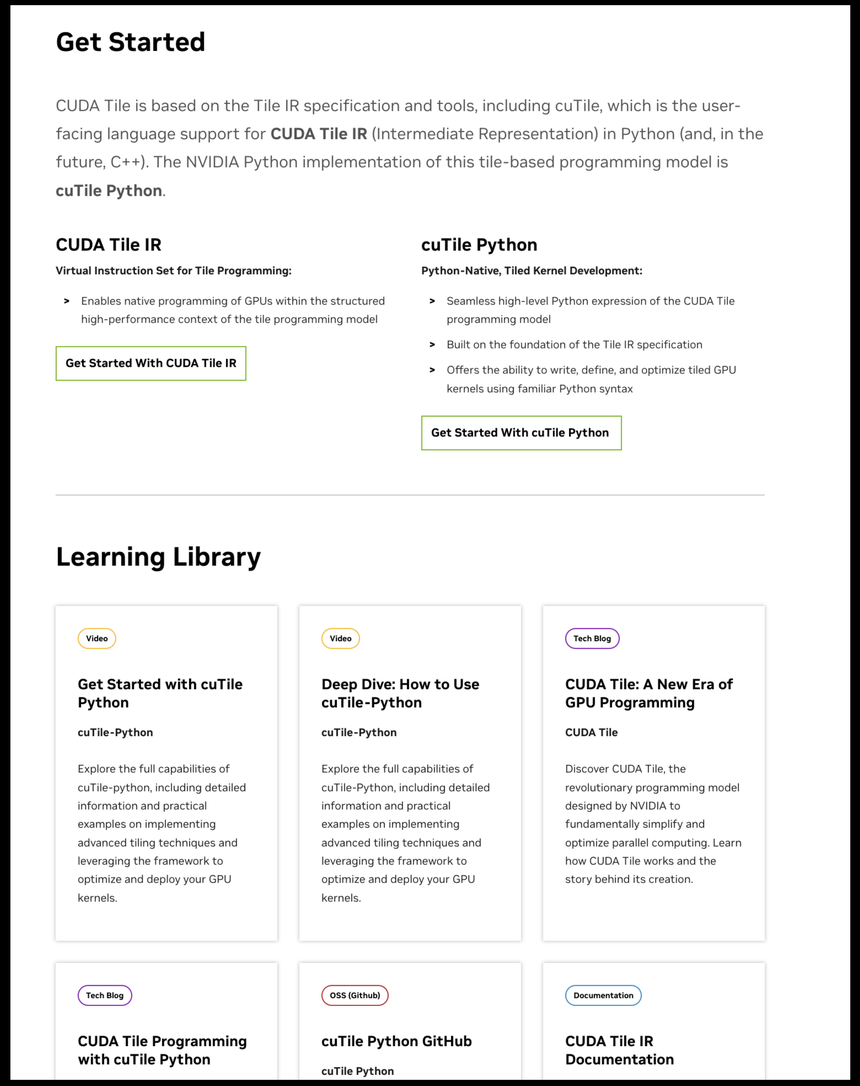
FP4 and more mixed precision support

Arbitrary-sized tiles

CUDA 13.1
initial public release

13.x
new features in every CUDA release

CUDA Tile: Next Steps



Get Started

CUDA Tile is based on the Tile IR specification and tools, including cuTile, which is the user-facing language support for **CUDA Tile IR** (Intermediate Representation) in Python (and, in the future, C++). The NVIDIA Python implementation of this tile-based programming model is **cuTile Python**.

CUDA Tile IR

Virtual Instruction Set for Tile Programming:

- Enables native programming of GPUs within the structured high-performance context of the tile programming model

Get Started With CUDA Tile IR

cuTile Python

Python-Native, Tiled Kernel Development:

- Seamless high-level Python expression of the CUDA Tile programming model
- Built on the foundation of the Tile IR specification
- Offers the ability to write, define, and optimize tiled GPU kernels using familiar Python syntax

Get Started With cuTile Python

Learning Library

Video

Get Started with cuTile Python

cuTile-Python

Explore the full capabilities of cuTile-python, including detailed information and practical examples on implementing advanced tiling techniques and leveraging the framework to optimize and deploy your GPU kernels.

Video

Deep Dive: How to Use cuTile-Python

cuTile-Python

Explore the full capabilities of cuTile-Python, including detailed information and practical examples on implementing advanced tiling techniques and leveraging the framework to optimize and deploy your GPU kernels.

Tech Blog

CUDA Tile: A New Era of GPU Programming

CUDA Tile

Discover CUDA Tile, the revolutionary programming model designed by NVIDIA to fundamentally simplify and optimize parallel computing. Learn how CUDA Tile works and the story behind its creation.

Tech Blog

CUDA Tile Programming with cuTile Python

OSS (Github)

cuTile Python GitHub

cuTile Python

Documentation

CUDA Tile IR Documentation

CUDA Tile is live and ready to use.

Can Learn more about CUDA Tile on our developer hub, including docs, specification, and pointers to code.

Install cuTile directly from PyPI: pip install cuda-tile

Or read the code on GitHub

Use the MLIR dialect now on GitHub (NVIDIA/cuda-tile)

Includes a bytecode disassembler!

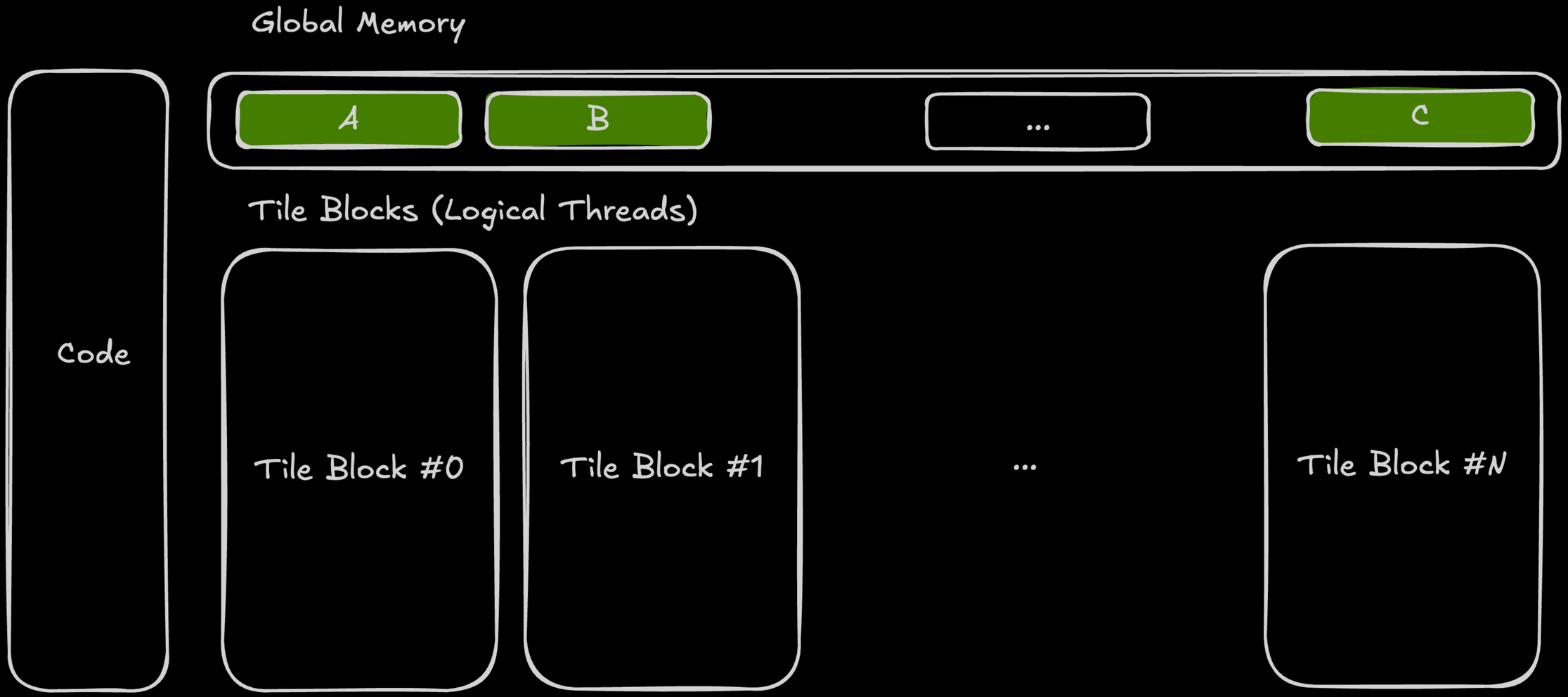
Report bugs or feedback on GitHub!


NVIDIA
Thank You!
Questions?

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

Tile IR Abstract Machine

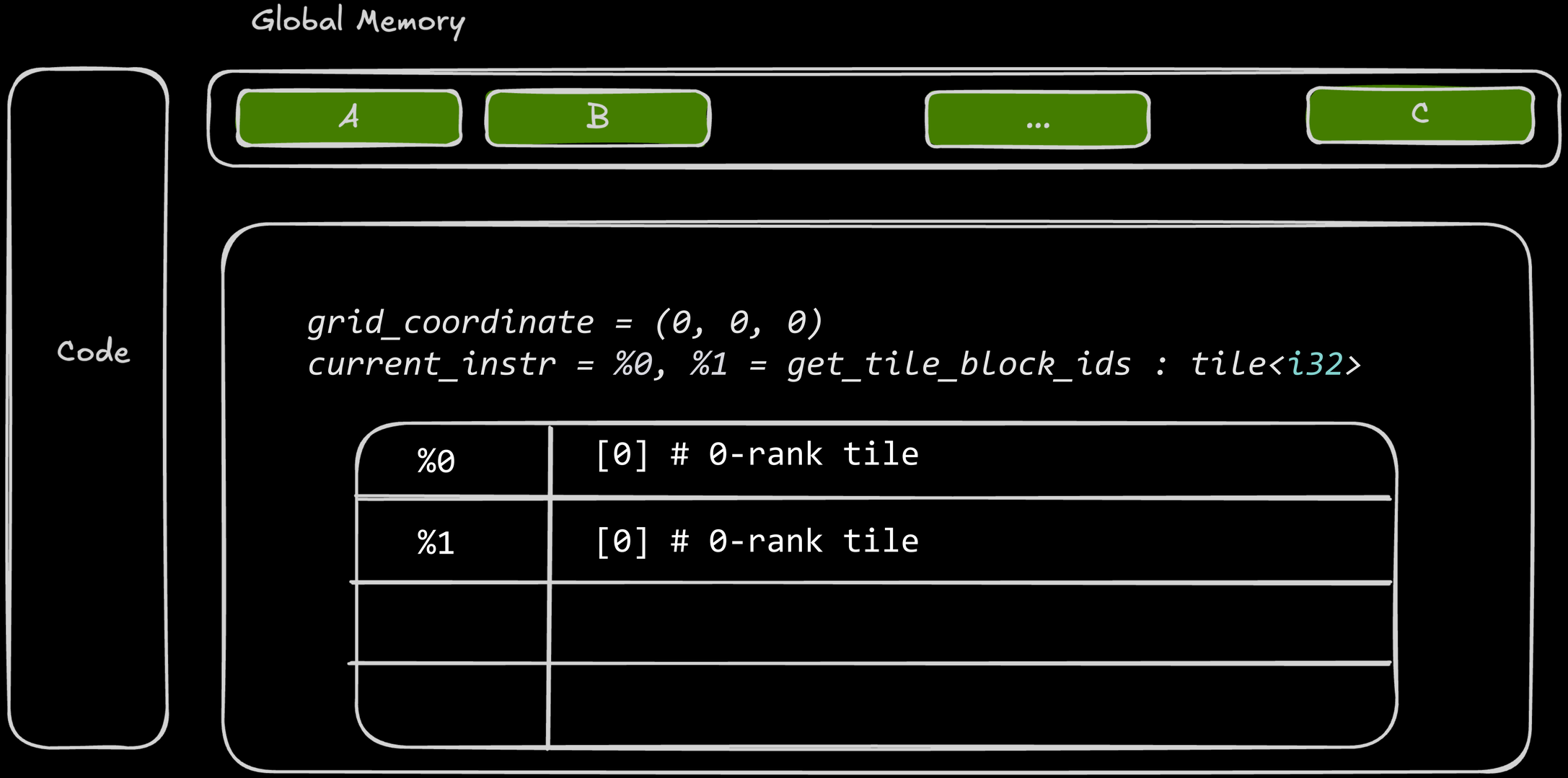


```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                          : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                              : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                  : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

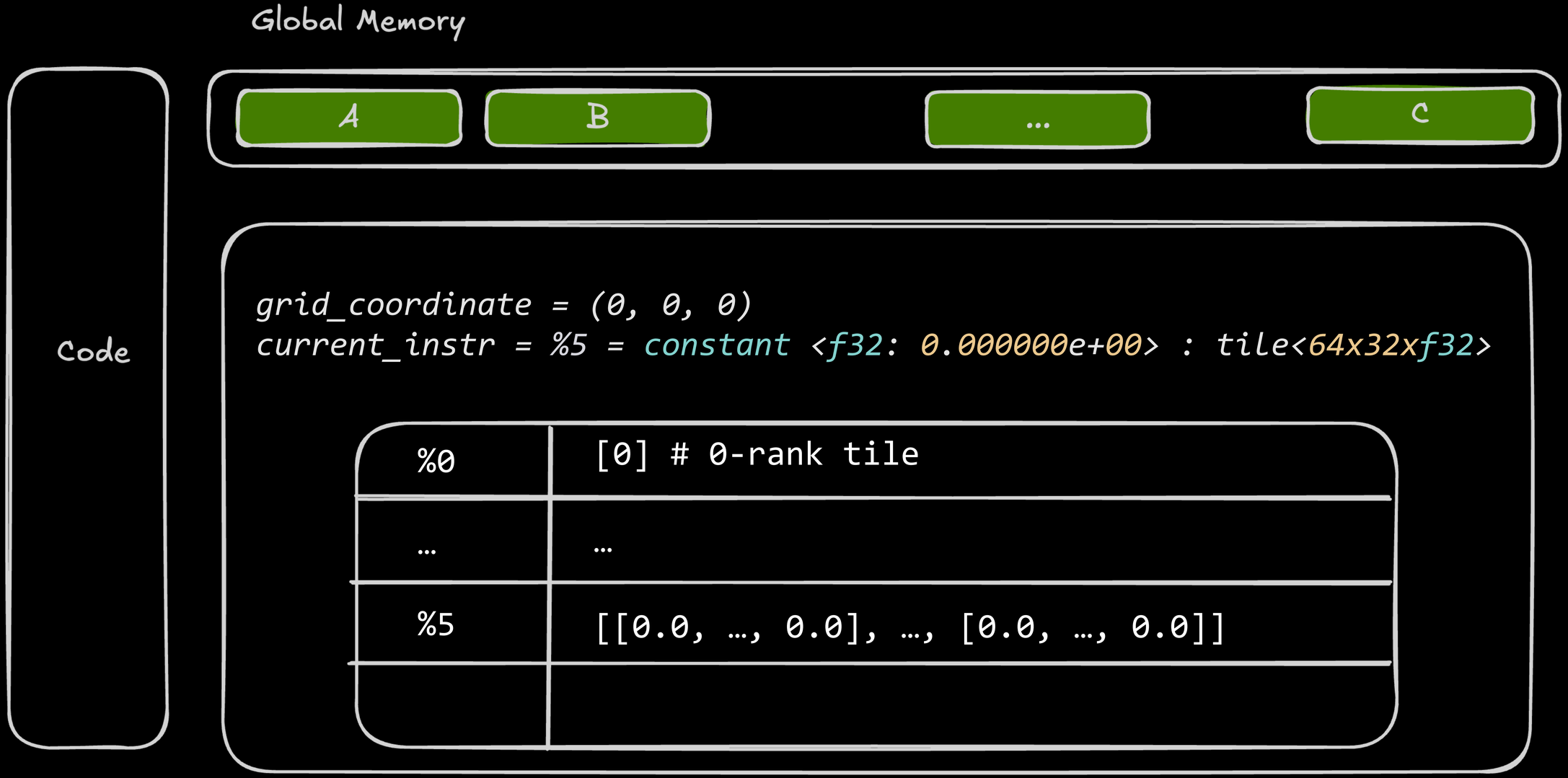
```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                         : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                 : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```

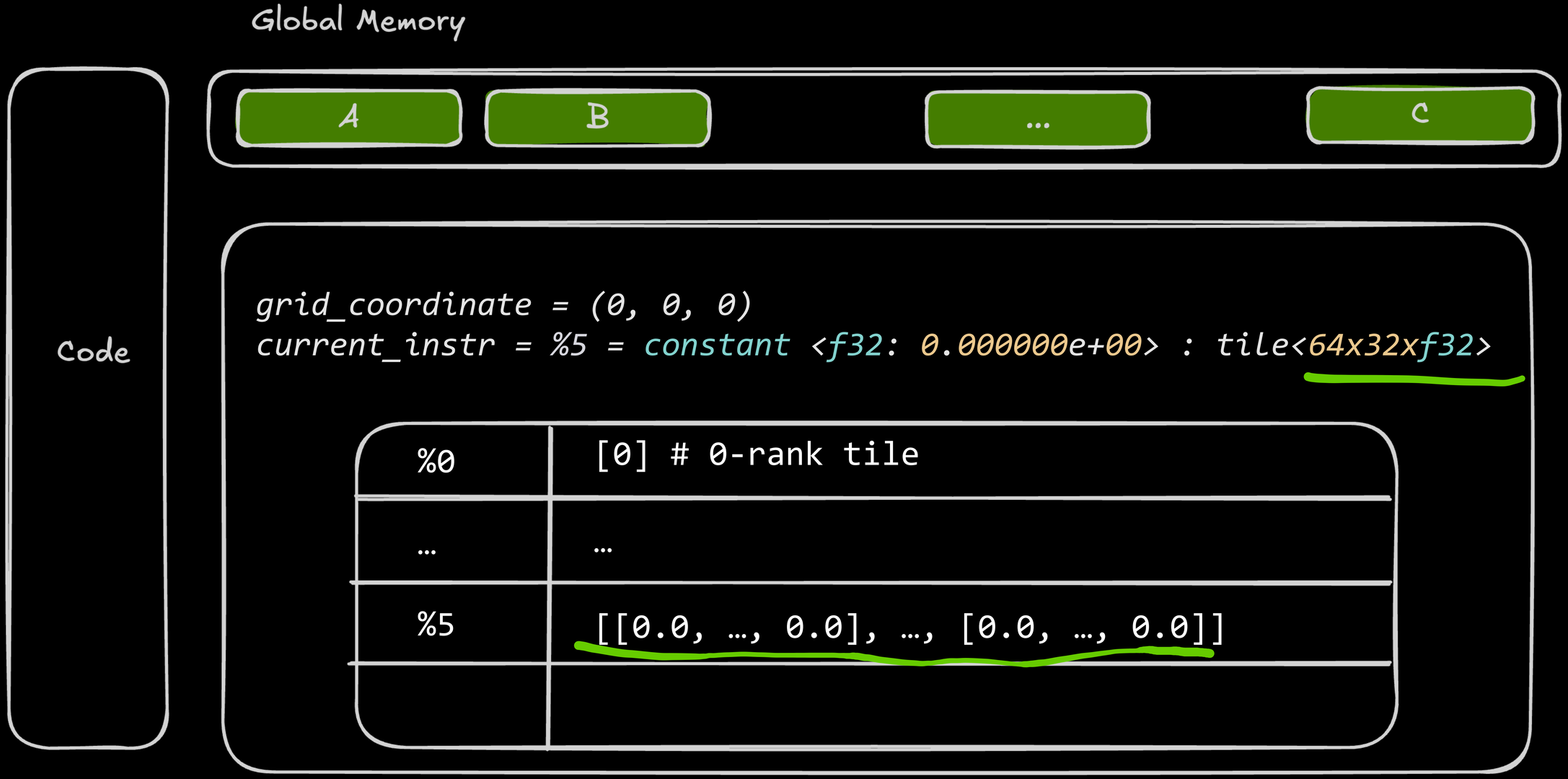

Global Memory
A
B
...
C
Code
grid_coordinate = (0, 0, 0)
current_instr = %0, %1 = get_tile_block_ids : tile<i32>
%0
[0] # 0-rank tile
%1
[0] # 0-rank tile

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                         : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                 : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```


Global Memory
A
B
...
C
Code
grid_coordinate = (0, 0, 0)
current_instr = %5 = constant <f32: 0.000000e+00> : tile<64x32xf32>
%0
[0] # 0-rank tile
…
…
%5
[[0.0, …, 0.0], …, [0.0, …, 0.0]]



The value in %5 is a 64 x 32 tile of f32 values.

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                         : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                 : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                         : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                 : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```

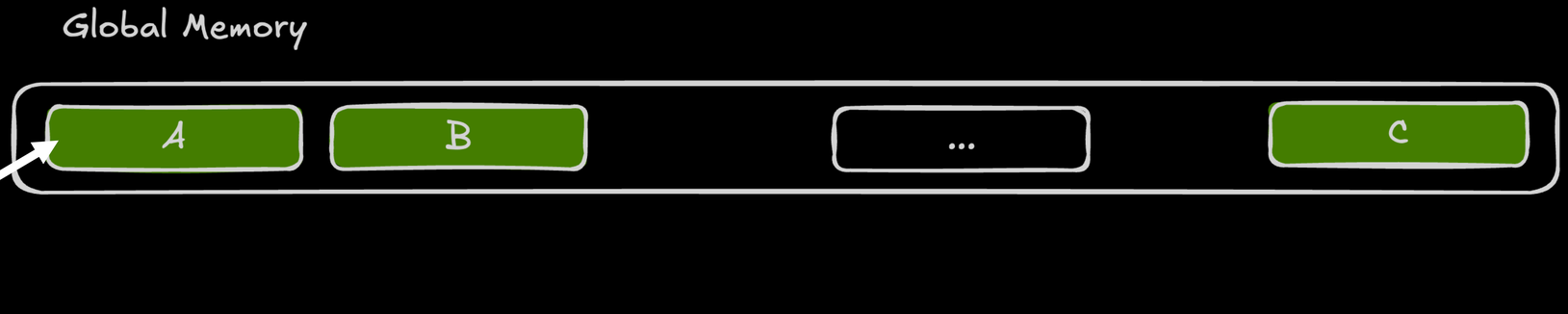
Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                         : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                 : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```

The following sequence realizes `ct.load`:

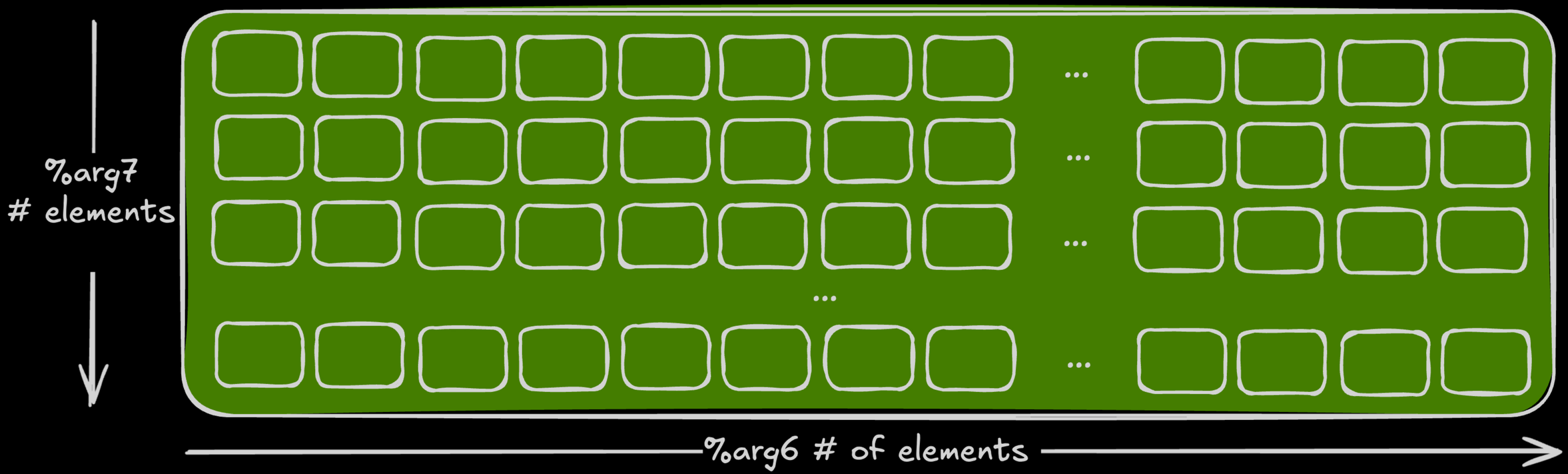
```
%tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
%pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
%load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                        : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
```



%arg0 is the base pointer pointing to A in global memory

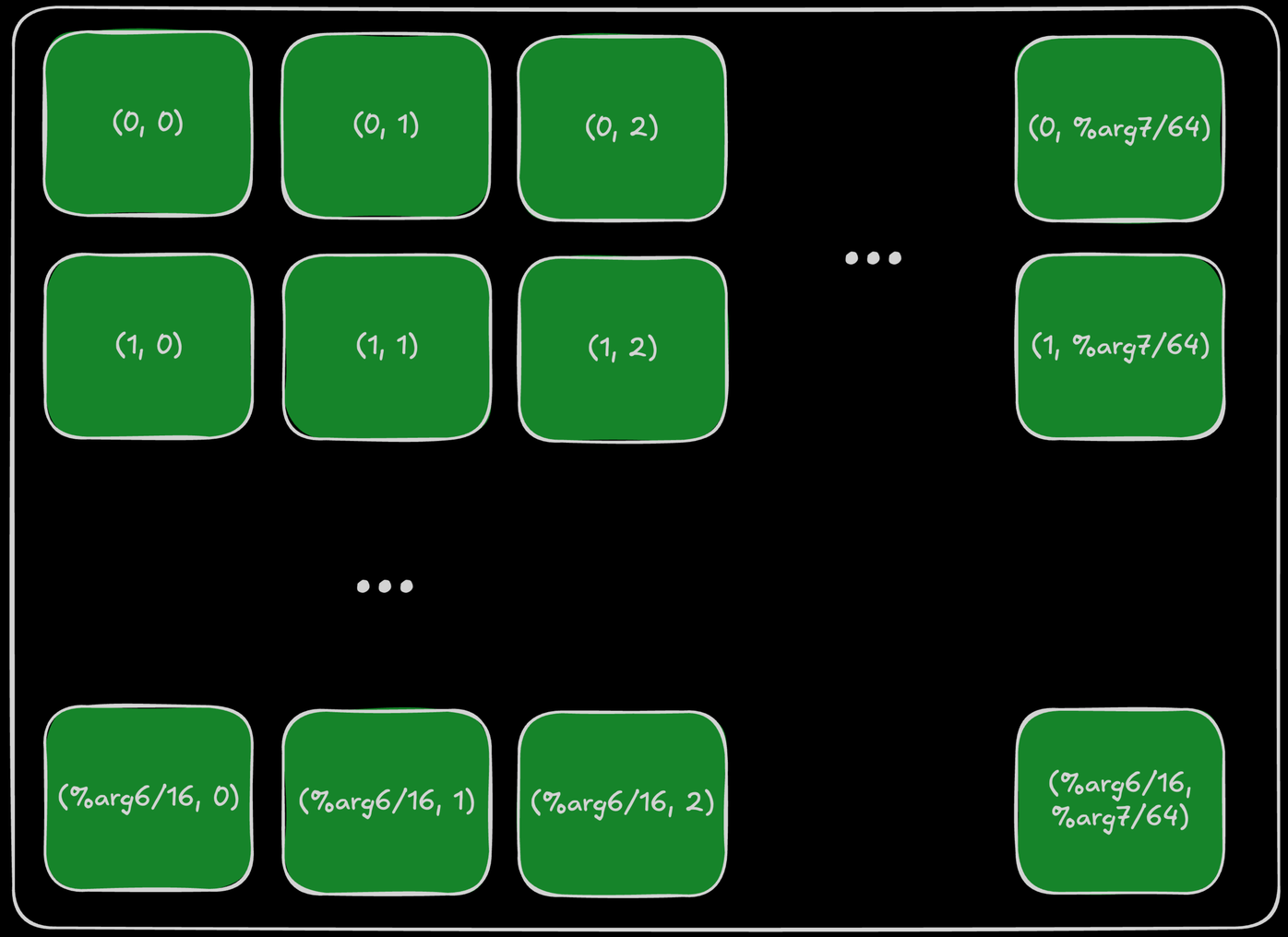
```
%tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1]
                : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
```

Register %tview_2 will first contain the tensor_view which specifies the global memory layout composed of its shape + strides.



```
%pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
```

Each tile will be 64x16



Register %pview_3 will contain a partition_view or a type of sub-view which allows us to logically index global tensors as a series of tiles.

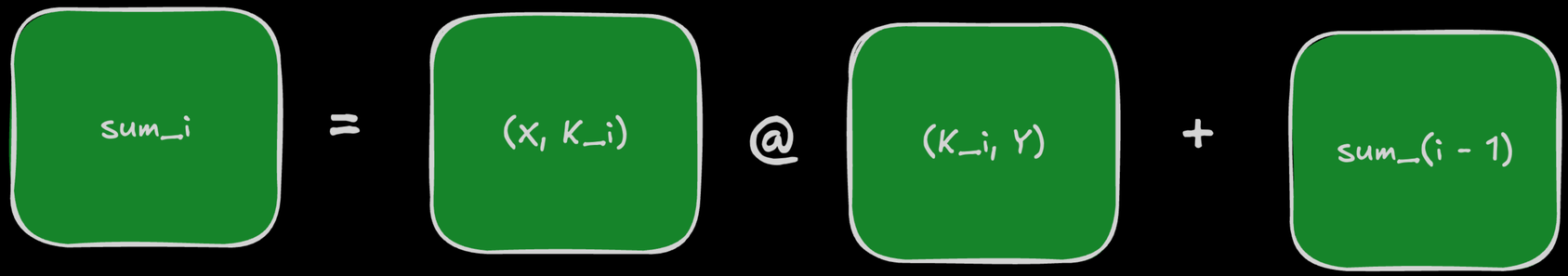
Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                          : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                  : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```

```
%load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
    : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
```

Once we have a `partition_view` we can load a specific tile in this case we load (X, K) tile from the grid, each tile block is computing the (X, Y) tile of the output matrix where X, Y are its grid coordinates.



Once we have loaded from both inputs we can use the mmaf operation which computes MMA over the tiles.

Matrix Multiplication on Tile IR

```python
import cuda.tile as ct

@ct.func
def matmul(A: ct.Array, B: ct.Array, C: ct.Array, ts: ct.Constant[tuple[int]]):
    x_bid, y_bid = ct.bid(0), ct.bid(1)
    sum = ct.zeros(shape=ts, dtype=A.dtype)

    num_K_tiles = ct.num_tiles(A, index=1, shape=(ts[0], ts[2]))
    for k in range(num_K_tiles):
        a = ct.load(A, index=(x_bid, k), shape=(ts[0], ts[2]), order='F')
        b = ct.load(B, index=(k, y_bid), shape=(ts[2], ts[1]), order='C')
        sum = ct.mma(a, b, sum)


    ct.store(C, index=(x_bid, y_bid), tile=sum)
```

```
entry @kernel_matmul(
    %arg0: tile<ptr<f32>>, %arg1: tile<ptr<f32>>, %arg2: tile<ptr<f32>>, %arg3: tile<i64>,
    %arg4: tile<i64>, %arg5: tile<i64>, %arg6: tile<i64>, %arg7: tile<i64>,
    %arg8: tile<i64>, %arg9: tile<i64>, %arg10: tile<i64>, %arg11: tile<i64>) {
  %blockId_x, %blockId_y, %blockId_z = get_tile_block_id : tile<i32>
  %exti = exti %blockId_y unsigned : tile<i32> -> tile<i64>
  %exti_0 = exti %blockId_x unsigned : tile<i32> -> tile<i64>
  %itof = itof %arg7 signed  : tile<i64> -> tile<f64>
  %itof = itof %arg5 signed  : tile<i64> -> tile<f64>
  %divf = divf %itof, %itof_1  : tile<f64>
  %cst_0_f32 = constant <f32: 0.000000e+00> : tile<64x32xf32>
  %ftoi = ftoi %divf signed  : tile<f64> -> tile<i64>
  %cst_0_i64 = constant <i64: 0> : tile<i64>
  %cst_1_i64 = constant <i64: 1> : tile<i64>
  %for = for %loopIdx in (%cst_0_i64 to %ftoi, step %cst_1_i64) : tile<i64> iter_values(%iterArg0 = %cst_0_f32) -> (tile<64x32xf32>) {
    %tview_2 = make_tensor_view %arg0, shape = [%arg6, %arg7], strides = [%arg7, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_3 = make_partition_view %tview_2 : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>
    %load, %load_tok = load_view_tko weak %pview_3[%exti_0, %loopIdx]
                         : partition_view<tile=(64x16), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<64x16xf32>, token
    %tview_4 = make_tensor_view %arg1, shape = [%arg8, %arg9], strides = [%arg9, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
    %pview_5 = make_partition_view %tview_4 : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>
    %load_6, %load_tok_7 = load_view_tko weak %pview_5[%loopIdx, %exti]
                             : partition_view<tile=(16x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> tile<16x32xf32>, token
    %mmaf = mmaf %load, %load_6, %iterArg0 : tile<64x16xf32>, tile<16x32xf32>, tile<64x32xf32>
    continue %mmaf : tile<64x32xf32>
  }
  %tview = make_tensor_view %arg2, shape = [%arg10, %arg11], strides = [%arg11, 1] : tile<i64> -> tensor_view<?x?xf32, strides=[?,1]>
  %pview = make_partition_view %tview : partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>
  %store_tok = store_view_tko weak %for, %pview [%exti_0, %exti]
                 : tile<64x32xf32>, partition_view<tile=(64x32), tensor_view<?x?xf32, strides=[?,1]>>, tile<i64> -> token
  return
}
```