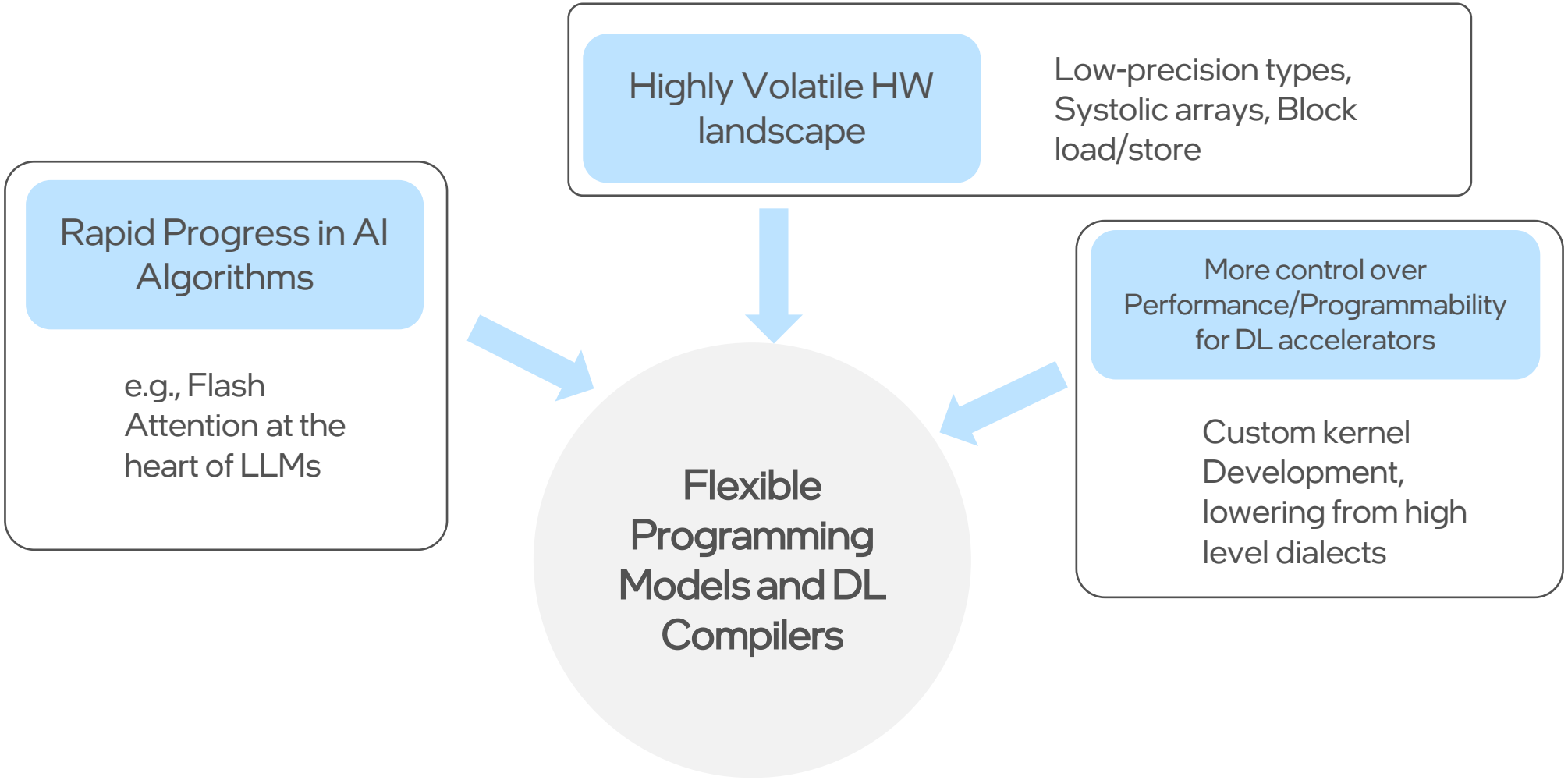
Extending MLIR Dialects for Deep Learning Compilers

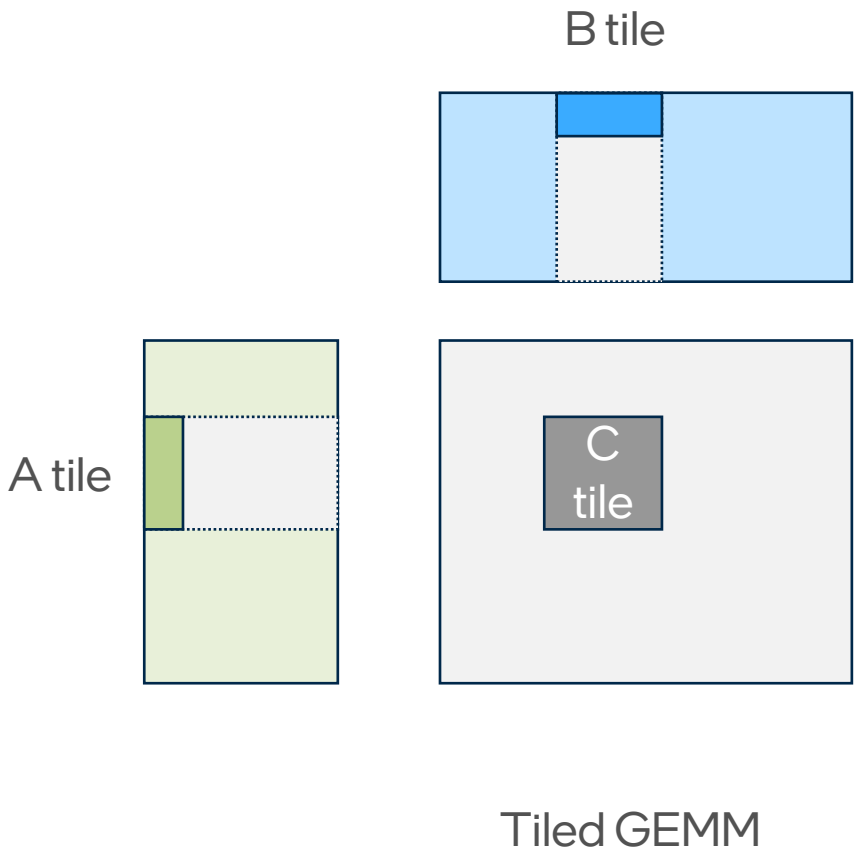
Charitha Saumya, Chao Chen, Jianhui Li
Intel Corporation

Compilers for Deep Learning Workloads



Tile-based Programming

- More flexibility in expressing the **algorithm** and **parallel strategy.**
- Easy to adopt in existing graph compilers using template-based codegen.
- Current MLIR dialects
  - Limitations in specifying data ownership in memref/vector.
  - Unavailability of workgroup/block level dialect.



Tiled GEMM

XeTile: A Tile Dialect for Workgroup-level Programming

XeTile Design Considerations

- Configurable tile sizes (Workgroups/subgroups/block size within subgroups)
- Support Advanced GEMM optimizations (Prefetch/software pipelining)
- Explicit control over Parallelization/Decomposition strategy

XeTile Implementation in MLIR

Problem: How to specify **Data Ownership** on memref/vector*?*

Extensions for **memref**

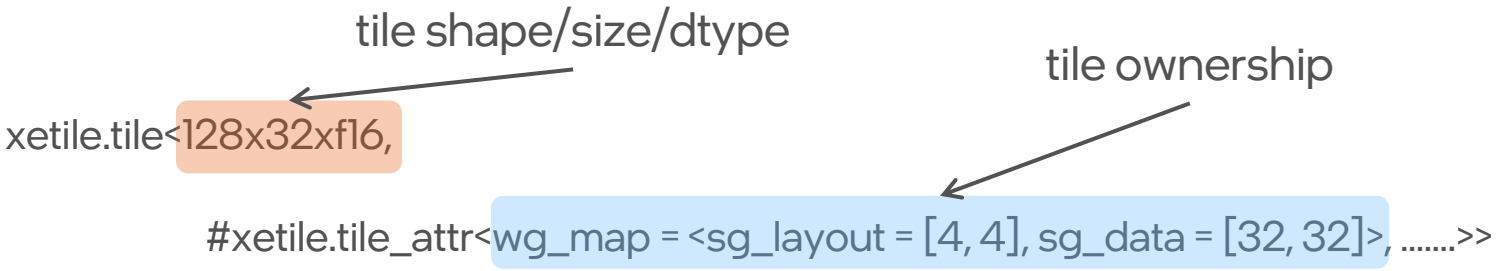
- **Tile data type** + ownership attributes.
- Ops for create/update/load/store tiles.

Extensions for **vector**

- Attach ownership attributes to the ops.

Tile Data Type

- Describes 2D memory region within an n-D memref.
- Specifies data decomposition into compute units (subgroups/work-item threads).
- Uses Round-robin data assignment.



Example 1

Tile : 128x128xf16

Sg_layout = [4, 4]

Sg_data = [32, 32]

128

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

128

SG data ownership

Example 2

Tile : 128x32xf16

Sg_layout = [4, 4]

Sg_data = [32, 32]

32

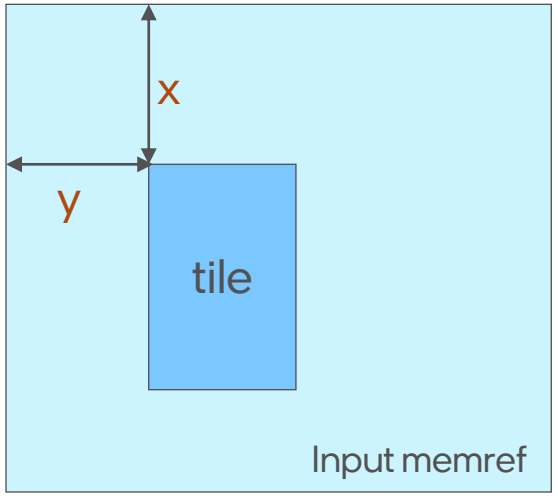
(0,0), (0,1), (0,2), (0,3)
(1,0), (1,1), (1,2), (1,3)
(2,0), (2,1), (2,2), (2,3)
(3,0), (3,1), (3,2), (3,3)

128

SG data ownership

XeTile Operations – Initializing a Tile

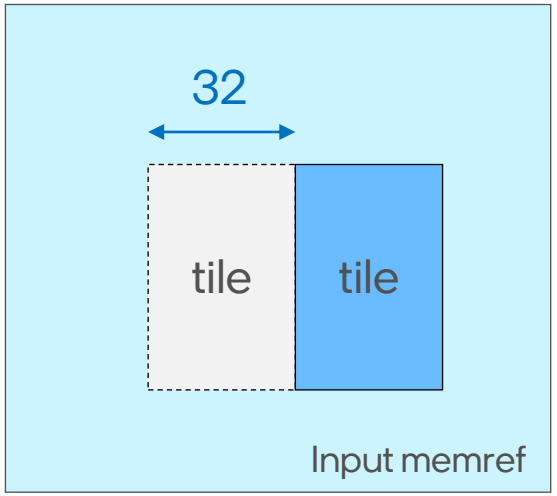
```
%2 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
%3 = load_tile %2 : !tile<256x256xf32, #map_c>-> vector<256x256xf32>

%4 = init_tile %arg0[%0, %c0] : memref<4096x4096xf16> -> !tile<256x32xf16, #map_a>
%5 = init_tile %arg1[%c0, %1] : memref<4096x4096xf16> -> !tile<32x256xf16, #map_b>

%6:3 = scf.for %arg3 = %c0 to %c4096 step %c32 iter_args(%arg4 = %4, %arg5 = %5, %arg6 = %3)
{
    %8 = load_tile %arg4 : !tile<256x32xf16, #map_a> -> vector<256x32xf16>
    %9 = load_tile %arg5 : !tile<32x256xf16, #map_b> -> vector<32x256xf16>

    %10 = update_tile_offset %arg4, [%c0,  %c32] : !tile<256x32xf16, #map_a>
    %11 = update_tile_offset %arg5, [%c32,  %c0] : !tile<32x256xf16, #map_b>

    %12 = tile_mma %8, %9, %arg6 {wg_map_c = #map_c}
       : vector<256x32xf16>, vector<32x256xf16>, vector<256x256xf32> -> vector<256x256xf32>

    scf.yield %10, %11, %12 : !tile<256x32xf16, #map_a>, !tile<32x256xf16, #map_b>, vector<256x256xf32>
}
%7 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
store_tile %6#2,  %7 : vector<256x256xf32>, !tile<256x256xf32, #map_c>
```



Initialize a tile on a memref

XeTile Operations – Move Tile

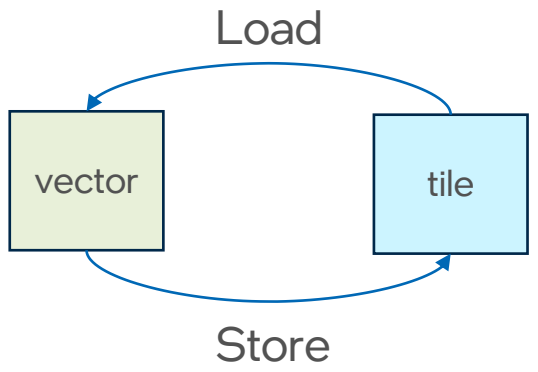
```
%2 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
%3 = load_tile %2 : !tile<256x256xf32, #map_c>-> vector<256x256xf32>

%4 = init_tile %arg0[%0, %c0] : memref<4096x4096xf16> -> !tile<256x32xf16, #map_a>
%5 = init_tile %arg1[%c0, %1] : memref<4096x4096xf16> -> !tile<32x256xf16, #map_b>

%6:3 = scf.for %arg3 = %c0 to %c4096 step %c32 iter_args(%arg4 = %4, %arg5 = %5, %arg6 = %3)
{
    %8 = load_tile %arg4 : !tile<256x32xf16, #map_a> -> vector<256x32xf16>
    %9 = load_tile %arg5 : !tile<32x256xf16, #map_b> -> vector<32x256xf16>

    %10 = update_tile_offset %arg4, [%c0,  %c32] : !tile<256x32xf16, #map_a>
    %11 = update_tile_offset %arg5, [%c32,  %c0] : !tile<32x256xf16, #map_b>

    %12 = tile_mma %8, %9, %arg6 {wg_map_c = #map_c}
       : vector<256x32xf16>, vector<32x256xf16>, vector<256x256xf32> -> vector<256x256xf32>

    scf.yield %10, %11, %12 : !tile<256x32xf16, #map_a>, !tile<32x256xf16, #map_b>, vector<256x256xf32>
}
%7 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
store_tile %6#2,  %7 : vector<256x256xf32>, !tile<256x256xf32, #map_c>
```



Moves the tile within memref

XeTile Operations – Tile Load/Store

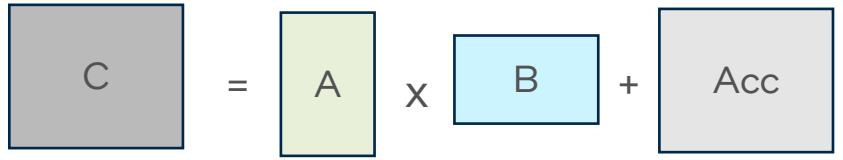
```
%2 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
%3 = load_tile %2 : !tile<256x256xf32, #map_c>-> vector<256x256xf32>

%4 = init_tile %arg0[%0, %c0] : memref<4096x4096xf16> -> !tile<256x32xf16, #map_a>
%5 = init_tile %arg1[%c0, %1] : memref<4096x4096xf16> -> !tile<32x256xf16, #map_b>

%6:3 = scf.for %arg3 = %c0 to %c4096 step %c32 iter_args(%arg4 = %4, %arg5 = %5, %arg6 = %3)
{
    %8 = load_tile %arg4 : !tile<256x32xf16, #map_a> -> vector<256x32xf16>
    %9 = load_tile %arg5 : !tile<32x256xf16, #map_b> -> vector<32x256xf16>

    %10 = update_tile_offset %arg4, [%c0,  %c32] : !tile<256x32xf16, #map_a>
    %11 = update_tile_offset %arg5, [%c32,  %c0] : !tile<32x256xf16, #map_b>

    %12 = tile_mma %8, %9, %arg6 {wg_map_c = #map_c}
       : vector<256x32xf16>, vector<32x256xf16>, vector<256x256xf32> -> vector<256x256xf32>

    scf.yield %10, %11, %12 : !tile<256x32xf16, #map_a>, !tile<32x256xf16, #map_b>, vector<256x256xf32>
}
%7 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
store_tile %6#2,  %7 : vector<256x256xf32>, !tile<256x256xf32, #map_c>
```



memref tile ↔ vector register tile

XeTile Operations – Tile MMA

```
%2 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
%3 = load_tile %2 : !tile<256x256xf32, #map_c>-> vector<256x256xf32>

%4 = init_tile %arg0[%0, %c0] : memref<4096x4096xf16> -> !tile<256x32xf16, #map_a>
%5 = init_tile %arg1[%c0, %1] : memref<4096x4096xf16> -> !tile<32x256xf16, #map_b>

%6:3 = scf.for %arg3 = %c0 to %c4096 step %c32 iter_args(%arg4 = %4, %arg5 = %5, %arg6 = %3)
{
    %8 = load_tile %arg4 : !tile<256x32xf16, #map_a> -> vector<256x32xf16>
    %9 = load_tile %arg5 : !tile<32x256xf16, #map_b> -> vector<32x256xf16>

    %10 = update_tile_offset %arg4, [%c0,  %c32] : !tile<256x32xf16, #map_a>
    %11 = update_tile_offset %arg5, [%c32,  %c0] : !tile<32x256xf16, #map_b>

    %12 = tile_mma %8, %9, %arg6 {wg_map_c = #map_c}
       : vector<256x32xf16>, vector<32x256xf16>, vector<256x256xf32> -> vector<256x256xf32>

    scf.yield %10, %11, %12 : !tile<256x32xf16, #map_a>, !tile<32x256xf16, #map_b>, vector<256x256xf32>
}
%7 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
store_tile %6#2,  %7 : vector<256x256xf32>, !tile<256x256xf32, #map_c>
```



MMA on vector register tiles

XeTile – Data Ownership and Decomposition

```
%2 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
%3 = load_tile %2 : !tile<256x256xf32, #map_c>-> vector<256x256xf32>

%4 = init_tile %arg0[%0, %c0] : memref<4096x4096xf16> -> !tile<256x32xf16, #map_a>
%5 = init_tile %arg1[%c0, %1] : memref<4096x4096xf16> -> !tile<32x256xf16, #map_b>

// Initialize prefetch tiles
// Prefetch A, B
%6:5 = scf.for %arg3 = %c0 to %c4096 step %c32 iter_args (…)
{
    %8 = load_tile %arg4 : !tile<256x32xf16, #map_a> -> vector<256x32xf16>
    %9 = load_tile %arg5 : !tile<32x256xf16, #map_b> -> vector<32x256xf16>
    prefetch_tile %arg6 : !tile<256x32xf16, #map_a_prefetch>
    prefetch_tile %arg7 : !tile<32x256xf16, #map_b_prefetch>

    // Update offsets for load and prefetch tiles.
    %12 = tile_mma %8, %9, %arg6 {wg_map_c = #map_c}
       : vector<256x32xf16>, vector<32x256xf16>, vector<256x256xf32> -> vector<256x256xf32>

    scf.yield …
}
%7 = init_tile %arg2[%0, %1] : memref<4096x4096xf32> -> !tile<256x256xf32, #map_c>
store_tile %6#2,  %7 : vector<256x256xf32>, !tile<256x256xf32, #map_c>
```

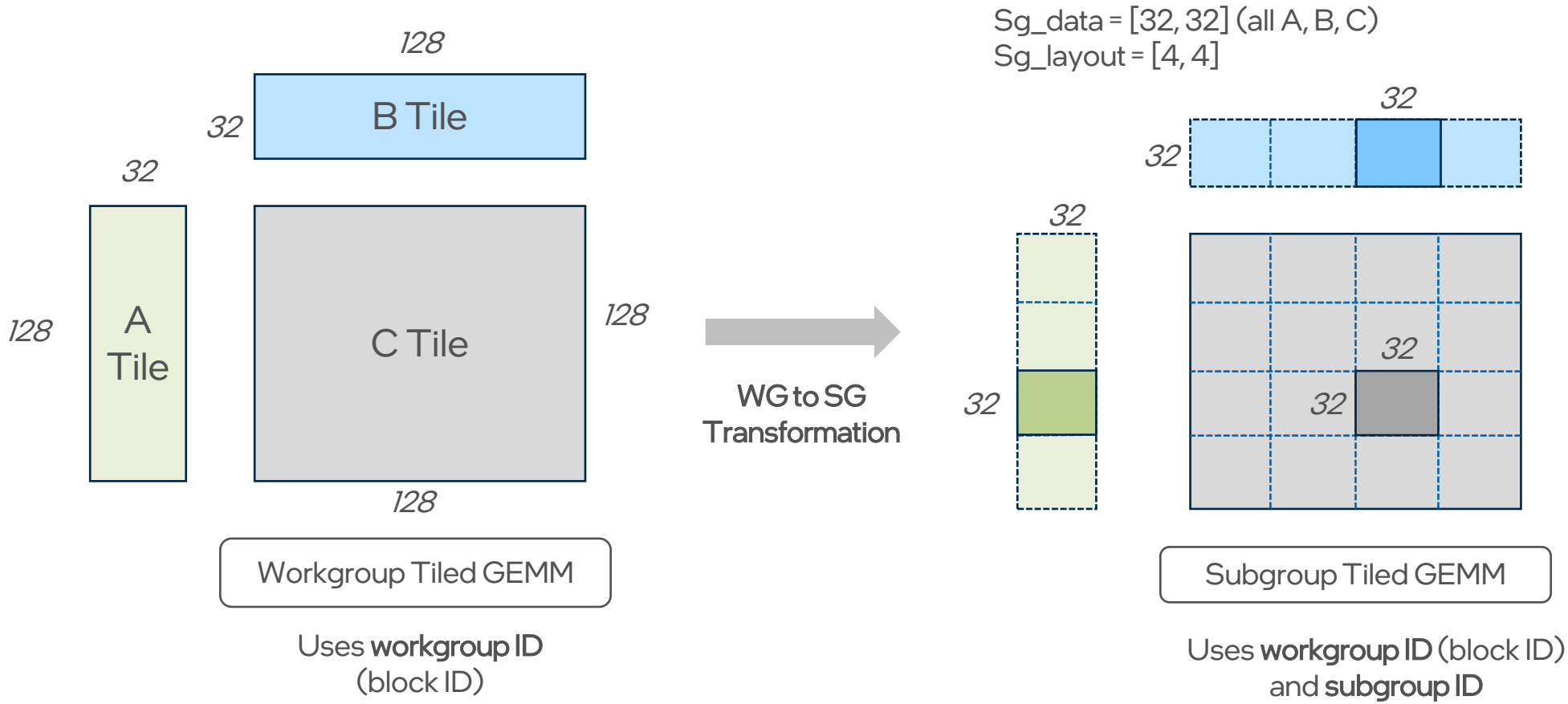
#map_c = { sg_layout = [8, 4], sg_data = [32, 64]

#map_a = { sg_layout = [8, 4], sg_data = [32, 32]

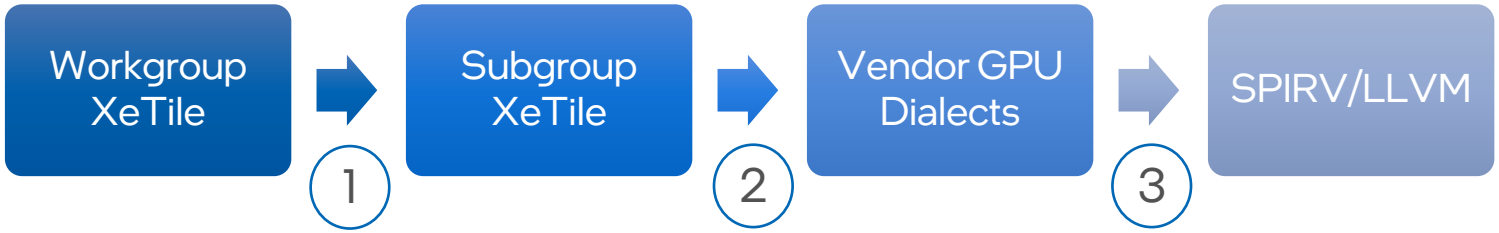
#map_b = {sg_layout = [8, 4], sg_data = [32,64]

#map_a_prefetch = { sg_layout = [32, 1], sg_data =[8, 32] }

Workgroup to Subgroup Decomposition



XeTile Lowering and Performance

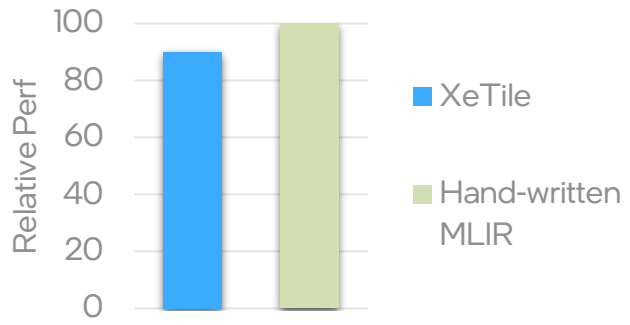


① Workgroup to subgroup decomposition

② Subgroup to vendor-specific GPU dialects (e.g., XeGPU)

③ Convert to LLVM/SPIRV and pass to backend compiler/driver

4K GEMM performance reaching ~90% of hand-written optimized MLIR performance



Summary

- MLIR lacks a workgroup/block-level tile dialect and ways to specify data ownership within tiles.
- We introduce XeTile, a workgroup-level tile dialect for GPUs.
- XeTile extends memref/vector to enable tile-based programming at workgroup level.
- Initial performance of XeTile is promising.

Our work is open-source

https://github.com/intel/mlir-extensions

Thank You