Modular

Simplifying GPU Programming With Parametric Tile-Level Tensors in Mojo 🔥

Ahmed Taei
ataei@modular.com

Hengjie Wang
hengjiewang@modular.com

LLVM Dev 2024

A unified team of thirty-two,
Working together, we're sure to pull through.
In synchronized fashion, no threads left behind,
Executing your MatrixMultiply with speed in mind.

Who am I ?

A unified team of thirty-two,
Working together, we're sure to pull through.
In synchronized fashion, no threads left behind,
Executing your MatrixMultiply with speed in mind.

Who am I ? Warp executing mma.sync.aligned instruction

Modern GPUs Architecture

- Massively parallel machine
  - Massive parallelism (128 SMs in GA100 GPUs)
  - Massive on chip memory
- Heterogeneous processing units
  - General purpose cores (CUDA Core) ~ 40 TFLOPS
  - Fixed function cores (Tensor Core) ~ 310 TFLOPS
- High bandwidth off-chip memory (HBM2) 1.5 TB/s

Modern GPUs Architecture

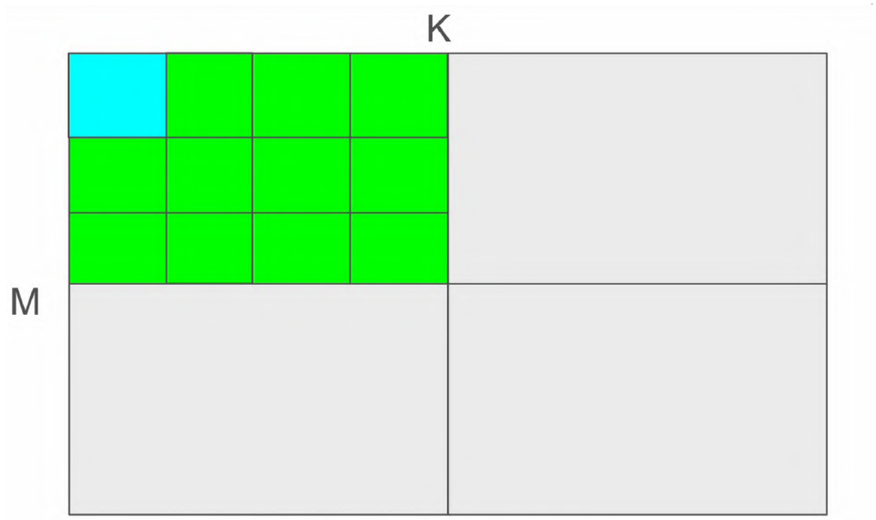
- Next generation (Hopper) introduces
  - More parallelism (144 SMs in GH100 GPUs)
  - More on chip memory
- Faster GP cores ~ (CUDA Cores) 120 TFLOPS
- **New & faster** fixed function cores (Tensor Core) ~ 1000 TFLOPS
- **New** specialized accelerators Tensor Memory Accelerator (TMA)
- Higher bandwidth for accessing off-chip memory (HBM3) 3 TB/s
- Performance isn't portable 😓
  - New ISA + New fixed function cores → New tricks
  - Compiler backends are catching up slowly!

Modern GPUs Programming Model

- Single Instruction Multiple Threads (SIMT)
  - All threads are executing same instruction in parallel
- Every 32 consecutive threads — warp — are scheduled together
  - Threads in a warp are executed in parallel ✌️
  - Warp level instructions, Tensor Core MMA, warp shuffles ...etc
  - New architectures define instructions wider than a single warp
    - Hopper's (W)arp(G)roupMMA instruction

The Missing Abstraction

- SIMT exposes GPU parallelism but!
  - HW groups threads together into warps
  - Programmers wants to assign work to a group of thread
- Most GPU programms especially **dense kernels** (e.g matmul) reduces to **data parallel operations on tiles**
  - Each thread block accesses a specific tile
  - Each warp in the thread blocks accesses a specific subtile
  - Each thread in the warp accesses a subtile of elements
  - The program is a operations on these tiles (load, copy, math...)
- Performant kernels wants to use fixed function cores within the GPU
  - TMA, Tensor Core ISA is naturally defines as an operations on a tile of data



Assign Tiles To Warps

The Missing Abstraction

- Tensor core MMA instruction
  - mma.sync.aligned.row.col.m16n8k8.bf16.bf16.tf32
  - Given a warp tile distribute its 32 threads in a particular layout
  - operand_a : Row major threads with 2 x [1x2] thread subtile
  - operand_b : Col major threads with 2x1 thread subtile
  - operand_c : Row major threads with 2 x [1x2] thread subtile

Row\Column	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15
0	T0:{a0,a1}	T1:{a0,a1}	T2:{a0,a1}	T3:{a0,a1}
1	T4:{a0,a1}	T5:{a0,a1}	T6:{a0,a1}	T6:{a0,a1}
2				
...				
7	T28:{a0,a1}	T29:{a0,a1}	T30:{a0,a1}	T31:{a0,a1}
8	T0:{a2,a3}	T1:{a2,a3}	T2:{a2,a3}	T3:{a2,a3}
9	T4:{a2,a3}	T5:{a2,a3}	T6:{a2,a3}	T7:{a2,a3}
10				
...				
15	T28:{a2,a3}	T29:{a2,a3}	T30:{a2,a3}	T31:{a2,a3}

Row\Column	0	1	2	..	7
0 1	$T0:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$	$T4:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$			$T28:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$
2 3	$T1:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$	$T5:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$			$T29:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$
4 5	$T2:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$	$T6:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$			$T30:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$
6 7	$T3:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$	$T7:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$			$T31:\begin{Bmatrix} b0 \\ b1 \end{Bmatrix}$

Row\Col	0 1	2 3	4 5	6 7
0	T0: {c0, c1}	T1: {c0, c1}	T2: {c0, c1}	T3: {c0, c1}
1	T4: {c0, c1}	T5: {c0, c1}	T6: {c0, c1}	T7: {c0, c1}
2				
..				
7	T28: {c0, c1}	T29: {c0, c1}	T30: {c0, c1}	T31: {c0, c1}
8	T0:{c2, c3}	T1:{c2, c3}	T2:{c2, c3}	T3:{c2, c3}
9	T4:{c2, c3}	T5:{c2, c3}	T6:{c2, c3}	T7:{c2, c3}
10				
..				
15	T28:{c2, c3}	T29:{c2, c3}	T30:{c2, c3}	T31:{c2, c3}

The Missing Abstraction

- GPU kernel developers wants explicit parallelism but not necessarily SIMT

- For dense kernels they want higher level primitives like a tensor and operations on it
  - Tile, allocation, copy, math ...etc
  - Given an operation on a warp tile how threads are organized to access each element
  - Expose knobs and parametrize their kernels for auto-tuning or even manual experimentation

The Missing Abstraction

- This isn't a novel problem
  - Libraries exist to provide **zero cost** higher level abstractions
  - The challenge with libraries is how much the language can offer

- More specifically the language empowering the libray needs to provide an easy way for
  - Transformation of compile time type information (meta-programming)
  - Access the hardware without inline assembly (remember compiler backends are catching up!)

Parametric Tensor Type

- Thanks to Mojo 🔥
  - Meta programming is much simpler and expressive 🎉
  - Raw access to MLIR operations 💪
- Our approach
  - **A Library defined Tensor** type that provides operations:
    - Tiled, distributed and binded to specific part of the compute hierarchy
    - Access to its elements can be explicitly vectorized
  - Tensor type parameterized by layout meta-types, which specify its shape and the way its elements are accessed
  - For this you **don't need a DSL or a compiler** you just need a library

Meta Programming in Mojo 🔥

- Parameters are like templates but they are compile time typed values (TypedAttr!) not AST substitutions!

```
struct Tensor[dtype: DType, rank: Int]:
    var data: UnsafePointer[Scalar[dtype]]
    var shape: IndexList[rank]
    var stride: IndexList[rank]
```

```
template <typename T, int rank> class Tensor {
  T *data;
  size_t shape[rank];
  size_t stride[rank];
};
```

- Same language for programming and metaprogramming

```
fn size(dims: List[Int]) -> Int:
    res = 1
    for i in range(len(dims)):
        res *= dims[i]
    return res

alias compile_time_size = size(List[Int](1, 2, 3))
run_time_size = size(List[Int](1, 2, 3))
```

```
template<int... Values>
constexpr int size(std::integer_sequence<int, Values...>) {
    int res = 1;
    int arr[] = {Values...};
    for (int i = 0; i < sizeof...(Values); ++i) {
        res *= arr[i];
    }
    return res;
}

constexpr int result = size(std::integer_sequence<int, 2, 3, 4>{});
```

Meta Programming in Mojo 🔥

- Meta types can also be a parametric closure, which is super powerful
  - Software pipelined mma reduction loop with an SIMD → SIMD elementwise epilogue

```
alias elementwise_epilogue_type = fn[
    type: DType, width: Int
] (Int, Int, SIMD[type, width]) capturing -> SIMD[type, width]

fn multistage_mma[
    num_stages: Int,
    elementwise_epilogue: elementwise_epilogue_type
](
    inout res: Tensor,
    lhs: Tensor,
    rhs: Tensor
):
    # construct a software pipelined mma with num_stages and inline
    # and apply elementwise epiloug
```

  - Useful for ML graph compilers / code gen:
    - No need to write complex software pipelined loops as compiler pass
    - Fused into pipelined loop is **easier** than pipeline a fused loop

Layout Parameterized Tensor Type

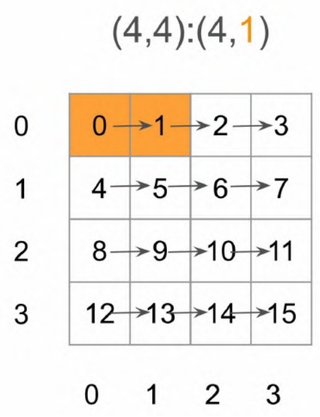
- A Tensor type with **Layout(s)** meta types
  - Data layout
  - Element layout
- What is the layout ?
  - A function logical (n-d) coords → linear coords / Integer
  - Represent how data or threads spatially organized
  - Defined by {Tuple(shape), Tuple(stride)}
  - Layout(coords) → dot(coords, stride)
- Same definition for Layout in CUTLASS/CUTE
  - Slightly different operations / algebra

```
struct Layout:
    var shape: Tuple
    var stride: Tuple

    fn __init__(inout self, shape: Tuple, stride: Tuple):
        self.shape = shape
        self.stride = shape

    fn rank(self) -> Int:
        return self.shape.rank()

    fn size(self) -> Int:
        return self.shape.size()

    fn __call__(self, *coords: Int) -> Int:
        return dot_product(coords, self.stride)
```

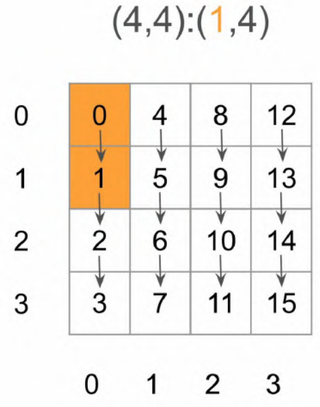
```
struct Tensor[
    dtype: DType,
    layout: Layout,
    /,
    *,
    element_layout: Layout = Layout(1, 1),
    address_space: AddressSpace = AddressSpace.GENERIC,
]:
    var ptr: UnsafePointer[Scalar[dtype], address_space]
```

Layout for Data And Threads

- Example: 4x4 row major matrix:
  - shape = (4, 4), stride = (4, 1)
  - layout(i, j) = dot((i, j), (4, 1)) = 4 * i + j
  - domain: (0, 0)..(3, 3)
  - range: 0 → 15



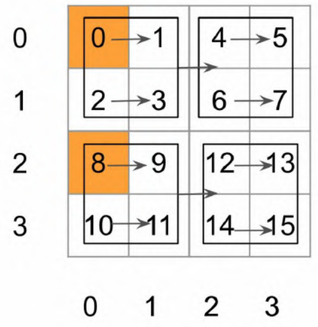
- Example: 4x4 column major matrix:
  - shape = (4, 4), stride = (1, 4)
  - layout(i, j) = dot((i, j), (1, 4)) = i + 4 * j
  - domain: (0, 0)..(3, 3)
  - range: 0 → 15



Layout For Data And Threads

- How about a more complicated layout ?
  - e.g 4x4 with 2x2 tile major
  - Split each dim (4, 4) → ((2,2), (2,2))
  - (4:4) → (2,2):(2,8), (4:1) → (2, 2):(1, 4)
  - Inner tile stride (2, 1), Tile stride (8, 4)
  - OpenXLA/IREE's mmt4d uses similar but flattened representation (2,2,2,2):(8,4,2,1)
- With this we can support more general layouts.

((2,2),(2,2)):((2,8),(1,4))



Tensor Layout Transforms

- Tile and return the tensor tile at specific coordinates

```
fn tile_layout(tile_sizes: VariadicList[Int], layout: Layout) -> Layout:
    return Layout(tile_sizes, layout.stride)
```

- Returns a vectorized tensor with specific vec_shape

```
fn vectorize_layout(vector_shape: VariadicList[Int], layout: Layout) -> Layout:
    return Layout(
        ceildiv(layout.shape, vector_shape), layout.stride * vector_shape
    )


fn vectorize_element(vector_shape: VariadicList[Int], layout: Layout) -> Layout:
    return Layout(vector_shape, layout.stride)
```

- Distribute the thread layout into the tensor and return thread_id's tile

```
fn distribute_layout(thread_layout: Layout, layout: Layout) -> Layout:
    return Layout(
        ceildiv(layout.shape, thread_layout.shape),
        thread_layout.shape * layout.stride,
    )
```

```
struct Tensor[
    dtype: DType,
    layout: Layout,
    /,
    *,
    element_layout: Layout = Layout(1, 1),
    address_space: AddressSpace = AddressSpace.GENERIC,
]:
    var ptr: UnsafePointer[Scalar[dtype], address_space]

    fn __init__(inout self, ptr: UnsafePointer[Scalar[dtype], address_space]):
        self.ptr = ptr

    fn tile[
        *tile_sizes: Int
    ](self, *coords: Int) -> Tensor[
        dtype,
        tile_layout(tile_sizes, layout),
        element_layout=element_layout,
        address_space=address_space,
    ] as res:
        tile_offset = dot_product(coords, tile_sizes)
        return __type_of(res)(self.ptr.offset(tile_offset))

    fn vectorize[
        *vec_shape: Int
    ](self) -> Tensor[
        dtype,
        vectorize_layout(vec_shape, layout),
        element_layout = vectorize_element(vec_shape, element_layout),
        address_space=address_space,
    ] as res:
        return __type_of(res)(self.ptr)

    fn distribute[
        thread_layout: Layout
    ](self, thread_id: Int) -> Tensor[
        dtype,
        distribute_layout(thread_layout, layout),
        element_layout=element_layout,
        address_space=address_space,
    ] as res:
        thread_offset = layout(thread_layout.to_coords(thread_id))
        return __type_of(res)(self.ptr.offset(thread_offset))
```

Tensor Layout Transforms

- Distributes tiles the data to the thread layout shape then distribute (repeat)

thread_layout
(2,2):(1,2)

0	2
1	3

(8,8):(8,1)

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(4,4):(16,2) + 8

8	10	12	14
24	26	28	30
40	42	44	46
56	58	60	62

Thread 1

Tile Level Tensor operations

- With tensor tiles tiles as the primitives we can define operations for:
  - Memory allocations
  - Data movement
  - Math

```
# Allocate BN x BK column major tensor on shared memory.
b_sram_tile = tb[f32]().col_major[BN, BK]().shared().alloc()
```

```
# allocate TM x 1 tensor in registers
a_reg_tile = tb[f32]().row_major[TM, 1]().local().alloc()
```

```
copy_dram_to_sram_async[
    thread_layout=thread_layout_loadb
](
    smem_tile.vectorize[1, simd_size]().,
    gmem_tile.vectorize[1, simd_size]().,
)

async_copy_commit_group()
```

```
# outer product and acc into res
res += outer_product(lhs, rhs)
```

Tile Level Tensor operations

- Simplify Tensor Core Programming

```
    @parameter
    for mma_k in range(BK // MMA_K):

        @parameter
        for mma_m in range(WM // MMA_M):

            @parameter
            for mma_n in range(WN // MMA_N):
                c_reg_m_n = c_reg.tile[1, TN](mma_m, mma_n)

                # Tile warp tiles into MMA shapes.
                A_mma_tile = A_warp_tile.tile[MMA_M, MMA_K](mma_m, mma_k)
                B_mma_tile = B_warp_tile.tile[MMA_K, MMA_N](mma_k, mma_n)

                # Load data from warps tiles to register tiles.
                a_reg = mma_op.load_a(A_mma_tile)
                b_reg = mma_op.load_b(B_mma_tile)

                # Compute MMA on data loaded from registers.
                c_reg_m_n = mma_op.mma_op(
                    a_reg,
                    b_reg,
                    c_reg_m_n,
                )

@parameter
for mma_m in range(WM // MMA_M):

    @parameter
    for mma_n in range(WN // MMA_N):
        C_mma_tile = C_warp_tile.tile[MMA_M, MMA_N](mma_m, mma_n)
        c_reg_m_n = c_reg.tile[1, TN](mma_m, mma_n)
        mma_op.store_d(C_mma_tile, c_reg_m_n)
```

```
shape for
.
N, MMA_K)]()

).alloc()
).alloc()
```

```
# Vectorized async copy for data from DRAM -> SRAM
# Vectorization for both reading data from DRAM & copy instruction
# itself.
copy_dram_to_sram_async[thread_layout = Layout.row_major(4, 8)](
    A_sram_tile.vectorize[1, TN](), A_dram_tile.vectorize[1, TN]()
)
copy_dram_to_sram_async[thread_layout = Layout.row_major(4, 8)](
    B_sram_tile.vectorize[1, TN](), B_dram_tile.vectorize[1, TN]()
)
async_copy_wait_all()
barrier()

# Tile SRAM data into Warp tiles
A_warp_tile = A_sram_tile.tile[WM, BK](warp_y, 0)
B_warp_tile = B_sram_tile.tile[BK, WN](0, warp_x)

@parameter
for mma_k in range(BK // MMA_K):

    @parameter
    for mma_m in range(WM // MMA_M):

        @parameter
        for mma_n in range(WN // MMA_N):
            c_reg_m_n = c_reg.tile[1, TN](mma_m, mma_n)
```

How GPU Primitives are Implemented?

- Mojo is syntax sugar on top of MLIR so we can:
  - Use an LLVM intrinsic operation
  - Use an MLIR operation
    - LLVM is catching up with Hopper
    - Thanks to MLIR's NVVM ops!
  - Mojo language type → LLVMIR types
    - High level types transforms in the library
    - No need for HL Dialect → NVVMOps

```
@always_inline
fn __to_llvm_ptr[
    type: AnyType
](ptr: UnsafePointer[type]) -> __mlir_type.`!llvm.ptr`:
    """Cast a pointer to LLVMPointer Type.

    Args:
        ptr: A pointer.

    Returns:
        A pointer of type !llvm.ptr.
    """
    return __mlir_op.`builtin.unrealized_conversion_cast`[
        _type = __mlir_type.`!llvm.ptr`
    ](ptr)


@always_inline
fn __to_i32(val: Int32) -> __mlir_type.i32:
    """Cast Scalar I32 value into MLIR i32.

    Args:
        val: Scalar I32 value.

    Returns:
       Input casted to MLIR i32 value.
    """
    return __mlir_op.`pop.cast_to_builtin`[_type = __mlir_type.`i32`](val.value)
```

```
f(c):
atrix-multiply and accumulate(MMA)

6.f16",
t16, 2], SIMD[DType.float16, 2]],
```

```
__mlir_op.`nvvm.cp.async.bulk.tensor.shared.cluster.global`(
    __to_llvm_shared_mem_ptr(dst_mem),
    __to_llvm_ptr(tma_descriptor),
    __to_i32(coords[0]),
    __to_i32(coords[1]),
    __to_llvm_shared_mem_ptr(mem_bar),
)
```

Fast GEMM On Modern GPUs

Map tiles seamlessly to memory/threads hierarchy

```
for k_tile in range(num_k_tiles):

    # Async copy from global memory to shared memory.
    # thread block tile.
    var a_gmem_tile = A.tile[BM, BK](BlockIdx.y(), k_tile)
    var a_smem_tile = LayoutTensor[a_type, Layout.row_major(BM, BK)]()

    async_copy[thread_layout](
        a_smem_tile.vectorize[simd_width]()
        a_gmem_tile.vectorize[simd_width]()
    )
    # ...

    # Warp tile.
    var a_warp_tile = a_smem_tile.tile[WM, WK](warp_m, warp_k)
    var b_warp_tile = b_smem_tile.tile[WN, WK](warp_n, warp_k)

    # Threads' register tile (fragments).
    mma.load_a(a_fragments, a_warp_tile)
    mma.load_b(b_fragments, b_warp_tile)

    # Tensor core mma
    mma.mma(c_fragments, a_fragments, b_fragments)
```

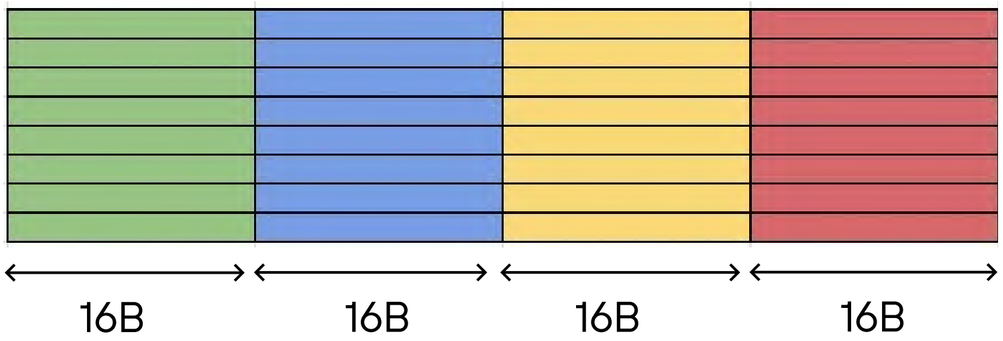
special layout to avoid bank conflict

Distribute workload to thread

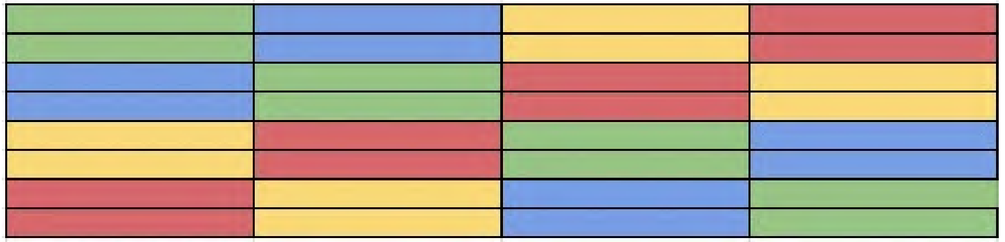
```
fn async_copy[thread_layout: Layout](
    smem_tile: LayoutTensor, gmem_tile: LayoutTensor
):
    # Distribute copy workload to threads.
    var smem_fragments = smem_tile.distribute[thread_layout](TheadIdx.x())
    var gmem_fragments = gmem_tile.distribute[thread_layout](TheadIdx.x())

    # Each thread copies its share.
    smem_fragments.copy(gmem_fragments)
```

Express thread swizzling

ldmatrix(.x1) loads one matrix of 8 rows and 16B per row.
Loading naively results in 4-way bank conflict!



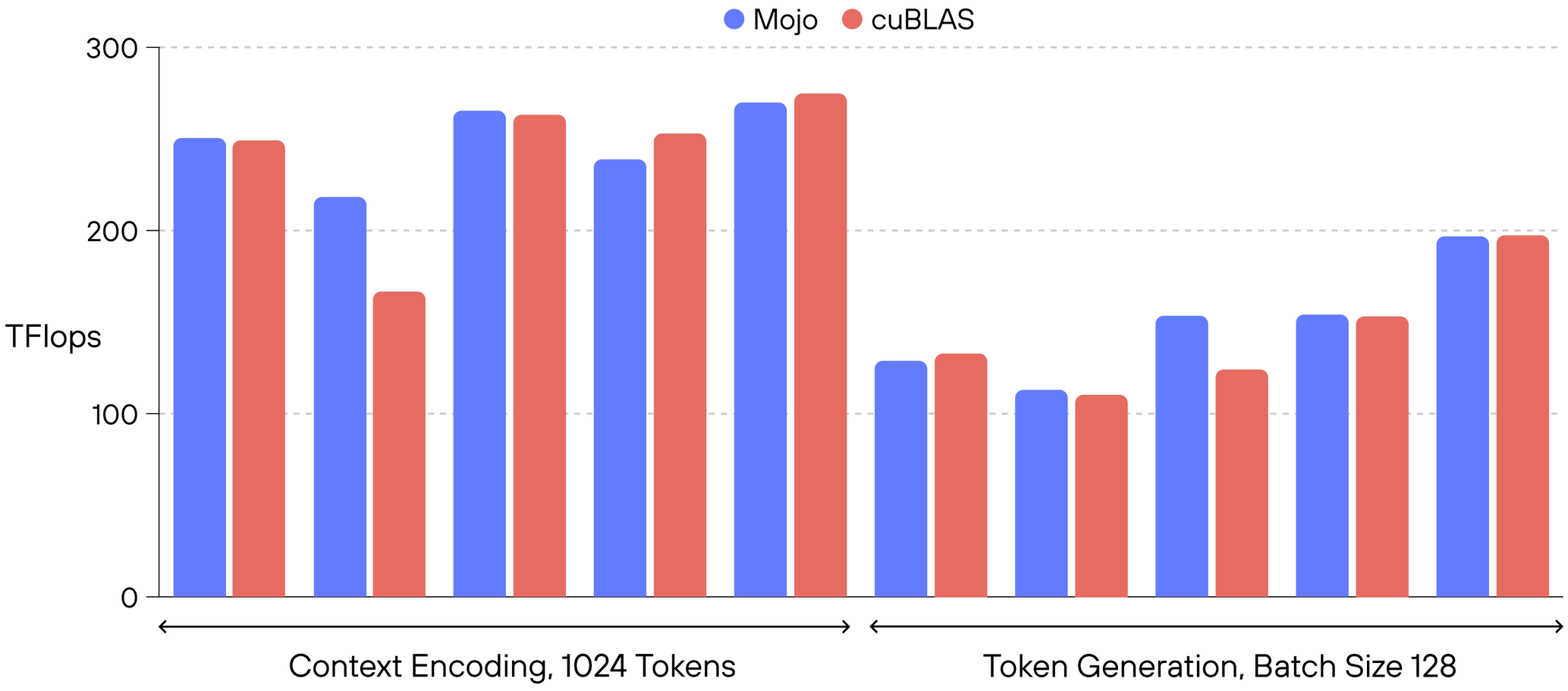
Need to "swizzle" the data layout to avoid conflicts.



```
var offset = ComposedLayout[
    composition(
        # shared memory tile layout
        Layout.row_major(BM, BK),
        # ldmatrix's required layout
        ldmatrix_layout,
    ),
    # swizzle to avoid bank conflict
    Swizzle[2, 3, 3](),
](ThreadIdx.x())
```

Tile-Based Gemm Performance

On-par performance with cuBLAS 12.6.1 on A100 for LLAMA3 BF16 serving



Parametric Tensor vs Triton Lang

- Both are tile level APIs, but Triton is explicitly only at the level of thread block tile.
- Triton is compiler based approach so, many things are implicit and done by the compiler
  - Mapping of thread block level operations to warps and threads
  - Allocation/reuse of intermediate shared memory and/or register tiles
  - Synchronization
- Our approach all scheduling aspects are explicit
  - Allocation and synchronization are explicit and user defined
  - Tiling and distribution to warps and threads is explicit and user defined
- We believe that we can go from explicit warp level to "optionally" implicit block level APIs
  - With a mixture of Library + Minimal compiler passes for lowering implicit high level operations

Conclusion

- GPU architecture is moving fast, faster than compiler and perofmance engineers can catchup
- The hardware exposes tile level instructions so as the algorithms
- SIMT is low level, with good language features and higher level abstractions it can be simplified

Thank You
Questions

Open roles at Modular ⬆️