Vortex proposal for triton & subsequent changes

Agenda

The essentials required by triton

TritonIR & TritonGPU IR

Pytorch support

Instructions vortex needs to support

architecture changes

CUDA/OpenCL

Memory Access: Allows direct load/store operations from global memory. Programmers manage global, shared, and local memory explicitly.

Thread Model: Uses threads organized in a hierarchy of blocks and grids. Threads within a block can share data via shared memory.

Synchronization: Provides explicit synchronization primitives within a thread block (e.g., `__syncthreads()`).

Operates on various kinds of data Tensors, buffers, images all of which may have differing access patterns

Triton

- Triton does not have a way to load/store from global memory by design
- instead users and can load a "slice" of a tensor from global memory to shared memory
- or store a "slice" of a tensor from shared memory to global memory

- Synchronization: Abstracts many synchronization details, simplifying the programming model for ML
- Pretty much only operates on tensors

Example of simple add triton kernel

```
@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.

    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM to shared memory, masking out any extra elements in case the
    # input is not a multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)
```

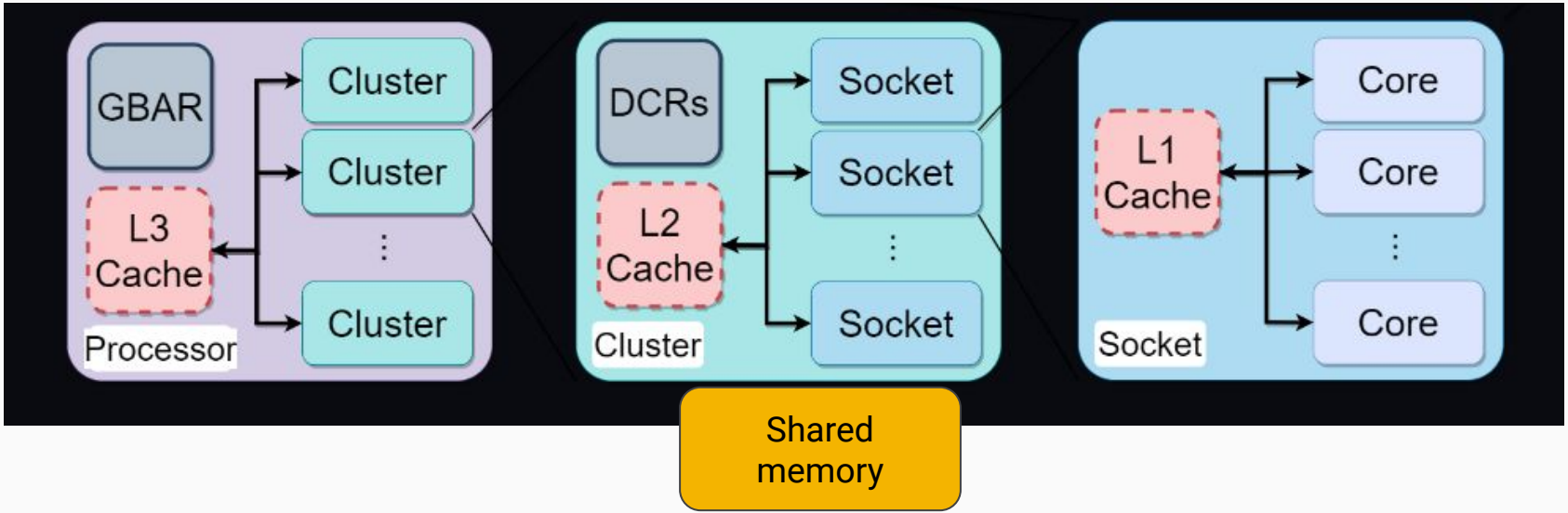
Triton Ops which requires Arch/ISA changes

tl.load/store: Vortex does not have a way to perform async shared memory operations

tl.dot: Vortex Tensor cores are WIP and can be used for dot ops*

Misc instructions: warp shuffle,tl.atomics, tan etc

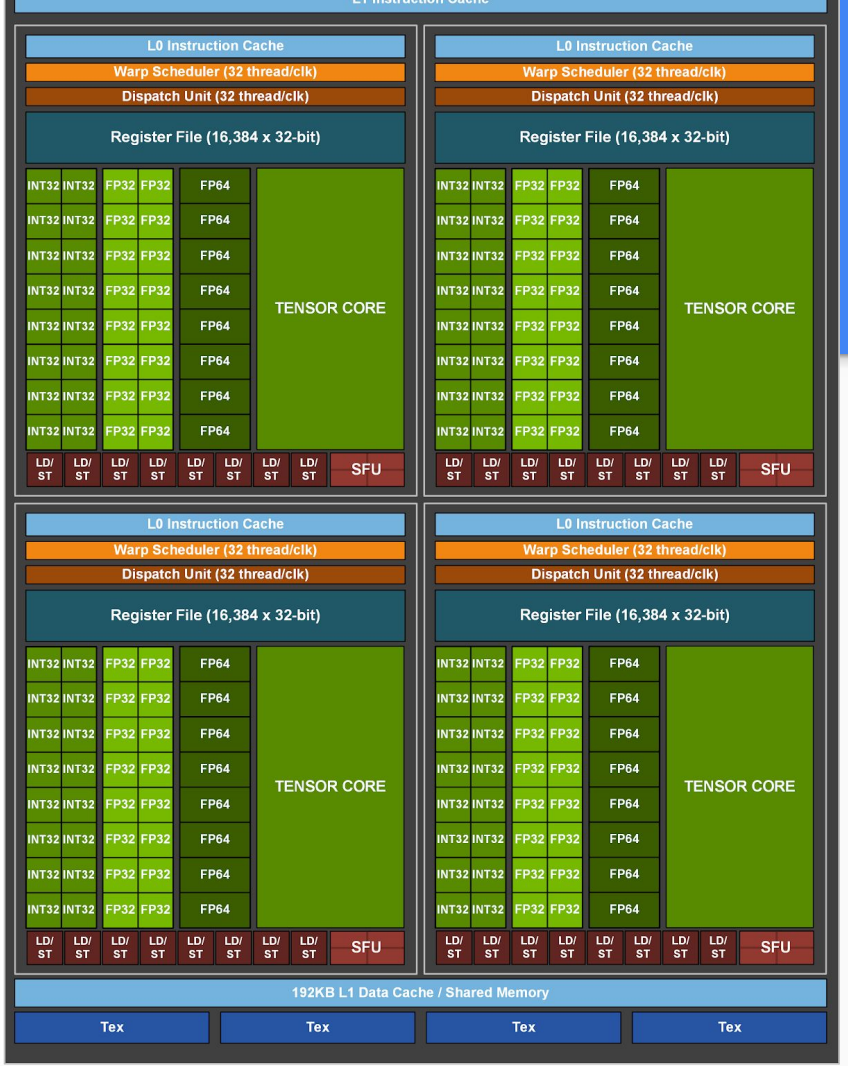
Shared memory



Shared memory

load from gmem to smem
store from smem to gmem
store from smem to smem(copy)

Will require a different Address space, ISA instructions



TritonIR vs TritonGPU IR

•Memory Operations: Triton IR uses generic loads/stores; TritonGPU IR has specialized async and local memory operations
•Execution Model: Triton IR is more abstract; TritonGPU IR explicitly manages asynchronous execution and synchronization
•Memory Hierarchy: TritonGPU IR exposes GPU-specific memory types (shared, local) and layouts (#blocked, #mma)
•Hardware Specificity: TritonGPU IR includes GPU-specific attributes (CTAs, warps) and optimizations for tensor cores
•Optimization Level: Triton IR is higher-level and more portable; TritonGPU IR allows for more fine-grained GPU-specific optimizations

Snippet of TritonIR

```
%47 = tt.splat %46 : i32 -> tensor<32x32xi32>
%48:3 = scf.for %arg9 = %c0_i32 to %45 step %c1_i32 iter_args(%arg10 = %cst, %arg11 = %34, %arg12 = %43) -> (tensor<128x3
  %66 = arith.muli %arg9, %c32_i32 : i32
  %67 = arith.subi %arg5, %66 : i32
  %68 = tt.splat %67 : i32 -> tensor<1x32xi32>
  %69 = arith.cmpi slt, %29, %68 : tensor<1x32xi32>
  %70 = tt.broadcast %69 : tensor<1x32xi1> -> tensor<128x32xi1>
  %71 = tt.load %arg11, %70, %cst_1 : tensor<128x32x!tt.ptr<f16>>
  %72 = tt.splat %67 : i32 -> tensor<32x1xi32>
  %73 = arith.cmpi slt, %35, %72 : tensor<32x1xi32>
  %74 = tt.broadcast %73 : tensor<32x1xi1> -> tensor<32x32xi1>
  %75 = tt.load %arg12, %74, %cst_0 : tensor<32x32x!tt.ptr<f16>>
  %76 = tt.dot %71, %75, %arg10, inputPrecision = tf32 : tensor<128x32xf16> * tensor<32x32xf16> -> tensor<128x32xf32>
  %77 = tt.addptr %arg11, %cst_2 : tensor<128x32x!tt.ptr<f16>>, tensor<128x32xi32>
  %78 = tt.addptr %arg12, %47 : tensor<32x32x!tt.ptr<f16>>, tensor<32x32xi32>
  scf.yield %76, %77, %78 : tensor<128x32xf32>, tensor<128x32x!tt.ptr<f16>>, tensor<32x32x!tt.ptr<f16>>
}
```

Snippet of TritonGPU IR

```
#blocked = #triton_gpu.blocked<{sizePerThread = [1, 8], threadsPerWarp = [8, 4], warpsPerCTA = [4, 1], order = [1, 0]}>
#mma = #triton_gpu.nvidia_mma<{versionMajor = 2, versionMinor = 0, warpsPerCTA = [4, 1], instrShape = [16, 8]}>
#shared = #triton_gpu.shared<{vec = 8, perPhase = 2, maxPhase = 4, order = [1, 0], hasLeadingOffset = false}>
module attributes {"triton_gpu.num-ctas" = 1 : i32, "triton_gpu.num-warps" = 4 : i32, triton_gpu.target = "cuda:86", "triton_gpu.threa
  tt.func public @matmul_kernel(%arg0: !tt.ptr<f16> {tt.divisibility = 16 : i32}, %arg1: !tt.ptr<f16> {tt.divisibility = 16 : i32}, %a
    ......
    %62 = tt.broadcast %61 : tensor<32x1xi1, #blocked> -> tensor<32x32xi1, #blocked>
    %63 = triton_gpu.memdesc_subview %50[%c0_i32, %c0_i32, %c0_i32] : !tt.memdesc<3x32x32xf16, #shared, mutable> -> !tt.memdesc<32x32x
    %94 = tt.broadcast %93 : tensor<1x32xi1, #blocked> -> tensor<128x32xi1, #blocked>
    %95 = triton_gpu.memdesc_subview %49[%c2_i32, %c0_i32, %c0_i32] : !tt.memdesc<3x128x32xf16, #shared, mutable> -> !tt.memdesc<128x3
    %96 = tt.splat %88 : i1 -> tensor<128x32xi1, #blocked>
    %97 = arith.andi %96, %94 : tensor<128x32xi1, #blocked>
    %98 = triton_gpu.async_copy_global_to_local %89, %95 mask %97 other %cst_0 : tensor<128x32x!tt.ptr<f16>, #blocked> -> <128x32xf16,
    %99 = triton_gpu.async_commit_group %98
    %100 = tt.splat %91 : i32 -> tensor<32x1xi32, #blocked>
    %101 = arith.cmpi slt, %36, %100 : tensor<32x1xi32, #blocked>
    %102 = tt.broadcast %101 : tensor<32x1xi1, #blocked> -> tensor<32x32xi1, #blocked>
    %103 = triton_gpu.memdesc_subview %50[%c2_i32, %c0_i32, %c0_i32] : !tt.memdesc<3x32x32xf16, #shared, mutable> -> !tt.memdesc<32x32
    %104 = tt.splat %88 : i1 -> tensor<32x32xi1, #blocked>
    %105 = arith.andi %104, %102 : tensor<32x32xi1, #blocked>
    %106 = triton_gpu.async_copy_global_to_local %90, %103 mask %105 other %cst_1 : tensor<32x32x!tt.ptr<f16>, #blocked> -> <32x32xf16
    %107 = triton_gpu.async_commit_group %106
    %108 = triton_gpu.async_wait %67 {num = 4 : i32}
    %109 = triton_gpu.memdesc_subview %55[%c0_i32, %c0_i32] : !tt.memdesc<128x32xf16, #shared, mutable> -> !tt.memdesc<128x16xf16, #sh
```

Async memory operations for TritonGPU IR

triton_gpu.async_commit_group

triton_gpu.async_copy_global_to_local

triton_gpu.async_wait

Will likely require copy queues and "DMA" units to asynchronously copy elements between Global memory and Shared memory

Adding compiler support

(TritionGPU* IR +Tensor + Arith, other MLIR dialects)->

(SPIRV + Vortex Instrics)->

(RISC V assembly)

PyTorch

triton requires tensors to be located on "device"

a Pytorch backend for vortex would be necessary

Only memory operations support is necessary