Triton CPU backend

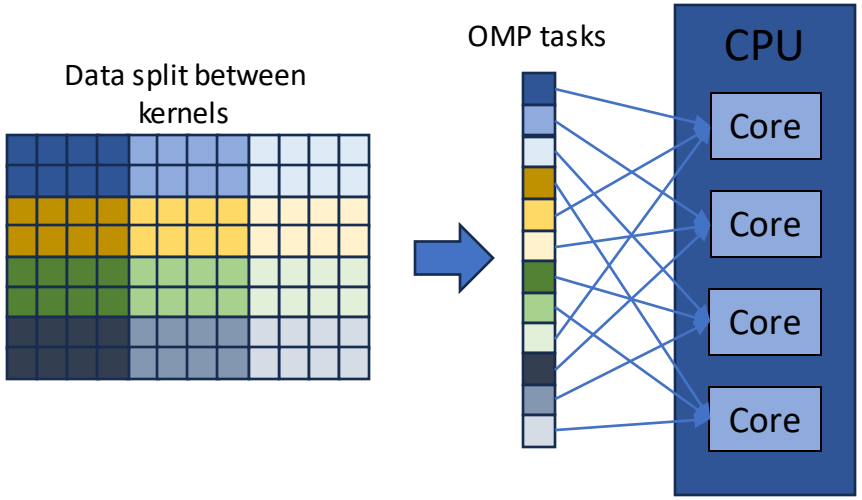
Goals and non-goals

- Goals
  - Portable, easy-to-use solution
    - Users don’t need to modify their kernels
    - Parameters tuning may be required to achieve the best performance
  - Provide performance on par with Inductor’s generated code on torchbench
  - Support multiple architectures
- Non-goals
  - Provide debugging functionality
    - Interpreter mode is used for that
  - Efficient GEMM lowering (at least short-term)
    - Use external solutions

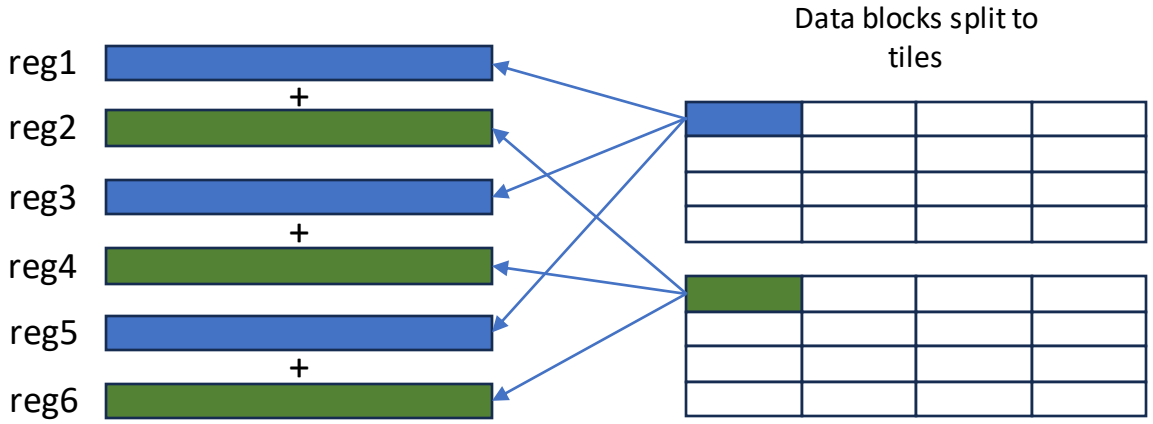
Threading and SIMD parallelism

```
triton_kernel[grid](*args)
```

```
#pragma omp for [schedule(static, N)]
for (auto [x, y, z] : grid) {
  triton_kernel[grid](args..., {x, y, z})
}
```




```
tt.Func public @triton_kernel(...) {
  scf.for (%i) = (%start) to (%end) step (%tile_size) {
    %tile_0 = vector.load %arg0[%i]
    %tile_1 = vector.load %arg1[%i]
    %tmp = arith.addf %tile_0, %tile_1
    vector.store %tmp, %arg2[%i]
  }
}
```



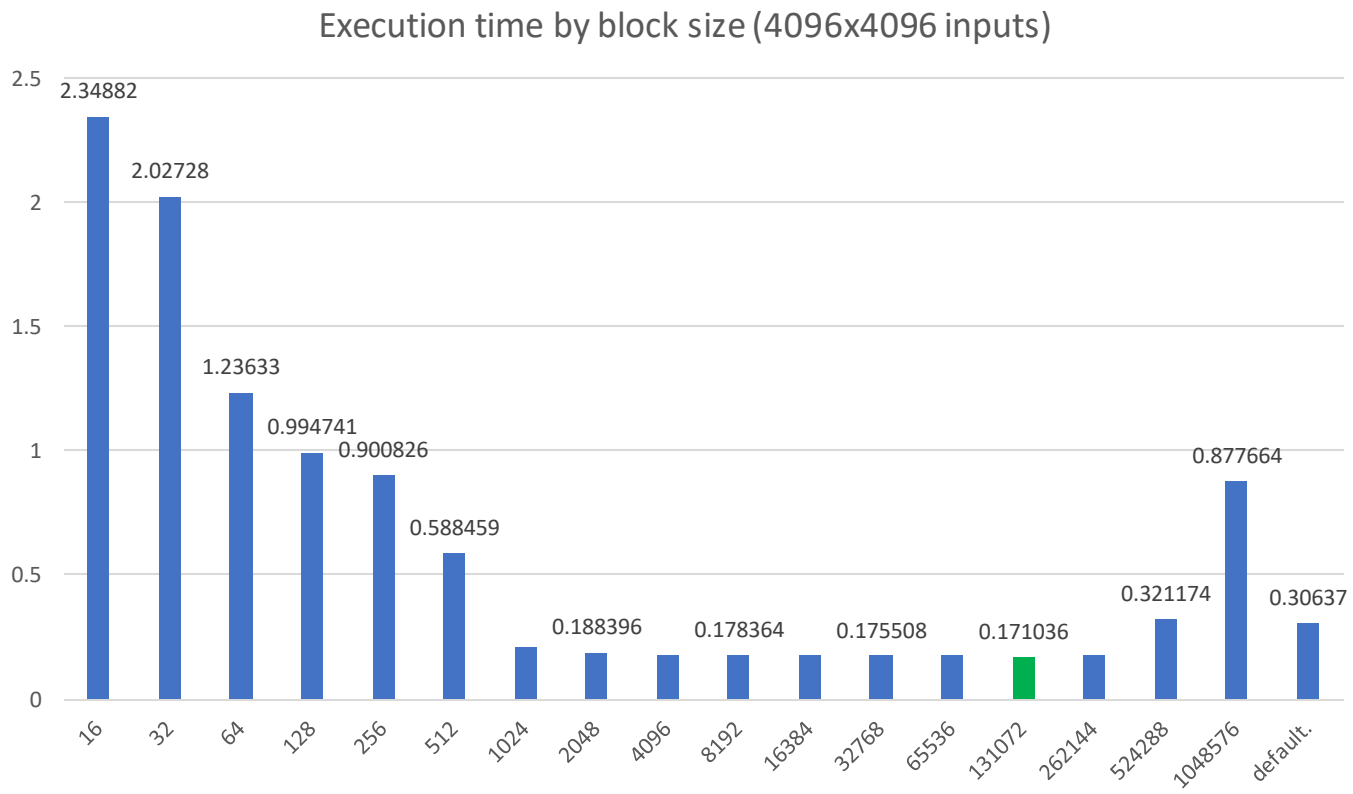
Block size != Tile size != Vector size

- Efficient tile size is defined by platform capabilities
  - Registers count and sizes
  - Specialized ISA (e.g. VNNI)
  - Tile is usually one or several native vector registers
- Efficient block size can far exceed tile
  - Thread task spawning has its cost
  - Data locality and prefetch are better for bigger blocks
  - Block size can be defined by input size
    - Reductions in the kernel, e.g. softmax

Block size. Vector-add example

```
@triton.jit
def kernel(a, b, c, BLOCK : tl.constexpr):
  index = tl.arange(0, BLOCK)
  tmp0 = tl.load(a + index)
  tmp1 = tl.load(b + index)
  tl.store(c + index, tmp0 + tmp1)
```

```
#pragma omp for [schedule(static, N)]
for(long i0 = 0; i0 < SIZE; i0 += 16) {
 for (long i1 = 0; i1 < 16; ++i1) {
  out_ptr0[i0 + i1] = in_ptr0[i0 + i1] + in_ptr1[i0 + i1];
 } // Translated into a single vector add instruction
}
```



Required transformations

- Memory accesses
  - Move from indirect accesses to contiguous ones
  - Can use memref and/or block pointers
- Tiling and fusion
  - To minimize operations with memory tensor operations needs to be tiled and fused
- Vectorization
  - Prefer vectorization in MLIR to LLVM auto-vectorizer

Memory access analysis

- Avoiding gathers/scatters whenever possible is crucial for performance
- Existing alignment analysis pass might be used/extended for Triton CPU
- Can also transform tensors of pointers to block pointers on Triton dialect
  - Can benefit GPU backends
- How to represent masked load/store?
  - Vector dialect
  - Triton dialect loads/stores on memrefs
  - New ops (TritonCPU dialect)

Tiling and fusion options

- Use MLIR upstream transformations vs own passes
- There are existing upstream transformations to lower ops into tiled scf.for/scf.parallel loop
  - Transformation is based on TilingInterface and PartialReductionOpInterface
  - Linalg operations implement those interfaces
- Two ways to re-use transfromations
  - Translate Triton ops to linalg dialect
  - Implement tiling interface for Triton ops
- We can write our own transformations working on Triton dialects

Which dialects to use?

- Use existing Triton dialect(s) when possible
- Prefer upstream dialects when it fits?
  - Linalg
  - Vector
  - Memref
- Use new dialect (TritonCPU) when it helps with analysis and transformations

Linalg on tensors as a midlayer dialect

- Upstream dialect used in ML/AI compilation flows, provides useful transformations
  - Tiling
  - Fusions
  - Vectorization
  - Bufferization
  - …
- Can simplify lowering to LLVM IR
- Triton ops on tensors map nicely to linalg ops

Vectors as a midlayer dialect

- Provide masked operations
- Closer to hardware capabilities, can express algorithms more precisely
- Vector dialect doesn’t provide generic forms for scans and reductions
  - Vector versions can be quite complex
  - It’s better to perform tiling before moving to vectors
- Vectorization is not always profitable – need a way to fall back to a scalar code

Using OpenMP threading runtime

Pros:

- Easy-to-use portable solution
- Accessible from MLIR through omp dialect
- Provides composability to avoid oversubscriptions in multiple scenarios
  - Run multiple kernels
  - Parallelize kernels
  - Run external parallel kernels (e.g. GEMM)

Cons:

- Introduces additional dependency

External dependencies

- Math functions
  - Would need scalar and vector variants (VML, DNN)
- GEMM and other kernels
  - MKL/DNN/XSMM
- Bytecode libraries vs native binaries
  - Might want to use vendor-specific libraries, so better not having all runtimes in Triton repo