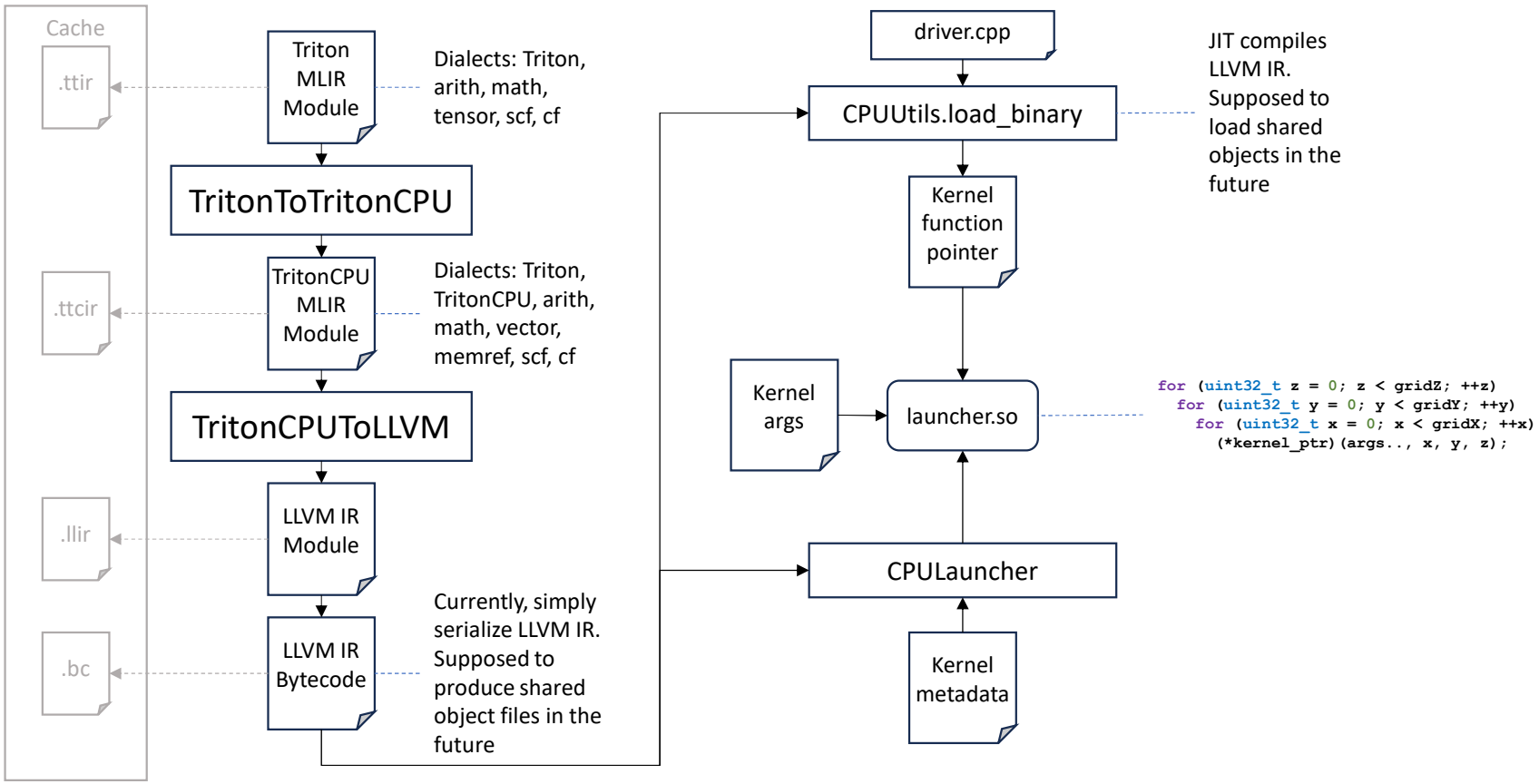
TritonCPU update

May 16, 2024

TritonCPU repo

- https://github.com/triton-lang/triton-cpu
- Main branch introduces CPU backend but still has no full compilation pipeline and runtime infrastructure
- Pull request to add basic lowering and execution flow is on review
  - https://github.com/triton-lang/triton-cpu/pull/2

Kernel compilation and execution flow



TirtonToTritonCPU

- Transition from tensors to vectors
- Loads/stores with block pointers are replaced with vector reads/writes
  - Block pointer is split into memref and indices for that
  - Masks are not supported yet
  - Boundary checks are supported
- Loads/stores with tensors of pointers are scalarized
  - Masks are supported
  - No pointer analysis yet to produce vector loads/stores

TirtonToTritonCPU (example)

```
@triton.jit
def kernel(in1_ptr, in2_ptr, out_ptr, BLOCK_SIZE: tl.constexpr):
    id = tl.program_id(0)
    offs = id * BLOCK_SIZE
    in1_block_ptr = tl.make_block_ptr(base=in1_ptr, shape=(BLOCK_SIZE, ), strides=(1, ),
                                      offsets=(offs, ), block_shape=(BLOCK_SIZE, ), order=(0, ))
    in2_block_ptr = tl.make_block_ptr(base=in2_ptr, shape=(BLOCK_SIZE, ), strides=(1, ),
                                      offsets=(offs, ), block_shape=(BLOCK_SIZE, ), order=(0, ))
    out_block_ptr = tl.make_block_ptr(base=out_ptr, shape=(BLOCK_SIZE, ), strides=(1, ),
                                     offsets=(offs, ), block_shape=(BLOCK_SIZE, ), order=(0, ))
    val1 = tl.load(in1_block_ptr, boundary_check=())
    val2 = tl.load(in2_block_ptr, boundary_check=())
    val = val1 + val2
    tl.store(out_block_ptr, val, boundary_check=())
```

```
module {
  tt.func public @kernel(%arg0: !tt.ptr<f32>, %arg1: !tt.ptr<f32>, %arg2: !tt.ptr<f32>) attributes {noinline = false} {
    %c1_i64 = arith.constant 1 : i64
    %c128_i64 = arith.constant 128 : i64
    %c128_i32 = arith.constant 128 : i32
    %0 = tt.get_program_id x : i32
    %1 = arith.muli %0, %c128_i32 : i32
    %2 = tt.make_tensor_ptr %arg0, [%c128_i64], [%c1_i64], [%1] {order = array<i32: 0>} : <tensor<128xf32>>
    %3 = tt.make_tensor_ptr %arg1, [%c128_i64], [%c1_i64], [%1] {order = array<i32: 0>} : <tensor<128xf32>>
    %4 = tt.make_tensor_ptr %arg2, [%c128_i64], [%c1_i64], [%1] {order = array<i32: 0>} : <tensor<128xf32>>
    %5 = tt.load %2 : !tt.ptr<tensor<128xf32>>
    %6 = tt.load %3 : !tt.ptr<tensor<128xf32>>
    %7 = arith.addf %5, %6 : tensor<128xf32>
    tt.store %4, %7 : !tt.ptr<tensor<128xf32>>
    tt.return
  }
}
```

TirtonToTritonCPU (example, TTCIR)

```
module {
  tt.func public @kernel(%arg0: !tt.ptr<f32>, %arg1: !tt.ptr<f32>, %arg2: !tt.ptr<f32>) attributes {noinline = false} {
    %cst = arith.constant 0.000000e+00 : f32
    %c1_i64 = arith.constant 1 : i64
    %c128_i64 = arith.constant 128 : i64
    %c128_i32 = arith.constant 128 : i32
    %0 = tt.get_program_id x : i32
    %1 = arith.muli %0, %c128_i32 : i32
    %2 = tt.make_tensor_ptr %arg0, [%c128_i64], [%c1_i64], [%1] {order = array<i32: 0>} : <tensor<128xf32>>
    %3 = triton_cpu.extract_memref %2 : <tensor<128xf32>> -> memref<?xf32>
    %4 = tt.make_tensor_ptr %arg1, [%c128_i64], [%c1_i64], [%1] {order = array<i32: 0>} : <tensor<128xf32>>
    %5 = triton_cpu.extract_memref %4 : <tensor<128xf32>> -> memref<?xf32>
    %6 = tt.make_tensor_ptr %arg2, [%c128_i64], [%c1_i64], [%1] {order = array<i32: 0>} : <tensor<128xf32>>
    %7 = triton_cpu.extract_memref %6 : <tensor<128xf32>> -> memref<?xf32>
    %8 = triton_cpu.extract_indices %2 : <tensor<128xf32>> -> index
    %9 = vector.transfer_read %3[%8], %cst {in_bounds = [true]} : memref<?xf32>, vector<128xf32>
    %10 = triton_cpu.extract_indices %4 : <tensor<128xf32>> -> index
    %11 = vector.transfer_read %5[%10], %cst {in_bounds = [true]} : memref<?xf32>, vector<128xf32>
    %12 = arith.addf %9, %11 : vector<128xf32>
    %13 = triton_cpu.extract_indices %6 : <tensor<128xf32>> -> index
    vector.transfer_write %12, %7[%13] {in_bounds = [true]} : vector<128xf32>, memref<?xf32>
    tt.return
  }
}
```

TritonCPUToLLVM

- Lower the rest of Triton ops
  - Function ops: Func, Call, Return
    - Similar to TritonGPU but with no stack pointer
    - Add program IDs as kernel arguments
  - Pointer ops: MakeTensorPtr, Advance, IntToPtr, PtrToInt
  - Scalar loads/stores
- Lower TritonCPU ops
  - Transfer data from block pointer structures to memref ones
- Convert math operations to math lib calls
- Lower the rest using upstream passes for conversion to LLVM dialect

TritonCPUToLLVM (example, LLVM dialect)

```
module {
  llvm.func @kernel(%arg0: !llvm.ptr, %arg1: !llvm.ptr, %arg2: !llvm.ptr, %arg3: i32, %arg4: i32, %arg5: i32) {
    %0 = llvm.mlir.constant(128 : i32) : i32
    %1 = llvm.mlir.constant(1 : i64) : i64
    %2 = llvm.mul %arg3, %0 : i32
    %3 = llvm.zext %2 : i32 to i64
    %4 = llvm.mul %3, %1 : i64
    %5 = llvm.getelementptr %arg0[%4] : (!llvm.ptr, i64) -> !llvm.ptr, f32
    %6 = llvm.load %5 {alignment = 4 : i64} : !llvm.ptr -> vector<128xf32>
    %7 = llvm.getelementptr %arg1[%4] : (!llvm.ptr, i64) -> !llvm.ptr, f32
    %8 = llvm.load %7 {alignment = 4 : i64} : !llvm.ptr -> vector<128xf32>
    %9 = llvm.fadd %6, %8  : vector<128xf32>
    %10 = llvm.getelementptr %arg2[%4] : (!llvm.ptr, i64) -> !llvm.ptr, f32
    llvm.store %9, %10 {alignment = 4 : i64} : vector<128xf32>, !llvm.ptr
    llvm.return
  }
}
```

TritonCPUToLLVM (example, LLVM IR)

```
define void @kernel(ptr nocapture readonly %0, ptr nocapture readonly %1,
                    ptr nocapture writeonly %2, i32 %3, i32 %4, i32 %5) local_unnamed_addr #0 {
  %7 = shl i32 %3, 7
  %8 = zext i32 %7 to i64
  %9 = getelementptr float, ptr %0, i64 %8
  %10 = load <128 x float>, ptr %9, align 4
  %11 = getelementptr float, ptr %1, i64 %8
  %12 = load <128 x float>, ptr %11, align 4
  %13 = fadd <128 x float> %10, %12
  %14 = getelementptr float, ptr %2, i64 %8
  store <128 x float> %13, ptr %14, align 4
  ret void
}
```

LLVM IR Compilation (example, ASM)

```
        .globl  kernel
        .p2align        4, 0x90
        .type   kernel,@function
kernel:
.Lfunc_begin0:
        .cfi_sections .debug_frame
        .cfi_startproc
        shll    $7, %ecx
        vmovups 384(%rdi,%rcx,4), %zmm0
        vmovups 448(%rdi,%rcx,4), %zmm1
        vmovups 256(%rdi,%rcx,4), %zmm2
        vmovups 320(%rdi,%rcx,4), %zmm3
        vmovups (%rdi,%rcx,4), %zmm4
        vmovups 64(%rdi,%rcx,4), %zmm5
        vmovups 128(%rdi,%rcx,4), %zmm6
        vmovups 192(%rdi,%rcx,4), %zmm7
        vaddps  64(%rsi,%rcx,4), %zmm5, %zmm5
        vaddps  (%rsi,%rcx,4), %zmm4, %zmm4
        vaddps  192(%rsi,%rcx,4), %zmm7, %zmm7
        vaddps  128(%rsi,%rcx,4), %zmm6, %zmm6
        vaddps  320(%rsi,%rcx,4), %zmm3, %zmm3
        vaddps  256(%rsi,%rcx,4), %zmm2, %zmm2
        vaddps  448(%rsi,%rcx,4), %zmm1, %zmm1
        vaddps  384(%rsi,%rcx,4), %zmm0, %zmm0
        vmovups %zmm0, 384(%rdx,%rcx,4)
        vmovups %zmm1, 448(%rdx,%rcx,4)
        vmovups %zmm2, 256(%rdx,%rcx,4)
        vmovups %zmm3, 320(%rdx,%rcx,4)
        vmovups %zmm6, 128(%rdx,%rcx,4)
        vmovups %zmm7, 192(%rdx,%rcx,4)
        vmovups %zmm4, (%rdx,%rcx,4)
        vmovups %zmm5, 64(%rdx,%rcx,4)
        vzeroupper
        retq
```

Current prototype

- https://github.com/triton-lang/triton-cpu/pull/2
- Supported operations:
  - Triton: GetProgramId, MakeRange, Splat, AddPtr, PtrToInt, IntToPtrOp, MakeTensorPtr, Advance, Load, Store, Bitcast, Broadcast, ExpandDims, PreciseDivF, PreciseSqrt, Reshape
  - Arith: basic set of arithmetic operations, comparison, conversions, etc.
  - Math: basic math functions (exp, log, sin, cos, etc.)
    - Lowered to scalar libm calls
- Synchronous, single-threaded kernels execution so far
- Passes ~3500 tests from core_test.py