Code Generation in MLIR

Harsh Menon (nod.ai)
Overview

• Motivation
• Code generation in LLVM/MLIR
• Dialects (Linalg, Vector, GPU, etc.)
• Walkthrough of code generation using IREE / SHARK
  • CPU and GPU code generation
  • Targeting custom accelerators
  • Auto-tuning
• Conclusion
• Acknowledgements
• References
Motivation

- Deep learning has become extremely pervasive spanning domains ranging from autonomous cars, natural language processing to medical imaging

- Large models are achieving state of the art results (such as transformers in NLP)

- Unfortunately, these large models take many months and millions of dollars to train on existing hardware

- On edge-based systems, inference dominates with latency being a key metric

- New hardware vendors have risen to the occasion with custom accelerators that address some of these concerns

- But as the number of models and hardware combinations explode, need a strong compiler infrastructure that can provide performance gains and is easily re-targetable to new hardware
Motivation

• LLVM provides a strong compiler infrastructure that already scales to various hardware targets and can be used to fill the gap

• But LLVM IR is too low-level and many opportunities for optimization are missed if we start at that level of abstraction

• MLIR provides compiler infrastructure to handle varying levels of abstraction and provides a way to progressively lower to LLVM IR, leveraging best of both worlds

• In machine learning (ML), neural networks are defined in Python-based frameworks such as Tensorflow, PyTorch and JAX

• MLIR helps progressively lower computation graphs from their pythonic representation to LLVM IR
Code generation in LLVM

• Final phase in compiler pipeline

• Must make effective use of available resources while preserving program semantics

• Deals with problems such as instruction selection, register allocation and assignment, instruction ordering

• LLVM has several different backends such as X86, NVPTX, RISC-V etc. which contain the specific hardware definition (in terms of hardware instructions, registers etc. expressed in tablegen)

• During code generation pipeline, LLVM IR is lowered to the SelectionDAG, then MachineInstr, MCInst and finally compiled to bitcode

• MLIR leverages LLVM for this part of code generation

• Code generation in MLIR operates at a higher-level of abstraction and attempts to provide the missing infrastructure between high level ML programs and LLVM IR
Code generation in MLIR

- Figure on left shows dialect ecosystem in MLIR

- Starting with dialects on top (that closely follow the native ML frameworks), there are many paths to LLVM dialect

- Each dialect progressively lowers the abstraction going from tensors (immutable SSA) to memrefs

- Perform high-level optimizations at upper dialects, more hardware-specific optimizations as we approach LLVM dialect

- Once we arrive at LLVM dialect, we can translate to LLVM IR using mlir-translate and using LLVM to generate the final binary
Linalg Dialect

- Linalg dialect is used to represent perfectly nested loop computations making it easy to perform transformations like fusion, tiling, and loop interchange.

- Operates on both tensors and memrefs.

- Can be lowered to loops or affine expressions with computation in loop body.

- Incorporates learnings from Halide, TVM, Tensor Comprehensions, XLA etc.

- Linalg defines a small set of core named ops (such as matmul, conv, pooling etc.) that the front-end dialects can lower to.

- This maps the large set of operations to a smaller set of operations that the compiler can focus on optimizing.

- The core workhorse of this dialect is the linalg.generic op.
Linalg GenericOp

- **Indexing Maps**
  - Capture data access patterns for each operand
  - Domain represents point in iteration space, range represents a point in the operand's data space

- **Iterator Types**
  - Specifies data dependence between iterations of the loop

- **Inputs and Outputs**
  - Tied output and result operands

- **Compute Payload**
  - Computation performed at each point in the iteration space
  - Yielded value is result of computation
  - Arguments are obtained from operands using indexing maps

```cpp
%5 = linalg.generic {
  indexing_maps = 
    [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3),
      affine_map<(d0, d1, d2, d3) -> (d0, d1))],
  iterator_types = ["parallel", "parallel", "reduction", "reduction"]
  ^bb0(%arg1 : f32, %arg2 : f32):
    %17 = arith.addf %arg2, %arg1 : f32
    linalg.yield %17 : f32
} -> tensor<?x?f32>

for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < K; k++) {
      for (int l = 0; l < P; l++) {
        c[i][j] = c[i][j] + a[i][j][k][l];
      }
    }
  }
}
```
Vector Dialect

- Provides generic retargetable n-D vector abstractions
- Operations in vector dialect can progressively decompose to lower rank variants
- Can lower to LLVM instructions or directly target hardware intrinsics (mma_compute)
- Examples:
  - vector.transfer_read/write: bridge the gap between memory and vectors and contain enough information to encode read/write patterns such as broadcasted, permuted and masked accesses
  - vector.outerproduct: outer product operation (typically obtained from lowering of matmul) that can further be lowered to llvm fused-multiply add (FMA) instructions
GPU Dialect

- Provides abstractions for retargetable GPU model
- Contains operations common to SIMT platforms
- Examples
  - Communication: gpu.all_reduce (reduction across a local workgroup)
  - Synchronization: gpu.barrier
  - Compute: gpu.subgroup_mma_compute
- Can be obtained by lowering from vector dialect
- Operates only on memrefs
Accelerator-Specific Dialects

- NVIDIA GPU specific dialects
  - NVVM dialect, NVGPU dialect
- AMD GPU specific dialects
  - ROCDL dialect
- ARM CPU specific dialects
  - ARM Neon dialect, ARM SVE dialect
- Intel CPU specific dialects
  - x86Vector dialect

Each of these dialects exposes specific hardware functionality present in those devices

- NVVM dialect has `nvvm.wmma.mma` for matrix multiplication using tensor cores, not present in other dialects
Code generation using IREE

- IREE is an open-source MLIR-based end-to-end compiler and runtime that lowers ML models for datacenter and edge workloads
- Supports X86, NVIDIA, AMD, RISC-V, Vulkan and ARM
- Supports Tensorflow, JAX, TFLite, PyTorch
- We will use IREE to demonstrate code generation using MLIR, specifically focusing on the compilation pipeline
Code generation using SHARK

- Builds on top of IREE pipeline
- Adds performance optimizations for CUDA
  - Caching allocator
  - Async memory prefetch
- Adds auto-tuning capabilities
- Adds backends for custom accelerators
- Contains a fully validated set of 100s of models
- Easy to deploy (integrated with triton server)
Perceptron Code Generation

• Perceptron is defined mathematically as shown below

\[ z = \max(0, XW) \]

• In ML frameworks, it is a matrix multiplication followed by a ReLU non-linearity

• The code on the top right shows how to define such a network in C++ and in Tensorflow on the bottom right

• By not having to start with a loop-based definition, we avoid any raising and perform optimizations on a higher level of abstraction than possible with other languages such as C++

• This results in reduced complexity during fusion, tiling and vectorization
Lowering to Linalg on Tensors

- TF is lowered to MHLO and then to the linalg dialect
- Both matrix multiplication and the ReLU are lowered to linalg.generic ops
- Important to note that linalg.matmul is

\[ C_{i,j} = C_{i,j} + A_{i,k}B_{k,j} \]

- Hence, we need a linalg.fill before the linalg.matmul

```cpp
#define affine_map<(d0, d1) -> (d0, d1)>

module {
  iree_input.global private @VV = dense<1.000000e+00> : tensor<1024x1024xf32>
  ...
  func.func private @__inference_predict_140 (arg0 : tensor<64x1024xf32>) {
    %cst = arith.constant 0.000000e+00 : f32
    ...
    %2 = linalg.init_tensor [64, 1024] : tensor<64x1024xf32>
    %3 = linalg.fill ins(%cst : f32) outs(%2 : tensor<64x1024xf32>) -> tensor<64x1024xf32>
    %4 = linalg.matmul ins(%arg0, %1 : tensor<64x1024xf32>, tensor<1024x1024xf32>)
         outs(%3 : tensor<64x1024xf32>) -> tensor<64x1024xf32>
    %5 = linalg.init_tensor [64, 1024] : tensor<64x1024xf32>
    %6 = linalg.generic (indexing_maps = [#map, #map], iterator_types = ["parallel", "parallel"])
         ins(%4 : tensor<64x1024xf32>) outs(%5 : tensor<64x1024xf32>) {
          ^bb0(%arg1 : f32, %arg2 : f32):
            %7 = arith.maxf %arg1, %cst : f32
            linalg.yield %7 : f32
          } -> tensor<64x1024xf32>
    return %6 : tensor<64x1024xf32>
  }
}```
Overview of CPU code generation pipeline

- Split computation graph into smaller sub-graphs (dispatch regions)
- Tiling and vectorization key pieces of CPU pipeline
- CPU pipeline performs bufferization late, preferring in-place bufferization while taking care to avoid RaW conflicts
- Finally operations are lowered to make mapping to hardware instructions efficient (such as lowering matrix multiplication to outer products to easily map to FMA instructions)
- Most of this pipeline shared with GPU
Dispatch Region Formation

- Dispatch region contains computation that has to be executed on device in an atomic fashion
- Large neural networks are partitioned into a finite number of dispatch regions
- Each dispatch region contains a root op (any linalg named op or generic op with reduction iterator type)
- Root ops are then fused with consumers if all uses of producer are dominated by it
- Also does elementwise fusion to fuse linalg.generic ops

```python
%1 = flow.dispatch.workgroups(%c64, %c1024) : (tensor<64x1024xf32>, %arg2) -&gt; tensor<64x1024xf32>
  %arg1 = flow.dispatch.tensor<readonly:64x1024xf32>, %arg2:
  
  %2 = flow.dispatch.tensor<writeonly:64x1024xf32> (
    %3 = flow.dispatch.tensor<load %arg1, offsets = [0, 0], sizes = [64, 1024], strides = [1, 1] :
      tensor<64x1024xf32>,
    %4 = linalg.init_tensor [64, 1024] : tensor<64x1024xf32>
  
  %5 = linalg.fill ins(%cst : f32) outs(%4 : tensor<64x1024xf32>) -&gt; tensor<64x1024xf32>
  %6 = linalg.matmul ins(%3, %cst_0 : tensor<64x1024xf32>, tensor<1024x1024xf32>)
    outs(%5 : tensor<64x1024xf32>) -&gt; tensor<64x1024xf32>
  %7 = linalg.generic (indexing_maps = [affine_map<d0, d1> -&gt; (d0, d1)],
    affine_map<d0, d1> -&gt; (d0, d1)),
    iterator_types = ["parallel", "parallel"]
  ins(%6 : tensor<64x1024xf32>)
    outs(%4 : tensor<64x1024xf32>)
  %8 = arith.maxf %arg3, %cst : f32
  linalg.yield %8 : f32
  ) -&gt; tensor<64x1024xf32>
  flow.dispatch.tensor<store %7, %arg2, offsets = [0, 0], sizes = [64, 1024], strides = [1, 1] :
    tensor<64x1024xf32> -&gt; tensor<writeonly:64x1024xf32>
  flow.return
  ) count(%arg1: index, %arg2: index) -&gt; (index, index, index) {
    %x, %y, %z = flow.dispatch.default_workgroup_count %arg1, %arg2
  flow.return %x, %y, %z : index, index, index
  }
```
Tile and Distribute to Workgroups

- Work partitioned along a 3-D grid of virtual processors (workgroups) that can be mapped to multi-core CPUs or GPUs
- Uses a block cyclic distribution to distribute the tiles
- Each dispatch region performs a tile of the computation
- The compute done is determined by its rank (workgroup_id_x, workgroup_id_y) and number of processors
- Only parallel dimensions are tiled
Tile and Distribute to Workgroups

```c
func @predict_dispatch_00 {
  %workgroup_id_x = halinterface.workgroup.id[0] : index
  %workgroup_count_x = halinterface.workgroup.count[0] : index
  %workgroup_id_y = halinterface.workgroup.id[1] : index
  %workgroup_count_y = halinterface.workgroup.count[1] : index
  %2 = affine.apply affine_map<0:s0> -> (s0 * 32)>0:0(%workgroup_id_y)
  %3 = affine.apply affine_map<0:s0> -> (s0 * 32)>0:0(%workgroup_count_y)
  scf.for %arg0 = %2 to %c64 step %3 {
    %4 = affine.apply affine_map<0:s0> -> (s0 * 32)>0:0(%workgroup_id_x)
    %5 = affine.apply affine_map<0:s0> -> (s0 * 32)>0:0(%workgroup_count_x)
    scf.for %arg1 = %4 to %c1024 step %5 {
      %8 = linalg.fill ins(%cst_0 : f32) outs(%6 : tensor<32x32xf32>) -> tensor<32x32xf32>
      %9 = linalg.matmul ins(%7, %cst : tensor<32x1024xf32>, tensor<1024x32xf32>)
      outs(%8 : tensor<32x32xf32>) -> tensor<32x32xf32>
      %10 = linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>],
            iterator_types = ["parallel", "parallel"]} outs(%9 : tensor<32x32xf32>) {
        ^bb0(%arg2 : f32):
          %11 = arith.maxf %arg2, %cst_0 : f32
          linalg.yield %11 : f32
        } -> tensor<32x32xf32>
      ...
    }
  }
}
```
Tile and Fuse

- Additional tiling along parallel dimensions

- Introduces subset operations (tensor.extract_slice, tensor.insert_slice) to access the tiled data

- Depending on chosen tile sizes, there may not exists a single static tensor type valid for every iteration

- The sub-tensor may be relaxed to a dynamic tensor

- Subsequent canonicalizations can be used to refine any shapes that are determined to be static

- Can use padding to handle these scenarios by introducing tensor.pad operations with appropriate sizes

```python
@predict_dispatch_0()
{ scf for %arg0 = %2 to %64 step %3 ( scf for %arg1 = %4 to %1024 step %5 ( %8 = scf for %arg2 = %0 to %32 step %cB iter_args(%arg3 = %7) -> (tensor<32x128xf32>) ( %9 = tensor.extract_slice %6(%arg2, 0, [8, 1024], [1, 1]) : tensor<32x1024xf32> to tensor<8x1024xf32> ) %10 = scf for %arg4 = %0 to %128 step %c32 iter_args(%arg5 = %arg3) -> (tensor<32x128xf32>) ( %11 = tensor.extract_slice %arg5(%arg2, %arg4, [8, 32], [1, 1]) : tensor<32x128xf32> to tensor<8x32xf32> ) %12 = linalg.fill_ins(%cst_0: f32) outs(%11) : tensor<8x32xf32> ) %13 = linalg.matmul_ins(%8, %cst : tensor<8x1024xf32>, tensor<1024x32xf32>) outs(%12) : tensor<8x32xf32> ) -> tensor<8x32xf32> ) %14 = linalg.generic outs(%13) : tensor<8x32xf32> ) ( ^bb0(%arg6, f32): %16 = arith.maxf %arg6, %cst_0 : f32 linalg.yield %16 : f32 ) -> tensor<8x32xf32> ) %15 = tensor.insert_slice %14 into %arg5(%arg2, %arg4, [8, 32], [1, 1]) : tensor<32x128xf32> into tensor<32x128xf32> ) scf.yield %15 : tensor<32x128xf32> ) scf.yield %10 : tensor<32x128xf32> )
}
```
Single Tiling Expert

- Multiple different ways of tiling the matmul
- DoubleTilingExpert, TripleTilingExpert, DoubleTilingPadExpert etc.
- Using SingleTilingExpert get a single tiling of reduction dimension
- Changes K dimension from 1024 to 16

```c
func @predict_dispatch_00 () {
  scf for %arg0 = %2 to %64 step %3 {
    scf for %arg1 = %4 to %1024 step %5 {
      %8 = scf for %arg2 = %0 to %32 step %8 iter_args(%arg3 = %7) -> (tensor<32x128xf32>) {
        %9 = scf for %arg4 = %0 to %128 step %32 iter_args(%arg5 = %arg3) -> (tensor<32x128xf32>) {
          %10 = tensor.extract_slice %arg5[8, %arg4] [8, 32, 1, 1] : tensor<32x128xf32>
          to tensor<8x32x32f32>
          %11 = linalg.fill ins(%cst_0 : f32) outs(%10 : tensor<8x32x32f32>) -> tensor<8x32x32f32>
        }
      }
    }
  }
  %12 = scf for %arg6 = %0 to %1024 step %16 iter_args(%arg7 = %11) -> (tensor<8x32x32f32>) {
    %15 = tensor.extract_slice %6[8, %arg6] [8, 16, 1, 1] : tensor<32x1024xf32> to tensor<8x16x32f32>
    %16 = linalg.matmul ins(%15, %cst : tensor<8x16x32f32>, tensor<16x32x32f32>)
    outs(%arg7 : tensor<8x32x32f32>) -> tensor<8x32x32f32>
    scf yield %16 : tensor<8x32x32f32>
  }
  %13 = linalg.generic ()
  %14 = tensor.insert_slice %13 into %arg5[%arg2, %arg4] [8, 32, 1, 1] : tensor<8x32x32f32>
          into tensor<32x128xf32>
  scf yield %14 : tensor<32x128xf32>
  }
  scf yield %9 : tensor<32x128xf32>
  }
  }
```
Vectorize

- Emits vector.transfer_read/write operations for each operand
- For elementwise operations, rewrite as pointwise vector variant
- For reductions, rewrite as vector.contract or multi-reduction
- Broadcasting lower dimensional operands is done by vector.broadcast
- Permutations are handled by vector.transpose
- Lowers n-D vectors to 1-D vectors supported by LLVM
Bufferize

- Allocate and copy as little memory as possible
- Always prefer re-using buffers in place
- Use destination-passing style as a heuristic for in-place bufferization
- Tie output tensor to results tensor to act as bufferization constraint
- Performs a future in-place bufferization analysis of the operands and checks if a Raw conflict is detected
- If not, then performs in-place bufferization

```c
func @predict_dispatch_00 {
  %7 = vector.transfer_read %0[0, 0], %cst (in_bounds = [true, true]) : memref<16x32xf32>, vector<16x32xf32>
  scf.for %arg0 = %3 to %64 step %4 {
    %8 = memref.subview %0[1, 0] [32, 1024] (1, 1) : memref<64x1024xf32> to memref<32x1024xf32, affine_map<d0, d1>s0 : (d0 * 1024 + s0 + d1)>
    scf.for %arg1 = %5 to %1024 step %6 {
      %9 = memref.subview %0[1, 0] [32, 128] (1, 1) : memref<64x1024xf32> to memref<32x128xf32, affine_map<d0, d1>s0 : (d0 * 1024 + s0 + d1)>
      scf.for %arg2 = %0 to %512 step %8 {
        scf.for %arg3 = %0 to %128 step %32 {
          %10 = scf.for %arg4 = %0 to %1024 step %16 iter_args(%arg5 = %cst, 0) -> (vector<8x32xf32>) (vector<16xf32>, vector<8x16xf32>) into vector<8x32xf32>}
          %12 = vector.transfer_read %8[1, %arg2, %arg1], %cst (in_bounds = [true, true]) : memref<32x1024xf32, affine_map<d0, d1>s0 : (d0 * 1024 + s0 + d1)>, vector<16x32xf32>
          %13 = vector.contract (indexing_maps = [affine_map<d0, d1, d2>s0 : (d0, d2) -> (d0, d2)],
                                      affine_map<d0, d1, d2>s0 : (d0, d2) -> (d2, d1),
                                      affine_map<d0, d1, d2>s0 : (d0, d1) -> (d0, d1)),
          iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>)}
        %14 = arith.maxf %10, %cst, 0 : vector<8x32xf32>
        vector.transfer_write %11, %9[1, %arg2, %arg3] (in_bounds = [true, true]) : vector<8x32xf32>, memref<32x128xf32, affine_map<d0, d1>s0 : (d0 * 1024 + s0 + d1)>
      }
    }
  }
  scf.yield %13 : vector<8x32xf32>}
}``
Lowering closer to hardware

- Apply vector unrolling
  - Breaks down vector sizes to sizes well supported by target
  - Pre-emptively handles non power of 2 sizes to avoid suboptimal code generation
- vector.contract is lowered to outer products to enable mapping to SIMD FMA instructions
- Further lowered to LLVM dialect and translated to LLVM IR
Overview of GPU code generation pipeline

- Tiling and vectorization key pieces of GPU pipeline

- GPU pipeline bufferizes early and focuses on optimizing shared memory copies and reducing bank conflicts

- Plan to move bufferization after vectorization

- GPU vectorization focuses on using tensor cores efficiently and emitting appropriate nvvm intrinsics
Tile and Distribute to Warps

- Tile and distribute to warps
  - After bufferization, tile the reduction dimension
  - Copy subviews of input memrefs to shared memory (workgroup memory) on the GPU prior to computation
  - Start introducing SIMD characteristics by introducing memref.copy (mapping to shared memory done later downstream)
  - Insert barriers after copying to workgroup memory
  - Do an additional level of tiling to distribute to a warps
Tile and Distribute to Warps

```c
func @predict_dispatch_00 {
  ... scifor %arg0 = %c0 to %c1024 step %c16 {
  ... memref.copy %19, %1 : memref<32x16xf32, affine_map<(d0, d1) for s0 -> (d0 * 1024 + s0 + d1)>> to memref<32x16xf32, 3>
  ... memref.copy %20, %3 : memref<16x32xf32, affine_map<(d0, d1) for s0 -> (d0 * 32 + s0 + d1)>> to memref<16x32xf32, 3>
  ...
  ... linalg.matmul ins(%25, %26 : memref<16x16xf32, affine_map<(d0, d1) for s0 -> (d0 * 16 + s0 + d1)>, 3>,
                           memref<16x16xf32, affine_map<(d0, d1) for s0 -> (d0 * 32 + s0 + d1)>, 3>)
  ... outs(%27 : memref<16x16xf32, affine_map<(d0, d1) for s0 -> (d0 * 1024 + s0 + d1)>>)
}
...}
```
Multi-Buffering

- In order to hide latency, we can use double/multi-buffering to break dependencies between consecutive iterations of a loop using the same temporary buffer.
- Required for pipelining.
- Number of copies determined by desired stages of pipeline.
Multi-Buffering

```c
func @predict_dispatch_00 () {
  ...
  %0 = memref.alloc : memref<16x32xf32, 3>
  ...
  memref.copy %20, %0 : memref<16x32xf32, affine_map<(d0, d1):s0 -> (d0 * 32 + s0 + d1)>> to
    memref<16x32xf32, 3>
}

func @predict_dispatch_00 () {
  ...
  %2 = memref.alloc : memref<4x16x32xf32, 3>
  ...
  %22 = memref.subview %2[0:1, 0:1, 16, 32][1, 1, 1] : memref<4x16x32xf32, 3> to
    memref<16x32xf32, affine_map<(d0, d1):s0 -> (d0 * 32 + s0 + d1)>>, 3>
  ...
  memref.copy %23, %0 : memref<32x16xf32, affine_map<(d0, d1):s0 -> (d0 * 1024 + s0 + d1)>> to
    memref<32x16xf32, affine_map<(d0, d1):s0 -> (d0 * 16 + s0 + d1)>>, 3>
  ...
```
Vectorize Shared Memory Copies

- Vectorize the shared memory copy (converts to vector.transfer_read, vector.transfer_write)

- For optimal performance, always want to copy 128 bits. This can be used to determine copy tile size.

- Unroll the vector transfer read and writes

```cpp
%3 = memref.alloc() : memref<1x16x32x32, 3>
%4 = memref.alloc() : memref<1x32x16x32, 3>
...
linalg.fill ins(%cst : f32) outs(%4 : memref<4x16x32x32x3>)
scf.for %arg0 = %c0 to %c1024 step %c16 {
  ...
  %22 = vector.transfer_read %18(%20, %21, %cst [in_bounds = [true, true]] :
      memref<32x16x32, affine_map<d0, d1>[s0] -> (d0 * 1024 + s0 + d1)>, vector<1x4x32>)
  vector.transfer_write %22, %16(%20, %21) [in_bounds = [true, true]] : vector<1x4x32>,
      memref<32x16x32, affine_map<d0, d1>[s0] -> (d0 * 16 + s0 + d1)>, 3>
  ...
  %25 = vector.transfer_read %19(%23, %24, %cst [in_bounds = [true, true]] :
      memref<16x32x32, affine_map<d0, d1>[s0] -> (d0 * 32 + s0 + d1)>, vector<1x4x32>)
  vector.transfer_write %25, %17(%23, %24) [in_bounds = [true, true]] : vector<1x4x32>,
      memref<16x32x32, affine_map<d0, d1>[s0] -> (d0 * 32 + s0 + d1)>, 3>
gpubarrier
...
} linalg.matmul ins(%27, %28 : memref<16x16x32, affine_map<d0, d1>[s0] -> (d0 * 16 + s0 + d1)>, 3>,
      memref<16x16x32, affine_map<d0, d1>[s0] -> (d0 * 32 + s0 + d1)>, 3>
      outs(%29 : memref<16x16x32, >) )
}
linalg.generic (indexing_maps = [affine_map<d0, d1> -> (d0, d1)],
      iterator_types = ["parallel", "parallel"])
    outs(%4 : memref<16x16x32, affine_map<d0, d1>[s0] -> (d0 * 1024 + s0 + d1)>>) {
  ...
}
```
Reduce Bank Conflicts

- Shared memory is arranged in banks (32 banks each of width 4 bytes)
- Each thread in a warp can access shared memory in parallel
- When 2 or more threads in a warp access 4 byte words in the same bank, results in serialized accesses and hence a reduction in overall bandwidth
- Pad inner dimensions of allocOp to reduce chances of having bank conflicts (with 16 bytes)
- Plan to switch to shared memory swizzle for better efficiency
Tensor Core Vectorization

- Replace all linalg ops with vector equivalents
- Lower linalg.matmul to vector.contract with vector.transfer_read/write
- Lower linalg.generic to arith ops with vector.transfer_read/write
- Optimize vector transfers by removing redundant ops
- Vector transfer/contract size determined by tensor core supported sizes

```c
%16 = scf.for %arg0 = %c0 to %c1024 step %c16 iter_args(%arg1 = %cst) -> (vector<16x16xf32>, )
   (gpu.barrier)
%25 = vector.transfer_read %6, %24, %cst, _0 : memref<64x1024xf32>, vector<4x32>
vector.transfer_write %25, %40, %12, %cst, _0 : memref<4x32x20xf32, 3>
vector.transfer_write %28, %3, %29, %4, %cst, _0 : memref<4x32x36xf32, 3>
vector.transfer_write %28, %32, %arg1 : (vector<16x8xf32>, vector<8x16xf32>) into vector<16x16xf32>
scf.yield %43 : vector<16x16xf32>

%17 = arith.maxf %16, %cst : vector<16x16xf32>
```
Convert to GPU Dialect

- Convert copies to shared memory to async copies (nvgpu.device_async_copy, nvgpu.device_async_create_group, nvgpu.device_async_wait)
- Adds gpu.subgroup_mma_load_matrix which loads a matrix using all threads in a subgroup
- Lowers vector.contract to gpu.subgroup_mma_compute which performs matrix-multiply accumulate using all threads in the subgroup
- Lowers arith elementwise ops to gpu.subgroup_mma_elementwise

```llvm
%13 = scf.for %arg0 = %c0 to %c1024 step %c16 (iter_args(%arg1 = %c0) ->
  (lgpu.mma_matrix<16x16xf32, "COp">) {%
    gpu.barrier
    %20 = nvgpu.device_async_copy %7(%18, %171, %5(%19, %9, %10), 4 : memref<64x1024xf32> to memref<4x32x20xf32, 3>
    %22 = nvgpu.device_async_copy %6(%21, %12), %4(%19, %11, %121, 4 : memref<1024x32xf32> to memref<4x16x36xf32, 3>
    %23 = nvgpu.device_async_create_group %20, %22
    nvgpu.device_async_wait %23
    gpubarrier
    ...
    %25 = gpu.subgroup_mma_load_matrix %5(%19), %24, %c01 (leadDimension = 20 : index) :
      memref<4x32x20xf32, 3> -> lgpu.mma_matrix<16x8xf32, "AOp">
    %28 = gpu.subgroup_mma_load_matrix %4(%19, %c0, %271 (leadDimension = 36 : index) :
      memref<4x16x36xf32, 3> -> lgpu.mma_matrix<8x16xf32, "BOp">
    %31 = gpu.subgroup_mma_compute %26, %29, %30 :
      lgpu.mma_matrix<16x8xf32, "AOp">, lgpu.mma_matrix<8x16xf32, "BOp">
      -> lgpu.mma_matrix<16x16xf32, "COp">
    scf.yield %31 : lgpu.mma_matrix<16x16xf32, "COp">
  })
%14 = gpu.subgroup_mma_elementwise maxf %13, %0 :
  (lgpu.mma_matrix<16x16xf32, "COp">, lgpu.mma_matrix<16x16xf32, "COp">) ->
  lgpu.mma_matrix<16x16xf32, "COp">
  gpubarrier
  ...
%16 = gpu.subgroup_mma_store_matrix %14, %8(%15, %161 (leadDimension = 1024 : index) :
    lgpu.mma_matrix<16x16xf32, "COp">, memref<64x1024xf32>
  }
```

GPU Pipelining

- GPU Pipelining
  - Implement software pipelining using modulo scheduling
  - Operations of original loop body are overlapped so that there is a fixed initiation interval (II) between the start of consecutive loop iterations
  - Scheduling can be constrained by available hardware resource or loop carried dependencies
  - Emit a prologue, kernel and epilogue

```c
for (int i = 0; i < N; i++) {
}
```
GPU Pipelining

- **Prologue** contains async copies to shared memory
- **Kernel** contains mma load matrix and compute and ends with async copies
- **Epilogue** contains remaining computation
GPU Lowering

- Next, we use LLVM to lower to PTX
- Use JIT compilation to convert PTX code to native GPU machine code
- Alternatively, could use PTX assembler and convert to SASS (and then CUBIN)
Additional Targets

- Can reuse existing pipeline to target other GPUs that have tensor core equivalent functionality
- Adjust lowering from linalg to vector dialect
- Add appropriate dialect and lowering for custom accelerator
Targeting custom accelerators

- New RISC-V based many-core architectures (such as Hammerblade)
- Create a new dialect to model multiple processing elements (PE) and memory hierarchy
- Can leverage vector dialect or experimental RVV dialect for vector code generation
- Need to develop cost model to determine how to place kernels on tiles of PEs
- Could have special function units (tensor core equivalents) that can be leveraged during code generation for better performance
Auto-tuning

• How to determine optimal tile sizes?

• What about loop interchange? How much to unroll the loop?

• How does this extend from a single operator to entire neural network? On distributed heterogeneous resources?

• Many hyperparameters and a large search space make it difficult to generate good code

• Unclear how much performance is left on table for a given set of hyperparameters

• Formulate as search problem imposing constraints to reduce search space

• Can choose from a variety of search algorithms ranging from reinforcement learning (RL) to genetic algorithms
Auto-tuning

- RL Framework (Compiler Gym) that can be used for compiler optimization tasks
- Gradient-free methods (Nevergrad) have also been used to tune performance
- Current list of knobs shown in table below
- Beyond operator level, we can also search for how to best partition tensors for distributed computations (for inference and training)
- Approaches such as ALPA provide framework for how to partition tensors across clusters of heterogeneous resources
- Additional variables such as checkpointing for training

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiling</td>
<td>sizes - array of tile sizes, interchange - order of loops after tiling, pad - whether to pad partial tiles, pack_paddings - non-removable padding for arrays, hoist_paddings - number of loops from which to hoist padding for arrays, peel - loops from which to peel off partial tiles, scalarize_dyn_dims - whether to emit scalar code for non-vectorizable (dynamic) dimensions</td>
</tr>
<tr>
<td>Vectorize</td>
<td>vectorize_paddings - whether to vectorize pad operations</td>
</tr>
<tr>
<td>PipelineOneParentLoop</td>
<td>parent_loop_num - which parent loop to pipeline, II - iteration interval, read_latency - Latency of a read operation</td>
</tr>
<tr>
<td>UnrollOneParentLoop</td>
<td>parent_loop_num - which parent loop to unroll, unroll_amount - by how many iterations to unroll the loop</td>
</tr>
<tr>
<td>UnrollOneVectorize</td>
<td>source_shape - source shape of the vector to unroll, source_shape - target shape to unroll the vector to</td>
</tr>
<tr>
<td>Bufferize</td>
<td>none</td>
</tr>
<tr>
<td>Sparset</td>
<td>none</td>
</tr>
<tr>
<td>LowerVectors</td>
<td>contraction_lowering - how to lower vector contractions (outer/inner product, LLVM intrinsic), multi_reduction_lowering - how to lower multidimensional reductions (inner or outer parallel), transpose_lowering - how to lower transpositions (elementwise, flat, vector shuffle, target-specific)</td>
</tr>
<tr>
<td>LowerToLLVMM</td>
<td>none</td>
</tr>
</tbody>
</table>
Conclusion

- MLIR Code generation focuses on taking high-level tensor computation primitives and lowering them to LLVM IR with appropriate intrinsics
- Attempts to take guesswork out of the backend, reducing dependence on black-box optimizers such as the LLVM auto-vectorizer
- Leverages LLVM for traditional code generation
- Many abstractions shared between CPU and GPU compilation pipeline (and potentially other new accelerators)
- Tiling and vectorization key components of both pipelines
- Additional work required to manage shared memory and target tensor cores on GPU
- Auto-tuning essential to obtaining good performance from code generated kernels
- Can be extended to handle sparse tensors, non-structured ops (linalg.ext)
Acknowledgements

• Nod.ai Team
  • Discord: https://discord.gg/RUqY2h2s9u

• Google IREE Team
  • Discord: https://discord.gg/26P4xW4
References

- Bradbury, A. LLVM backend development by example (RISC-V). https://youtu.be/AFaIP-dF-RA
- Codegen Dialect Overview. https://discourse.llvm.org/t/codegen-dialect-overview/2723
- Images on Motivation slide taken from OpenAI, NVIDIA, AMD, Sambanova and Cerebras websites.