nod

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

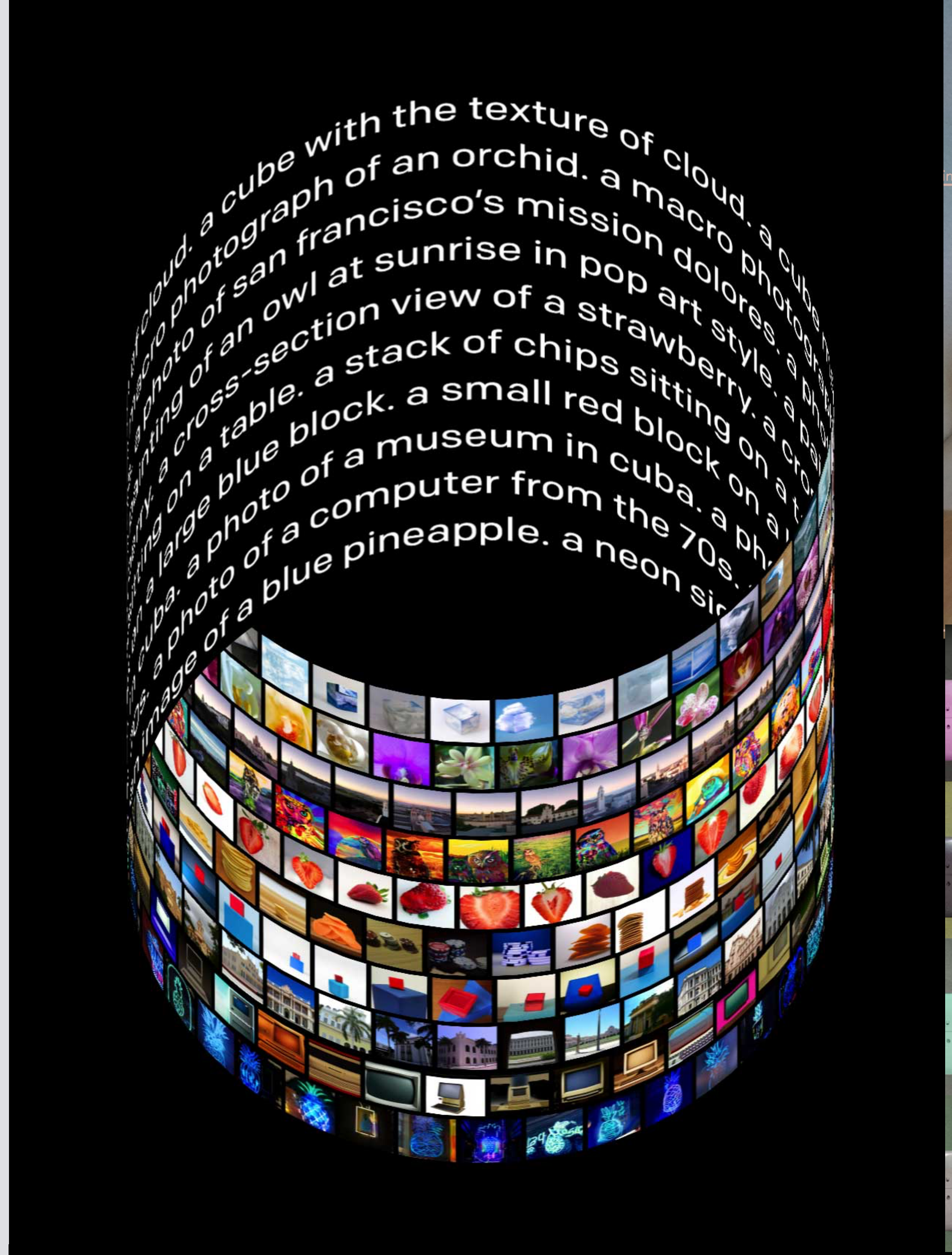
- Large models are achieving state of the art results (such as transformers in NLP)

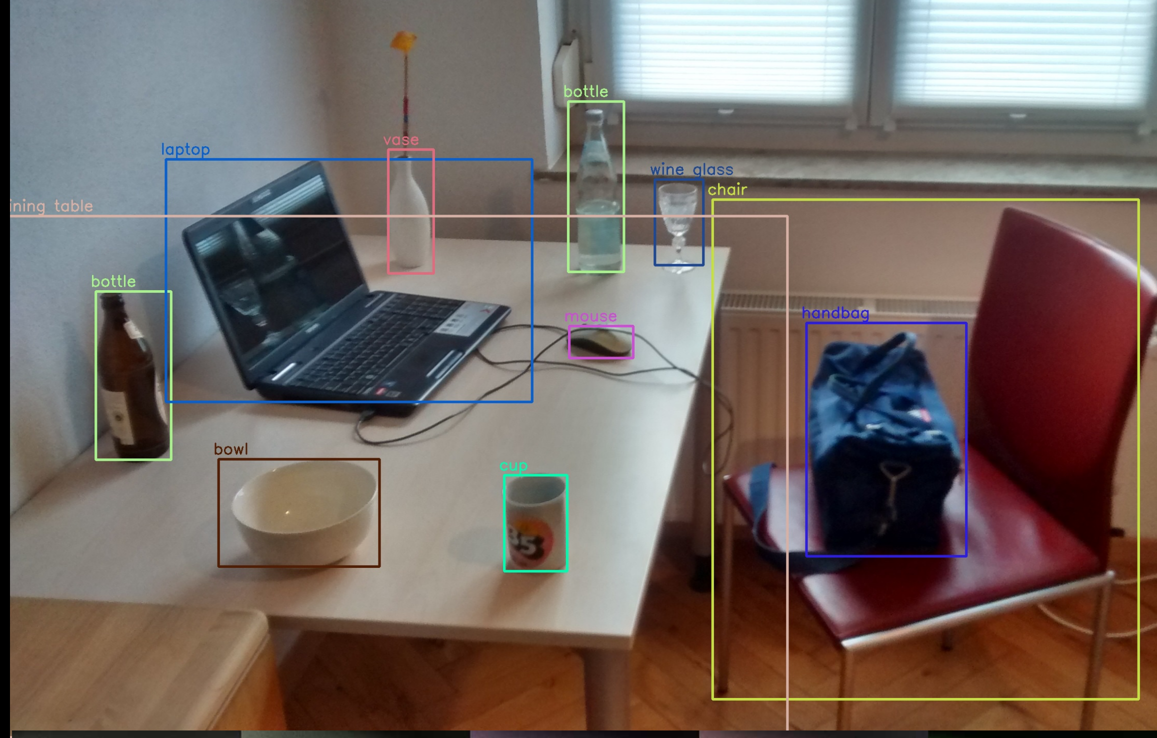
- Unfortunately, these large models take many months and millions of dollars to train on existing hardware

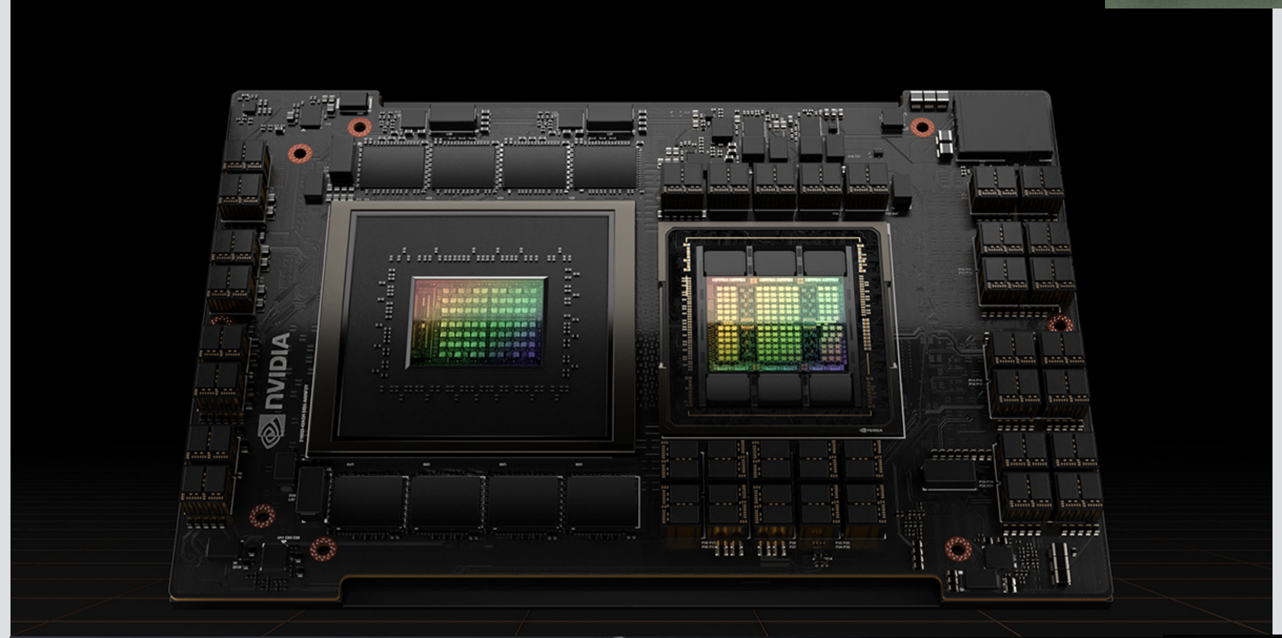
- On edge-based systems, inference dominates with latency being a key metric

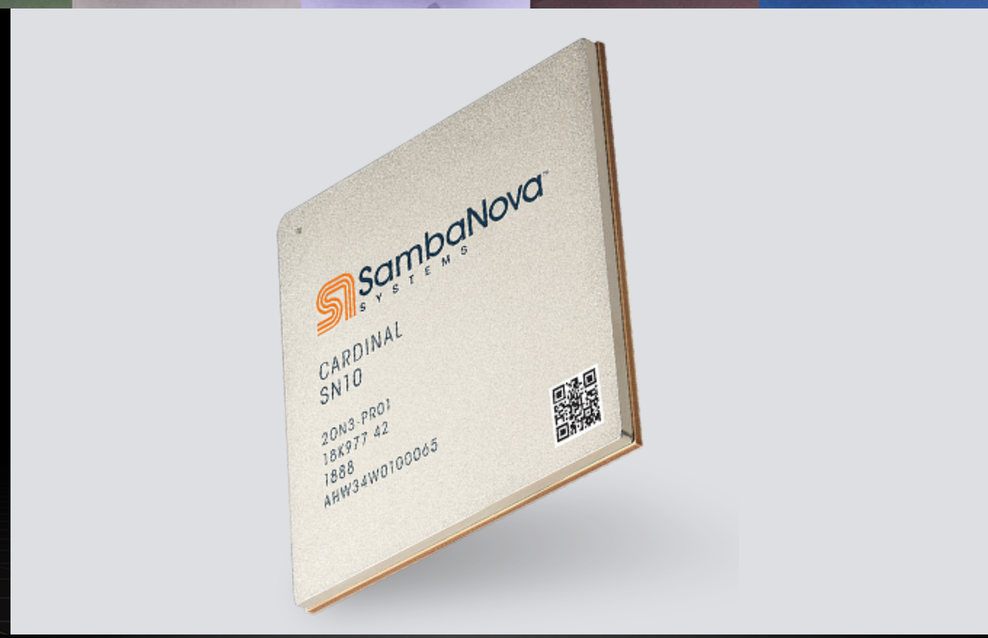
- New hardware vendors have risen to the occasion with custom accelerators that address some of these concerns

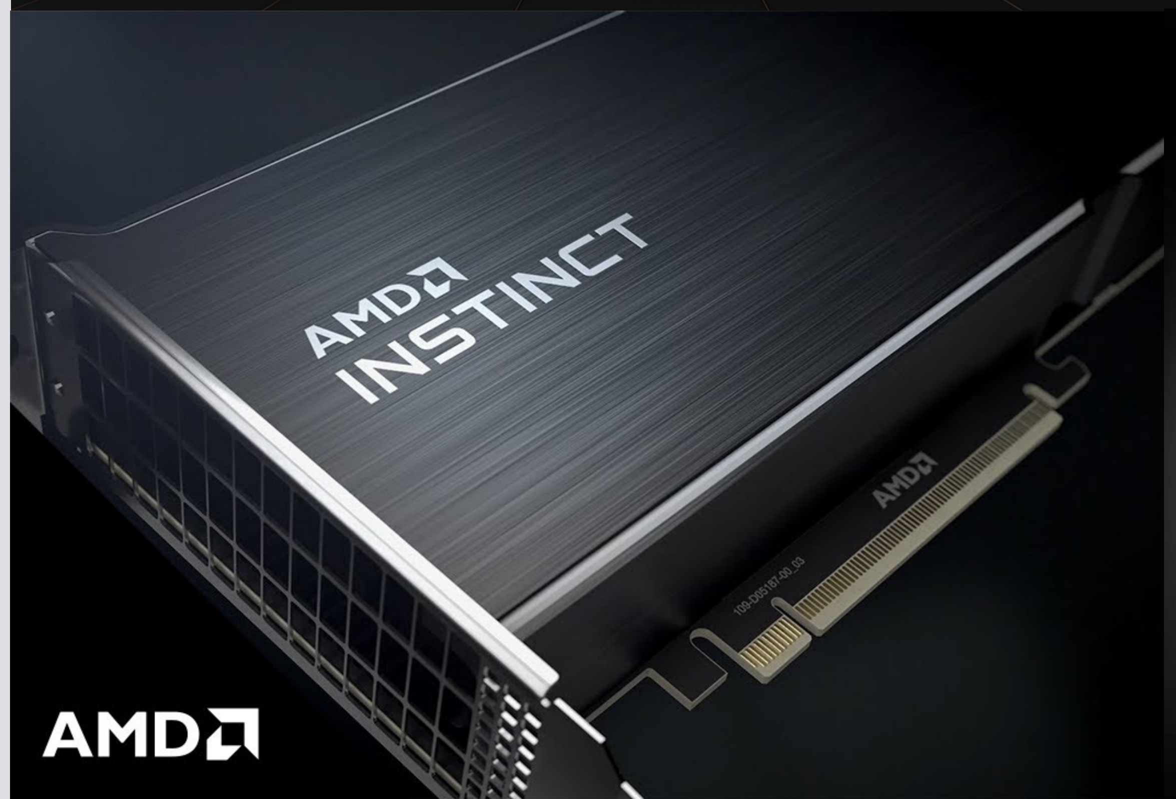
- But as the number of models and hardware combinations explode, need a strong compiler infrastructure that can provide performance gains and is easily re-targetable to new hardware

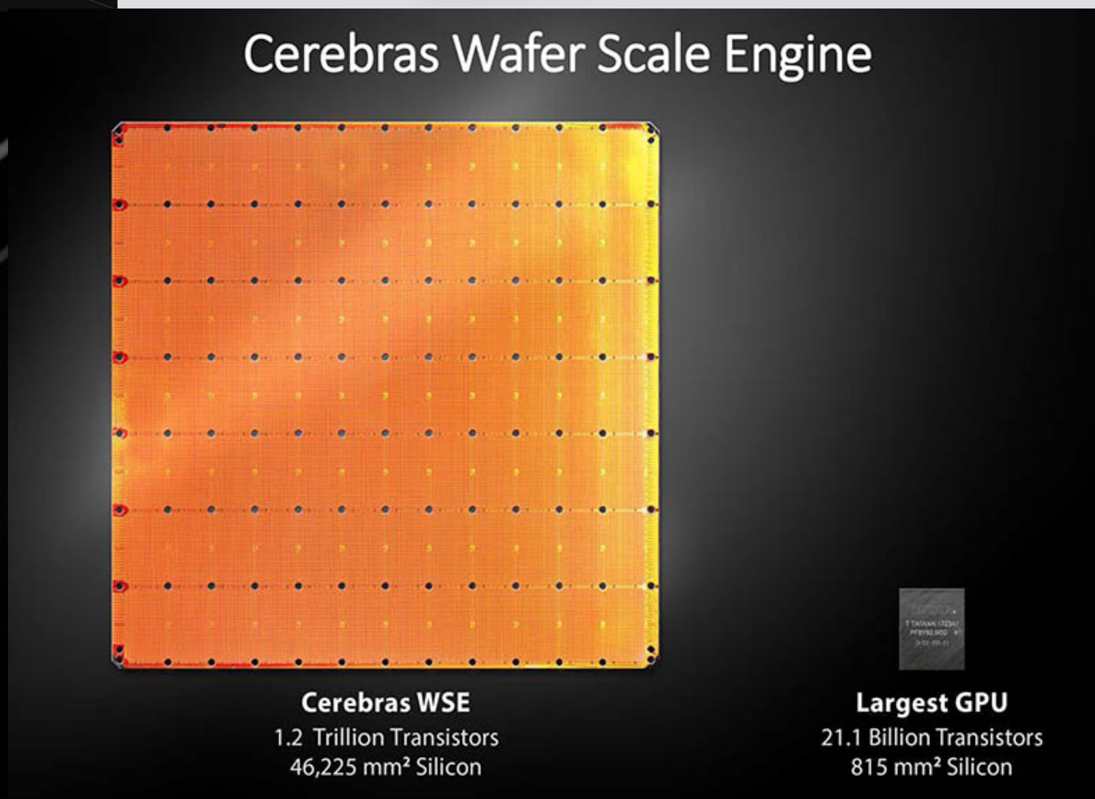

a cube with the texture of cloud.
photograph of an orchid. a macro photo
of san francisco's mission dolores
an owl at sunrise in pop art style
cross-section view of a strawberry.
a stack of chips sitting on a
table. a small red block on a
blue block. a photo of a museum in cuba.
a photo of a computer from the 70s.
a blue pineapple. a neon sign









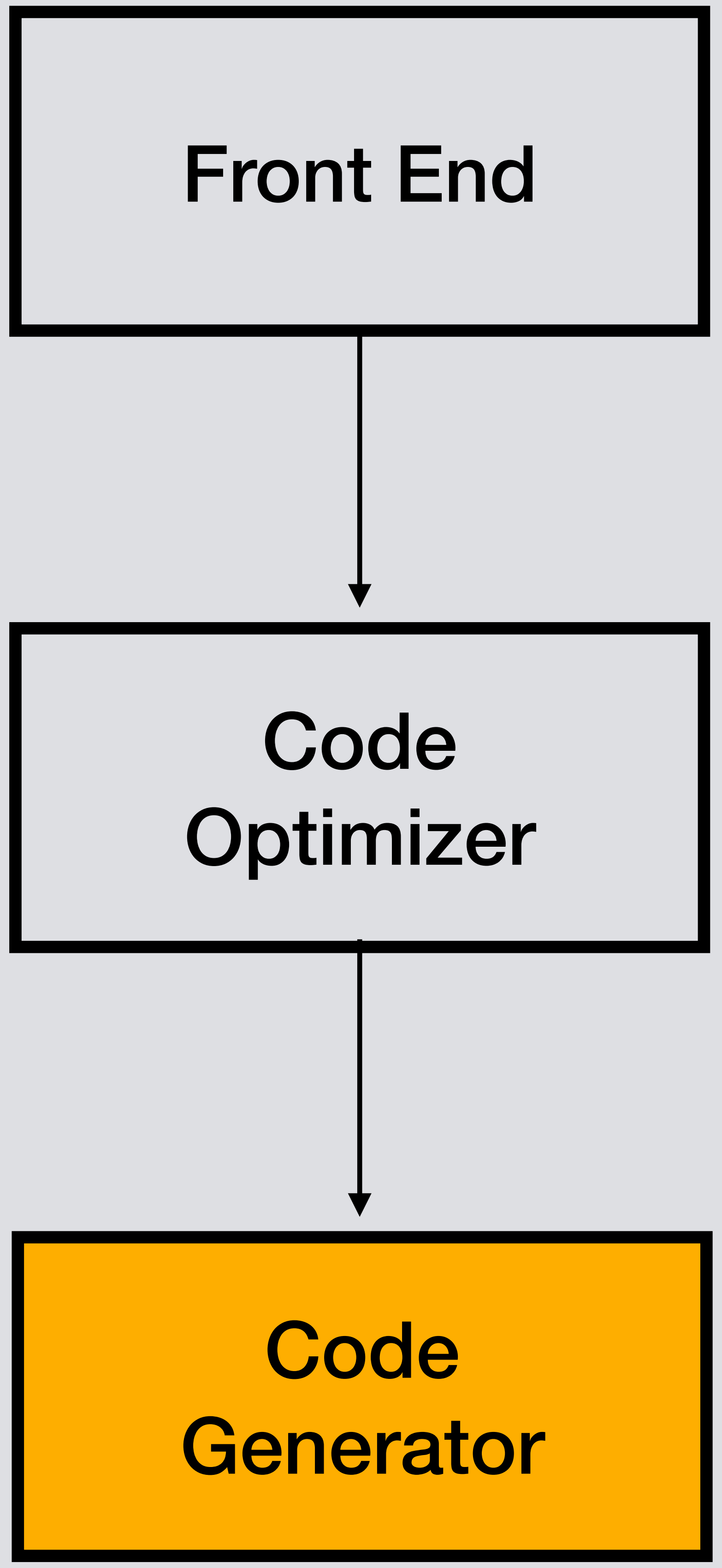


Motivation

- 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

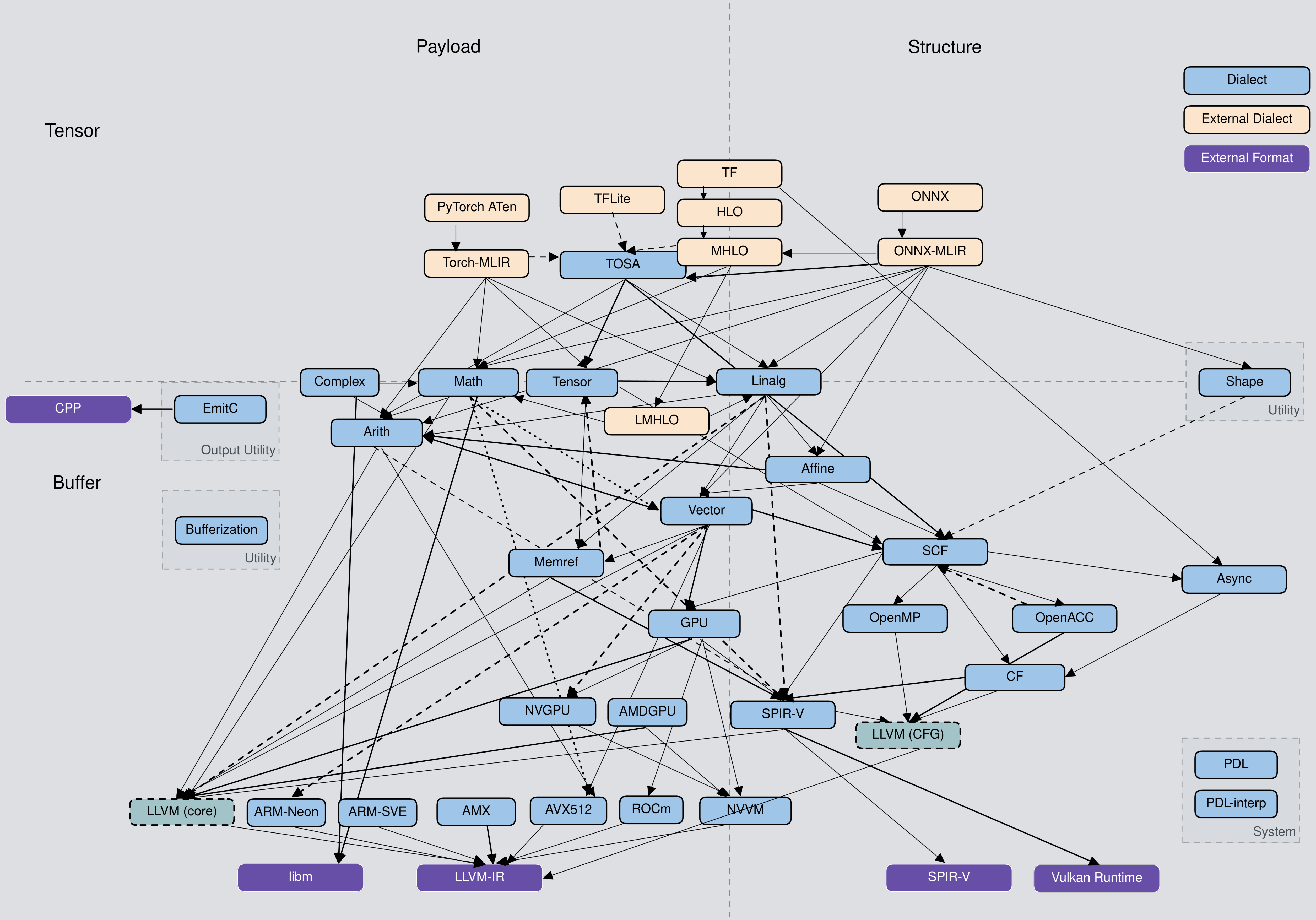
Code generation in LLVM

- 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

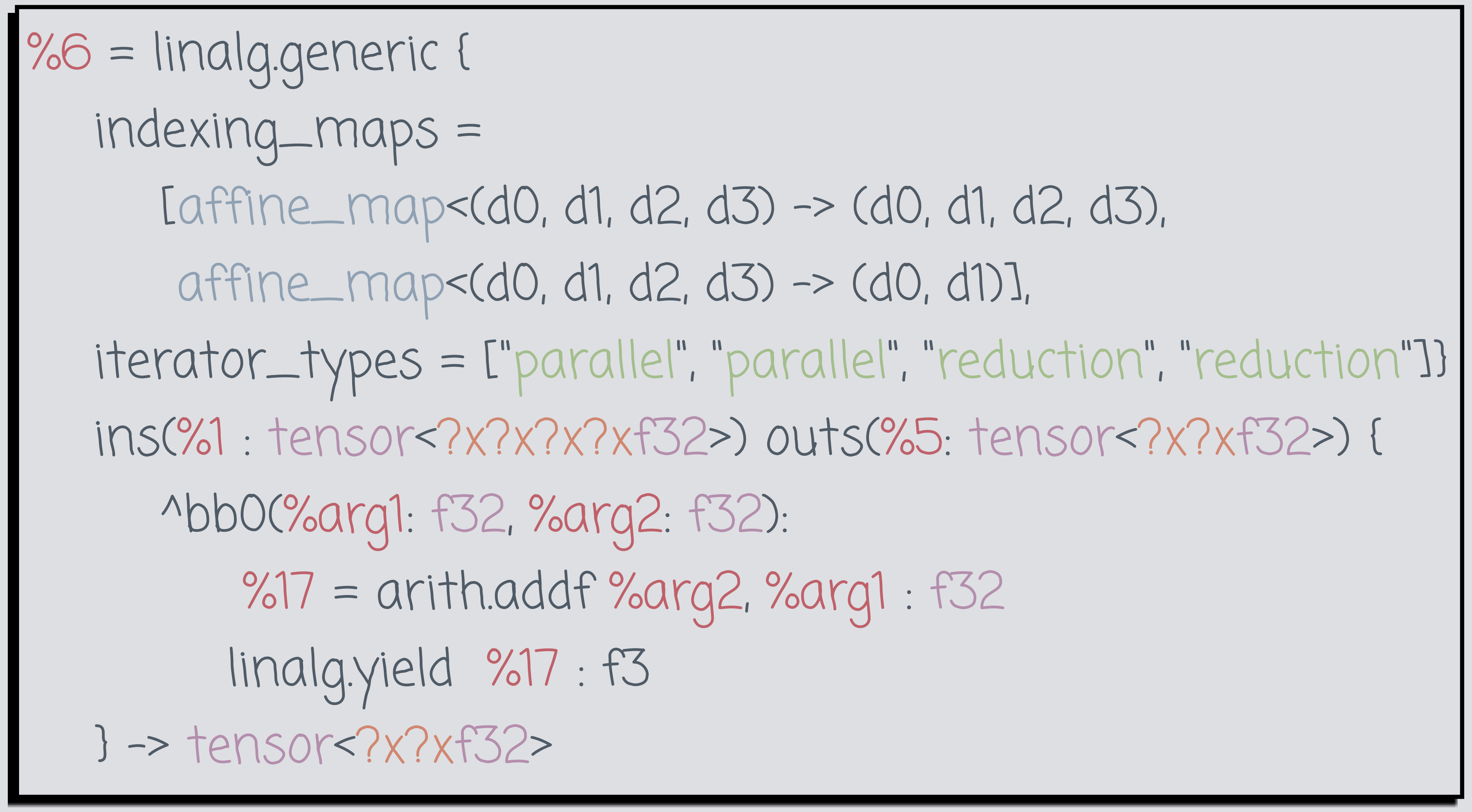


Linalg Dialect

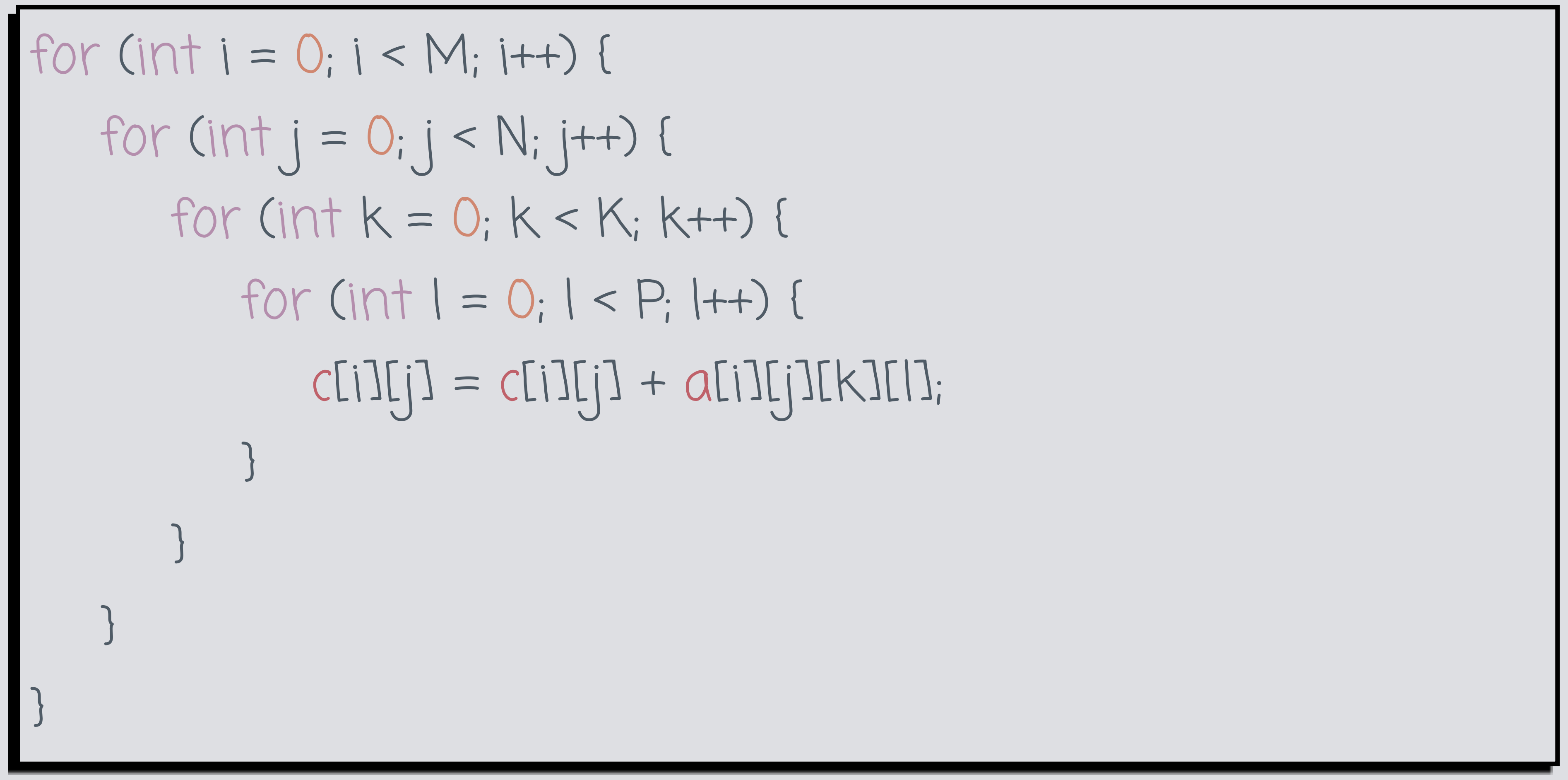
- Linalg dialect is used to represent perfectly nested loop computations making it easy to perform transformations like fusion, tiling, and loop interchange
- Operates on both tensors and memrefs
- Can be lowered to loops or affine expressions with computation in loop body
- Incorporates learnings from 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



```
%6 = 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"]}
    ins(%1 : tensor<?x?x?x?xf32>) outs(%5: tensor<?x?xf32>) {
        ^bb0(%arg1: f32, %arg2: f32):
            %17 = arith.addf %arg2, %arg1 : f32
            linalg.yield  %17 : f3
    } -> tensor<?x?xf32>
```



```
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

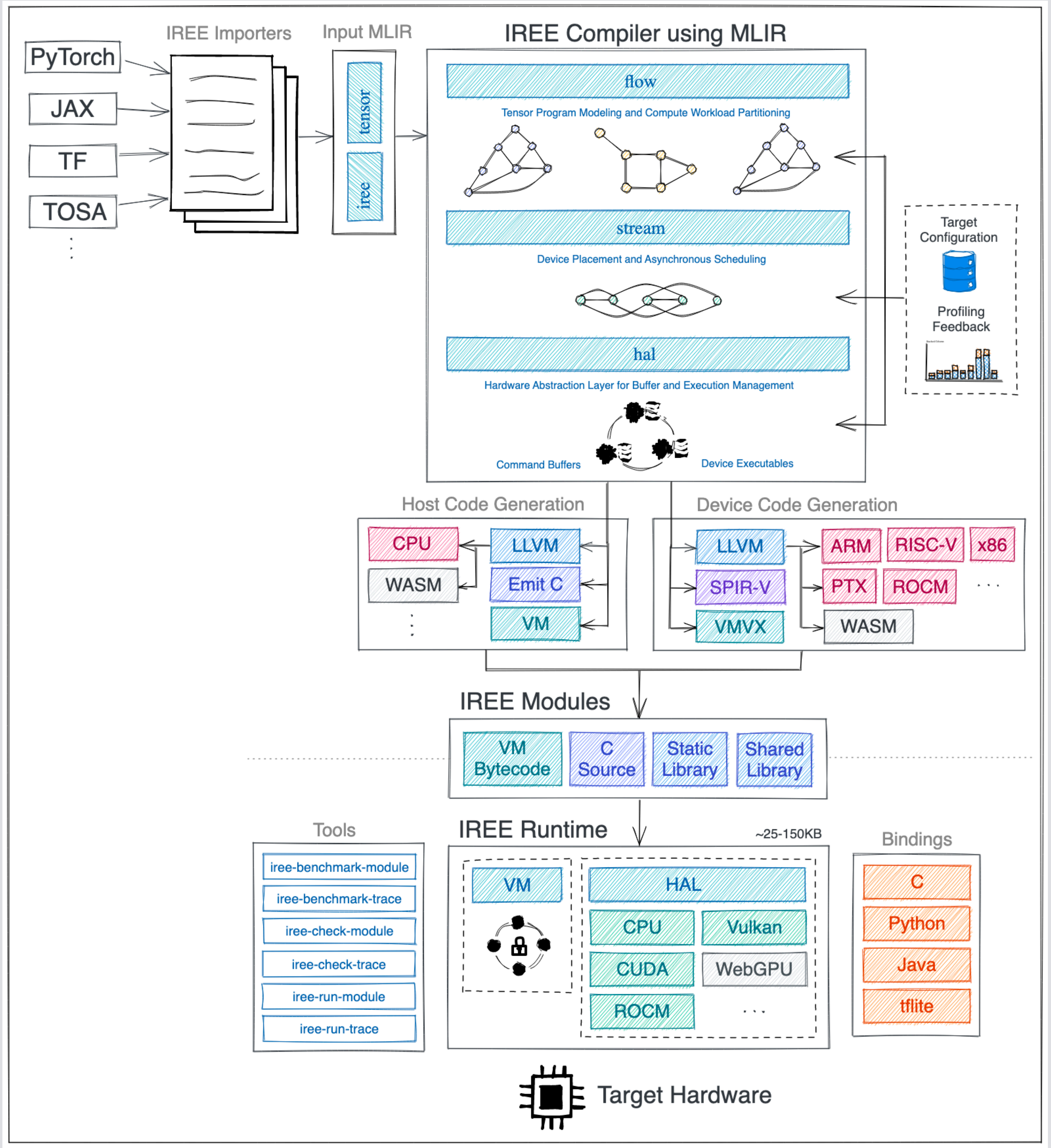
- Provides abstractions for retargetable GPU model
- Contains operations common to SIMT platforms
- Examples
    - Communication: gpu.all_reduce (reduction across a local workgroup)
    - Synchronization: gpu.barrier
    - Compute: gpu.subgroup_mma_compute
- Can be obtained by lowering from vector dialect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

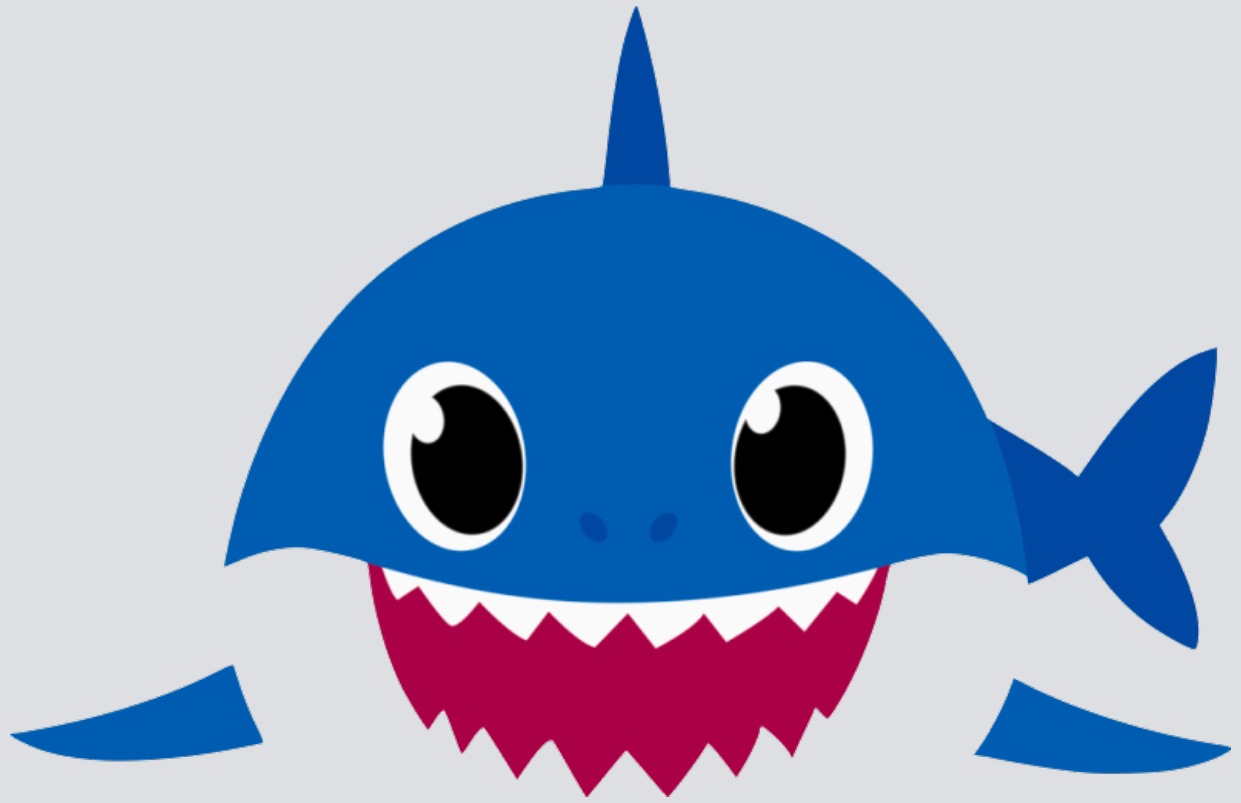
Code generation using IREE

- 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)

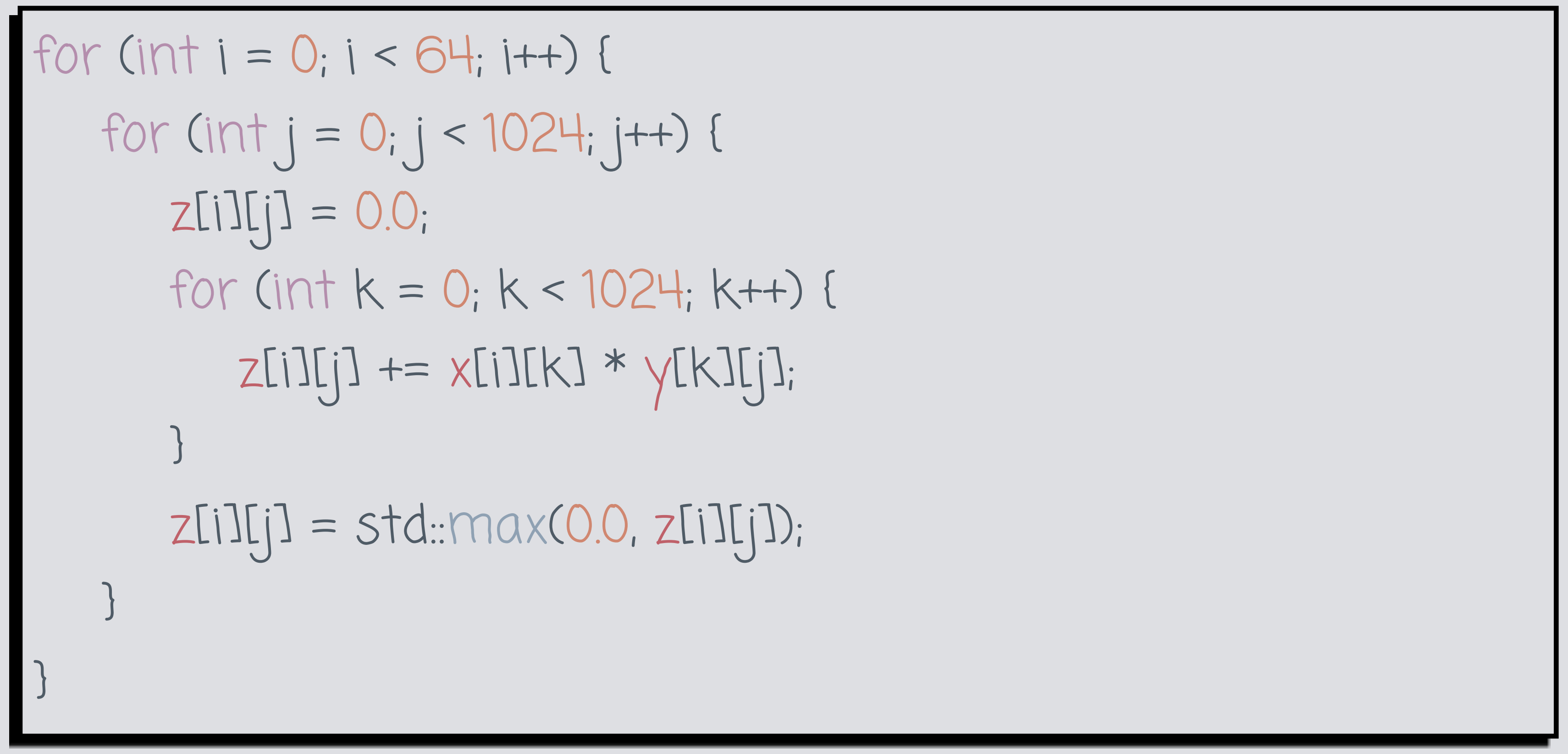


Perceptron Code Generation

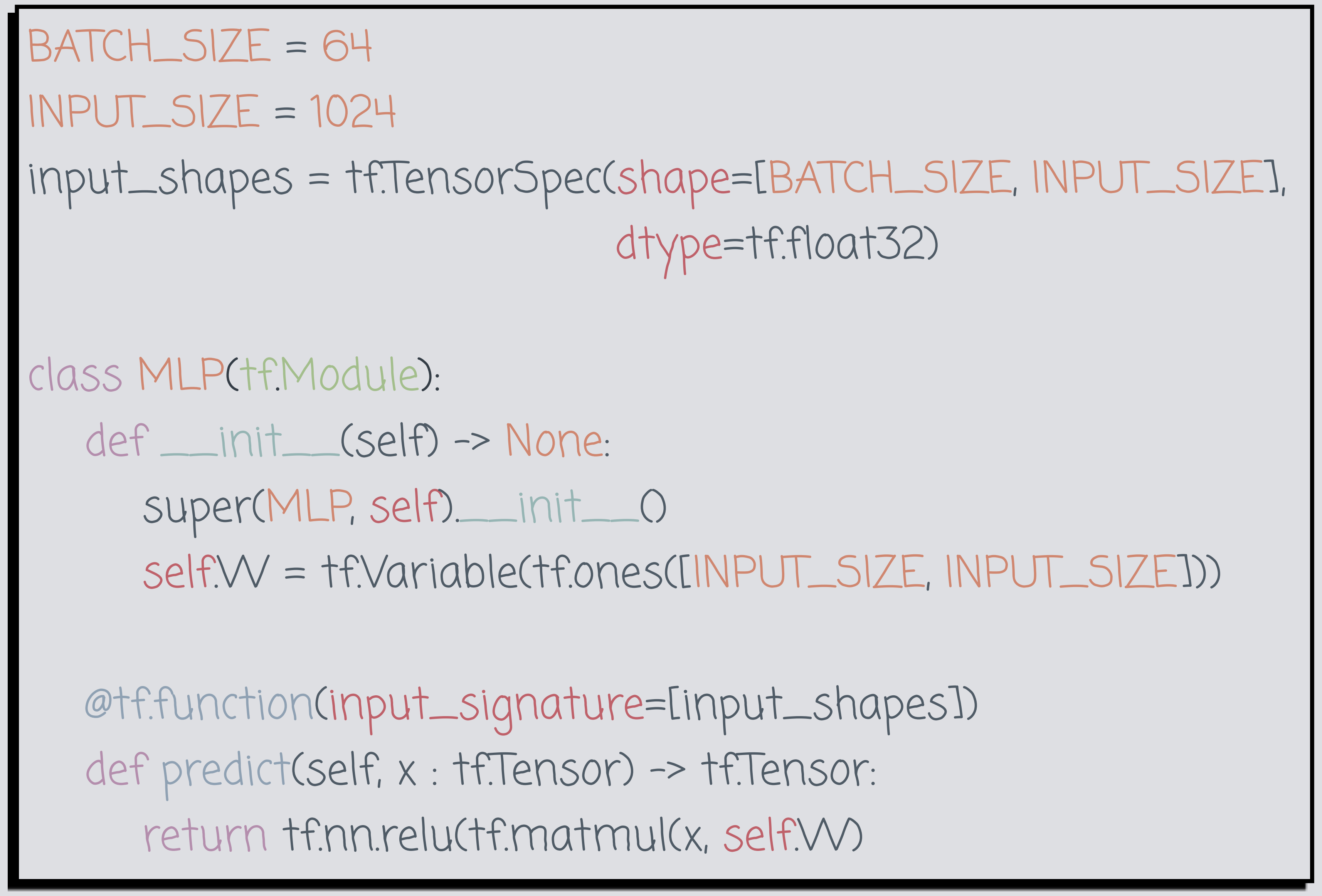
- Perceptron is defined mathematically as shown below

$$z = max(0, XW)$$

- 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



```
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 1024; j++) {
        z[i][j] = 0.0;
        for (int k = 0; k < 1024; k++) {
            z[i][j] += x[i][k] * y[k][j];
        }
        z[i][j] = std::max(0.0, z[i][j]);
    }
}
```



```
BATCH_SIZE = 64
INPUT_SIZE = 1024
input_shapes = tf.TensorSpec(shape=[BATCH_SIZE, INPUT_SIZE],
                             dtype=tf.float32)

class MLP(tf.Module):
    def __init__(self) -> None:
        super(MLP, self).__init__()
        self.W = tf.Variable(tf.ones([INPUT_SIZE, INPUT_SIZE]))

    @tf.function(input_signature=[input_shapes])
    def predict(self, x : tf.Tensor) -> tf.Tensor:
        return tf.nn.relu(tf.matmul(x, self.W))
```

Lowering to Linalg on Tensors

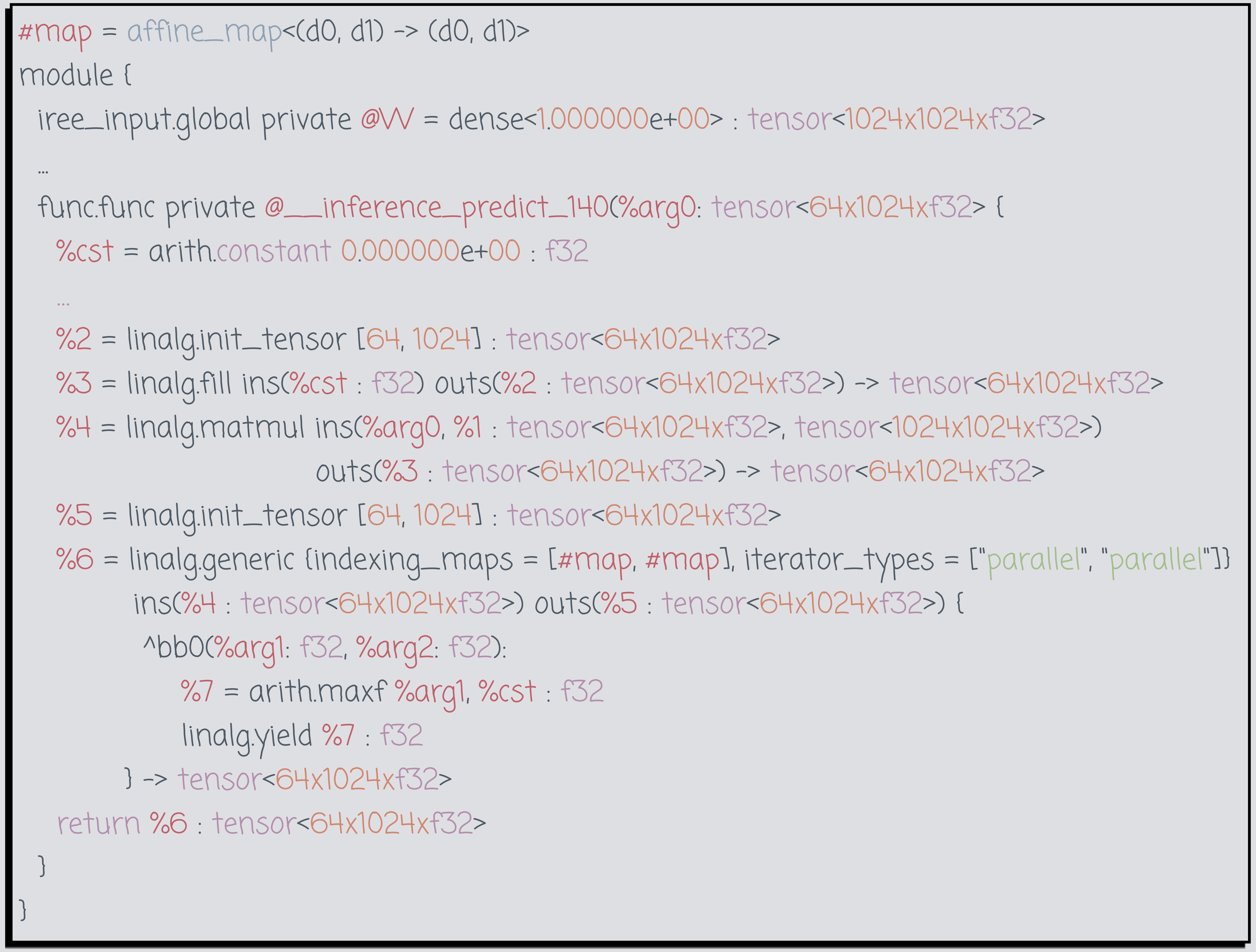
- TF is lowered to MHLO and then to the linalg dialect

- Both matrix multiplication and the ReLU are lowered to linalg.generic ops

- Important to note that linalg.matmul is

$$C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$$

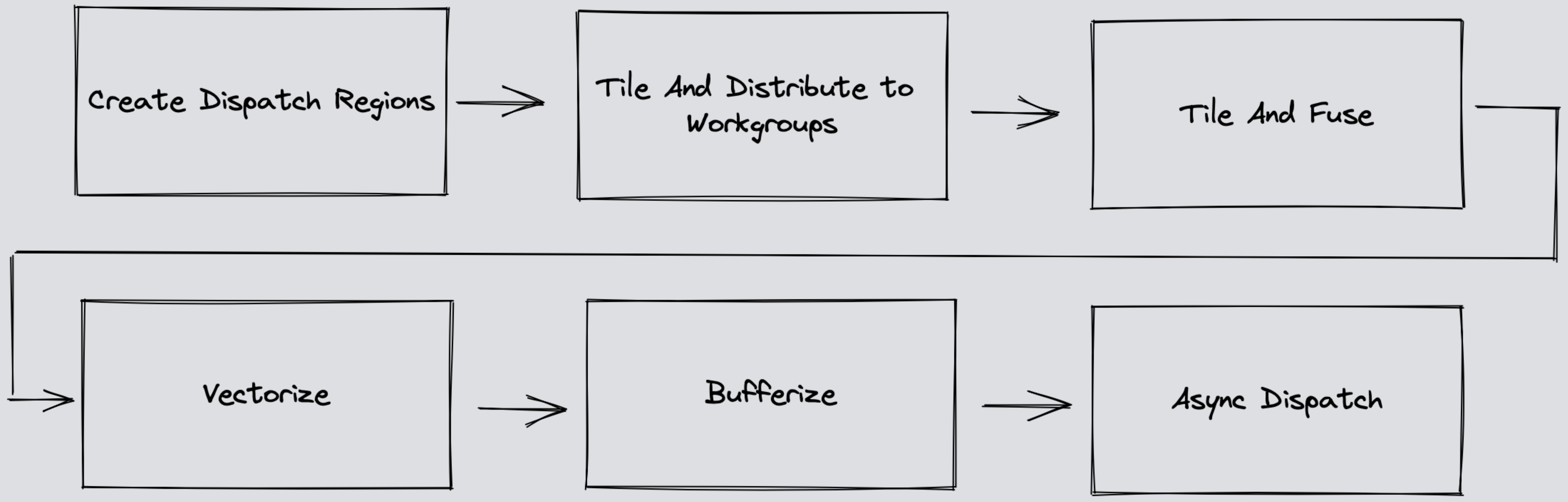
- Hence, we need a linalg.fill before the linalg.matmul



```
#map = affine_map<(d0, d1) -> (d0, d1)>
module {
  iree_input.global private @W = 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, prefering 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 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

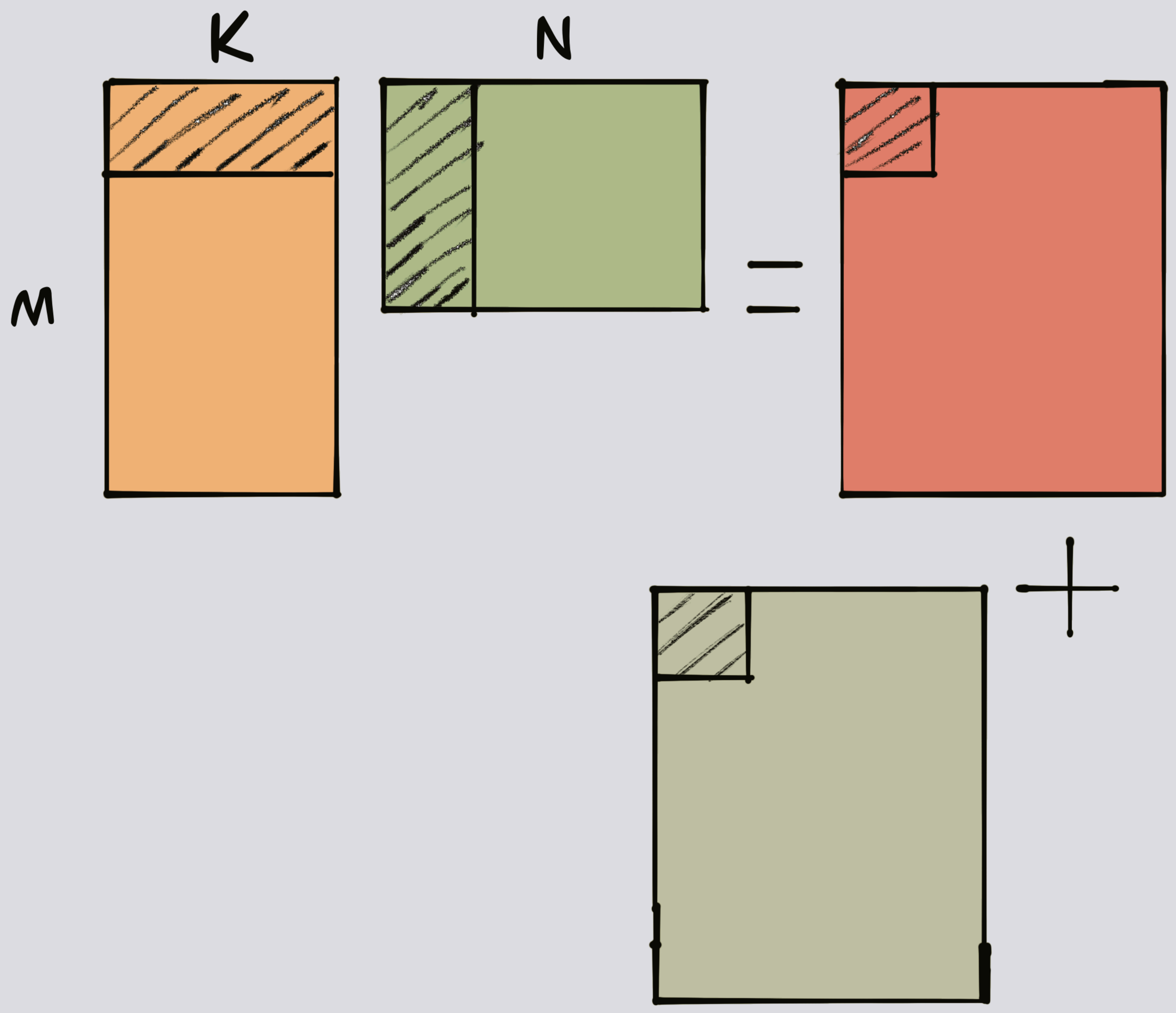
- Also does elementwise fusion to fuse linalg.generic ops

```
%1 = flow.dispatch.workgroups[%c64, %c1024](%0) : (tensor<64x1024xf32>) -> tensor<64x1024xf32>
    (%arg1: !flow.dispatch.tensor<readonly:64x1024xf32>, %arg2:
      !flow.dispatch.tensor<writeonly:64x1024xf32>) {
  %3 = flow.dispatch.tensor.load %arg1, offsets = [0, 0], sizes = [64, 1024], strides = [1, 1] :
        !flow.dispatch.tensor<readonly:64x1024xf32> -> tensor<64x1024xf32>
  %4 = linalg.init_tensor [64, 1024] : tensor<64x1024xf32>
  %5 = linalg.fill ins(%cst : f32) outs(%4 : tensor<64x1024xf32>) -> tensor<64x1024xf32>
  %6 = linalg.matmul ins(%3, %cst_0 : tensor<64x1024xf32>, tensor<1024x1024xf32>)
        outs(%5 : tensor<64x1024xf32>) -> tensor<64x1024xf32>
  %7 = linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>,
                                        affine_map<(d0, d1) -> (d0, d1)>],
                       iterator_types = ["parallel", "parallel"]}
  ins(%6 : tensor<64x1024xf32>) outs(%4 : tensor<64x1024xf32>) {
  ^bb0(%arg3: f32, %arg4: f32):
    %8 = arith.maxf %arg3, %cst : f32
    linalg.yield %8 : f32
  } -> tensor<64x1024xf32>
  flow.dispatch.tensor.store %7, %arg2, offsets = [0, 0], sizes = [64, 1024], strides = [1, 1] :
      tensor<64x1024xf32> -> !flow.dispatch.tensor<writeonly:64x1024xf32>
  flow.return
} count(%arg1: index, %arg2: index) -> (index, index, index) {
  %x, %y, %z = flow.dispatch.default_workgroup_count %arg1, %arg2
  flow.return %x, %y, %z : index, index, index
}
```

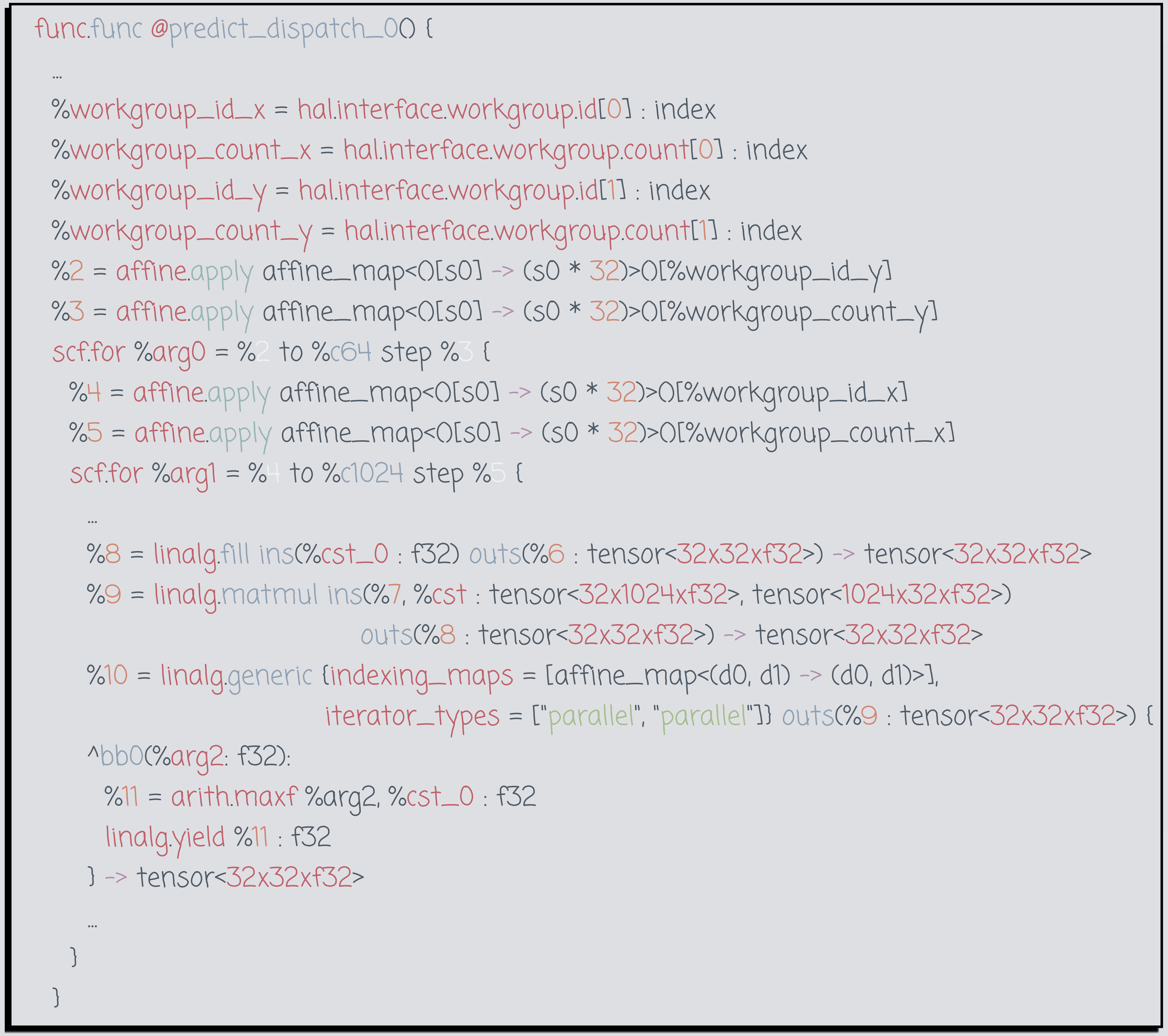
Body of Dispatch Region

Tile and Distribute to Workgroups

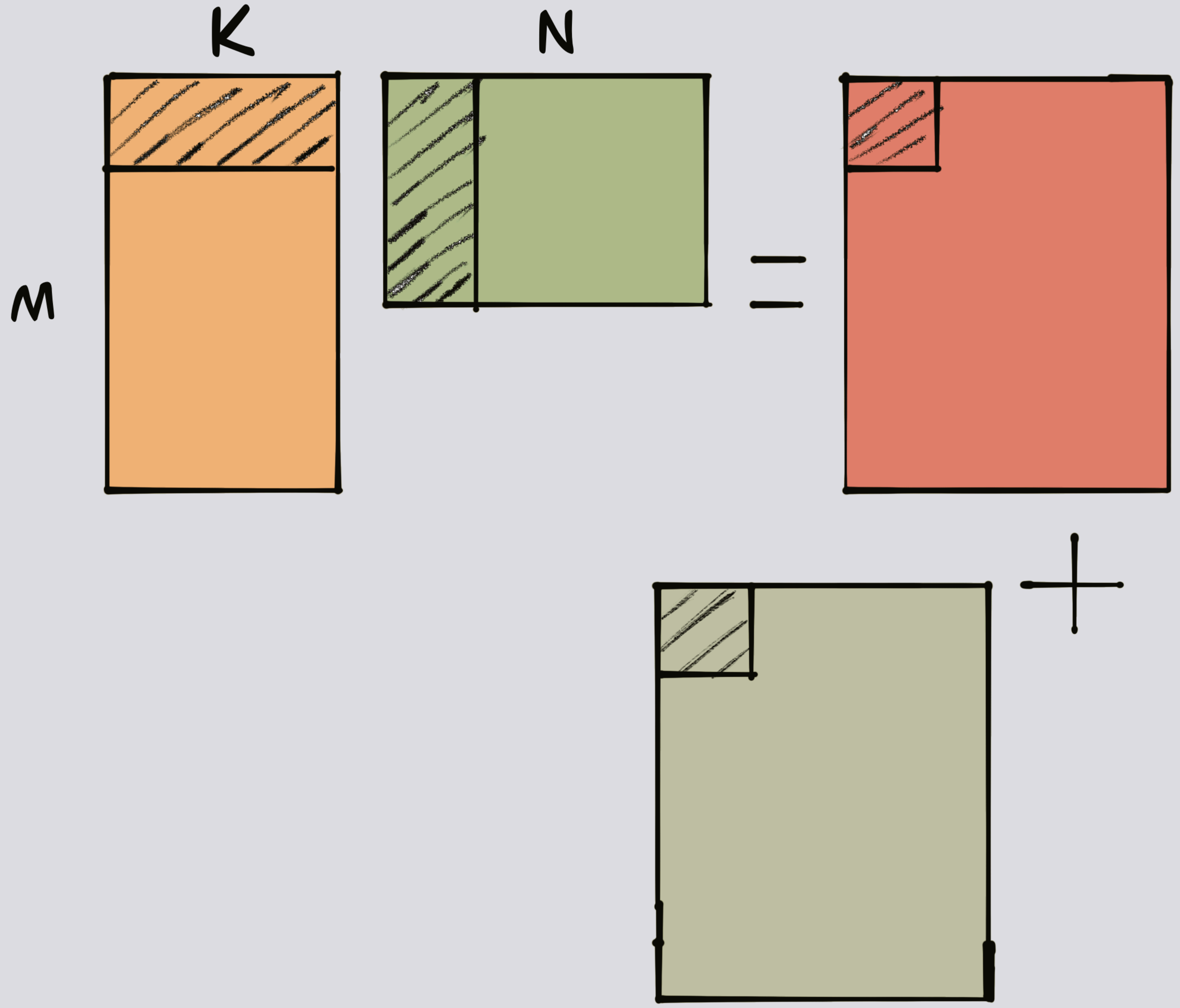
- Work partitioned along a 3-D grid of virtual processors (workgroups) that can be mapped to multi-core CPUs or GPUs
- Uses a block cyclic distribution to distribute the tiles
- Each dispatch region performs a tile of the computation
- The compute done is determined by its rank (workgroup_id_x, workgroup_id_y)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



Tile and Distribute to Workgroups

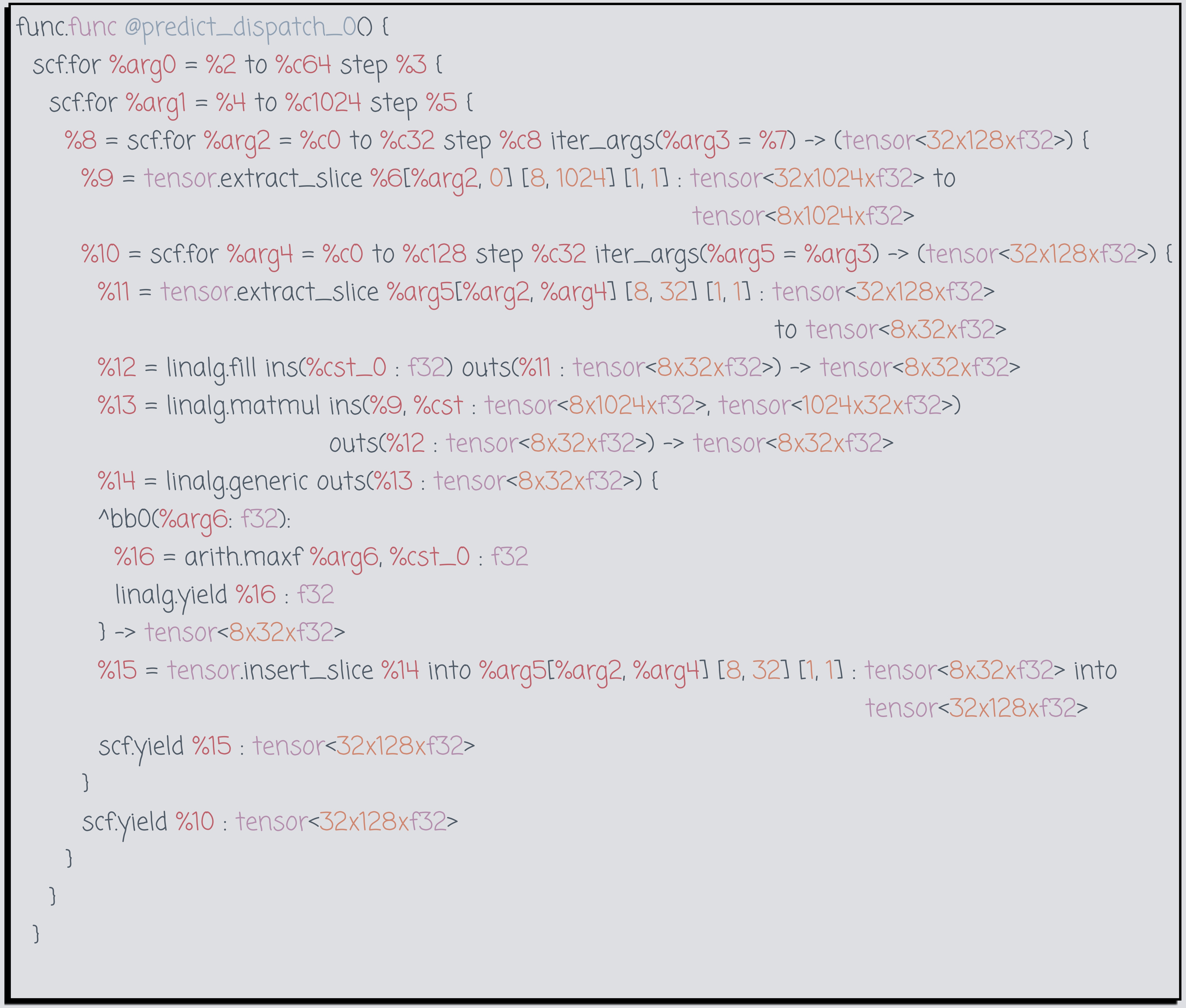


```
func.func @predict_dispatch_0() {
  ...
  %workgroup_id_x = hal.interface.workgroup.id[0] : index
  %workgroup_count_x = hal.interface.workgroup.count[0] : index
  %workgroup_id_y = hal.interface.workgroup.id[1] : index
  %workgroup_count_y = hal.interface.workgroup.count[1] : index
  %2 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_id_y]
  %3 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_count_y]
  scf.for %arg0 = %2 to %c64 step %3 {
    %4 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%workgroup_id_x]
    %5 = affine.apply affine_map<()[s0] -> (s0 * 32)>()[%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

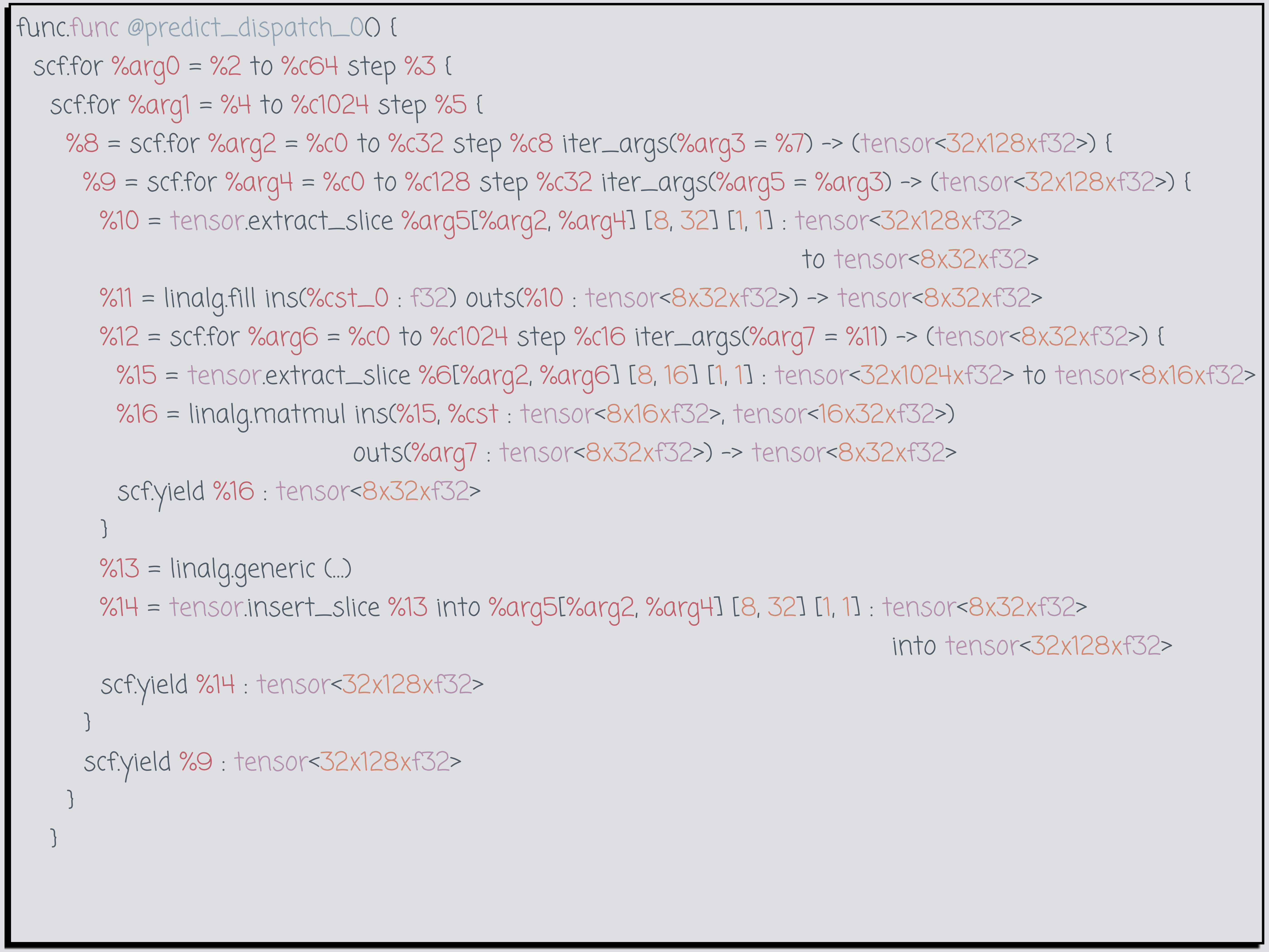
- Additional tiling along parallel dimensions
- Introduces subset operations (tensor.extract_slice, tensor.insert_slice) to access the tiled data
- Depending on chosen tile sizes, there may not exists a single static tensor type valid for every iteration
- The sub-tensor may be relaxed to a dynamic tensor
- 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



```
func.func @predict_dispatch_0() {
  scf.for %arg0 = %2 to %c64 step %3 {
    scf.for %arg1 = %4 to %c1024 step %5 {
      %8 = scf.for %arg2 = %c0 to %c32 step %c8 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 = %c0 to %c128 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>) -> tensor<8x32xf32>
          %13 = linalg.matmul ins(%9, %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<8x32xf32> into
                                                                           tensor<32x128xf32>
          scf.yield %15 : tensor<32x128xf32>
        }
        scf.yield %10 : tensor<32x128xf32>
      }
    }
  }
```

Single Tiling Expert

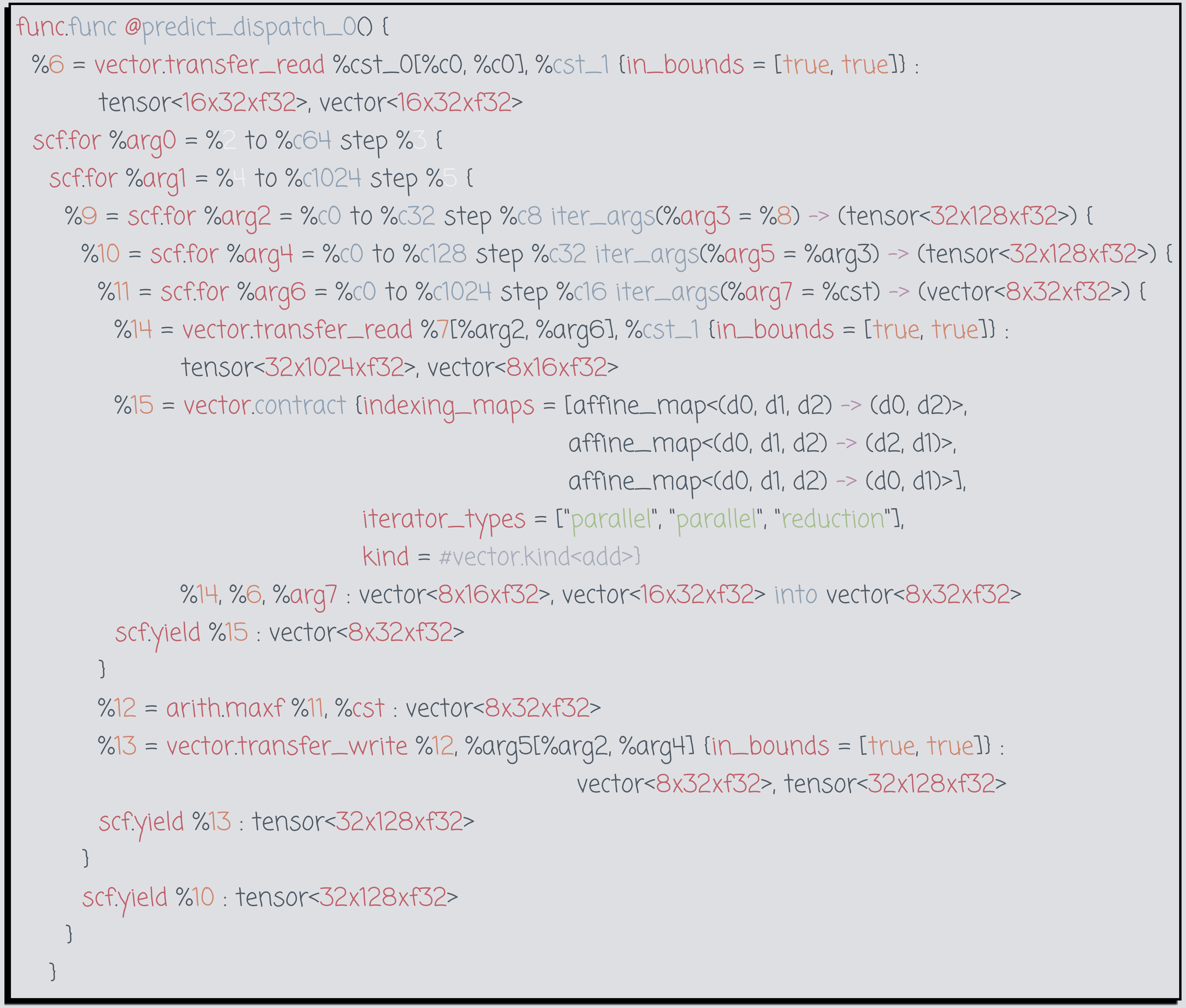
- Multiple different ways of tiling the matmul
  - DoubleTilingExpert, TripleTilingExpert, DoubleTilingPadExpert etc.
- Using SingleTilingExpert get a single tiling of reduction dimension
- Changes K dimension from 1024 to 16



```
func.func @predict_dispatch_0() {
  scf.for %arg0 = %2 to %c64 step %3 {
    scf.for %arg1 = %4 to %c1024 step %5 {
      %8 = scf.for %arg2 = %c0 to %c32 step %c8 iter_args(%arg3 = %7) -> (tensor<32x128xf32>) {
        %9 = scf.for %arg4 = %c0 to %c128 step %c32 iter_args(%arg5 = %arg3) -> (tensor<32x128xf32>) {
          %10 = tensor.extract_slice %arg5[%arg2, %arg4] [8, 32] [1, 1] : tensor<32x128xf32>
                                                                          to tensor<8x32xf32>
          %11 = linalg.fill ins(%cst_0 : f32) outs(%10 : tensor<8x32xf32>) -> tensor<8x32xf32>
          %12 = scf.for %arg6 = %c0 to %c1024 step %c16 iter_args(%arg7 = %11) -> (tensor<8x32xf32>) {
            %15 = tensor.extract_slice %6[%arg2, %arg6] [8, 16] [1, 1] : tensor<32x1024xf32> to tensor<8x16xf32>
            %16 = linalg.matmul ins(%15, %cst : tensor<8x16xf32>, tensor<16x32xf32>)
                                   outs(%arg7 : tensor<8x32xf32>) -> tensor<8x32xf32>
            scf.yield %16 : tensor<8x32xf32>
          }
          %13 = linalg.generic (...)
          %14 = tensor.insert_slice %13 into %arg5[%arg2, %arg4] [8, 32] [1, 1] : tensor<8x32xf32>
                                                                                  into tensor<32x128xf32>
          scf.yield %14 : tensor<32x128xf32>
        }
        scf.yield %9 : tensor<32x128xf32>
      }
    }
```

Vectorize

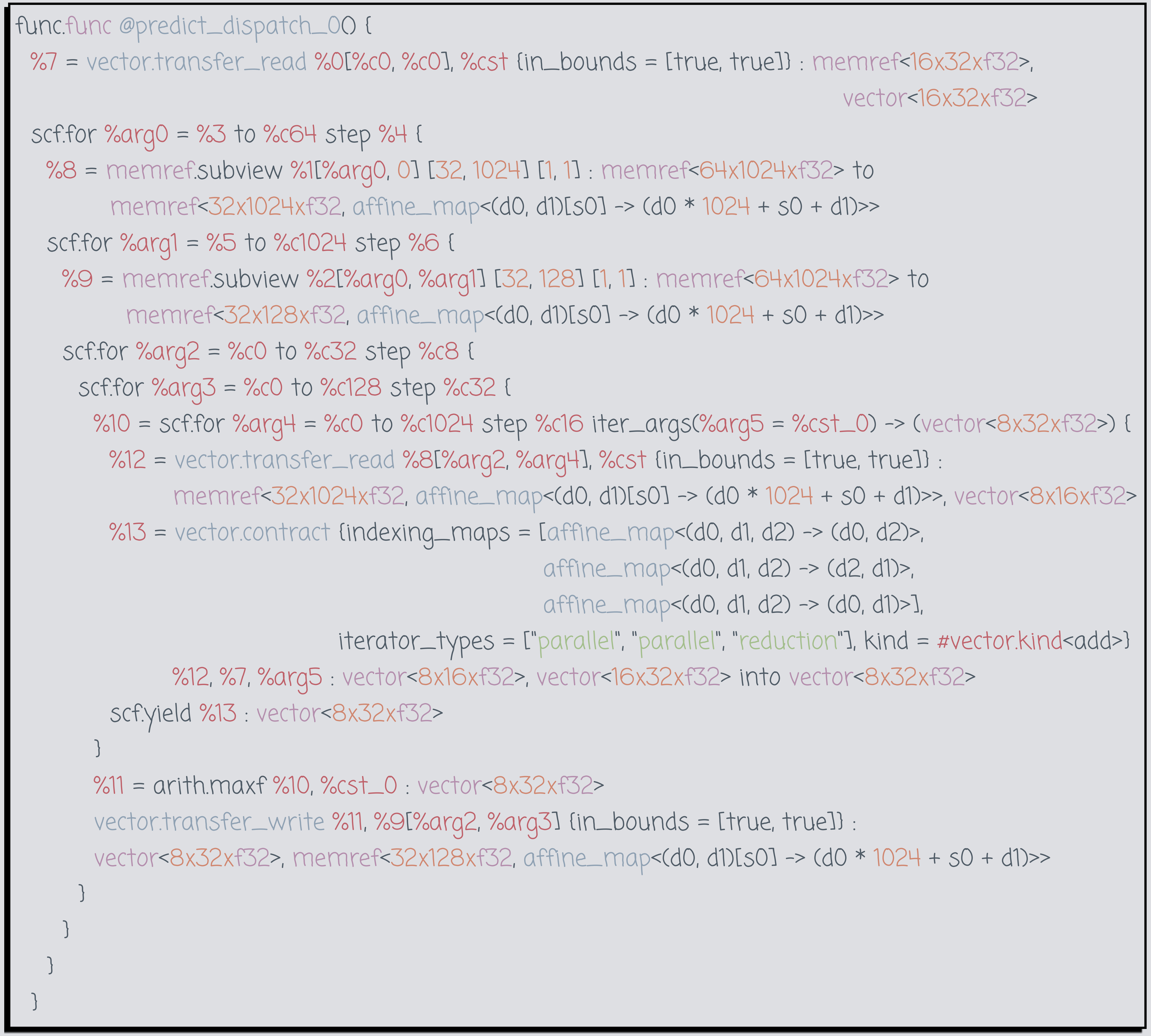
- Emits vector.transfer_read/write operations for each operand
- For elementwise operations, rewrite as pointwise vector variant
- For reductions, rewrite as vector.contract or multi-reduction
- Broadcasting lower dimensional operands is done by vector.broadcast
- Permutations are handled by vector.transpose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



```
func.func @predict_dispatch_0() {
  %6 = vector.transfer_read %cst_0[%c0, %c0], %cst_1 {in_bounds = [true, true]} :
        tensor<16x32xf32>, vector<16x32xf32>
  scf.for %arg0 = %2 to %c64 step %3 {
   scf.for %arg1 = %4 to %c1024 step %5 {
    %9 = scf.for %arg2 = %c0 to %c32 step %c8 iter_args(%arg3 = %8) -> (tensor<32x128xf32>) {
     %10 = scf.for %arg4 = %c0 to %c128 step %c32 iter_args(%arg5 = %arg3) -> (tensor<32x128xf32>) {
      %11 = scf.for %arg6 = %c0 to %c1024 step %c16 iter_args(%arg7 = %cst) -> (vector<8x32xf32>) {
       %14 = vector.transfer_read %7[%arg2, %arg6], %cst_1 {in_bounds = [true, true]} :
             tensor<32x1024xf32>, vector<8x16xf32>
       %15 = vector.contract {indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                                              affine_map<(d0, d1, d2) -> (d2, d1)>,
                                              affine_map<(d0, d1, d2) -> (d0, d1)>],
                              iterator_types = ["parallel", "parallel", "reduction"],
                              kind = #vector.kind<add>}
             %14, %6, %arg7 : vector<8x16xf32>, vector<16x32xf32> into vector<8x32xf32>
       scf.yield %15 : vector<8x32xf32>
      }
      %12 = arith.maxf %11, %cst : vector<8x32xf32>
      %13 = vector.transfer_write %12, %arg5[%arg2, %arg4] {in_bounds = [true, true]} :
                                                  vector<8x32xf32>, tensor<32x128xf32>
      scf.yield %13 : tensor<32x128xf32>
     }
     scf.yield %10 : tensor<32x128xf32>
    }
   }
  }
```

Bufferize

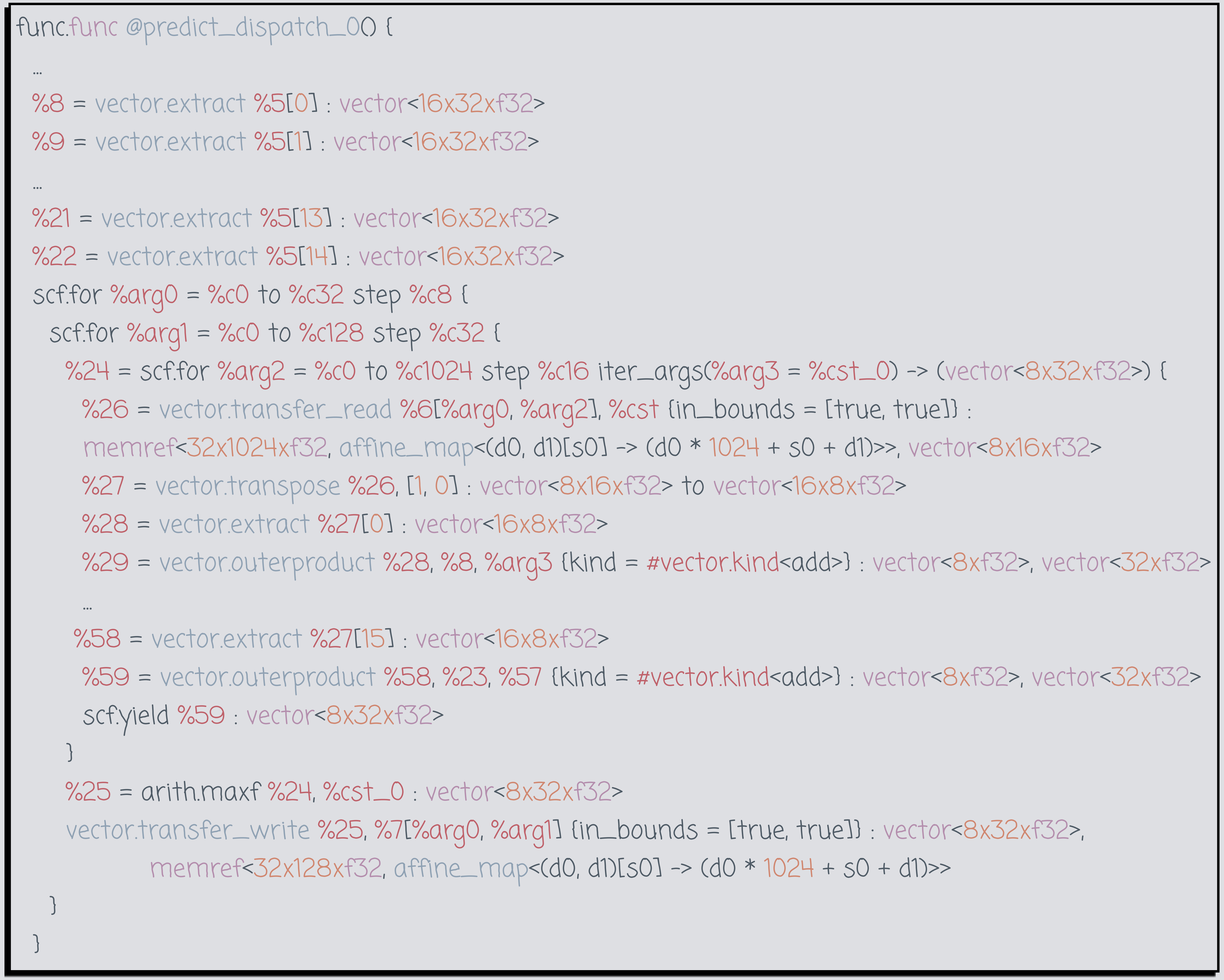
- Allocate and copy as little memory as possible
- Always prefer re-using buffers in place
- Use destination-passing style as a heuristic for in-place bufferization
- Tie output tensor to results tensor to act as bufferization constraint
- Performs a future in-place bufferization analysis of a the operands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



```
func.func @predict_dispatch_0() {
 %7 = vector.transfer_read %0[%c0, %c0], %cst {in_bounds = [true, true]} : memref<16x32xf32>,
                                                                            vector<16x32xf32>
 scf.for %arg0 = %3 to %c64 step %4 {
  %8 = memref.subview %1[%arg0, 0] [32, 1024] [1, 1] : memref<64x1024xf32> to
       memref<32x1024xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>
  scf.for %arg1 = %5 to %c1024 step %6 {
   %9 = memref.subview %2[%arg0, %arg1] [32, 128] [1, 1] : memref<64x1024xf32> to
        memref<32x128xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>
   scf.for %arg2 = %c0 to %c32 step %c8 {
    scf.for %arg3 = %c0 to %c128 step %c32 {
     %10 = scf.for %arg4 = %c0 to %c1024 step %c16 iter_args(%arg5 = %cst_0) -> (vector<8x32xf32>) {
      %12 = vector.transfer_read %8[%arg2, %arg4], %cst {in_bounds = [true, true]} :
            memref<32x1024xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>, vector<8x16xf32>
      %13 = vector.contract {indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                                              affine_map<(d0, d1, d2) -> (d2, d1)>,
                                              affine_map<(d0, d1, d2) -> (d0, d1)>],
                             iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>}
            %12, %7, %arg5 : vector<8x16xf32>, vector<16x32xf32> into vector<8x32xf32>
      scf.yield %13 : vector<8x32xf32>
     }
     %11 = arith.maxf %10, %cst_0 : vector<8x32xf32>
     vector.transfer_write %11, %9[%arg2, %arg3] {in_bounds = [true, true]} :
     vector<8x32xf32>, memref<32x128xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>
    }
   }
  }
 }
}
```

Lowering closer to hardware

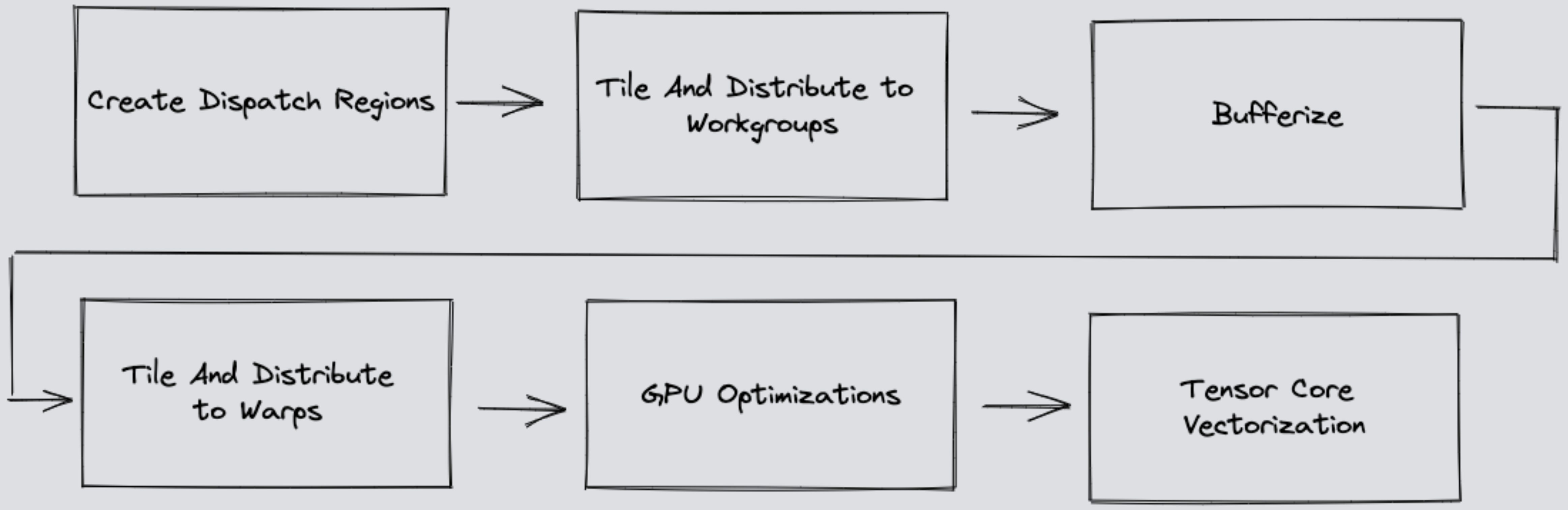
- Apply vector unrolling
  - Breaks down vector sizes to sizes well supported by target
  - Pre-emptively handles non power of 2 sizes to avoid suboptimal code generation
- vector.contract is lowered to outer products to enable mapping to SIMD FMA instructions
- Further lowered to LLVM dialect and translated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



```
func.func @predict_dispatch_0() {
 ...
 %8 = vector.extract %5[0] : vector<16x32xf32>
 %9 = vector.extract %5[1] : vector<16x32xf32>
 ...
 %21 = vector.extract %5[13] : vector<16x32xf32>
 %22 = vector.extract %5[14] : vector<16x32xf32>
 scf.for %arg0 = %c0 to %c32 step %c8 {
  scf.for %arg1 = %c0 to %c128 step %c32 {
   %24 = scf.for %arg2 = %c0 to %c1024 step %c16 iter_args(%arg3 = %cst_0) -> (vector<8x32xf32>) {
    %26 = vector.transfer_read %6[%arg0, %arg2], %cst {in_bounds = [true, true]} :
    memref<32x1024xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>, vector<8x16xf32>
    %27 = vector.transpose %26, [1, 0] : vector<8x16xf32> to vector<16x8xf32>
    %28 = vector.extract %27[0] : vector<16x8xf32>
    %29 = vector.outerproduct %28, %8, %arg3 {kind = #vector.kind<add>} : vector<8xf32>, vector<32xf32>
    ...
   %58 = vector.extract %27[15] : vector<16x8xf32>
    %59 = vector.outerproduct %58, %23, %57 {kind = #vector.kind<add>} : vector<8xf32>, vector<32xf32>
    scf.yield %59 : vector<8x32xf32>
   }
   %25 = arith.maxf %24, %cst_0 : vector<8x32xf32>
   vector.transfer_write %25, %7[%arg0, %arg1] {in_bounds = [true, true]} : vector<8x32xf32>,
             memref<32x128xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>
  }
 }
```

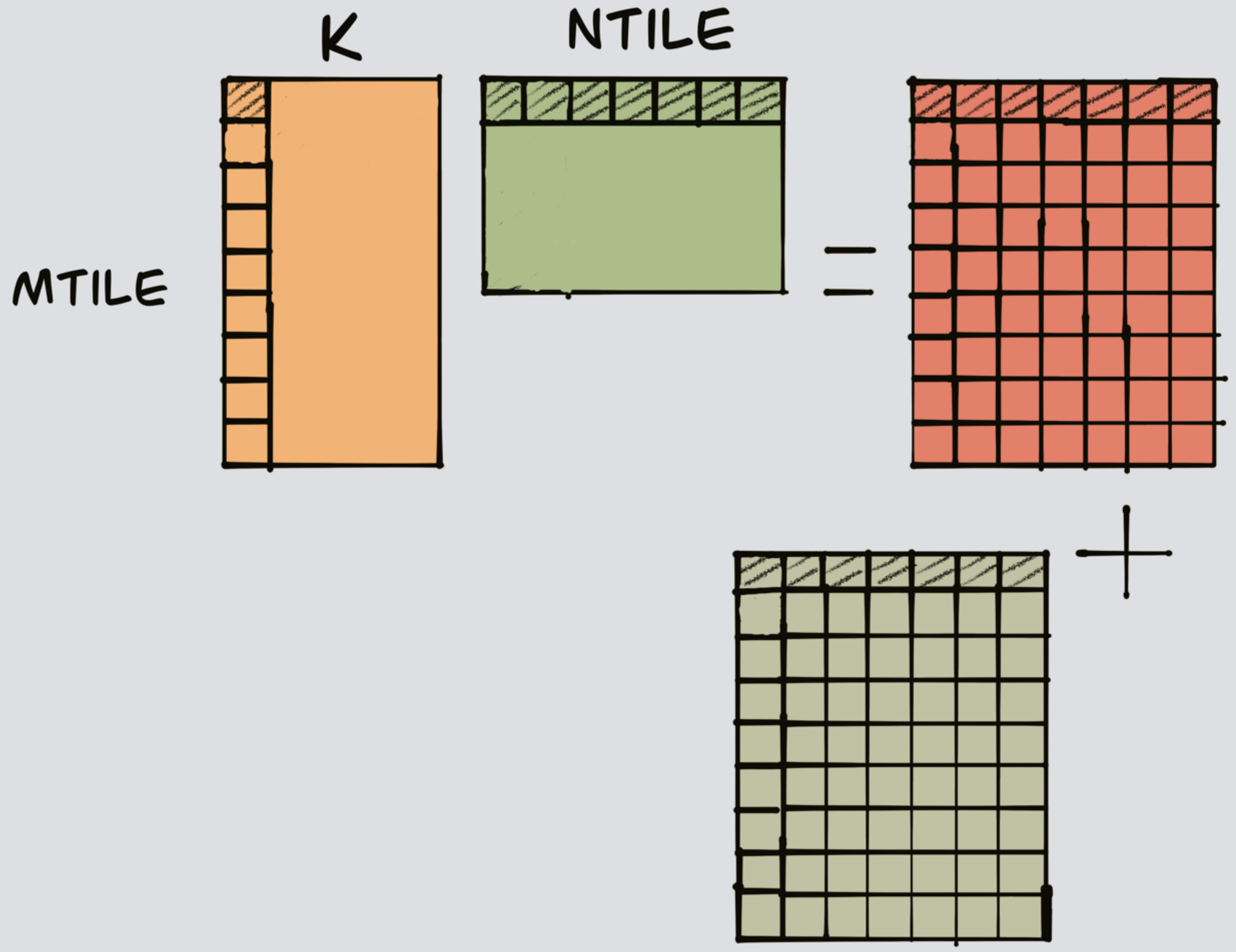
Overview of GPU code generation pipeline

- 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

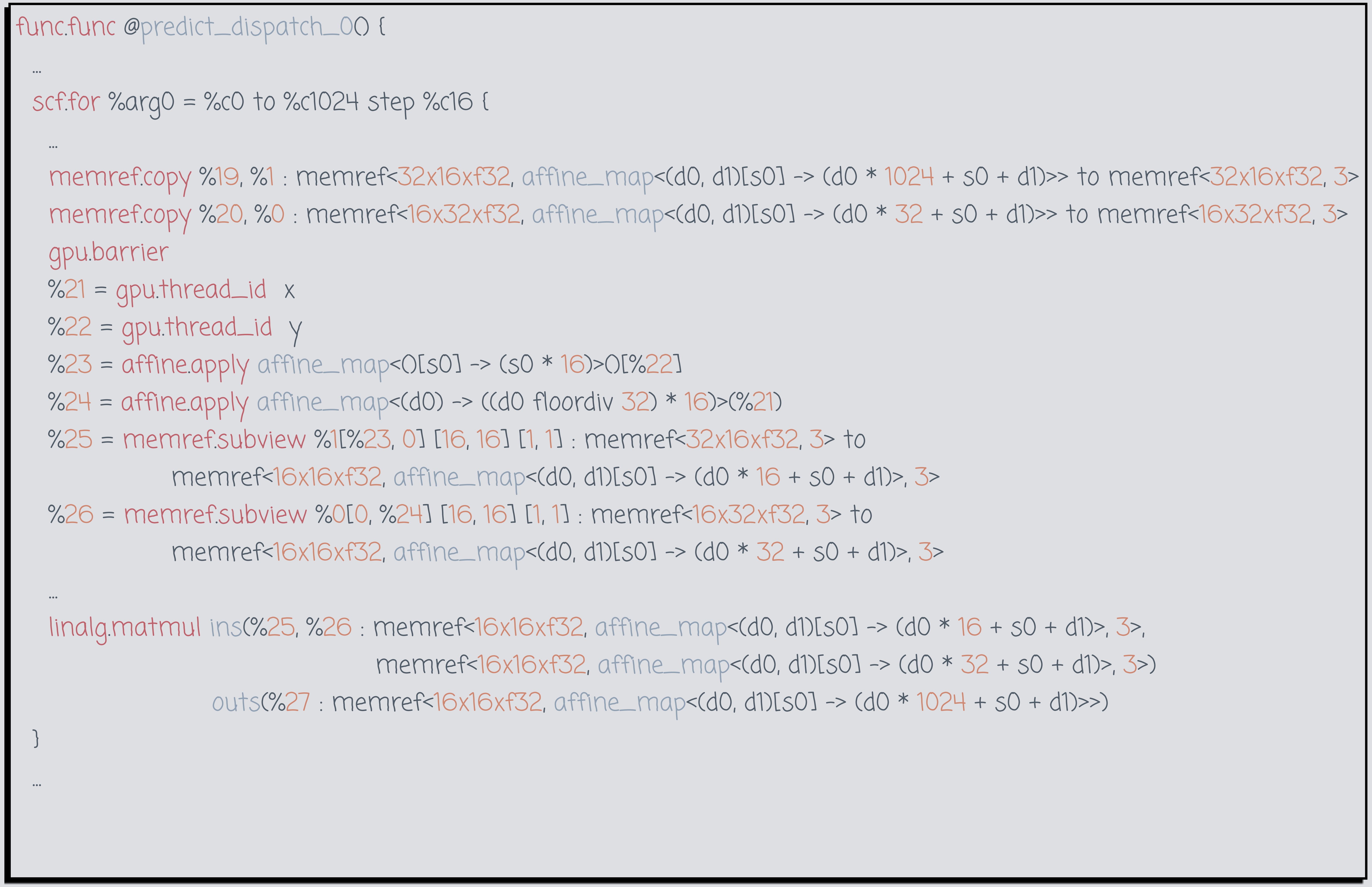

Create Dispatch Regions
Tile And Distribute to Workgroups
Bufferize
Tile And Distribute to Warps
GPU Optimizations
Tensor Core Vectorization

Tile and Distribute to Warps

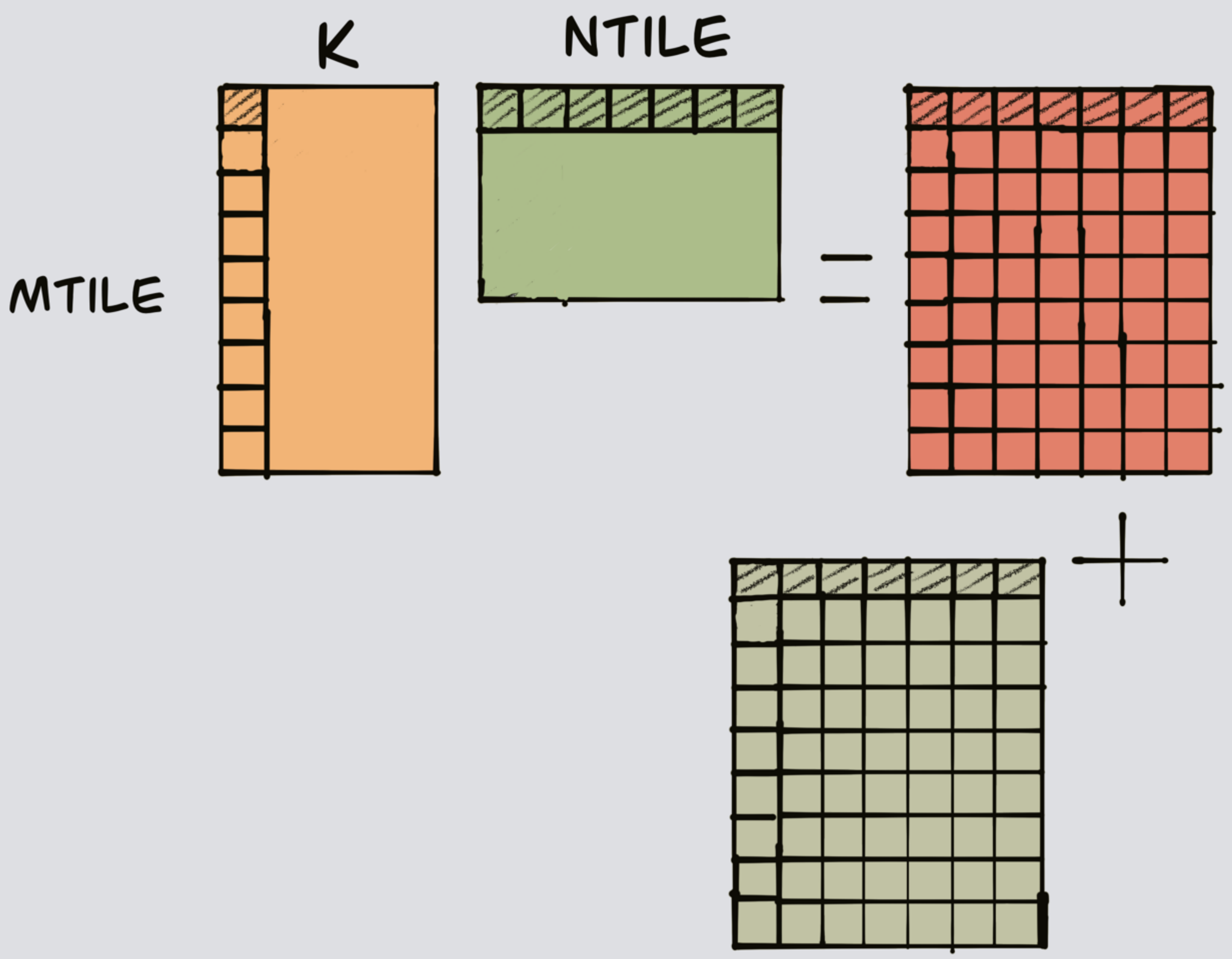
- Tile and distribute to warps
  - After bufferization, tile the reduction dimension
  - Copy subviews of input memrefs to shared memory (workgroup memory) on the GPU prior to computation
    - Start introducing SIMD characteristics by introducing memref.copy (mapping to shared memory done later downstream)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



Tile and Distribute to Warps



```
func.func @predict_dispatch_0() {
  ...
  scf.for %arg0 = %c0 to %c1024 step %c16 {
    ...
    memref.copy %19, %1 : memref<32x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>> to memref<32x16xf32, 3>
    memref.copy %20, %0 : memref<16x32xf32, affine_map<(d0, d1)[s0] -> (d0 * 32 + s0 + d1)>> to memref<16x32xf32, 3>
    gpu.barrier
    %21 = gpu.thread_id  x
    %22 = gpu.thread_id  y
    %23 = affine.apply affine_map<()[s0] -> (s0 * 16)>()[%22]
    %24 = affine.apply affine_map<(d0) -> ((d0 floordiv 32) * 16)>(%21)
    %25 = memref.subview %1[%23, 0] [16, 16] [1, 1] : memref<32x16xf32, 3> to
               memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 16 + s0 + d1)>, 3>
    %26 = memref.subview %0[0, %24] [16, 16] [1, 1] : memref<16x32xf32, 3> to
               memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 32 + s0 + d1)>, 3>
    ...
    linalg.matmul ins(%25, %26 : memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 16 + s0 + d1)>, 3>,
                                 memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 32 + s0 + d1)>, 3>)
                  outs(%27 : memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>)
  }
  ...
```

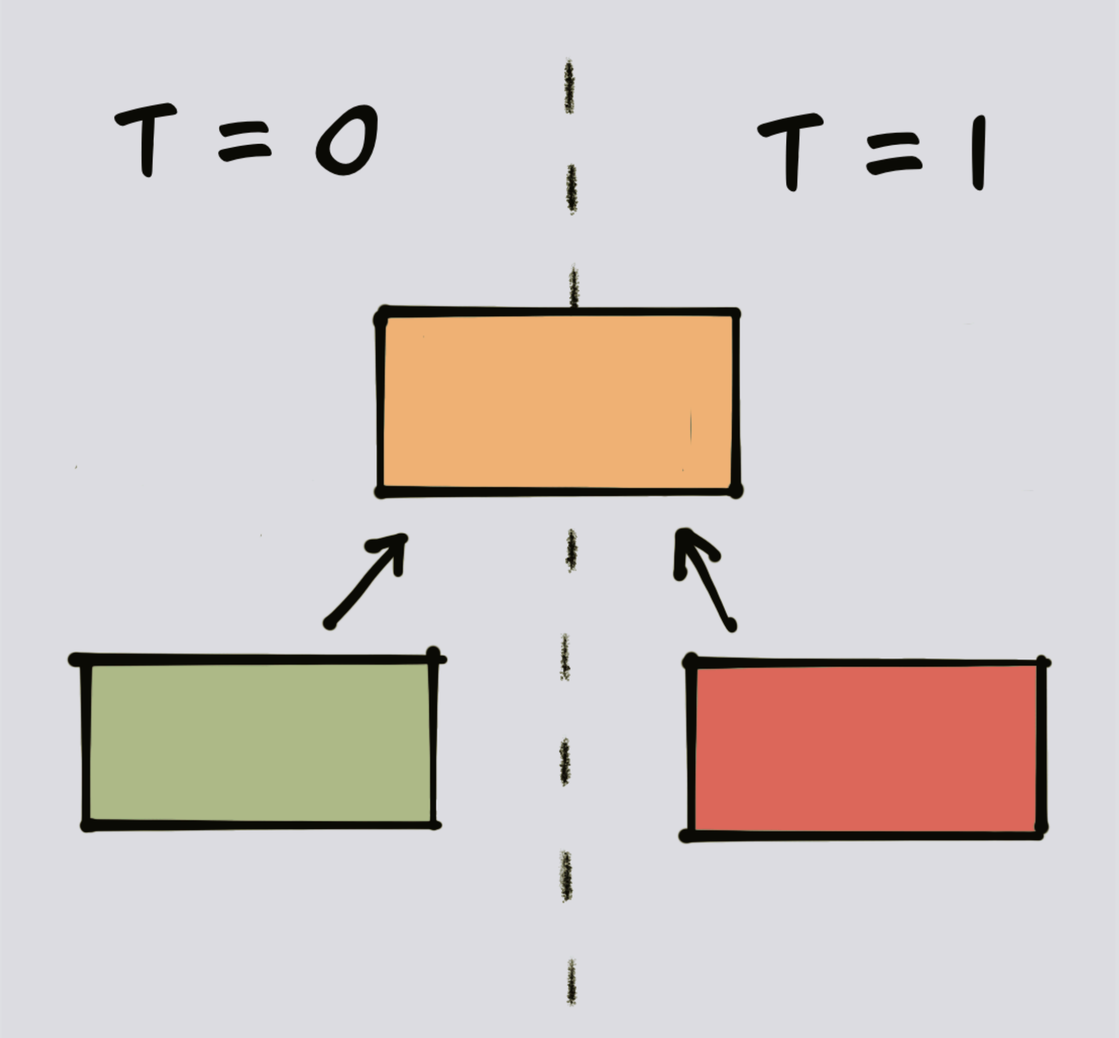


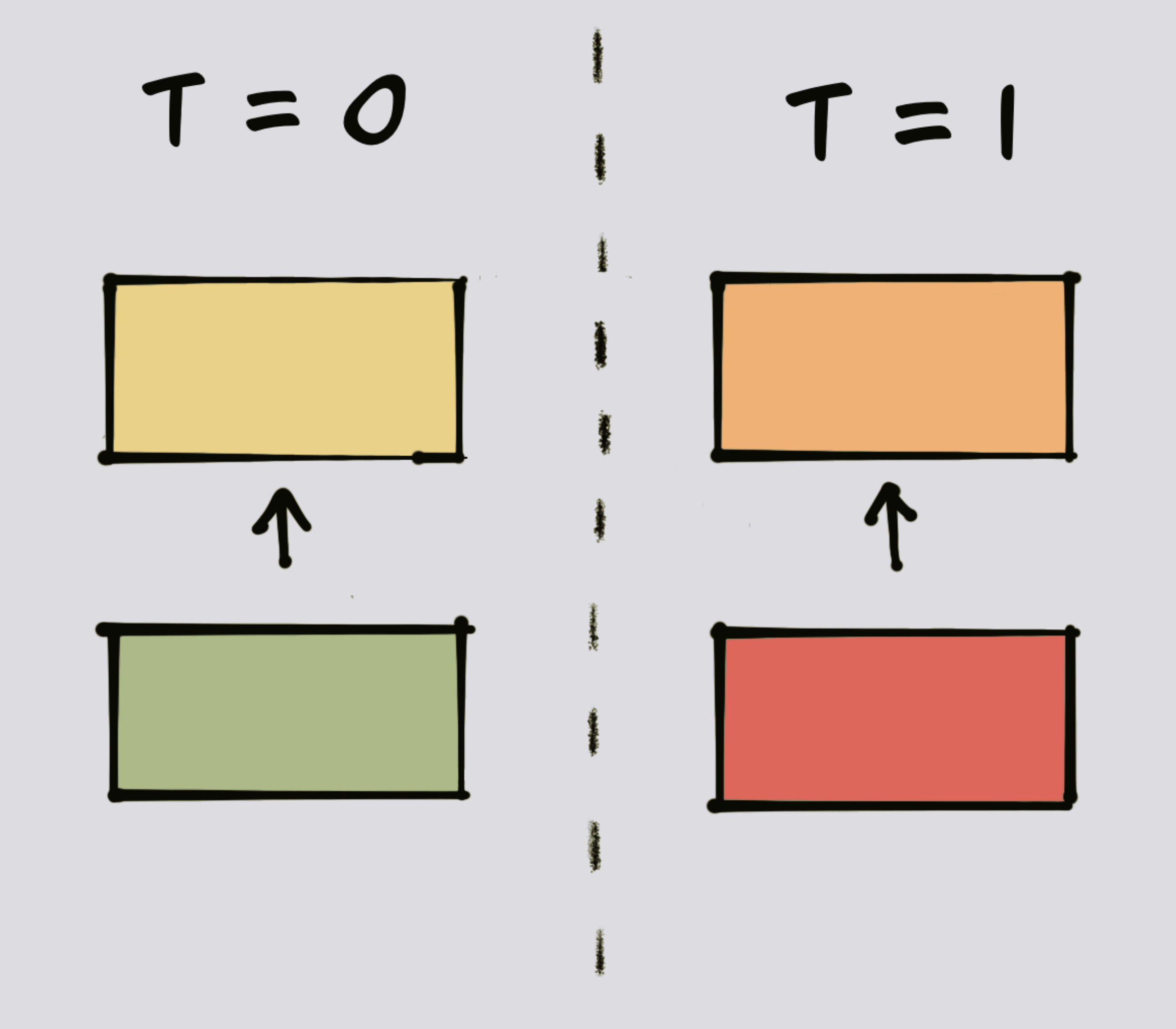
Multi-Buffering

- 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

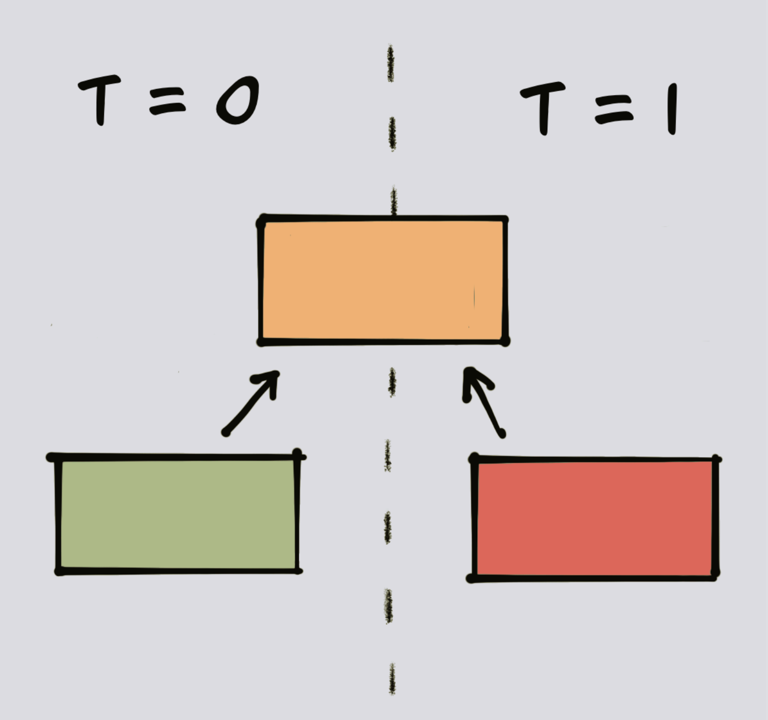
- Number of copies determined by desired stages of pipeline



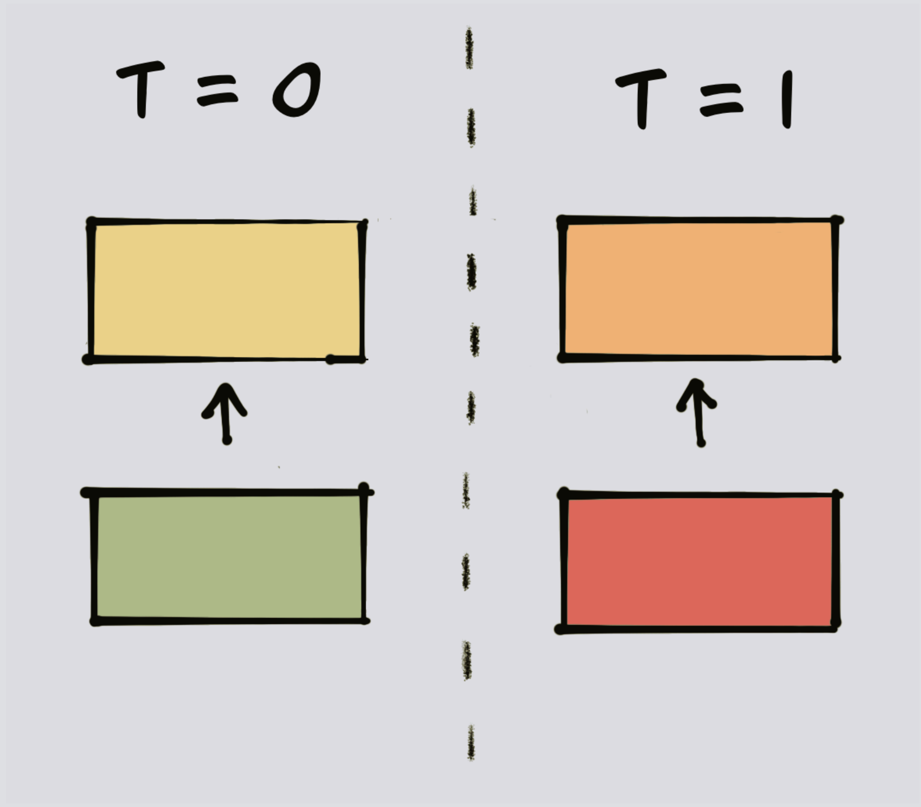


Multi-Buffering

```
func.func @predict_dispatch_0() {
  ...
  %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.func @predict_dispatch_0() {
  ...
  %0 = memref.alloc() : memref<4x16x32xf32, 3>
  ...
   %22 = memref.subview %0[%21, 0, 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, %20 : memref<32x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>> to
                          memref<32x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 16 + s0 + d1)>, 3>
}
```

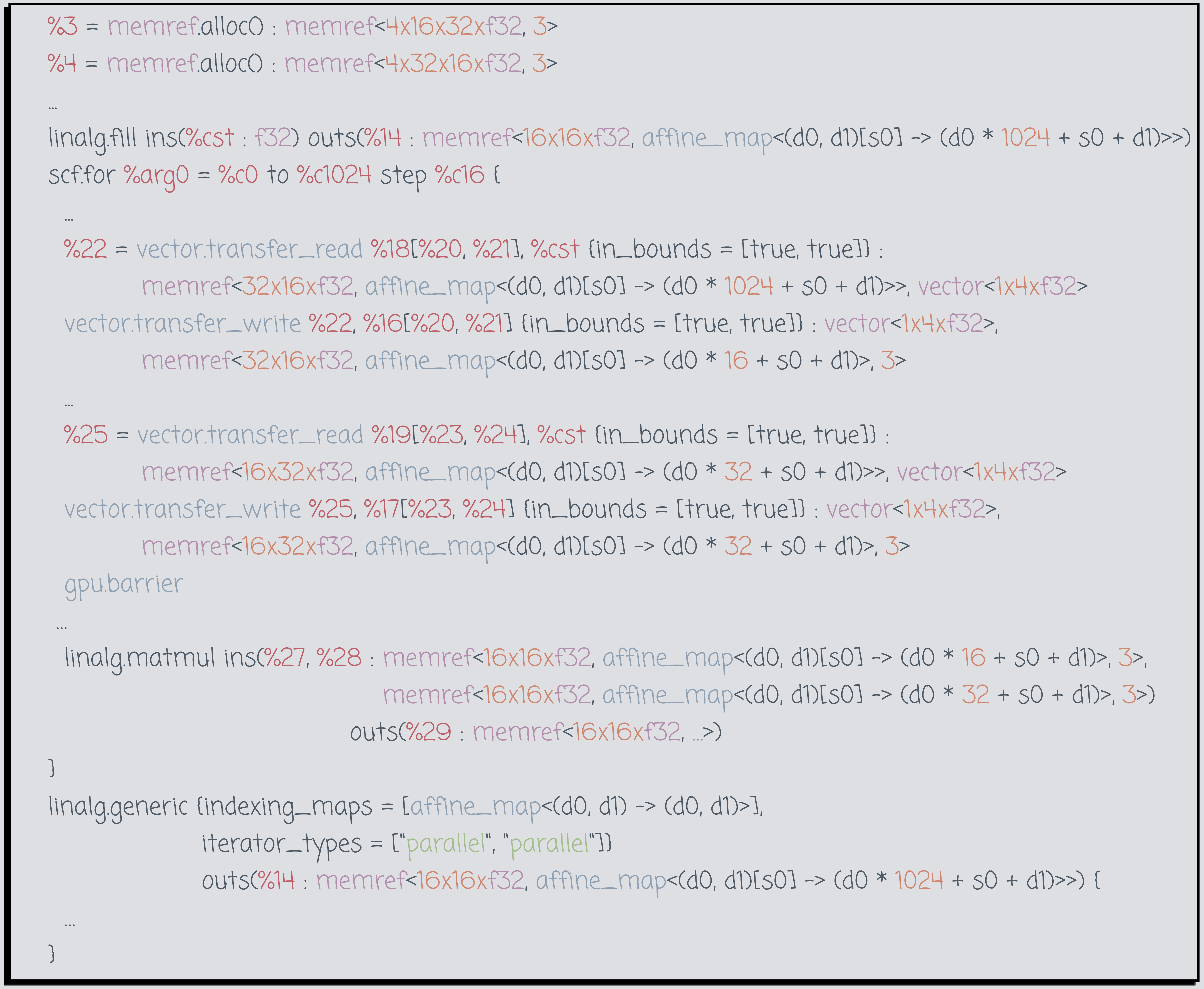


Vectorize Shared Memory Copies

- 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.

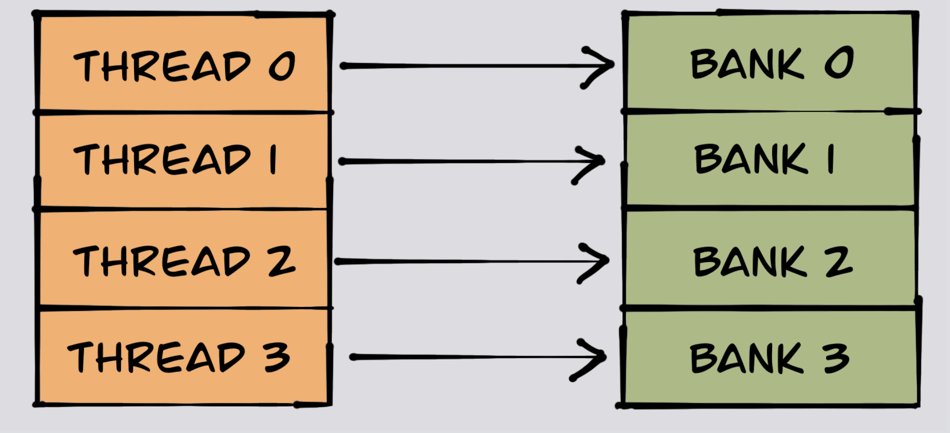
- Unroll the vector transfer read and writes

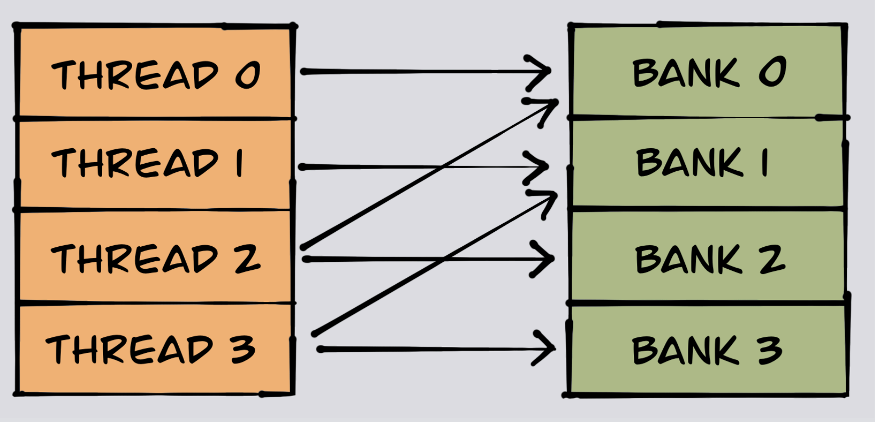


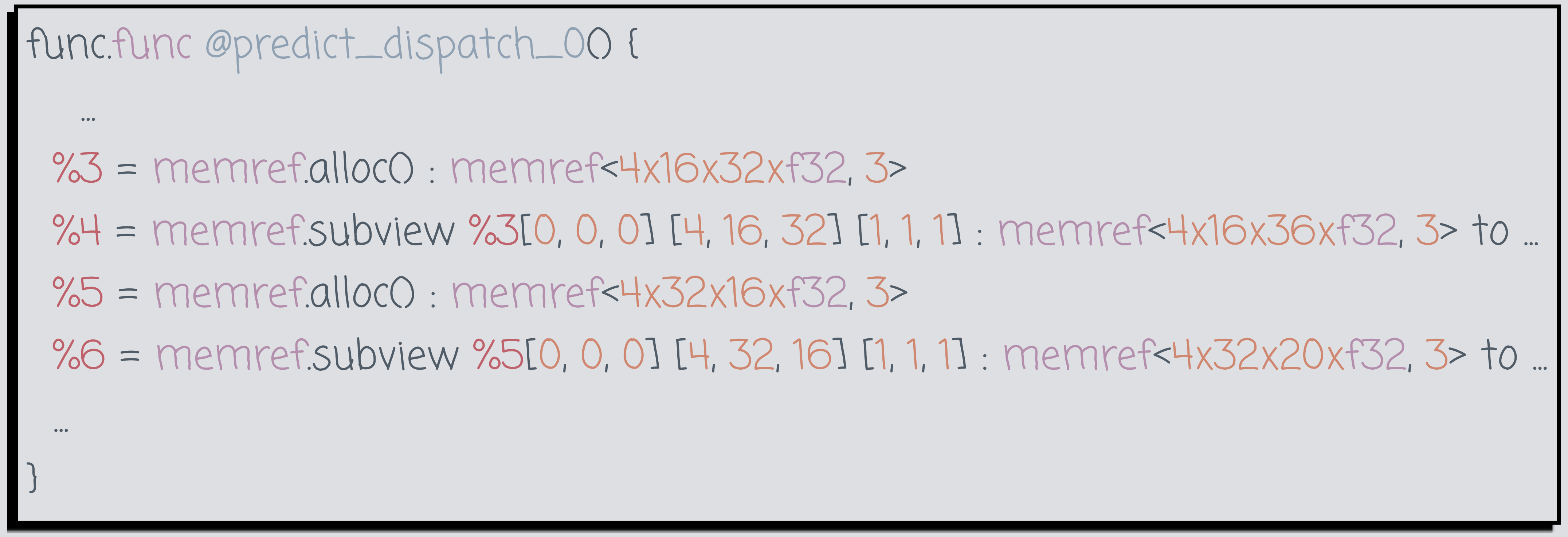
```
%3 = memref.alloc() : memref<4x16x32xf32, 3>
%4 = memref.alloc() : memref<4x32x16xf32, 3>
...
linalg.fill ins(%cst : f32) outs(%14 : memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>)
scf.for %arg0 = %c0 to %c1024 step %c16 {
  ...
  %22 = vector.transfer_read %18[%20, %21], %cst {in_bounds = [true, true]} :
          memref<32x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>, vector<1x4xf32>
  vector.transfer_write %22, %16[%20, %21] {in_bounds = [true, true]} : vector<1x4xf32>,
          memref<32x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 16 + s0 + d1)>, 3>
  ...
  %25 = vector.transfer_read %19[%23, %24], %cst {in_bounds = [true, true]} :
          memref<16x32xf32, affine_map<(d0, d1)[s0] -> (d0 * 32 + s0 + d1)>>, vector<1x4xf32>
  vector.transfer_write %25, %17[%23, %24] {in_bounds = [true, true]} : vector<1x4xf32>,
          memref<16x32xf32, affine_map<(d0, d1)[s0] -> (d0 * 32 + s0 + d1)>, 3>
  gpu.barrier
 ...
  linalg.matmul ins(%27, %28 : memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 16 + s0 + d1)>, 3>,
                               memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 32 + s0 + d1)>, 3>)
                             outs(%29 : memref<16x16xf32, ...>)
}
linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>],
                iterator_types = ["parallel", "parallel"]}
                outs(%14 : memref<16x16xf32, affine_map<(d0, d1)[s0] -> (d0 * 1024 + s0 + d1)>>) {
  ...
}
```

Reduce Bank Conflicts

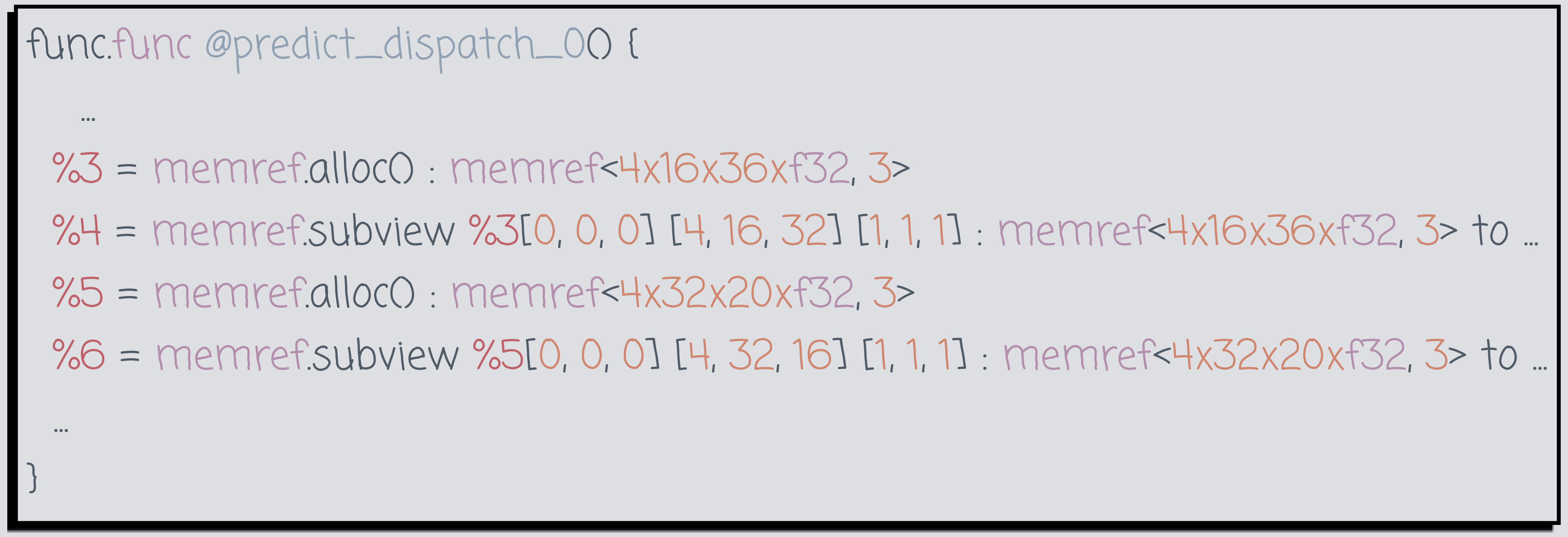
- Shared memory is arranged in banks (32 banks each of width 4 bytes)
- Each thread in a warp can access shared memory in parallel
- When 2 or more threads in a warp access 4 byte words in the same bank, results in serialized accesses and hence a reduction in overall bandwidth
- Pad inner dimensions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







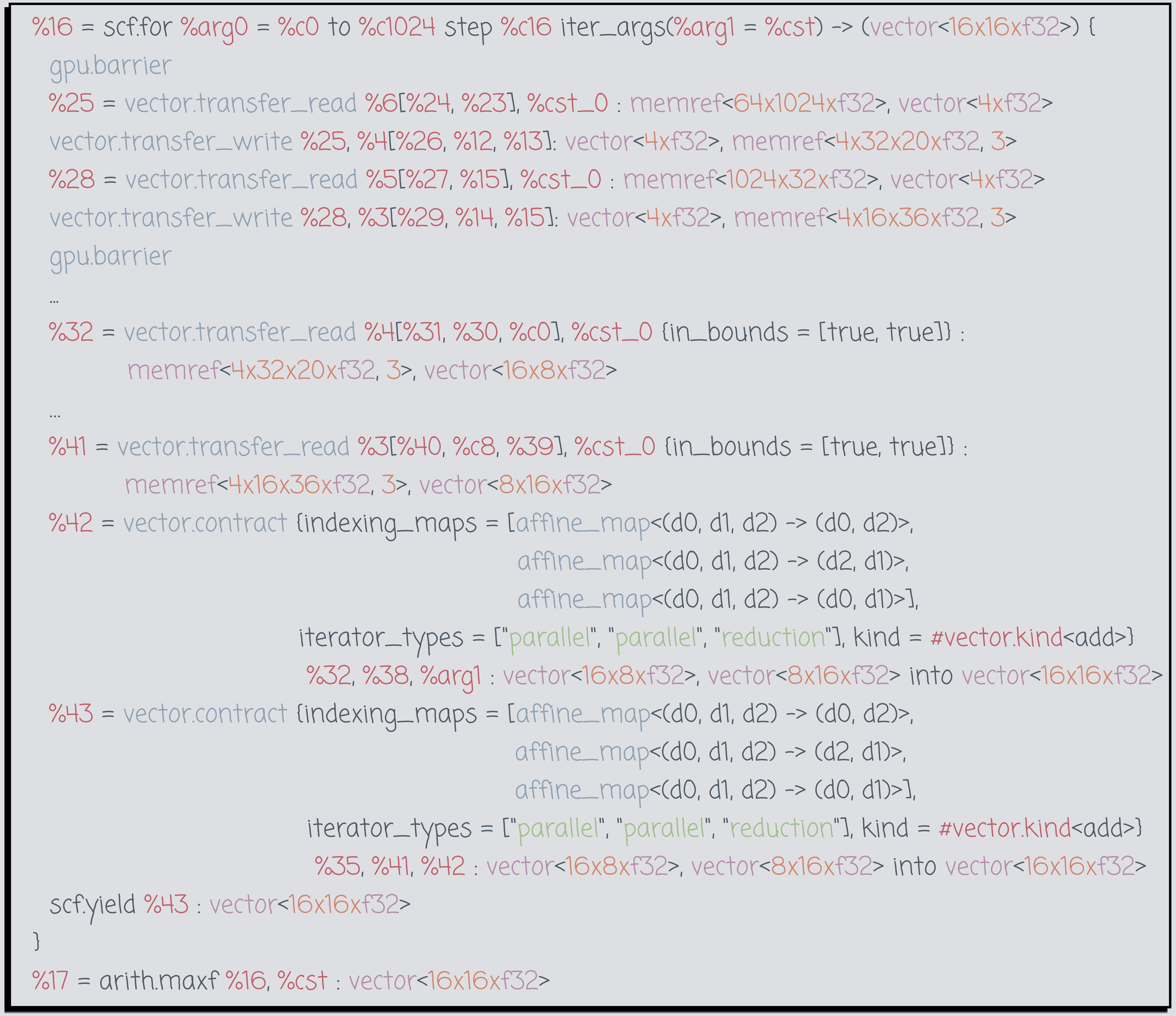
```
func.func @predict_dispatch_0() {
    ...
  %3 = memref.alloc() : memref<4x16x32xf32, 3>
  %4 = memref.subview %3[0, 0, 0] [4, 16, 32] [1, 1, 1] : memref<4x16x36xf32, 3> to ...
  %5 = memref.alloc() : memref<4x32x16xf32, 3>
  %6 = memref.subview %5[0, 0, 0] [4, 32, 16] [1, 1, 1] : memref<4x32x20xf32, 3> to ...
  ...
}
```



```
func.func @predict_dispatch_0() {
    ...
  %3 = memref.alloc() : memref<4x16x36xf32, 3>
  %4 = memref.subview %3[0, 0, 0] [4, 16, 32] [1, 1, 1] : memref<4x16x36xf32, 3> to ...
  %5 = memref.alloc() : memref<4x32x20xf32, 3>
  %6 = memref.subview %5[0, 0, 0] [4, 32, 16] [1, 1, 1] : memref<4x32x20xf32, 3> to ...
  ...
}
```

Tensor Core Vectorization

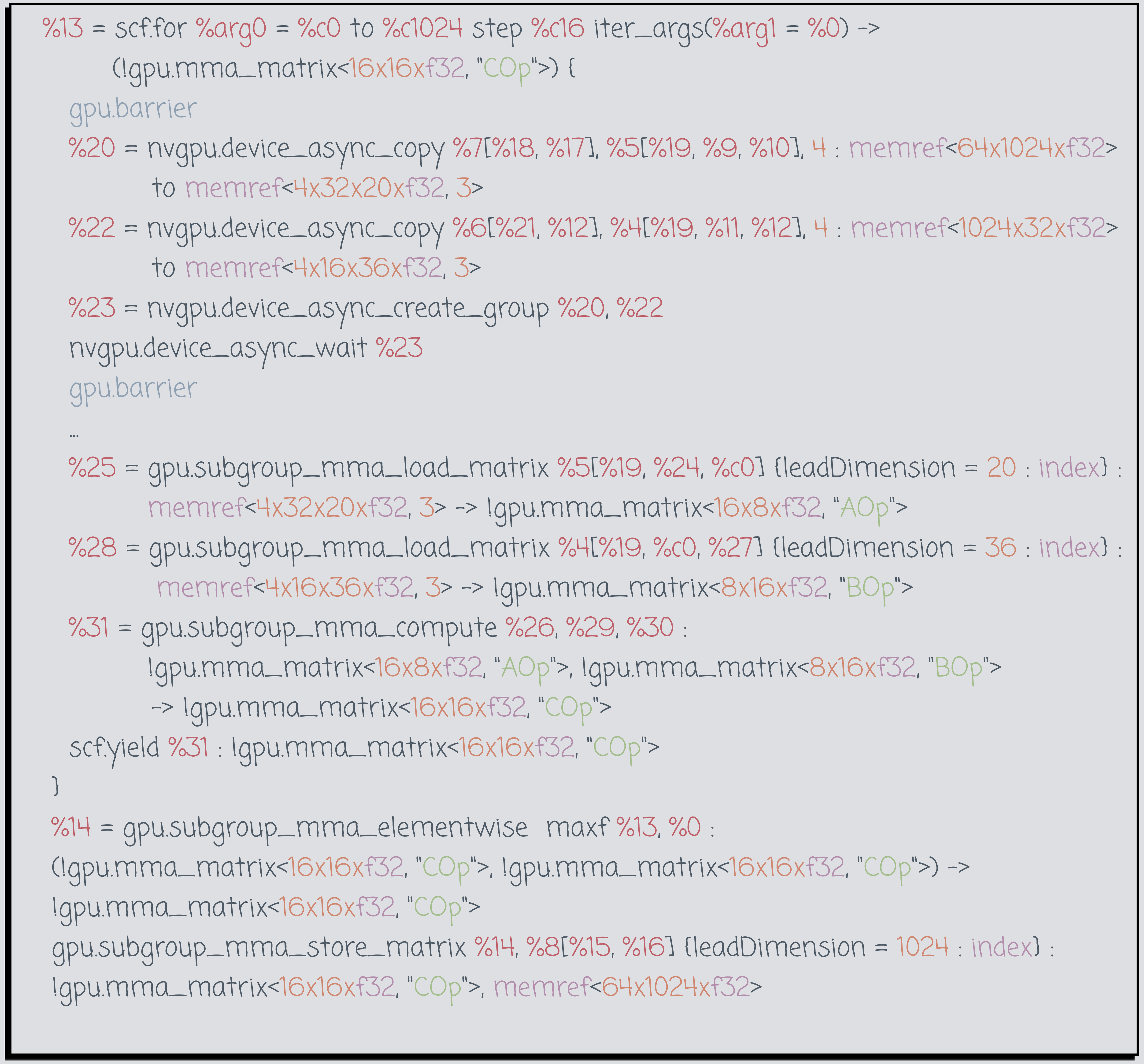
- Replace all linalg ops with vector equivalents
- Lower linalg.matmul to vector.contract with vector.transfer_read/write
- Lower linalg.generic to arith ops with vector.transfer_read/write
- Optimize vector transfers by removing redundant ops
- Vector transfer/contract size determined by tensor core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



```
%16 = scf.for %arg0 = %c0 to %c1024 step %c16 iter_args(%arg1 = %cst) -> (vector<16x16xf32>) {
  gpu.barrier
  %25 = vector.transfer_read %6[%24, %23], %cst_0 : memref<64x1024xf32>, vector<4xf32>
  vector.transfer_write %25, %4[%26, %12, %13]: vector<4xf32>, memref<4x32x20xf32, 3>
  %28 = vector.transfer_read %5[%27, %15], %cst_0 : memref<1024x32xf32>, vector<4xf32>
  vector.transfer_write %28, %3[%29, %14, %15]: vector<4xf32>, memref<4x16x36xf32, 3>
  gpu.barrier
  ...
  %32 = vector.transfer_read %4[%31, %30, %c0], %cst_0 {in_bounds = [true, true]} :
          memref<4x32x20xf32, 3>, vector<16x8xf32>
  ...
  %41 = vector.transfer_read %3[%40, %c8, %39], %cst_0 {in_bounds = [true, true]} :
          memref<4x16x36xf32, 3>, vector<8x16xf32>
  %42 = vector.contract {indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                                         affine_map<(d0, d1, d2) -> (d2, d1)>,
                                         affine_map<(d0, d1, d2) -> (d0, d1)>],
                         iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>}
                          %32, %38, %arg1 : vector<16x8xf32>, vector<8x16xf32> into vector<16x16xf32>
  %43 = vector.contract {indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                                         affine_map<(d0, d1, d2) -> (d2, d1)>,
                                         affine_map<(d0, d1, d2) -> (d0, d1)>],
                         iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>}
                          %35, %41, %42 : vector<16x8xf32>, vector<8x16xf32> into vector<16x16xf32>
  scf.yield %43 : vector<16x16xf32>
}
%17 = arith.maxf %16, %cst : vector<16x16xf32>
```

Convert to GPU Dialect

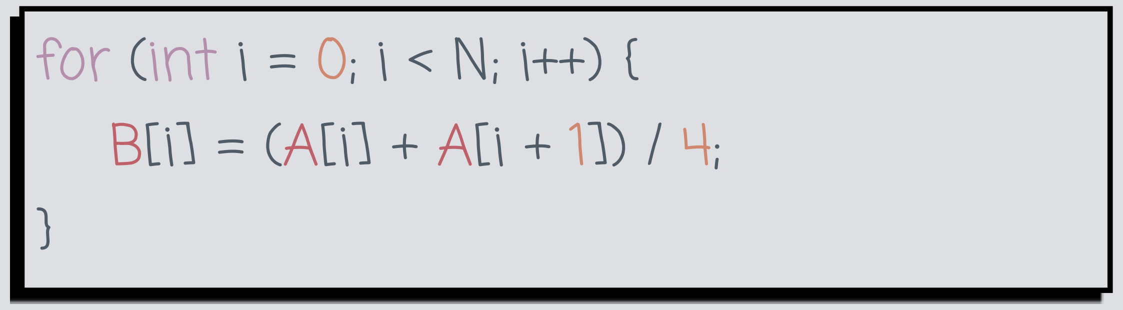
- Convert copies to shared memory to async copies (nvgpu.device_async_copy, nvgpu.device_async_create_group, nvgpu.device_async_wait)
- Adds gpu.subgroup_mma_load_matrix which loads a matrix using all threads in a subgroup
- Lowers vector.contract to gpu.subgroup_mma_compute which performs matrix-multiply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



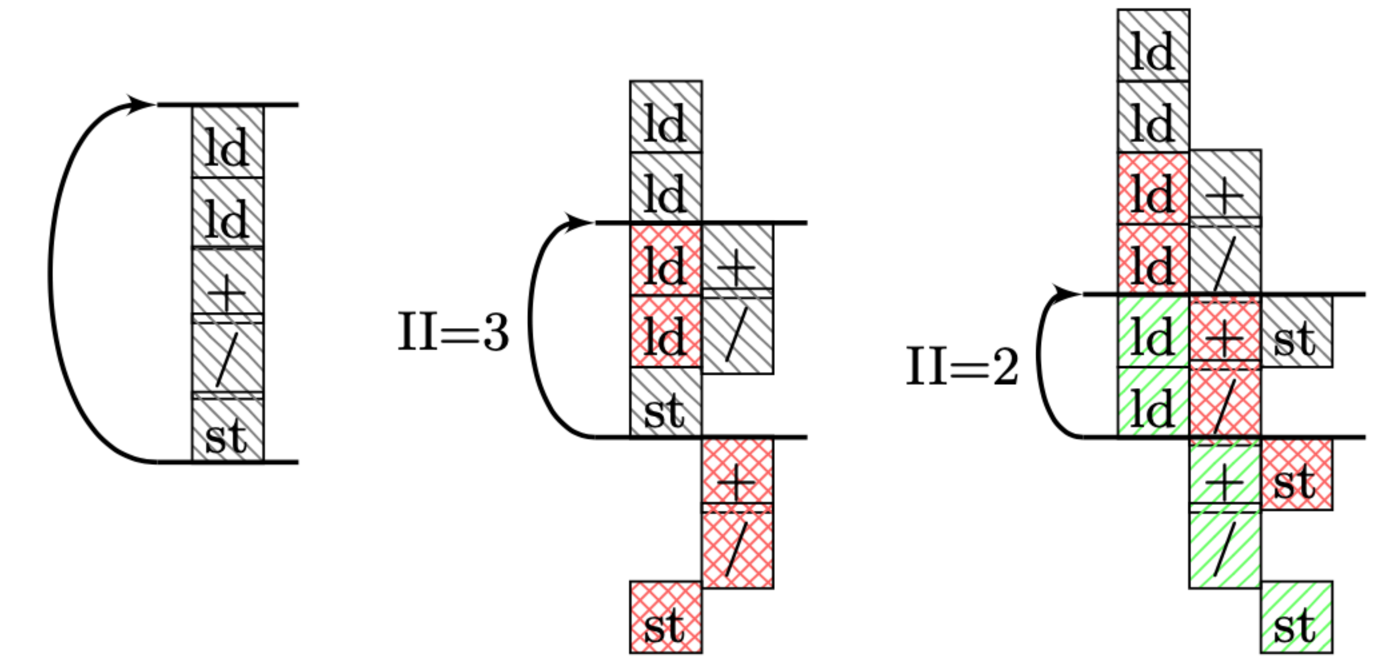
```
%13 = scf.for %arg0 = %c0 to %c1024 step %c16 iter_args(%arg1 = %0) ->
      (!gpu.mma_matrix<16x16xf32, "COp">) {
  gpu.barrier
  %20 = nvgpu.device_async_copy %7[%18, %17], %5[%19, %9, %10], 4 : memref<64x1024xf32>
          to memref<4x32x20xf32, 3>
  %22 = nvgpu.device_async_copy %6[%21, %12], %4[%19, %11, %12], 4 : memref<1024x32xf32>
          to memref<4x16x36xf32, 3>
  %23 = nvgpu.device_async_create_group %20, %22
  nvgpu.device_async_wait %23
  gpu.barrier
  ...
  %25 = gpu.subgroup_mma_load_matrix %5[%19, %24, %c0] {leadDimension = 20 : index} :
          memref<4x32x20xf32, 3> -> !gpu.mma_matrix<16x8xf32, "AOp">
  %28 = gpu.subgroup_mma_load_matrix %4[%19, %c0, %27] {leadDimension = 36 : index} :
           memref<4x16x36xf32, 3> -> !gpu.mma_matrix<8x16xf32, "BOp">
  %31 = gpu.subgroup_mma_compute %26, %29, %30 :
          !gpu.mma_matrix<16x8xf32, "AOp">, !gpu.mma_matrix<8x16xf32, "BOp">
          -> !gpu.mma_matrix<16x16xf32, "COp">
  scf.yield %31 : !gpu.mma_matrix<16x16xf32, "COp">
}
%14 = gpu.subgroup_mma_elementwise  maxf %13, %0 :
(!gpu.mma_matrix<16x16xf32, "COp">, !gpu.mma_matrix<16x16xf32, "COp">) ->
!gpu.mma_matrix<16x16xf32, "COp">
gpu.subgroup_mma_store_matrix %14, %8[%15, %16] {leadDimension = 1024 : index} :
!gpu.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



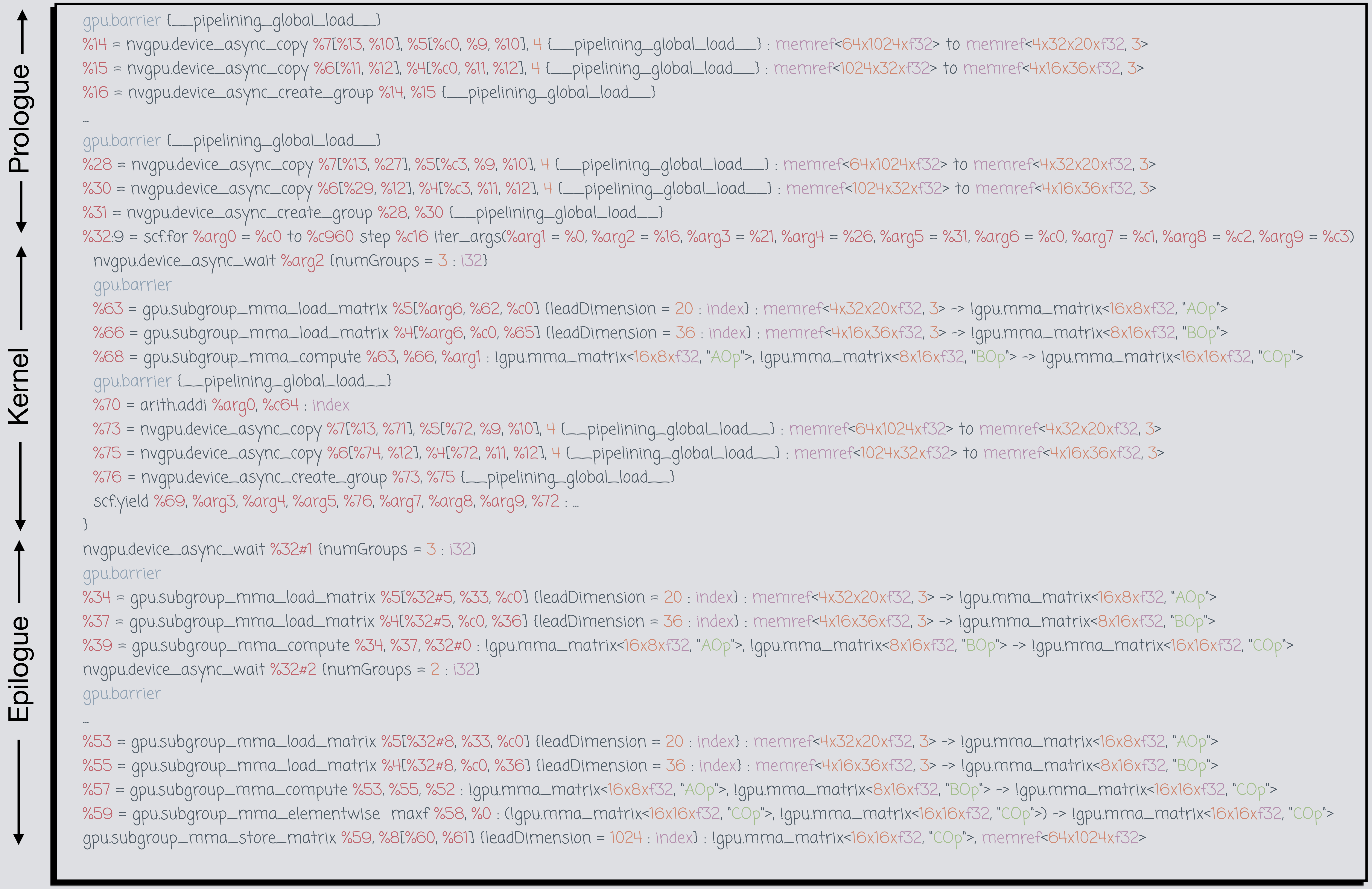
```
for (int i = 0; i < N; i++) {
    B[i] = (A[i] + A[i + 1]) / 4;
}
```



(a) Original (b) Single load-store (c) Dual load-store

GPU Pipelining

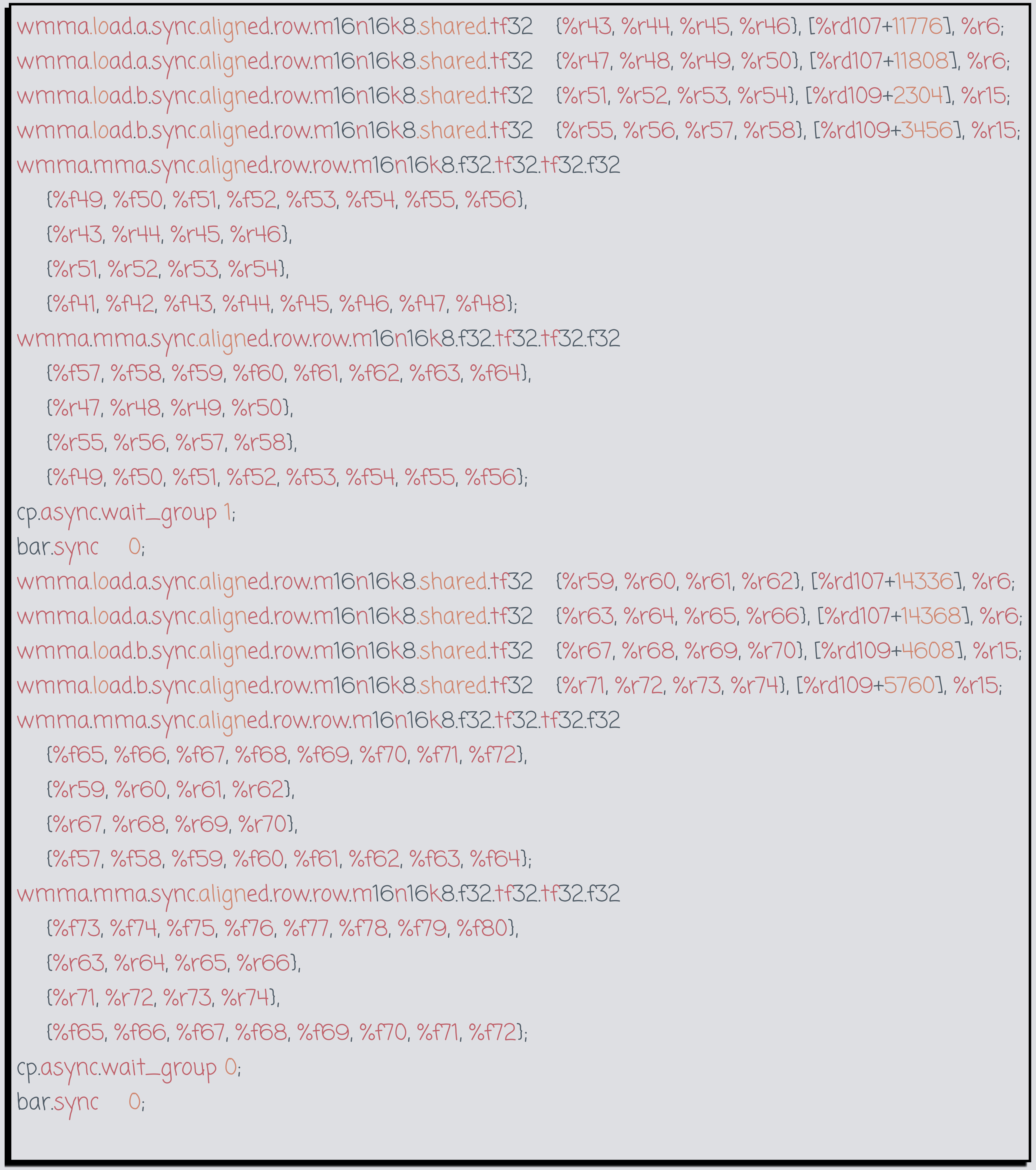
- Prologue contains async copies to shared memory
- Kernel contains mma load matrix and compute and ends with async copies
- Epilogue contains remaining computation


Prologue
Kernel
Epilogue

```
gpu.barrier {__pipelining_global_load__}
%14 = nvgpu.device_async_copy %7[%13, %10], %5[%c0, %9, %10], 4 {__pipelining_global_load__} : memref<64x1024xf32> to memref<4x32x20xf32, 3>
%15 = nvgpu.device_async_copy %6[%11, %12], %4[%c0, %11, %12], 4 {__pipelining_global_load__} : memref<1024x32xf32> to memref<4x16x36xf32, 3>
%16 = nvgpu.device_async_create_group %14, %15 {__pipelining_global_load__}
...
gpu.barrier {__pipelining_global_load__}
%28 = nvgpu.device_async_copy %7[%13, %27], %5[%c3, %9, %10], 4 {__pipelining_global_load__} : memref<64x1024xf32> to memref<4x32x20xf32, 3>
%30 = nvgpu.device_async_copy %6[%29, %12], %4[%c3, %11, %12], 4 {__pipelining_global_load__} : memref<1024x32xf32> to memref<4x16x36xf32, 3>
%31 = nvgpu.device_async_create_group %28, %30 {__pipelining_global_load__}
%32:9 = scf.for %arg0 = %c0 to %c960 step %c16 iter_args(%arg1 = %0, %arg2 = %16, %arg3 = %21, %arg4 = %26, %arg5 = %31, %arg6 = %c0, %arg7 = %c1, %arg8 = %c2, %arg9 = %c3)
  nvgpu.device_async_wait %arg2 {numGroups = 3 : i32}
  gpu.barrier
  %63 = gpu.subgroup_mma_load_matrix %5[%arg6, %62, %c0] {leadDimension = 20 : index} : memref<4x32x20xf32, 3> -> !gpu.mma_matrix<16x8xf32, "AOp">
  %66 = gpu.subgroup_mma_load_matrix %4[%arg6, %c0, %65] {leadDimension = 36 : index} : memref<4x16x36xf32, 3> -> !gpu.mma_matrix<8x16xf32, "BOp">
  %68 = gpu.subgroup_mma_compute %63, %66, %arg1 : !gpu.mma_matrix<16x8xf32, "AOp">, !gpu.mma_matrix<8x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
  gpu.barrier {__pipelining_global_load__}
  %70 = arith.addi %arg0, %c64 : index
  %73 = nvgpu.device_async_copy %7[%13, %71], %5[%72, %9, %10], 4 {__pipelining_global_load__} : memref<64x1024xf32> to memref<4x32x20xf32, 3>
  %75 = nvgpu.device_async_copy %6[%74, %12], %4[%72, %11, %12], 4 {__pipelining_global_load__} : memref<1024x32xf32> to memref<4x16x36xf32, 3>
  %76 = nvgpu.device_async_create_group %73, %75 {__pipelining_global_load__}
  scf.yield %69, %arg3, %arg4, %arg5, %76, %arg7, %arg8, %arg9, %72 : ...
}
nvgpu.device_async_wait %32#1 {numGroups = 3 : i32}
gpu.barrier
%34 = gpu.subgroup_mma_load_matrix %5[%32#5, %33, %c0] {leadDimension = 20 : index} : memref<4x32x20xf32, 3> -> !gpu.mma_matrix<16x8xf32, "AOp">
%37 = gpu.subgroup_mma_load_matrix %4[%32#5, %c0, %36] {leadDimension = 36 : index} : memref<4x16x36xf32, 3> -> !gpu.mma_matrix<8x16xf32, "BOp">
%39 = gpu.subgroup_mma_compute %34, %37, %32#0 : !gpu.mma_matrix<16x8xf32, "AOp">, !gpu.mma_matrix<8x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
nvgpu.device_async_wait %32#2 {numGroups = 2 : i32}
gpu.barrier
...
%53 = gpu.subgroup_mma_load_matrix %5[%32#8, %33, %c0] {leadDimension = 20 : index} : memref<4x32x20xf32, 3> -> !gpu.mma_matrix<16x8xf32, "AOp">
%55 = gpu.subgroup_mma_load_matrix %4[%32#8, %c0, %36] {leadDimension = 36 : index} : memref<4x16x36xf32, 3> -> !gpu.mma_matrix<8x16xf32, "BOp">
%57 = gpu.subgroup_mma_compute %53, %55, %52 : !gpu.mma_matrix<16x8xf32, "AOp">, !gpu.mma_matrix<8x16xf32, "BOp"> -> !gpu.mma_matrix<16x16xf32, "COp">
%59 = gpu.subgroup_mma_elementwise  maxf %58, %0 : (!gpu.mma_matrix<16x16xf32, "COp">, !gpu.mma_matrix<16x16xf32, "COp">) -> !gpu.mma_matrix<16x16xf32, "COp">
gpu.subgroup_mma_store_matrix %59, %8[%60, %61] {leadDimension = 1024 : index} : !gpu.mma_matrix<16x16xf32, "COp">, memref<64x1024xf32>
```

GPU Lowering

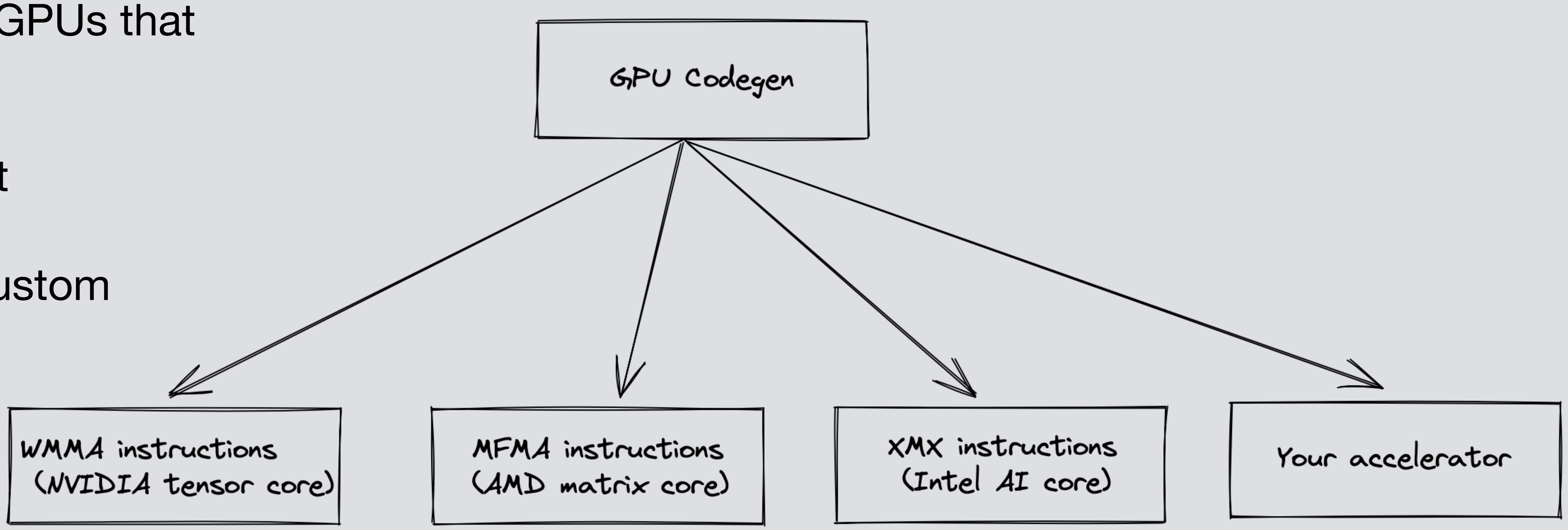
- Next, we use LLVM to lower to PTX
- Use JIT compilation to convert PTX code to native GPU machine code
- Alternatively, could use PTX assembler and convert to SASS (and then CUBIN)



```
wmma.load.a.sync.aligned.row.m16n16k8.shared.tf32   {%r43, %r44, %r45, %r46}, [%rd107+11776], %r6;
wmma.load.a.sync.aligned.row.m16n16k8.shared.tf32   {%r47, %r48, %r49, %r50}, [%rd107+11808], %r6;
wmma.load.b.sync.aligned.row.m16n16k8.shared.tf32   {%r51, %r52, %r53, %r54}, [%rd109+2304], %r15;
wmma.load.b.sync.aligned.row.m16n16k8.shared.tf32   {%r55, %r56, %r57, %r58}, [%rd109+3456], %r15;
wmma.mma.sync.aligned.row.row.m16n16k8.f32.tf32.tf32.f32
    {%f49, %f50, %f51, %f52, %f53, %f54, %f55, %f56},
    {%r43, %r44, %r45, %r46},
    {%r51, %r52, %r53, %r54},
    {%f41, %f42, %f43, %f44, %f45, %f46, %f47, %f48};
wmma.mma.sync.aligned.row.row.m16n16k8.f32.tf32.tf32.f32
    {%f57, %f58, %f59, %f60, %f61, %f62, %f63, %f64},
    {%r47, %r48, %r49, %r50},
    {%r55, %r56, %r57, %r58},
    {%f49, %f50, %f51, %f52, %f53, %f54, %f55, %f56};
cp.async.wait_group 1;
bar.sync    0;
wmma.load.a.sync.aligned.row.m16n16k8.shared.tf32   {%r59, %r60, %r61, %r62}, [%rd107+14336], %r6;
wmma.load.a.sync.aligned.row.m16n16k8.shared.tf32   {%r63, %r64, %r65, %r66}, [%rd107+14368], %r6;
wmma.load.b.sync.aligned.row.m16n16k8.shared.tf32   {%r67, %r68, %r69, %r70}, [%rd109+4608], %r15;
wmma.load.b.sync.aligned.row.m16n16k8.shared.tf32   {%r71, %r72, %r73, %r74}, [%rd109+5760], %r15;
wmma.mma.sync.aligned.row.row.m16n16k8.f32.tf32.tf32.f32
    {%f65, %f66, %f67, %f68, %f69, %f70, %f71, %f72},
    {%r59, %r60, %r61, %r62},
    {%r67, %r68, %r69, %r70},
    {%f57, %f58, %f59, %f60, %f61, %f62, %f63, %f64};
wmma.mma.sync.aligned.row.row.m16n16k8.f32.tf32.tf32.f32
    {%f73, %f74, %f75, %f76, %f77, %f78, %f79, %f80},
    {%r63, %r64, %r65, %r66},
    {%r71, %r72, %r73, %r74},
    {%f65, %f66, %f67, %f68, %f69, %f70, %f71, %f72};
cp.async.wait_group 0;
bar.sync    0;
```

Additional Targets

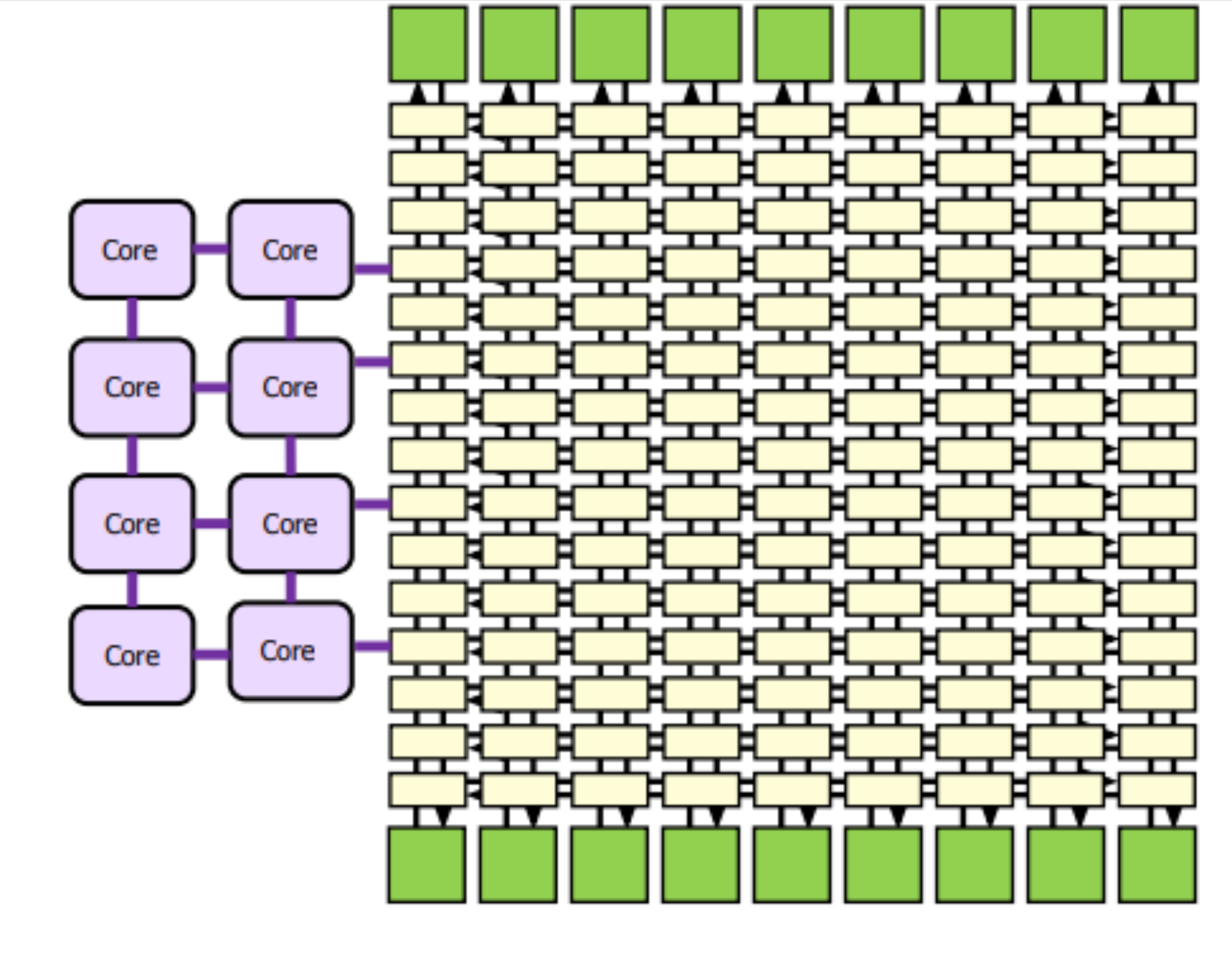
- Can reuse existing pipeline to target other GPUs that have tensor core equivalent functionality
- Adjust lowering from linalg to vector dialect
- Add appropriate dialect and lowering for custom accelerator



GPU Codegen
WMMA instructions (NVIDIA tensor core)
MFMA instructions (AMD matrix core)
XMX instructions (Intel AI core)
Your accelerator

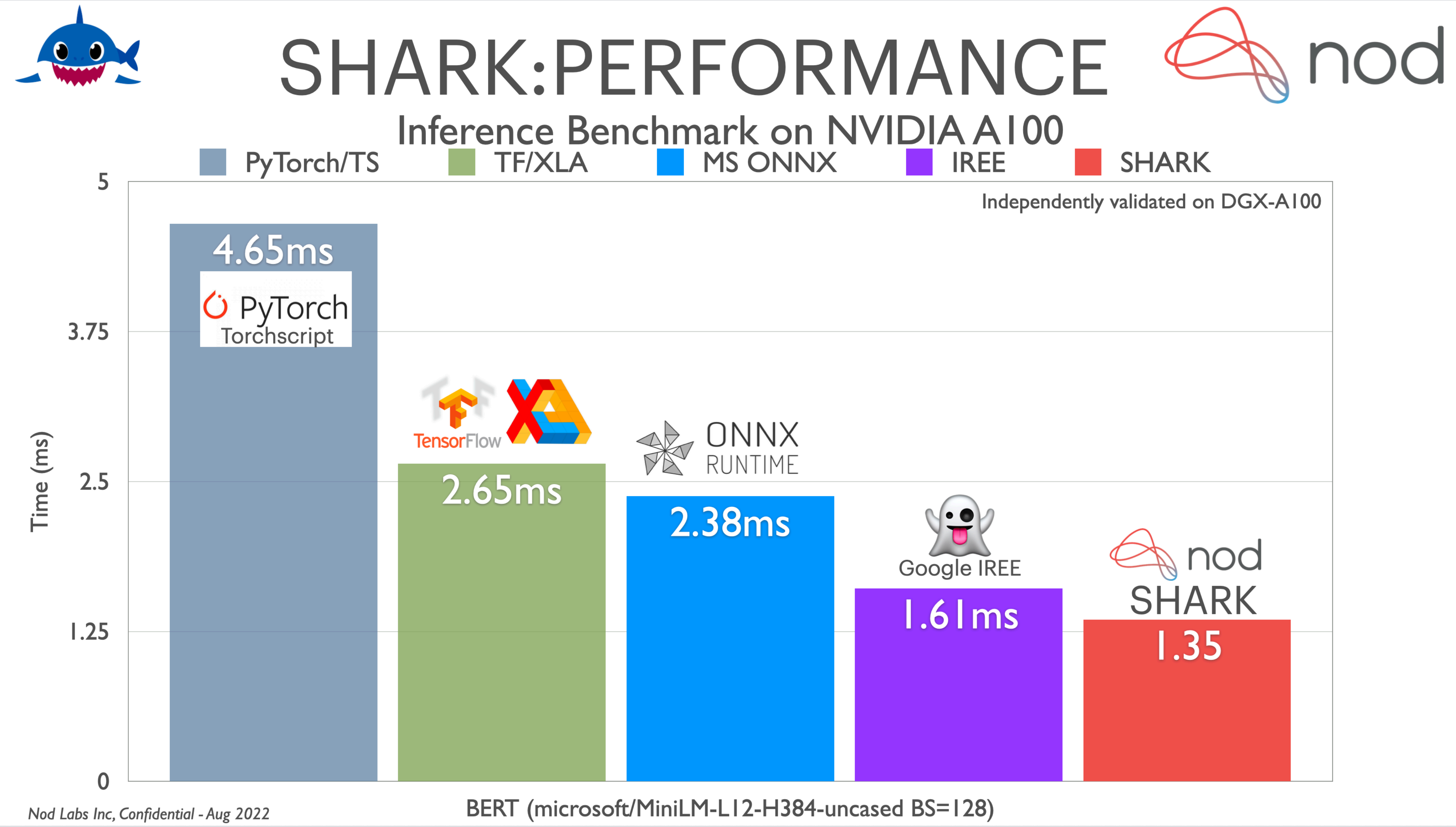
Targeting custom accelerators

- 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


SHARK:PERFORMANCE

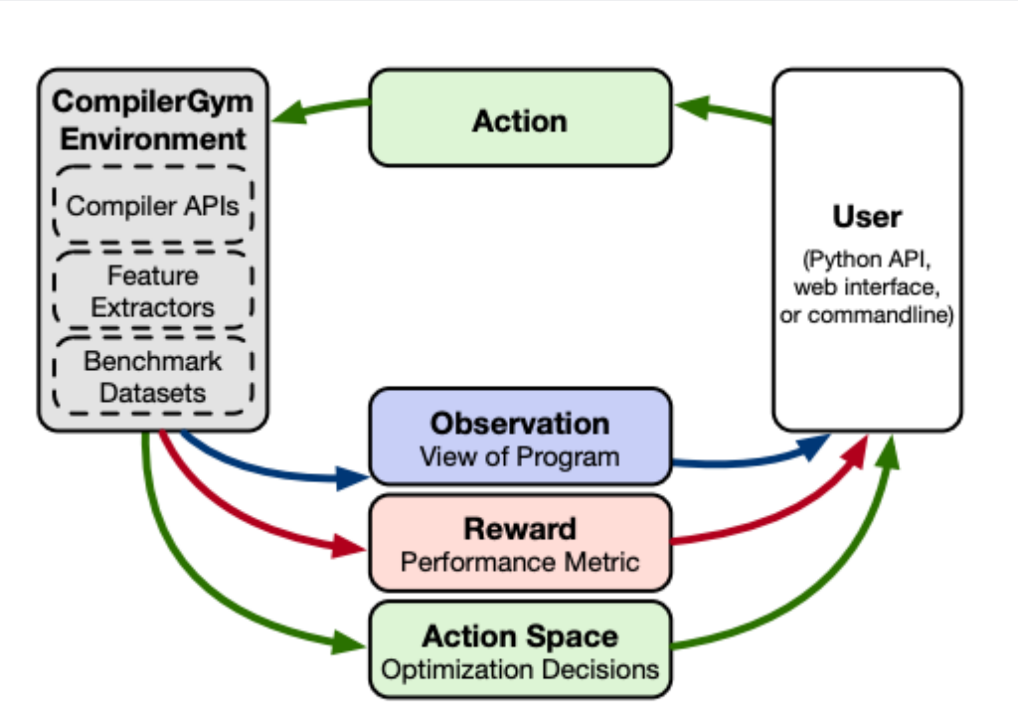
Inference Benchmark on NVIDIA A100

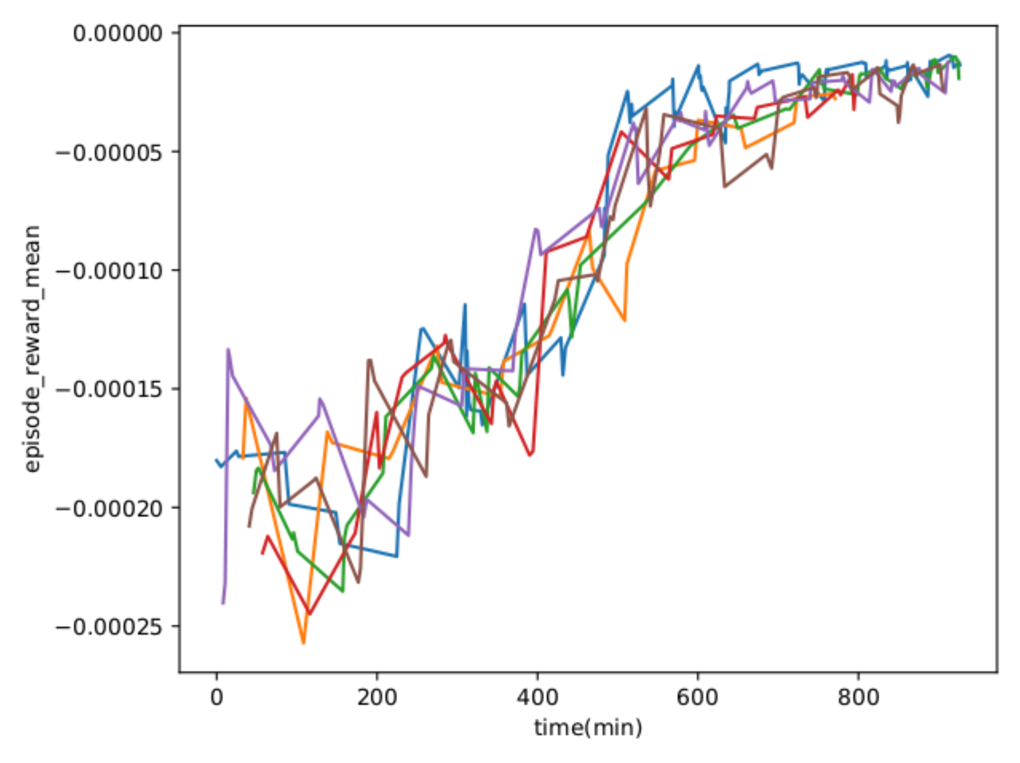
Independently validated on DGX-A100

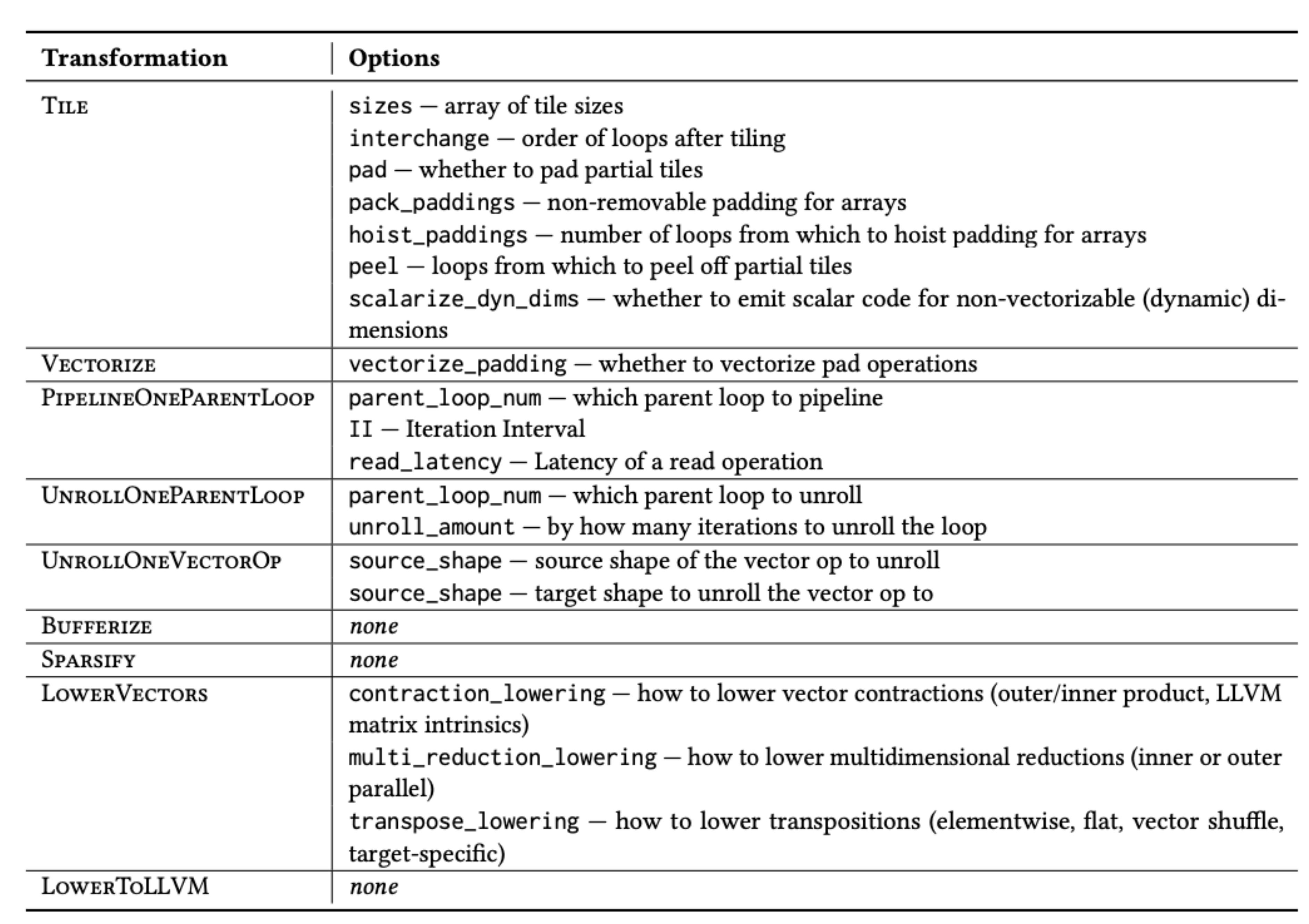
Nod Labs Inc, Confidential - Aug 2022

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







Transformation	Options
Tile	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
Vectorize	vectorize_padding — whether to vectorize pad operations
PipelineOneParentLoop	parent_loop_num — which parent loop to pipeline II — Iteration Interval read_latency — Latency of a read operation
UnrollOneParentLoop	parent_loop_num — which parent loop to unroll unroll_amount — by how many iterations to unroll the loop
UnrollOneVectorOp	source_shape — source shape of the vector op to unroll source_shape — target shape to unroll the vector op to
Bufferize	*none*
Sparsify	*none*
LowerVectors	contraction_lowering — how to lower vector contractions (outer/inner product, LLVM matrix intrinsics) multi_reduction_lowering — how to lower multidimensional reductions (inner or outer parallel) transpose_lowering — how to lower transpositions (elementwise, flat, vector shuffle, target-specific)
LowerToLLVM	*none*

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

- Vasilache, N. et al. (2022) Composable and Modular Code Generation in MLIR. https://arxiv.org/pdf/2202.03293.pdf
- Hsin-I, C. L. et al. (2022) TinyIREE: An ML Execution Environment for Embedded Systems from Compilation to Deployment. https://arxiv.org/pdf/2205.14479.pdf
- 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
- Dawkins, Q. Updated MLIR Dialect Overview Diagram. https://discourse.llvm.org/t/rfc-updated-mlir-dialect-overview-diagram/64266
- Linalg Dialect Rationale: The Case for Compiler Friendly Custom Operations. https://mlir.llvm.org/docs/Rationale/RationaleLinalgDialect/
- Anatomy of Linalg.generic. https://youtu.be/A805W2KSCxQ
- Loop double-buffering/multi-buffering. https://discourse.llvm.org/t/loop-double-buffering-multi-buffering/59979
- CUTLASS: Cuda Template Library for dense linear algebra at all levels and scales. https://on-demand.gputechconf.com/gtc/2018/presentation/s8854-cutlass-software-primitives-for-dense-linear-algebra-at-all-levels-and-scales-within-cuda.pdf
- Bohm, et al. A Novel Hilbert Curve for Cache-locality Preserving Loops. https://eprints.cs.univie.ac.at/5726/1/loops.pdf
- Jordans, R. et al. (2015) High-level software-pipelining in LLVM.
- Hammerblade Manycore Technical Reference Manual. https://docs.google.com/document/d/1b2g2nnMYidMkcn6iHJ9NGjpQYfZeWEmMdLeO_3nLtgo/edit
- Add RISC-V Vector Extension (RVV) Dialect. https://discourse.llvm.org/t/rfc-add-risc-v-vector-extension-rvv-dialect/4146
- Cummins, C. et al. (2021) CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. https://arxiv.org/pdf/2109.08267.pdf
- Zheng, L. et al (2022). Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. https://arxiv.org/pdf/2201.12023.pdf
- Images on Motivation slide taken from OpenAI, NVIDIA, AMD, Sambanova and Cerebras websites.