



# SHARK

High performance compiler & runtime

# Overview

- About SHARK
- System Architecture
- IREE Codegen in SHARK
- Transform Dialect Based Codegen
- Targeting Matmul Primitives
- Convolution Code Generation
- VLIW LLVM backend

# About SHARK

## **Performant and power efficient Model Deployment using Neural Network Codegen Search**

Supports heterogeneous clusters of CPUs, GPUs and custom accelerators with codegen, execution graph partitioning, auto-scheduling and efficient compute / communication overlap.

## **Automatic Kernel Codegen**

No need for thousands of hand optimized kernels. SHARK generates kernels on the fly and fuses them for optimal execution on target hardware.

## **Portable & Retargetable Codegen for new novel A.I hardware support using MLIR**

Enable performant accelerated ML and HPC workloads on hardware such as new A.I Hardware architectures and complex memory hierarchies using LLVM/MLIR

## **Native Framework Integration**

Directly lowers from Tensorflow, JAX and Pytorch for ML workloads removing the need to always be catching up with the latest framework release

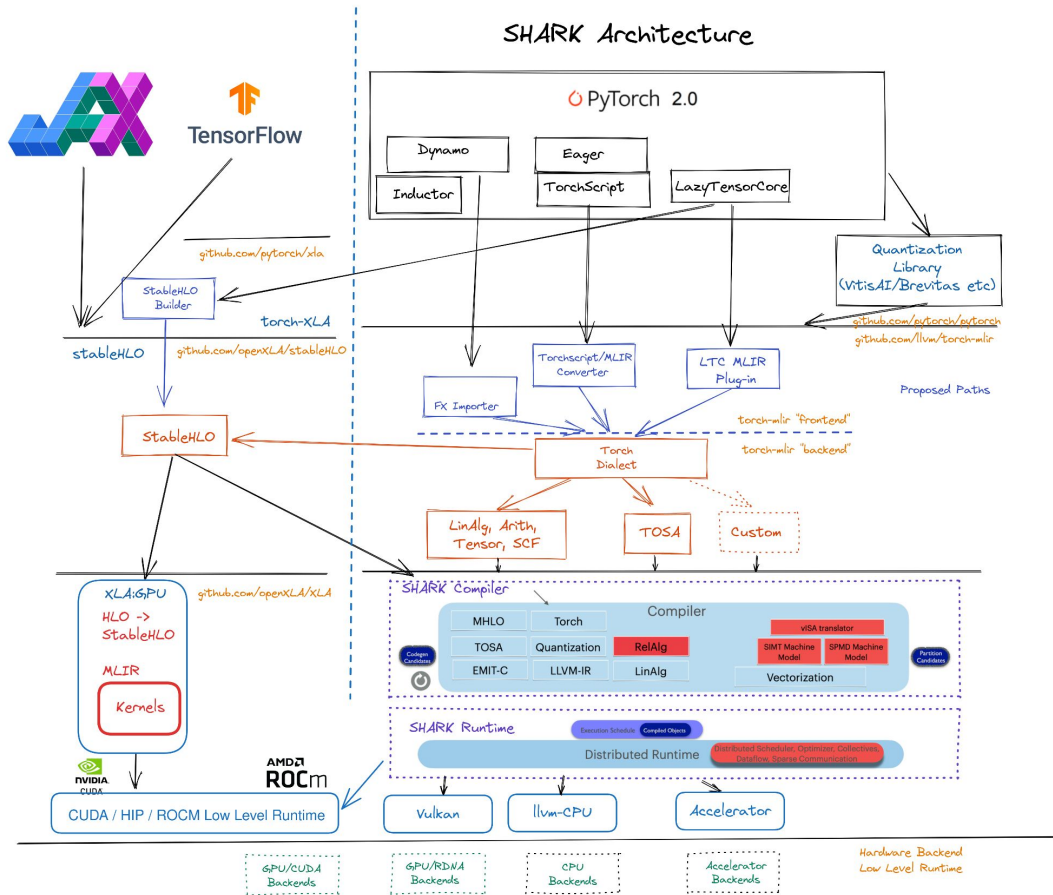
## **Open Source, Built on LLVM / MLIR / IREE**

SHARK is built on permissibly licensed Open Source software including LLVM / MLIR / IREE. Nod.ai provides custom tailored performance enhancement tools, expert Codegen strategies and professional services to further enhance the performance of custom accelerators and large scale deployments of heterogeneous clusters.



# System Architecture

## closer look at the SHARK Architecture

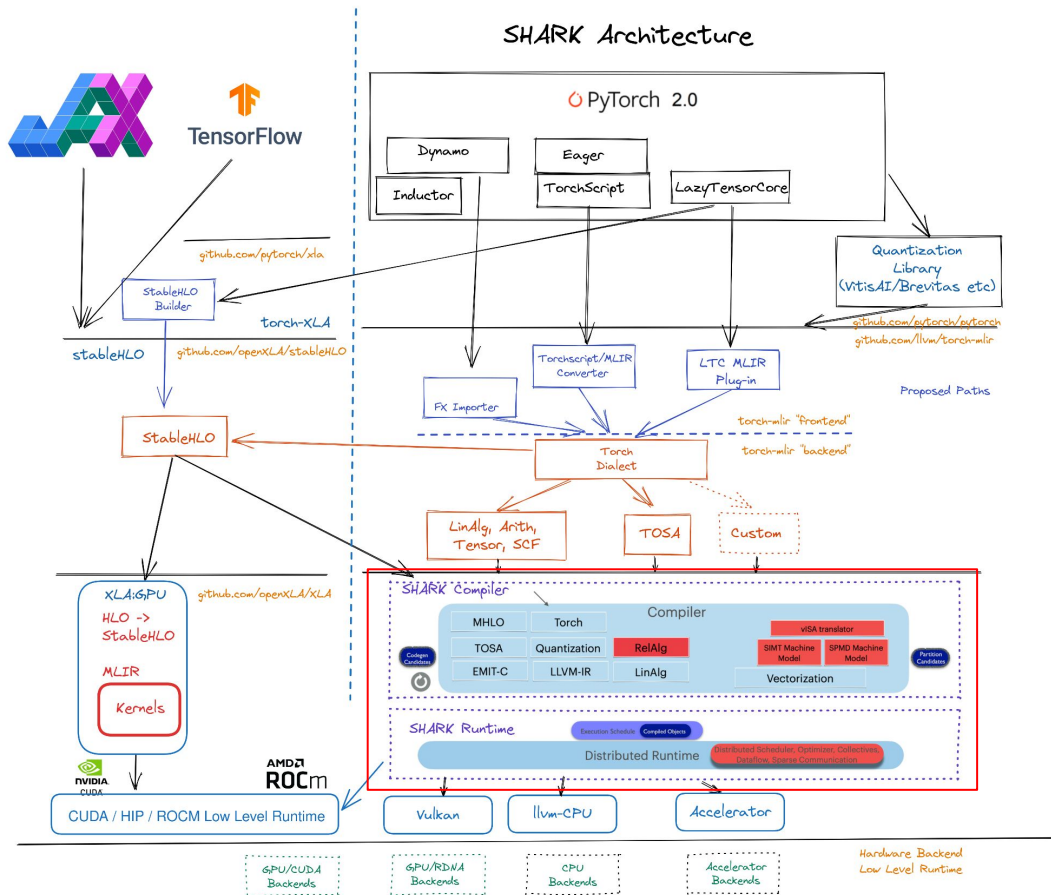


# System Architecture

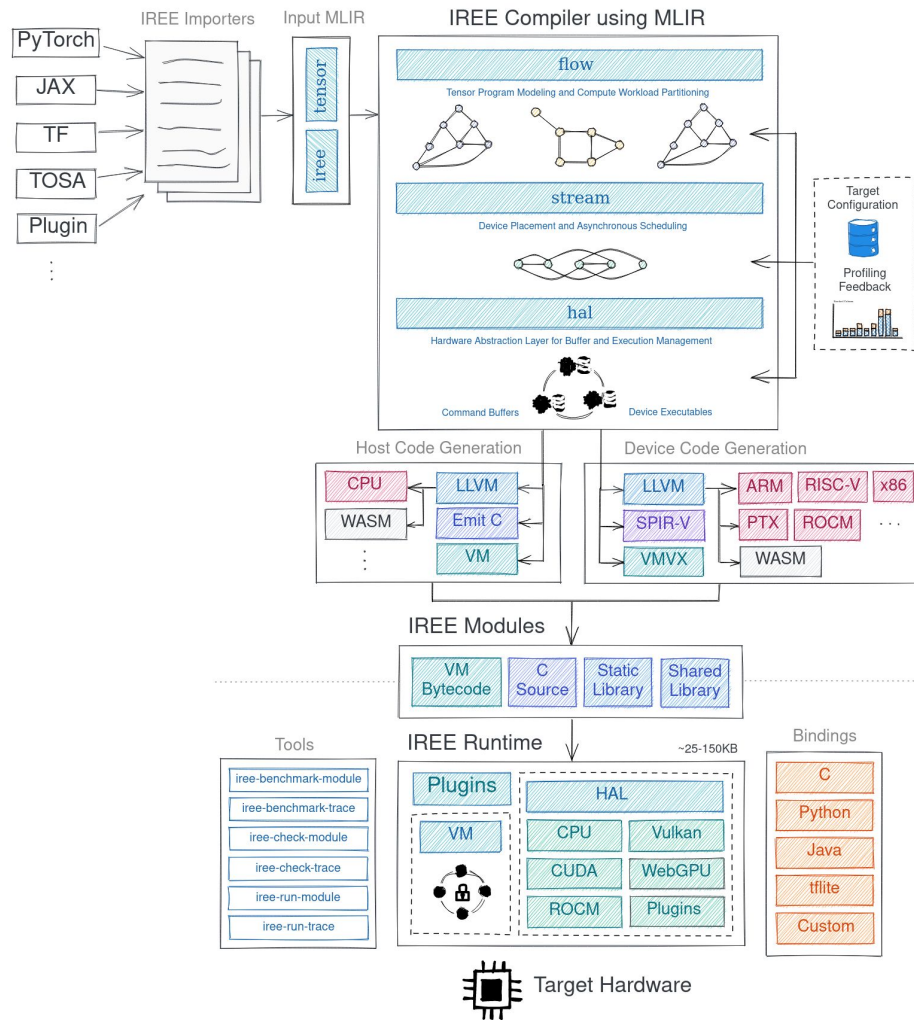
Focus here is on IREE Codegen

- Covers the lowering from Linalg to LLVM-IR/SPIR-V

closer look at the SHARK Architecture



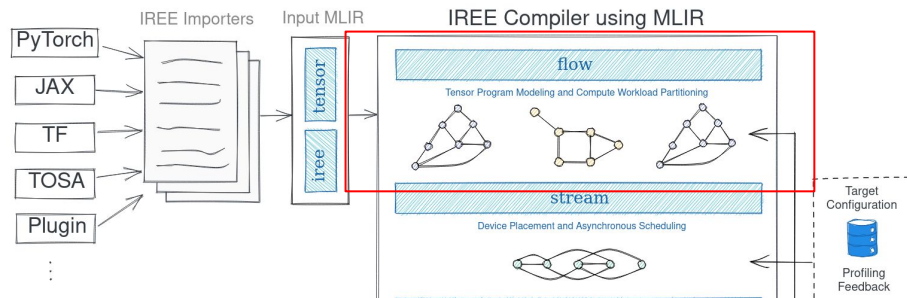
# IREE Architecture + Codegen





# IREE Architecture

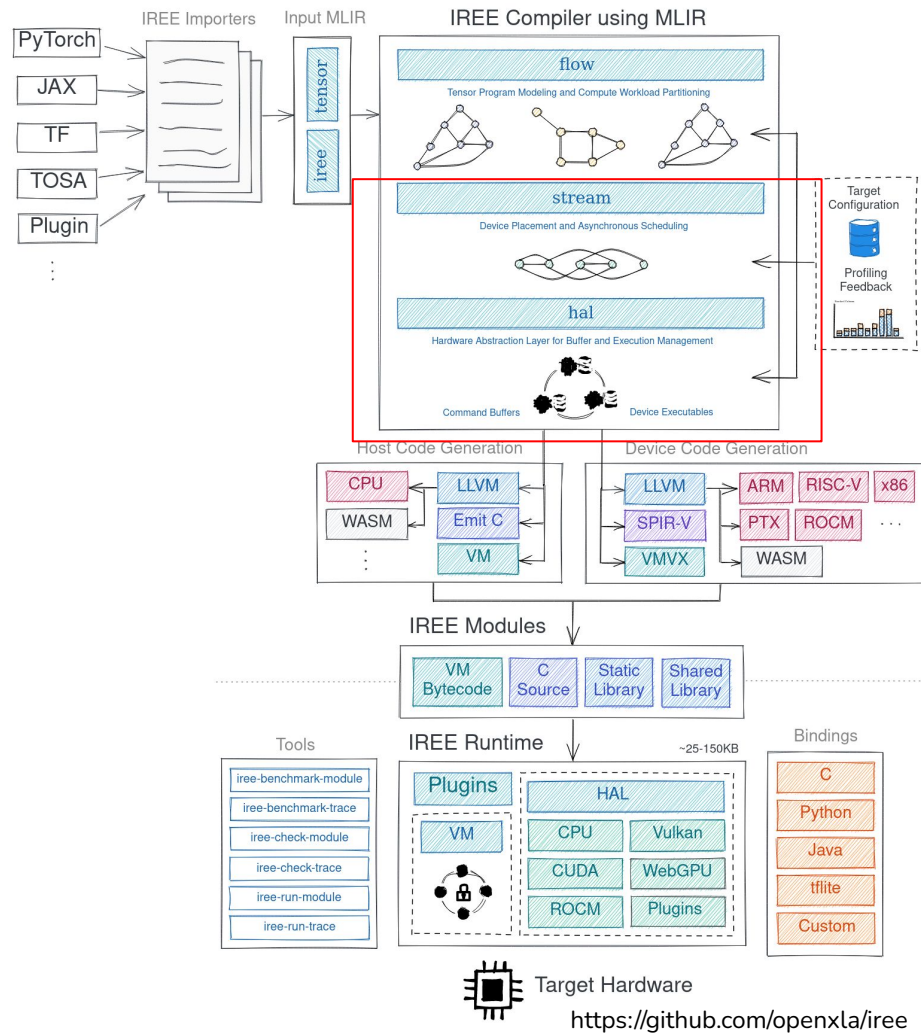
- Flow models a program as regions of dense computation and the data flow between them
  - <https://openxla.github.io/iree/reference/mlir-dialects/Flow/>
- Partitions a model into a dispatch graph



```
hal.executable private @matmul_dispatch_0 {
  hal.executable.variant public @cuda_nvptx_fb, target = #executable_target_cuda_nvptx_fb {
    hal.executable.export public @matmul_dispatch_0_matmul_128x384x1536_f32 ordinal(0) layout(#pipeline_layout) {
      ^bb0(%arg0: !hal.device):
        %x, %y, %z = flow.dispatch.workgroup_count_from_slice
        hal.return %x, %y, %z : index, index, index
    }
  }
  builtin.module {
    func.func @matmul_dispatch_0_matmul_128x384x1536_f32() {
      %c0 = arith.constant 0 : index
      %0 = hal.interface.binding.subspan set(0) binding(0) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : !flow.dispatch.tensor<readonly:tensor<128x1536xf32>>
      %1 = hal.interface.binding.subspan set(0) binding(1) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : !flow.dispatch.tensor<readonly:tensor<1536x384xf32>>
      %2 = hal.interface.binding.subspan set(0) binding(2) type(storage_buffer) alignment(64) offset(%c0) : !flow.dispatch.tensor<readwrite:tensor<128x384xf32>>
      %3 = flow.dispatch.tensor.load %0, offsets = [0, 0], sizes = [128, 1536], strides = [1, 1] : !flow.dispatch.tensor<readonly:tensor<128x1536xf32>> -> tensor<128x1536xf32>
      %4 = flow.dispatch.tensor.load %1, offsets = [0, 0], sizes = [1536, 384], strides = [1, 1] : !flow.dispatch.tensor<readonly:tensor<1536x384xf32>> -> tensor<1536x384xf32>
      %5 = flow.dispatch.tensor.load %2, offsets = [0, 0], sizes = [128, 384], strides = [1, 1] : !flow.dispatch.tensor<readwrite:tensor<128x384xf32>> -> tensor<128x384xf32>
      %6 = linalg.matmul ins(%3, %4 : tensor<128x1536xf32>, tensor<1536x384xf32>) outs(%5 : tensor<128x384xf32>) -> tensor<128x384xf32>
      flow.dispatch.tensor.store %6, %2, offsets = [0, 0], sizes = [128, 384], strides = [1, 1] : tensor<128x384xf32> -> !flow.dispatch.tensor<readwrite:tensor<128x384xf32>>
      return
    }
  }
}
```

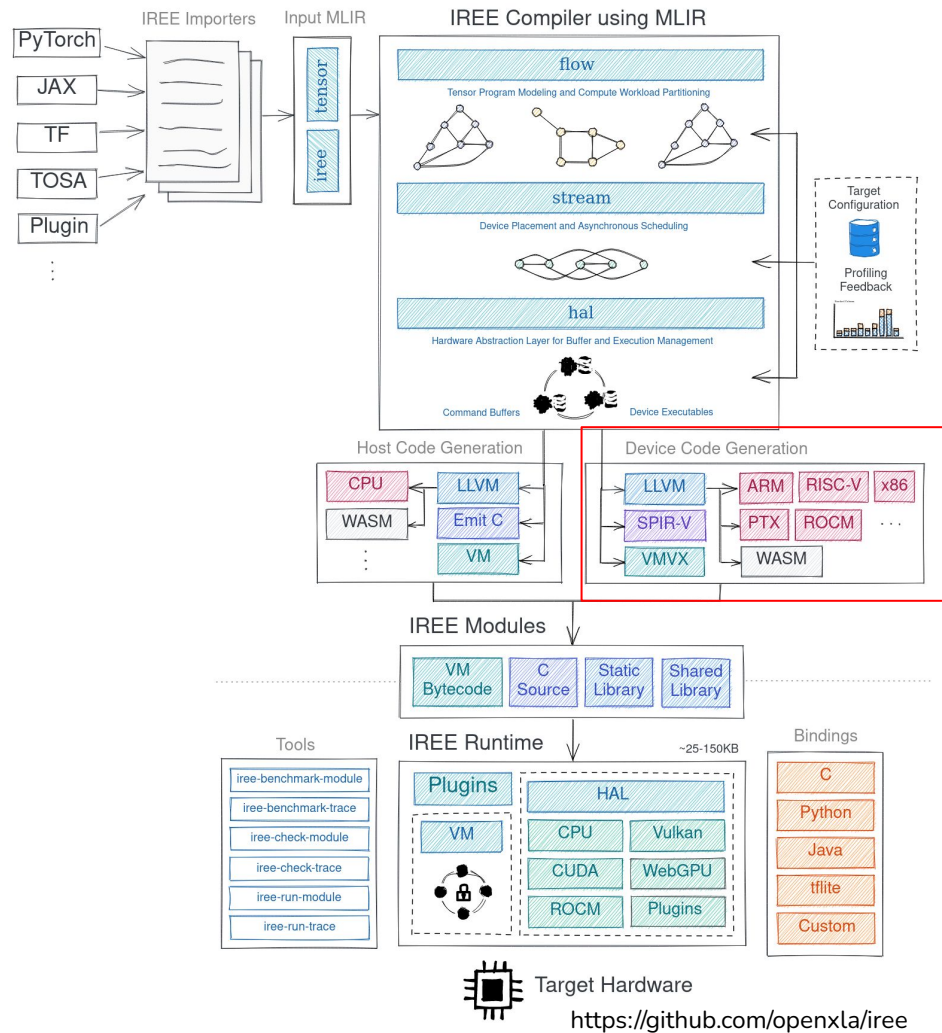
# IREE Architecture

- Stream models execution partitioning and scheduling
  - <https://openxla.github.io/iree/reference/mlir-dialects/Stream/>
- HAL (Hardware Abstraction Layer) abstracts out hardware concepts similar to a Vulkan-like model, only without the focus on graphics
  - <https://openxla.github.io/iree/reference/mlir-dialects/HAL/>
- These dialects model what will be executed on the host side



# IREE Architecture

- Device code generation
- Takes dispatches (Linalg) through to a serialized format
  - LLVM-IR and then through to a target specific format (e.g. x86 or PTX)



# IREE Codegen General Flow

- Codegen of a dispatch is facilitated by `___LowerExecutableTargetPass`
  - Matches the dispatch to a strategy in the form of a pass pipeline
    - e.g. `SPIRVCooperativeMatrixVectorize`
    - Lowers to a mix of (tilted) `Linalg/Arith/SCF/Vector` on `MemRefs`
- Output of `LowerExecutableTargetPass` is taken to serialization (`SPIR-V/LLVM`) with a unified pipeline
  - `addSPIRVLoweringPasses`
    - Unroll to supported 1/2/3/4 vector sizes
    - Flatten memrefs
    - Convert to `SPIR-V`
  - `addLowerToLLVMGPUPasses`
  - `addLowerToLLVMPasses`

Transform Dialect Based Codegen

# Pass Pipelines vs Transform Dialect

- Pass pipeline based codegen is good at handling a breadth of common inputs
  - Different flavors of elementwise/broadcasts/transposes/reductions with some degree of fusion, anchored on a “root” operation.
  - Difficult to scale to specialized strategies (i.e. Flash Attention, Implicit GEMM) and fusions
  - Requires rematching key operations within each pass
  - Pipelines are mostly static; hard to configure on the fly
- Transform dialect allows specifying a “recipe”
  - Recipes are less resistant to input variations, but make up for it by constructing it on the fly
  - Strategy specific lowering information (tile sizes, workgroup size) is carried in the recipe rather than the IR itself

# Transform Dialect Codegen

- Codegen “recipes” that can be passed in externally, or constructed on the fly using standard MLIR OpBuilders
  - `--iree-codegen-llvmgpu-use-transform-dialect=/path/to/codegen_spec.mlir`
    - ```
#transform = #iree_codegen.translation_info<TransformDialectCodegen>
#config = #iree_codegen.lowering_config<codegen_spec_file_name = "/path/to/codegen_spec.mlir">
#transpose = #iree_codegen.compilation_info<lowering_config = #config, translation_info = #transform>
```
- <https://github.com/openxla/iree/tree/main/compiler/src/iree/compiler/Codegen/TransformStrategies>
  - See the above for examples of on the fly construction of transform scripts
- Transform IR is applied with the TransformDialectInterpreter pass

# Unaligned Matmul Case Study

```
hal.executable.variant public @cuda_nvptx_fb, target = <"cuda", "cuda-nvptx-fb", {target_arch = "sm_80"}> {
  hal.executable.export public @matmul_dispatch_0_matmul_130x400x1500_f32 ordinal(0) layout(#hal.pipeline.layout<push_constants = 0, sets = [<0, bindings = [<0, storage_buffer,
ReadOnly>, <1, storage_buffer, ReadOnly>, <2, storage_buffer>]>)]> attributes {translation_info = #iree_codegen.translation_info<TransformDialectCodegen>} {
  ^bb0(%arg0: !hal.device):
    %x, %y, %z = flow.dispatch.workgroup_count_from_slice
    hal.return %x, %y, %z : index, index, index
  }
builtin.module {
  func.func @matmul_dispatch_0_matmul_130x400x1500_f32() {
    %c0 = arith.constant 0 : index
    %cst = arith.constant 0.000000e+00 : f32
    %0 = hal.interface.binding.subspan set(0) binding(0) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : !flow.dispatch.tensor<readonly:tensor<130x1500xf32>>
    %1 = hal.interface.binding.subspan set(0) binding(1) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : !flow.dispatch.tensor<readonly:tensor<1500x400xf32>>
    %2 = hal.interface.binding.subspan set(0) binding(2) type(storage_buffer) alignment(64) offset(%c0) : !flow.dispatch.tensor<writeonly:tensor<130x400xf32>>
    %3 = flow.dispatch.tensor.load %0, offsets = [0, 0], sizes = [130, 1500], strides = [1, 1] : !flow.dispatch.tensor<readonly:tensor<130x1500xf32>> -> tensor<130x1500xf32>
    %4 = flow.dispatch.tensor.load %1, offsets = [0, 0], sizes = [1500, 400], strides = [1, 1] : !flow.dispatch.tensor<readonly:tensor<1500x400xf32>> -> tensor<1500x400xf32>
    %5 = tensor.empty() : tensor<130x400xf32>
    %6 = linalg.fill ins(%cst : f32) outs(%5 : tensor<130x400xf32>) -> tensor<130x400xf32>
    %7 = linalg.matmul ins(%3, %4 : tensor<130x1500xf32>, tensor<1500x400xf32>) outs(%6 : tensor<130x400xf32>) -> tensor<130x400xf32>
    flow.dispatch.tensor.store %7, %2, offsets = [0, 0], sizes = [130, 400], strides = [1, 1] : tensor<130x400xf32> -> !flow.dispatch.tensor<writeonly:tensor<130x400xf32>>
    return
  }
  module {
    transform.sequence failures(propagate) {
      ^bb0(%arg0: !transform.any_op):
        transform.iree.register_match_callbacks
        %0:3 = transform.iree.match_callback failures(propagate) "matmul"(%arg0) : (!transform.any_op -> (!transform.any_op, !transform.any_op, !transform.any_op)
        ...
      }
    }
  }
}
```



# Unaligned Matmul Case Study: Block level tiling

```
transform.iree.register_match_callbacks
%0:3 = transform.iree.match_callback failures(propagate) "matmul"(%arg0) : (!transform.any_op)
-> (!transform.any_op, !transform.any_op, !transform.any_op)
%forall_op, %tiled_op = transform.structured.tile_to_forall_op %0#1    num_threads [] tile_sizes
[128, 128](mapping = [#gpu.block<y>, #gpu.block<x>]) : (!transform.any_op) ->
(!transform.any_op, !transform.any_op)
%1 = transform.structured.match ops[["func.func"]] in %arg0 : (!transform.any_op) ->
!transform.any_op
apply_patterns to %1 {
  transform.apply_patterns.linalg.tiling_canonicalization
  transform.apply_patterns.iree.fold_fill_into_pad
  transform.apply_patterns.scf.for_loop_canonicalization
  transform.apply_patterns.canonicalization
} : !transform.any_op
transform.iree.apply_licm %1 : !transform.any_op
transform.iree.apply_cse %1 : !transform.any_op
```

# Unaligned Matmul Case Study: Block level tiling

```
%7 = scf.forall (%arg0, %arg1) in (2, 4) shared_outs(%arg2 = %6) -> (tensor<130x400xf32>) {  
  %8 = affine.min #map(%arg0)  
  %9 = affine.min #map1(%arg1)  
  %10 = affine.apply #map2(%arg0)  
  %11 = affine.apply #map2(%arg1)  
  %extracted_slice = tensor.extract_slice %3[%10, 0] [%8, 1500] [1, 1] : tensor<130x1500xf32> to  
  tensor<?x1500xf32>  
  %extracted_slice_0 = tensor.extract_slice %4[0, %11] [1500, %9] [1, 1] : tensor<1500x400xf32> to  
  tensor<1500x?xf32>  
  %extracted_slice_1 = tensor.extract_slice %arg2[%10, %11] [%8, %9] [1, 1] : tensor<130x400xf32> to  
  tensor<?x?xf32>  
  %12 = linalg.matmul ins(%extracted_slice, %extracted_slice_0 : tensor<?x1500xf32>, tensor<1500x?xf32>)  
  outs(%extracted_slice_1 : tensor<?x?xf32>) -> tensor<?x?xf32>  
  scf.forall.in_parallel {  
    tensor.parallel_insert_slice %12 into %arg2[%10, %11] [%8, %9] [1, 1] : tensor<?x?xf32> into  
    tensor<130x400xf32>  
  }  
} {mapping = [#gpu.block<y>, #gpu.block<x>]}
```

# Unaligned Matmul Case Study: Padding to tile size

```
%fused_op, %new_containing_op = transform.structured.fuse_into_containing_op %0#0 into
%forall_op : (!transform.any_op, !transform.any_op) -> (!transform.any_op,
!transform.any_op)
transform.iree.populate_workgroup_count_region_using_num_threads_slice %forall_op :
(!transform.any_op) -> ()
%tilled_linalg_op, %loops = transform.structured.tile %tilled_op[0, 0, 16] :
(!transform.any_op) -> (!transform.any_op, !transform.any_op)
%padded, %pad = transform.structured.pad %tilled_linalg_op {copy_back = false, pack_paddings
= [1, 1, 1], pad_to_multiple_of = [1, 1, 1], padding_dimensions = [0, 1, 2], padding_values
= [0.000000e+00 : f32, 0.000000e+00 : f32, 0.000000e+00 : f32]} : (!transform.any_op) ->
(!transform.any_op, !transform.any_op)
%2 = get_producer_of_operand %padded[2] : (!transform.any_op) -> !transform.any_op
%3 = cast %2 : !transform.any_op to !transform.op<"tensor.pad">
%4 = transform.structured.hoist_pad %3 by 1 loops : (!transform.op<"tensor.pad">) ->
!transform.any_op
```

# Unaligned Matmul Case Study: Padding to tile size

```
%11 = scf.for %arg3 = %c0 to %c1500 step %c16 iter_args(%arg4 = %10) -> (tensor<128x128xf32>) {
  %14 = affine.min #map2(%arg3)
  %15 = affine.apply #map3(%arg0)
  %extracted_slice_0 = tensor.extract_slice %3[%15, %arg3] [%7, %14] [1, 1] : tensor<130x1500xf32> to tensor<?x?xf32>
  %16 = affine.apply #map3(%arg1)
  %extracted_slice_1 = tensor.extract_slice %4[%arg3, %16] [%14, %8] [1, 1] : tensor<1500x400xf32> to tensor<?x?xf32>
  %17 = affine.apply #map4(%7)
  %18 = affine.apply #map5(%14)
  %padded = tensor.pad %extracted_slice_0 nofold low[0, 0] high[%17, %18] {
    ^bb0(%arg5: index, %arg6: index):
      tensor.yield %cst : f32
  } : tensor<?x?xf32> to tensor<128x16xf32>
  %19 = affine.apply #map5(%14)
  %20 = affine.apply #map4(%8)
  %padded_2 = tensor.pad %extracted_slice_1 nofold low[0, 0] high[%19, %20] {
    ^bb0(%arg5: index, %arg6: index):
      tensor.yield %cst : f32
  } : tensor<?x?xf32> to tensor<16x128xf32>
  %21 = linalg.matmul ins(%padded, %padded_2 : tensor<128x16xf32>, tensor<16x128xf32>) outs(%arg4 : tensor<128x128xf32>)
-> tensor<128x128xf32>
  scf.yield %21 : tensor<128x128xf32>
}
```

# Unaligned Matmul Case Study: SIMT

```
%9 = transform.structured.insert_slice_to_copy %8 : (!transform.any_op) -> !transform.any_op
%10 = get_producer_of_operand %padded[0] : (!transform.any_op) -> !transform.any_op
%11 = get_producer_of_operand %padded[1] : (!transform.any_op) -> !transform.any_op
%forall_op_0, %tiled_op_1 = transform.structured.tile_to_forall_op %10    num_threads [32, 4] tile_sizes
[] (mapping = [#gpu.linear<y>, #gpu.linear<x>]) : (!transform.any_op) -> (!transform.any_op,
!transform.any_op)
%13 = transform.structured.match_ops{["scf.if"]} in %forall_op_0 : (!transform.any_op) ->
!transform.any_op
transform.scf.take_assumed_branch %13 take_else_branch : (!transform.any_op) -> ()
%forall_op_2, %tiled_op_3 = transform.structured.tile_to_forall_op %11    num_threads [4, 32] tile_sizes
[] (mapping = [#gpu.linear<y>, #gpu.linear<x>]) : (!transform.any_op) -> (!transform.any_op,
!transform.any_op)
%15 = transform.structured.match_ops{["scf.if"]} in %forall_op_2 : (!transform.any_op) ->
!transform.any_op
transform.scf.take_assumed_branch %15 take_else_branch : (!transform.any_op) -> ()
%forall_op_4, %tiled_op_5 = transform.structured.tile_to_forall_op %9    num_threads [4, 32] tile_sizes
[] (mapping = [#gpu.linear<y>, #gpu.linear<x>]) : (!transform.any_op) -> (!transform.any_op,
!transform.any_op)
```

# Unaligned Matmul Case Study: SIMT

```
%15 = scf.for %arg3 = %c0 to %c1500 step %c16 iter_args(%arg4 = %10) -> (tensor<128x128xf32>) {
  %17 = affine.min #map3(%arg3)
  %extracted_slice_1 = tensor.extract_slice %3[%11, %arg3] [%7, %17] [1, 1] : tensor<130x1500xf32> to tensor<?x?xf32>
  %extracted_slice_2 = tensor.extract_slice %4[%arg3, %12] [%17, %8] [1, 1] : tensor<1500x400xf32> to tensor<?x?xf32>
  %18 = scf.forall (%arg5, %arg6) in (32, 4) shared_outs(%arg7 = %13) -> (tensor<128x16xf32>) {
    # Index arithmetic
    %extracted_slice_3 = tensor.extract_slice %extracted_slice_1[%23, %22] [%25, %28] [1, 1] : tensor<?x?xf32> to tensor<?x?xf32>
    %padded = tensor.pad %extracted_slice_3 nofold low[0, 0] high[%26, %29] {
      ^bb0(%arg8: index, %arg9: index):
        tensor.yield %cst : f32
    } : tensor<?x?xf32> to tensor<4x4xf32>
    scf.forall.in_parallel {
      tensor.parallel_insert_slice %padded into %arg7[%21, %22] [4, 4] [1, 1] : tensor<4x4xf32> into tensor<128x16xf32>
    }
  } {mapping = [#gpu.linear<y>, #gpu.linear<x>]}
  %19 = scf.forall (%arg5, %arg6) in (4, 32) shared_outs(%arg7 = %14) -> (tensor<16x128xf32>) {
    # Index arithmetic
    %extracted_slice_3 = tensor.extract_slice %extracted_slice_2[%21, %26] [%24, %28] [1, 1] : tensor<?x?xf32> to tensor<?x?xf32>
    %padded = tensor.pad %extracted_slice_3 nofold low[0, 0] high[%25, %29] {
      ^bb0(%arg8: index, %arg9: index):
        tensor.yield %cst : f32
    } : tensor<?x?xf32> to tensor<4x4xf32>
    scf.forall.in_parallel {
      tensor.parallel_insert_slice %padded into %arg7[%21, %22] [4, 4] [1, 1] : tensor<4x4xf32> into tensor<16x128xf32>
    }
  } {mapping = [#gpu.linear<y>, #gpu.linear<x>]}
  %20 = linalg.matmul ins(%18, %19 : tensor<128x16xf32>, tensor<16x128xf32>) outs(%arg4 : tensor<128x128xf32>) -> tensor<128x128xf32>
  scf.yield %20 : tensor<128x128xf32>
}
```

# Unaligned Matmul Case Study: SIMD

```
%forall_op_6, %tiled_op_7 = transform.structured.tile_to_forall_op %padded
num_threads [2, 2] tile_sizes [] (mapping = [#gpu.warp<y>, #gpu.warp<x>]) :
(!transform.any_op) -> (!transform.any_op, !transform.any_op)

%forall_op_8, %tiled_op_9 = transform.structured.tile_to_forall_op %6
num_threads [2, 2] tile_sizes [] (mapping = [#gpu.warp<y>, #gpu.warp<x>]) :
(!transform.any_op) -> (!transform.any_op, !transform.any_op)
```

# Unaligned Matmul Case Study: SIMD

```
%20 = scf.forall (%arg5, %arg6) in (2, 2) shared_outs(%arg7 = %arg4) -> (tensor<128x128xf32>) {  
  %21 = affine.apply #map2(%arg5)  
  %22 = affine.apply #map2(%arg6)  
  %extracted_slice_3 = tensor.extract_slice %18[%21, 0] [64, 16] [1, 1] : tensor<128x16xf32> to  
  tensor<64x16xf32>  
  %extracted_slice_4 = tensor.extract_slice %19[0, %22] [16, 64] [1, 1] : tensor<16x128xf32> to  
  tensor<16x64xf32>  
  %extracted_slice_5 = tensor.extract_slice %arg7[%21, %22] [64, 64] [1, 1] : tensor<128x128xf32> to  
  tensor<64x64xf32>  
  %23 = linalg.matmul ins(%extracted_slice_3, %extracted_slice_4 : tensor<64x16xf32>,  
  tensor<16x64xf32>) outs(%extracted_slice_5 : tensor<64x64xf32>) -> tensor<64x64xf32>  
  scf.forall.in_parallel {  
    tensor.parallel_insert_slice %23 into %arg7[%21, %22] [64, 64] [1, 1] : tensor<64x64xf32> into  
    tensor<128x128xf32>  
  }  
} {mapping = [#gpu.warp<y>, #gpu.warp<x>]}
```



# Unaligned Matmul Case Study: Vector Masking

```
transform.structured.masked_vectorize %tiled_op_1 vector_sizes [4, 4] : !transform.any_op
transform.structured.masked_vectorize %tiled_op_3 vector_sizes [4, 4] : !transform.any_op
transform.structured.masked_vectorize %tiled_op_5 vector_sizes [32, 4] : !transform.any_op
%20 = transform.structured.match ops{["func.func"]} in %arg0 : (!transform.any_op) ->
!transform.any_op
apply_patterns to %20 {
  transform.apply_patterns.vector.lower_masked_transfers
} : !transform.any_op
```

# Unaligned Matmul Case Study: Vector Masking

```
%18 = scf.forall (%arg5, %arg6) in (32, 4) shared_outs(%arg7 = %13) -> (tensor<128x16xf32>) {  
  ...  
  %extracted_slice_3 = tensor.extract_slice %extracted_slice_1[%23, %22] [%25, %27] [1, 1] : tensor<?x?xf32> to tensor<?x?xf32>  
  %28 = tensor.empty() : tensor<4x4xf32>  
  %29 = vector.create_mask %25, %27 : vector<4x4xi1>  
  %30 = vector.transfer_read %extracted_slice_3[%c0, %c0], %cst, %29 {in_bounds = [true, true]} : tensor<?x?xf32>, vector<4x4xf32>  
  %31 = vector.transfer_write %30, %28[%c0, %c0] {in_bounds = [true, true]} : vector<4x4xf32>, tensor<4x4xf32>  
  scf.forall.in_parallel {  
    tensor.parallel_insert_slice %31 into %arg7[%21, %22] [4, 4] [1, 1] : tensor<4x4xf32> into tensor<128x16xf32>  
  }  
} {mapping = [#gpu.linear<y>, #gpu.linear<x>]}
```

```
%19 = scf.forall (%arg5, %arg6) in (4, 32) shared_outs(%arg7 = %14) -> (tensor<16x128xf32>) {  
  ...  
  %extracted_slice_3 = tensor.extract_slice %extracted_slice_2[%21, %25] [%24, %27] [1, 1] : tensor<?x?xf32> to tensor<?x?xf32>  
  %28 = tensor.empty() : tensor<4x4xf32>  
  %29 = vector.create_mask %24, %27 : vector<4x4xi1>  
  %30 = vector.transfer_read %extracted_slice_3[%c0, %c0], %cst, %29 {in_bounds = [true, true]} : tensor<?x?xf32>, vector<4x4xf32>  
  %31 = vector.transfer_write %30, %28[%c0, %c0] {in_bounds = [true, true]} : vector<4x4xf32>, tensor<4x4xf32>  
  scf.forall.in_parallel {  
    tensor.parallel_insert_slice %31 into %arg7[%21, %22] [4, 4] [1, 1] : tensor<4x4xf32> into tensor<16x128xf32>  
  }  
} {mapping = [#gpu.linear<y>, #gpu.linear<x>]}
```

# Unaligned Matmul Case Study: Vectorization

```
%22 = transform.structured.vectorize %21 : (!transform.any_op) -> !transform.any_op
-----
%20 = scf.forall (%arg5, %arg6) in (2, 2) shared_outs(%arg7 = %arg4) -> (tensor<128x128xf32>) {
  %21 = affine.apply #map2(%arg5)
  %22 = affine.apply #map2(%arg6)
  %extracted_slice_4 = tensor.extract_slice %arg7[%21, %22] [64, 64] [1, 1] : tensor<128x128xf32> to tensor<64x64xf32>
  %23 = vector.transfer_read %18[%21, %c0], %cst_0 {in_bounds = [true, true]} : tensor<128x16xf32>, vector<64x16xf32>
  %24 = vector.transfer_read %19[%c0, %22], %cst_0 {in_bounds = [true, true]} : tensor<16x128xf32>, vector<16x64xf32>
  %25 = vector.transfer_read %arg7[%21, %22], %cst_0 {in_bounds = [true, true]} : tensor<128x128xf32>,
vector<64x64xf32>
  %26 = vector.contract {indexing_maps = [#map13, #map14, #map15], iterator_types = ["parallel", "parallel",
"reduction"], kind = #vector.kind<add>} %23, %24, %25 : vector<64x16xf32>, vector<16x64xf32> into vector<64x64xf32>
  %27 = vector.transfer_write %26, %extracted_slice_4[%c0, %c0] {in_bounds = [true, true]} : vector<64x64xf32>,
tensor<64x64xf32>
  scf.forall.in_parallel {
    tensor.parallel_insert_slice %27 into %arg7[%21, %22] [64, 64] [1, 1] : tensor<64x64xf32> into tensor<128x128xf32>
  }
} {mapping = [#gpu.warp<y>, #gpu.warp<x>]}
```

# Unaligned Matmul Case Study: Bufferization

```
%24 = transform.iree.bufferize {target_gpu} %arg0 : (!transform.any_op) ->
!transform.any_op
%25 = transform.structured.match ops{["func.func"]} in %24 :
(!transform.any_op) -> !transform.any_op
transform.iree.apply_buffer_optimizations %25 : (!transform.any_op) -> ()
```

# Unaligned Matmul Case Study: Bufferization

```
%0 = hal.interface.binding.subspan set(0) binding(0) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : memref<130x1500xf32>
memref.assume_alignment %0, 64 : memref<130x1500xf32>
%1 = hal.interface.binding.subspan set(0) binding(1) type(storage_buffer) alignment(64) offset(%c0) flags(ReadOnly) : memref<1500x400xf32>
memref.assume_alignment %1, 64 : memref<1500x400xf32>
%2 = hal.interface.binding.subspan set(0) binding(2) type(storage_buffer) alignment(64) offset(%c0) : memref<130x400xf32>
...
memref.assume_alignment %2, 64 : memref<130x400xf32>
%alloc = memref.alloc() {alignment = 64 : i64} : memref<128x128xf32, #gpu.address_space<workgroup>>
%alloc_1 = memref.alloc() {alignment = 64 : i64} : memref<128x16xf32, #gpu.address_space<workgroup>>
%alloc_2 = memref.alloc() {alignment = 64 : i64} : memref<16x128xf32, #gpu.address_space<workgroup>>
...
%subview_4 = memref.subview %0[%5, %arg2] [%3, %7] [1, 1] : memref<130x1500xf32> to memref<?x?xf32, strided<[1500, 1], offset: ??>
%subview_5 = memref.subview %1[%arg2, %6] [%7, %4] [1, 1] : memref<1500x400xf32> to memref<?x?xf32, strided<[400, 1], offset: ??>
...
%subview_6 = memref.subview %subview_4[%10, %9] [%12, %14] [1, 1] : memref<?x?xf32, strided<[1500, 1], offset: ??> to memref<?x?xf32, strided<[1500, 1], offset: ??>
%subview_7 = memref.subview %alloc_1[%8, %9] [4, 4] [1, 1] : memref<128x16xf32, #gpu.address_space<workgroup>> to memref<4x4xf32, strided<[16, 1], offset: ??>,
#gpu.address_space<workgroup>>
%15 = vector.create_mask %12, %14 : vector<4x4xi1>
%16 = vector.transfer_read %subview_6[%c0, %c0], %cst_0, %15 {in_bounds = [true, true]} : memref<?x?xf32, strided<[1500, 1], offset: ??>, vector<4x4xf32>
vector.transfer_write %16, %subview_7[%c0, %c0] {in_bounds = [true, true]} : vector<4x4xf32>, memref<4x4xf32, strided<[16, 1], offset: ??>, #gpu.address_space<workgroup>>
...
%subview_6 = memref.subview %subview_5[%8, %12] [%11, %14] [1, 1] : memref<?x?xf32, strided<[400, 1], offset: ??> to memref<?x?xf32, strided<[400, 1], offset: ??>
%subview_7 = memref.subview %alloc_2[%8, %9] [4, 4] [1, 1] : memref<16x128xf32, #gpu.address_space<workgroup>> to memref<4x4xf32, strided<[128, 1], offset: ??>,
#gpu.address_space<workgroup>>
%15 = vector.create_mask %11, %14 : vector<4x4xi1>
%16 = vector.transfer_read %subview_6[%c0, %c0], %cst_0, %15 {in_bounds = [true, true]} : memref<?x?xf32, strided<[400, 1], offset: ??>, vector<4x4xf32>
vector.transfer_write %16, %subview_7[%c0, %c0] {in_bounds = [true, true]} : vector<4x4xf32>, memref<4x4xf32, strided<[128, 1], offset: ??>, #gpu.address_space<workgroup>>
...
%subview_6 = memref.subview %alloc[%8, %9] [64, 64] [1, 1] : memref<128x128xf32, #gpu.address_space<workgroup>> to memref<64x64xf32, strided<[128, 1], offset: ??>,
#gpu.address_space<workgroup>>
%10 = vector.transfer_read %alloc_1[%8, %c0], %cst_0 {in_bounds = [true, true]} : memref<128x16xf32, #gpu.address_space<workgroup>>, vector<64x16xf32>
%11 = vector.transfer_read %alloc_2[%c0, %9], %cst_0 {in_bounds = [true, true]} : memref<16x128xf32, #gpu.address_space<workgroup>>, vector<16x64xf32>
%12 = vector.transfer_read %alloc[%8, %9], %cst_0 {in_bounds = [true, true]} : memref<128x128xf32, #gpu.address_space<workgroup>>, vector<64x64xf32>
%13 = vector.contract {indexing_maps = [#map13, #map14, #map15], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %10, %11, %12 : vector<64x16xf32>,
vector<16x64xf32> into vector<64x64xf32>
vector.transfer_write %13, %subview_6[%c0, %c0] {in_bounds = [true, true]} : vector<64x64xf32>, memref<64x64xf32, strided<[128, 1], offset: ??>, #gpu.address_space<workgroup>>
```

# Unaligned Matmul Case Study: Distribution

```
transform.iree.map_nested_forall_to_gpu_threads %26 workgroup_dims = [64,  
2, 1] warp_dims = [2, 2, 1] : (!transform.any_op) -> ()  
%27 = transform.iree.eliminate_gpu_barriers %26 : (!transform.any_op) ->  
!transform.any_op
```

```
-----  
%workgroup_id_y = hal.interface.workgroup.id[1] : index  
%workgroup_id_x = hal.interface.workgroup.id[0] : index  
%5 = gpu.thread_id x  
%6 = gpu.thread_id y
```

# Unaligned Matmul Case Study: Unrolling + Loop Cleanup

```
transform.iree.hoist_static_alloc %27 : (!transform.any_op) -> ()
apply_patterns to %27 {
  transform.apply_patterns.memref.fold_memref_alias_ops
} : !transform.any_op
apply_patterns to %27 {
  transform.apply_patterns.memref.extract_address_computations
} : !transform.any_op
apply_patterns to %27 {
  transform.apply_patterns.iree.unroll_vectors_gpu_mma_sync
} : !transform.any_op
%28 = transform.structured.match ops{["scf.for"]} in %27 : (!transform.any_op) ->
!transform.op<"scf.for">
transform.iree.synchronize_loop %28 : (!transform.op<"scf.for">) -> ()
%29 = transform.structured.hoist_redundant_vector_transfers %27 : (!transform.any_op) ->
!transform.any_op
```

# Unaligned Matmul Case Study: Unrolling + Loop Cleanup

```
%80 = vector.transfer_read %subview_5[%c0, %c0], %cst_0 {in_bounds = [true, true]} : memref<?x16xf32, strided<[16, 1], offset: ?>, #gpu.address_space<workgroup>>, vector<16x8xf32> x 8

%88 = vector.transfer_read %subview_6[%c0, %c0], %cst_0 {in_bounds = [true, true]} : memref<16x?xf32, strided<[128, 1], offset: ?>, #gpu.address_space<workgroup>>, vector<8x8xf32> x 16

%104 = vector.contract {indexing_maps = [#map26, #map27, #map28], iterator_types = ["parallel", "parallel", "reduction"], kind = #vector.kind<add>} %80, %88, %arg1 : vector<16x8xf32>, vector<8x8xf32> into vector<16x8xf32> x 64
```



# Unaligned Matmul Case Study: Conversion to MMA

```
transform.iree.vector.vector_to_mma_conversion %29 {use_mma_sync} :  
(!transform.any_op) -> ()  
-----  
%147 = nvgpu.ldmatrix %subview_6[%28, %29] {numTiles = 4 : i32, transpose =  
false} : memref<?x16xf32, strided<[16, 1], offset: ?>,  
#gpu.address_space<workgroup>> -> vector<4x1xf32>  
%194 = nvgpu.mma.sync(%147, %193, %arg8) {mmaShape = [16, 8, 8], tf32Enabled} :  
(vector<4x1xf32>, vector<2x1xf32>, vector<2x2xf32>) -> vector<2x2xf32>
```

# Unaligned Matmul Case Study: Pipelining + async.copy

```
%31 = transform.memref.multibuffer %30 {factor = 3 : i64, skip_analysis} :  
(!transform.op<"memref.alloc">) -> !transform.any_op  
apply_patterns to %29 {  
  transform.apply_patterns.vector.transfer_to_scf    max_transfer_rank = 1 full_unroll = true  
} : !transform.any_op  
transform.iree.create_async_groups %29 {use_mma_sync} : (!transform.any_op) -> ()  
%32 = transform.structured.match ops{["nvgpu.mma.sync"]} in %29 : (!transform.any_op) ->  
!transform.any_op  
%33 = transform.loop.get_parent_for %32 : (!transform.any_op) -> !transform.any_op  
%34 = transform.iree.pipeline_shared_memory_copies %33 {depth = 3 : i64, use_mma_sync} :  
(!transform.any_op) -> !transform.any_op  
apply_patterns to %29 {  
  transform.apply_patterns.vector.lower_masks  
} : !transform.any_op  
apply_patterns to %29 {  
  transform.apply_patterns.vector.materialize_masks  
} : !transform.any_op
```

# Unaligned Matmul Case Study: Pipelining + async.copy

```
%workgroup_id_y = hal.interface.workgroup.id[1] : index
%workgroup_id_x = hal.interface.workgroup.id[0] : index
%5 = gpu.thread_id x
%6 = gpu.thread_id y

# Shared memory allocations
%alloc = memref.alloc() {alignment = 64 : i64} : memref<128x128xf32, #gpu.address_space<workgroup>>
%alloc_4 = memref.alloc() {alignment = 64 : i64} : memref<3x128x16xf32, #gpu.address_space<workgroup>>
%alloc_5 = memref.alloc() {alignment = 64 : i64} : memref<3x16x128xf32, #gpu.address_space<workgroup>>

# Prologue copy to shared memory
%58 = nvgpu.device_async_copy %0[%19, %21], %alloc_4[%c0, %20, %21], 4, %57 {bypassL1} : memref<130x1500xf32, #hal.descriptor_type<storage_buffer>> to memref<3x128x16xf32,
#gpu.address_space<workgroup>>
%109 = nvgpu.device_async_create_group %84, %86, %88, %90, %96, %100, %104, %108
nvgpu.device_async_wait %79 {numGroups = 1 : i32}
gpu.barrier

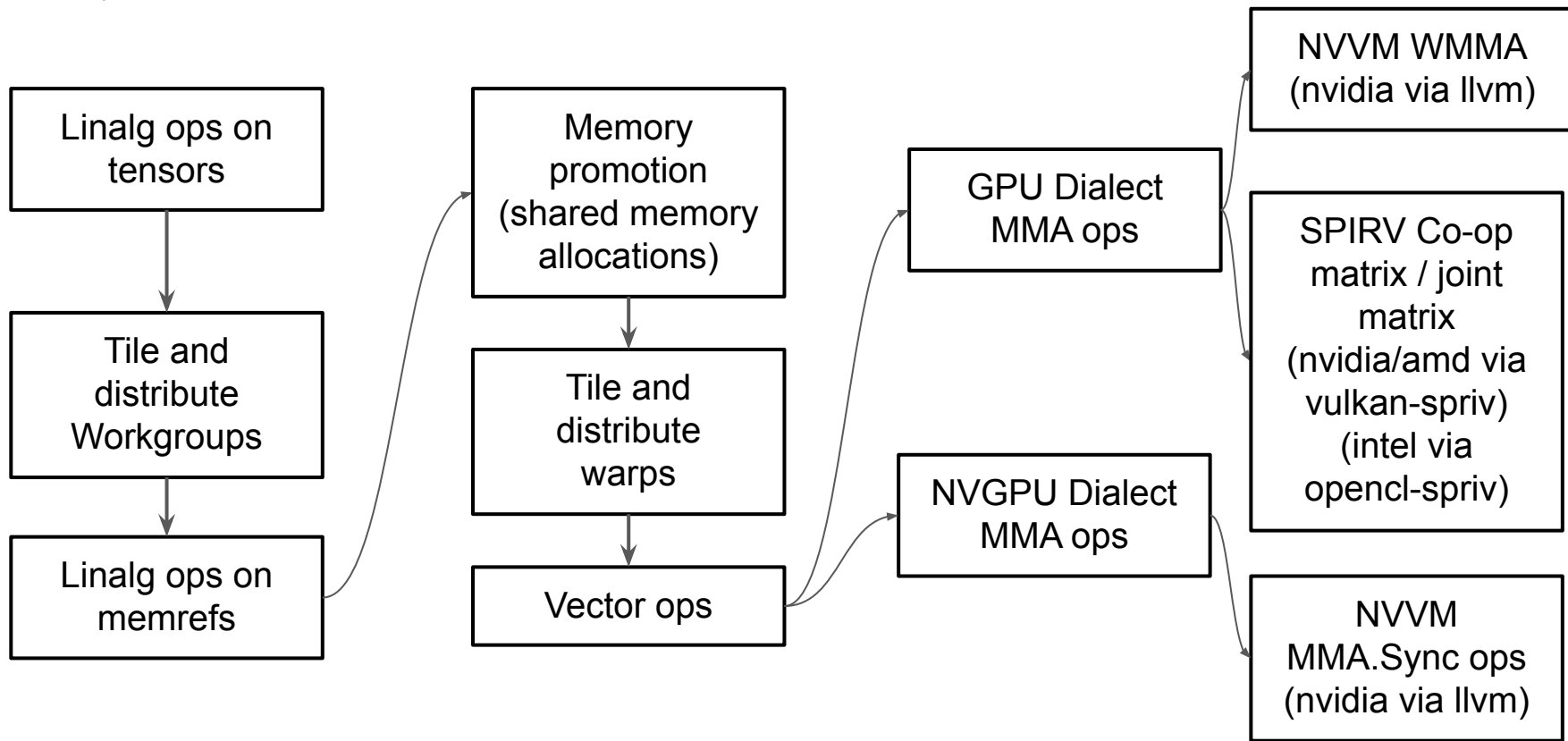
%110 = nvgpu.ldmatrix %alloc_4[%c0, %37, %38] {numTiles = 4 : i32, transpose = false} : memref<3x128x16xf32, #gpu.address_space<workgroup>> -> vector<4x1xf32>
%146:47 = scf.for %arg0 = %c0 to %c1500 step %c16 {
  # Pipelined loop over contracting dimension
  %335 = nvgpu.ldmatrix %alloc_4[%arg33, %37, %39] {numTiles = 4 : i32, transpose = false} : memref<3x128x16xf32, #gpu.address_space<workgroup>> -> vector<4x1xf32>
  %371 = nvgpu.mma.sync(%arg35, %arg36, %arg1) {mmaShape = [16, 8, 8], tf32Enabled} : (vector<4x1xf32>, vector<2x1xf32>, vector<2x2xf32>) -> vector<2x2xf32>
  %410 = nvgpu.device_async_copy %0[%19, %407], %alloc_4[%408, %20, %21], 4, %409 : memref<130x1500xf32, #hal.descriptor_type<storage_buffer>> to memref<3x128x16xf32,
#gpu.address_space<workgroup>>
  gpu.barrier
  %443 = nvgpu.ldmatrix %alloc_4[%arg34, %37, %38] {numTiles = 4 : i32, transpose = false} : memref<3x128x16xf32, #gpu.address_space<workgroup>> -> vector<4x1xf32>
  %479 = nvgpu.mma.sync(%335, %342, %371) {mmaShape = [16, 8, 8], tf32Enabled} : (vector<4x1xf32>, vector<2x1xf32>, vector<2x2xf32>) -> vector<2x2xf32>
}

# Copy back
vector.store %147, %subview[%148, %149] : memref<?x?xf32, strided<[128, 1], offset: ?>, #gpu.address_space<workgroup>>, vector<2xf32>
gpu.barrier

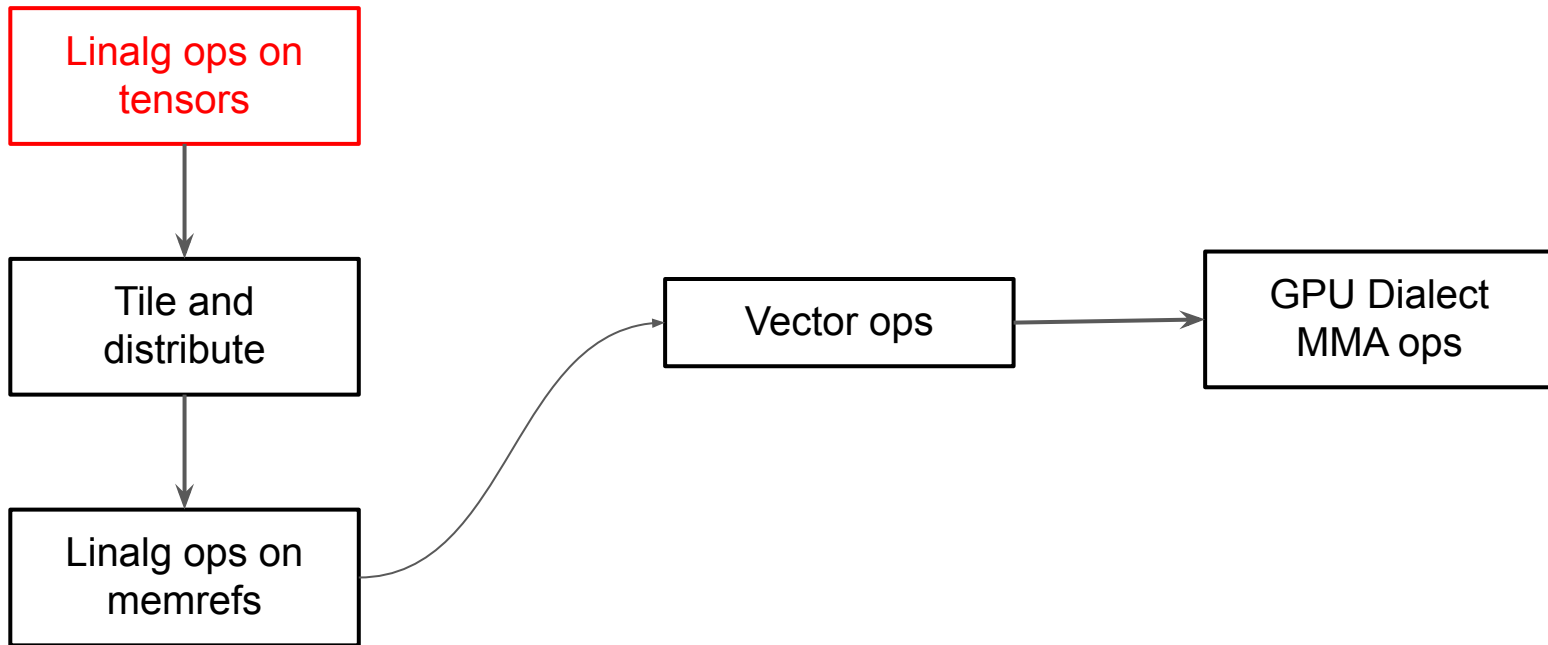
%302 = vector.transfer_read %subview_6[%c0, %c0], %cst_3, %239 {in_bounds = [true]} : memref<?x?xf32, strided<[128, 1], offset: ?>, #gpu.address_space<workgroup>>, vector<4xf32>
vector.transfer_write %302, %subview_7[%c0, %c0], %239 {in_bounds = [true]} : vector<4xf32>, memref<?x?xf32, strided<[400, 1], offset: ?>, #hal.descriptor_type<storage_buffer>>
```

# Targeting Matmul Primitives

# Compilation flow for Matmul Primitives



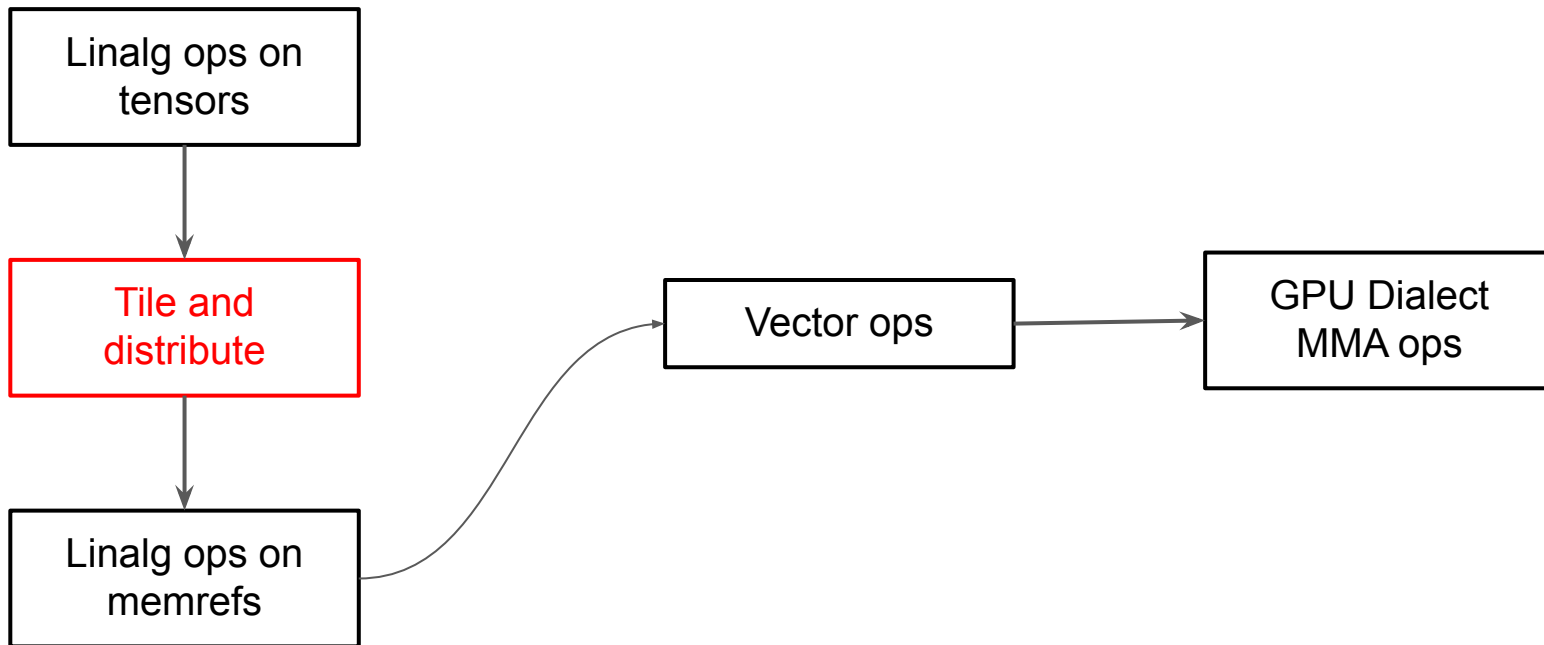
# Compilation flow for Tensor Cores (main states)



# Matmul Example: Initial Linalg

```
func @matmul(%arg0: tensor<128x1536xf16>, %arg1: tensor<1536x384xf16>, %arg2:  
tensor<128x384xf16>) -> tensor<128x384xf16> {  
    %0 = linalg.matmul ins(%arg0, %arg1: tensor<128x1536xf16>, tensor<1536x384xf16>) outs(%arg2:  
tensor<128x384xf16>) -> tensor<128x384xf16>  
    return %0 : tensor<128x384xf16>  
}
```

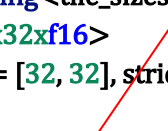
# Compilation flow for Tensor Cores (main states)





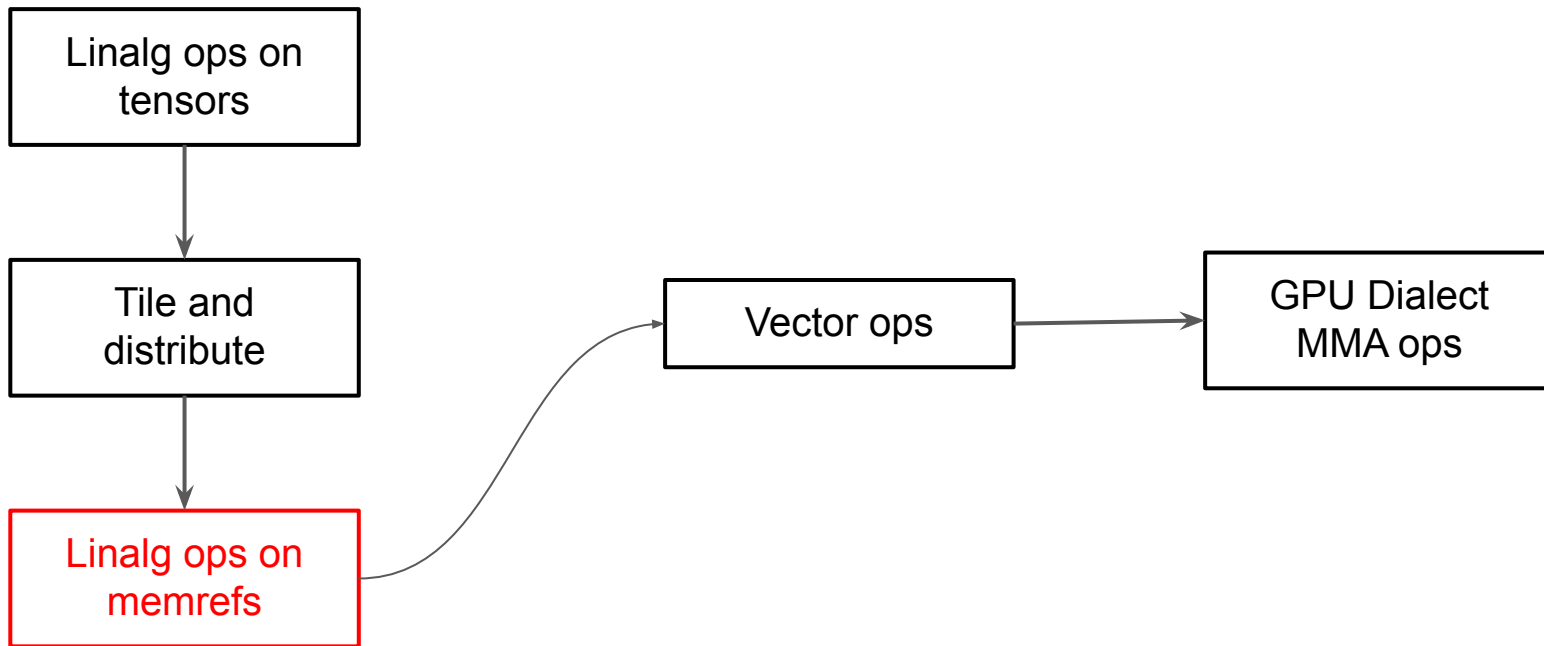
# Matmul Example: After Tile and Distribute

```
%3 = affine.apply affine_map<O[s0] -> (s0 * 32)>O[%workgroup_id_y] ← %blockId.y
%4 = affine.apply affine_map<O[s0] -> (s0 * 32)>O[%workgroup_count_y] ← %gridDim.y
scf.for %arg0 = %3 to %c128 step %4 {
  %5 = affine.apply affine_map<O[s0] -> (s0 * 32)>O[%workgroup_id_x] ← %blockId.x
  %6 = affine.apply affine_map<O[s0] -> (s0 * 32)>O[%workgroup_count_x] ← %gridDim.x
  scf.for %arg1 = %5 to %c384 step %6 {
    %7 = flow.dispatch.tensor.load %0, offsets = [%arg0, 0], sizes = [32, 1536], strides = [1, 1] : !flow.dispatch.tensor<readonly:128x1536xf16> ->
    tensor<32x1536xf16>
    %8 = flow.dispatch.tensor.load %1, offsets = [0, %arg1], sizes = [1536, 32], strides = [1, 1] : !flow.dispatch.tensor<readonly:1536x384xf16> ->
    tensor<1536x32xf16>
    %9 = flow.dispatch.tensor.load %2, offsets = [%arg0, %arg1], sizes = [32, 32], strides = [1, 1] : !flow.dispatch.tensor<readwrite:128x384xf16> ->
    tensor<32x32xf16>
    %10 = linalg.matmul {lowering_config = #iree_codegen.lowering_config<tile_sizes = [[32, 32, 16]]>} ins(%7, %8 : tensor<32x1536xf16>,
    tensor<1536x32xf16>) outs(%9 : tensor<32x32xf16>) -> tensor<32x32xf16>
    flow.dispatch.tensor.store %10, %2, offsets = [%arg0, %arg1], sizes = [32, 32], strides = [1, 1] : tensor<32x32xf16> ->
    !flow.dispatch.tensor<readwrite:128x384xf16>
  }
}
```



Tile size for M,N,K  
determines work  
done by each  
workgroup

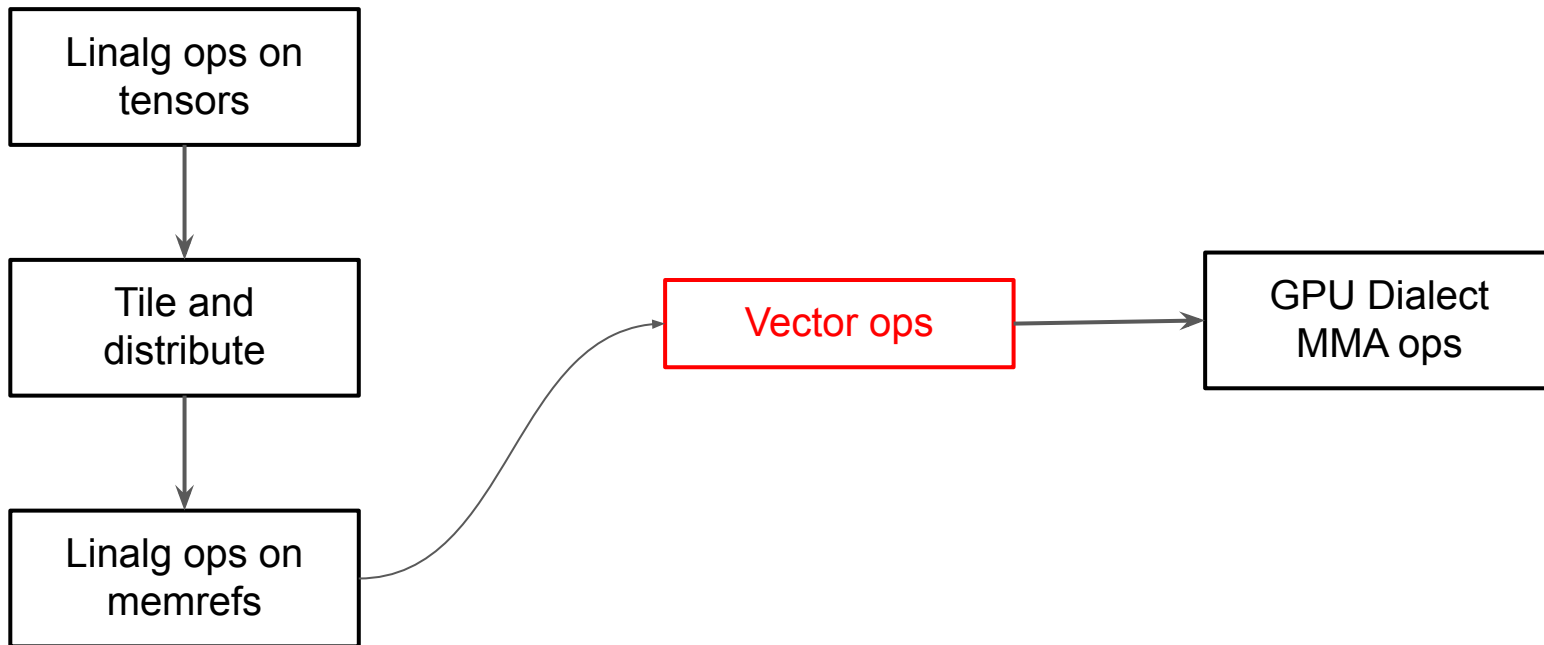
# Compilation flow for Tensor Cores (main states)



# Matmul Example: Bufferization

```
%6 = affine.apply affine_map<() [s0] -> (s0 * 32)> () [%workgroup_id_y]
%7 = affine.apply affine_map<() [s0] -> (s0 * 32)> () [%workgroup_count_y]
scf.for %arg0 = %6 to %c128 step %7 {
  %8 = affine.apply affine_map<() [s0] -> (s0 * 32)> () [%workgroup_id_x]
  %9 = affine.apply affine_map<() [s0] -> (s0 * 32)> () [%workgroup_count_x]
  scf.for %arg1 = %8 to %c384 step %9 {
    %10 = memref.subview %0[%arg0, 0] [32, 1536] [1, 1] : memref<128x1536xf16> to memref<32x1536xf16, affine_map<(d0, d1)[s0]
-> (d0 * 1536 + s0 + d1)>>
    %11 = memref.subview %2[0, %arg1] [1536, 32] [1, 1] : memref<1536x384xf16> to memref<1536x32xf16, affine_map<(d0, d1)[s0]
-> (d0 * 384 + s0 + d1)>>
    %12 = memref.subview %4[%arg0, %arg1] [32, 32] [1, 1] : memref<128x384xf16> to memref<32x32xf16, affine_map<(d0, d1)[s0] ->
(d0 * 384 + s0 + d1)>>
    linalg.matmul {lowering_config = #iree_codegen.lowering_config<tile_sizes = [[32, 32, 16]]>} ins(%10, %11 : memref<32x1536xf16,
affine_map<(d0, d1)[s0] -> (d0 * 1536 + s0 + d1)>>, memref<1536x32xf16, affine_map<(d0, d1)[s0] -> (d0 * 384 + s0 + d1)>>)
outs(%12 : memref<32x32xf32, affine_map<(d0, d1)[s0] -> (d0 * 384 + s0 + d1)>>)
  }
}
```

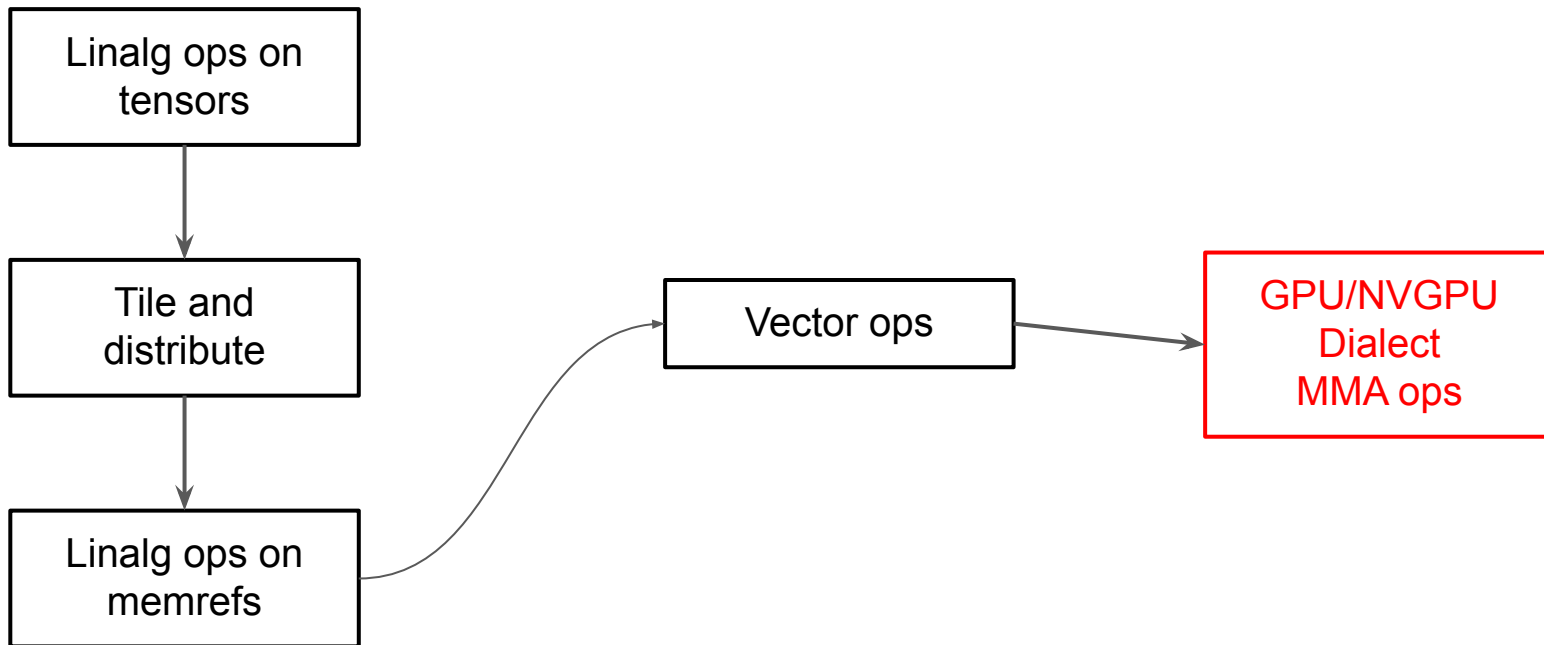
# Compilation flow for Tensor Cores (main states)



# Matmul Example: Vector Ops

```
%37 = 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>} %33, %35, %arg1 : vector<16x16xf16>, vector<16x16xf16> into vector<16x16xf16>  
  
%38 = 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>} %34, %36, %37 : vector<16x16xf16>, vector<16x16xf16> into vector<16x16xf16>
```

# Compilation flow for Tensor Cores (main states)



# Matmul Example: GPU Ops

## GPU ops

```
%29 = gpu.subgroup_mma_load_matrix %4[%28, %27, %c0] {leadDimension = 20 : index} : memref<4x32x20xf16, 3> ->
!gpu.mma_matrix<16x16xf16, "AOp">
%35 = gpu.subgroup_mma_load_matrix %3[%34, %c0, %33] {leadDimension = 36 : index} : memref<4x16x36xf16, 3> ->
!gpu.mma_matrix<16x16xf16, "BOp">
%39 = gpu.subgroup_mma_compute %29, %35, %arg1 : !gpu.mma_matrix<16x16xf16, "AOp">, !gpu.mma_matrix<16x16xf16, "BOp"> ->
!gpu.mma_matrix<16x16xf16, "COp">
```

## NVGPU ops

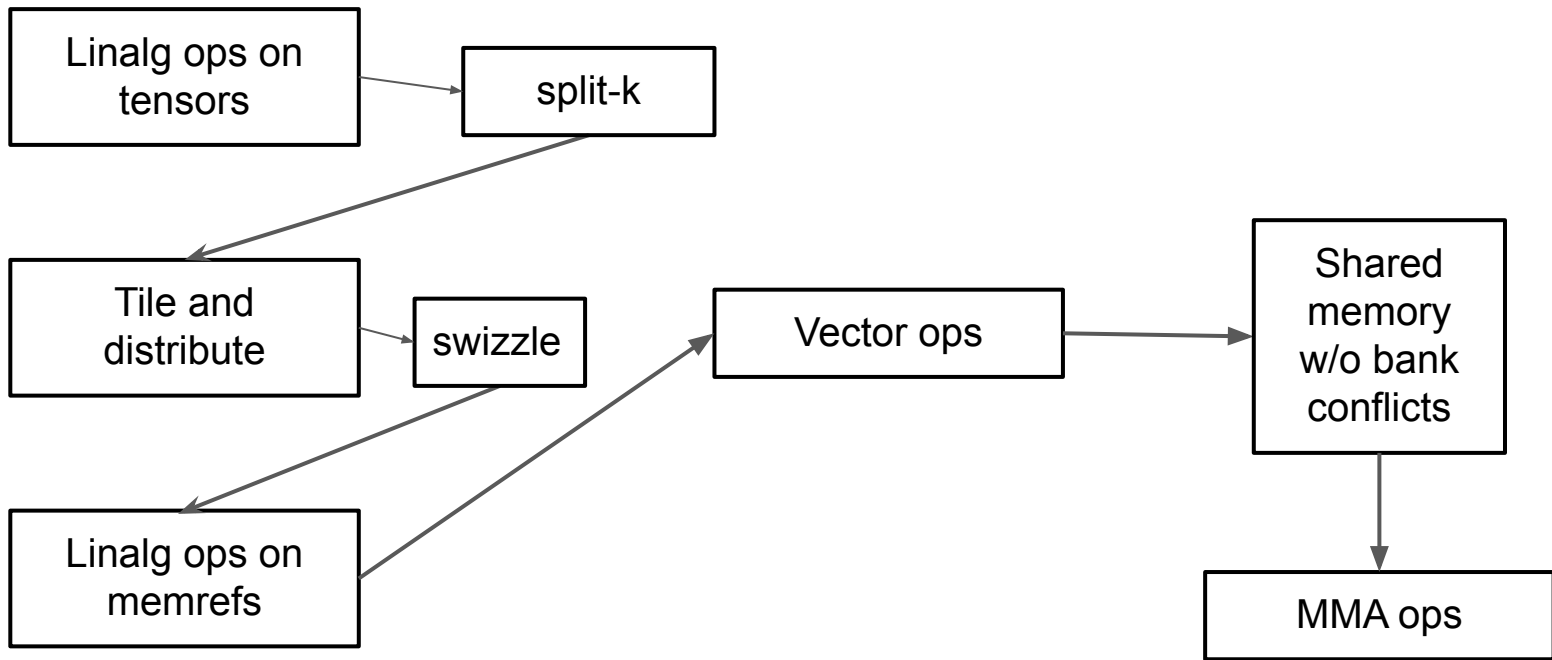
```
%16 = nvgpu.ldmatrix %alloc_1[%14, %15] {numTiles = 4 : i32, transpose = false} : memref<32x32xf16, #gpu.address_space<workgroup>> ->
vector<4x2xf16>
%21 = nvgpu.ldmatrix %alloc_2[%20, %19] {numTiles = 4 : i32, transpose = true} : memref<32x32xf16, #gpu.address_space<workgroup>> ->
vector<4x2xf16>
%22 = affine.apply affine_map<>()[s0] -> (s0 mod 16 + 16)>()[%13]
%23 = nvgpu.ldmatrix %alloc_2[%22, %19] {numTiles = 4 : i32, transpose = true} : memref<32x32xf16, #gpu.address_space<workgroup>> ->
vector<4x2xf16>
%24 = vector.extract_strided_slice %21 {offsets = [0, 0], sizes = [2, 2], strides = [1, 1]} : vector<4x2xf16> to vector<2x2xf16>
%25 = nvgpu.mma.sync(%16, %24, %cst) {mmaShape = [16, 8, 16]} : (vector<4x2xf16>, vector<2x2xf16>, vector<2x2xf16>) -> vector<2x2xf16>
```

# Optimizations

- Matmul specific
  - Split-k
  - Thread block Swizzling
  - Shared memory use without bank conflicts
  - Software pipelining and async copy
- General
  - Warp reductions
  - Caching allocator (runtime optimization)



# Optimizations



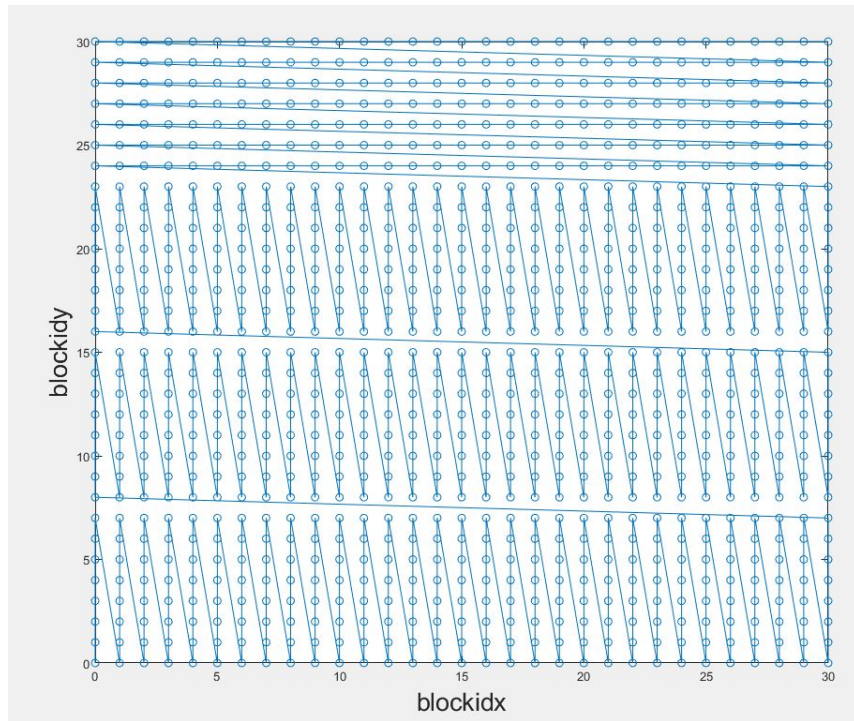
# Split-k

```
%3 = tensor.expand_shape %0 [[0], [1, 2]] : tensor<128x1536xf16> into tensor<128x16x96xf16>
%4 = tensor.expand_shape %1 [[0, 1], [2]] : tensor<1536x384xf16> into tensor<16x96x384xf16>
%5 = linalg.init_tensor [16, 128, 384] : tensor<16x128x384xf16>
%6 = linalg.fill ins(%cst : f16) outs(%5 : tensor<16x128x384xf16>) -> tensor<16x128x384xf16>
%7 = linalg.generic ins(%3, %4 : tensor<128x16x96xf16>, tensor<16x96x384xf16>) outs(%6 : tensor<16x128x384xf16>){
  ^bb0(%arg3: f16, %arg4: f16, %arg5: f16):
    %10 = arith.mulf %arg3, %arg4 : f16
    %11 = arith.addf %arg5, %10 : f16
    linalg.yield %11 : f16
} -> tensor<16x128x384xf16>
%8 = linalg.generic ins(%7 : tensor<16x128x384xf16>) outs(%2 : tensor<128x384xf16>){
  ^bb0(%arg3: f16, %arg4: f16):
    %10 = arith.addf %arg3, %arg4 : f16
    linalg.yield %10 : f16
} -> tensor<128x384xf16>
```

# Thread block swizzling

We are experimenting with the following swizzling logic:

```
void getTiledId(unsigned x, unsigned y, unsigned  
*tiledx, unsigned *tiledy) {  
    unsigned t_tiledx = (x + (y % tile) * grid_size_x) / tile;  
    unsigned t_tiledy = (y / tile) * tile + (x + (y % tile) *  
grid_size_x) % tile;  
    bool c = grid_size_y % tile != 0 && ((y / tile) * tile +  
tile) > grid_size_y;  
    *tiledx = c ? x : t_tiledx;  
    *tiledy = c ? y : t_tiledy;  
}
```



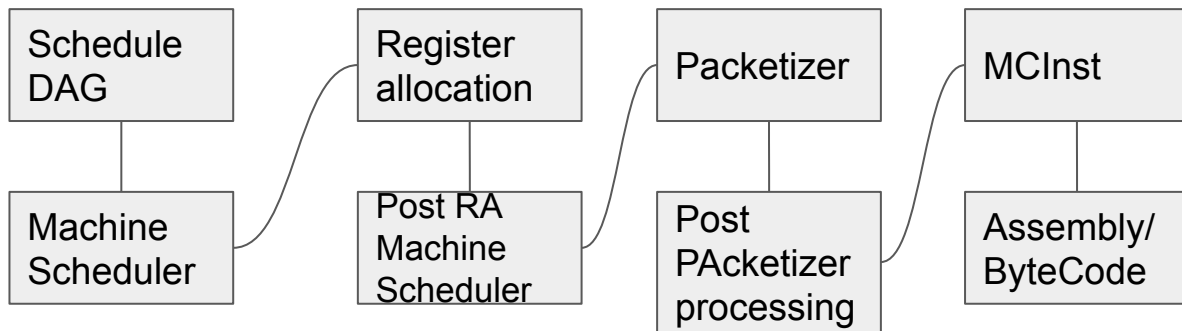
Implicit GEMM

(shown as part of a separate slide deck)

# VLIW Codegen

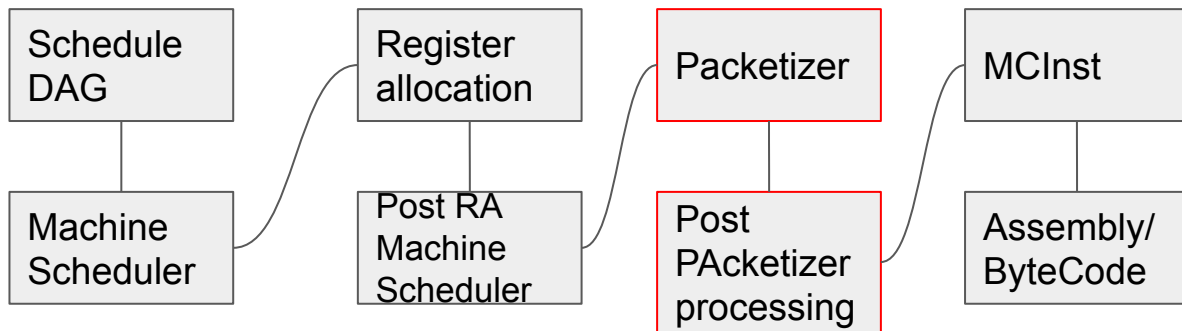
# VLIW LLVM Backend

- Selection DAG ISEL:
  - LLVM IR with vendor intrinsics -> Selection DAG -> Machine Instr
- Scheduling/Register allocation/ Packetization



# VLIW LLVM Backend

- All stages not marked in red are done by carefully adding target specific details in tablegen files and adding some functions enabling existing llvm infrastructure
- New passes are needed for stages marked in red



# Packetizer

- Follows the schedulers ordering
- Starts new packet when it runs out of slots
- Checks for dependencies, if not satisfied start new packet



# Post Packet processing

- For a fixed length VLIW, if there are empty slots they are filled with NOPs at this stage
- The Machine Instructions (MIs) are also ordered according to their slot numbers
- NOP packets are inserted if the hardware does not have hard detection logic so that latencies of instructions are always satisfied

Questions?