Upstream SPIR-V Conversion

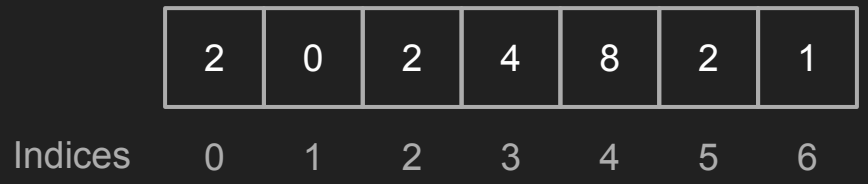
MLIR Open Design Meeting August 22, 2024

Angel Zhang anzhouzhang913@gmail.com

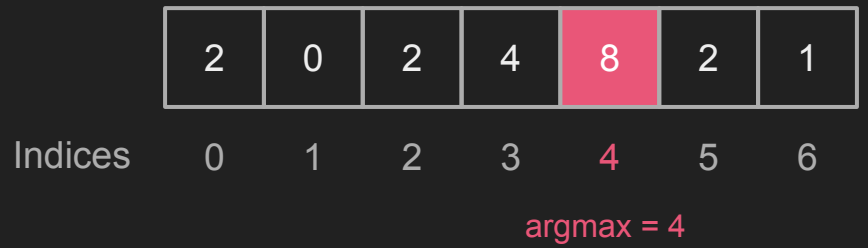
SPIR-V

- Khronos Group’s intermediate language for representing graphics shaders and compute kernels
- Consumed by APIs like Vulkan, OpenGL and OpenCL
- Enables various high-level language front-ends to run on diverse hardware architectures
- Fully defined in a human-readable specification

Argmax



Argmax



“Argmax” Example

GLSL

```
// A simple argmax kernel.
void main() {
  uint laneID = gl_LocalInvocationID.x;
  uint laneCount = gl_WorkGroupSize.x;

  uint laneResult = 0;
  float laneMax = Input.data[laneID];

  uint numBatches = totalCount / laneCount;

  for (uint i = 1; i < numBatches; ++i) {
    uint idx = laneCount * i + laneID;
    float elem = Input.data[idx];
    if (elem > laneMax) {
      laneResult = idx;
      laneMax = elem;
    }
  }

  // Find the max of workgroup (containing only one subgroup).
  float wgMax = subgroupMax(laneMax);

  // Find the smallest thread ID with the max element.
  bool bit = laneMax == wgMax;
  uvec4 mask = subgroupBallot(bit);
  uint smallestID = subgroupBallotFindLSB(mask);

  // The thread is responsible for outputing result.
  if (laneID == smallestID)
    Output.data = laneResult;
}
```

“Argmax” Example

GLSL

```
// A simple argmax kernel.
void main() {
  uint laneID = gl_LocalInvocationID.x;
  uint laneCount = gl_WorkGroupSize.x;

  uint laneResult = 0;
  float laneMax = Input.data[laneID];

  uint numBatches = totalCount / laneCount;

  for (uint i = 1; i < numBatches; ++i) {
    uint idx = laneCount * i + laneID;
    float elem = Input.data[idx];
    if (elem > laneMax) {
      laneResult = idx;
      laneMax = elem;
    }
  }

  // Find the max of workgroup (containing only one subgroup).
  float wgMax = subgroupMax(laneMax);

  // Find the smallest thread ID with the max element.
  bool bit = laneMax == wgMax;
  uvec4 mask = subgroupBallot(bit);
  uint smallestID = subgroupBallotFindLSB(mask);

  // The thread is responsible for outputing result.
  if (laneID == smallestID)
    Output.data = laneResult;
}
```

“Argmax” Example

GLSL

```
// A simple argmax kernel.
void main() {
  uint laneID = gl_LocalInvocationID.x;
  uint laneCount = gl_WorkGroupSize.x;

  uint laneResult = 0;
  float laneMax = Input.data[laneID];

  uint numBatches = totalCount / laneCount;

  for (uint i = 1; i < numBatches; ++i) {
    uint idx = laneCount * i + laneID;
    float elem = Input.data[idx];
    if (elem > laneMax) {
      laneResult = idx;
      laneMax = elem;
    }
  }

  // Find the max of workgroup (containing only one subgroup).
  float wgMax = subgroupMax(laneMax);

  // Find the smallest thread ID with the max element.
  bool bit = laneMax == wgMax;
  uvec4 mask = subgroupBallot(bit);
  uint smallestID = subgroupBallotFindLSB(mask);

  // The thread is responsible for outputing result.
  if (laneID == smallestID)
    Output.data = laneResult;
}
```

“Argmax” Example

GLSL

```
// A simple argmax kernel.
void main() {
  uint laneID = gl_LocalInvocationID.x;
  uint laneCount = gl_WorkGroupSize.x;

  uint laneResult = 0;
  float laneMax = Input.data[laneID];

  uint numBatches = totalCount / laneCount;

  for (uint i = 1; i < numBatches; ++i) {
    uint idx = laneCount * i + laneID;
    float elem = Input.data[idx];
    if (elem > laneMax) {
      laneResult = idx;
      laneMax = elem;
    }
  }

  // Find the max of workgroup (containing only one subgroup).
  float wgMax = subgroupMax(laneMax);

  // Find the smallest thread ID with the max element.
  bool bit = laneMax == wgMax;
  uvec4 mask = subgroupBallot(bit);
  uint smallestID = subgroupBallotFindLSB(mask);

  // The thread is responsible for outputing result.
  if (laneID == smallestID)
    Output.data = laneResult;
}
```

“Argmax” Example

SPIR-V Assembly

```
; Argmax kernel function definition.
%4 = OpFunction %void None %3
%5 = OpLabel
     OpSelectionMerge %67 None
     OpSwitch %uint_0 %68
%68 = OpLabel
%14 = OpAccessChain %_ptr_Input_uint %gl_LocalInvocationID %uint_0
%15 = OpLoad %uint %14
%18 = OpINotEqual %bool %15 %uint_0
     OpSelectionMerge %20 None
     OpBranchConditional %18 %19 %20
%19 = OpLabel
     OpBranch %67
%20 = OpLabel
%33 = OpAccessChain %_ptr_StorageBuffer_float %29 %int_0 %int_0
%34 = OpLoad %float %33
     OpBranch %37
%37 = OpLabel
%75 = OpPhi %float %34 %20 %80 %38
%74 = OpPhi %uint %uint_0 %20 %81 %38
%73 = OpPhi %uint %uint_1 %20 %58 %38
%44 = OpULessThan %bool %73 %43
     OpLoopMerge %39 %38 None
     OpBranchConditional %44 %38 %39
%38 = OpLabel
%47 = OpAccessChain %_ptr_StorageBuffer_float %29 %int_0 %73
%48 = OpLoad %float %47
%51 = OpFOrdGreaterThan %bool %48 %75
%80 = OpSelect %float %51 %48 %75
%81 = OpSelect %uint %51 %73 %74
%58 = OpIAdd %uint %73 %int_1
     OpBranch %37
%39 = OpLabel
%64 = OpAccessChain %_ptr_StorageBuffer_uint %61 %int_0
     OpStore %64 %74
     OpBranch %67
%67 = OpLabel
     OpReturn
     OpFunctionEnd
```

“Argmax” Example

MLIR SPIR-V Dialect

```
spirv.mlir.loop {
  spirv.Branch ^bb1(%cst1_i32, %cst0_i32, %3 : i32, i32, f32)
^bb1(%13: i32, %14: i32, %15: f32):  // 2 preds: ^bb0, ^bb2
  %16 = spirv.SLessThan %13, %cst0_i32 : i32
  spirv.BranchConditional %16, ^bb2, ^bb3
^bb2:  // pred: ^bb1
  %17 = spirv.IMul %13, %cst32_i32 : i32
  %18 = spirv.IAdd %17, %1 : i32
  %19 = spirv.AccessChain %arg0[%cst0_i32, %18] : !spirv.ptr<...>, i32, i32
  %20 = spirv.Load "StorageBuffer" %19 : f32
  %21 = spirv.FOrdGreaterThan %20, %15 : f32
  %22 = spirv.Select %21, %18, %14 : i1, i32
  %23 = spirv.Select %21, %20, %15 : i1, f32
  spirv.Store "Function" %4, %22 : i32
  spirv.Store "Function" %5, %23 : f32
  %24 = spirv.IAdd %13, %cst1_i32 : i32
  spirv.Branch ^bb1(%24, %22, %23 : i32, i32, f32)
^bb3:  // pred: ^bb1
  spirv.mlir.merge
}
%6 = spirv.Load "Function" %5 : f32
%7 = spirv.Load "Function" %4 : i32
%8 = spirv.GroupNonUniformFMax "Subgroup" "Reduce" %6 : f32
%9 = spirv.FOrdEqual %6, %8 : f32
%10 = spirv.GroupNonUniformBallot <Subgroup> %9 : vector<4xi32>
%11 = spirv.GroupNonUniformBallotFindLSB <Subgroup> %10 : vector<4xi32>, i32
%12 = spirv.IEqual %1, %cst1_i32 : i32
spirv.mlir.selection {
  spirv.BranchConditional %12, ^bb1, ^bb2
^bb1:  // pred: ^bb0
  %13 = spirv.AccessChain %arg1[%cst0_i32, %cst0_i32] : !spirv.ptr<...>, i32, i32
  spirv.Store "StorageBuffer" %13, %7 : i32
  spirv.Branch ^bb2
^bb2:  // 2 preds: ^bb0, ^bb1
  spirv.mlir.merge
}
```

“Argmax” Example

MLIR SPIR-V Dialect

```
spirv.mlir.loop {
  spirv.Branch ^bb1(%cst1_i32, %cst0_i32, %3 : i32, i32, f32)
^bb1(%13: i32, %14: i32, %15: f32):  // 2 preds: ^bb0, ^bb2
  %16 = spirv.SLessThan %13, %cst0_i32 : i32
  spirv.BranchConditional %16, ^bb2, ^bb3
^bb2:  // pred: ^bb1
  %17 = spirv.IMul %13, %cst32_i32 : i32
  %18 = spirv.IAdd %17, %1 : i32
  %19 = spirv.AccessChain %arg0[%cst0_i32, %18] : !spirv.ptr<...>, i32, i32
  %20 = spirv.Load "StorageBuffer" %19 : f32
  %21 = spirv.FOrdGreaterThan %20, %15 : f32
  %22 = spirv.Select %21, %18, %14 : i1, i32
  %23 = spirv.Select %21, %20, %15 : i1, f32
  spirv.Store "Function" %4, %22 : i32
  spirv.Store "Function" %5, %23 : f32
  %24 = spirv.IAdd %13, %cst1_i32 : i32
  spirv.Branch ^bb1(%24, %22, %23 : i32, i32, f32)
^bb3:  // pred: ^bb1
  spirv.mlir.merge
}
%6 = spirv.Load "Function" %5 : f32
%7 = spirv.Load "Function" %4 : i32
%8 = spirv.GroupNonUniformFMax "Subgroup" "Reduce" %6 : f32
%9 = spirv.FOrdEqual %6, %8 : f32
%10 = spirv.GroupNonUniformBallot <Subgroup> %9 : vector<4xi32>
%11 = spirv.GroupNonUniformBallotFindLSB <Subgroup> %10 : vector<4xi32>, i32
%12 = spirv.IEqual %1, %cst1_i32 : i32
spirv.mlir.selection {
  spirv.BranchConditional %12, ^bb1, ^bb2
^bb1:  // pred: ^bb0
  %13 = spirv.AccessChain %arg1[%cst0_i32, %cst0_i32] : !spirv.ptr<...>, i32, i32
  spirv.Store "StorageBuffer" %13, %7 : i32
  spirv.Branch ^bb2
^bb2:  // 2 preds: ^bb0, ^bb1
  spirv.mlir.merge
}
```

```
// An argmax kernel.
void main() {
  uint laneID = gl_LocalInvocationID.x;
  uint laneCount = gl_WorkGroupSize.x;

  uint laneResult = 0;
  float laneMax = Input.data[laneID];

  uint numBatches = totalCount / laneCount;

  for (uint i = 1; i < numBatches; ++i) {
    uint idx = laneCount * i + laneID;
    float elem = Input.data[idx];
    if (elem > laneMax) {
      laneResult = idx;
      laneMax = elem;
    }
  }

  // Find the max of workgroup (containing only one subgroup).
  float wgMax = subgroupMax(laneMax);

  // Find the smallest thread ID with the max element.
  bool bit = laneMax == wgMax;
  uvec4 mask = subgroupBallot(bit);
  uint smallestID = subgroupBallotFindLSB(mask);

  // The thread is responsible for outputing result.
  if (laneID == smallestID)
    Output.data = laneResult;
}
```

```
// The MLIR representation.
%num_batches = arith.divui %cst_1_i32, %cst_32 : i32
%tx = gpu.thread_id x
%tx_i32 = index.castu %tx : index to i32
%ub = index.castu %num_batches : i32 to index
%lane_res_init = arith.constant 0 : i32
%lane_max_init = memref.load %input[%tx] : memref<4xf32>

%lane_res, %lane_max = scf.for %iter = %cst_1_idx to %ub step %cst_1_idx
  iter_args(%lane_res_iter = %lane_res_init, %lane_max_iter = %lane_max_init)
-> (i32, f32) {
  %iter_i32 = index.castu %iter : index to i32
  %mul = arith.muli %cst_32, %iter_i32 : i32
  %idx_i32 = arith.addi %mul, %tx_i32 : i32
  %idx = index.castu %idx_i32 : i32 to index
  %elem = memref.load %input[%idx] : memref<4xf32>
  %gt = arith.cmpf ogt, %elem, %lane_max_iter : f32
  %lane_res_next = arith.select %gt, %idx_i32, %lane_res_iter : i32
  %lane_max_next = arith.select %gt, %elem, %lane_max_iter : f32
  scf.yield %lane_res_next, %lane_max_next : i32, f32
}
%subgroup_max = gpu.subgroup_reduce maximumf %lane_max : (f32) -> (f32)
%eq = arith.cmpf oeq, %lane_max, %subgroup_max : f32
%ballot = spirv.GroupNonUniformBallot <Subgroup> %eq : vector<4xi32>
%lsb = spirv.GroupNonUniformBallotFindLSB <Subgroup> %ballot : vector<4xi32>, i32
%cond = arith.cmpi eq, %cst_1_i32, %tx_i32 : i32

scf.if %cond {
  memref.store %lane_res, %output[] : memref<i32>
}
```

```
// An argmax kernel.
void main() {
  uint laneID = gl_LocalInvocationID.x;
  uint laneCount = gl_WorkGroupSize.x;

  uint laneResult = 0;
  float laneMax = Input.data[laneID];

  uint numBatches = totalCount / laneCount;

  for (uint i = 1; i < numBatches; ++i) {
    uint idx = laneCount * i + laneID;
    float elem = Input.data[idx];
    if (elem > laneMax) {
      laneResult = idx;
      laneMax = elem;
    }
  }

  // Find the max of workgroup (containing only one subgroup).
  float wgMax = subgroupMax(laneMax);

  // Find the smallest thread ID with the max element.
  bool bit = laneMax == wgMax;
  uvec4 mask = subgroupBallot(bit);
  uint smallestID = subgroupBallotFindLSB(mask);

  // The thread is responsible for outputing result.
  if (laneID == smallestID)
    Output.data = laneResult;
}
```

```
// The MLIR representation.
%num_batches = arith.divui %cst_1_i32, %cst_32 : i32
%tx = gpu.thread_id x
%tx_i32 = index.castu %tx : index to i32
%ub = index.castu %num_batches : i32 to index
%lane_res_init = arith.constant 0 : i32
%lane_max_init = memref.load %input[%tx] : memref<4xf32>

%lane_res, %lane_max = scf.for %iter = %cst_1_idx to %ub step %cst_1_idx
  iter_args(%lane_res_iter = %lane_res_init, %lane_max_iter = %lane_max_init)
-> (i32, f32) {
  %iter_i32 = index.castu %iter : index to i32
  %mul = arith.muli %cst_32, %iter_i32 : i32
  %idx_i32 = arith.addi %mul, %tx_i32 : i32
  %idx = index.castu %idx_i32 : i32 to index
  %elem = memref.load %input[%idx] : memref<4xf32>
  %gt = arith.cmpf ogt, %elem, %lane_max_iter : f32
  %lane_res_next = arith.select %gt, %idx_i32, %lane_res_iter : i32
  %lane_max_next = arith.select %gt, %elem, %lane_max_iter : f32
  scf.yield %lane_res_next, %lane_max_next : i32, f32
}
%subgroup_max = gpu.subgroup_reduce maximumf %lane_max : (f32) -> (f32)
%eq = arith.cmpf oeq, %lane_max, %subgroup_max : f32
%ballot = spirv.GroupNonUniformBallot <Subgroup> %eq : vector<4xi32>
%lsb = spirv.GroupNonUniformBallotFindLSB <Subgroup> %ballot : vector<4xi32>, i32
%cond = arith.cmpi eq, %cst_1_i32, %tx_i32 : i32

scf.if %cond {
  memref.store %lane_res, %output[] : memref<i32>
}
```

SPIR-V Conversion?

- Individual dialect conversions available in tree
  - Arith, Func, GPU, Index, MemRef, Tensor, Vector, …
- No upstream conversion to SPIR-V
  - Only in IREE
- Want to have a lowering pipeline, similar to `ConvertToLLVM`

Convert To SPIR-V

- Generic MLIR lowering pass to SPIR-V
- Goals:
  - Better test coverage of SPIR-V compilation upstream
    - Previous discussion: Open MLIR Meeting - Vector dialect, reshape, and handling of unit dimensions
  - Write simple kernels by hand (e.g., argmax)

Supported Input Dialects

- Arith
- Builtin
- Func
- GPU (in progress)
- Index
- MemRef
- Vector
- SCF
- SPIR-V
- UB

Convert To SPIR-V: Complexities

- Individual Dialect Conversion Patterns
- Vector Type Legalization
  - Signature Conversion
  - Function Body Vector Unrolling
- MemRef Types to SPIR-V Storage Classes
- Lower ops like `vector.maskedload`/`maskedstore` to simpler vector ops
  - Currently not available in SPIR-V dialect
- Support narrow element types (e.g., `i8`) by performing emulation
- Option of emulating `i64` for targets without 64-bit integers

Interface-based Approach?

- `ConvertToLLVM` uses an interface-based approach to delegate to dialects the injection of conversion patterns

```
for (Dialect *dialect : context->getLoadedDialects()) {
  auto *iface = dyn_cast<ConvertToLLVMPatternInterface>(dialect);
  if (!iface)
    continue;
  iface->populateConvertToLLVMConversionPatterns(*target, *typeConverter, tempPatterns);
}
```

- Attempted to use interfaces for `ConvertToSPIRV` as well
  - #102046 Update the ConvertToSPIRV pass to use dialect interfaces
- Due to complexities, decided to start with a multi-stage lowering approach

Signature Conversion: Introduction

- Goal: Unroll vectors into 1-D, with a length of 2/3/4
- Why?
  - SPIR-V only supports 1-D vectors of length 2/3/4
  - Length 1 ⇒ scalars
  - Length 8/16 ⇒ available via the `Vector16` capability

Signature Conversion: Examples

```
func.func @simple_vector_6(%arg0 : vector<6xi32>) -> vector<6xi32>
==>
func.func @simple_vector_6(%arg0: vector<3xi32>, %arg1: vector<3xi32>) -> (vector<3xi32>, vector<3xi32>)

// -----

func.func @simple_vector_2d(%arg0 : vector<2x4xi32>) -> vector<2x4xi32>
==>
func.func @simple_vector_2d(%arg0: vector<4xi32>, %arg1: vector<4xi32>) -> (vector<4xi32>, vector<4xi32>)

// -----

func.func @vector_3and8(%arg0 : vector<3xi32>, %arg1 : vector<8xi32>) -> (vector<3xi32>, vector<8xi32>)
==>
func.func @vector_3and8(%arg0: vector<3xi32>, %arg1: vector<4xi32>, %arg2: vector<4xi32>) -> (vector<3xi32>,
vector<4xi32>, vector<4xi32>)
```

Signature Conversion: Compute Target Vector Size

```
// Utilizing mlir/lib/Dialect/Vector/Transforms/VectorUnroll.cpp
int getComputeVectorSize(int64_t size) {
  for (int i : {4, 3, 2}) {
    if (size % i == 0)
      return i;
  }
  return 1;
}
```

Signature Conversion: Get Target Shape

```
// Utilizing mlir/lib/Dialect/Vector/Transforms/VectorUnroll.cpp
std::optional<SmallVector<int64_t>> getTargetShape(VectorType vecType) {
  SmallVector<int64_t, 4> unrollShape = llvm::to_vector<4>(vecType.getShape());
  std::optional<SmallVector<int64_t>> targetShape = SmallVector<int64_t>(
      1, mlir::spirv::getComputeVectorSize(vecType.getShape().back()));
  if (!targetShape)
    return std::nullopt;
  auto maybeShapeRatio = computeShapeRatio(unrollShape, *targetShape);
  if (!maybeShapeRatio)
    return std::nullopt;
  if (llvm::all_of(*maybeShapeRatio, [](int64_t v) { return v == 1; }))
    return std::nullopt;
  return targetShape;
}
```

Signature Conversion: Caveats

- Using existing utility functions from `VectorUnroll.cpp`
- Easy to implement but not the optimal solution
- Consider:

```
vector<5xi32>
==>
vector<1xi32>, vector<1xi32>, vector<1xi32>, vector<1xi32>, vector<1xi32>
```

Signature Conversion: Caveats

- Using existing utility functions from `VectorUnroll.cpp`
- Easy to implement but not the optimal solution
- Consider:

```
vector<5xi32>
==>
vector<1xi32>, vector<1xi32>, vector<1xi32>, vector<1xi32>, vector<1xi32>
```

vs.

```
vector<5xi32>
==>
vector<4xi32>, vector<1xi32>
```

Signature Conversion: Input & Output Conversion

- Not a vector / a vector of legal size ⇒
  - No need to unroll
- Vector of illegal size in input ⇒
  - New `vector.insert_strided_slice` ops
  - Insert target types into original type
- Vector of illegal size in output ⇒
  - New `vector.extract_strided_slice` ops
  - Extract target types from original shape

Signature Conversion: Example

```
func.func @foo(%arg0 : vector<8xi32>) -> vector<8xi32> {
  %0 = ... %arg0 ...
  return %0 : vector<8xi32>
}

==>

func.func @foo(%arg0: vector<4xi32>, %arg1: vector<4xi32>) -> (vector<4xi32>, vector<4xi32>) {
  %cst = arith.constant dense<0> : vector<8xi32>
  %0 = vector.insert_strided_slice %arg0, %cst {offsets = [0], strides = [1]} : vector<4xi32> into vector<8xi32>
  %1 = vector.insert_strided_slice %arg1, %0 {offsets = [4], strides = [1]} : vector<4xi32> into vector<8xi32>
  ...
  %2 = ... %1 ...
  ...
  %3 = vector.extract_strided_slice %2 {offsets = [0], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  %4 = vector.extract_strided_slice %2 {offsets = [4], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  return %3, %4 : vector<4xi32>, vector<4xi32>
}
```

Signature Conversion: Example

```
func.func @foo(%arg0 : vector<8xi32>) -> vector<8xi32> {
  %0 = ... %arg0 ...
  return %0 : vector<8xi32>
}

==>

func.func @foo(%arg0: vector<4xi32>, %arg1: vector<4xi32>) -> (vector<4xi32>, vector<4xi32>) {
  %cst = arith.constant dense<0> : vector<8xi32>
  %0 = vector.insert_strided_slice %arg0, %cst {offsets = [0], strides = [1]} : vector<4xi32> into vector<8xi32>
  %1 = vector.insert_strided_slice %arg1, %0 {offsets = [4], strides = [1]} : vector<4xi32> into vector<8xi32>
  ...
  %2 = ... %1 ...
  ...
  %3 = vector.extract_strided_slice %2 {offsets = [0], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  %4 = vector.extract_strided_slice %2 {offsets = [4], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  return %3, %4 : vector<4xi32>, vector<4xi32>
}
```

Signature Conversion: Example

```
func.func @foo(%arg0 : vector<8xi32>) -> vector<8xi32> {
  %0 = ... %arg0 ...
  return %0 : vector<8xi32>
}

==>

func.func @foo(%arg0: vector<4xi32>, %arg1: vector<4xi32>) -> (vector<4xi32>, vector<4xi32>) {
  %cst = arith.constant dense<0> : vector<8xi32>
  %0 = vector.insert_strided_slice %arg0, %cst {offsets = [0], strides = [1]} : vector<4xi32> into vector<8xi32>
  %1 = vector.insert_strided_slice %arg1, %0 {offsets = [4], strides = [1]} : vector<4xi32> into vector<8xi32>
  ...
  %2 = ... %1 ...
  ...
  %3 = vector.extract_strided_slice %2 {offsets = [0], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  %4 = vector.extract_strided_slice %2 {offsets = [4], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  return %3, %4 : vector<4xi32>, vector<4xi32>
}
```

Signature Conversion: Example

```
func.func @foo(%arg0 : vector<8xi32>) -> vector<8xi32> {
  %0 = ... %arg0 ...
  return %0 : vector<8xi32>
}

==>

func.func @foo(%arg0: vector<4xi32>, %arg1: vector<4xi32>) -> (vector<4xi32>, vector<4xi32>) {
  %cst = arith.constant dense<0> : vector<8xi32>
  %0 = vector.insert_strided_slice %arg0, %cst {offsets = [0], strides = [1]} : vector<4xi32> into vector<8xi32>
  %1 = vector.insert_strided_slice %arg1, %0 {offsets = [4], strides = [1]} : vector<4xi32> into vector<8xi32>
  ...
  %2 = ... %1 ...
  ...
  %3 = vector.extract_strided_slice %2 {offsets = [0], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  %4 = vector.extract_strided_slice %2 {offsets = [4], sizes = [4], strides = [1]} : vector<8xi32> to vector<4xi32>
  return %3, %4 : vector<4xi32>, vector<4xi32>
}
```

Vector Unrolling: In Function Bodies

1. Unroll vectors to native vector size using pre-defined patterns
2. Convert transpose ops into extract/insert pairs
3. Cast away leading size-one dimensions
4. Decompose `vector.insert_strided_slice` and `vector.extract_strided_slice`

Vector Unrolling: Compute Target Vector Size

```
// Again, utilizing mlir/lib/Dialect/Vector/Transforms/VectorUnroll.cpp
int getComputeVectorSize(int64_t size) {
  for (int i : {4, 3, 2}) {
    if (size % i == 0)
      return i;
  }
  return 1;
}
```

Vector Unrolling: Currently Supported Op Categories

Elementwise Ops

```
%a = arith.addf %b, %c : vector<4xf32>
```

Transpose Ops

```
%1 = vector.transpose %0, [1, 0] : vector<2x3xf32> to vector<3x2xf32>
```

Reduction Ops

```
%1 = vector.reduction <add>, %0 : vector<16xf32> into f32
```

Vector Unrolling: Get Native Vector Shape

- In `mlir/lib/Dialect/SPIRV/Transforms/SPIRVConversion.cpp`
- Custom implementations for different op categories
- Example:

```
mlir::spirv::getNativeVectorShape(Operation *op) {
  if (OpTrait::hasElementwiseMappableTraits(op) && op->getNumResults() == 1) {
    if (auto vecType = dyn_cast<VectorType>(op->getResultTypes()[0])) {
      SmallVector<int64_t> nativeSize(vecType.getRank(), 1);
      nativeSize.back() =
          mlir::spirv::getComputeVectorSize(vecType.getShape().back());
      return nativeSize;
    }
  }
```

- ❖ Easy to cover new op categories in the future - just implement this function

Vector Unrolling: Get Native Vector Shape

- In `mlir/lib/Dialect/SPIRV/Transforms/SPIRVConversion.cpp`
- Custom implementations for different op categories
- Example:

```
mlir::spirv::getNativeVectorShape(Operation *op) {
  if (OpTrait::hasElementwiseMappableTraits(op) && op->getNumResults() == 1) {
    if (auto vecType = dyn_cast<VectorType>(op->getResultTypes()[0])) {
      SmallVector<int64_t> nativeSize(vecType.getRank(), 1);
      nativeSize.back() =
          mlir::spirv::getComputeVectorSize(vecType.getShape().back());
      return nativeSize;
    }
  }
```

- ❖ Easy to cover new op categories in the future - just implement this function

Vector Unrolling: Get Native Vector Shape

- In `mlir/lib/Dialect/SPIRV/Transforms/SPIRVConversion.cpp`
- Custom implementations for different op categories
- Example:

```
mlir::spirv::getNativeVectorShape(Operation *op) {
  if (OpTrait::hasElementwiseMappableTraits(op) && op->getNumResults() == 1) {
    if (auto vecType = dyn_cast<VectorType>(op->getResultTypes()[0])) {
      SmallVector<int64_t> nativeSize(vecType.getRank(), 1);
      nativeSize.back() =
          mlir::spirv::getComputeVectorSize(vecType.getShape().back());
      return nativeSize;
    }
  }
```

- ❖ Easy to cover new op categories in the future - just implement this function

Vector Unrolling: Transpose ⇒ Extract/Insert

TransposeOpLowering

```
// Replace:
  %x = vector.transpose %y, [1, 0]

// with:
  %z = arith.constant dense<0.000000e+00>
  %0 = vector.extract %y[0, 0]
  %1 = vector.insert %0, %z [0, 0]
  ..
  %x = vector.insert .., .. [.., ..]
```

Transpose2DWithUnitDimToShapeCast

```
// Replace:
  vector.transpose %0, [1, 0] : vector<4x1xi32>> to
                                vector<1x4xi32>
// with:
  vector.shape_cast %0 : vector<4x1xi32> to vector<1x4xi32>
```

Vector Unrolling: Decompose inserts/extracts

- May have `vector.insert_strided_slice` inserting 1-D native vectors into n-D larger vectors from the previous patterns
- Need to break them down into extract/insert pairs to cancel with each other

```
// Decompose different rank insert_strided_slice and n-D extract_slided_slice.
void vector::populateVectorInsertExtractStridedSliceDecompositionPatterns(
    RewritePatternSet &patterns, PatternBenefit benefit) {
  patterns.add<DecomposeDifferentRankInsertStridedSlice,
               DecomposeNDExtractStridedSlice>(patterns.getContext(), benefit);
}
```

- ❖ May be obvious for the experienced
- ❖ Challenging for beginners to understand and choose the right patterns to use

Vector Unrolling Result: Elementwise Op Example

```
func.func @vaddi(%arg0 : vector<6xi32>, %arg1 : vector<6xi32>) -> (vector<6xi32>) {
  %0 = arith.addi %arg0, %arg1 : vector<6xi32>
  return %0 : vector<6xi32>
}

==>

func.func @vaddi(%arg0: vector<3xi32>, %arg1: vector<3xi32>, %arg2: vector<3xi32>, %arg3: vector<3xi32>) ->
(vector<3xi32>, vector<3xi32>) {
  %0 = arith.addi %arg0, %arg2 : vector<3xi32>
  %1 = arith.addi %arg1, %arg3 : vector<3xi32>
  return %0, %1 : vector<3xi32>, vector<3xi32>
}
```

Vector Unrolling Result: Reduction Op Example

```
func.func @reduction(%arg0 : vector<8xi32>) -> (i32) {
  %0 = vector.reduction <add>, %arg0 : vector<8xi32> into i32
  return %0 : i32
}

==>

func.func @reduction(%arg0: vector<4xi32>, %arg1: vector<4xi32>) -> (i32) {
  %0 = vector.reduction <add>, %arg0 : vector<4xi32> into i32
  %1 = vector.reduction <add>, %arg1 : vector<4xi32> into i32
  %2 = arith.addi %0, %1 : i32
  return %2 : i32
}
```

Vector Unrolling Result: Transpose Op Example

```
func.func @transpose(%arg0 : vector<2x3xi32>) -> (vector<3x2xi32>) {
  %0 = vector.transpose %arg0, [1, 0] : vector<2x3xi32> to vector<3x2xi32>
  return %0 : vector<3x2xi32>
}
==>
func.func @transpose(%arg0: vector<3xi32>, %arg1: vector<3xi32>) -> (vector<2xi32>, vector<2xi32>, vector<2xi32>) {
  %cst = arith.constant dense<0> : vector<2xi32>
  %0 = vector.extract %arg0[0] : i32 from vector<3xi32>
  %1 = vector.insert %0, %cst [0] : i32 into vector<2xi32>
  %2 = vector.extract %arg1[0] : i32 from vector<3xi32>
  %3 = vector.insert %2, %1 [1] : i32 into vector<2xi32>
  %4 = vector.extract %arg0[1] : i32 from vector<3xi32>
  %5 = vector.insert %4, %cst [0] : i32 into vector<2xi32>
  %6 = vector.extract %arg1[1] : i32 from vector<3xi32>
  %7 = vector.insert %6, %5 [1] : i32 into vector<2xi32>
  %8 = vector.extract %arg0[2] : i32 from vector<3xi32>
  %9 = vector.insert %8, %cst [0] : i32 into vector<2xi32>
  %10 = vector.extract %arg1[2] : i32 from vector<3xi32>
  %11 = vector.insert %10, %9 [1] : i32 into vector<2xi32>
  return %3, %7, %11 : vector<2xi32>, vector<2xi32>, vector<2xi32>
}
```

Vector Unrolling: Future Directions

- Set up `OneToNTypeConversion` and `DialectConversion` to replace the current signature conversion
  - Attempted, but too complicated for an initial version
- Handle `func.call` and function declarations
- Optimize how we split the original shape into target shapes
  - Splitting `vector<5xi32>` into a `vector<4xi32>` and `vector<1xi32>`
  - Instead of five `vector<1xi32>`
- Check for the `Vector16` capability that supports vectors of length 8/16

MemRef: Map To SPIR-V Storage Classes (Vulkan)

```
memref<4xi32>
==>
!spirv.ptr<!spirv.struct<(!spirv.array<4 x i32, stride=4> [0])>

// -----

memref<vector<4xi32>>
==>
!spirv.ptr<!spirv.struct<(!spirv.array<1 x vector<4xi32>, stride=16> [0])>
```

Convert To SPIR-V: Progress & Future Directions

- Covered most things for converting the argmax kernel to SPIR-V
- Convert a `gpu.module` and its ops into a `spirv.module`
- Next:
  - Plug into the `mlir-runner` for integration testing
  - Try it! Feedback is welcomed :)

Thank you

Questions?