Vector Reshapes

MLIR Open Design Meeting December 14th, 2023

Quinn Dawkins (qdawkins@amd.com) with help from others
Goals of this presentation

- Focusing on a few questions and defining the problems
  - How should we think about virtual vectors?
  - What do reshapes mean on vectors?
  - How is this related to SPIR-V?
- Primarily to understand the problem, not provide concrete proposals
- Expecting follow up RFCs and/or ODMs
  - Time permitting: Layout & SIMT
What is a virtual vector?

- Before discussing op semantics, how do we define the builtin `VectorType`
  - Specifically in the context of the vector dialect, other clients can do whatever they want
- **Docs** describe it as a “nested aggregate type of 1-D vector” when lowering to LLVM
  - Alternatively "stacks" of 1-D vectors
  - The inner-most size is the 1-D vector width
  - All vectors are expected to be "unrolled" to 1-D eventually

![Diagram of vector<3x2x4>](https://mlir.llvm.org/docs/Dialects/Builtin/# vectortype)

![Diagram of vector<3x2x4>](https://mlir.llvm.org/docs/Dialects/Vector/#conclusion)
What is a virtual vector?

- For accelerators with native k-D vectors, the natural extension is a "stack" of k-D vectors
- For example, a stack of 2-D vectors of the same vector shape
What is a virtual vector?

- Based on the current documentation, the most general interpretation does not imply any sort of native dimensionality.
- Interpret it as the data residing in a virtual SIMD register of infinite width and dimensionality.
  - Vector type maps elements to locations in a virtual element space (i.e. virtual register).
- Any operation that changes the linearized bitvector, or locations of elements in the vector has cost (until a lowering asserts otherwise).

```
vector<3x5x32>:
```

<table>
<thead>
<tr>
<th>Virtual Register</th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
<th>0,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td></td>
</tr>
</tbody>
</table>
```
Shape Cast

`vector.shape_cast %0 : vector<3x2x4> to vector<2x3x4>`

With 1-D stacked vectors, this has no cost
Shape Cast

`vector.shape_cast %0 : vector<3x2x4> to vector<2x3x4>`
Shape Cast

`vector.shape_cast %0 : vector<3x2x4> to vector<2x3x4>`

With 2-D stacked vectors, is this still a no-op?

```
(vector<3x2x4>)

0.0 0.1 0.2 0.3
1.0 1.1 1.2 1.3
2.0 2.1 2.2 2.3

(vector<6x4>)

1.0 1.1 1.2 1.3
2.0 2.1 2.2 2.3
3.0 3.1 3.2 3.3
4.0 4.1 4.2 4.3
5.0 5.1 5.2 5.3

(vector<2x3x4>)

1.0 1.1 1.2 1.3
2.0 2.1 2.2 2.3
3.0 3.1 3.2 3.3
4.0 4.1 4.2 4.3
5.0 5.1 5.2 5.3
6.0 6.1 6.2 6.3
```
Shape Cast

%1 = vector.shape_cast %0 : vector<3x2x4> to vector<2x3x4>

%1 = vector.reduction <add> %1 : vector<2x3x4> to vector<3x4>
**Shape Cast**

`vector.shape_cast %0 : vector<1x4> to vector<4x1>`

Not a no-op, even with 1-D vectors
Other Reshaping Operations

- General shape_cast is difficult to handle, especially for >1-D vector sizes
  - Many of the hardest to handle cases are likely uncommon
    - 4x1x5 -> 10x2
- Idea: "Peel off" some of the more common and easier to handle cases as different ops
  - A good litmus test for a reshape op is whether various lowering patterns would expect to handle different kinds of reshapes significantly differently

```plaintext
vector.expand_shape %0 : vector<16> to vector<4x4>

vector.collapse_shape %0 : vector<4x4> to vector<16>
```
Other Reshaping Operations

- As identified in this [discourse thread](https://discourse.llvm.org/t/improving-handling-of-unit-dimensions-in-the-vector-dialect/75216), unit dimensions in particular are a special case where a number of vector ops converge
  - `vector.squeeze` and `vector.unsqueeze` for reshapes only on unit dimensions

```c
vector.broadcast %0 : vector<16> to vector<1x1x16>
```

```c
==

vector.unsqueeze %0 : vector<16> to vector<1x1x16>
```
Other Reshaping Operations

- As identified in this [discourse thread](https://discourse.llvm.org/t/improving-handling-of-unit-dimensions-in-the-vector-dialect/75216), unit dimensions in particular are a special case where a number of vector ops converge
  - `vector.squeeze` and `vectorunsqueeze` for reshapes only on unit dimensions

```plaintext
vector.extract @0, 0 : vector<1x1x16> to vector<16>
```

```
=*

vector.squeeze @0 : vector<1x1x16> to vector<16>
```
Other Reshaping Operations

// No-ops for any vector dimensionality

vector.squeeze %0 : \textit{vector}<1x1x16> to \textit{vector}<16>

vector.unsqueeze %0 : \textit{vector}<16> to \textit{vector}<1x1x16>
Other Reshaping Operations

- Potential cons
  - All of the reshape ops include nothing beyond type information
  - We could achieve something similar with nice matching helpers on `vector.shape_cast`
    - `isCollapseShape`
    - `isExpandShape`
    - `isSqueeze`
    - `isUnsqueeze`
  - Many passes + patterns will need updating to handle each of the different reshape types
    - Duplicate code unless the semantics of `shape_cast` are restricted
Virtual Vector -> SPIR-V
Virtual Vector -> SPIR-V

- Like LLVM, SPIR-V only supports 1-D vectors
- Additional constraint that all vectors must be ≤ 4 elements wide
- Currently managed with upstream unrolling patterns + canonicalization
  - SPIRVInitialVectorLowering
- Other one-off patterns for breaking down large vectors in other cases
  - Breaking down loop carried variables
  - Breaking down other large vectors as a fallback
- Converts to memref<vector> + bitcasts to get 128 bit read/store width on < 32 bit data types
  - SPIRVVectorizeLoadStore
- Today, SPIR-V can be sensitive to small changes to canonicalizations/other vector patterns as a result
  - Goal: decouple canonicalizations and vector unrolling from target requirements
Virtual Vector -> SPIR-V

- Like LLVM, SPIR-V only supports 1-D vectors
- Additional constraint that all vectors must be <= 4 elements wide
- Today, SPIR-V can be sensitive to small changes to canonicalizations/other vector patterns as a result

```plaintext
%2 = arith.addi %0, %1 : vector<7>  %2#0 = arith.addi %0#0, %1#0 : vector<4>
vector.reduction <add> %2 : vector<7>  %2#1 = arith.addi %0#1, %1#1 : vector<3>
                                          %3#0 = vector.reduction <add> %2#0 : vector<4>
                                          %3#1 = vector.reduction <add> %2#1 : vector<3>
                                          arith.addi %3#0, %3#1 : i32
```
Consider the same vector $\langle 3 \times 5 \times 32 \rangle$ mapping to 1-D hardware registers of maximum width 4

- 6x 4-element vectors, with 3 elements masked off of 3 of the vectors
Virtual Vector -> Native Virtual Vector

- Consider the same vector $3 \times 5 \times 132$ mapping to 2-D vectors of shape $3 \times 2$
Virtual Vector -> Native Virtual Vector

- Consider vector<5x132> mapping to 2-D hardware registers of shape 3x2
  - Masking/padding to get efficient register usage
- Could allow other partitioning schemes, this is just the most natural
Other Reshaping Operations

// Always a transpose or "unshuffle to scalars"

vector.unsqueeze %0 : vector<6> to vector<6x1>

==

vector.unsqueeze %0#0 : vector<4> to vector<4x1>

vector.unsqueeze %0#1 : vector<2> to vector<2x1>

vector<4> + vector<2>

| 0 | 1 | 2 | 3 | 4 | 5 |
vector<4x1> + vector<2x1>

| 0 | 1 | 2 | 3 |
| 4 | 5 |
Bonus: SIMD Vector -> SIMT Vector
Vector vs Tensor

- **(Ranked)TensorType**
  - Value semantic "container" for some underlying data
  - Encoding attributes can be used to specify structure, otherwise there is no specific structure
  - The shape of a tensor could be metadata

- **Example usage of the tensor type**
  - **IREE** in most cases treats them simply as densely packed, contiguous bags of bytes
  - **Triton** uses tensors like a SIMD vector with a layout attribute describing how the underlying elements are held across threads, before explicitly converting to SIMT
  - In-tree efforts in the **sparse_tensor** dialect use encodings to describe various structures for how the data is stored
Even without a layout, with the above view of vectors we can still distribute

- Implied in the rationale behind the vector dialect
- Grid of [2] threads with an inner SIMD width of 4
Separating Distribution From Layout

- The above leaves out the analysis needed to distribute tensors/vectors
  - Separate dedicated discussion needed for layout
Summary

- Original discourse thread
- Defining a virtual vector in a lowering agnostic way
- Peeling off functionality from `vector.shape_cast` to cover a few common cases
  - `squeeze/unsqueeze` for the special properties of unit dimensions
- LLVM centric vector lowerings require extra work on the SPIR-V side
  - Seeing a similar trend with the scalable vector bring-up