



Storing Arrays in MLIR

A Field Guide to Elements Attributes and Custom Element Types

Matthias Springer (NVIDIA)

LLVM Developers' Meeting 2026 – Quick Talk – October 28, 2026

ArrayAttr: An Array of Attributes

Example: `[3 : i64, 3 : i64, 5.4 : f32]`

Example: `[#ub.poison, affine_map<...>, 8 : i32, 9 : i64]`

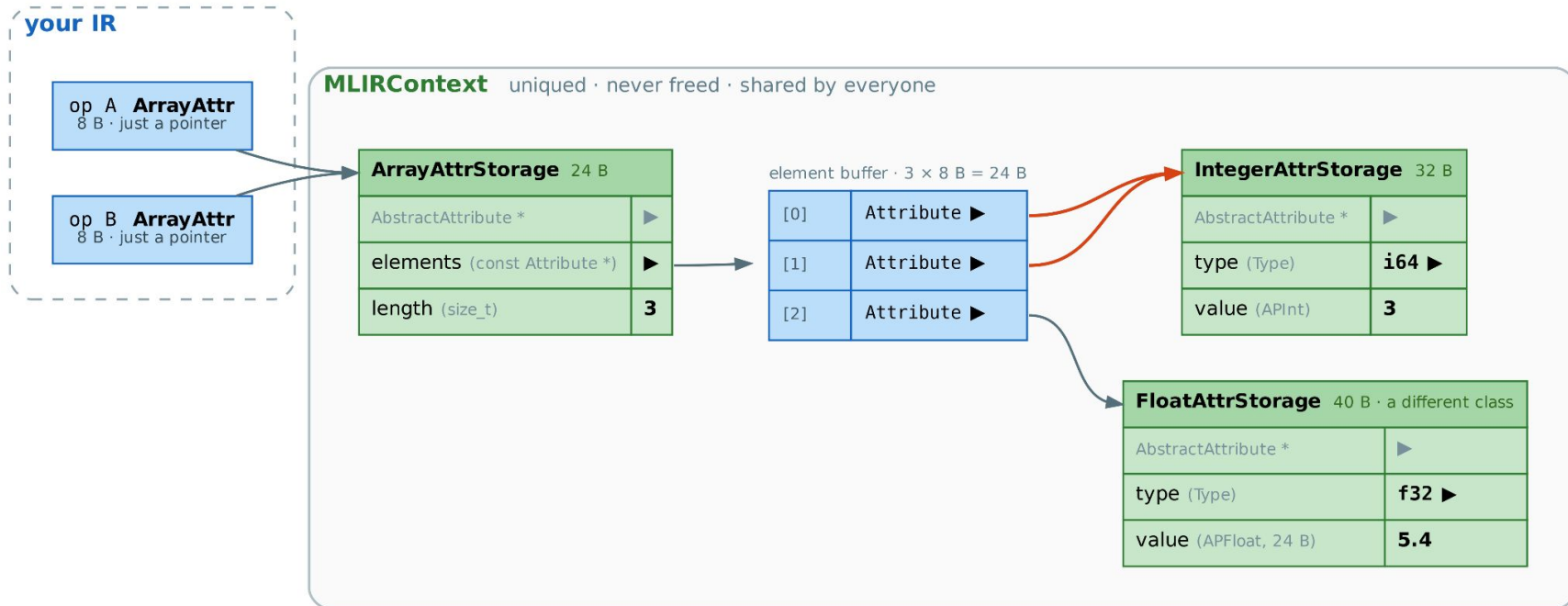
- Array of attributes.
- Array elements can be different kinds of attributes.
- Array elements are stored separately in the context.

C++ API:

- `int64_t numElements = arrayAttr.size();`
- `Attribute x = arrayAttr[pos];`
- `ArrayRef<Attribute> attrs = arrayAttr.getValue();`

ArrayAttr: An Array of Attributes

Example: [3 : i64, 3 : i64, 5.4 : f32]



DenseArrayAttr: A Raw Buffer with Convenience API

Example: `array<i64: 0, 1, 0>`

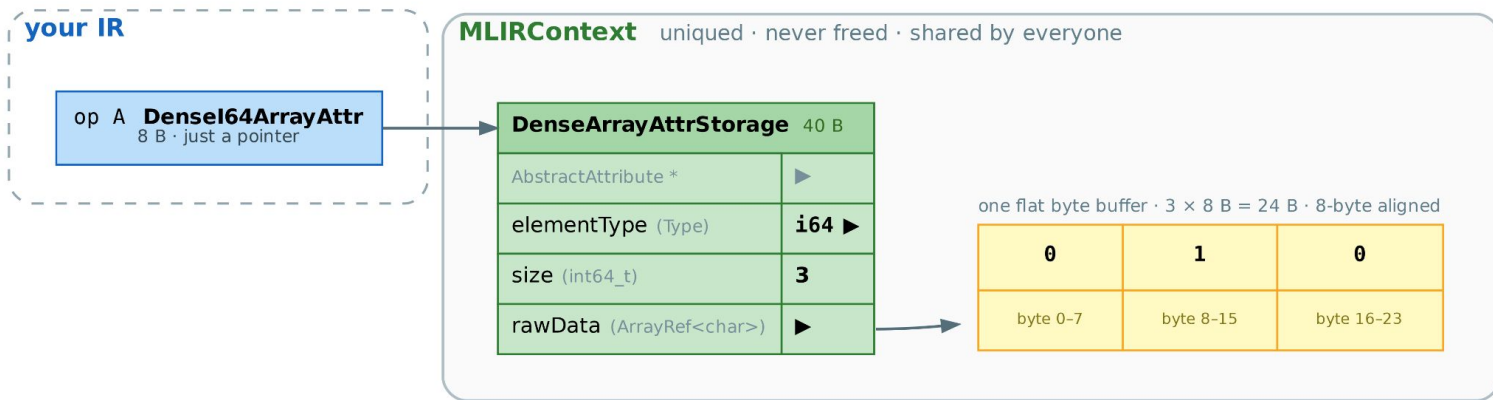
- Concrete implementations: `DenseI64ArrayAttr` etc.
- All array elements have the same type.
- Array elements are stored in a single buffer (“unboxed”).

C++ API:

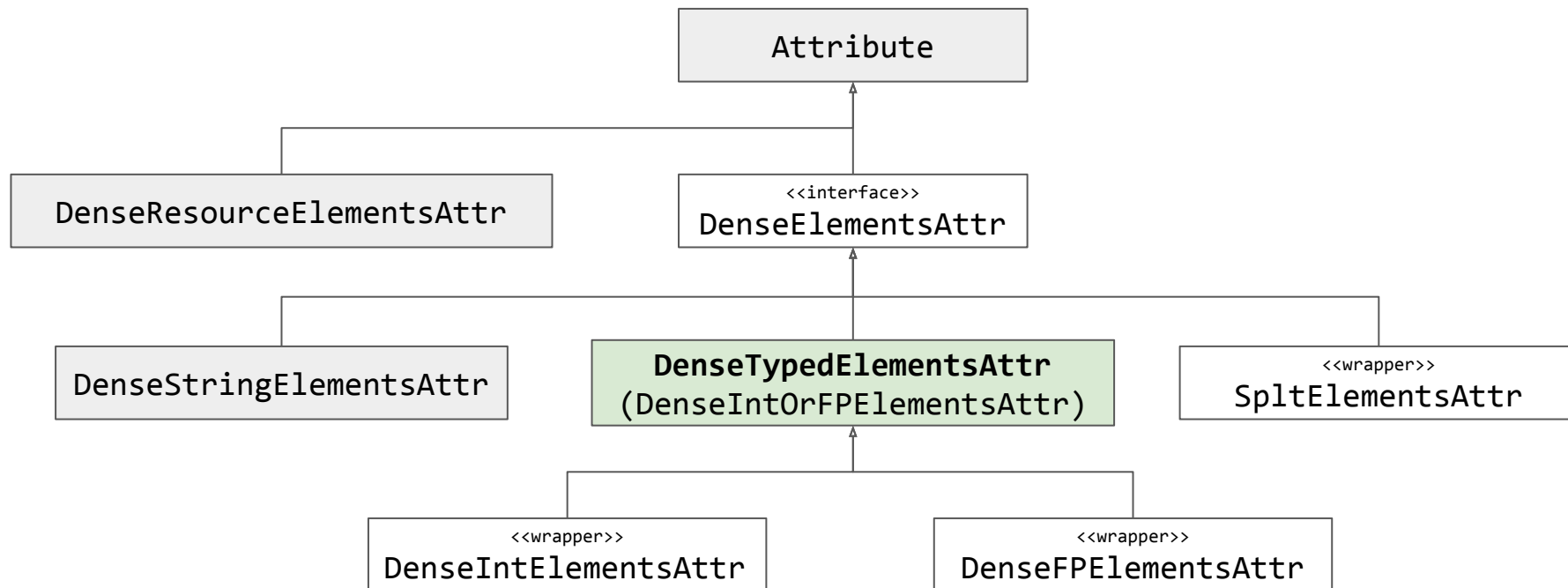
- `int64_t numElements = attr.size();`
- `int64_t x = attr[0];`
- `ArrayRef<int64_t> elements = attr.asArrayRef<int64_t>();`

DenseArrayAttr: A Raw Buffer with Convenience API

Example: `array<i64: 0, 1, 0>`



DenseElementsAttr



DenseTypedElementsAttr: Buffer with Attr. Serialization

Example: `dense<[[1,0,0,0], [0,0,5,0], [0,0,0,0]]> : tensor<3x4xi32>`

Example: `dense<tensor<3x4xi32> : [`
 `[1 : i32, 0 : i32, 0 : i32, 0 : i32],`
 `[0 : i32, 0 : i32, 5 : i32, 0 : i32],`
 `[0 : i32, 0 : i32, 0 : i32, 0 : i32]]>`

C++ API:

- `ArrayRef<char> data = attr.getRawData();`
- `SmallVector<Attribute> elements =`
 `to_vector(attr.getValues<Attribute>());`
- Convenience overloads for common builtin types exist.

DenseTypedElementsAttr literal-first syntax only for selected builtin types

Example: `dense<[[1,0,0,0], [0,0,5,0], [0,0,0,0]]> : tensor<3x4xi32>`

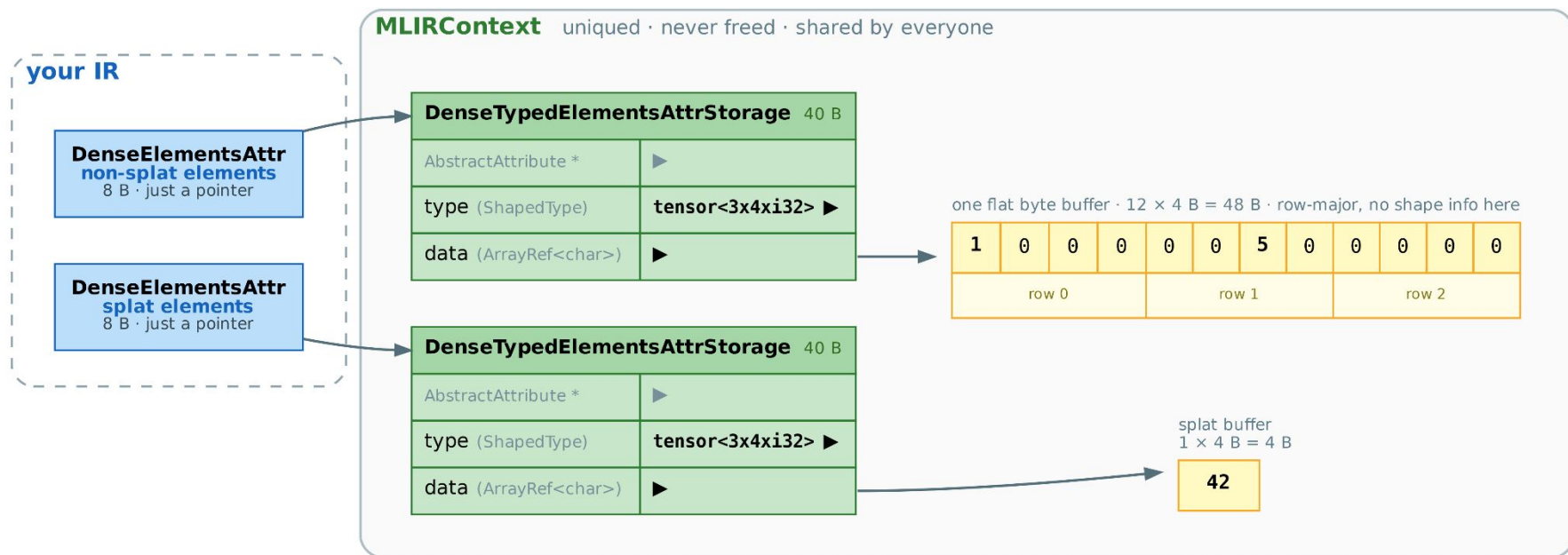
Example: `dense<tensor<3x4xi32> : [
 [1 : i32, 0 : i32, 0 : i32, 0 : i32],
 [0 : i32, 0 : i32, 5 : i32, 0 : i32],
 [0 : i32, 0 : i32, 0 : i32, 0 : i32]]>`

type-first syntax
works for any dense
element type

DenseElementTypeInterface establishes a mapping from element type → attribute (mainly for parsing/printing textual IR). For efficiency reasons, C++ code should operate on raw char buffers.

DenseTypedElementsAttr: Buffer with Attr. Serialization

Example: `dense<tensor<3x4xi32> : [[1 : i32, 0 : i32, ...], ...]>`



DenseElementsTypeInterface: Attribute Factory

- Element type of DenseElementsAttr must implement DenseElementInterface.
- The interface specifies how to convert between raw data and an attribute.

```
Attribute IntegerType::convertToAttribute(ArrayRef<char> rawData) const {  
    APInt value = parseInteger(rawData);  
    return IntegerAttr::get(*this, value);  
}
```

```
LogicalResult IntegerType::convertFromAttribute(Attribute attr, SmallVectorImpl<char> &result) const {  
    writeAPIntToVector(cast<IntegerAttr>(attr).getValue(), result);  
    return success();  
}
```

DenseElementsTypeInterface: Attribute Factory

Element type $\leftrightarrow$ Attribute mapping is 1:N. The same attribute kind can be used for multiple element types.

Example: `dense<tensor<3x4x!my_dialect.my_int<32 div_by 128>> : [`
 `[128 : i32, 0 : i32, 0 : i32, 0 : i32],`
 `[0 : i32, 0 : i32, 256 : i32, 0 : i32],`
 `[1024 : i32, 1024 : i32, 0 : i32, 0 : i32]]>`

Properties

Stored directly in the operation. No round trip through the MLIR context.

one malloc'd block — the Operation

one malloc, addresses increasing downwards

results
Operation name, loc, attr dict, counts
operand storage
Properties inline here, ≤ 2048 B
successors
regions
operands

inside that Properties slab — two different things can live here

an ODS attribute, moved into properties	DenseI64ArrayAttr 8 B · still just a pointer ▶
a native property, a plain C++ value	SmallVector<int64_t> the values themselves · no context involved

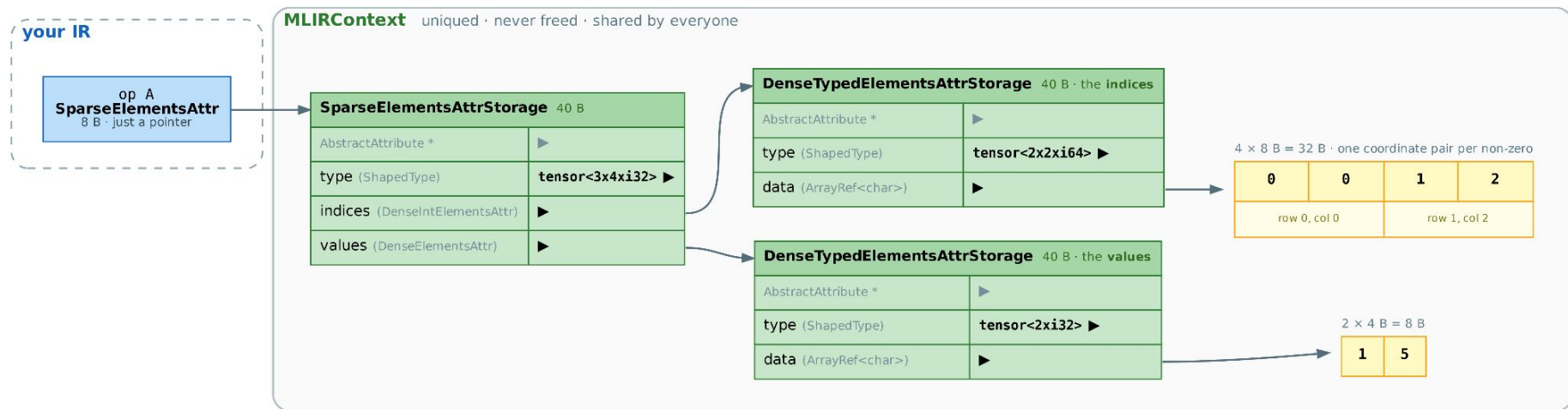
MLIRContext

DenseArrayAttrStorage 40 B + bytes	
still hashed on every get(), still immortal	...

SparseElementsAttr: Indices and Values

Example: `sparse<[[0,0], [1,2]], [1,5]>`

All values are zero, except for `attr[0,0] = 1` and `attr[1,2] = 5`.



Summary

- Prefer properties for best performance. No round trip through `MLIRContext`.
Exception: Large arrays that are used in multiple places.
- Use `DenseArrayAttr` for 1D arrays with supported element type.
Supported element types: `i1` and integer/float types with a bit width that is divisible by 8.
- Use `DenseTypedElementsAttr` for multi-dim. arrays with element type that implements `DenseElementTypeInterface`. Splat optimization is available.
- Use `ArrayAttr` when elements have different types.
- Use `SparseElementsAttr` for large sparse arrays. The element type must implement `DenseElementTypeInterface`.