1:N Dialect Conversion

Matthias Springer (NVIDIA)

EuroLLVM 2025 – Quick Talk – April 15, 2025

Timeline

Listener / Rewriter API Improvements

Feb 2023 March 2024

Internal Refactorings, Bug Fixes, Cleanups (`IRRewrite`, etc.)

Feb 2024 Aug 2024

1:N Support

Sept 2024 Jan 2025

One-Shot Dialect Conversion

Apr 2025 ...

MemRef 1:N

Dialect Conversion

- One of the two main pattern drivers in MLIR, more powerful API than "regular" rewrite API (`PatternRewriter`)

`PatternRewriter`	`ConversionPatternRewriter`
`replaceOp(Operation *, ValueRange);`	`replaceOp(Operation *, ValueRange);`
a combination of: `createBlock`, `inlineBlockBefore`, create unrealized_conversion_cast ops, `replaceAllUsesWith`, `eraseBlock`	`applySignatureConversion(Block *, SignatureConversion &, TypeConverter &);`

Dialect Conversion

- One of the two main pattern drivers in MLIR, more powerful API than “regular” rewrite API (`PatternRewriter`)

`PatternRewriter`	`ConversionPatternRewriter`
`replaceOp(Operation *, ValueRange);`	`replaceOp(Operation *, ValueRange);`
a combination of: `createBlock`, `inlineBlockBefore`, create unrealized_conversion_cast ops, `replaceAllUsesWith`, `eraseBlock`	`applySignatureConversion(Block *, SignatureConversion &, TypeConverter &);`

For each block argument, specifies the new type in the converted block.

Dialect Conversion

- One of the two main pattern drivers in MLIR, more powerful API than “regular” rewrite API (PatternRewriter)
- What’s new: 1:N op replacements, 1:N conversion patterns

PatternRewriter	ConversionPatternRewriter
replaceOp(Operation *, ValueRange);	replaceOp(Operation *, ValueRange); replaceOpWithMultiple(Operation *, ArrayRef<ValueRange>);
a combination of: createBlock, inlineBlockBefore, create unrealized_conversion_cast ops, replaceAllUsesWith, eraseBlock	applySignatureConversion(Block *, SignatureConversion &, TypeConverter &);

For each block argument, specifies the new types in the converted block.

Outline of This Work

- **Cleanup** of duplicate + non-composable frameworks:
  - Incomplete implementation: 1:N used to be partially supported in the dialect conversion framework (only signature conversions).
  - There is a separate 1:N dialect conversion framework that supports 1:N op replacements, but it’s not really a dialect conversion.
- Why is this important?
  - 1:N conversions appear in **load-bearing + performance critical** lowering passes of real-world compilers.
    - MLIR: MemRef → LLVM Lowering
    - MLIR: Sparse Tensor → Sparse Tensor Descriptor (codegen path)
    - Triton: Tile → SIMT
    - Multiple NVIDIA-internal projects: cuTile, Cutlass, …
  - These passes must resort to packing/unpacking to work around dialect conversion limitations. That’s **inefficient** (increases compilation time) and makes code/IR **complex**.

Example: MemRef → LLVM

Example: 1:1 MemRef → LLVM Lowering

```
memref<?x?xf32, strided<[?, ?], offset: ?>>
```

1 SSA value → 1 SSA value

```
!llvm.struct<(!llvm.ptr, !llvm.ptr,       // ptr
              i64,                        // offset
              !llvm.struct<(i64, i64)>,   // sizes
              !llvm.struct<(i64, i64)>)   // strides
```

Example: 1:N MemRef → LLVM Lowering

```
memref<?x?xf32, strided<[?, ?], offset: ?>>
```

1 SSA value → 7 SSA values

```
!llvm.ptr, !llvm.ptr,                       // ptr
i64,                                        // offset
i64, i64                                    // sizes
i64, i64                                    // strides
```

Example: 1:1 MemRef → LLVM Lowering

```
memref<*xf32>
```

1 SSA value → 1 SSA value

```
!llvm.struct<(!llvm.ptr,    // ptr to descriptor
              i64)>         // rank
```

Example: 1:N MemRef → LLVM Lowering

```
memref<*xf32>
```

1 SSA value → 2 SSA values

```
llvm.ptr,                    // ptr to descriptor
i64                          // rank
```

Test Case

```
// RUN: mlir-opt %s -expand-strided-metadata -convert-to-llvm

func.func @test_case(%m: memref<5xf32>, %offset: index) -> f32 {
    %c1 = arith.constant 1 : index
    %0 = memref.subview %m[%offset][2][1] : memref<5xf32> to memref<2xf32, strided<[1], offset: ?>>
    %1 = memref.load %0[%c1] : memref<2xf32, strided<[1], offset: ?>>
    return %1 : f32
}
```

Test Case

```
// RUN: mlir-opt %s –convert-to-llvm

func.func @test_case(%m: memref<5xf32>, %offset: index) -> f32 {
  %c1 = arith.constant 1 : index
  %base_buffer, %offset, %sizes, %strides = memref.extract_strided_metadata %m
    : memref<5xf32> -> memref<f32>, index, index, index
  %reinterpret_cast = memref.reinterpret_cast %base_buffer to offset: [%offset], sizes: [2], strides: [1]
    : memref<f32> to memref<2xf32, strided<[1], offset: ?>>
  %0 = memref.load %reinterpret_cast[%c1] : memref<2xf32, strided<[1], offset: ?>>
  return %0 : f32
}
```

34 operations

Test Case: 1:1 Lowering (current lowering)

```
llvm.func @test_case(%arg0: !llvm.ptr, %arg1: !llvm.ptr,
                     %arg2: i64, %arg3: i64, %arg4: i64,
                     %arg5: i64) -> f32 {
  %0 = llvm.mlir.poison
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %1 = llvm.insertvalue %arg0, %0[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %2 = llvm.insertvalue %arg1, %1[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %3 = llvm.insertvalue %arg2, %2[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %4 = llvm.insertvalue %arg3, %3[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %5 = llvm.insertvalue %arg4, %4[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %6 = llvm.mlir.constant(1 : index) : i64
  %7 = llvm.extractvalue %5[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %8 = llvm.extractvalue %5[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %9 = llvm.mlir.poison : !llvm.struct<(ptr, ptr, i64)>
  %10 = llvm.insertvalue %7, %9[0] : !llvm.struct<(ptr, ptr, i64)>
  %11 = llvm.insertvalue %8, %10[1] : !llvm.struct<(ptr, ptr, i64)>
  %12 = llvm.mlir.constant(0 : index) : i64
  %13 = llvm.insertvalue %12, %11[2] : !llvm.struct<(ptr, ptr, i64)>
  %14 = llvm.extractvalue %5[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %15 = llvm.extractvalue %5[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %16 = llvm.extractvalue %5[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %17 = llvm.mlir.poison
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %18 = llvm.extractvalue %13[0] : !llvm.struct<(ptr, ptr, i64)>
  %19 = llvm.extractvalue %13[1] : !llvm.struct<(ptr, ptr, i64)>
  %20 = llvm.insertvalue %18, %17[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %21 = llvm.insertvalue %19, %20[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %22 = llvm.insertvalue %arg5, %21[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %23 = llvm.mlir.constant(2 : index) : i64
  %24 = llvm.insertvalue %23, %22[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %25 = llvm.mlir.constant(1 : index) : i64
  %26 = llvm.insertvalue %25, %24[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %27 = llvm.extractvalue %26[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %28 = llvm.extractvalue %26[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %29 = llvm.getelementptr %27[%28] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %30 = llvm.getelementptr %29[%6] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %31 = llvm.load %30 : !llvm.ptr -> f32
  llvm.return %31 : f32
}
```

Test Case: 1:1 Lowering (current lowering)

```
llvm.func @test_case(%arg0: !llvm.ptr, %arg1: !llvm.ptr,
                     %arg2: i64, %arg3: i64, %arg4: i64,
                     %arg5: i64) -> f32 {
```

target materialization: pack

```
%0 = llvm.mlir.poison
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%1 = llvm.insertvalue %arg0, %0[0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%2 = llvm.insertvalue %arg1, %1[1]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%3 = llvm.insertvalue %arg2, %2[2]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%4 = llvm.insertvalue %arg3, %3[3, 0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%5 = llvm.insertvalue %arg4, %4[4, 0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%6 = llvm.mlir.constant(1 : index) : i64
```

ExtractStridedMetadataOpLowering: unpack

```
%7 = llvm.extractvalue %5[0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%8 = llvm.extractvalue %5[1]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
```

ExtractStridedMetadataOpLowering: pack

```
%9 = llvm.mlir.poison : !llvm.struct<(ptr, ptr, i64)>
%10 = llvm.insertvalue %7, %9[0] : !llvm.struct<(ptr, ptr, i64)>
%11 = llvm.insertvalue %8, %10[1] : !llvm.struct<(ptr, ptr, i64)>
%12 = llvm.mlir.constant(0 : index) : i64
%13 = llvm.insertvalue %12, %11[2] : !llvm.struct<(ptr, ptr, i64)>
```

ExtractStridedMetadataOpLowering: unpack

```
%14 = llvm.extractvalue %5[2]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%15 = llvm.extractvalue %5[3, 0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%16 = llvm.extractvalue %5[4, 0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
```

MemRefReinterpretCastOpLowering: unpack

```
%17 = llvm.mlir.poison
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%18 = llvm.extractvalue %13[0] : !llvm.struct<(ptr, ptr, i64)>
%19 = llvm.extractvalue %13[1] : !llvm.struct<(ptr, ptr, i64)>
```

MemRefReinterpretCastOpLowering : pack

```
%20 = llvm.insertvalue %18, %17[0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%21 = llvm.insertvalue %19, %20[1]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%22 = llvm.insertvalue %arg5, %21[2]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%23 = llvm.mlir.constant(2 : index) : i64
%24 = llvm.insertvalue %23, %22[3, 0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%25 = llvm.mlir.constant(1 : index) : i64
%26 = llvm.insertvalue %25, %24[4, 0]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
```

LoadOpLowering :unpack

```
%27 = llvm.extractvalue %26[1]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%28 = llvm.extractvalue %26[2]
  : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
%29 = llvm.getelementptr %27[%28] : (!llvm.ptr, i64) -> !llvm.ptr, f32
%30 = llvm.getelementptr %29[%6] : (!llvm.ptr, i64) -> !llvm.ptr, f32
%31 = llvm.load %30 : !llvm.ptr -> f32
llvm.return %31 : f32
}
```

13 operations: 61% reduction in number of operations

Test Case: 1:N Lowering (better lowering)

```
llvm.func @bar(%arg0: !llvm.ptr, %arg1: !llvm.ptr, %arg2: i64, %arg3: i64, %arg4: i64, %arg5: i64) -> f32 {
  %0 = llvm.mlir.constant(1 : index) : i64
  %1 = llvm.mlir.poison : !llvm.ptr
  %2 = llvm.mlir.poison : i64
  %3 = llvm.mlir.constant(0 : index) : i64
  %4 = llvm.mlir.poison : !llvm.ptr
  %5 = llvm.mlir.poison : i64
  %6 = llvm.mlir.constant(2 : index) : i64
  %7 = llvm.mlir.constant(1 : index) : i64
  %8 = llvm.getelementptr %arg1[%arg5] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %9 = llvm.getelementptr %8[%0] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %10 = llvm.load %9 : !llvm.ptr -> f32
  llvm.return %10 : f32
}
```

Prototype: llvm/llvm-project:users/matthias-springer/memref_1_to_n

1:N Conversion API

Type Converter

These are the types of the `ValueRange` that the adaptor returns.

```
converter.addConversion([&](MemRefType type,
                            SmallVectorImpl<Type> &result) -> std::optional<LogicalResult> {
  result.push_back(llvmPtrTy);            // allocated ptr
  result.push_back(llvmPtrTy);            // aligned ptr
  result.push_back(i64Ty);                // offset
  for (int64_t i = 0; i < rank; ++i)
    result.push_back(i64Ty);              // sizes
  for (int64_t i = 0; i < rank; ++i)
    result.push_back(i64Ty);              // strides
  return success();
});

// previously: returned !llvm.struct<(…)>
```

Conversion Pattern

```
// Simplified: Assume that source is a ranked memref and index is static.
class DimOpLowering : public OpConversionPattern<memref::DimOp> {
  LogicalResult matchAndRewrite(memref::DimOp op, OneToNOpAdaptor adaptor,
                                ConversionPatternRewriter &rewriter) const override {
    int64_d dim = op.getIndex();
    int64_t descriptorPos = 2 + 1 + dim;
    ValueRange descriptor = adaptor.getSource();
    rewriter.replaceOp(op, descriptor[descriptorPos]);
    return success();
  }
};

// previously: OpAdaptor
```

Conversion Pattern (incorrect example)

```
// Simplified: Assume that source is a ranked memref and index is static.
class DimOpLowering : public OpConversionPattern<memref::DimOp> {
  LogicalResult matchAndRewrite(memref::DimOp op, OpAdaptor adaptor,
                                ConversionPatternRewrit    iter) const override {
    int64_d dim = op.getIndex();
    int64_t descriptorPos = 2 + 1 + dim;
    Value descriptor = adaptor.getSource();
    rewriter.replaceOp(op, ???);
    return success();
  }
};

// previously: OpAdaptor
```

What if I use a regular adaptor in a 1:N conversion?

`LLVM fatal error: ‘DimOpLowering’ does not support 1:N conversion`

Note: You can use a regular adaptor if all converted values are guaranteed to be single SSA values.

Migration Path from Deprecated 1:N Dialect Conversion

`OneToNTypeConversion.h`

Migration Guide

- Will be deleted from upstream MLIR after this conference!
- Migrate patterns, driver invocation and tests
  - `OneToNConversionPattern` → `ConversionPattern`
  - `applyPartialOneToNConversion` → `applyPartialConversion`. You're going to have to define a `ConversionTarget`.
  - `replaceOp(Operation *, ValueRange, OneToNTypeMapping)` → `replaceOpWithMultiple(Operation*, ArrayRef<ValueRange>)`
  - `OneToNTypeMapping` → `SignatureConversion`
  - Update test cases: Old framework was actually a greedy pattern rewrite that performs additional CSE'ing, folding, region simplification.

Questions?

`applyPartialConversion`

`applyPartialOneToNConversion`

`applySignatureConversion`

`ConversionPattern`

`ConversionPatternRewriter`

`ConversionTarget`

`LLVMStructType`

`MemRefType`

`OpAdaptor`

`OneToNOpAdaptor`

`OneToNTypeMapping`

`replaceOp`

`replaceOpWithMultiple`

`SignatureConversion`

source materialization

`TypeConverter`

target materialization

`UnrankedMemRefType`

`unrealized_conversion_cast`