MLIR-PyTACO: An End-to-End Use Case For the Sparse Tensor Compiler

Bixia Zheng (bixia@google.com)

Aart Bik (ajcbik@google.com)

For: MLIR ODM, 2/10/2022

Outline

- What is MLIR-PyTACO
- Sparse Tensor Algebra Support in MLIR
- Implementing PyTACO with MLIR
- Source Code Organization
- Ongoing/Future Work

MLIR-PyTACO

- TACO: The Tensor Algebra Compiler
- PyTACO: Python-based DSL for using TACO
- MLIR-PyTACO
  - Use MLIR to implement the DSL
    - Index notation based tensor algebra syntax
    - Tensor I/O

Index Notation Based Tensor Algebra

- Tensor: ***A = tensor([2,3], [compressed, dense])***
- Index_var: ***i, j = get_index_var(2)***
  - Represents dimensions in the iteration space
- Tensor access expression: A[i, j]
- Tensor assignment: ***C[i, j] = A[i, j] + B[i, j]***
  - The same index_var corresponds to dimensions of the same size
  - Element-wise operations: +, -, *, /
  - Implicit reduction operations: +

MLIR

Tensor I/O

- Coordinate format
  - (coordinates, non-zero-value)
- Support variants of coordinate formats
  - Matrix Market format (.mtx)
  - FROSTT format (.tns)
- Tensor input: ***A = read("input.mtx", [compressed, dense])***
- Tensor output: ***write("output.tns", C)***
- Create dense tensor from numpy array: from_array

MLIR

SpMV via MLIR-PyTACO

```
import mlir_pytaco_api as pt
import numpy as np

csr = pt.format([pt.dense, pt.compressed])
dv = pt.format([pt.dense])

A = pt.read("pwtk.mtx", csr)
X = pt.from_array(np.random.uniform(size=A.shape[1]))
Z = pt.from_array(np.random.uniform(size=A.shape[0]))
Y = pt.tensor([A.shape[0]], dv)

i, j = pt.get_index_vars(2)

Y[i] = A[i, j] * X[j] + Z[i]
pt.write("Y.tns", Y)
```

Outline

- What Is MLIR-PyTACO
- Sparse Tensor Algebra Support in MLIR

Tensor Types And Sparsity

```
tensor<3x4xf64>
```

```
#CSR   = #sparse_tensor.encoding<{ dimLevelType = [ "dense", "compressed" ] }>
tensor<3x4xf64, #CSR>
```

```
#DCSR  = #sparse_tensor.encoding<{ dimLevelType = [ "compressed", "compressed" ] }>
tensor<3x4xf64, #DCSR>
```

MLIR

Tensor Algebra via linalg.generic

C[i] = A[i, j] * B[j, i]

```
#trait = {
  indexing_maps = [
    affine_map<(i,j) -> (i,j)>,  // A
    affine_map<(i,j) -> (j,i)>,  // B
    affine_map<(i,j) -> (i)>   // C
  ],
  iterator_types = ["parallel", "reduction"], // Can we derive this from the result tensor affine_map?
}
%0 = linalg.generic #trait
  ins(%argA, %argB : !typeA, !typeB)
  outs(%argC : !typeC) {
    ^bb0(%A: f64, %B: f64, %C: f64):
      %1 = arith.mulf %A, %B : f64
      %2 = arith.addf %C, %1 : f64
      linalg.yield %2 : f64
    } -> !typeC
```

MLIR

Linalg Tensor Algebra Semantics

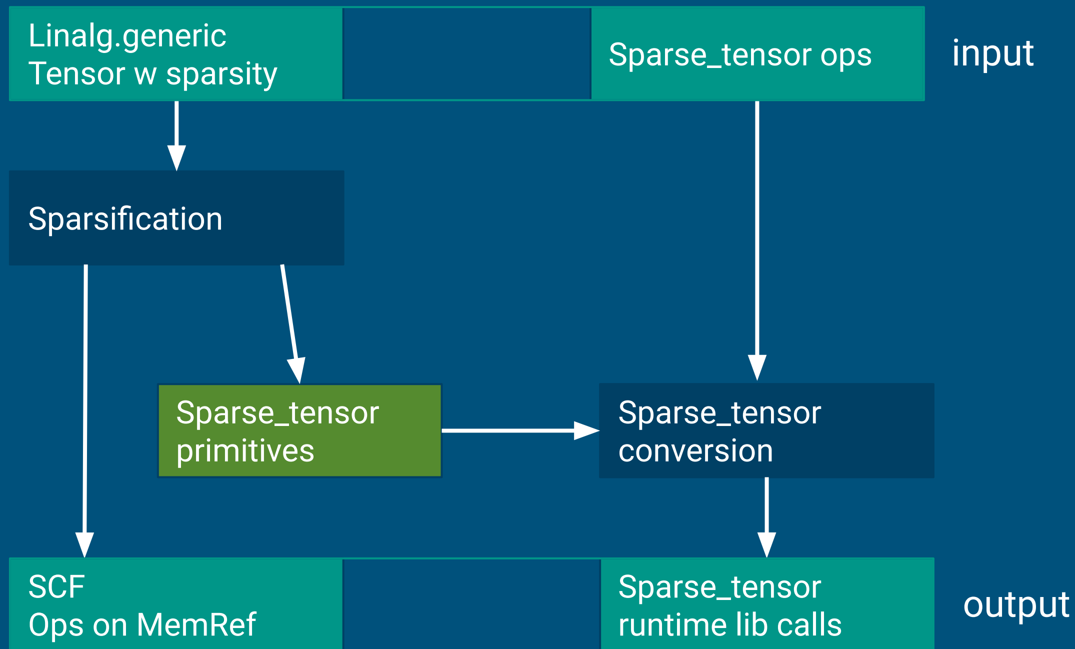
- Implicit broadcasting: broadcast along “missing dimensions”
  - Expression: ***C[i, j] = A[i, j] + B[i]***
  - Semantics: ***C[i, j] = A[i, j] + broadcast(j, B[i])***
- Reduction: op-reduce over “reduction dimension”
  - Expression: ***C[i] = A[i, j] + B[i, j]***
  - Semantics: ***C[i] = op(j, A[i, j] + B[i, j])***
- Reduction is performed on the outer expression
  - Expression: ***D[i] = A[i, j] + B[i, j] + C[i]***
  - Semantics: ***D[i] = sum(j, A[i, j] + B[i, j], broadcast(j, C[i]))***
  - PyTACO Semantics: ***D[i] = sum(j, A[i, j] + B[i, j]) + C[i]***

Same as PyTACO

Similar to PyTACO

Different from PyTACO

Sparse Tensor Compiler



MLIR

Outline

- What Is MLIR-PyTACO
- Sparse Tensor Algebra Support in MLIR
- Implementing PyTACO With MLIR

MLIR Python Bindings

- Thanks for the easy to use Python bindings for
  - Constructing instructions, functions
    - builtin.FuncOp.from_py_func
  - Constructing compilation pipelines
  - Building a JIT engine for a compiled module
  - JIT engine execution
- Linalg.generic through linalg.OpDsl

Linalg.OpDsl Python Bindings

- Construct the AST representation for tensor expressions
  - LinalgOpDef
    - Comprehension: a tensor assignment
  - OperandDef: Input and Output of the function
    - SymbolDef: Dimension in the iteration space
  - TensorUse: represent operands
    - DimDef: Dimension in the operand
  - Operations
    - ReduceApply
    - PrimApply
- Convert the AST to lingalg.generic
  - DefineOpCallable

MLIR

Tensor Expression to Linalg.opdsl AST

C[i]

*

A[i,j]

B[i,j]

DefineOpCallable
LinalgOpDef

Comprehension

ReduceApply(“add”)

PrimApply(“mul”)

TensorUse

TensorUse

TensorUse

OperandDef
(“A”, I)

DimDef(“j”)

OperandDef
(“B”, I)

DimDef(“i”)

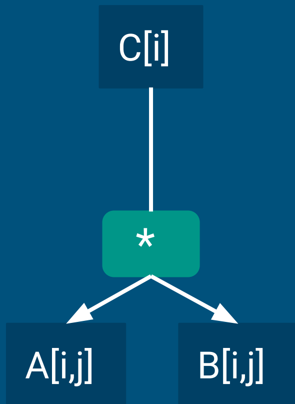
OperandDef
(“C”, O)

SymbolDef(“j”)

SymbolDef(“i”)

MLIR

Linalg.opdsl AST to linalg.generic




```
#trait = {
  indexing_maps = [
    affine_map<(i,j) -> (i,j)>,  // A
    affine_map<(i,j) -> (i,j)>,  // B
    affine_map<(i,j) -> (i)>   // C
  ],
  iterator_types = ["parallel", "reduction"],
}
%0 = linalg.generic #trait
  ins(%argA, %argB : !typeA, !typeB)
  outs(%argC : !typeC) {
    ^bb0(%A: f64, %B: f64, %C: f64):
      %1 = arith.mulf %A, %B : f64
      %2 = arith.addf %C, %1 : f64
      linalg.yield %2 : f64
    } -> !typeC
```

MLIR

Reduction on “smallest subexpressions”

D[i] = A[i, j] * B[i, j] + C[i]

Implemented with one linalg.generic gives incorrect result:

D[i] = sum(j, A[i, j] * B[i, j] + broadcast(j, C[i]))

Need to implement with two linalg.generic:

// D[i] = sum(j, A[i, j] * B[i, j])+ C[i]

T[i] = A[i, j] * B[i, j]

D[i] = T[i] + C[i]

MLIR

Estimate the Sparsity of Temporary Tensors

- op(1, 0) = 0, computing disjunction of zero values
  - The result dimension is sparse if either source dimensions is sparse
- op(1, 0) != 0, computing conjunction of zero values
  - The result dimension is sparse if both source dimensions are sparse

Supporting Python Classes

- IndexVar
- ModeFormat
- Tensor
  - __getitem__: construct a tensor Access
  - __setitem__: construct an IndexExpr assigned to the tensor
  - evaluate: evaluate the tensor assignment
- Access
  - __add__, __sub__, etc: Construct IndexExpr
- IndexExpr

MLIR

Evaluating Tensor Assignment

1. Validate the assignment and collect info to support codegen
2. Identify subexpressions corresponding to a linalg.generic
3. Create an MLIR module

   For each identified subexpression, in reversed topological order

   Emit linalg.generic via DefinedOpCallable
4. Compile the MLIR module
5. Build an execution engine for the MLIR module
6. Get ctype pointers for the input and output tensor storage
7. Invoke the execution engine and retrieve the result

Source Code Organization

https://github.com/llvm/llvm-project/tree/main/mlir/test/Integration/Dialect/SparseTensor/taco

Aliases to expose PyTACO API
mlir_pytaco_api.py

Tensor
mlir_pytaco.py

Tensor I/O
mlir_pytaco_io.py

PyTACO API test
Test_simple_tensor_algebra.py
test_SpMM.py
test_SpMV.py
test_MTTKRP.py

Unit Test
unit_test_tensor_utils.py
unit_test_tensor_io.py

MLIR

Ongoing/Future Work

- Support transpose
- Performance tuning
- Support more PyTACO functionality

Paper "**Compiler Support for Sparse Tensor Computations in MLIR**"

Aart J.C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad

Available as pre-print on ArXiv

MLIR

For questions on MLIR-PyTACO, please contact

Aart Bik ajcbik@google.com

Bixia Zheng bixia@google.com