Compiler Support for Sparse Tensor Computations in MLIR

2021 LLVM Developers' Meeting

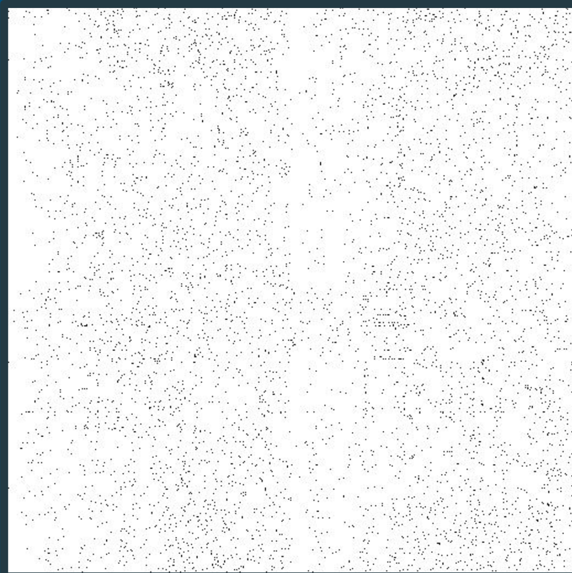
Aart J.C. Bik
ajcbik@google.com
https://www.aartbik.com/

Part I:
Preliminaries on Sparse Computations and Compilers

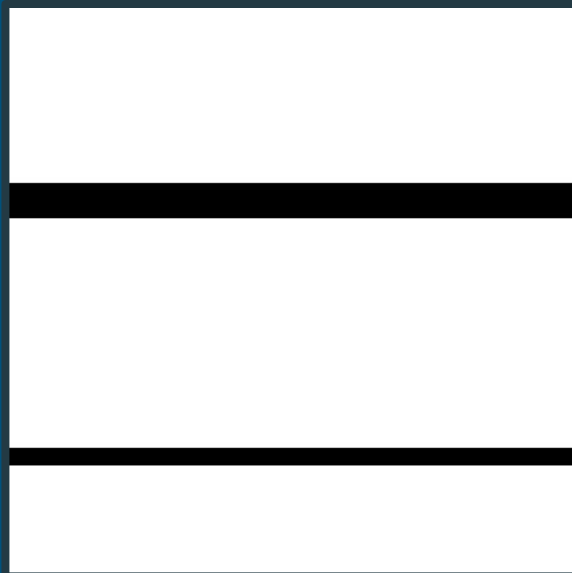
Sparse Vectors, Matrices, Tensors

A vector, matrix, tensor with *many zero elements* is called a ***sparse*** vector, matrix, tensor

General Sparse



Structured Sparse



Taking Advantage of Sparsity

Sparsity can be used to

(1) Reduce ***storage requirements***, by not storing the zeros (*primary storage* for the nonzero elements and *secondary storage* to reconstruct enveloping tensor)

(2) Reduce ***computational time***, by only operating on nonzero elements (use properties like $X+0=X, X*0=0$ etc.)

Complications with Sparse Code

Exploiting sparsity comes at a cost, though, since developing and maintaining sparse code by hand is a complex and error-prone task

```
// Fortran example for B(I) += A(I,J) * X(J)
for I = 1, M
  for JJ = APTR(I), APTR(I+1) - 1
    J = AJDX(JJ)
    B(I) = B(I) + AVAL(JJ) * X(J)
```

Sparse code also lacks spatial and temporal locality, and disables most compiler optimizations

Solution: Sparse Compiler Support

Treat sparsity as a ***property*** (or type), not a tedious implementation detail, and let a ***sparse compiler*** generate sparse code automatically from a sparsity-agnostic (viz. “dense”) definition

This idea was

- pioneered for linear algebra in MT1 (1996)
- formalized to tensor algebra in TACO (2017)

Aart J. C. Bik

Part II:
Sparse Tensor Support in MLIR

MLIR Support for Sparse Tensors

In Nov 2020, we started to add ***sparse tensor*** support to MLIR (LLVM's extensible infrastructure for building domain specific compilers)

Most attributes, types, operations, and rewriting rules related to sparse tensor support reside in the SparseTensor Dialect

Design Philosophy

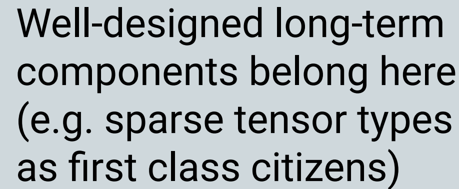
Our “*North Star*”

- Provide an excellent ***sparse tensor ecosystem*** to academia and industry based on first principles
- Design MLIR with the most innovative foundations for developing new technologies related to sparse compilers

Our “*Reference Implementation*”

- Provide a working implementation of the sparse ecosystem as a first basis against which future improvements can be compared

Design Philosophy



Our “*North Star*”

- Provide an excellent ***sparse tensor ecosystem*** to academia and industry based on first principles
- Design MLIR with the most innovative foundations for developing new technologies related to sparse compilers

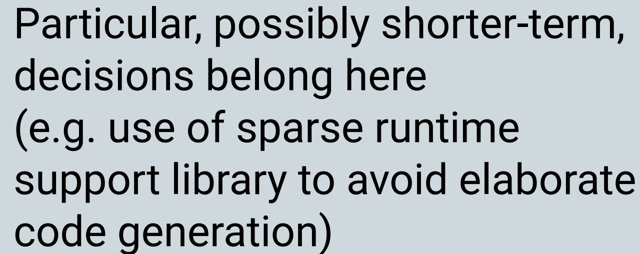
Our “*Reference Implementation*”

- Provide a working implementation of the sparse ecosystem as a first basis against which future improvements can be compared

Design Philosophy

Our "*North Star*"

- Provide an excellent ***spars***
  industry based on first
- Design MLIR with the m
  new technologies related to s



Our "*Reference Implementation*"

- Provide a working implementation of the sparse ecosystem as a first basis against which future improvements can be compared

Sparse Tensor Types

Central to this support was the introduction of sparse tensor types as first-class citizens to MLIR using TACO-flavored annotations

```
// Compressed Sparse Column
#CSC = #sparse_tensor.encoding<{
  dimLevelType = [ "dense", "compressed" ], // per-dimension dense/sparse
  dimOrdering = affine_map<(i,j) -> (j,i)>, // dimension ordering
  pointerBitWidth = 32, indexBitWidth = 32  // storage bit widths
}>

tensor<100x100xf64, #CSC>
```

Sparse Tensor Storage

```
%t = arith.constant dense<[
   [1.0,  0.0,  3.0],
   [0.0,  2.0,  4.0],
   [0.0,  0.0,  0.0],
   [0.0,  0.0,  5.0] ]> : tensor<4x3xf64>

%s = sparse_tensor.convert %t
   : tensor<4x3xf64> to tensor<4x3xf64, #CSC>
```

Eventually, the sparse tensor is bufferized into something like

```
pointers   : 0 1 2 5
indices    : 0 1 0 1 3
values     : 1.0 2.0 3.0 4.0 5.0
```

Sparse Tensor Usage

Dense matrix multiplication kernel

```
%0 = linalg.matmul
  ins(%a, %b: tensor<10x20xf32>, tensor<20x30xf32>)
  outs(%c: tensor<10x30xf32>) -> tensor<10x30xf32>
```

Sparse matrix multiplication kernel (yes, it’s that simple!)

```
%0 = linalg.matmul
    ins(%a, %b: tensor<10x20xf32, #CSC>, tensor<20x30xf32>)
    outs(%c: tensor<10x30xf32>) -> tensor<10x30xf32>
```

Reference Implementation

A set of ***rewriting and lowering rules*** takes care of converting annotated IR to sparse storage schemes and imperative constructs that only store and (co-)iterate over the nonzero elements

- Bufferization of dense and sparse tensors (memrefs)
- “One-size-fits-all” sparse runtime support library in lieu of excessive code generation for commonly used routines
- Clear “dense” semantics allow for on-the-fly
  - Loop optimization
  - Vectorization (SIMD)
  - Parallelization

MLIR Sparse Tensor Support Modi Operandi

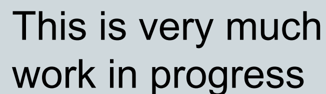
(1) Complete end-to-end JIT/AOT execution

A novice programmer uses an array programming language like Python with sparse annotations to generate and execute sparse code fully automatically

(2) Sparse library development (possibly through search)

An expert programmer experiments with many different sparse storage schemes and/or compiler code generation strategies with “the push of a button” before “freezing” final library version that ships in production

MLIR Sparse Tensor Support M



(1) Complete end-to-end JIT/AOT execution

A novice programmer uses an array programming language like Python with sparse annotations to generate and execute sparse code fully automatically

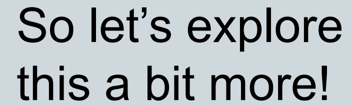
(2) Sparse library development (possibly through search)

An expert programmer experiments with many different sparse storage schemes and/or compiler code generation strategies with “the push of a button” before “freezing” final library version that ships in production

MLIR Sparse Tensor Support Modi Operandi

(1) Complete end-to-end JIT/AOT execution

A novice programmer uses an array programmi
Python with sparse annotations to generate and
code fully automatically



(2) Sparse library development (possibly through search)

An expert programmer experiments with many different sparse storage schemes and/or compiler code generation strategies with "the push of a button" before "freezing" final library version that ships in production

Sampled Dense-Dense Matrix Multiplication

SDDMM is a reusable core primitive for many ML algorithms (Alternating Least Squares, Latent Dirichlet Allocation, Sparse Factor Analysis, Gamma Poisson)

$$P = (A \cdot B^T) \otimes S$$ ***// matrix mult and elt-wise sampling***

A simple DGEMM followed by sampling library composition performs ***asymptotically worse*** than a fused sparse kernel implementation: $O(m \cdot n \cdot k)$ vs. $O(k \cdot nnz)$

Fused SDDMM Intermediate Representation

```
%0 = linalg.generic #trait_sampled_dense_dense
      ins(%args, %arga, %argb:
          tensor<64x64xf64, #SparseMatrix>, // what to use?
          tensor<64x64xf64>,
          tensor<64x64xf64>)
      outs(%argx: tensor<64x64xf64>) {
        ^bb(%s: f64, %a: f64, %b: f64, %x: f64):
          %0 = mulf %a, %b : f64
          %1 = mulf %s, %0 : f64
          %2 = addf %x, %1 : f64
          linalg.yield %2 : f64
    } -> tensor<64x64xf64>
```

Performance State Space (Combinatorial Explosion)

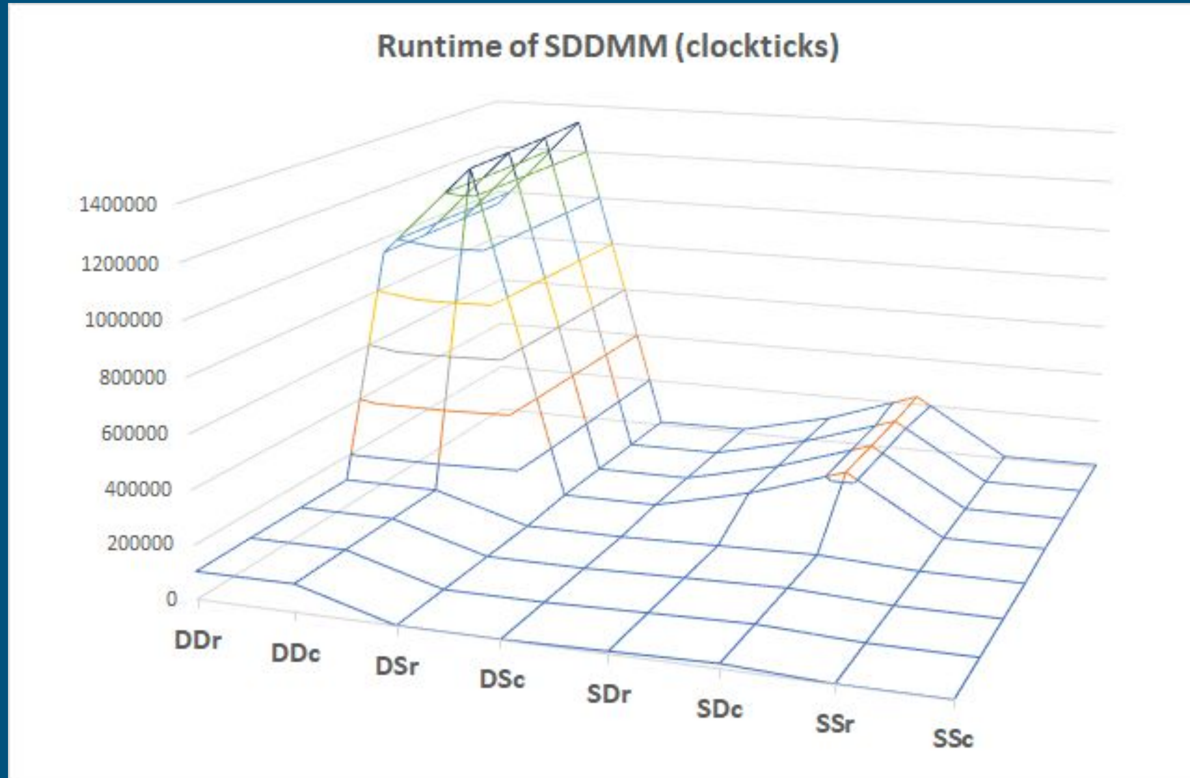
- **Storage Selection**
  Single 2-dim sparse tensor: dense-sparse, row-/column-wise, 32-/64-bit pointers and indices : $2^2 \times 2 \times 2 \times 2 = 32$ combinations
- **Compiler strategies**
  Different loop orderings (ijk, ikj, kij), vectorization strategies, vector lengths, etc. (just a subset here) : an additional 96 combinations
- **Representative Sparse Tensors**
  Only consider 64x64 matrices with 16 elements (very sparse 99.6%)
- **Targets**
  Only consider single-threaded Intel Xeon W-2135 3.7GHz with AVX512

For a total of “just” ***3072 combinations***

Python DSL in MLIR to Implement Search

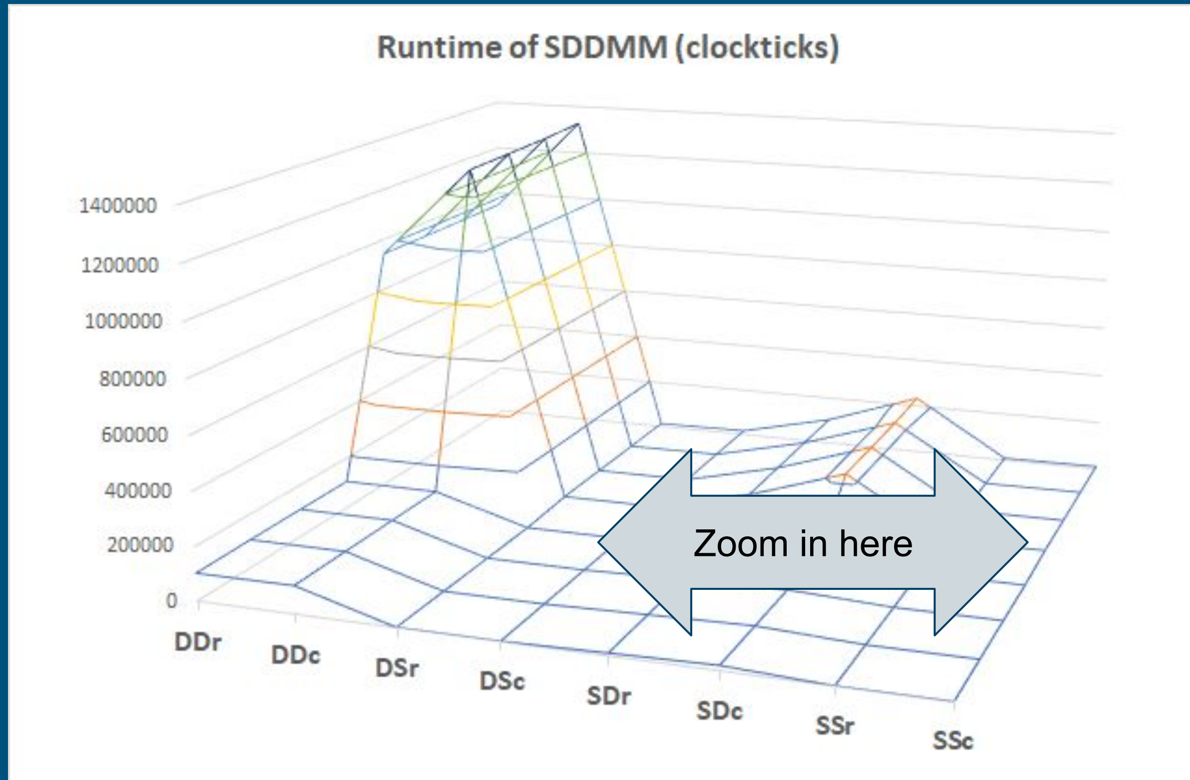
```
from mlir.dialects import sparse_tensor as st

levels = [[st.DimLevelType.dense, st.DimLevelType.dense],
          [st.DimLevelType.dense, st.DimLevelType.compressed],
          [st.DimLevelType.compressed, st.DimLevelType.dense],
          [st.DimLevelType.compressed, st.DimLevelType.compressed]]
orderings = [ ir.AffineMap.get_permutation([0, 1]),
              ir.AffineMap.get_permutation([1, 0]) ]
bitwidths = [32, 64]


for level in levels:
  for ordering in orderings:
    for pwidth in bitwidths:
      for iwidth in bitwidths:
        attr = st.EncodingAttr.get(level, ordering, pwidth, iwidth)
          ...
          build_compile_and_run_SDDMM(attr, ...)
```

SDDMM Performance State Space Search



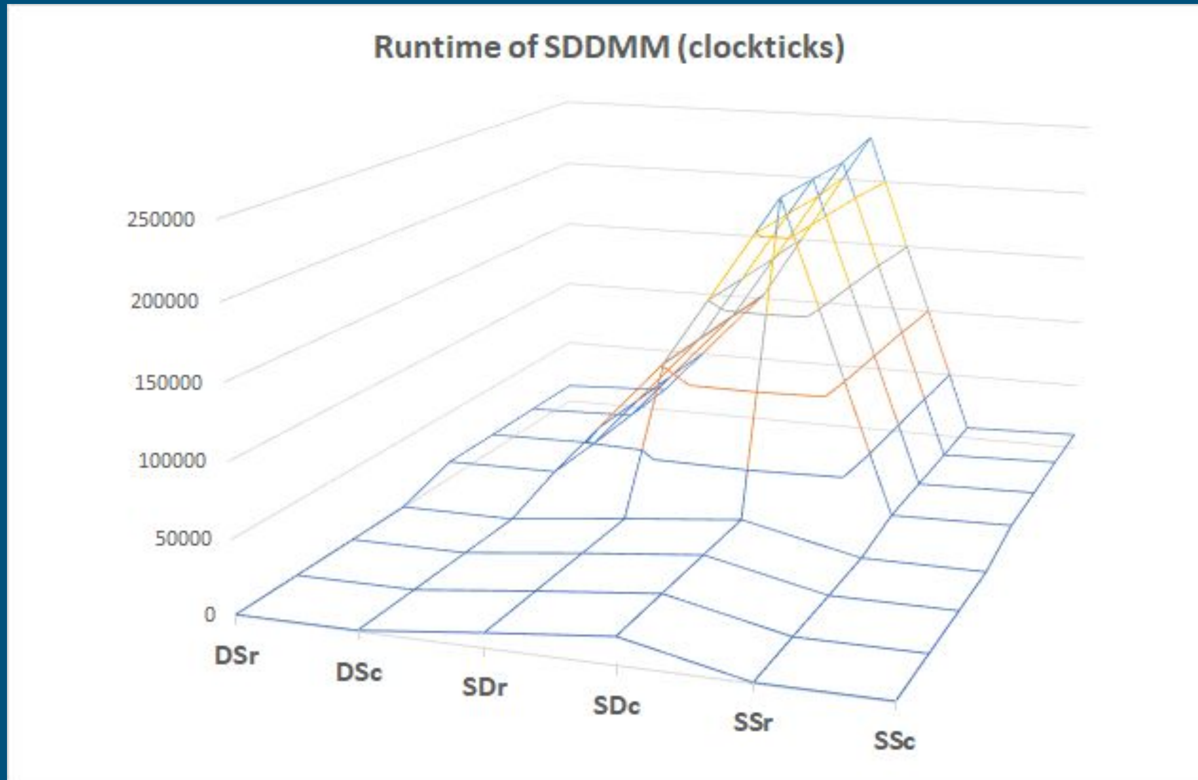
height: runtime in clock ticks (lower is better)
left-to-right: **DDr** = dense-dense row-wise, **DSc** = dense-sparse column-wise, etc.
front-to-back: various optimization settings, loop ordering, vectorization, etc.

SDDMM Performance State Space Search



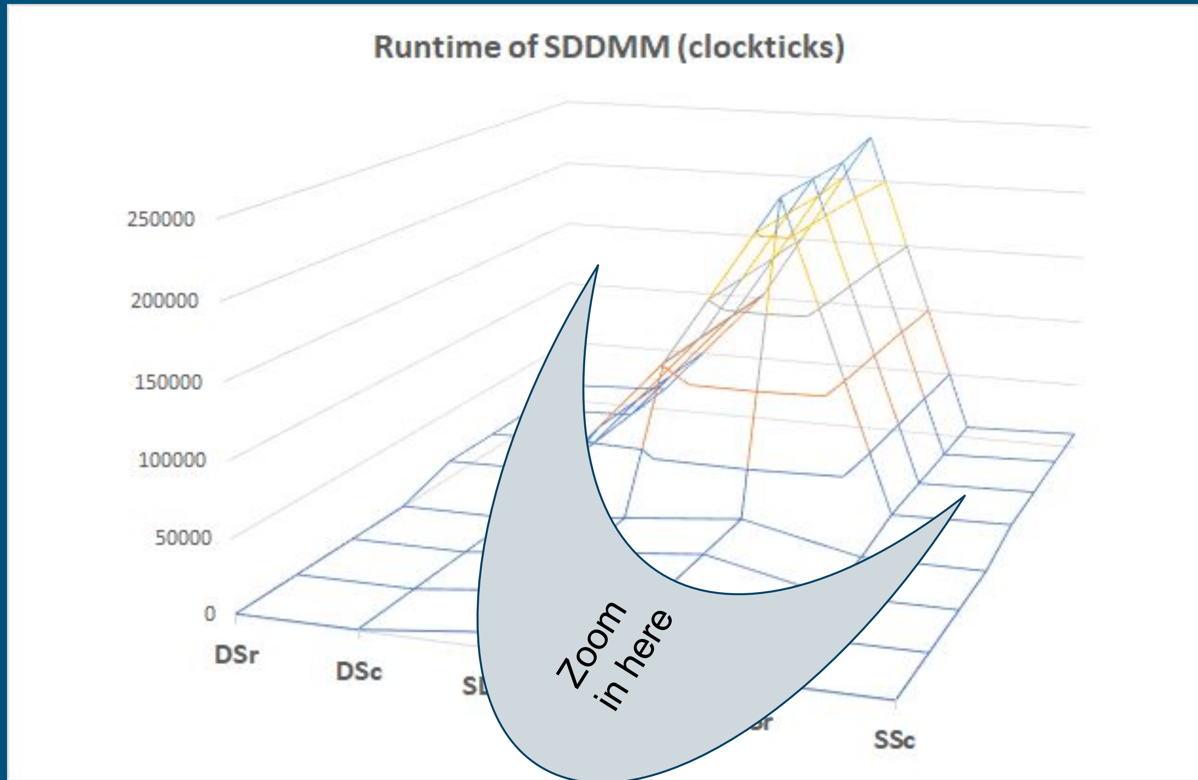
height: runtime in clock ticks (lower is better)
left-to-right: **DDr** = dense-dense row-wise, **DSc** = dense-sparse column-wise, etc.
front-to-back: various optimization settings, loop ordering, vectorization, etc.

SDDMM Performance State Space Search



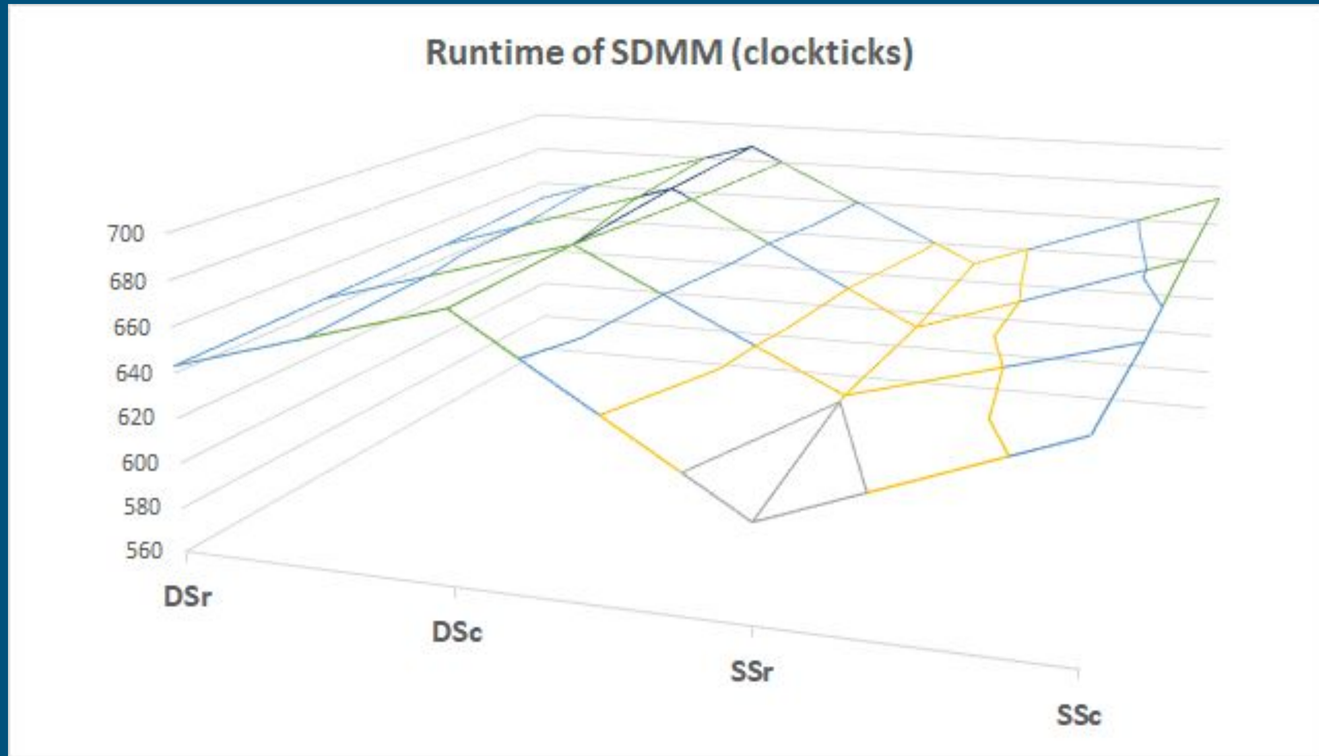
height: runtime in clock ticks (lower is better)
left-to-right: **DDr** = dense-dense row-wise, **DSc** = dense-sparse column-wise, etc.
front-to-back: various optimization settings, loop ordering, vectorization, etc.

SDDMM Performance State Space Search



height: runtime in clock ticks (lower is better)
left-to-right: **DDr** = dense-dense row-wise, **DSc** = dense-sparse column-wise, etc.
front-to-back: various optimization settings, loop ordering, vectorization, etc.

SDDMM Performance State Space Search



height: runtime in clock ticks (lower is better)
left-to-right: **DDr** = dense-dense row-wise, **DSc** = dense-sparse column-wise, etc.
front-to-back: various optimization settings, loop ordering, vectorization, etc.

State Space Search Result

Best sparse tensor type annotation

```
#sparse_tensor.encoding<{
   dimLevelType = [ "compressed", "compressed" ],
   dimOrdering = affine_map<(d0, d1) -> (d0, d1)>,
   pointerBitWidth = 64,
   indexBitWidth = 32
}>
```

Best optimization

i-j-k loop, all vectorized, VL=16, enable-SIMD-index32=True

Part III:
A Few Reference Implementation Details

Running Example

Simple scaling `x(i) *= 2.0` operation on a sparse vector

```
#SV = #sparse_tensor.encoding<{ dimLevelType = [ "compressed" ] }>

func @scale(%argx: tensor<32xf32, #SV> {linalg.inplaceable = true})
                -> tensor<32xf32, #SV> {
  %c = arith.constant 2.0 : f32
  %0 = linalg.generic #trait_scale
  outs(%argx: tensor<32xf32, #SV>) {
      ^bb(%x: f32):
        %1 = arith.mulf %x, %c : f32
        linalg.yield %1 : f32
  } -> tensor<32xf32, #SV>
  return %0 : tensor<32xf32, #SV>
}
```

Lowering to Loops

```
func @scale(%argx: tensor<32xf32, #SV> ...

  %0 = sparse_tensor.pointers %argx, %c0 : memref<?xindex>
  %1 = sparse_tensor.indices  %argx, %c0 : memref<?xindex>
  %2 = sparse_tensor.values   %argx      : memref<?xf32>

  %3 = memref.load %0[%c0] : memref<?xindex>
  %4 = memref.load %0[%c1] : memref<?xindex>
  scf.for %arg1 = %3 to %4 step %c1 { // only stored entries
    %6 = memref.load %2[%arg1] : memref<?xf32>
    %7 = arith.mulf %6, %cst : f32
    memref.store %7, %2[%arg1] : memref<?xf32>
  }
```

Bufferization of Sparse Storage Schemes

Unlike dense tensors, which bufferize directly to memrefs, sparse tensors first bufferize to opaque pointers: the actual sparse storage scheme is implemented in a “one-size-fits-all” sparse runtime support library which provides access to the underlying memrefs

```
func @scale(%argx: !llvm.ptr<i8>)
  ...
  %0 = call @sparsePointers(%argx, %c0)
     : (!llvm.ptr<i8>, index) -> memref<?xindex>
  %1 = call @sparseIndices(%argx, %c0)
     : (!llvm.ptr<i8>, index) -> memref<?xindex>
  %2 = call @sparseValuesF32(%argx)
     : (!llvm.ptr<i8>) -> memref<?xf32>
  ...
```

Vectorization

Supported via the Vector Dialect, which is an architectural neutral dialect for SIMD that relies on the LLVM backend to generate architectural specific SIMD code

```
scf.for %arg1 = %4 to %6 step  %c8 {
  %8 = affine.min #map(%6, %arg1)[%c8]
  %9 = vector.create_mask %8 : vector<8xi1>
  %10 = vector.maskedload %2[%arg1], %9, %cst
        : memref<?xf32>, vector<8xi1>, vector<8xf32> into vector<8xf32>
  %11 = arith.mulf %10, %cst_0 : vector<8xf32>
  vector.maskedstore %2[%arg1], %9, %11
        : memref<?xf32>, vector<8xi1>, vector<8xf32>
}
```

```
// AVX512
   vmovdqa ymm0, ymmword ptr [rip + .LCPI3_0] # ymm0 = [0,1,2,3,4,5,6,7]
L: vpbroadcastd ymm1, edi
   vpcmpgtd k1, ymm1, ymm0
   vmovups ymm1 {k1} {z}, ymmword ptr [rsi + 4*rax]
   vaddps  ymm1, ymm1, ymm1
   vmovups ymmword ptr [rsi + 4*rax] {k1}, ymm1
   ...
```

Part IV:
Final Remarks

Conclusions

MLIR simplifies sparse code development and maintenance

Sparsity is treated as a ***property***, not a tedious implementation detail

Design philosophy consists of long-term “north star” vision together with a shorter-term “reference implementation”

Two modi operandi

- End-to-end JIT/AOT execution of array programming languages with sparse annotations
- Focused library development that explores the state space of all possible sparse storage schemes, compiler optimizations, tensor inputs, and targets

How to Contribute

Design

- Extend sparsity annotations (for example, generalizing dimension level formats as shown in [Chou2018 et al.], but also generalizing to structured sparsity support)
- Introduce sparse iteration abstractions that allow for more gradual progressive lowering from IR with sparse types to co-iterating constructs
- Introduce sparse memref abstractions that allow for more gradual progressive bufferization of sparse types
- Participate in ongoing discussions or post new ideas as RFCs at discourse forum

Stability and Performance Evaluation

- Contribute tests or benchmarks
- Analyze compile-time and run-time performance

How to Contribute (continued)

Code

- Implement MLIR Front-End that maps your favorite array language with sparse tensor annotations to MLIR IR with sparse tensor types
- Explore completely different alternative implementations, our “reference implementation” is just a starting point
- MLIR accepts code contributions via Phabricator
- Bugzilla sparsity related starter tasks and feature requests

Use

- Use MLIR and the Python DSL to explore many different sparse storage schemes, compiler optimizations, sparse tensor properties, and targets while developing new sparse code; Please let us know your experience!

Thanks for Listening!

More information about MLIR, please refer to
https://mlir.llvm.org/

For questions on sparse tensor support, please contact
Aart Bik at ajcbik@google.com

Many thanks to the MLIR team at-large, with a special shout-out to Mehdi Amini, Sanjoy Das, Stephan Herhut, Penporn Koanantakool, Stella Laurenzo, Andrew Leaver, Jacques Pienaar, Thomas Raoux, River Riddle, Tatiana Shpeisman, Sean Silva, Gus Smith, Reid Tatge, Jake van der Plas, Nicolas Vasilache, Bixia Zheng, Alex Zinenko