Sparse Matrix Support in LLVM

COMS 6156 Final Project Report

Eric Feng and Chris Yoon (Group Name: "EricAndChris")

ef2648@columbia.edu cjy2129@columbia.edu

Contents

1 Introduction

1.1 Sparse Linear Algebra

Sparse matrices are matrices where a significant number of its entries are zeros. They are ubiquitous in practical applications, such as machine learning, deep learning, computer graphics, networks, and graphs [1]. Treating sparse matrices like *dense matrices* (matrices whose elements are mostly nonzero), would waste memory and time: since operations on zeroes are often no-ops (e.g. $x+0=0$, $x\times 0=0$), they do not need to be stored explicitly in memory and sparsity-aware linear algebra operations can avoid wasting CPU/GPU cycles on no-op operations.

Sparsification of matrices is a program optimization technique that compresses sparse matrices by encoding zeros more compactly, and provides sparsity-aware linear algebra operations (such as transposition, multiplication, addition, and subtraction). Through sparsification, programs performing operations on large sparse matrices can significantly reduce both memory usage and runtime.

Mature sparse tensor libraries and compilers typically offer a variety of storage formats, enabling users to select their desired format or automatically generating the most suitable representation for their needs through heuristics. A fundamental and widely-used format is the *Compressed Sparse Column* (CSC) format, which supports efficient arithmetic operations among matrices of the same format, as well as effective column slicing and relatively fast matrix-vector products [2]. CSC is the default format for sparse matrices in many prominent environments, such as MATLAB [3]. In this project, we specifically focused on the CSC format because of its common use as a fundamental representation in previous studies, its efficient and straightforward operations, and its compatibility with LLVM dense matrices, which are stored by default in column-major order. This compatibility facilitates easier conversions between column-major and CSC formats. However, it is important to note the existence of more size-optimized variants, such as the *Doubly Compressed Sparse Column* format as well as other representations such as COO (Coordinate list), List of lists (LIL), and Dictionary of keys (DOK).

The CSC format stores the only the nonzero elements of a matrix, along with its indices and pointers:

$$A = \begin{pmatrix} a_{0,0} & 0 & 0 & a_{0,3} \\ 0 & 0 & 0 & 0 \\ a_{2,0} & 0 & 0 & 0 \end{pmatrix}$$

`pointers[1]:`	0	2	2	3
`indices[1]:`	0	3	0	
`values:`	$a_{0,0}$	$a_{0,3}$	$a_{2,0}$	

Figure 1: The CSC format

The `indices` array denotes that `value[i]` is stored in column `i`; and the `pointers` arrays denote up to which indices the elements in `values` belong to a specific row — the elements `a[0][0]` and `a[0][3]` (from index 0 to 2 noninclusive) belong to the first row; 2 is repeated since there are no elements in the second row, and the last element `a[2][0]` belongs to the third row.

1.2 Sparse Linear Algebra as a Compiler Feature

Typically, sparse matrix linear algebra were optimizations provided via external (and often proprietary) accelerated linear algebra libraries, such as Intel MKL [4] and BLAST [5]. Yet, recent directions in compiler development began to consider sparse linear algebra as a "first-class language feature" that can be implemented at the compiler level for programming language developers. This is aligned with the compiler community recognizing constructs like tensors and dense linear algebra as first-class features of modern programming languages.

The prominent value propositions of this trend is that existing and new languages that want to implement sparse linear algebra need not implement the whole framework themselves, nor rely on proprietary or bulky external frameworks, but just call built-in compiler pipelines to generate sparse linear algebra code. For the case of mature and large compiler development infrastructure like the LLVM and MLIR, this means programming language developers can also enjoy *optimized* code that use the infrastructure's existing optimization passes (such as auto-vectorization, auto-parallelization, GPU offloading, and loop optimizations) for free. The remainder of this section provides a brief history of sparse compilers; as well as the recent *first* adoption of built-in sparse matrix support in a commercial-grade compiler infrastructure, MLIR.

MT1: A Sparse Compiler

The concept of exploiting sparsity at the compiler level is not novel and indeed boasts a substantial historical backdrop. The seminal work in sparse compiler support was laid by Aart J.C Bik in his PhD thesis, *Compiler Support for Sparse Matrix computations* [6]. In this work, Bik introduced a sparse compiler — MT1 — which is specifically designed for linear algebra applications. The compiler treats sparsity as a property rather than an implementation detail and allows the automatic generation of sparse code from a density-agnostic (i.e dense) representation.

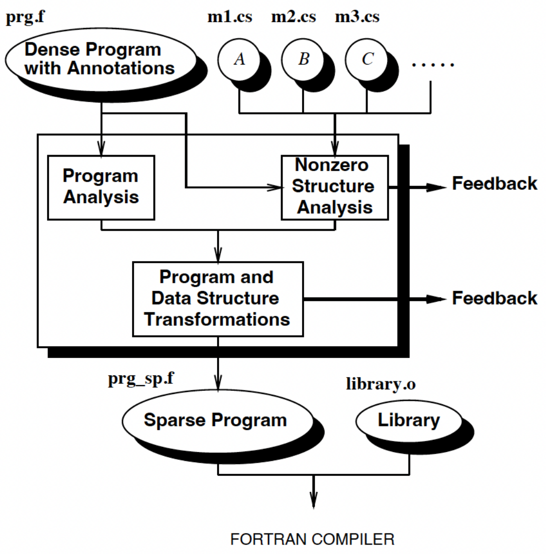

Figure 2: Organization of MT1 compiler [6]

MT1 is a kind of source-to-source "restructuring compiler" — it takes in an assumed dense Fortran program with linear algebra operations structured using two-dimensional arrays, and returns a semantically equivalent sparse program. This conversion process reconfigures the source program's primitive data structures and data accesses with more complex sparse structures that reduce the storage demands and computational time of the original program. The structure of this process, including the compilers' decisions for choosing the sparse structures to be used, is based on perceived sparse accesses to the arrays. It also supports annotations from the programmer, which may influence certain decisions in the code transformation process.

TACO: The Tensor Algebra Compiler

The MT1 compiler was limited to 2-dimensional matrices. Tensors, which generalize vectors and matrices to N-dimensions, permit multilinear computations. The concept originally proposed in MT1, which involved generating code for sparse matrices, was extended to tensor algebra with the introduction of TACO in 2017 [7]. TACO enables users to easily specify the sparse computations in which they are interested. TACO may be invoked via its C++ library or command-line interface to generate C code, which is then compiled using the system's compiler.

```
Format csr({Dense,Sparse});
Tensor<double> A({64,42}, csr);

Format csf({Sparse,Sparse,Sparse});
Tensor<double> B({64,42,512}, csf);

Format svec({Sparse});
Tensor<double> c({512}, svec);

B.insert({0,0,0}, 1.0);
B.insert({1,2,0}, 2.0);
B.insert({1,2,1}, 3.0);
B.pack();

c.insert({0}, 4.0);
c.insert({1}, 5.0);
c.pack();

IndexVar i, j, k;
A(i,j) = B(i,j,k) * c(k);

A.compile();
A.assemble();
A.compute();
```

```
$taco "A(i,j) = B(i,j,k) * c(k)" -f=A:ds -f=B:sss -f=c:s
// ...
int pA2 = A2_pos[0];
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_idx[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_idx[pB2];
    double tk = 0.0;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while ((pB3 < B3_pos[pB2+1]) && (pc1 < c1_pos[1])) {
      int kB = B3_idx[pB3];
      int kc = c1_idx[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        tk += B_vals[pB3] * c_vals[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
    A_vals[pA2] = tk;
    pA2++;
  }
}
```

Figure 3: The left shows SpMV with the taco C++ library and the right shows the same using the taco command line interface. The generated code is shown after the first line on the right [7]

Sparse Tensor Dialect in MLIR

MLIR [8] is a compiler infrastructure designed for building compilers tailored to domain-specific languages and applications. Its core feature is an extensible framework that allows for the specification of "new" IRs through "dialects" and transformations of these dialects. During the compilation process, multiple dialects can coexist, forming a hybrid representation of a program. MLIR also supports the progressive lowering of dialects towards the target representation. In this context, sparse tensor operations in MLIR are facilitated through the sparse tensor dialect, which assumes it receives a sequence of operations in the Linalg dialect (presumed to be lowered from a higher-level dialect, e.g., TOSA) [9]. At this level, sparse tensor values may be easily constructed or converted from their dense counterparts through TACO-like annotations. Following the paths set by MT1 and TACO, the sparse tensor dialect then automatically generates sparse code in the form of the SCF dialect. The SCF, or Structured Control Flow dialect, represents imperative constructs such as loops and conditionals at a high level of abstraction. The SCF dialect is then further lowered to lower-level dialects (e.g., control-flow or CF) until eventually being translated to the target representation (e.g., LLVM IR).

Listing 1: Dense matrix multiplication can be changed to a sparse operation in MLIR by just adding a sparse "attribute"

```
// dense operation
%C = linalg.matmul ins(%A, %B: tensor<?x?xf64>, tensor<?x?xf64>) -> tensor<?x?xf64>

// sparse operation
%C = linalg.matmul ins(%A, %B: tensor<?x?xf64, #CSC>, tensor<?x?xf64>) ->
    tensor<?x?xf64
```

1.3 Project Objectives and Research Questions

The works presented in MT1, TACO, and MLIR demonstrate, in one form or another, the concept of a sparse "compiler" in the sense of automatically generating sparse code for linear and tensor algebra. However, code generation in all of the aforementioned projects involves either translation to a semantically equivalent sparse program (e.g., Fortran to Fortran for MT1) or generating code at the same or a slightly lower level of abstraction (e.g., C++ to C for TACO's library and SparseTensor dialect to SCF in MLIR). In this project, we aim to explore the support of sparse linear algebra as first-class citizens at a lower level of abstraction, generating instruction-level IR within the LLVM compiler infrastructure. We believed that the value propositions were clear:

1. Languages that target LLVM can incorporate native sparse matrix algebra support; not all languages currently (or plan to) compile to MLIR
2. Users of such languages can then immediately enjoy the adoption of sparse compilation in the language
3. With the easy incorporation of sparse compilation, linear algebra code will be *more memory efficient* and *faster*

We believed that this was also aligned with LLVM's recent directions, since they have also recently implemented first-class support for dense linear algebra.

That said, our final project was formulated to investigate the following research questions:

- **RQ1:** How can first-class support for sparse matrices be implemented inside LLVM with more lower-level abstractions than those available in MLIR?
- **RQ2:** What abstractions are needed to implement code generation for sparse matrices?
- **RQ3:** Does first-class support for sparse matrices in LLVM ease the implementation of adopting sparse matrices in the front-end language?

2 Attempts at Sparse Matrix Support in LLVM

Our main objective for the final project was to implement built-in support for sparse matrices in the LLVM infrastructure, inspired by the development of sparse tensor support in MLIR. Due to the additional complexity involved in algorithms for tensors, we limited our goal to 2D matrices, like the recent addition of built-in support for dense matrix linear algebra in LLVM. With this in mind, we envisioned support for sparse matrices for LLVM to be implemented similarly to dense matrices.

However, we discovered that the limitations of LLVM make it extremely difficult to implement built-in sparse matrix support, and concluded LLVM is not at the appropriate level of abstraction for sparse matrices. In this section, we first detail how we envisioned it would be implemented, then our attempts at its implementation, the limitations of LLVM that eventually stopped us, and finally a discussion of how MLIR overcame such limitations.

2.1 Matrices in LLVM and LLVM Intrinsics

Our main inspiration for our approach was implementation of built-in matrix linear algebra support in LLVM. Matrix support in LLVM was implemented via LLVM Intrinsics. LLVM Intrinsics are operations that can be matched in the LLVM backend to run the specific corresponding code generation pipeline. Consider the following `C` program, which will eventually call the matrix Intrinsics:

```
typedef float mat __attribute__((matrix_type(2, 2)));

void main() {
    float pa[4] = {1.0, 2.0, 3.0, 4.0};
    float pb[4] = {1.0, 0.0, 0.0, 1.0};

    mat a = __builtin_matrix_column_major_load(pa, 2, 2, 4);
    mat b = __builtin_matrix_column_major_load(pb, 2, 2, 4);
    mat r = a * b // overloaded matmul Intrinsincs

    __builtin_matrix_column_major_store(r, pa, 4);
}
```

The program above would generate the following LLVM IR:

```
  ... // (abbreviated for brevity)
  store <2 x float> %col.load24, ptr %18, align 4
  %vec.gep27 = getelementptr float, ptr %18, i64 4
  store <2 x float> %col.load26, ptr %vec.gep27, align 4
  ret void
}

; Function Attrs: nocallback nofree nosync nounwind willreturn memory(argmem: read)
declare <4 x float> @llvm.matrix.column.major.load.v4f32.i64(ptr nocapture, i64, i1
    immarg, i32 immarg, i32 immarg) #1

; Function Attrs: nocallback nofree nosync nounwind speculatable willreturn
    memory(none)
declare <4 x float> @llvm.matrix.multiply.v4f32.v4f32.v4f32(<4 x float>, <4 x float>,
    i32 immarg, i32 immarg, i32 immarg) #2

; Function Attrs: nocallback nofree nosync nounwind willreturn memory(argmem: write)
declare void @llvm.matrix.column.major.store.v4f32.i64(<4 x float>, ptr nocapture
    writeonly, i64, i1 immarg, i32 immarg, i32 immarg) #3
```

The LLVM backend would match the matrix Intrinsics (`@llvm.matrix.*`) to call the specific code generation pipeline. For instance, `@llvm.matrix.column.major.load` Intrinsic would call the `LowerColumnMajorLoad` function in the LLVM backend, which will directly generate the IR for loading matrices in the column major format; the implementation can be found in `llvm/lib/Transforms/Scalar/LowerMartixIntrinsics.cpp`.

Our idea for sparse matrix support was to implement it as an LLVM Intrinsics; given dense matrix support was implemented as Intrinsics, we assumed it would be a natural place for sparse matrix support as well. With this rough pipeline in mind, we now move onto describing how sparse matrices are organized in memory so that `loads` and `store` can be called, along with sparse linear algebra operations.

2.2 Sparse Matrices in Memory

Recall the **Compressed Sparse Column** (CSC) format, which stores the only the nonzero elements of a matrix, along with its indices and pointers:

$$A = \begin{pmatrix} a_{0,0} & 0 & 0 & a_{0,3} \\ 0 & 0 & 0 & 0 \\ a_{2,0} & 0 & 0 & 0 \end{pmatrix}$$

`pointers[1]:`	0	2	2	3
`indices[1]:`	0	3	0	
`values:`	$a_{0,0}$	$a_{0,3}$	$a_{2,0}$	

Figure 4: The CSC format

LLVM Intrinsic functions are limited to return up to one return value that is a first-class primitive type, which includes pointers, vectors, and numeric values. Since the CSC matrix is represented with three vector-like primitives, this meant that we needed to compress this representation into one first-class primitive type somehow from the perspective of the compiler front end. Since LLVM matrices handle this problem by storing 2D matrices as a 1D flat vector in column major/row major form, we decided to follow a similar path and decided to flatten the three vectors into one by concatenating them one after another. That is, the CSC matrix in Figure 4 would be arranged in memory as

$a_{0,0}$	$a_{0,3}$	$a_{2,0}$	0	3	0	0	2	2	3

That is the `load` Intrinsics would take pointers to the `pointers`, `indices` and `values` array, concatenate them as above, and return a pointer to the head of the concatenated array. Since we need to be able to recover the three separate arrays, we also experimentally store the number of non-zeros (NNZ) as the first element in the concatenated array above:

3	$a_{0,0}$	$a_{0,3}$	$a_{2,0}$	0	3	0	0	2	2	3

with this, we know that the `values` array ends at the offset NNZ, and the `indices` arrays ends at the offset $2 \cdot \text{NNZ}$, and the remaining elements form the `pointers` array. The NNZ can also be recovered from the last element of the column pointers, but it is difficult to know how to access this element without user-provided boundaries since it, along with the row indices and values vectors, must be stored in one form or another as a flat 1D vector.

2.3 Implementation of Sparse Matrix Compilation

We initially started with implementing the Intrinsics operations for lowering loading, and storing sparse matrices to the LLVM IR: `LowerCSCLoad`, `LowerCSCStore`. This involves defining those functions in the Intrinsics table, found in `llvm/include/IR/Intrinsics.td`, and defining our sparse matrix module `LowerSparseMatrixIntrinsics`. This allows sparse matrix operations like `llvm.csc.matrix.load` to match to our lowering code and emit the corresponding IR.

We have also modified the `clang` `C` compiler to expose our sparse matrix LLVM Intrinsics, such that sparse matrix code can be called directly from `C` programs:

```c
#include <stdio.h>

typedef float mat __attribute__((matrix_type(2, 2)));

void main() {
    float pa[4] = {1.0, 0.0, 0.0, 1.0};
    mat a = __builtin_csc_matrix_load(pa, 2, 2, 2);
    __builtin_csc_matrix_store(r, pa, 4);
}
```

This means, if support for sparse matrices have been fully implemented as we envisioned, a compiler/programming language developer would be able to call our sparse matrix Intrinsics directly to lower their sparse linear algebra language constructs to the LLVM IR, or if they compile to the `C` programming language, then call the `__builtin_sparse_matrix_*` functions within our fork of the `clang` `C` compiler. Our implementation work describing can be found in our fork of the LLVM project, and accessing it is described in section 3.

2.4 Limitations of LLVM

The main issue we faced is that algorithms operating on sparse matrices are much more complex and requires more information than dense matrices equivalents, which makes it difficult to express at the LLVM IR level. This is likely also the reason that LLVM dense matrix support is much more primitive than higher level compilers and libraries — among the LLVM dense matrix intrinsics, there is only support for load, store, and matrix multiplication, and its intended use is limited to very small matrices [10]. In comparison, the MLIR Linalg dialect is much more extensive [11].

We hit a crucial roadblock towards LLVM sparse matrix support while attempting sparse matrix multiplication (SpGEMM). Our plan was to multiply in a column-by-column fashion, which is preferred for the CSC storage format. In this method, each column c_j of the resultant matrix C is computed by aggregating the intermediate multiplication results of each non-zero entry b_kj in b_j with the corresponding column a_k [12]. A prototype implementation is available in our Python repository, which acted as test bed for our implementation before we tried to implement in LLVM.

$$\boldsymbol{c}_{*j} = \sum_{k \in I_j(\boldsymbol{B})} \boldsymbol{a}_{*k} * \boldsymbol{b}_{kj}, j = 1, 2, ..., \boldsymbol{r}.$$

Figure 5: Column-by-column method [12]

However, we began running into unanticipated issues when implementing in LLVM. In CSC format, the NNZ of a sparse matrix is the dominant memory footprint, yet the sparsity of the resultant matrix cannot be known in advance. Thus, the first issue is not being able to appropriately allocate the size of the result sparse matrix. In contrast, the memory footprint of the result for dense matrix multiplication is immediately apparent from the dimensions of the input matrices. Since LLVM dense matrix support expects the user to provide this information at compile time, the dimensions of the resultant matrix multiplication is also always known at compile time. In practice, there is no similar reasonable information that the user may provide aside from the number of non-zero elements (NNZ) for us to appropriate allocate a resultant sparse matrix. However, this knowledge cannot be known without the actual computation, unlike shape information with respect to dense matrices.

We explored how existing SpGEMM (Sparse General Matrix-Matrix Multiplication) algorithms address this issue and reviewed relevant literature, including a systematic review by Gao et al. [12], which identified four methods:

- **Precise Prediction**: This method involves a two-path system where the exact NNZ for each row or column of the output matrix is calculated based on the input sparse matrices, followed by the numeric computation of actual values.

- **Probabilistic Method**: This approach transforms the problem of estimating NNZ in the output matrix into estimating the size of reachability sets in a directed graph, using a Monte Carlo-based algorithm.
- **Upper-Bound Prediction**: This method calculates an upper bound for NNZ by counting the non-zero elements in the corresponding rows of matrix B for each non-zero entry in matrix A.
- **Progressive Method**: This method dynamically allocates memory as needed based on ongoing calculations.

The primary challenge is that all of the propose solutions needs to access the values within the CSC vectors either immediately or eventually. Even with the progressive method (which is what our Python implementation implicitly used), we will need to eventually perform the actual computation and result accumulation process (we had planned to use a variation of Gustavson's algorithm for Column-By-Column multiplication [13]), which is difficult at our level of abstraction. To conduct any computation, it's essential to navigate the column pointer array, then the row indices array, and finally the values array. We underestimated the complexity of performing these operations in LLVM. For instance, implementing column-by-column multiplication requires iterating over the column pointers array and reading each element, a process that involves pinpointing the correct element pointer via `getelementptr` and loading its value in LLVM. This operation must then be repeated for accessing data from the row indices and values arrays.

Such methods are highly prone to errors due to the reliance on pointer arithmetic at compile time and at the instruction level. In contrast, dense matrix multiplication in LLVM avoids these complications as it can simply iterate based on the shape information of the matrices; knowing the actual values of the matrices is not necessary for the computation. The challenges are further intensified by our need to interpret the CSC matrix from a flat 1D representation, which introduces an additional layer of arithmetic indirection. Although pointer arithmetic is also necessary to convert a flat 1D vector into a series of column vectors for dense matrices in LLVM, the access pattern remains consistent each time (e.g., the number of rows for column-major order) and is provided by the user, unlike in our scenario where it needs to be dynamically computed.

2.5 Why it worked for MLIR

With our attempt at implementing built-in sparse matrix support being unsuccessful due to the challenging restrictions of LLVM, we then pivoted our research question to *how the design and structure of MLIR allowed for the implementation of built-in sparse linear algebra.*

The main takeaway that enables MLIR's support for sparse tensors is that the sparse tensor dialect operates at a much higher level of abstraction compared to LLVM IR, making it significantly more expressive. For example, the limitation of returning a flat 1D vector is not an issue. The MLIR infrastructure is designed to be extensible for domain-specific compilers and thus boasts the ability to define new operations easily. Consequently, it can return sparse tensors as first-class types, unlike the requirement to conform to existing primitives in the LLVM system. Let's discuss this in more detail by examining the implementation of SpGEMM in the sparse tensor dialect between two CSR matrices (CSR stands for Compressed Sparse Row, which is the row-focused equivalent of CSC), as demonstrated in the work by Aart et al [9]. The following demonstrates matrix multiplication between two sparse matrices:

```
%C = linalg.matmul ins(%A, %B: tensor<?x?xf64, #CSR>, tensor<?x?xf64, #CSR>) ->
    tensor<?x?xf64, #CSR>
```

is lowered to the following, in simplified form:

```
scf.for %i = %c0 to %m step %c1 {
    %c_i_values, %filled, %added, %count = sparse_tensor.expand %C
    ...
    scf.for %kk = %a_lo to %a_hi step %c1 {
        %k = memref.load %a_indices[%kk] : memref<?xindex>
        %aik = memref.load %a_values[%kk] : memref<?xf64>
        ...
        scf.for %jj = %b_lo to %b_hi step %c1 {
        %j = memref.load %b_indices[%jj] : memref<?xindex>
        %bkj = memref.load %b_values[%jj] : memref<?xf64>
        %cij = memref.load %c_i_values[%j] : memref<?xf64>
        %0 = arith.mulf %aik, %bkj : f64
        %1 = arith.addf %cij, %0 : f64
        scf.if (not %filled[%j]) {
            memref.store %true, %filled[%j] : memref<?xi1>
            record insertion as %added[ %count++ ] = %j
        }
        memref.store %1, %c_i_values[%j] : memref<?xf64>
        }
    }
    sparse_tensor.compress %C, %c_i_values, %filled, %added, %count
}
```

We observe that they adopt a form of Gustavson's algorithm in the helper vectors returned by `sparse_tensor.expand` and `sparse_tensor.compress`, respectively. More importantly, however, we note that the necessary pointer arithmetic for indexing the pointers, indices, and values arrays is much simpler to execute. The `memref.load` operation handles the more cumbersome indexing and load sequence, which is difficult to manage succinctly at the instruction level. Furthermore, by utilizing the SCF dialect, iteration over the helper arrays can be expressed extremely concisely (e.g., `scf.for`) using values stored in the tensor types, for which there is no parallel in LLVM. The main advantage is that MLIR's implementation abstracts to a higher level than is necessary at the LLVM IR level. The memref and SCF dialects can then be progressively lowered to a low-level target like LLVM. This progressive lowering capability allows MLIR developers to express the intricacies of SpGEMM without worrying about instruction-level code generation.

3 Summary of Artifacts

Non-LLVM changes can be found in the `/6156` directory of our LLVM fork, linked below:

https://github.com/sparse-matrix-llvm/sparse-matrix.

3.1 Python Prototype of `LowerSparseMatrixIntrinsics`

A Python prototype of sparse linear algebra, containing our experimentation on loading and storing CSC matrices, as well as performing sparse matrix multiplication, can be found in the linked repository: https://github.com/sparse-matrix-llvm/sparse-matrix-python

3.2 Our Attempts of Sparse Matrix Support in LLVM

Our attempts to implement built-in sparse matrix support in LLVM can be found in our fork of the LLVM Project, linked here: https://github.com/sparse-matrix-llvm/sparse-matrix.

The HEAD of our `main` branch will include implementation of LLVM Intrinsics for loading and parsing sparse matrices in the CSC format, as well as changes in the `clang` `C` compiler expose those LLVM Intrinsics via `clang` `__builtin`'s.

Test programs can be found in the `sparse-test` directory of the project root.

To run the tests, the fork of our LLVM project must first be built. The following commands have worked for us:

```
$ cd ./build
$ cmake -DCMAKE_C_COMPILER=clang \
       -DCMAKE_CXX_COMPILER=clang++ \
       -DCMAKE_BUILD_TYPE=Debug \
       -DLLVM_ENABLE_PROJECTS=mlir \
       -GNinja ../llvm
$ ninja -j<N_THREADS>
```

The commands above will build LLVM and Clang. Using the built binaries, we emit the LLVM IR of test `C` programs via:

```
$ <LLVM_ROOT>/build/bin/clang -S -fenable-matrix -emit-llvm <FILE>.c -o <FILE>.ll
```

We also designed a test for our `store` feature, which may be run via

```
$ <LLVM_ROOT>/build/bin/llvm-lit
    llvm/test/Transforms/LowerSparseIntrinsics/sparse-store-double.ll
```

For WIP code that depicts some of the details of our attempt in implementing more complex sparse matrix operations, please defer to the branches on our GitHub repository that is not `main`.

4 Conclusion

4.1 Answers to the Research Questions

RQ1: How can first-class support for sparse matrices be implemented inside LLVM with more lower-level abstractions than those available in MLIR?

We concluded that the abstractions in the LLVM are not expressive enough to implement built-in support for sparse matrices. Our roadblocks in the LLVM were detailed in section 2.4.

RQ2: What abstractions are needed to implement code generation for sparse matrices?

We concluded that we need more expressive abstractions for control flow over run-time variables, as well as dynamic memory allocation. We wrote about why these abstractions are necessary and how sparse matrix support can be implemented with them through a case study of MLIR, in section 2.5

RQ3: Does first-class support for sparse matrices in LLVM ease the implementation of adopting sparse matrices in the front-end language?

We were not able to sufficiently reach a conclusion on this research question. However, based on our experimentation with LLVM's built-in dense matrix support (which inspired this project), we believe that even this support is not appropriately expressive for ease of incorporation in higher-level languages. Namely, too much information (such as matrix dimensions) than what can reasonably be expected for the programmer to provide is required at compile time, meaning complex chains of linear algebra operations will not be feasible to be expressed cleanly without extensive `typdefs` or equivalent. We believe the same problems will arise with sparse matrix support, if it was possible to be implemented.

On the other hand, MLIR's higher-level abstractions will certainly allow for easy adoption of sparse matrices in higher-level languages. As detailed in section 1.2, if a language targeting the MLIR uses the `linalg` and `tensor` (both of which are widely used among languages targetting the MLIR), very minimal changes are needed to support sparse tensor algebra.

4.2 Lessons Learned and Individual Contributions

Eric

From this project, I gained insights into the steps required to add new LLVM Intrinsics and the implementation of dense matrix operations within LLVM, including its strengths (such as highly optimized matrix multiplication with loop optimizations like tiling and fusion) and limitations (for instance, size constraints and the need for pre-defined shape information from the user). Additionally, having had no prior background in sparse matrices, their storage formats, or the algorithms for their transformations, I learned about the trade-offs among various sparse storage formats when deciding which one to use, as well as algorithms like SpGEMM in relation to Gustavson's algorithm. Finally, I learned about certain implementation details of existing sparse compilers in from our literature review. Although our project underscored the challenges of implementing first-class sparse matrix support in LLVM due to the unanticipated expressing the algorithmic complexity of sparse matrix operations at the level of LLVM IR, I would not have been able to anticipate these issues without getting our hands dirty and learning about the intricacies of the LLVM ecosystem first hand. In terms of the code, the parts I contributed include the Python prototype of the sparse matrix, the introduction of the LLVM intrinsics infrastructure for sparse matrices as well as the store intrinsic.

Chris

My individual contributions include

- exposing our sparse matrix LLVM Intrinsics to the `clang C` compiler as `clang __builtin__` functions, and testing the front-end pipeline for invoking them
- prototyping of loading CSC matrices to memory in the LLVM Intrinsics
- discovery work on MLIR's approach to sparse matrix support (loading, storing, matrix multiplication).

It would have been great if our sparse matrix support in LLVM worked out as imagined, but we were not able to foresee the limitations of LLVM until we really dug into prototyping our project. Perhaps with a deeper understanding of the LLVM infrastructure, we would have been able to identify these issues earlier.

Despite that, my biggest takeaway was having the change to dig into the LLVM and MLIR codebase, and understanding first-hand their strengths and limitations. Each dialect in the MLIR having their own lowering passes made if difficult to trace how a program gets eventually lowered into the lowest-level IR, but the complexity here enables enough expressivity to sparse linear algebra more cleanly, as a compiler feature. While I found adding new features to LLVM and exposing them to the front-end to be significantly easier than in MLIR, this simplicity takes away the expressivity to cleanly implement the sparse matrix support that we wanted. Both the LLVM and MLIR were incredibly complex — even a trivial piece of program will hit large code paths until it gets lowered. In the future, I hope to deepen my understanding of the moving parts in the LLVM and MLIR infrastructures!

References

[1] SCOTT JENNIFER TUMA MIROSLAV. *Algorithms for sparse linear systems.* BIRKHAUSER VERLAG AG, 2023.

[2] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17 (3):261–272, 2020.

[3] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM journal on matrix analysis and applications*, 13(1):333–356, 1992.

[4] Intel. Intel math kernel library. 2024. URL `https://software.intel.com/en-us/mkl`.

[5] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, jun 2002. ISSN 0098-3500. doi: 10.1145/567806.567810. URL `https://doi.org/10.1145/567806.567810`.

[6] A. J. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, Department of Computer Science, Leiden University, 1996.

[7] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):1–29, 2017.

[8] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[9] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in mlir. *ACM Trans. Archit. Code Optim.*, 19(4), sep 2022. ISSN 1544-3566. doi: 10.1145/3544559. URL `https://doi.org/10.1145/3544559`.

[10] Adam Nemet. Llvm matrix support rfc. 2018. URL `https://groups.google.com/g/llvm-dev/c/wsN3X4tBrxE/m/BZF5smRzDQAJ`.

[11] MLIR Community. Mlir 'linalg' dialect. 2020. URL `https://mlir.llvm.org/docs/Dialects/Linalg/`.

[12] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. A systematic survey of general sparse matrix-matrix multiplication. *ACM Computing Surveys*, 55(12):1–36, 2023.

[13] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.