MOM: Matrix Operations in MLIR

do we want to add a "Towards" ?

Compiler Support for Linear Algebra Computations in MLIR

Anonymous Author(s)

Abstract

Modern research in code generators for dense linear algebra computation has shown the ability to produce optimized code with a performance which compares and often exceeds the one of state-of-the-art implementations by domain experts. However, the underlying infrastructure is often developed in isolation making the interconnection of logically combinable systems complicated if not impossible. In this paper, we propose to leverage MLIR as a unifying compiler infrastructure for the optimization of dense linear algebra operations. We propose a new MLIR dialect for expressing linear algebraic computations including matrix properties to enable high-level algorithmic transformations. The integration of this new dialect in MLIR enables end-to-end compilation of matrix computations via conversion to existing lower-level dialects already provided by the framework.

1 Introduction

A significant part of processor time is spent on mathematical algorithms used in simulations, machine learning, communication, signal processing, computer vision, and other domains. Of course, the mathematics used in these domains may differ widely. Still, the actual computations often fall into the realm of linear algebra, meaning sequences of computations on matrices and vectors. Research in the area of linear algebraic domain-specific languages (DSLs) have demonstrated that expert-level optimizations can be carried out automatically when taking the mathematical semantics of the computation into account (e.g., [1, 4, 5]). Over time, however, the code generation infrastructure supporting such research has grown into a fragmented software ecosystem, where different components that could logically cooperate towards a shared performance goal are in practice unable to interoperate with one another.

This paper proposes an implementation of a subset of the state-of-the-art linear algebra code generator Linnea [1] in MLIR. Specifically, our contributions are:

- An IR representation to model linear algebra operations, types and properties.
- A lowering from such representation to LLVM IR.

2 The Linnea Dialect

We introduce a new linear algebraic dialect in MLIR called Linnea [1]. Linnea provides MLIR attributes, types, operations, and transformations required to make dense computation as first-class citizens in the compiler IR. Linnea bridges dense linear algebra computations expressed in our Python DSL and the Linalg dialect as shwon in Fig. 1. From Linalg we lower to LLVM IR and then machine code.

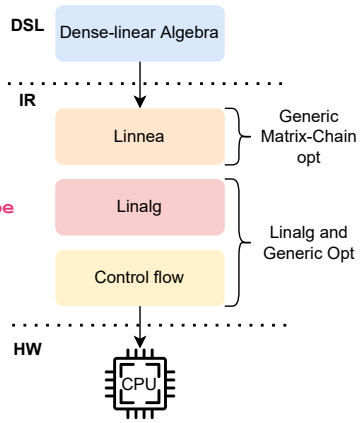

Figure 1. The MOM compiler for dense-linear algebra.

Matrix Attribute. Inspired by the Sparse Tensor Dialect, we introduce `Matrix` attributes to encode non-conflicting, compile-time properties of a matrix. For example, the property that a matrix is lower triangular is expressed using the following attribute:

```
#linnea.property<['lowerTri']>
```

During bufferization [1], the underneath algebraic object is materialized using a straightforward `memref` where attributes dictates how the memory is filled (i.e., set to zeros the element below the diagonal for an upper triangular matrix). In the future, we would like to depart from `memref` and use more appropriate containers.

Types: Matrix, Term and Identity. Linnea provides three built-in types: Matrix, Term, and Identity. Matrix represents a 2-dimensional tensor type which accepts an attribute that describes a set of non-conflicting properties, a static dimension, and an MLIR built-in type that describes each stored element's type (i.e., f32). For example, the following is the type declaration of a 5×5 lower-triangular matrix:

```
!linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
```

A term type represents the result of Linnea `equationOp`. It is a placeholder type, meaning that it will be replaced by a concrete one (i.e., `MatrixType`). In fact, only after optimization and simplification the result of an `equationOp` is known and thus the placeholder types can be swapped with a concrete one. Finally, an Identity type represents the identity matrix –

[1] bufferization materializes memref (i.e., malloc) from tensors.

Op name	Description
Init	Materialize the algebraic object using a `memref`
Fill	Initialize the algebraic object according to its properties
Equation	Represents a linear algebra equation
Add	Variadic addition of different algebraic objects
Yield	Return the result of an Equation
Mul	Variadic multiplication of different algebraic objects
Transpose	Transpose the algebraic object

Table 1. Operations exposed by the Linnea dialect.

a square matrix where all the elements on the diagonal are 1. The Identity type enables the expression of mathematical identities such as $A \cdot I = A$ and $I \cdot I = I$.

Table 1

Operations. Table shows the operations exposed by the Linnea dialect 1. `Init` and `Fill` initialize and fill an algebraic object, respectively. `Mul` and `Add` are variadic operations that takes as input an arbitrary numbers of Values of Linnea types. The `Equation` represents a "bag" of Linnea operations that together represent a given mathematical expression. We build a symbolic representation of the program by walking each equation starting from the yield operation. An equation gets simplified and rematerialized to new low-level Linnea IR (not shown). For example, variadic multiplications are rewritten as binary multiplications with an optimal parenthesization.

take

Transformations. Currently, we limit ourselves to three transformations: matrix-chain reordering, identity simplification and properties propagation. Matrix chain reordering is an algorithmic improvement that minimizes the number of scalar multiplications when multiplying a chain of matrices. We implement the algorithm described in [3] and generalize it to account for matrix properties as in [1]. Identity simplification consists of exploiting the identity matrix to simplify the computation. For example, $A \cdot I \rightarrow A$. Finally, Linnea's IR makes it possible to annotate matrices with properties. But, it is also essential to understand the properties of intermediate results as the computation unfold, thus we introduce a symbolic engine and encode a set of inference rules such as $lowerTriangular(A) \rightarrow upperTriangular(A^T)$. We use the symbolic engine to replace a `TermType` with a concrete type.

3 MOM Compiler Usage

MOM provides users with a convenient Python DSL language to express their computation. It does not require any knowledge about the internal intermediate representation or the different compiler passes involved in lowering a Python specification to binary. For example, Listing 1 shows how a user can specify a multiplication between two lower triangular matrices.

```
1  n = 5
2  m = 5
3  Matrix A(n, m) <LowerTriangular>
4  Matrix B(n, m) <LowerTrinagular>
5  Matrix C(n, m) <>
6  C = A * B
7  print(C)
```

Listing 1. MOM specification for a triangular matrix multiplication.

```
// Initialize matrix A. Default initialize to 1 (%fc = 1).
%A = linnea.init [5, 5] :
  !linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
%Af = linnea.fill(%fc, %A) :
  f32, !linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
// Initialize matrix B.
%B = linnea.init [5, 5] :
  !linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
%Bf = linnea.fill(%fc, %B) :
  f32, !linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
// Multiply the two and print the result.
%0 = linnea.equation {
  %1 = linnea.mul %Af, %Bf :
    !linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
    !linnea.matrix#linnea.property<['lowerTri', [5, 5], f32]>
    -> !linnea.term
  linnea.yield %1 : !linnea.term
}
linnea.print %0 : !linnea.term
```

Listing 2. A Linnea IR representation for a multiplication between two lower triangular matrices.

Behind the scene, MOM lowers the Python specification to the Linnea dialect (see Listing 2). Line 3 and 4 map to the initialization for the A and B matrix. Line 6 maps to the `linnea.equation` operation.

4 Results

The experiments have been conducted on an Intel Core i7-10750H. All results were obtained considering the minimal execution time of five independent runs for single-precision (`i32`) operands as in [2]. For a chain of 4 matrices $800 \times 1100 \times 900 \times 1200 \times 100$ with initial parenthesization $(((A_1 \times A_2) \times A_3) \times A_4)$ we obtain a $1.24X$ speedup by re-parenthesize as $(A_1 \times (A_2 \times (A_3 \times A_4)))$.

by re-parenthesizing

5 Conclusion

We presented MOM, an end-to-end flow for dense linear algebra operations based on MLIR. We proposed a new linear algebraic dialect for capturing important matrix properties and enabling the expression of domain-expert optimization strategies. Future work includes the integration of more properties and algebraic transformations, as well as the investigation of suitable extensions of the existing infrastructure to support them (e.g., the multiplication of two triangular matrices could map to a non-rectangular iteration space currently not supperted by Linalg).

cite Henrik's "Generalized Matrix Chain" paper: https://dl.acm.org/doi/10.1145/3168804

References

[1] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. 2021. Linnea: Automatic Generation of Efficient Linear Algebra Programs. *ACM Transactions on Mathematical Software (TOMS)* 47, 3 (June 2021), 1–26. https://arxiv.org/pdf/1912.12924.pdf

[2] Lorenzo Chelini, Andi Drebes, Alex Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive Raising in Multi-level IR. In *International Conference on Code Generation and Optimization (CGO)*. February 27th - March 3rd, 2021, Virtual Conference.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[4] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[5] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-Scale Linear Algebra Applications. In *International Symposium on Code Generation and Optimization (CGO)*. 327–339.