Using MLIR to Optimize Basic Linear Algebraic Subprograms

Steven Varoumas
Huawei Technologies Research & Development (UK)
Cambridge Research Centre – Compiler Lab
Writing libraries is a time-consuming task:

- Many man-hours spent fine-tuning code to achieve best performance.
- Has to be adapted and optimized for any new hardware.

Can we give compilers the task of optimizing libraries that can compete with hand-written ones?

In this work, we intend to generate an optimized math library using compiler technologies.

- Aim to support the Basic Linear Algebra Subprograms (BLAS) specification.
- Reduce time taken optimizing/fine-tuning math functions.
- Automatize creation of hardware-specific code.
- Leverage the functionalities and extensibility of the MLIR framework.

**Objective:** Explore what performance results we can get from this approach (expectation: reach 90% of the performance of an in-house hand-tuned BLAS library).
Context: KunpengBLAS library

> BLAS: specification that defines a set of linear algebra functions (e.g. dot product, matrix multiplication).
> Reference implementation of BLAS: KunpengBLAS (“KPL”) library (we use the single-thread version).
> Hardware for measurements: Huawei Kunpeng 920 (64bits ARMv8-based processor).
> We particularly focus on GEMM (General Matrix-Matrix multiplication): performance critical.

→ GEMM is \( C = \alpha AB + \beta C \) (A, B and C are matrices, \( \alpha \) and \( \beta \) are scalars)
→ KPL is able to reach >90% of the theoretical peak of the hardware for sgemm/dgemm:

\[ \begin{align*}
&\text{sgemm (single precision gemm)} \\
&M=N=K=16 \\
&A: M \times K | B: K \times N | C: M \times N
\end{align*} \]

\[ \begin{align*}
&\text{dgemm (double precision gemm)} \\
&M=N=K=16 \\
&A: M \times K | B: K \times N | C: M \times N
\end{align*} \]
Context: GEMM Core Transformations

We rely on the following core transformations:

- **Tiling** – Apply the operation on subsets (tiles) of the matrices.
- **Packing** – Re-mapping data in the A and B tiles to get sequential memory accesses.

This follows the work of Goto & Van De Geijn [2] to compile an efficient GEMM. Their use in an MLIR pipeline has been described by Bondhugula [1].

```plaintext
for j = 0 to N-1 by steps of n_c:
  for p = 0 to K-1 by steps of k_c:
    Bc = B(p:p+kc-1, j:j+nc-1) // Pack into Bc
  for i = 0 to M-1 by steps of m_c:
    Ac = A(i:i+m-1, p:p+k-1) // Pack into Ac
  for jj = 0 to nc-1 by steps of n_r:
    for ii = 0 to mc-1 by steps of m_r:
      for pp = 0 to kc-1 by steps of 1:
        // Microkernel
        C(ii:ii+mr-1, jj:jj+nr-1) += Ac(ii,ii+mr-1,pp) * Bc(pp,jj:jj+nr-1)
```

**Optimized matrix multiplication (pseudocode)**

Project Overview: Compilation Pipeline

**Full pipeline** to generate/optimize/compile BLAS functions:

> A high-level definition of the function is generated directly in the linalg dialect (does *not* come from a frontend... *yet*).

> The generated file is given to an optimizing MLIR compiler (**mlirc**), with a list of transformations to apply and their arguments. The optimized functions are packaged into a library (**libblas_mlir.so**).

---

```mlir
func.func @gemm(%A: tensor<?x?xf32>, %B: tensor<?x?xf32>, %C: tensor<?x?xf32>) -> tensor<?x?xf32> {
  %res = linalg.generic ins(%A, %B : tensor<?x?xf32>, tensor<?x?xf32>) outs(%C : tensor<?x?xf32>) {
    ^bb0(%a: f32, %b: f32, %c: f32):
      %m = arith.mulf %a, %b : f32
      %a = arith.addf %out, %m : f32
      linalg.yield %m : f32
  } -> tensor<?x?xf32>
  return %res : tensor<?x?xf32>
}
```

*actual gemm is $C = \alpha AB + \beta C$*
Multi-Kernel Approach

Transformations may depend on the specific inputs of the function: one set of transformations/parameters is not always good for all possible inputs. For example, packing is not always helpful for small matrices [1].

→ We use a multi-kernel approach to enhance each function’s performance:
  > For each BLAS function (e.g. gemm), we generate a set of kernels.
  > Kernels are optimized variants of the function, tuned for specific inputs.
  > At runtime, a kernel selector chooses the “best” kernel, based on dynamic information.

Multi-Kernel Approach: Example (axpy)

Using a different kernel for small input vectors and large input vectors gives results consistently >90% of the baseline (KPL) for saxpy (single-precision axpy):

\[ \text{axpy is } \hat{y} = \alpha \hat{x} + \hat{y} \] (scalar multiplication + vector addition)

![Graph showing performance comparison between different kernels]

- **Small Kernel / Baseline**
- **Large Kernel / Baseline**
- **Multi Kernel / Baseline**

length of input vectors
Several optimizations have been implemented at various levels of the pipeline in order to increase performance/functionalities, such as:

**High-level optimizations at linalg level:**

- Dimensions of A: MxK, dimensions of B: KxN, dimensions of C: MxN
- When N<M: reordering $C = \alpha AB + \beta C$ into $C = A(\alpha B) + \beta C$ can improve performance:

![Graph showing performance improvement with different input sizes](image)
Support for extensions of BLAS and new transformations:
Example: supporting mixed-precision GEMM (i.e. element types of A, B and C can differ).

> Easily enabled in MLIR by injecting truncation/extension ops in the MLIR `linalg.generic` definition.
> Building on a similar transform for transpose operations, we hoist casting ops into the packing loops of the corresponding matrix:

```plaintext
for j = 0 to N-1 by steps of nc:
  for p = 0 to K-1 by steps of kc:
    Bc = B(p:p+kc-1, j:j+nc-1) // Pack into Bc
  for i = 0 to M-1 by steps of mc:
    Ac = A(i:i+m-1,p:p+k-1) // Pack into Ac
      for jj = 0 to nc-1 by steps of nr:
        for ii = 0 to mc-1 by steps of mr:
          for pp = 0 to kc-1 by steps of 1:
            // Microkernel
            Ac' = cast(Ac(ii,ii+mr-1,pp)): Ta into Tc
            Bc' = cast(Bc(pp,jj+jj+nr-1)): Tb into Tc
            C(ii:ii+mr-1, jj:jj+nr-1) += Ac' * Bc'
```

Optimisations (2)
Optimisations (3)

Optimisations of MLIR code:
Example: hoisting of `vector.reduce` outside of loops:

```mlir
%x = (...): f32
%loop = scf.for %i = %lb to %ub step %step iter_args(%arg = %x) -> f32 {
  %v1 = (...): vector<32xf32>
  %v2 = (...): vector<32xf32>
  %m = arith.mulf %v1, %v2 : vector<32xf32>
  %r = vector.reduce <add>, %m : vector<32xf32> into f32
  %a = arith.addf %r, %arg : f32
  scf.yield %a : f32
}
```

→ This also applies when the accumulator is a vector (using `vector.multi_reduction`)

```mlir
%x = (...): f32
%zerovec = arith.constant dense<0.000000e+00> : vector<32xf32>
%loop = scf.for %i = %lb to %ub step %step iter_args(%arg = %zerovec) -> vector<32xf32> {
  %v1 = (...): vector<32xf32>
  %v2 = (...): vector<32xf32>
  %m = arith.mulf %v1, %v2 : vector<32xf32>
  %a = arith.addf %m, %arg : vector<32xf32>
  scf.yield %a : vector<32xf32>
}
%r = vector.reduce <add>, %loop, %x : vector<32xf32> into f32
```
Optimisations (3 – cont.)

Optimisations of MLIR code:
Example: hoisting of `vector.reduction` outside of loops:

→ This optimisation has a significant impact on `gemv`* (`general matrix-vector` multiplication):

\[ \text{gemv is } \tilde{y} = \alpha A \tilde{x} + \beta \tilde{y} \]
Handling of Complex Type

To cover the BLAS API, we need to provide operations on complex inputs \( (\text{cgemv}, \text{cgemm}, \text{zgemm}, \ldots) \)

> High-level definition of the ops in \( \text{linalg} \) is straightforward: inputs with a \( \text{complex}<t> \) element type.

> However, compiling these operations into efficient code poses some problem:
  - Complex tensors are \textit{not} vectorized.
  - The complex dialect lowers to extraction functions (\texttt{complex.im}, \texttt{complex.re}).
  - Our current hardware target supports some ARMv8.3-specific complex vector instructions (e.g. \texttt{fcmla} – complex multiply and add).
    → We would like to make use of them, instead of splitting complex values.

→ Existing conversion passes gave us \textit{less than 1\%} of KPL's performance for \texttt{cgemm}. 
Handling of Complex Type: Conversion into Real

We considered several options to handle complex operations:

> Transform complex GEMM into a series of real GEMM (cf. 3m and 4m methods for cgemm [1]).
  - Manual implementation and analysis did not show good performance.
  - Prevents use of complex-specific instructions (fcm1a).
  - Not easily extensible to other complex operations.

Handling of Complex Type: Conversion into Real

We considered several options to handle complex operations:

- Transform complex GEMM into a series of real GEMM (cf. 3m and 4m methods for cgemm [1]).
  - Manual implementation and analysis did not show good performance.
  - Prevents use of complex-specific instructions (fcmla).
  - Not easily extensible to other complex operations.

> Our solution/suggestion (WIP!):
  - Support vectorization into vectors of complex\langle t\rangle.
  - Type conversion of complex ranked types into “doubled” ranked types:

\[
\text{vector}\langle M\times N\times \text{complex}\langle t\rangle \rangle \rightarrow \text{vector}\langle M\times N\times 2\times t \rangle
\]

- Extend \text{vector.contraction/outerproduct} with \text{kind=\langle complexadd\rangle}.
- Enable lowering to fcmla in the backend by creating a new fcmu1add intrinsic.

\[\triangleright \ D148068 \ [AArch64] \ Lower \ fused \ complex \ multiply-add \ intrinsic \ to \ AArch64::FCMA \ (llvm.org)\]

Handling of Complex Type: Conversion into Real

Vector operations are updated accordingly:

```%
%cst = complex.constant [0.000000e+00 : f32, 0.000000e+00 : f32] : complex<f32>
%v = vector.transfer_read %t[%, %c0, %c0], %cst : tensor<?x1xcomplex<f32>>, vector<8x1xcomplex<f32>>
%vt = vector.transpose %v, [1, 0] : vector<8x1xcomplex<f32>> to vector<1x8xcomplex<f32>>
%t3 = vector.transfer_write %vt, %t2[%x, %y, %c0, %c0] : vector<1x8xcomplex<f32>>, tensor<?x?x1x8xcomplex<f32>>
```
Handling of Complex Type: Conversion into Real

Contraction is done with last two dimensions “flattened”:

\[
\text{vector}\langle M \times N \times 2 \times t \rangle \rightarrow \text{vector}\langle M \times 2N \times t \rangle
\]

→ Prevents splitting between real and imaginary values when lowering vectors.
→ Adapted to the input expected by ARMv8.3 \text{fcmla}: interleaved real and imaginary parts.

\[
\%v = \text{vector.contract}\ \{\ldots\}, \text{kind} = \#\text{vector.kind}\langle\text{complexadd}\rangle\ \%a, \%b, \%c : \text{vector}\langle 1 \times 8 \times \text{complex}\langle f32\rangle \rangle, \\
\text{vector}\langle 1 \times 4 \times \text{complex}\langle f32\rangle \rangle \ \text{into} \ \text{vector}\langle 8 \times 4 \times \text{complex}\langle f32\rangle \rangle
\]

\[
\%a1 = \text{vector.shape_cast}\ \%a : \text{vector}\langle 1 \times 8 \times 2 \times f32 \rangle \ \text{to} \ \text{vector}\langle 1 \times 16 \times f32 \rangle \\
\%b1 = \text{vector.shape_cast}\ \%b : \text{vector}\langle 1 \times 4 \times 2 \times f32 \rangle \ \text{to} \ \text{vector}\langle 1 \times 8 \times f32 \rangle \\
\%c1 = \text{vector.shape_cast}\ \%c : \text{vector}\langle 8 \times 4 \times 2 \times f32 \rangle \ \text{to} \ \text{vector}\langle 8 \times 8 \times f32 \rangle \\
\%v0 = \text{vector.contract}\ \{\ldots\}, \text{kind} = \#\text{vector.kind}\langle\text{complexadd}\rangle\ \%a1, \%b1, \%c1 : \text{vector}\langle 1 \times 16 \times f32 \rangle, \\
\text{vector}\langle 1 \times 8 \times f32 \rangle \ \text{into} \ \text{vector}\langle 8 \times 8 \times f32 \rangle \\
\%v = \text{vector.shape_cast}\ \%v0 : \text{vector}\langle 8 \times 8 \times f32 \rangle \ \text{to} \ \text{vector}\langle 8 \times 4 \times 2 \times f32 \rangle
\]
Optimisations for Complex Pipeline (1)

Hoisting of `vector.shape_cast` operations outside of loops:

```cpp
%loop = scf.for %i = %lb to %ub step %step iter_args(%arg = %v) -> (vector<4x4x2xf32>) {
  %c = vector.shape_cast %arg : vector<4x4x2xf32> to vector<4x8xf32>
  %w = (...) : vector<4x8xf32> // use of %c
  %r = vector.shape_cast %w : vector<4x8xf32> to vector<4x4x2xf32>
  scf.yield %r: vector<4x4x2xf32>
}

%c = vector.shape_cast %v : vector<4x4x2xf32> to vector<4x8xf32>
%loop0 = scf.for %i = %lb to %ub step %step iter_args(%arg = %c) -> (vector<4x8xf32>) {
  %w = (...) : vector<4x8xf32> // use of %c (unchanged)
  scf.yield %w: vector<4x8xf32>
}
%loop = vector.shape_cast %loop0 : vector<4x8xf32> to vector<4x4x2xf32>
```

→ This transformation moves `vector.shape_cast` operations out of the microkernel loop.
Optimisations for Complex Pipeline (2)

“Lifting” `vector.transfer_read+vector.shape_cast` to `tensor.collapse_shape+vector.transfer_read`:

\[
\begin{align*}
%0 &= \text{vector.transfer_read} \%\text{arg0}[%c0, %c0, %c0], \%\text{arg1} : \text{tensor}<1x4x2xf32>, \text{vector}<1x4x2xf32> \\
%1 &= \text{vector.shape_cast} \%0 : \text{vector}<1x4x2xf32> \text{ to } \text{vector}<1x8xf32> \\
%0 &= \text{tensor.collapse_shape} \%\text{arg0} [[0], [1, 2]] : \text{tensor}<1x4x2xf32> \text{ into } \text{tensor}<1x8xf32> \\
%1 &= \text{vector.transfer_read} \%0 [%c0, %c0], \%\text{arg1} : \text{tensor}<1x8xf32>, \text{vector}<1x8xf32>
\end{align*}
\]

> A similar transformation replaces `shape_cast+transfer_write` with `transfer_write+expand_shape`.

→ **Significant performance improvement** (+50%), as `tensor.collapse/expand_shape` does not involve data copy, unlike `vector.shape_cast`.  

![HUAWEI](https://example.com/huawei-logo.png)
Handling of Complex Types: Limitations

**Genericity:**
- Conversion assumes that the complex type layout fits with complex\(<t> \rightarrow 2xt.\)
- Heavily targeted towards specific hardware with specific instructions for complex type (fcmla).

**Interface changes:**
- A function taking in a vector<8xcomplex<f32>> now takes in vector<8x2xf32>.
  - We use special wrappers at the interface with the packager.
  - Working on extending the complex dialect with casting operations complex\(<t> \rightarrow 2xt \text{ and } 2xt \rightarrow \text{complex}<t>\).
Results: Performance vs KPL (Real GEMM)

Running `sgemm/dgemm` on Huawei Kunpeng 920, 1000 random points:

<table>
<thead>
<tr>
<th>Colour (value)</th>
<th>sgemm (single precision)</th>
<th>dgemm (double precision)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Red</strong> (0% - 49% of KPL)</td>
<td>None</td>
<td>0.3% of points</td>
</tr>
<tr>
<td><strong>Orange</strong> (50% - 89% of KPL)</td>
<td>5.6% of points</td>
<td>4.7% of points</td>
</tr>
<tr>
<td><strong>Green</strong> (90% - 99% of KPL)</td>
<td><strong>92.4% of points</strong></td>
<td><strong>95% of points</strong></td>
</tr>
<tr>
<td><strong>Blue</strong> (≥100% of KPL)</td>
<td>2% of points</td>
<td>None</td>
</tr>
</tbody>
</table>
Results: Performance vs KPL (Complex GEMM)

Running `cgemm/zgemm` on Huawei Kunpeng 920, 1000 random points:

<table>
<thead>
<tr>
<th>Colour (value)</th>
<th>cgemm (single precision)</th>
<th>zgemm (double precision)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Red</strong> (0% - 49% of KPL)</td>
<td>0.1% of points</td>
<td>None</td>
</tr>
<tr>
<td><strong>Orange</strong> (50% - 89% of KPL)</td>
<td>0.4% of points</td>
<td>1% of points</td>
</tr>
<tr>
<td><strong>Green</strong> (90% - 99% of KPL)</td>
<td>72.5% of points</td>
<td>18.2% of points</td>
</tr>
<tr>
<td><strong>Blue</strong> (≥100% of KPL)</td>
<td>27% of points</td>
<td>80.8% of points</td>
</tr>
</tbody>
</table>
Conclusion & Future Work

We have leveraged the functionalities of the MLIR framework to:

> Build a full pipeline to generate optimized functions of a BLAS library.
> Use a multi-kernel approach able to dynamically adapt to specific inputs.
> Provide optimizations to achieve results competing with hand-written assembly code.

Ongoing/future work:

> Connect to a DSL (ALP[1]) that would lower to MLIR and use our pipeline.
  → move beyond simply building a library
> Fuse operations to improve performance (some promising results for GEMM already).
> Enable parallelism for a multithread version of the library.
> Target more diverse hardware.

[1] Algebraic Programming @ https://algebraic-programming.github.io
Thank you.
Backup Slides
Running \texttt{sgemm/dgemm} on Huawei Kunpeng 920, 1000 random points:

<table>
<thead>
<tr>
<th>Colour (value)</th>
<th>sgemm</th>
<th>dgemm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Red</strong> (0% - 49% of OpenBLAS)</td>
<td>None</td>
<td>0.3% of points</td>
</tr>
<tr>
<td><strong>Orange</strong> (50% - 89% of OpenBLAS)</td>
<td>0.5% of points</td>
<td>7% of points</td>
</tr>
<tr>
<td><strong>Green</strong> (90% - 99% of OpenBLAS)</td>
<td>0.7% of points</td>
<td>72.4% of points</td>
</tr>
<tr>
<td><strong>Blue</strong> (100%-124% of OpenBLAS)</td>
<td>15.0% of points</td>
<td>20.3% of points</td>
</tr>
<tr>
<td><strong>Purple</strong> (≥125% of OpenBLAS)</td>
<td>83.8% of points</td>
<td>None</td>
</tr>
</tbody>
</table>
Results: Performance vs OpenBLAS (complex gemm)

Results of \texttt{cgemm/zgemm} (1000 random points):

<table>
<thead>
<tr>
<th>Colour (value)</th>
<th>\texttt{cgemm}</th>
<th>\texttt{zgemm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Orange} (50% - 89% of OpenBLAS)</td>
<td>None</td>
<td>0.3% of points</td>
</tr>
<tr>
<td>\textbf{Green} (90% - 99% of OpenBLAS)</td>
<td>None</td>
<td>0.1% of points</td>
</tr>
<tr>
<td>\textbf{Blue} (100% - 124% of OpenBLAS)</td>
<td>0.3% of points</td>
<td>1.7% of points</td>
</tr>
<tr>
<td>\textbf{Purple} (≥125% of OpenBLAS)</td>
<td>99.7% of points</td>
<td>97.9% of points</td>
</tr>
</tbody>
</table>
Results: gemv

92.1% of points >= 90% of KPL

88.7% of points >= 90% of KPL