MLIR
Compiler Construction for Heterogeneity

Cambium Seminar
Albert Cohen   albertcohen@google.com
(presenting the work of many)

January 24 2022
Personal Background

https://scholar.google.com/citations?user=_KMsPngAAAAJ
https://research.google/people/106208

- Parallelizing compilation
- Polyhedral compilation
- Compiler construction
- Machine learning applied to compiler construction
- Data-flow and synchronous programming languages
- Task-parallel programming languages
ML $\leftrightarrow$ Compilers
ML is: Data, Algorithms and Compute

Data drives the continuous improvement cycle for ML models.

Researchers provide new algorithmic innovations unlocking new techniques and models.

Compute allows it all to scale as datasets get larger and algorithms need to scale on that accordingly.
ML is: Data, Algorithms, Compile and Compute

Compilation is key to ML systems performance and portability

Tensor compilers in particular

Diversity and competing requirements from users, and hardware

ML is key to solving future compilation problems
Machine Learning
SW and HW

- TensorFlow
- JAX
- PyTorch
- CNTK
- mxnet
- ONNX

Edge TPU
TensorFlow
Intel Nervana
ONNX
None of this is scaling

Relief from Programming Languages?
Compiler Construction?
MLIR:
Scaling Compiler Infrastructure for Domain Specific Computation
CGO, March 1, 2021

Chris Lattner\textsuperscript{1,2}, Mehdi Amini\textsuperscript{1}, Uday Bondhugula\textsuperscript{1,3}, Albert Cohen\textsuperscript{1}, Andy Davis\textsuperscript{1}, Jacques Pienaar\textsuperscript{1}, River Riddle\textsuperscript{1}, Tatiana Shpeisman\textsuperscript{1}, Nicolas Vasilache\textsuperscript{1}, Oleksandr Zinenko\textsuperscript{1}

(and many more MLIR contributors)

\textsuperscript{1}Google Inc. \hspace{1em} \textsuperscript{2}Now at SiFive modular.ai \hspace{1em} \textsuperscript{3}Indian Institute of Science, Bangalore
A collection of modular and reusable software components that enables the progressive lowering of ML operations, to efficiently target hardware in a common way.

https://mlir.llvm.org
Why build the (N+1)-th compiler infrastructure?
LLVM: Industry Standard for Compiler Infrastructure

LLVM IR is not enough for high-level representations

There is a huge abstraction gap between ASTs and LLVM IR, covered in a one-shot conversion in Clang

Clang has a representation parallel to ASTs used in static analyzers, advanced diagnostics

LLVM IR is not enough for low-level representations

Multiple lower levels of abstraction introduced over time
Newer languages/compilers define custom intermediate representations between AST and LLVM IR for language-specific analyses and transformations.
Also Domain-Specific Languages...

Modern ML frameworks include domain-specific compilers

Yet there is no common infrastructure (and sometimes even understanding) to support this
How much code in this picture is unique?

- Type system support
- CSE, DCE and other “canonicalizations”
- Location tracking and diagnostics
- Pass management
- Regions, basic blocks, statements
- Conversions and validations
- Tooling for tests, benchmarks, etc
MLIR Design
Design principles

**Parsimony**

In compilers, some things are intrinsically complex, avoid making easy things incidentally complex. A small set of versatile built-in concepts enables wide extensibility of the system.

**Traceability**

It is almost always easier to preserve information than to recover it. Keep the compiler accountable: systematic verification and serializability of the IR. Declarative specification of IR elements and transformations.

**Progressivity**

In compilers, premature lowering is the predecessor of all evil. Preserve high-level abstractions as long as necessary, lower them consciously. Embrace diverging flows and extensibility. Intermediate state is important in an IR.
Design requirements

**Parsimony**
- Everything extensible
- SSA graphs + regions

**Traceability**
- Pervasive source location info
- Declarative specification

**Progressivity**
- Support high-level abstractions
- Progressive lowering
Operation

Operation is the unit of semantics wrt execution. The semantics of operations specify what is computed and how. There is no fixed set of operations.

Region

A container attached to an operation that can (indirectly) contain other operations. Either SSA dominance-based CFG or graph.

Block

A list of operations contained in a region with no control flow. The last operation in a block is a terminator that can transfer control flow to blocks or regions.

```plaintext
%res:2 = "mydialect.morph"(%input#3) { some.attribute : true, other_attribute : 1.5 }
({
  ^bb0:
    "mydialect.nested"() : () -> ()
    "mydialect.terminator"() : () -> ()
})
: (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
loc(callsite("foo" at "mysource.cc":10:8))
```
IR Structure

**Operation**

Operation is the unit of semantics wrt execution. The semantics of operations specify what is computed and how.

**Region**

A container attached to an operation that can (indirectly) contain other operations. Either SSA dominance-based CFG or data-flow graph. Lexically scoped.

**Block**

A list of operations contained in a region with no control flow. The last operation in a block is a terminator that can transfer control flow to blocks or regions.

```
%res:2 =
    {{
        %input#3 = { some.attribute : true, other_attribute : 1.5 } {
            ^bb0 = 
                "mydialect.nested"() : () -> ()
                "mydialect.terminator"() : () -> ()
        }
    } = (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
loc(callsite("foo" at "mysource.cc":10:8))
```
IR Structure

**Operation**

Operation is the unit of semantics wrt execution. The semantics of operations specify what is computed and how. There is no fixed set of operations.

**Region**

A container attached to an operation that can (indirectly) contain other operations. Either SSA dominance-based CFG or graph.

**Block**

A list of operations contained in a region with no control flow. The last operation in a block may be a terminator that can transfer control flow to other blocks.

```%res:2 = "mydialect.morph"(%input#3) { some.attribute : true, other_attribute : 1.5 }
{{
  ^bb0:
  "mydialect.nested"() : () -> ()
  "mydialect.terminator"()[^bb0] : () -> ()
}} : (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
loc(callsite("foo" at "mysource.cc":10:8))```
IR Structure is Recursive
IR Objects

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values are units of runtime data. They are defined and used by operations. Values obey static single assignment (SSA) rule. Value names are transient.</td>
<td>Types describe compile-time information about a value. Each value has a type. Operation specifies types of defined and used values. The type system is open.</td>
<td>Attributes describe compile-time information about an operation. They may be optional or mandatory as per operation semantics. The attribute system is open.</td>
</tr>
</tbody>
</table>

```%
%res:2 = "mydialect.morph"(%input#3) { some.attribute : true, other_attribute : 1.5 }
{
  ^bb0:
    "mydialect.nested"() : () -> ()
    "mydialect.terminator"() : () -> ()
}
: (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
loc(callsite("foo" at "mysource.cc":10:8))
```
IR Extensibility Hooks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>No fixed set of operations. Examples:</td>
<td>The type system is open. Examples:</td>
<td>The attribute system is open. Examples:</td>
</tr>
<tr>
<td>- “machine” integer arithmetic;</td>
<td>- $nD$ “machine” vectors;</td>
<td>- integer or string values;</td>
</tr>
<tr>
<td>- saturating integer arithmetic;</td>
<td>- ranked and unranked tensors;</td>
<td>- file:line:col locations;</td>
</tr>
<tr>
<td>- LLVM IR intrinsics (first-class!);</td>
<td>- all of LLVM IR types;</td>
<td>- affine maps;</td>
</tr>
<tr>
<td>- TensorFlow operations;</td>
<td>- functions;</td>
<td>- opaque AST node pointers;</td>
</tr>
<tr>
<td>- affine loops and conditionals;</td>
<td>- Fortran types, ...</td>
<td>- binary blobs;</td>
</tr>
<tr>
<td>- semiconductor circuits, ...</td>
<td></td>
<td>- containers of other attributes, ...</td>
</tr>
</tbody>
</table>
Dialects: families of attributes, operations, types

Dialect ~ abstraction level:
LLVM IR, Fortran FIR, Swift SIL, XLA HLO, TensorFlow Graph, …

A dialect can define:
Operations
Type system(s)
Customization hooks: constant folding, decoding, …

An operation can define:
Invariants on # operands, types, results, attributes, …
Custom parser, printer, verifier, …
Canonicalization patterns, …
Syntax In a Nutshell

%res:2 = "mydialect.morph"(%input#3) { some.attribute = true, other_attribute = 1.5 } : (!mydialect"custom_type") -> (!mydialect"other_type", !mydialect"other_type")
loc(callsite("foo" at "mysource.cc":10:8))
Users and Uses
TensorFlow

- Multiple internal representations (graph, protobuf)
- Conversions between TF ecosystem parts (TF, TFLite)
- Ad-hoc in-memory data structures
%0 = tf.graph (%arg0 : tensor<f32>, %arg1 : tensor<f32>,
               %arg2 : !tf.resource) {
    // Execution of these operations is asynchronous, the %control
    // return value can be used to impose extra runtime ordering,
    // for example the assignment to the variable %arg2 is ordered
    // after the read explicitly below.
    %1, %control = tf.ReadVariableOp(%arg2)
              : (!tf.resource) -> (tensor<f32>, !tf.control)
    %2, %control_1 = tf.Add(%arg0, %1)
              : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
    %control_2 = tf.AssignVariableOp(%arg2, %2, %control)
              : (!tf.resource, tensor<f32>) -> !tf.control
    %3, %control_3 = tf.Add(%2, %arg1)
              : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
    tf.fetch %3, %control_2 : tensor<f32>, !tf.control
}
TensorFlow Graphs

\[
\%0 = \text{tf.graph} (\%\text{arg0} : \text{tensor<f32>}, \%\text{arg1} : \text{tensor<f32>},
\quad \%\text{arg2} : !\text{tf.resource}) \{
\]

// Execution of these operations is asynchronous, the %control
// return value can be used to impose extra runtime ordering,
// for example the assignment to the variable %arg2 is ordered
// after the read explicitly below.

\%1, %control = \text{tf.ReadVariableOp}(\%\text{arg2})
: (!\text{tf.resource}) -> (\text{tensor<f32>}, !\text{tf.control})

\%2, %control_1 = \text{tf.Add}(\%\text{arg0}, \%1)
: (\text{tensor<f32>}, \text{tensor<f32>}) -> (\text{tensor<f32>}, !\text{tf.control})

%control_2 = \text{tf.AssignVariableOp}(\%\text{arg2}, \%2, %control)
: (!\text{tf.resource}, \text{tensor<f32>}) -> !\text{tf.control}

%3, %control_3 = \text{tf.Add}(\%2, \%\text{arg1})
: (\text{tensor<f32>}, \text{tensor<f32>}) -> (\text{tensor<f32>}, !\text{tf.control})

tf.fetch %3, %control_2 :
\quad \text{tensor<f32>}, !\text{tf.control}
\}
%0 = tf.graph (%arg0 : tensor<f32>, %arg1 : tensor<f32>,
    %arg2 : !tf.resource) {
    // Execution of these operations is asynchronous, the %control
    // return value can be used to impose extra runtime ordering,
    // for example the assignment to the variable %arg2 is ordered
    // after the read explicitly below.
    %1, %control = tf.ReadVariableOp(%arg2)
        : (!tf.resource) -> (tensor<f32>, !tf.control)
    %2, %control_1 = tf.Add(%arg0, %1)
        : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
    %control_2 = tf.AssignVariableOp(%arg2, %2, %control)
        : (!tf.resource, tensor<f32>) -> !tf.control
    %3, %control_3 = tf.Add(%2, %arg1)
        : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
    tf.fetch %3, %control_2 : tensor<f32>, !tf.control
}
TensorFlow Graphs

%0 = tf.graph (%arg0 : tensor<f32>, %arg1 : tensor<f32>,
%arg2 : !tf.resource) {
// Execution of these operations is asynchronous, the %control
// return value can be used to impose extra runtime ordering,
// for example the assignment to the variable %arg2 is ordered
// after the read explicitly below.
%1, %control = tf.ReadVariableOp(%arg2)
 : (!tf.resource) -> (tensor<f32>, !tf.control)
%2, %control_1 = tf.Add(%arg0, %1)
 : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
%control_2 = tf.AssignVariableOp(%arg2, %2, %control)
 : (!tf.resource, tensor<f32>) -> !tf.control
%3, %control_3 = tf.Add(%2, %arg1)
 : (tensor<f32>, tensor<f32>) -> (tensor<f32>, !tf.control)
tf.fetch %3, %control_2
 : tensor<f32>, !tf.control
}
TensorFlow Graph Lowering: Mix and Match in a Single IR

- **TensorFlow**
  \[
  \%x = "tf.Conv2d"(%input, %filter) \\
  \quad \{\text{strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}\} \\
  \quad : (\text{tensor<\*xf32>, tensor<\*xf32>}) \rightarrow \text{tensor<\*xf32>}
  \]

- **XLA HLO**
  \[
  \%m = "xla_hlo.AllToAll"(%z) \\
  \quad \{\text{split_dimension: 1, concat_dimension: 0, split_count: 2}\} \\
  \quad : (\text{memref<300x200x32xf32>}) \rightarrow \text{memref<600x100x32xf32>}
  \]

- **LLVM IR**
  \[
  \%f = \text{llvm.add} \ %a, \ %b \\
  \quad : !\text{llvm.float}
  \]
Polyhedral Optimization

**Widely explored in compiler research**
Great success in HPC and image processing kernels. Tensor abstraction gives full control over memory layout.

**Strong mathematical foundation**
Powerful loop dependence analysis and loop transformations.

**Simplified polyhedral form in MLIR**
func @matmul_square(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {  
    %zero = constant 0 : f32  
    %n = dim %A, 0 : memref<?x?xf32>

    affine.for %i = 0 to %n {  
        affine.for %j = 0 to %n {  
            affine.store %zero, %C[%i, %j] : memref<?x?xf32>
            affine.for %k = 0 to %n {  
                %a = affine.load %A[%i, %k] : memref<?x?xf32>  
                %b = affine.load %B[%k, %j] : memref<?x?xf32>  
                %prod = mulf %a, %b : f32  
                %c = affine.load %C[%i, %j] : memref<?x?xf32>  
                %sum = addf %c, %prod : f32  
                affine.store %sum, %C[%i, %j] : memref<?x?xf32>
            }
        }
    }
    return
}
func @matmul_square(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {  %zero = constant 0 : f32  %n = dim %A, 0 : memref<?x?xf32>  
affine.for %i = 0 to %n {  affine.for %j = 0 to %n {  affine.store %zero, %C[%i, %j] : memref<?x?xf32>  affine.for %k = 0 to %n {  %a    = affine.load %A[%i, %k] : memref<?x?xf32>  %b    = affine.load %B[%k, %j] : memref<?x?xf32>  %prod = mulf %a, %b    : f32  %c    = affine.load %C[%i, %j] : memref<?x?xf32>  %sum  = addf %c, %prod    : f32  affine.store %sum, %C[%i, %j] : memref<?x?xf32>  }  }  return  }

Leverages nD structure of standard types.
func @matmul_square(
    %zero = constant 0 : f32
    %n = dim %A, 0 : memref<%x%xf32>
)

    affine.for %i = 0 to %n {
        affine.for %j = 0 to %n {
            affine.store %zero, %C[%i, %j] : memref<%x%xf32>
            affine.for %k = 0 to %n {
                %a = affine.load %A[%i, %k] : memref<%x%xf32>
                %b = affine.load %B[%k, %j] : memref<%x%xf32>
                %prod = mulf %a, %b : f32
                %c = affine.load %C[%i, %j] : memref<%x%xf32>
                %sum = addf %c, %prod : f32
                affine.store %sum, %C[%i, %j] : memref<%x%xf32>
            }
        }
    }
return

Leverages nD structure of standard types.
Affine loops are first-class operations; affine constraints are implemented in the verifier.
func @matmul_square(
    %zero = constant 0 : f32
) {

    %n = dim %A, 0 : memref<?x?xf32>

    affine.for %i = 0 to %n {
        affine.for %j = 0 to %n {
            affine.store %zero, %C[%i, %j] : memref<?x?xf32>

            affine.for %k = 0 to %n {
                %a = affine.load %A[%i, %k] : memref<?x?xf32>
                %b = affine.load %B[%k, %j] : memref<?x?xf32>
                %prod = mulf %a, %b : f32
                %c = affine.load %C[%i, %j] : memref<?x?xf32>
                %sum = addf %c, %prod : f32

                affine.store %sum, %C[%i, %j] : memref<?x?xf32>
            }
        }
    }

    return
}

Leverages nD structure of standard types.
Affine loops are first-class operations; affine constraints are implemented in the verifier.
Load/store operations accept affine maps.
Leverages nD structure of standard types.  
Affine loops are first-class operations; affine constraints are implemented in the verifier.  
Load/store operations accept affine maps.  
Introduce operations from other dialects for computation.
Unified Accelerator and Host Representation

```mlir
@global (42 : i64) : !llvm.i64

func @some_func(%arg0 : memref<?xf32>) {
  %cst = constant 8 : index
  gpu.launch blocks(%bx, %by, %bz) in (%grid_x = %cst, %grid_y = %cst,
  %grid_z = %cst)
    threads(%tx, %ty, %tz) in (%block_x = %cst, %block_y = %cst,
    %block_z = %cst) {
      gpu.call @device_function() : () -> ()
    %0 = llvm.mlir.addressof @global : !llvm<i64*>'
      gpu.return
  }
  return
}

gpu.func @device_function() {
  gpu.call @recursive_device_function() : () -> ()
  gpu.return
}

gpu.func @recursive_device_function() {
  gpu.call @recursive_device_function() : () -> ()
  gpu.return
}
```

Structured Ops

High-performance codegen approach based on *keeping high-level information available in the IR*

- A way to represent operations in the IR that makes them *easy to analyze and transform*
  - e.g. matmul, kfac, conv, pointwise etc -> configurations of a “generic custom op”
    - TC/einsum-like definition encoded in the IR but much more powerful:
      - Matmul -> $C(i, j) += A(i, k) + B(k, j)$
      - Conv1d -> $O(n, w, f) += I(n, w + kw, c) \times K(kw, c, f)$
Structured Ops

High-performance codegen approach based on **keeping high-level information available in the IR**

- A way to represent operations in the IR that makes them easy to analyze and transform
  - e.g. matmul, kfac, conv, pointwise etc -> configurations of a “generic custom op”
  - TC/einsum-like definition encoded in the IR but much more powerful:
    - Matmul \( C(i, j) += A(i, k) + B(k, j) \)
    - Conv1d \( O(n, w, f) += I(n, w + kw, c) \times K(kw, c, f) \)

- A way to decouple op specification from the data type it operates on:
  - matmul(%a: sparse_tensor<4x?xf32, #CSC>, %b: tensor<?x8xf32>, c: tensor<4x8xf32>) \(\rightarrow\) (tensor<4x8xf32>)
  - matmul(%a: buffer<4x?xf32>, %b: buffer<?x8xf32>, c: buffer<4x8xf32>)
  - matmul(%a: vector<4x16xf32>, %b: vector<16x8xf32>, c: vector<4x8xf32>) \(\rightarrow\) (vector<4x8xf32>)
Structured Ops

High-performance codegen approach based on *keeping high-level information available in the IR*

- A way to represent operations in the IR that makes them easy to analyze and transform
  - e.g. matmul, kfac, conv, pointwise etc -> configurations of a “generic custom op”
  - TC/einsum-like definition encoded in the IR but much more powerful:
    - Matmul -> $C(i, j) += A(i, k) + B(k, j)$
    - Conv1d -> $O(n, w, f) += I(n, w + kw, c) * K(kw, c, f)$

- A way to decouple op specification from the data type it operates on:
  - matmul(%a: sparse_tensor<4x?xf32, #CSC>, %b: tensor<?x8xf32>, c: tensor<4x8xf32>)->(tensor<4x8xf32>)
  - matmul(%a: buffer<4x?xf32>, %b: buffer<?x8xf32>, c: buffer<4x8xf32>)
  - matmul(%a: vector<4x16xf32>, %b: vector<16x8xf32>, c: vector<4x8xf32>) -> (vector<4x8xf32>)

- A way to decouple op specification from the control flow required to implement it
  - matmul(%a: buffer<4x?xf32>, %b: buffer<?x8xf32>, c: buffer<4x8xf32>)->(buffer<4x8xf32>)
    - Implies a 3-D control-flow iteration space of size 4x?x8
What does this look like?

// linalg.sdot computes C += A(i) * B(i)

linalg.sdot ins(%A, %B: memref<4xf32>, memref<4xf32>) outs(%C: memref<f32>)
What does this look like?

// linalg.sdot computes C += A(i) * B(i)
linalg.sdot ins(%A, %B: memref<4xf32>, memref<4xf32>) outs(%C: memref<f32>)

%0 = arith.constant 0: index
%1 = arith.constant 1: index
%d0 = memref.dims %A, %c0: memref<4xf32>
snf.for %i = %c0 to %d0 step %c1 {
    %lhs_subset = subset %A(%i, sz=1):
        memref<4xf32> to memref<f32>
    %rhs_subset = subset %B(%i, sz=1):
        memref<4xf32> to memref<f32>
    %acc_subset = subset %C(%i, sz=1):
        memref<f32> to memref<f32>
    linalg.sdot ins(%lhs_subset, %rhs_subset:
        memref<4xf32>, memref<4xf32>)
    outs(%acc_subset: memref<f32>)
}
What does this look like?

```c
// linalg.sdot computes C += A(i) * B(i)
linalg.sdot ins(%A, %B: memref<4xf32>, memref<4xf32>) outs(%C: memref<f32>)

%c0 = arith.constant 0: index
%c1 = arith.constant 1: index
%d0 = memref.dim %A, %c0: memref<4xf32>
scf.for %i = %c0 to %d0 step %c1 {
    %lhs_subset = subset %A(%i), sz=1):
        memref<4xf32> to memref<f32>
    %rhs_subset = subset %B(%i), sz=1):
        memref<4xf32> to memref<f32>
    %acc_subset = subset %C(%i), sz=1):
        memref<f32> to memref<f32>
linalg.sdot ins(%lhs_subset, %rhs_subset:
        memref<4xf32>, memref<4xf32>)
    outs(%acc_subset: memref<f32>)
}
%c0 = arith.constant 0: index
%c1 = arith.constant 1: index
%d0 = memref.dim %A, %c0: memref<4xf32>
scf.for %i = %c0 to %d0 step %c1 {
    %lhs = memref.load %A[%i]: memref<4xf32>
    %rhs = memref.load %B[%i]: memref<4xf32>
    %acc = memref.load %C[]: memref<f32>
    %tmp = math.mulf %lhs, %rhs: f32
    %res = math.addf %acc, %tmp: f32
    %acc = memref.store %res, %C[]: memref<f32>
}
Transformations

Tile, Fuse, Interchange, Multi-Level Vectorize, Bufferize, Pipeline, etc etc etc
Transformations

Tile, Fuse, Interchange, Multi-Level Vectorize, Bufferize, Pipeline, etc etc etc

The result of each transformation is materialized in the IR and composes with all the rest.

- Avoids “C++ in-memory”-only representation fishiness and action at a distance
Transformations

Tile, Fuse, Interchange, Multi-Level Vectorize, Bufferize, Pipeline, etc etc etc

The result of each transformation is materialized in the IR and composes with all the rest.

- Avoids “C++ in-memory”-only representation fishiness and action at a distance

```python
DoubleTilingExpert(
    'matmul_on_tensors',
    'linalg.matmul',
    sizes1=[256, 128, 256],
    interchange1=[1, 2, 0],
    peel1=False,
    pad1=False,
    pack_padding1=[],
    hoist_padding1=[0],
    sizes2=[8, 16, 32],
    interchange2=[0, 1, 2],
    peel2=False,
    pad2=True,
    pack_padding2=[0, 1],
    hoist_padding2=[3, 4])
```

Every value is a tunable knob
Sparse code generation

- Tensor Linear Algebra Compiler (TACO)
- Particularly interesting for its flexibility in sparse code generation

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe.
The tensor algebra compiler. Proc. ACM Program. Lang. 1, OOPSLA, Article 77 (October 2017)
Sparse code generation in MLIR: Sparsity as a Property

```mlir
#trait_matvec = {
  indexing_maps = [
    affine_map<(i,j) -> (i,j)>, // A
    affine_map<(i,j) -> (j)>,    // x
    affine_map<(i,j) -> (i)>     // b
  ],
  // Per-tensor, per-dimension annotation
  sparse = [
    [ "D", "S" ],  // A
    [ "D" ],       // x
    [ "D" ]        // b
  ],
  iterator_types = [
    "parallel",
    "reduction"
  ],
  doc = "b(i) += A(i,j) * x(j)"
}
```

```mlir
func @matvec(%argA: tensor<16x32xf32>, %argx: tensor<32xf32>, %argb: tensor<16xf32>) -> tensor<16xf32> {
  %0 = linalg.generic #trait_matvec
  ins(%argA, %argx : tensor<16x32xf32>, tensor<32xf32>)
  init(%argb : tensor<16xf32>) {
    ^bb(%A: f32, %x: f32, %b: f32):
      %0 = mulf %A, %x : f32
      %1 = addf %0, %b : f32
      linalg.yield %1 : f32
  } -> tensor<16xf32>
  return %0 : tensor<16xf32>
}
```
MLIR Pattern Matching and Rewrite
~ Instruction Selection problem.

Diagram showing pattern matching and rewrite examples, including operations like add, mul, pow, and madd.
MLIR Pattern Matching and Rewrite
An MLIR dialect to manipulate MLIR IR

```mlir
func @matcher(%0 : !Operation) {
  ^bb0:
    CheckArgCount(%0) [^bb1, ^ex0] {count = 2}
      : (!Operation) -> ()
  ^bb1:
    CheckOpName(%0) [^bb2, ^bb5] {name = "add"}
      : (!Operation) -> ()
  ^bb2:
    %1 = GetOperand(%0) {index = 0} : (!Operation) -> !Value
    %2 = GetOperand(%0) {index = 1} : (!Operation) -> !Value
    ValueEqualTo(%1, %2) [^rr0, ^bb3] : (!Value, !Value) -> ()
  ^rr0:
    // Save x
    RegisterResult(%1) [^bb3] {id = 0} : (!Value) -> ()
  ^bb3:
    %3 = GetDefiningOp(%2) : (!Value) -> !Operation
    CheckOpName(%3) [^bb4, ^bb5] {name = "mul"}
      : (!Operation) -> ()
  ^bb4:
    CheckArgCount(%3) [^rr1, ^bb5] {count = 2}
      : (!Operation) -> ()
  ^rr1:
    // Save x, y, and z
    %4 = GetOperand(%3) {index = 0} : (!Operation) -> !Value
    %5 = GetOperand(%4) {index = 1} : (!Operation) -> !Value
    RegisterResult(%1, %4, %5) [^bb5] {id = 1}
      : (!Value, !Value, !Value) -> ()
  ^bb5:
    // Previous calls are not necessarily visible here
    %6 = GetOperand(%0) {index = 0} : (!Operation) -> !Value
    %7 = GetOperand(%0) {index = 1} : (!Operation) -> !Value
    ValueEqualTo(%6, %7) [^bb6, ^ex0] : (!Value, !Value) -> ()
  ^bb6:
    CheckOpName(%0) [^rr2, ^ex0] {name = "mul"}
      : (!Operation) -> ()
  ^rr2:
    // Save x
    RegisterResult(%6) [^ex0] {id = 2} : (!Value) -> ()
  ^ex0:
    return
}
```
Implications of MLIR Design
Designing Abstractions for Reuse

<table>
<thead>
<tr>
<th>Traits</th>
<th>Interfaces</th>
<th>Passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many transformations need not care about specific operations, but can be expressed on generic traits. Establish operation/transform contracts.</td>
<td>Good old OOP is helpful to specialize pass behavior for specific operations. E.g., operations that know how to constant-fold themselves implement an interface.</td>
<td>Generic passes may be expressed on traits and interfaces. Dialect-specific passes are a great tool to perform domain-specific transformations.</td>
</tr>
</tbody>
</table>
Example: Loop-Invariant Code Motion

**Top-Level Op Structure**
An operation with regions. No need to know if it’s an affine “for”, a C-like “while”, or anything else.

**“Loop-Like” Op Interface**
Functions to:
- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

**Nested Op Traits**
- Has no side effects (extensible to side-effects interface);
- Has recursive side effects.
### Example: Loop-Invariant Code Motion

#### Top-Level Op Structure

An operation with regions.

*No need to know if it’s an affine “for”, a C-like “while”, or anything else.*

#### “Loop-Like” Op Interface

Functions to:

- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

#### Nested Op Traits

- Has no side effects (extensible to side-effects interface);
- Has recursive side effects.

For all **loop-like operations**:

*Get the body and for all operations in it:*

  - Ignore operations with side effects (no traits);
  - Ignore operations *containing* side-effecting operations;
  - If all operands are defined outside the loop:
    - Hoist out of the body;
    - On next iterations, the hoisted values are defined outside.*
### Example: Loop-Invariant Code Motion

#### Top-Level Op Structure

An operation with regions. 
No need to know if it’s an affine “for”, a C-like “while”, or anything else.

#### “Loop-Like” Op Interface

Functions to:
- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

#### Nested Op Traits

- Has no side effects
  (extensible to side-effects interface);
- Has recursive side effects.

For all loop-like operations:

**Get the body** and for all operations in it:

- Ignore operations with side effects (no traits);
- Ignore operations containing side-effecting operations;
- If all operands are defined outside the loop:
  - Hoist out of the body;
  - On next iterations, the hoisted values are defined outside.
Example: Loop-Invariant Code Motion

Top-Level Op Structure

An operation with regions.
No need to know if it’s an affine “for”, a C-like “while”, or anything else.

“Loop-Like” Op Interface

Functions to:
- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

Nested Op Traits

- Has no side effects
  (extensible to side-effects interface);
- Has recursive side effects.

For all loop-like operations:

Get the body and for all operations in it:
- Ignore operations with side effects (no traits);
- Ignore operations containing side-effecting operations;
If all operands are defined outside the loop:
- Hoist out of the body;
On next iterations, the hoisted values are defined outside.
**Example: Loop-Invariant Code Motion**

### Top-Level Op Structure
An operation with regions.  
No need to know if it’s an affine “for”, a C-like “while”, or anything else.

### “Loop-Like” Op Interface
Functions to:
- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

### Nested Op Traits
- Has no side effects  
  (extensible to side-effects interface);
- Has recursive side effects.

For all loop-like operations:
Get the body and for all operations in it:
- Ignore operations with side effects (no traits);
- Ignore operations containing side-effecting operations;
If all operands are defined outside the loop:
- Hoist out of the body;
On next iterations, the hoisted values are defined outside.
Example: Loop-Invariant Code Motion

Top-Level Op Structure
An operation with regions.
No need to know if it’s an affine “for”, a C-like “while”, or anything else.

“Loop-Like” Op Interface
Functions to:
- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

Nested Op Traits
- Has no side effects (extensible to side-effects interface);
- Has recursive side effects.

For all loop-like operations:
Get the body and for all operations in it:
Ignore operations with side effects (no traits);
Ignore operations containing side-effecting operations;
If all operands are defined outside the loop:
Hoist out of the body;
On next iterations, the hoisted values are defined outside.
Example: Loop-Invariant Code Motion

Top-Level Op Structure

An operation with regions.
No need to know if it’s an affine “for”, a C-like “while”, or anything else.

“Loop-Like” Op Interface

Functions to:
- check if a value is defined outside the loop (not necessarily a flat SSA CFG);
- get the loop body region;
- hoist operations out of the body.

Nested Op Traits

- Has no side effects (extensible to side-effects interface);
- Has recursive side effects.

For all loop-like operations:
Get the body and for all operations in it:
Ignore operations with side effects (no traits);
Ignore operations containing side-effecting operations;
If all operands are defined outside the loop:
Hoist out of the body;
On next iterations, the hoisted values are defined outside.
Designing Abstractions for Composition

**Mixing Dialects**

Dialects are not necessarily hermetic. Reuse other abstractions when possible and deconstruct larger dialects if needed. Always assume abstractions co-exist.

**External Interoperability**

External formats are messy, often binary or otherwise hard to test. Map them to a dialect and make the translation as simple as possible, then transform within MLIR.
Designing Abstractions for Composition

**Mixing Dialects**

Dialects are not necessarily hermetic. Reuse other abstractions when possible and deconstruct larger dialects if needed. Always assume abstractions co-exist.

**External Interoperability**

External formats are messy, often binary or otherwise hard to test. Map them to a dialect and make the translation as simple as possible, then transform within MLIR.
What the future holds
Driving HW/SW Research

**Domain and HW-specific IRs**
Domain-specific constructs represented as MLIR dialects, leveraged by advanced transformations. No separation between “instructions” and “intrinsics”, support entire ISAs as target dialects. Hardware design as software problem.

**Extensible Type Systems**
Build and experiment with unconventional data types (quantized numbers or mixed-precision floating point). More expressive type systems from functional languages, separation logic, borrow checking.

**Built for Optimization**
Search and ML for Compilers

Expose Compiler Knobs
Separate implementations of program transformations from compiler heuristics. Give control to the expert user or to external tools to enable cross-pollination between compiler and ML research.

Tackle NP-hard Problems
Replace handwritten heuristics, which are often suboptimal and expensive to deploy, with learned transformation strategies. Prepare for the “jungle” of upcoming hardware by automating (re)optimization.
MLIR Is Changing Compiler Construction

**Minimalist Principles**

- Parsimony;
- Traceability;
- Progressivity

MLIR is a novel compiler infrastructure based on the principles of:

**Flexible Core Concepts**

- Nested structure of operations, regions, blocks;
- Operating on typed (SSA) values and attributes

The built-in IR concepts:

allow for expressing various abstractions.

**Reusable Transformations**

Rethink compiler transformations in terms of abstract properties of operations rather than exhaustive lists.

Mix-and-match different abstractions, easy to experiment.
MLIR Is Changing Compiler Construction

**Minimalist Principles**

MLIR is a novel compiler infrastructure based on the principles of:
- Parsimony;
- Traceability;
- Progressivity supporting unprecedented extensibility.

**Flexible Core Concepts**

The built-in IR concepts:
- Nested structure of operations, regions, blocks;
- Operating on typed (SSA) values and attributes allow for expressing various abstractions.

**Reusable Transformations**

Rethink compiler transformations in terms of abstract properties of operations rather than exhaustive lists.

Mix-and-match different abstractions, easy to experiment.
MLIR is Changing Compiler Construction

<table>
<thead>
<tr>
<th>Minimalist Principles</th>
<th>Flexible Core Concepts</th>
<th>Reusable Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLIR is a novel compiler infrastructure based on the principles of:</td>
<td>The built-in IR concepts:</td>
<td>Rethink compiler transformations in terms of abstract properties of operations rather than exhaustive lists.</td>
</tr>
<tr>
<td>- Parsimony;</td>
<td>- Nested structure of operations, regions, blocks;</td>
<td>Mix-and-match different abstractions, easy to experiment.</td>
</tr>
<tr>
<td>- Traceability;</td>
<td>- Operating on typed (SSA) values and attributes</td>
<td></td>
</tr>
<tr>
<td>- Progressivity</td>
<td>allow for expressing various abstractions.</td>
<td></td>
</tr>
<tr>
<td>supporting unprecedented extensibility.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Getting involved
MLIR is Open-Source within LLVM project

MLIR is available
Code: https://mlir.dev/src
Forum: https://mlir.dev/forum
Chat: https://mlir.dev/chat
Main: https://mlir.dev

MLIR is designed for out-of-tree users
Most examples in this presentation are out of LLVM code tree.

Interested? mlir-hiring@google.com
Google Brain PAR/ZRH — C2L2C

**Compile to Learn**
- High-performance ML layers, generated automatically
- Compilation algorithms tailored for tensor computing

**Learn to Compile**
- Automatic construction of profitability models, heuristics
- Heuristics, performance auto-tuning