Abstractions, Algorithms and Infrastructure for Post-Moore Optimizing Compilers

AccML Workshop — January 20th 2020

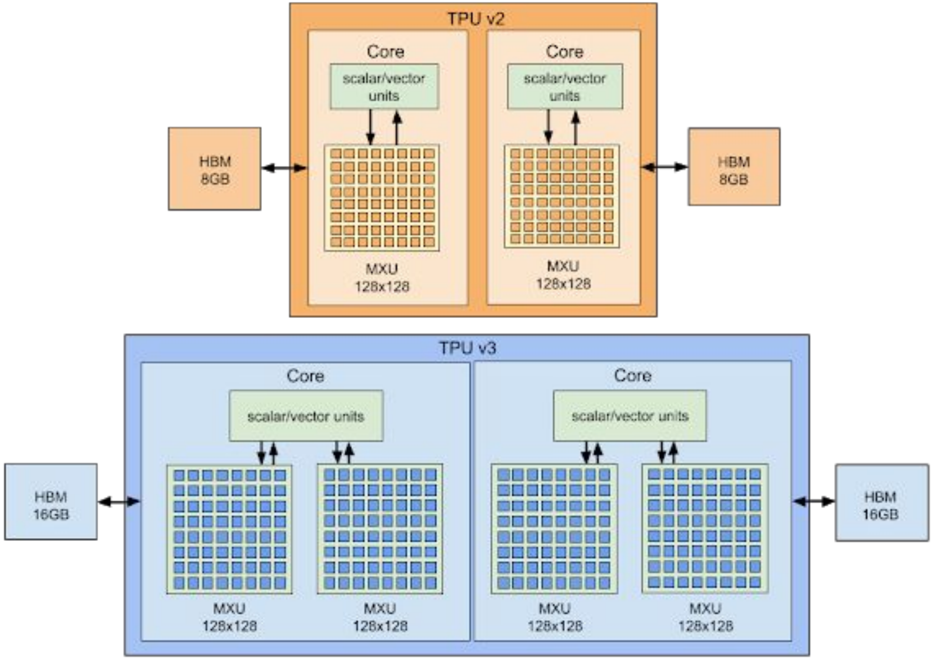
Albert Cohen

presenting the work of many

Accelerated Computing:
… a Detour Through Tiling

Tiles in Accelerated Computing

1. **Hardware**

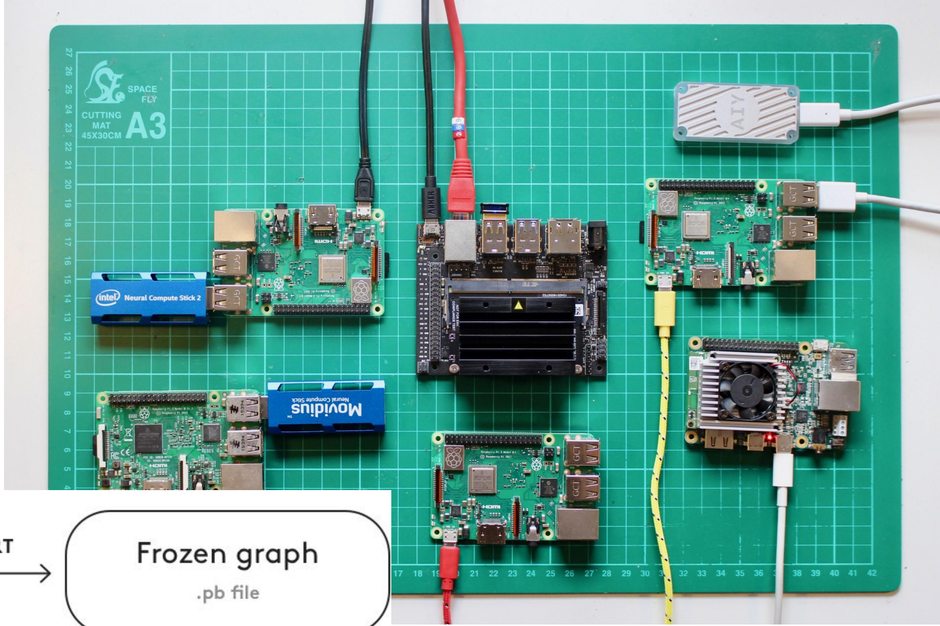


Examples: Google Cloud TPU, Nvidia GPU
Architectural Scalability With Tiling

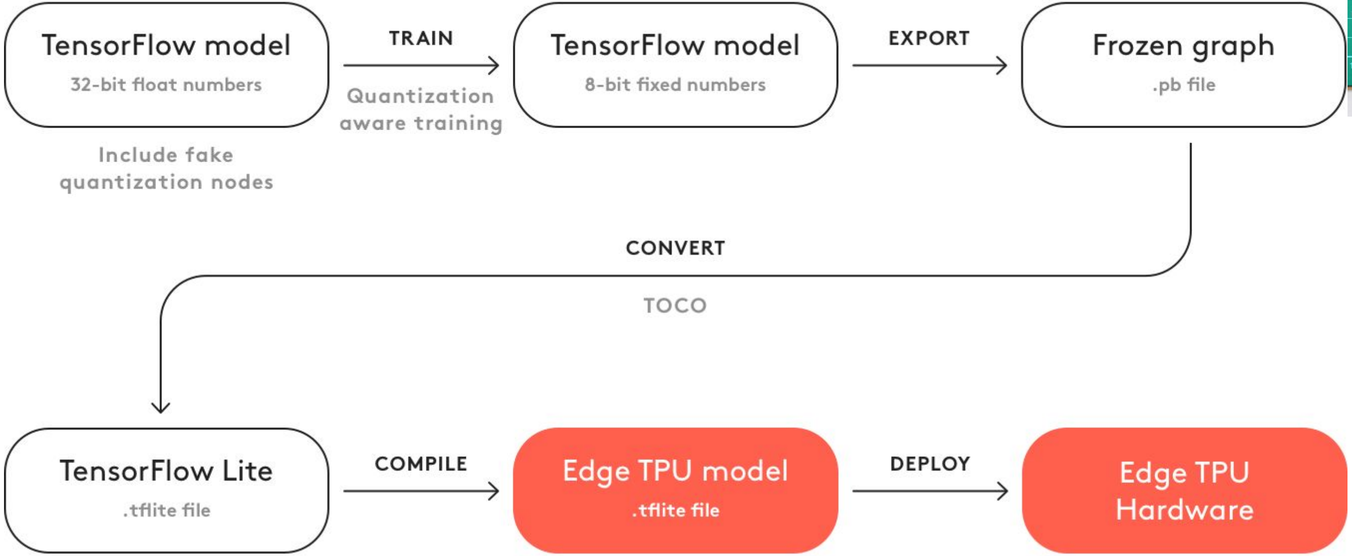
Tiles Everywhere

1. Hardware
2. Tool flow

Example: Google Edge TPU

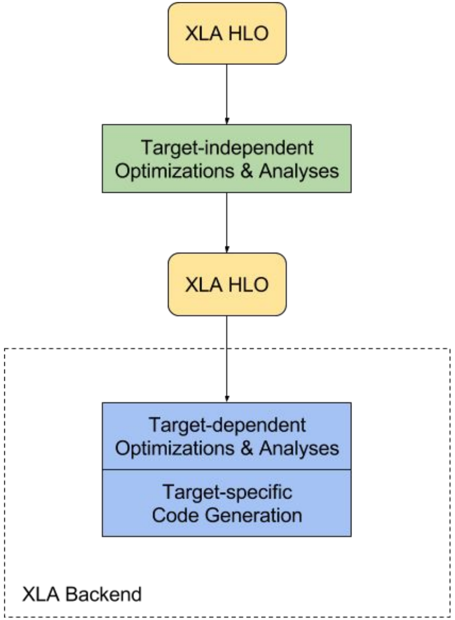


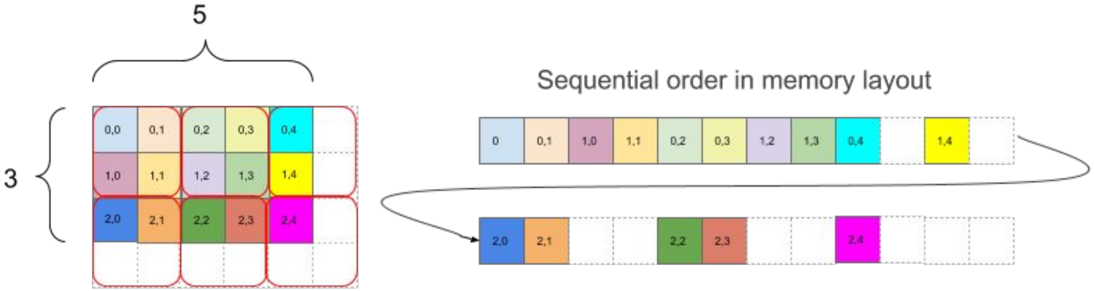
Edge computing zoo



Tiles Everywhere

1. Hardware
2. Tool flow
3. Data layout





F32[3,5] with tile size of 2x2

Example: XLA domain-specific compiler, Tiled data layout

Repeated/Hierarchical Tiling
e.g., BF16 (bfloat16)
on Cloud TPU
(should be 8x128 then 2x1)

Tiles Everywhere

1. Hardware
2. Tool flow
3. Data layout
4. Control flow
5. Data flow
6. Data parallelism

Example: “Single-Op Compiler”

Halide for image processing pipelines

https://halide-lang.org

Meta-programming API and domain-specific language (DSL) for loop transformations, numerical computing kernels

Google

[redacted]

Tiling in Halide

Tiled schedule:
- strip-mine (a.k.a. split)
- permute (a.k.a. reorder)

[redacted]

Vectorized schedule:
- strip-mine
- vectorize inner loop

[redacted]

Non-divisible bounds/extent:
- strip-mine
- shift left/up
- redundant computation

(also forward substitute/inline operand)

Tiles Everywhere

1. Hardware
2. Tool flow
3. Data layout
4. Control flow
5. Data flow
6. Data parallelism

Example: Halide for image processing pipelines

https://halide-lang.org

And also TVM for neural networks

https://tvm.ai

TVM example: scan cell (RNN)

```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_do, s_state, inputs=[X])
s = tvm.create_schedule(s_scan.op)

// Schedule to run the scan cell on a CUDA device
block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis("threadIdx.x")
xo,xi = s[s_init].split(s_init.op.axis[1], factor=num_thread)
s[s_init].bind(xo, block_x)
s[s_init].bind(xi, thread_x)
xo,xi = s[s_do].split(s_do.op.axis[1], factor=num_thread)
s[s_do].bind(xo, block_x)
s[s_do].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

Stepping Back

Context

1. **Heterogeneity** in the **domains** (HPC, ML, signal...) and DSLs
2. **Heterogeneity** in the **hardware** and **infrastructure**
3. Many data **types**, storage **formats**

Compiler

4. **Other transformations**: fusion, fission, pipelining, unrolling...
5. **Composition** of **transformations** and **mapping** decisions
6. **Evaluating** cost functions, **enforcing** resource constraints

⟶ Question

What is the impact on compiler construction,
intermediate representations,
program analyses and transformations?

Compiler Construction for Acceleration

- **Multi-level parallelism**

  CPU — typically 3 levels: system threads or finer grain tasks, vectors, instruction-level parallelism

  GPU — 2 to 8 levels: work groups, work items, warps and vectors, instruction-level parallelism

  and related features on other HW accelerators

- **Deep memory hierarchies**

  + temporal, spatial locality, coalescing, latency hiding through multithreading

  − cache conflicts, false sharing

  ... and many other: capacity constraints, alignment, exposed pipelines

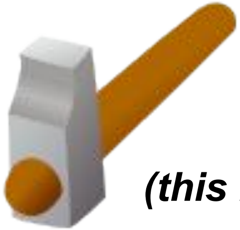
Google

Compiler Construction for Acceleration

- Need a program representation to reason about individual array elements, individual iterations, relations among these, and with hardware resources
  - Programming languages may provide high level abstractions for nested loops and arrays, tensor algebra, graphics...
  - The need for performance portability leads to domain-specific approaches
    E.g., for ML high-performance kernels alone:
    XLA, TVM, Tensor Comprehensions, Glow, Tiramisu, etc.

- Yet few compiler intermediate representations reconcile these with
  1. the ability to model hardware features
  2. while capturing complex transformations
  3. supporting both general-purpose domain-specific optimizers

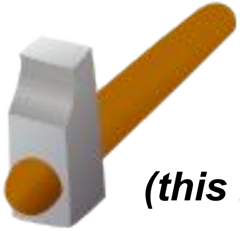
Google



(this is a hammer)

Best-Effort Optimizer

E.g. Intel ICC, Pluto, PPCG, LLVM/Polly



(this is a hammer)

Best-Effort Optimizer

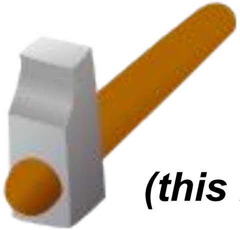
E.g. Intel ICC, Pluto, PPCG, LLVM/Polly

E.g., XLA, Halide, TVM, Polymage

Domain-Specific Optimizer and Code Generator

(these are not nails)

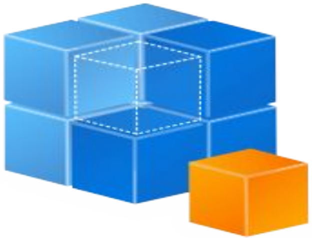
Google



(this is a hammer)

Best-Effort Optimizer

E.g. LLVM



semantical and algorithmic abstractions and design pattern

for program representation, analysis, transformation, optimization, code generation

E.g., XLA, Halide, TVM, Polymage

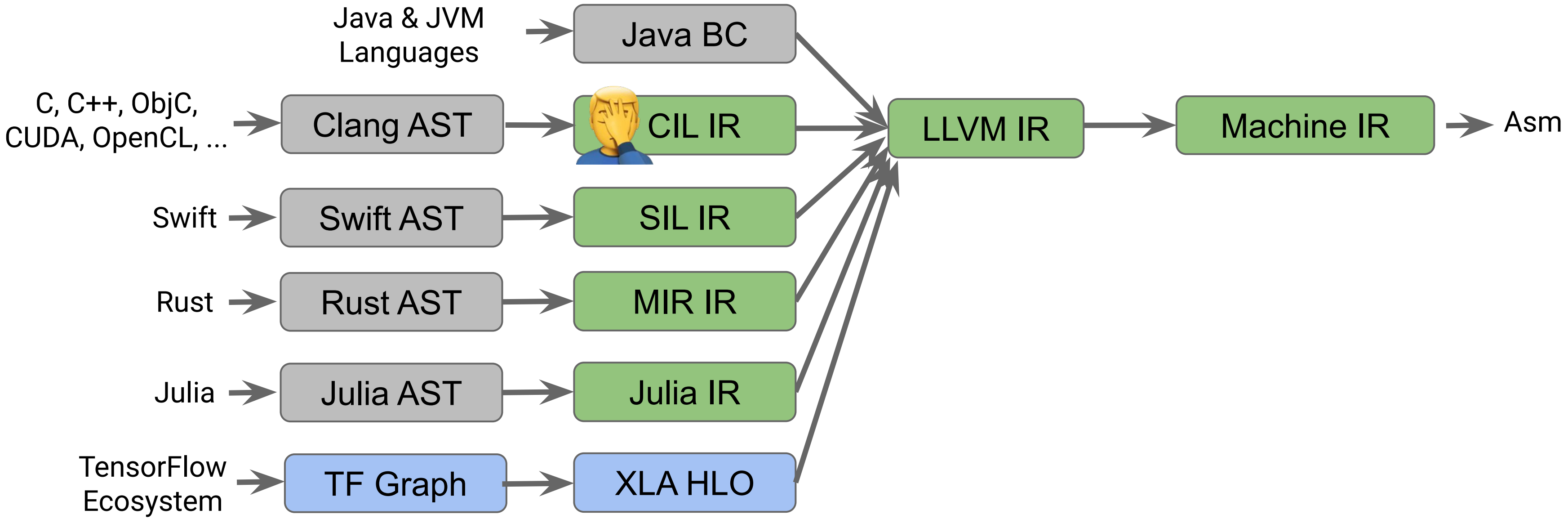
Domain-Specific Optimizer and Code Generator

(these are not nails)

MLIR — What? Why?

1. The right compute/data abstraction at the right time
2. Progressive conversion and lowering of “**ops**”
3. Extend and reuse
4. Industry standard
   → now an LLVM subproject

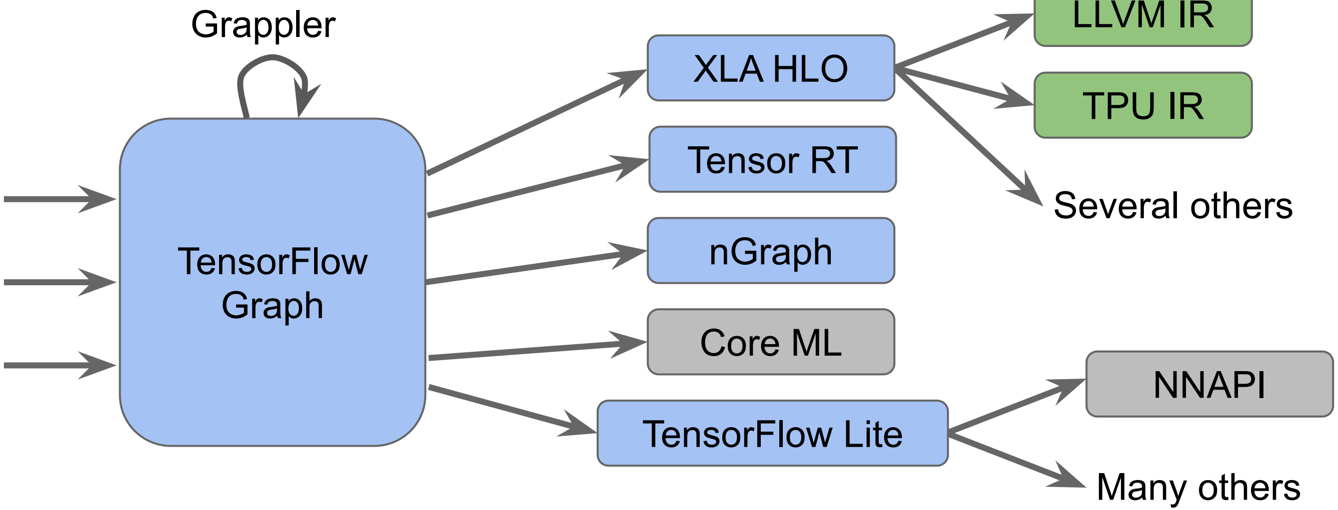
From Programming Languages to the TensorFlow Compiler



- Domain specific optimizations, progressive lowering
- Common LLVM platform for mid/low-level optimizing compilation in SSA form

Google

The TensorFlow Compiler Ecosystem



Many “Graph” IRs, each with challenges:

- Similar-but-different proprietary technologies: not going away anytime soon
- Fragile, poor UI when failures happen: e.g. poor/no location info, or even crashes
- Duplication of infrastructure at all levels

Google

MLIR: A toolkit for representing and transforming “code”

Represent and transform IR ⇄⭯⇓

Represent Multiple Levels of IR at the same time

- tree-based IRs (ASTs)
- data-flow graph IRs (TF Graph, SSA)
- control-flow graph IRs (TF Graph, SSA)
- target-specific parallelism (CPU, GPU, TPU)
- machine instructions

While enabling

Common compiler infrastructure

- location tracking
- richer type system(s)
- common set of conversion passes
- LLVM-inspired infrastructure

And much more

MLIR Core Concepts

Very few core-defined aspects, MLIR is generic and favor extensibility:

- **Region**: a list of basic blocks chained through their terminators to form a CFG.
- **Basic block**: a sequential list of Operations. They take arguments instead of using phi nodes.
- **Operation**: a generic single unit of “code”.
  - takes individual Values as operands,
  - produces one or more SSA Values as results.
  - A terminator operation also has a list of successors blocks, as well as arguments matching the blocks.

There aren’t any hard-coded structures or specific operations in MLIR:

even Module and Function are defined just as regular operations!

MLIR Operations Syntax

```
%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))
```

- Number of value returned
- Dialect prefix
- Op Id
- Argument
- Index in the producer's results
- List of attributes: constant named arguments
- Name of the results
- Dialect prefix for the type
- Opaque string / Dialect specific type
- Mandatory and Rich Location

Example

```
func @some_func(%arg0: !random_dialect<"custom_type">) ->
    !another_dialect<"other_type"> {
  %result = "custom.operation"(%arg0) :
      (!random_dialect<"custom_type">) -> !another_dialect<"other_type">
  return %result : !another_dialect<"other_type">
}
```

Yes: this is a fully valid textual IR module: try round-tripping with *mlir-opt*!

MLIR Operations have Regions

```
%result = "custom.operation"(%arg0) ({
   // Here is a region (new CFG) containing blocks of ops
   ^block:
     %inner_op = "custom.operation"(%input) ...
     %other_op = "custom.operation"(%inner_op) ...
     ...
}, {
   // Possibly multiple regions per operation
})
{ attribute = value : !dialect<"type"> } :
         (!random_dialect<"custom_type">) -> !another_dialect<"other_type">
```

Google

(Operations→Regions→Blocks)+

MLIR is infinitely nested through a recursively defined structure

- Nested regions with control flow, modules, semantic assumptions and guarantees
- Modules and functions are operations with a nested region

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops.                                        Region
  ^block(%argument: !d.type):                                      Block
    // Ops have function types
    %value = "nested.operation"() ({
      // Nested region                                           Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                    Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
  })
// Ops have a list of attributes
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

MLIR SSA Values

```
func @foo(%cond : tensor<i1>, %arg1 : tensor<...>) : (tensor<...>, tensor<...>) {
  %relu = tf.Relu %arg1 : tensor<...>
  %produced_values:2 = tf.if(%cond) {
    %true_branch = tf.Add %arg1, %relu : tensor<...>
    tf.yield %true_branch, %relu  : tensor<...>, tensor<...>
  } else {
    %false_branch = tf.Sub %arg1, %relu : tensor<...>
    tf.yield %relu, %false_branch : tensor<...>, tensor<...>
  }
  tf.print %true_branch : tensor<...>
  return %produced_values#1, %produced_values#0 : tensor<...>, tensor<...>
}
```

- “Implicit capture” of a value inside a region is OK
  (actually only if allowed by the operation holding the region, here `tf.if`)
- For other purposes, a region is similar to a function call, where there is a single user of this function and we see all the context -> more flexibility.
- On the other hand, the values defined in a region can’t escape

The "Catch"

```
func @main() {
  %0 = "libc.printf"() : () -> tensor<10xi1>
}
```

Yes: this is also fully valid textual IR module!

It is not valid though! Broken on many aspects:

- *printf* is not a terminator,
- it should take an operand
- it shouldn't return a tensor value

XML/JSON of compiler IRs?!?

Google

Extensible Operations Allow Multi-Level IR

TensorFlow

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

XLA HLO

```
%m = "xla.AllToAll"(%z)
          {split_dimension: 1, concat_dimension: 0, split_count: 2}
    : (memref<300x200x32xf32>) -> memref<600x100x32xf32>
```

LLVM IR

```
%f = "llvm.add"(%a, %b)
    : (f32, f32) -> f32
```

And many other abstractions for compute, control, data, interfaces…

Operations Have Nested Regions... in a Linear IR ?!

```
%2 = xla.fusion (%0 : tensor<f32>, %1 : tensor<f32>) : tensor<f32> {
^bb0(%a0 : tensor<f32>, %a1 : tensor<f32>):
  %x0 = xla.add %a0, %a1 : tensor<f32>
  %x1 = xla.relu %x0 : tensor<f32>
  return %x1
}

%7 = tf.If(%arg0 : tensor<i1>, %arg1 : tensor<2xf32>) -> tensor<2xf32> {
  … "then" code...
  return ...
} else {
  … "else" code...
  return ...
}
```

Common data flow (SSA) and control flow graph (CFG) of all operations in a region

→ lambdas/closures, parallelism, offloading, etc.

Google

Extensibility Through Dialects

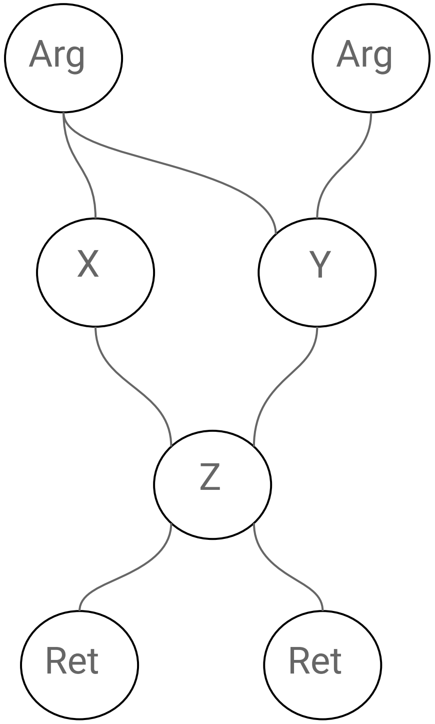
A MLIR dialect is a logical grouping including:

- A prefix (“namespace” reservation)
- A list of types, each one with its C++ class implementation
- A list of operations, each one with its C++ class implementation
  - Verifier for operation invariants (e.g. *printf* first operand is a string)
  - Traits for generic semantics (side-effects, constant-folding, CSE-allowed, etc.)
- Possibly custom parser and printer
- Compilation passes: custom analysis, transformations, and dialect conversions
- Interfaces to register/query transformations and analyses

Example: TensorFlow in MLIR

Computational data-flow graphs,
and modeling control flow, asynchrony

TensorFlow in MLIR — Computational Graph Dialect




```
func @foo( %arg0 : tensor<i1>, %arg1 : tensor<...>) ... {
    %X    = tf.X %arg0 : tensor<...>
    %Y    = tf.Y %arg0, %arg1 : tensor<...>, tensor<...>
    %Z:2 = tf.Z %X, %Y : tensor<...>, tensor<...>
    return %Z#0, %Z#1 : tensor<...>, tensor<
}
```

TensorFlow in MLIR — Control Flow and Concurrency

Control flow and dynamic features of TF1, TF2

- Conversion from control to data flow
- Lazy evaluation

Concurrency

- Sequential execution in blocks
- Distribution
- Offloading
- Implicit concurrency in `tf.graph` regions
    - Implicit **futures** for SSA-friendly, asynchronous task parallelism

→ **Research: task parallelism, memory models, separation logic**

Google

TensorFlow in MLIR — Control Flow and Concurrency

```
%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
}
```

Google

Example: Linalg Dialect

Better and beyond single-op compilers: composition and decomposition of structured operations

Single-Op Compiler: Better and Beyond

- **Code generation path** mixing different styles of **abstraction** and **transformation**
  - Combinators (tile, fuse, communication generation on high level operations)
  - Loop-based (dependence analysis, fuse, vectorize, pipeline, unroll-and-jam)
  - SSA (data flow)
- That **does not require heroic analyses** and transformations
  - Declarative properties enable transformations w/o complex analyses
  - If/when good analyses exist, we can use them
- Beyond **black-box** numerical libraries
  - **Compiling loops + native library calls or hardware blocks**

Linalg Type System And Type Building Ops

- Range type: create a (min, max, step)-triple of `index`

```
%0 = linalg.range %c0:%arg1:%c1 : !linalg.range
```

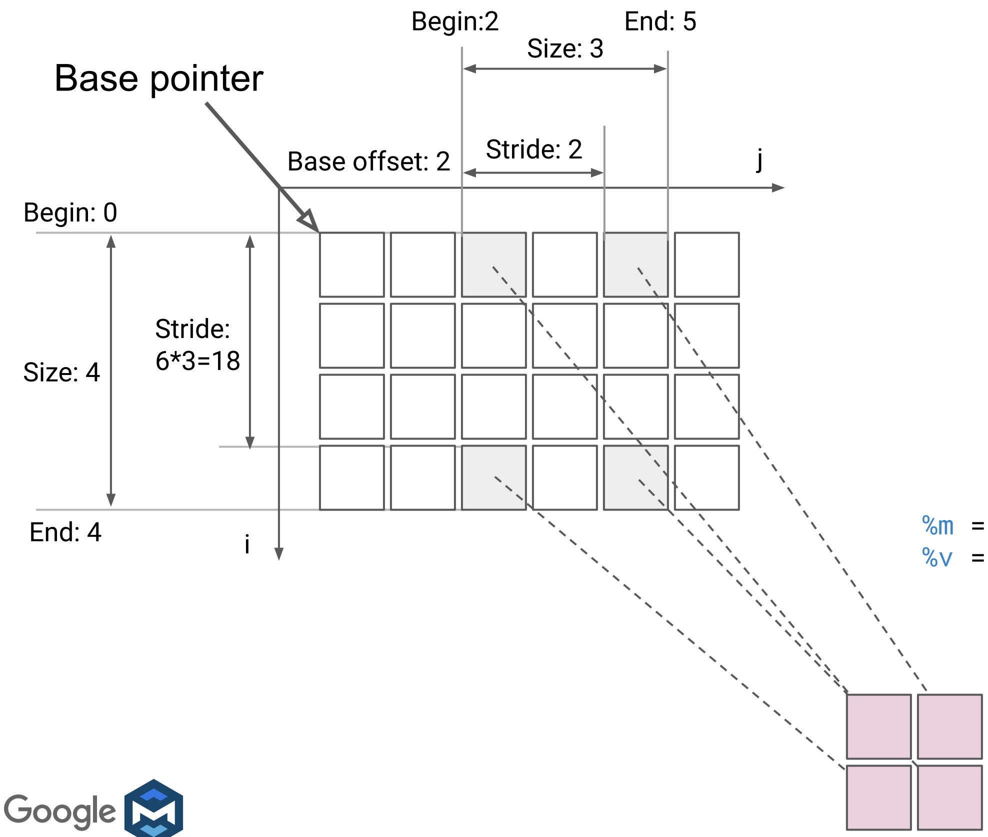
→ for stepping over loop iterations (loop bounds) & data structures

- Strided memref type: create an n-d *"indexing"* over a `memref` buffer

```
%8 = std.view %7[%c0][%s0, %s1] : memref<?x?xf32, offset=0, strides=[?, 1]>
```

Google

Strided MemRef Type and Descriptor

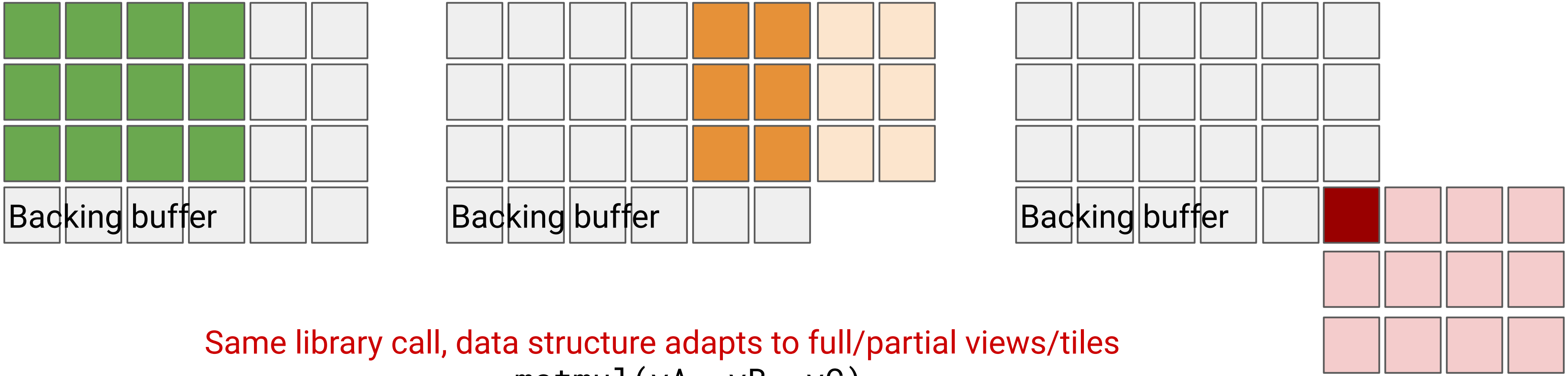



```
{ float*,   # base pointer
  i64,      # base offset
  i64[2]    # sizes
  i64[2] } # strides
```

```
%m = alloc() : memref<4x6 x f32>
%v = view %m[%c0][%r,%r] : memref<?x?xf32, offset = 0,
                                        strides = [?, 1]>
```

Linalg View

- Simplifying assumptions for analyses and IR construction
  - E.g. non-overlapping rectangular memory regions (symbolic shapes)
  - Data abstraction encodes boundary conditions



Same library call, data structure adapts to full/partial views/tiles

```
matmul(vA, vB, vC)
```

Defining Matmul

- `linalg.matmul` operates on strided memrefs (including contiguous memref with canonical strides)

```
func @call_linalg_matmul(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>){
  linalg.matmul(%A, %B, %C) : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
  return
}
```

Lowering Between Linalg Ops: Matmul to Matvec

```
func @matmul_as_matvec(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  %rM = linalg.range %c0:%M:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  loop.for %col = 0 to %N step %c1 {
    %7 = linalg.slice %B[%rK, %col] : memref<?x?xf32>, !linalg.range, index
    %8 = linalg.slice %C[%rM, %col] : memref<?x?xf32>, !linalg.range, index
    linalg.matvec(%A, %7, %8) : memref<?x?xf32>, memref<?xf32>, memref<?xf32>
  }
  return
}
```

Matmul to Matvec: Implementation

```
// Drop the `j` loop from matmul(i, j, k).
// Parallel dimensions permute.
// TODO: Specify as a composable rewrite pattern.
void MatmulOp::rewriteAsMatvec() {
  auto *op = getOperation();
  ScopedContext scope(FuncBuilder(op), op->getLoc());
  IndexHandle j;
  auto *vA(getInputView(0)), *vB(...), *vC(...);
  Value *range = getViewRootIndexing(vB, 1).first;
  LoopNestRangeBuilder(&j, range)({
      matvec(vA, slice(vB, j, 1), slice(vC, j, 1)),
  });
}
```

Extracting/analyzing this information from transformed and tiled loops would take much more effort

With high-level dialects it is a simple rewrite rule

Loop Tiling

```
linalg.matmul(%A, %B, %C) :
  memref<?x?xf32>, ...
```

tileSizes = {8, 9}

```
func @matmul_tiled_loops(%arg0: memref<?x?xf32>,
      %arg1: memref<?x?xf32>, %arg2: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %cst = constant 0.000000e+00 : f32
  %M = dim %arg0, 0 : memref<?x?xf32>
  %N = dim %arg2, 1 : memref<?x?xf32>
  %K = dim %arg0, 1 : memref<?x?xf32>
  loop.for %i0 = 0 to %M step 8 {
    loop.for %i1 = 0 to %N step 9 {
      loop.for %i2 = 0 to %K {
        loop.for %i3 = max(%i0, %c0) to min(%i0 + 8, %M) {
          affine.for %i4 = max(%i1, %c0) to min(%i1 + 9, %N) {
            %3 = cmpi "eq", %i2, %c0 : index
            %6 = load %arg2[%i3, %i4] : memref<?x?xf32>
            %7 = select %3, %cst, %6 : f32
            %9 = load %arg1[%i2, %i4] : memref<?x?xf32>
            %10 = load %arg0[%i3, %i2] : memref<?x?xf32>
            %11 = mulf %10, %9 : f32
            %12 = addf %7, %11 : f32
            store %12, %arg2[%i3, %i4] : memref<?x?xf32>
```

Boundary conditions

View Tiling

```
func @matmul_tiled_views(%A: memref<?x?xf32>, %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c8 = constant 8 : index
  %c9 = constant 9 : index
  %M = dim %A, 0 : memref<?x?xf32>
  %N = dim %C, 1 : memref<?x?xf32>
  %K = dim %A, 1 : memref<?x?xf32>
  loop.for %i0 = 0 to %M step %c8 {
   loop.for %i1 = 0 to %N step %c9 {
      %4 = affine.min (%i0 + %c8, %M)
      %5 = affine.min (%i1 + %c9, %N)
      %6 = linalg.subview %A[%i0, %c0][%4, %K] : memref<?x?xf32, offset = ?, strides = [?, 1]>
      %7 = linalg.subview %B[%c0, %i1][%K, %5] : memref<?x?xf32, offset = ?, strides = [?, 1]>
      %8 = linalg.subview %C[%i0, %i1][%M, %N] : memref<?x?xf32, offset = ?, strides = [?, 1]>
      linalg.matmul(%6, %7, %C) : memref<?x?xf32, offset = ?, strides = [?, 1]>,
                                   memref<?x?xf32, offset = ?, strides = [?, 1]>,
                                   memref<?x?xf32, offset = ?, strides = [?, 1]>,
  }}
```

Nested linalg.**matmul** call

Example: Affine Dialect

For general-purpose loop nest optimization, vectorization, data parallelization, optimization of array layout, storage, transfer

Affine Dialect for Polyhedral Compilation

Affine constraints in this dialect: the if condition is an affine function of the enclosing loop indices.

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (d0 - 1 >= 0, -d0 + 8 >= 0)(%k) {
        // Call foo except on the first and last iteration of %k
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

Google

https://github.com/tensorflow/mlir/blob/master/g3doc/Dialects/Affine.md

MLIR Affine Dialect's Custom Parser and Printer

```
#map0 = () -> (0)
#map1 = () -> (10)
#set0 = (d0) : (d0 * 8 - 4 >= 0, d0 * -8 + 7 >= 0)
func @test() {
  "affine.for"() ( {
  ^bb0(%arg0: index):
    "affine.for"() ( {
    ^bb0(%arg1: index):
      "affine.if"(%arg0) ( {
        "foo"(%arg0) : (index) -> ()
        "affine.terminator"() : () -> ()
      },  {
      }) {condition = #set0} : (index) -> ()
      "affine.terminator"() : () -> ()
    }) {lower_bound = #map0, step = 1 : index, upper_bound = #map1} : () -> ()
    "affine.terminator"() : () -> ()
  }) {lower_bound = #map0, step = 1 : index, upper_bound = #map1} : () -> ()
  "std.return"() : () -> ()
}
```

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (d0 - 1 >= 0, -d0 + 8 >= 0)(%k) {
        // Call foo except on the first and last iteration
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

You get the code on the left from the code on the right with:

```
mlir-opt -- affine.mlir --mlir-print-op-generic
```

Google

Affine Control Flow and Data Layout

- Polynomial multiplication kernel: $C(i+j) \mathrel{+}= A(i) \times B(j)$

```
// Affine loops are Ops with regions.
affine.for %arg0 = 0 to %N {
  // Only loop-invariant values, loop iterators, and affine
  // functions of those are allowed.
  affine.for %arg1 = 0 to %N {
    // Body of affine for loops obey SSA.
    %0 = affine.load %A[%arg0] : memref<? x f32>
    // Structured memory reference (memref) type can have
    // affine layout maps.
    %1 = affine.load %B[%arg1]
           : memref<? x f32, (d0)[s0] -> (d0 + s0)>
    %2 = mulf %0, %1 : f32
    // Affine load/store can have affine expressions as subscripts
    %3 = affine.load %C[%arg0 + %arg1] : memref<? x f32>
    %4 = addf %3, %2 : f32
    affine.store %4, %C[%arg0 + %arg1] : memref<? x f32>
  }
}
```

(static) affine layout map

Affine Dialect for Polyhedral Compilation

- Related work and tool flows
  - Intel Tile and Stripe dialects (from vertex.ai PlaidML)
  - ETHZ/Vulcan/MeteoSwiss Stencil dialect

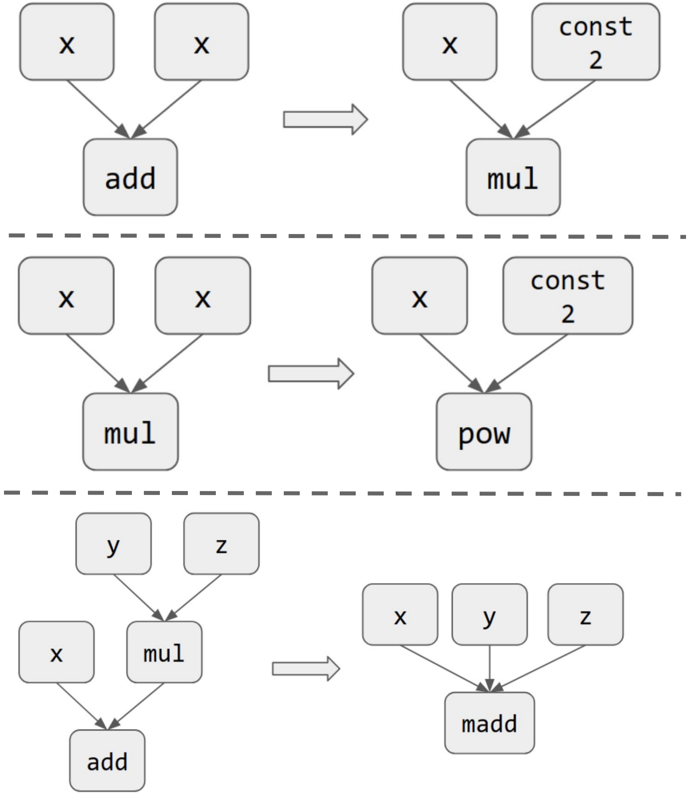
- And many others: DSLs, low level dialects, transformation frameworks...

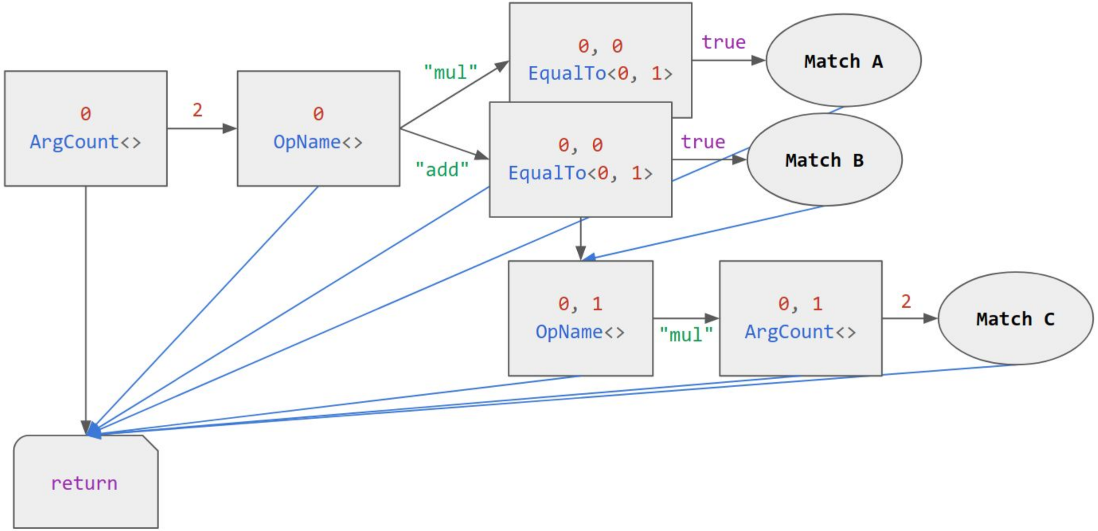
Example: MLIR PatternMatch Execution

Meta-level: MLIR applied to MLIR internals!

MLIR Pattern Matching and Rewrite

~ Instruction Selection problem.





Google

MLIR Pattern Matching and Rewrite

An MLIR dialect to manipulate MLIR IR!

```
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
}
```

Example: Stencil Computation

MLIR for accelerating climate modelling

Open Climate Compiler Initiative

VULCAN

MeteoSwiss

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

SPCL

ETH zürich

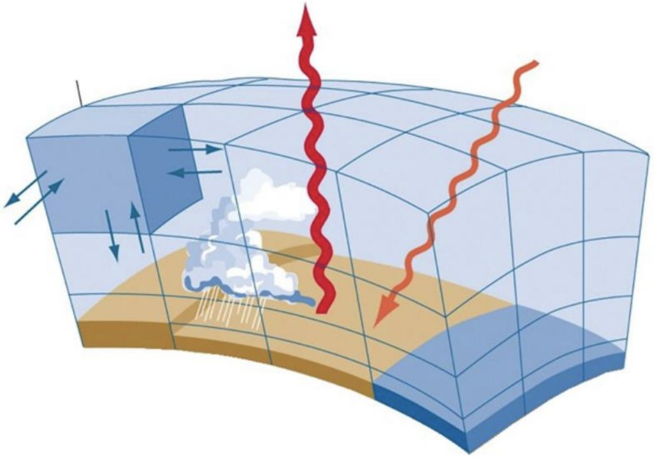
A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

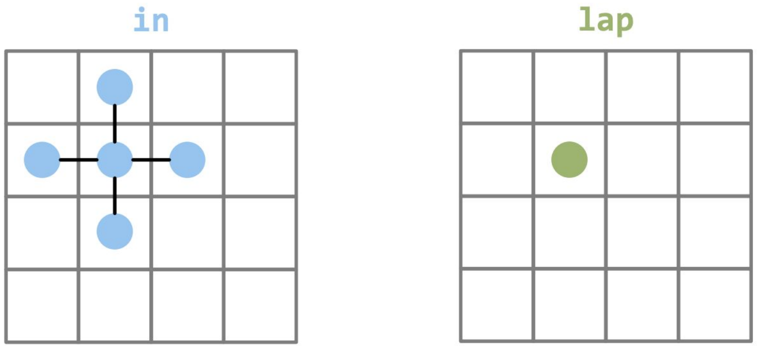
Domain-Science vs Computer-Science

- solve PDE
- finite differences
- structured grid

- element-wise computation
- fixed neighborhood



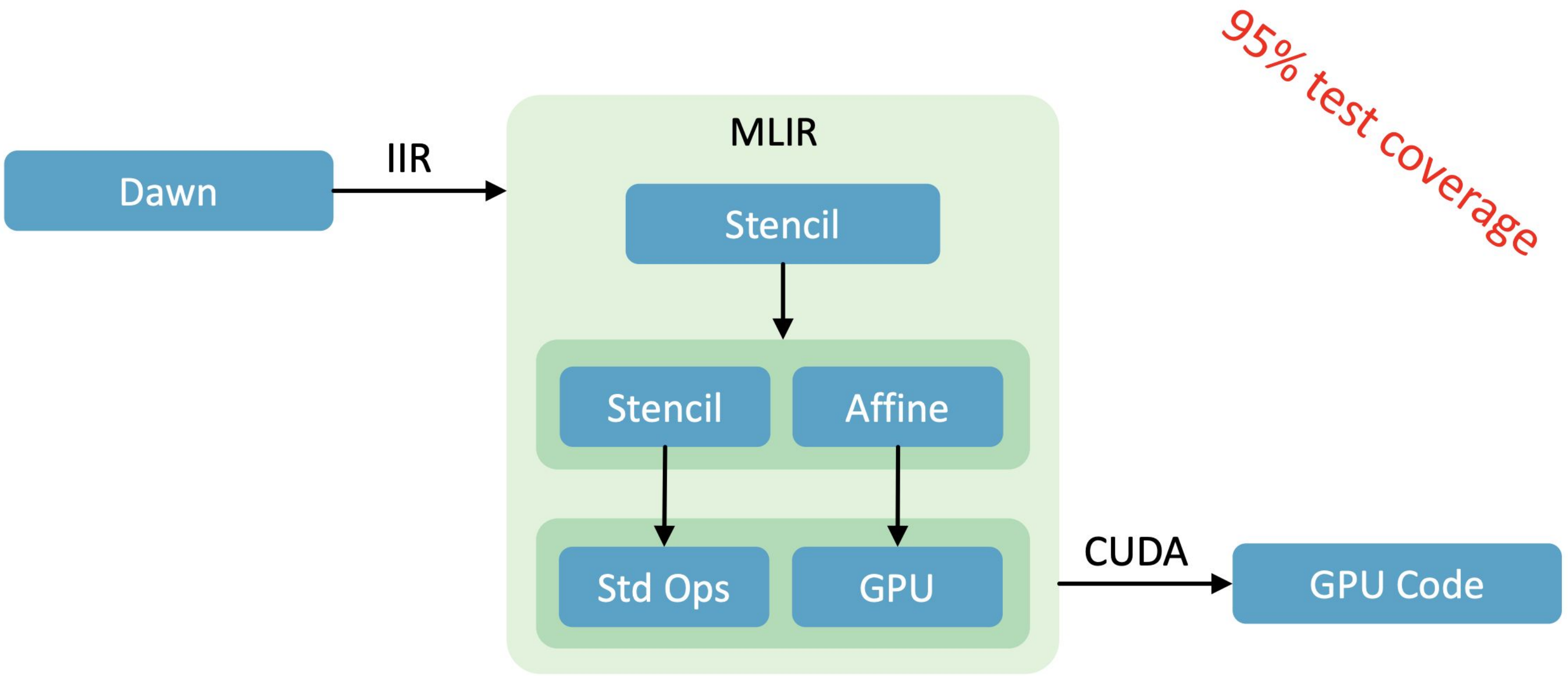
```
lap(i,j) = -4.0 * in(i,j) +
  in(i-1,j) + in(i+1,j) +
  in(i,j-1) + in(i,j+1)
```



A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

Our Current Toolchain



A Compiler Intermediate Representation for Stencils

JEAN-MICHEL GORIUS, TOBIAS WICKY, TOBIAS GROSSER, AND TOBIAS GYSI

Low-level Dialect (IIR)

```
stencil.iir {
  stencil.stencil(%arg0: !stencil<"field:f64">, %arg1: !stencil<"field:f64">) {
    stencil.multi_stage "Parallel" {
      stencil.stage {
        stencil.do_method [0, 0, 60, 0] {
          %0 = stencil.field_access %arg1 [0, 0, 0] : !stencil<"ptr:f64">
          %1 = stencil.field_access %arg0 [0, 0, 0] : !stencil<"ptr:f64">
          %2 = stencil.get_value %0 : f64
          %3 = stencil.get_value %1 : f64
          %4 = addf %2, %3 : f64
          %cst = constant 4.000000e+00 : f64
          %5 = mulf %4, %cst
          stencil.write %0, %5 : f64
        }
      }
    }
  }
}
```

Google

Example: Fortran IR

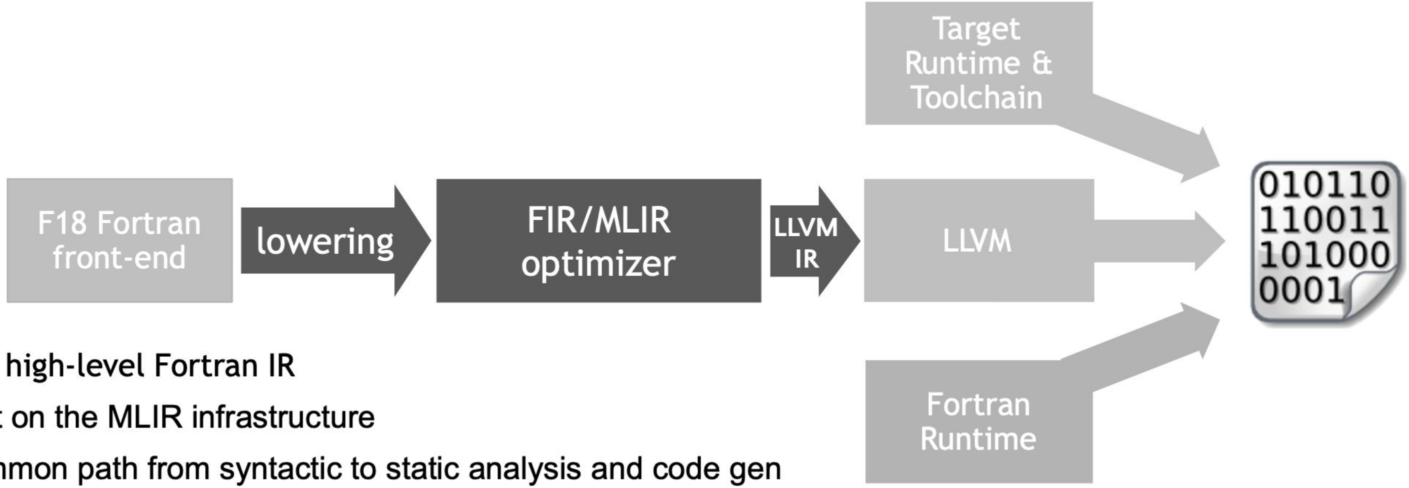
Flang: the LLVM Fortran Fortrend

An MLIR Dialect for High-Level Optimization of Fortran

Eric Schweitz (NVIDIA)

FLANG

The LLVM Fortran compiler



FIR: high-level Fortran IR

Built on the MLIR infrastructure

Common path from syntactic to static analysis and code gen

Shrink abstraction gap: core Fortran operational properties

Focus on writing Fortran aware optimizations

Separation of concerns: constraints checking vs. optimizing computation

An MLIR Dialect for High-Level Optimization of Fortran

Eric Schweitz (NVIDIA)

LOOPS

An example of loop optimization

```
// subroutine convolution(r, f, g)
func @convolution(%r : !fir.box<!fir.array<?:f32>>, %f : !fir.box<...>, %g : !fir.box<...>) {
 %uf:3 = fir.box_dims %f, 0 : (!fir.box<...>, index) -> (index, index, index)  ... // and %ug:3
 fir.loop %n = 1 to %uf#1 {
  fir.loop %k = 1 to %ug#1 {
   %2 = subi %n, %k : index
   %3 = fir.coordinate_of %f, %2 : (!fir.box<...>, index) -> !fir.ref<f32>
   %4 = fir.load %3 : !fir.ref<f32>  ... // and likewise %6 = load g[k]
   %7 = mulf %6, %4 : f32            ... // and likewise %9 = load r[n]
   %10 = addf %9, %7 : f32
   fir.store %10 to %8 : !fir.ref<f32>
}}}
```

An MLIR Dialect for High-Level Optimization of Fortran

Eric Schweitz (NVIDIA)

OBJECT-ORIENTED PROGRAMMING

FIR: Devirtualization

```
// dispatch table for type(u)
fir.dispatch_table @dtable_type_u {
  fir.dt_entry "method", @u_method
}

  %uv = fir.alloca !fir.type<u> : !fir.ref<!fir.type<u>>
  fir.dispatch "method"(%uv) : (!fir.ref<!fir.type<u>>) -> ()
```

Google

Dialect Combination: Heterogeneous Compiler IR

Unified Accelerator and Host Representation

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

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

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

Google

Nested Module Allows Splitting Host/Device, Still in the Same IR

```
module attributes {gpu.container_module} {
  func @some_func(%arg0: memref<?xf32>) {
    %c8 = constant 8 : index
    gpu.launch_func(%c8, %c8, %c8, %c8, %c8, %c8)
       {kernel = "function_call_kernel", kernel_module = @function_call_kernel}
       : (index, index, index, index, index, index) -> ()
    return
  }
  module @function_call_kernel attributes {gpu.kernel_module} {
    func @function_call_kernel() attributes {gpu.kernel} {
      %0 = gpu.block_id() {dimension = "x"} : () -> index
      ...
      %3 = gpu.thread_id() {dimension = "x"} : () -> index
      ...
      call @device_function() : () -> ()
      return
    }
    func @device_function() {
      call @recursive_device_function() : () -> ()
      gpu.return
    }
    llvm.mlir.global internal @global(42 : i64) : !llvm.i64
    func @recursive_device_function() {
      call @recursive_device_function() : () -> ()
      gpu.return
    }
  }
```

Stepping Back: Strengths of Polyhedral Compilation

Decouple intricate optimization problems

Candidate Implementations

- Optimizations and lowering, choices and transformations
  e.g., tile? unroll? ordering?
- Generate imperative code, calls to native libraries, memory management

Constraints

- Semantics
  e.g., def-use, array dependences
- Resource constraints
  e.g., local memory, DMA

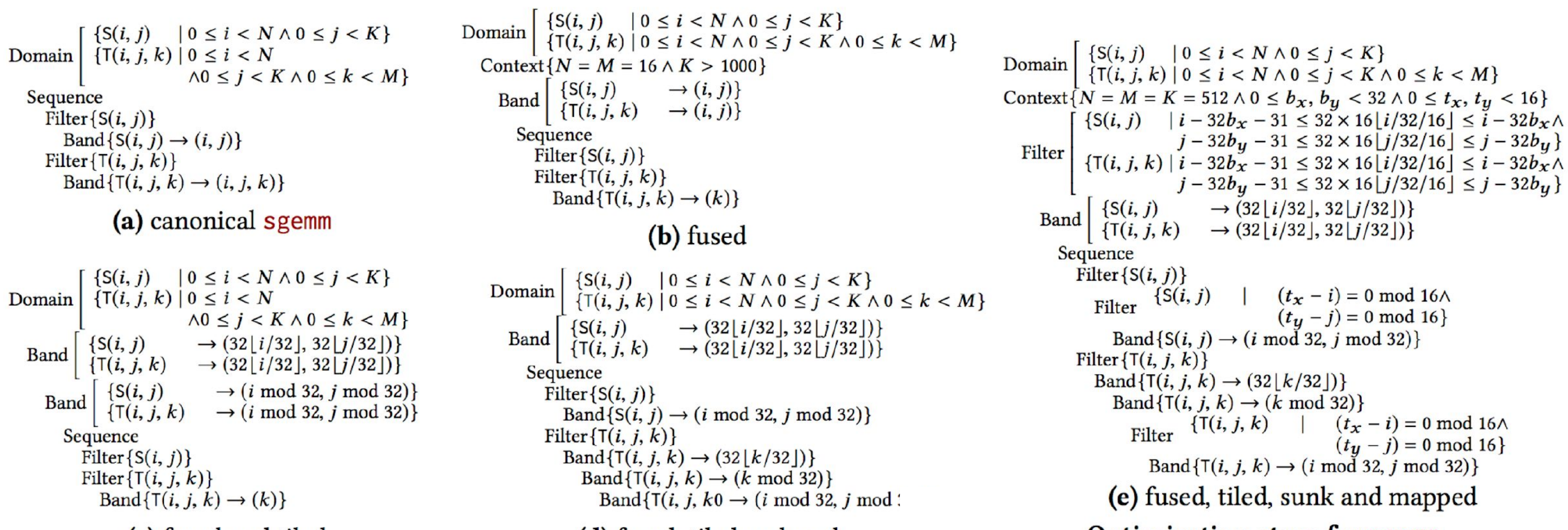
Optimization / Search

- Objective functions
  linear approximations, resource counting, roofline modeling...
- Feedback from actual execution
  profile-directed, JIT, trace-based...
- Combinatorial optimization
  ILP, SMT, CSP, graph algorithms, reinforcement learning...

Google

Then, Isn't it Much More Than Affine Loops and Sets/Maps?

- Example: **isl** schedule trees, inspiration for the MLIR affine dialect



(d) fused, tiled and sunk

Optimization steps for sgemm

Integer Set Library (isl)

- Mathematical core: parametric linear optimization, Presburger arithmetic
  used in **LLVM Polly**
  and many research projects including **Pluto**, **PPCG, PoCC, Tensor Comprehensions**...
- Building on **12 years of collaboration**

  Inria, ARM, ETH Zürich

  AMD, Qualcomm, Xilinx, Facebook

  IISc, IIT Hyderabad

  Ohio State University, Colorado State University, Rice University

  Google Summer of Code

Observation

Most program analyses and transformations over numerical computations can be captured using **symbolic/parametric intervals**

⟶ need an abstraction for **symbolic (parametric) integral hyper-rectangles:** a **sub-polyhedral abstraction**

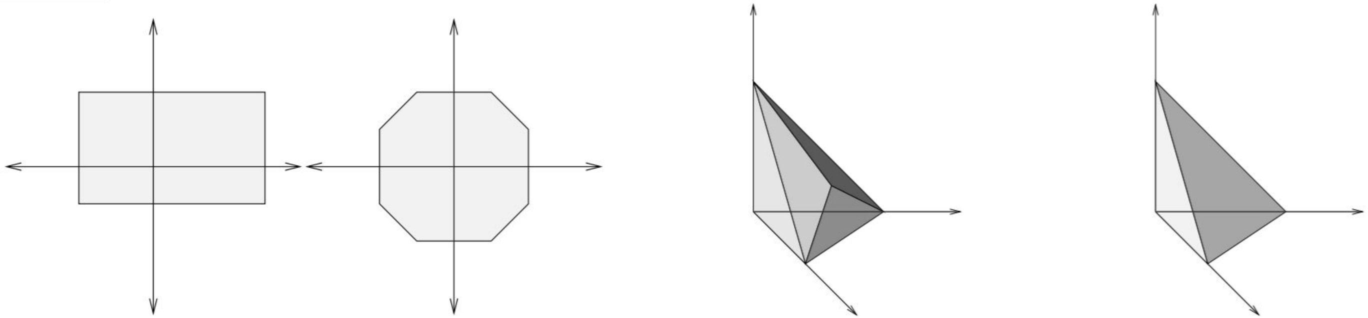
⟶ support **tiling on dynamic shapes**

⟶ support **shifting/pipelining**

⟶ **transformation composition is key**

(Sub-)Polyhedral Abstraction Examples (not integer-precise)

Theme: Trade precision for cost.



Interval	Octagon (UTVPI)	TVPI	Convex Polyhedra
	Unit Two Variable Per Inequality	Two Variable Per Inequality	
$a \leq x_i \leq b$	$\pm x_i \pm x_j \leq c$	$ax_i + bx_j \leq c$	$\sum a_i x_i \leq c$

$$\text{Intervals} \subset \text{Octagons (UTVPI)} \subset \text{TVPI} \subset \text{Conv.Poly}$$

Precision →

Complexity →

Google

See Upadrasta's thesis, POPL 2013

MLIR's Research Proposal for a Polyhedral-Lite Framework

1. Sufficiently rich abstraction and collection of algorithms to support a **complete**, low complexity, easy to implement, easy to adopt, **sub-polyhedral** compilation flow that includes **tiling**
   "complete" = loop nest + layout + data movement + vectorization + operator graph + composable
   "sub-polyhedral" = less expressive than Presburger arithmetic, but still integer sets

2. Implemented on **two's complement machine arithmetic**, rather than natural/relative numbers (bignums, e.g., GMP)
   aiming for correctness-by-construction whenever possible, resorting to static safety checks when not, and to runtime safety checks as a rare last resort

MLIR

MLIR for Accelerated Computing in a Nutshell

MLIR is a powerful infrastructure for

- The compilation of high-level abstractions and domain-specific constructs for ML and HPC
- Reducing the impedance mismatch across languages, abstraction levels, specific ISAs and APIs
- Gradual and partial lowering, legalization from dialect to dialect, mixing dialects
- Code reuse in a production environment, using a robust SSA-based LLVM-style infrastructure
- **Research across the computing system stack**

Check out github, mailing list, chat room, weekly public meeting
Stay tuned for further announcements
Get started with the tutorial (slides)
Workshops: LLVM Dev Meetings
LCPC MLIR4HPC - HiPEAC AccML - CGO C4ML - more to come