Google

# MLIR: An Optimizing Compiler Framework for the End of Moore's Law

U. of Oxford — Dpt. of CS — 4th December 2019

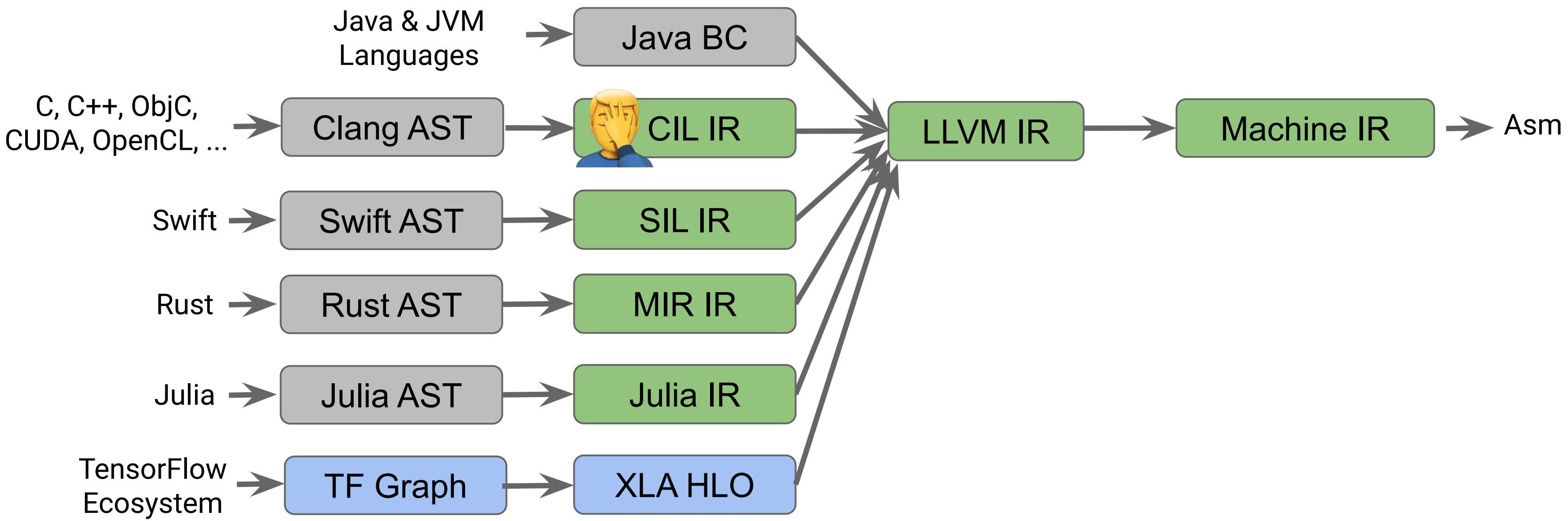
albertcohen@google.com
presenting the work of many

# The LLVM Ecosystem: Clang Compiler


Green boxes are **Static Single Assignment (SSA) Intermediate Representations (IRs)**

- Different levels of abstraction — operations and types are different
- Abstraction-specific optimization at both levels

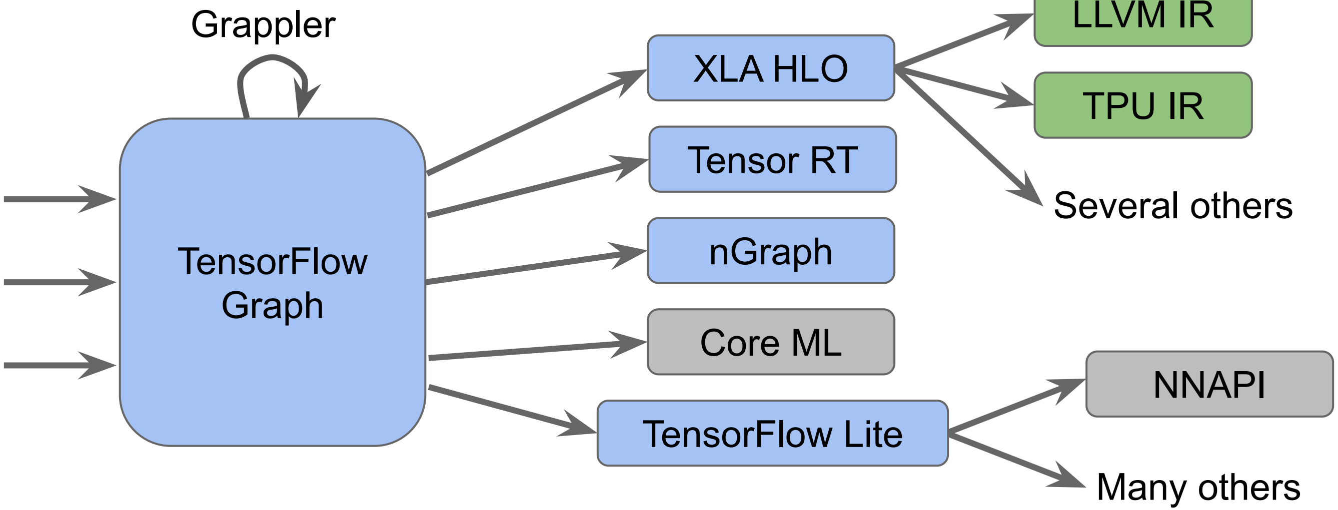
# From Programming Languages to TensorFlow Compiler




- 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

# Goal: Global improvements to TensorFlow infrastructure

SSA-based designs to generalize and improve ML

- Better side effect modeling and control flow r
- Improve generality of the lowering passes
- Dramatically increase code reuse
- Fix location tracking and other pervasive issues for better user experience

But why stop there?

No reasonable existing answers!

- … and we refuse to copy and paste SSA-based optimizers 6 more times!

# What is MLIR?

A collection of modular and reusable software components that enables the progressive lowering of operations, to efficiently target hardware in a common way

# How is MLIR different?

### State of Art Compiler Technology

MLIR is NOT just a common graph serialization format nor is there anything like it

Shared abstractions spanning languages to machine code

### Modular & Extensible

Progressive abstraction lowering, from graph representation and analysis to code generation

Mix and match representations to fit problem space

### Not opinionated

Choose the level of representation that is right for your problem or target device

We want to enable whole new class of compiler research

# 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

And much more

Google

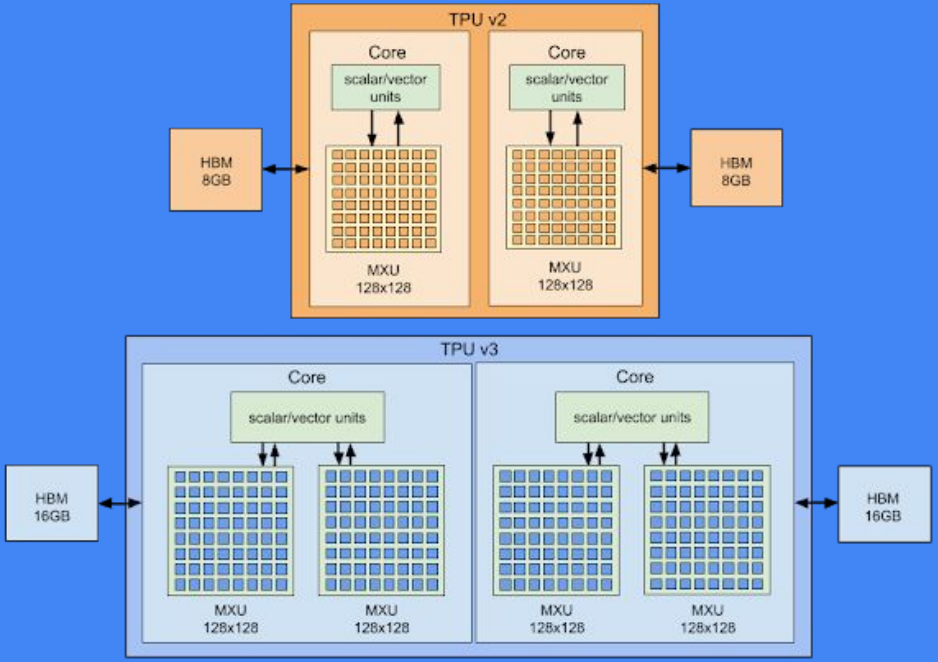
# MLIR Story

1. The right abstraction at the right time
2. Progressive conversion and lowering
3. Extend and reuse
4. Industry standard

Contributed to LLVM (very soon)

*We listen & learn as we go*

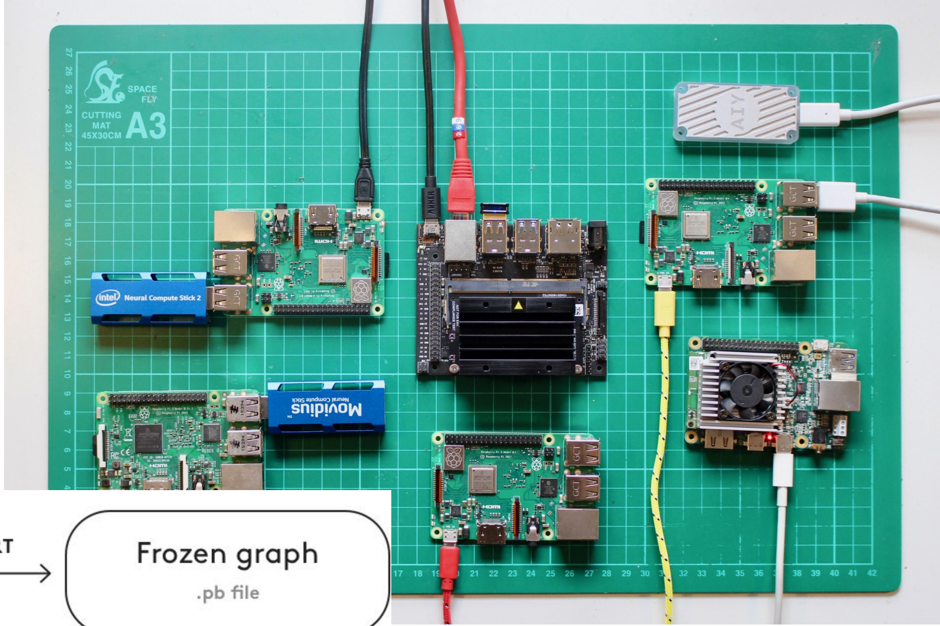
# Focus: Programming Tiled SIMD Hardware





Google

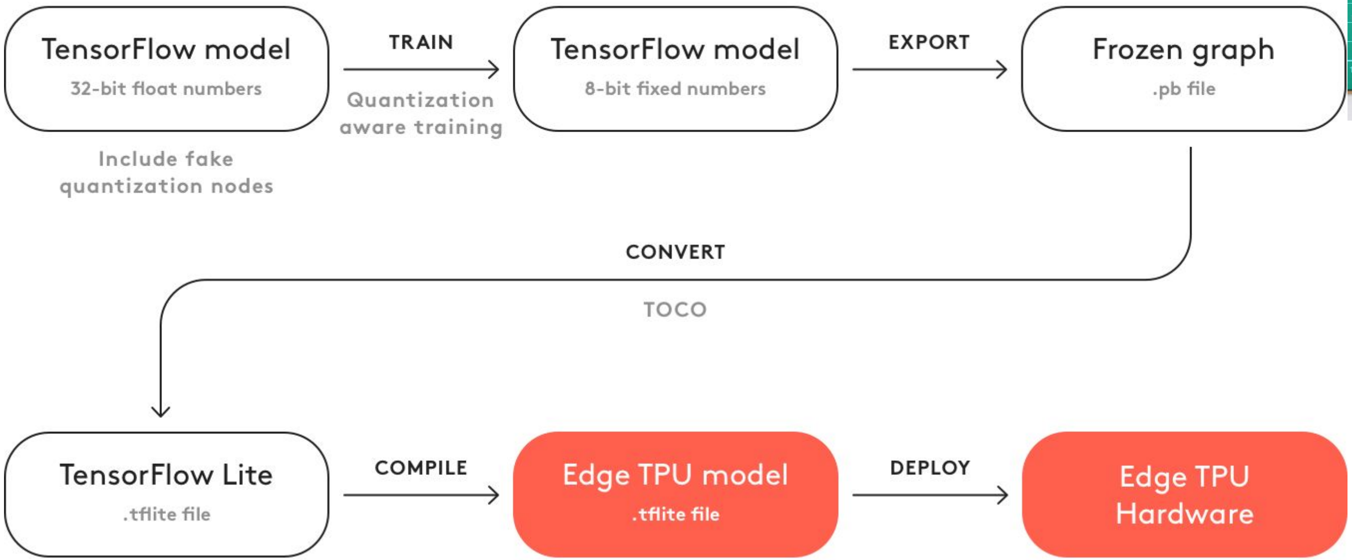
# From Supercomputing to Embedded HPC




Edge and embedded computing zoo

## Highly specialized hardware

**e.g. Google Edge TPU**



# Single Op Compiler

## Tiled and specialized hardware

1. data layout
2. control flow
3. data flow
4. data parallelism

**Examples:** Meta-programming APIs and domain-specific languages (DSLs) for loop transformations

**Halide for image processing pipelines**
**XLA, TVM for neural networks**

https://halide-lang.org https://www.tensorflow.org/xla https://tvm.ai

## TVM example: scan cell (RNN)

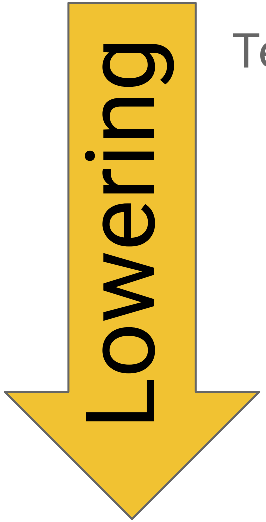
```
m = tvm.var("m")
n = tvm.var("n")
X = tvm.placeholder((m,n), name="X")
s_state = tvm.placeholder((m,n))
s_init = tvm.compute((1,n), lambda _,i: X[0,i])
s_do = tvm.compute((m,n), lambda t,i: s_state[t-1,i] + X[t,i])
s_scan = tvm.scan(s_init, s_do, s_state, inputs=[X])

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))
```

# Tiling... And Beyond?

1. But what about **symbolic bounds, sizes**, **shapes**?
2. **Other transformations**: fusion, fission, pipelining, unrolling...
3. **Composition** & **consistency** with other **transformations**, **mapping** decisions
4. **Code reuse** across compiler flows, code generation frameworks
5. Evaluating **cost** functions, enforcing **resource** constraints

→ Impact on compiler construction,
intermediate representations,
program analyses and transformations

# MLIR's Answer: Extensible Operations Through Dialects




TensorFlow

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

Linalg

```
%m = “linalg.tensor_contraction"(%y, %z)
         {reduction_dim: 1}
     : (memref<?x?x?xf32>, memref<?x?x?xf32>) -> memref<?x?xf32>
```

LLVM IR

```
%f = llvm.add %a, %b
     : !llvm.float
```

Also: TF-Lite, other frontends, other backends…
and much more than a single-op compiler (abstraction, algorithm and code reuse)

Dialects are a modular vehicle for carrying these extensions
and keeping them consistent

Google

# What Dialects Must Comply With and Provide

- MLIR semantics
    - SSA values, block arguments
    - Sequential execution in blocks, control flow through terminator operations
    - Tree of regions, functions and modules as operations
- Single source of truth for dialect-specific objects
    - Operation definition specification
    - Using traits, constraints, legalization actions
    - IR Builders
    - IR verifier
    - Documentation

Google

Essential so that we don’t end up with the XML/JSON of compiler IRs!

# What Dialects May Extend and Customize

- **Types**: `linalg.range`, `llvm.float`, etc.
- **Operations**: `tf.Add`, `tf_executor.graph`, `linalg.view`, `affine.apply`, etc.
- **Attributes**: constants, affine maps, etc.

- Dialect-specific
  - support functions and state
  - canonicalization
  - pretty printer and parser
  - static analyses
  - declarative pattern rewriting
  - passes

# Dialect-Specific Operations, Types, Attributes

- Multiple levels of abstraction in the type and operation definition

Number of values returned

Dialect prefix

Op Id

Argument

Index in the producer's results

List of attributes: named, constant arguments (optional dialect prefix, defaults to op)

```
%res:2 = "mydialect.morph"(%arg#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))
```

Name of the results

Dialect prefix for the type

Opaque string
|
Dialect specific type

Mandatory and rich location

# Dialect-Specific Operations, Types, Attributes

- Multiple levels of abstraction in the type and operation definition
    - Nested regions with control flow, modules, semantic assumptions and guarantees
    - Modules and functions are operations with a nested region (belonging to a builtin dialect)

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

**→ Research: formalization, soundness and equivalence proofs**

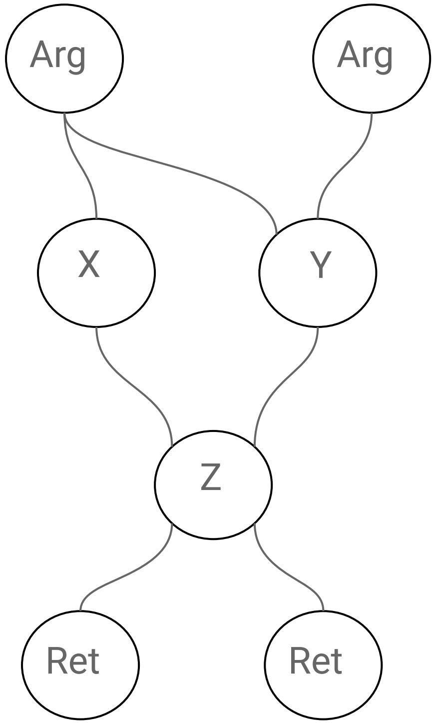
# (Operations→Regions→Blocks)+

```
%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)
```

# 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
- Explicit concurrency in `tf.graph` regions
  - Implicit **futures** for SSA-friendly, asynchronous task parallelism

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

# 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
}
```

# Example: Linalg Dialect

Composition and structural decomposition of linear algebra operations

# Linalg Rationale

Propose a multi-purpose code generation path

- For mixing different styles of compiler transformations
  - 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**
  - Can evolve **beyond affine** loops and data
  - **Optimize across loops and library calls for locality and customization**

# 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
```

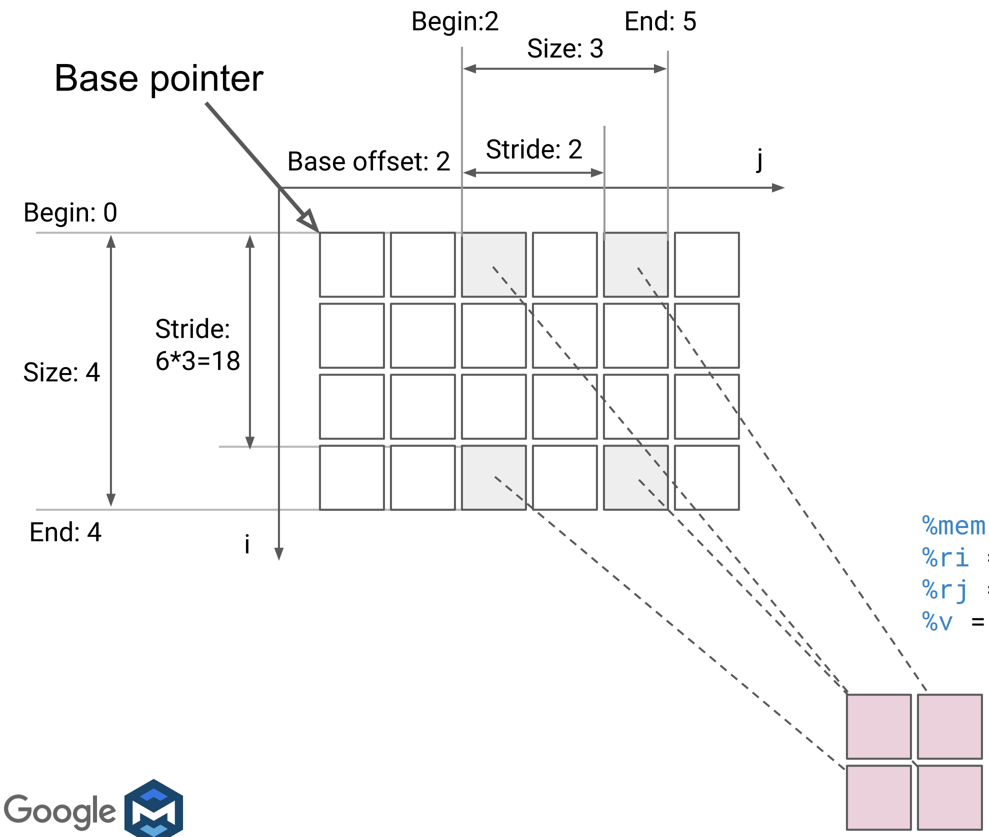
→ for stepping over loop iterations (loop bounds) & data structures

- View type: create an n-d *“indexing”* over a `memref` buffer

```
%8 = linalg.view %7[%r0, %r1] : !linalg.view<?x?xf32>
```

Google

# View Type Descriptor in LLVM IR

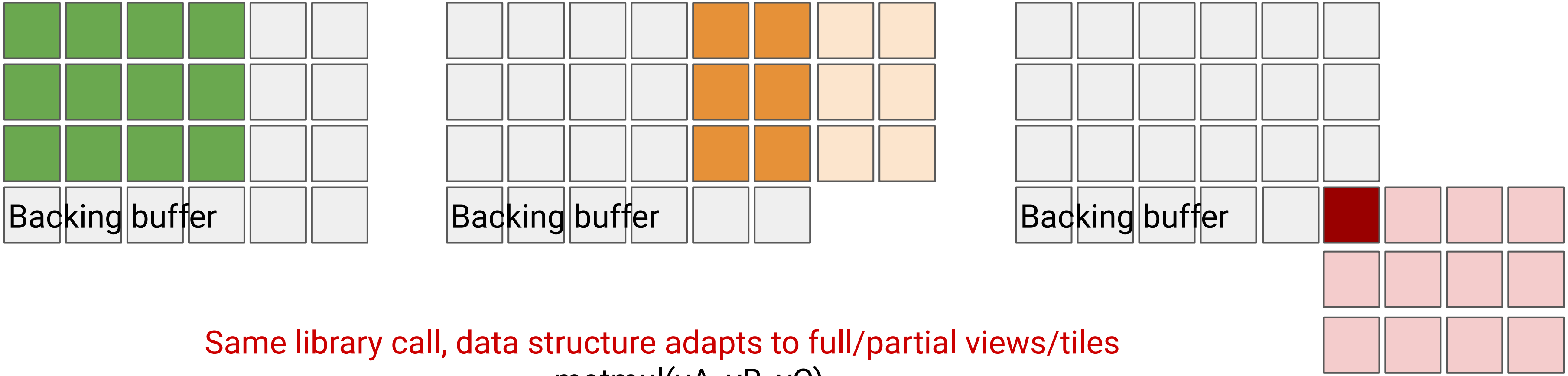



```
{ float*,   # base pointer
  i64,      # base offset
  i64[2]    # sizes
  i64[2] } # strides
```

```
%memref = alloc() : memref<4x6 x f32>
%ri = linalg.range %c2:%c5:%c2 : !linalg.range
%rj = linalg.range %c0:%c4:%c3 : !linalg.range
%v = linalg.view %memref[%ri, %rj] : !linalg.view<?x?xf32>
```

# 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)

Google

# Defining Matmul

- `linalg.matmul` operates on `view<?x?xf32>, view<?x?xf32>, view<?x?xf32>`

```
func @call_linalg_matmul(%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
  %rN = linalg.range %c0:%N:%c1 : !linalg.range
  %rK = linalg.range %c0:%K:%c1 : !linalg.range
  %4 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  %6 = linalg.view %B[%rK, %rN] : !linalg.view<?x?xf32>
  %8 = linalg.view %C[%rM, %rN] : !linalg.view<?x?xf32>
  linalg.matmul(%4, %6, %8) : !linalg.view<?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:%N:%c1 : !linalg.range
  %5 = linalg.view %A[%rM, %rK] : !linalg.view<?x?xf32>
  affine.for %col = 0 to %N {
    %7 = linalg.view %B[%rK, %col] : !linalg.view<?xf32>
    %8 = linalg.view %C[%rM, %col] : !linalg.view<?xf32>
    linalg.matvec(%5, %7, %8) : !linalg.view<?xf32>
  }
  return
}
```

"Interchange" due to library impedance mismatch

# Lowering Between Linalg Ops: Matmul to Matvec

```
// Drop the `j` loop from matmul(i,j,k).
// Parallel dimensions permute.
void linalg::MatmulOp::emitFinerGrainForm()
  auto *op = getOperation();
  ScopedContext scope(FuncBuilder(op), op->getLoc());
  IndexHandle j;
  auto *vA(getInputView(0)), *vB(...), *vC(...);
  Value *range = getViewRootIndexing(vB, 1).first;
  linalg::common::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 the problem goes away

# Loop Tiling

```
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>
  affine.for %i0 = 0 to %M step 8 {
    affine.for %i1 = 0 to %N step 9 {
      affine.for %i2 = 0 to %K {
        affine.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

*tileSizes = {8, 9}*

```
%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 :
%rN = linalg.range %c0:%N:%c1 :
%rK = linalg.range %c0:%K:%c1 :
%4 = linalg.view %A[%rM, %rK] :
%6 = linalg.view %B[%rK, %rN] :
%8 = linalg.view %C[%rM, %rN] :
linalg.matmul(%4, %6, %8) :
```

Google

# Loop Tiling Declaration

- An op "*declares*" how to tile itself maximally on loops
  - For LinalgBase this is easy: perfect loop nests
  - Can be tiled declaratively with **mlir::tile**

```
void linalg::lowerToTiledLoops(mlir::Function *f,
                               ArrayRef<uint64_t> tileSizes) {
  f->walk([tileSizes](Operation *op) {
    if (emitTiledLoops(op, tileSizes).hasValue())
      op->erase();
  });
}
```

Works with imperfectly nested loops + interchange →

```
llvm::Optional<SmallVector<mlir::AffineForOp, 8>>
linalg::emitTiledLoops(Operation *op, ArrayRef<uint64_t> tileSizes) {
  auto loops = emitLoops(op);
  if (loops.hasValue())
    return mlir::tile(*loops, tileSizes, loops->back());
  return llvm::None;
}
```

# View Tiling

```
func @matmul_tiled_views(%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>
  affine.for %i0 = 0 to %M step 8 {
    affine.for %i1 = 0 to %N step 9 {
      %4 = affine.apply (d0) -> (d0 + 8)(%i0)
      %5 = linalg.range %i0:%4:%c1 : !linalg.range    needs range intersection
      %7 = linalg.range %c0:%K:%c1 : !linalg.range
      %8 = linalg.view %A[%5, %7] : !linalg.view<?x?xf32>
      %10 = linalg.range %c0:%M:%c1 : !linalg.range
      %12 = affine.apply (d0) -> (d0 + 9)(%i1)
      %13 = linalg.range %i1:%12:%c1 : !linalg.range    needs range intersection
      %14 = linalg.view %B[%10, %13] : !linalg.view<?x?xf32>
      %15 = linalg.view %C[%5, %13] : !linalg.view<?x?xf32>
      linalg.matmul(%8, %14, %15) : !linalg.view<?x?xf32>
```

Nested linalg.**matmul** call

Google

# Example: Affine Dialect

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

# Affine Dialect for Polyhedral Compilation

```
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
}
```

Custom parsing/printing: an `affine.for` operation with an attached region feels like a regular for loop.

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

```
#set0 = (d0) : (d0 - 1 >= 0, -d0 + 8 >= 0)
func @test() {
  "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> () {
  ^bb1(%i0: index):
    "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> ()
{
    ^bb2(%i1: index):
      "affine.if"(%i0) {condition: #set0} : (index) -> () {
        "foo"(%i0) : (index) -> ()
        "affine.terminator"() : () -> ()
      } { // else block
      }
      "affine.terminator"() : () -> ()
    }
    ...
```

Same code without custom parsing/printing: closer to the internal in-memory representation.

# Affine Dialect for Polyhedral Compilation

- 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

Google

# Stepping Back: Strengths of Polyhedral Compilation

## Decouple intricate optimization problems

### Candidates

Partially Specified Implementations

- Optimizations and lowering, choices and transformations
  *e.g., tile? unroll? ordering?*
- Generate imperative code, calls to native libraries
  *infer buffers, control flow*

### Constraints

Functional Semantics and Resource Modeling

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

### Search

Optimization Algorithms

- 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...*

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

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

Domain $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \\ \quad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right.$

Sequence
- Filter $\{\mathsf{S}(i,j)\}$
  - Band $\{\mathsf{S}(i,j) \to (i,j)\}$
- Filter $\{\mathsf{T}(i,j,k)\}$
  - Band $\{\mathsf{T}(i,j,k) \to (i,j,k)\}$

**(a)** canonical sgemm

Domain $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right.$

Context $\{N = M = 16 \wedge K > 1000\}$

Band $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \to (i,j)\} \\ \{\mathsf{T}(i,j,k) \to (i,j)\} \end{array} \right.$

Sequence
- Filter $\{\mathsf{S}(i,j)\}$
- Filter $\{\mathsf{T}(i,j,k)\}$
  - Band $\{\mathsf{T}(i,j,k) \to (k)\}$

**(b)** fused

Domain $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right.$

Context $\{N = M = K = 512 \wedge 0 \le b_x, b_y < 32 \wedge 0 \le t_x, t_y < 16\}$

Filter $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \mid i - 32b_x - 31 \le 32 \times 16\lfloor i/32/16 \rfloor \le i - 32b_x \wedge \\ \quad j - 32b_y - 31 \le 32 \times 16\lfloor j/32/16 \rfloor \le j - 32b_y\} \\ \{\mathsf{T}(i,j,k) \mid i - 32b_x - 31 \le 32 \times 16\lfloor i/32/16 \rfloor \le i - 32b_x \wedge \\ \quad j - 32b_y - 31 \le 32 \times 16\lfloor j/32/16 \rfloor \le j - 32b_y\} \end{array} \right.$

Band $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathsf{T}(i,j,k) \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array} \right.$

Sequence
- Filter $\{\mathsf{S}(i,j)\}$
  - Filter $\{\mathsf{S}(i,j) \mid (t_x - i) = 0 \bmod 16 \wedge (t_y - j) = 0 \bmod 16\}$
    - Band $\{\mathsf{S}(i,j) \to (i \bmod 32, j \bmod 32)\}$
- Filter $\{\mathsf{T}(i,j,k)\}$
  - Band $\{\mathsf{T}(i,j,k) \to (32\lfloor k/32 \rfloor)\}$
    - Band $\{\mathsf{T}(i,j,k) \to (k \bmod 32)\}$
      - Filter $\{\mathsf{T}(i,j,k) \mid (t_x - i) = 0 \bmod 16 \wedge (t_y - j) = 0 \bmod 16\}$
        - Band $\{\mathsf{T}(i,j,k) \to (i \bmod 32, j \bmod 32)\}$

**(e)** fused, tiled, sunk and mapped

Domain $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \\ \quad \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right.$

Band $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathsf{T}(i,j,k) \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array} \right.$

Band $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \to (i \bmod 32, j \bmod 32)\} \\ \{\mathsf{T}(i,j,k) \to (i \bmod 32, j \bmod 32)\} \end{array} \right.$

Sequence
- Filter $\{\mathsf{S}(i,j)\}$
- Filter $\{\mathsf{T}(i,j,k)\}$
  - Band $\{\mathsf{T}(i,j,k) \to (k)\}$

**(c)** fused and tiled

Domain $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \mid 0 \le i < N \wedge 0 \le j < K\} \\ \{\mathsf{T}(i,j,k) \mid 0 \le i < N \wedge 0 \le j < K \wedge 0 \le k < M\} \end{array} \right.$

Band $\left[ \begin{array}{l} \{\mathsf{S}(i,j) \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \\ \{\mathsf{T}(i,j,k) \to (32\lfloor i/32 \rfloor, 32\lfloor j/32 \rfloor)\} \end{array} \right.$

Sequence
- Filter $\{\mathsf{S}(i,j)\}$
  - Band $\{\mathsf{S}(i,j) \to (i \bmod 32, j \bmod 32)\}$
- Filter $\{\mathsf{T}(i,j,k)\}$
  - Band $\{\mathsf{T}(i,j,k) \to (32\lfloor k/32 \rfloor)\}$
    - Band $\{\mathsf{T}(i,j,k) \to (k \bmod 32)\}$
      - Band{T(i, j, k0 → (i mod 32, j mod :

**(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

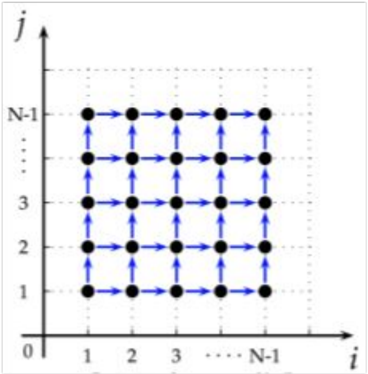
  Ohio State University, Colorado State University, Rice University

  Google Summer of Code

# Candidates?
# Representing Partially Specified Programs

- Polyhedral compilation
  - Affine scheduling
    *optimization: often ILP-based*
  - Code generation
    *from affine schedules to nested loops*

- Meta-programming array processing code
  - Halide / TVM specific combinators
    and scheduling/mapping primitives
  - Polyhedral: URUK, CHiLL
    with automatic schedule completion




**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_update = tvm.compute((m,n), lambda t, i: s_state[t-1,i] +
X[t,i])
s_scan = tvm.scan(s_init, s_update, 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_update].split(s_update.op.axis[1], factor=num_thread)
s[s_update].bind(xo, block_x)
s[s_update].bind(xi, thread_x)
print(tvm.lower(s, [X, s_scan], simple_mode=True))
```

# Constraints?
# Functional Correctness and Resource Usage

- Polyhedral compilation
  - Model data dependences, data flow, memory accesses and footprint at compile time, symbolically
  - Beyond scalar data flow: **symbolic affine expressions on indexing/iterations**
- Program synthesis
  - Start from denotational specification, possibly partial (sketching), or (counter-)examples
  - Guess possible implementations by (guided) sampling lots of random ones
    *Or guess efficient implementations by (guided) sampling lots of stupid ones*
  - Filter correct implementations using SMT solver or theorem prover
    *Model both correctness and hardware mapping*
- Superoptimization
  - Typically on basic blocks, with SAT solver or theorem prover and search
  - Architecture and performance modeling, e.g., EXEgesis

# Search?
# Inspired From Adaptive Libraries and Autotuning

- Feedback-directed and iterative compiler optimization, lots of work since the late 90s
- Adaptive libraries
  - SPIRAL: *Domain-Specific Language (DSL) + Rewrite Rules + Multi-Armed Bandit or MCTS* http://www.spiral.net
  - ATLAS, FFTW, etc.: *hand-written fixed-size kernels + micro-benchmarks + meta-heuristics*
- Polyhedral compilation
  - Traditionally based on Integer Linear Programming (ILP)
  - Pouchet et al. (affine), Park et al. (affine and CFG): *Genetic Algorithm, SVM, Graph Kernels*

Google

# 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**

⟶ support **tiling on dynamic shapes**

⟶ support **shifting/pipelining**

⟶ **transformation composition is key**

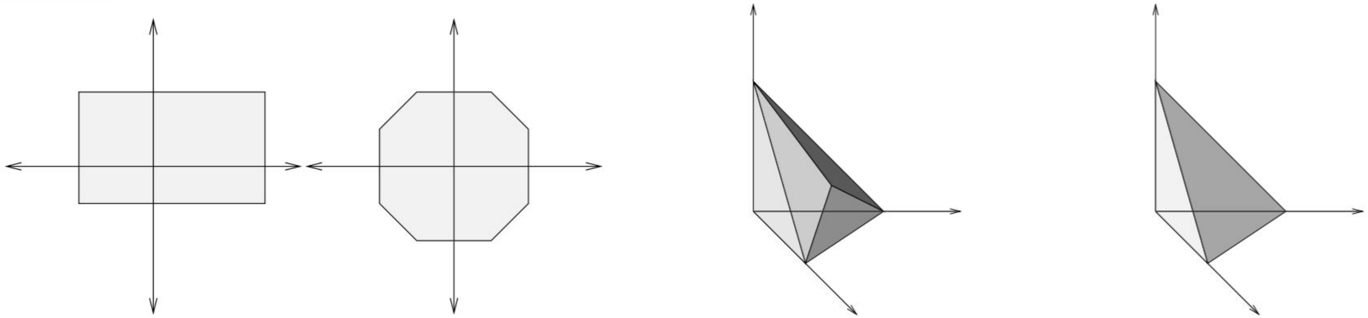
# MLIR’s Research Proposal for a Polyhedral-Lite Framework

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 strip-mining and 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

# (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 | |
| $a \leq x_i \leq b$ | $\pm x_i \pm x_j \leq c$ | $ax_i + bx_j \leq c$ | $\sum a_i x_i \leq c$ |

$$\xrightarrow{\text{Precision}}$$

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

$$\xrightarrow[\text{Complexity}]{}$$

See Upadrasta's thesis, POPL 2013

MLIR

# MLIR in a Nutshell

MLIR is a powerful infrastructure for

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

**Check out github, mailing list, stay tuned for further announcements**

**Workshops: LCPC MLIR4HPC HiPEAC AccML CGO C4ML**