



TensorFlow Graph Optimizations

Rasmus Munk Larsen
rmlarsen@google.com

Tatiana Shpeisman
shpeisman@google.com

Presenting the work of many people at Google & open source contributors



TensorFlow



<http://tensorflow.org/>

and

<https://github.com/tensorflow/tensorflow>

Open, standard software for
general machine learning

Great for Deep Learning in
particular

First released Nov 2015

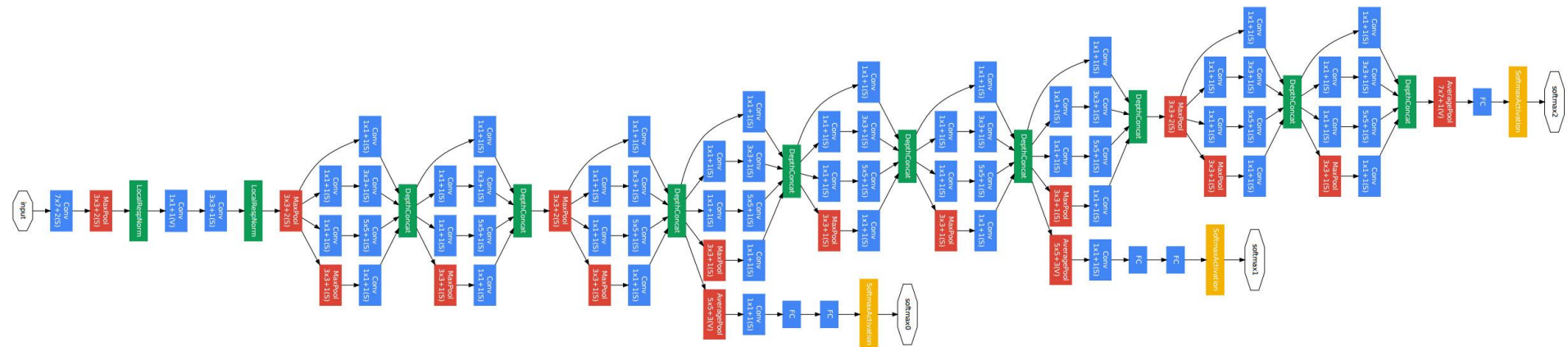
Apache 2.0 license

Powers many Google products

TensorFlow Graph concepts

- TensorFlow (v1.x) programs generate a **DataFlow** (directed, multi-) **Graph**
 - Device independent intermediate program representation
 - TensorFlow v2.x uses a mix of imperative (**Eager**) execution mode and graphs functions
- Graph **nodes** represent operations “**Ops**” (**Add, MatMul, Conv2D, ...**)
 - Abstract device-, execution backend-, and language independent API
 - Implemented by **Op Kernels** written in C++, specialized on <Type, Device>
- Graph **edges** represent “data” flowing between ops
 - **Tensors** (ref-counted, n-dimensional array buffers in device memory)
 - **Control dependencies**: **A->B** means **A** must finish before **B** can run
 - **Resource handles** to state (e.g. variables, input data pipelines)

Graph example: The Inception Architecture (2014)

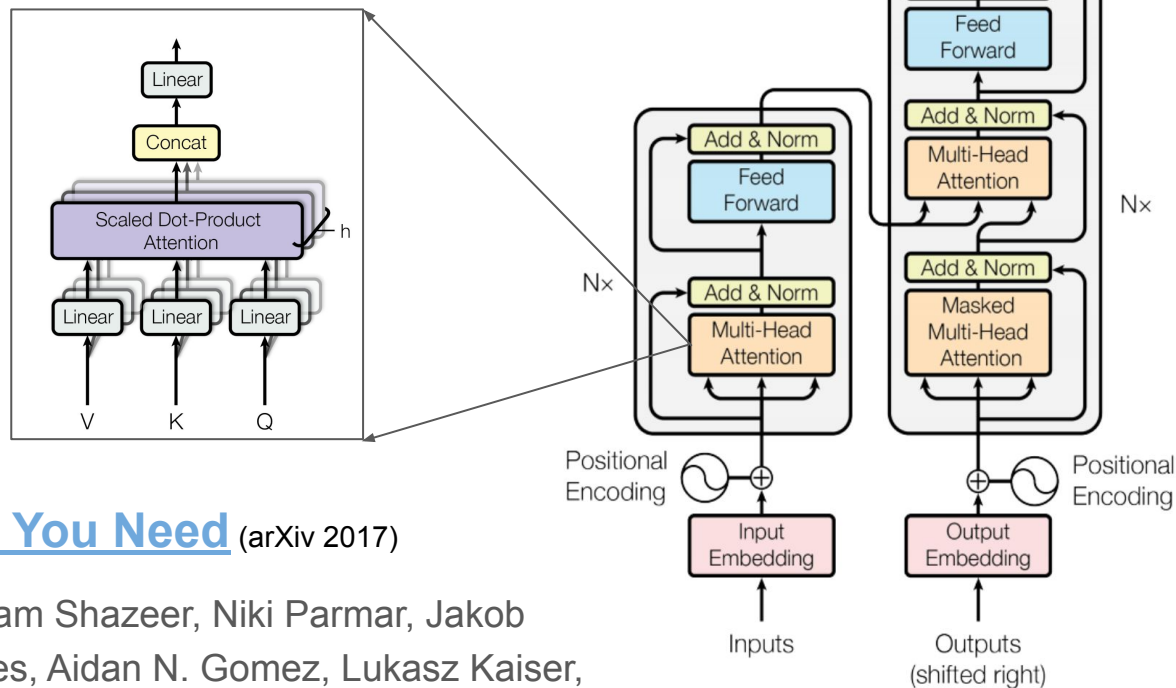


Going Deeper with Convolutions

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich



Graph example: The Transformer



[Attention Is All You Need](#) (arXiv 2017)

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

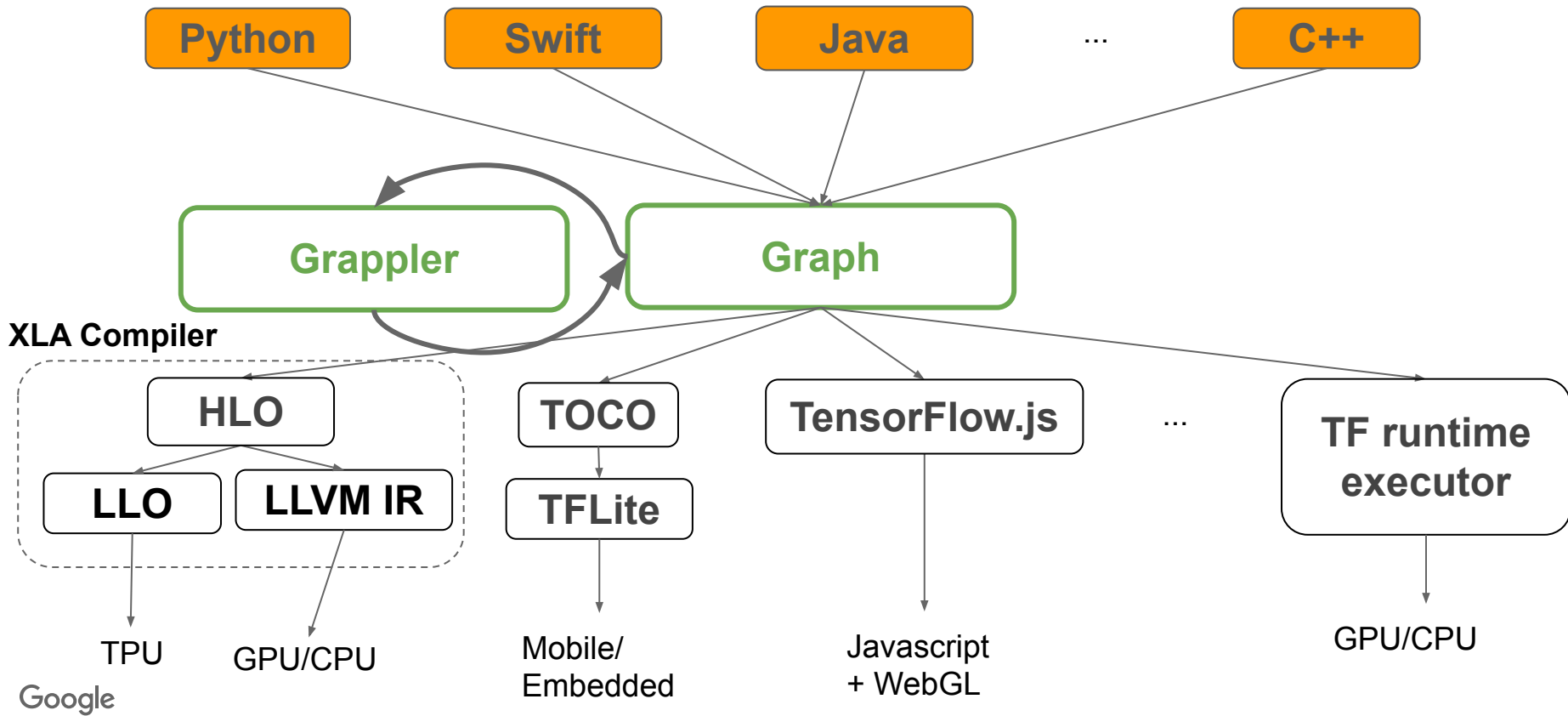


Grappler

Grappler: Grappling with TF Graphs

- Grappler: Default graph optimization system in the TF runtime
 - Re-writes graphs to improve out-of-the-box TensorFlow performance
 - Provides a plugin infrastructure to register custom optimizers/rewriters
- Main goals:
 - **Automatically improve TF performance through graph simplifications & high-level optimizations that benefit most target HW architectures (CPU/GPU/TPU/mobile etc.)**
 - **Reduce device peak memory usage to enable larger models to run**
 - **Improve hardware utilization by optimizing the mapping of graph nodes to compute resources**
- Provides cost models to drive optimization and help diagnose model performance

Grappler: TensorFlow Context



Why transformations at the graph level?

- **Pros:**

- Many optimizations can be easier to discover and express as high-level graph transformations
 - Example: **Matmul(Transpose(x), y) => Matmul(x,y, transpose_x=True)**
- Graph is backend independent (TF runtime, XLA, TensorRT, TensorFlow.js, ...)
- Interoperable with TensorFlow supported languages (protocol buffer format)
- Optimizations can be applied at **runtime** or **offline** using our standalone tool
- Lots of existing models (TF Hub, Google production models) available for learning
- Pragmatic: Helps the most existing TensorFlow users get better “out-of-the-box” performance

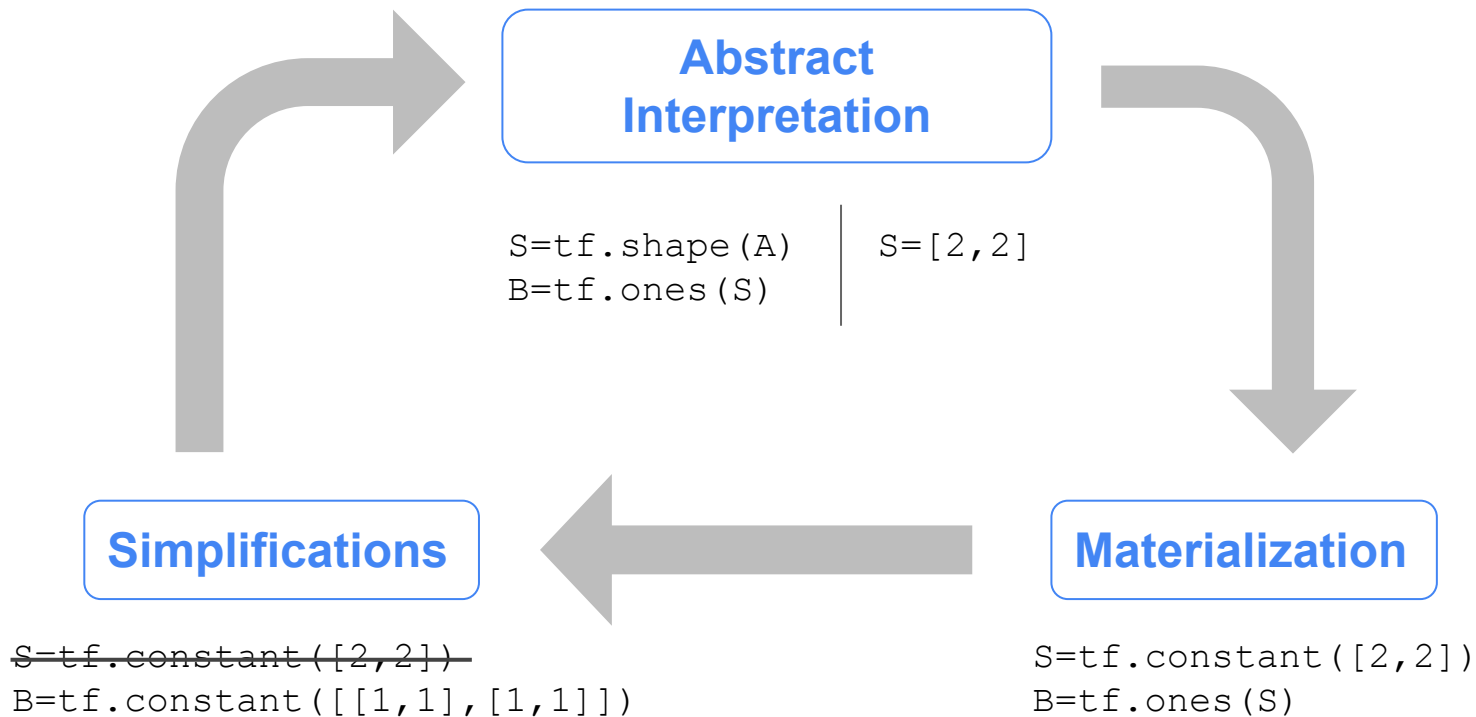
- **Cons:**

- Rewrites can be tricky to implement correctly, because of loosely defined graph semantics
 - In-place ops, side-effects, control flow, control dependencies
- Protocol buffer dependence increases binary size
- Currently requires extra graph format conversions in TF runtime

Examples of Graph Simplifications

- Graph minimization and canonicalization
 - Redundant computation removal through constant folding, CSE, redundant control edge removal by transitive reduction on graph
 - Whole graph analysis to identify and remove hidden identity and other unnecessary ops (e.g. shuffling a Tensor of size 1 or reductions along empty set of dimensions are identity ops)
- Algebraic simplifications
 - Take advantage of commutativity, associativity, and distributivity to simplify computations
 - Example: $A + 2*B + 2*C + \text{Identity}(A) \Rightarrow 2*A + 2*B + 2*C \Rightarrow 2*\text{AddN}(A, B, C)$
- Synergy: Each optimization builds upon the previous ones
- Graph optimizers at <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/core/grappler/optimizers>

Graph Simplifications



MetaOptimizer

- Top-level driver invoked by runtime or [standalone tool](#)
- Controlled by [RewriterConfig](#) in TF [Config](#)
- Runs multiple sub-optimizers in a loop: (* = not on by default):

```
i = 0
while i < config.meta_optimizer_iterations (default=2):
    Pruning()           # Remove nodes not in fanin of outputs, unused functions
    Function()          # Function specialization & inlining, symbolic gradient inlining
    DebugStripper() *   # Remove assert, print, check_numerics
    ConstFold()         # Constant folding and materialization
    Shape()             # Symbolic shape arithmetic
    Remapper()          # Op fusion
    Arithmetic()        # Node deduping (CSE) & arithmetic simplification
    if i==0: Layout()   # Layout optimization for GPU
    if i==0: Memory()   # Swap-out/Swap-in, Recompute*, split large nodes
    Loop()              # Loop Invariant Node Motion*, Stack Push & Dead Node Elimination
    Dependency()        # Prune/optimize control edges, NoOp/Identity node pruning
    Custom()            # Run registered custom optimizers (e.g. TensorRT)
    i += 1
```

Constant folding optimizer

```
do:
    InferShapesStatically() # Fixed-point iteration with symbolic shapes
    graph_changed = MaterializeConstants() # grad broadcast, reduction dims
    q = NodesWithKnownInputs()
    while not q.empty():
        node = q.pop()
        graph_changed |= FoldGraph(node, &q) # Evaluate node on host
    graph_changed |= SimplifyGraph()
while graph_changed
```

Constant folding optimizer: `SimplifyGraph()`

- Removes trivial ops, e.g. identity `Reshape`, `Transpose` of 1-d tensors, `Slice(x) = x`, etc.
- Rewrites that enable further constant folding, e.g.
 - Constant propagation through `Enter`
 - `Switch(pred=x, value=x) => propagate False through port0, True through port1`
 - Partial constant propagation through `IdentityN`
- Arithmetic rewrites that rely on known shapes or inputs, e.g.
 - Constant push-down:
 - `Add(c1, Add(x, c2)) => Add(x, c1 + c2)`
 - `ConvND(c1 * x, c2) => ConvND(x, c1 * c2)`
 - Partial constfold:
 - `AddN(c1, x, c2, y) => AddN(c1 + c2, x, y)`,
 - `Concat([x, c1, c2, y]) = Concat([x, Concat([c1, c2]), y)`
 - Operations with neutral & absorbing elements:
 - `x * Ones(s) => Identity(x)`, if `shape(x) == output_shape`
 - `x * Ones(s) => BroadcastTo(x, Shape(s))`, if `shape(s) == output_shape`
 - Same for `x + Zeros(s)`, `x / Ones(s)`, `x * Zeros(s)` etc.
 - `Zeros(s) - y => Neg(y)`, if `shape(y) == output_shape`
 - `Ones(s) / y => Recip(y)` if `shape(y) == output_shape`

Arithmetic optimizer

1. Node deduplication (common subexpression elimination)
2. Arithmetic simplifications & optimizations

```
DedupComputations():
    do:
        stop = true
        UniqueNodes reps
        for node in graph.nodes():
            rep = reps.FindOrInsert(node, IsCommutative(node))
            if rep == node or !SafeToDedup(node, rep):
                continue
            for fanout in node.fanout():
                ReplaceInputs(fanout, node, rep)
            stop = false
        while !stop
```

Arithmetic optimizer:

- Arithmetic simplifications

- Flattening: $a+b+c+d \Rightarrow \text{AddN}(a, b, c, d)$
- Hoisting: $\text{AddN}(x * a, b * x, x * c) \Rightarrow x * \text{AddN}(a+b+c)$
- Simplification to reduce number of nodes:
 - Numeric: $x+x+x \Rightarrow 3*x$
 - Logic: $!(x > y) \Rightarrow x \leq y$

- Broadcast minimization

- Example: $(\text{matrix1} + \text{scalar1}) + (\text{matrix2} + \text{scalar2}) \Rightarrow (\text{matrix1} + \text{matrix2}) + (\text{scalar1} + \text{scalar2})$

- Better use of intrinsics

- $\text{Matmul}(\text{Transpose}(x), y) \Rightarrow \text{Matmul}(x, y, \text{transpose_x}=\text{True})$

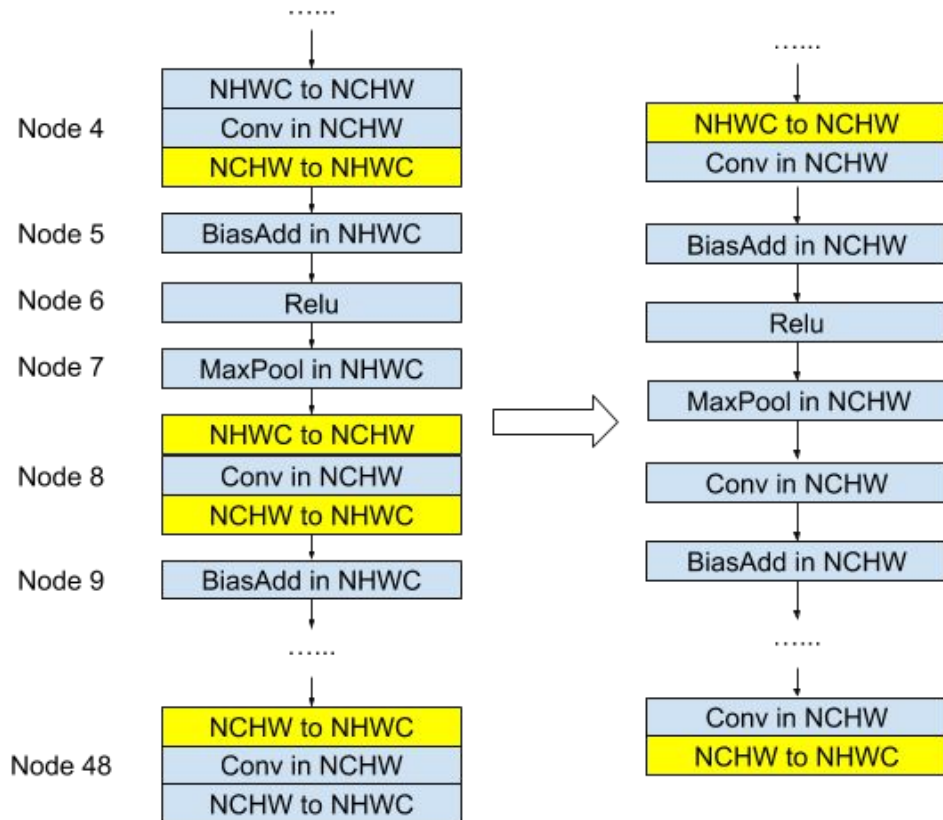
- Remove redundant ops or op pairs

- $\text{Transpose}(\text{Transpose}(x, \text{perm}), \text{inverse_perm})$
- $\text{BitCast}(\text{BitCast}(x, \text{dtype1}), \text{dtype2}) \Rightarrow \text{BitCast}(x, \text{dtype2})$
- Pairs of elementwise involutions $f(f(x)) \Rightarrow x$ ([Neg](#), [Conj](#), [Reciprocal](#), [LogicalNot](#))
- Repeated Idempotent ops $f(f(x)) \Rightarrow f(x)$ ([DeepCopy](#), [Identity](#), [CheckNumerics...](#))

- Hoist chains of unary ops at [Concat](#)/[Split](#)/[SplitV](#)

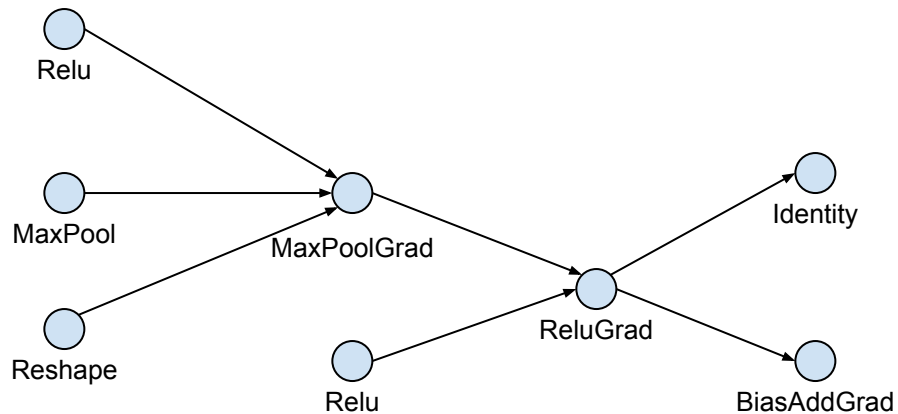
- $\text{Concat}([\text{Exp}(\text{Cos}(x)), \text{Exp}(\text{Cos}(y)), \text{Exp}(\text{Cos}(z))]) \Rightarrow \text{Exp}(\text{Cos}(\text{Concat}([x, y, z])))$
- $[\text{Exp}(\text{Cos}(y)) \text{ for } y \text{ in } \text{Split}(x)] \Rightarrow \text{Split}(\text{Exp}(\text{Cos}(x)), \text{num_splits})$

Layout optimizer



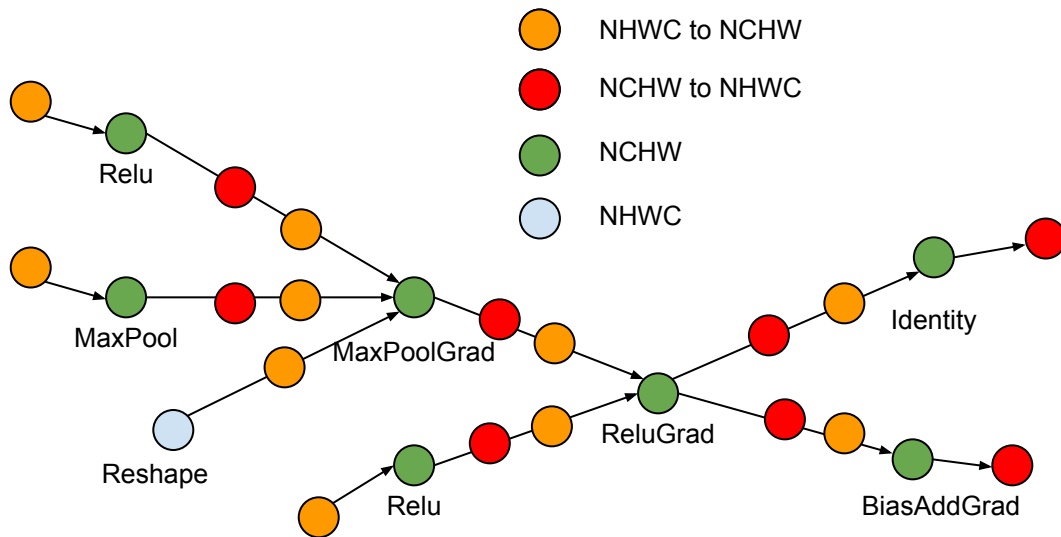
Layout optimizer

Example: Original graph with all ops in NHWC format



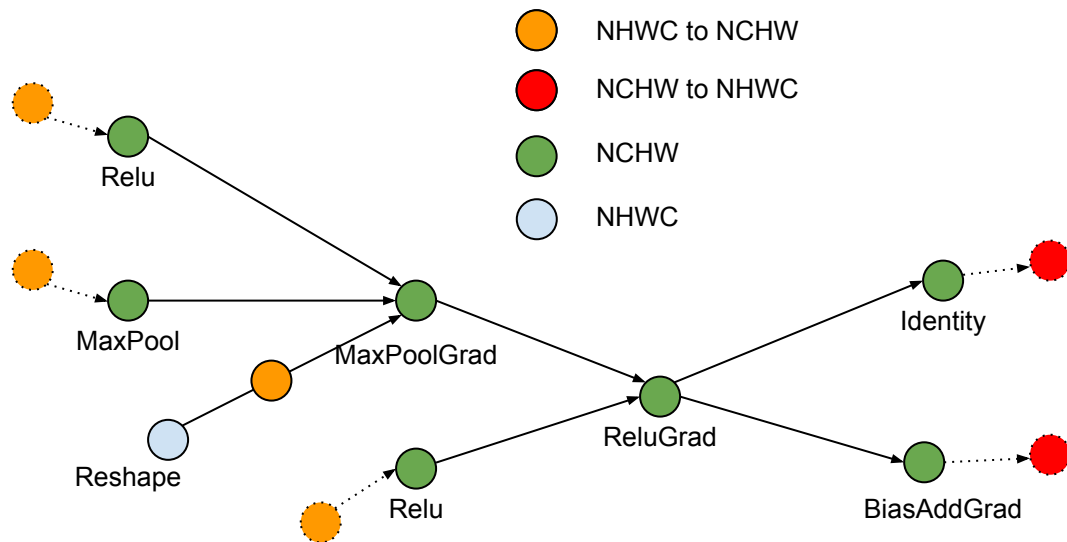
Layout optimizer

Phase 1: Expand by inserting conversion pairs



Layout optimizer

Phase 2: Collapse adjacent conversion pairs



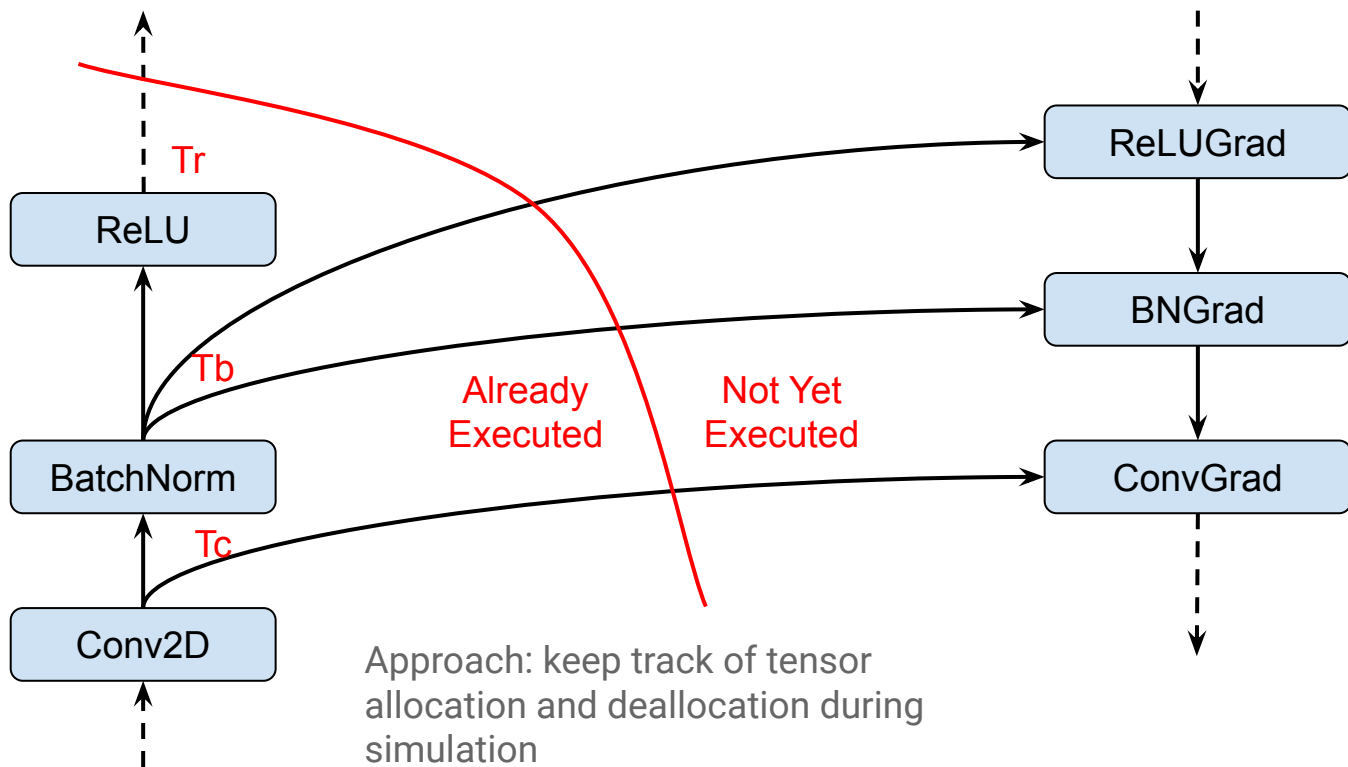
Remapper optimizer: Op fusion

- Replaces commonly occurring subgraphs with optimized fused “monolithic” kernels
 - Examples of patterns fused:
 - Conv2D + BiasAdd + <Activation>
 - Conv2D + FusedBatchNorm + <Activation>
 - Conv2D + Squeeze + BiasAdd
 - MatMul + BiasAdd + <Activation>
- Fusing ops together provides several performance advantages:
 - Completely eliminates Op scheduling overhead (big win for cheap ops)
 - Increases opportunities for ILP, vectorization etc.
 - Improves temporal and spatial locality of data access
 - E.g. MatMul is computed block-wise and bias and activation function can be applied while data is still “hot” in cache.
- A separate mechanism allows the TensorFlow compiler to cluster subgraphs and generate fused kernel code on-the-fly

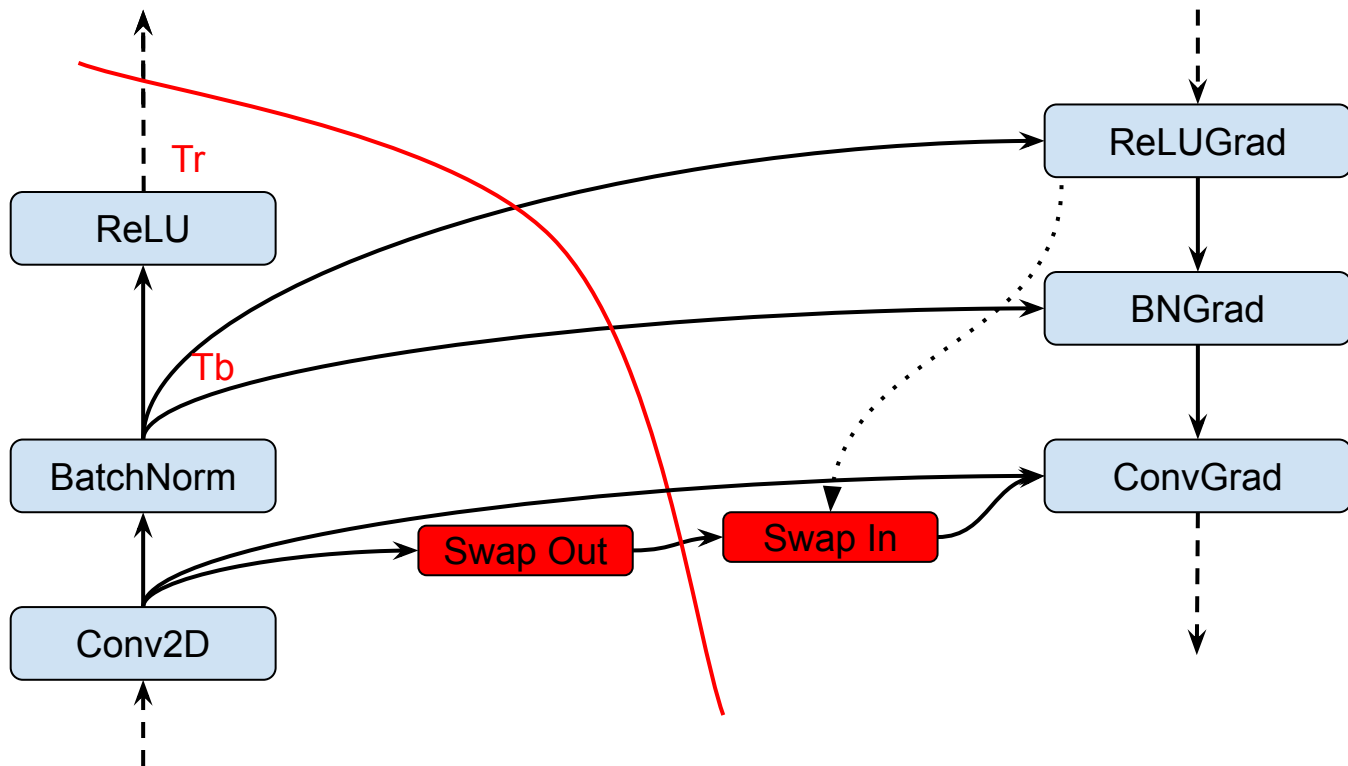
Memory optimizer

- Memory optimization based on abstract interpretation
 - Swap-out / Swap-in optimization
 - Reduces device memory usage by swapping to host memory
 - Uses memory cost model to estimate peak memory
 - Uses op cost model to schedule Swap-In at (roughly) the right time
 - Enables models for Waymo, Cerebra mobilenet
 - Recomputation optimization (not on by default)
- Rewrites large aggregation nodes to fit in device memory
- Allocator constraint relaxation
 - Fixes 2x memory peak bug and removes explicit copy in AssignOp
 - Adds more opportunities for *buffer forwarding* in TF runtime

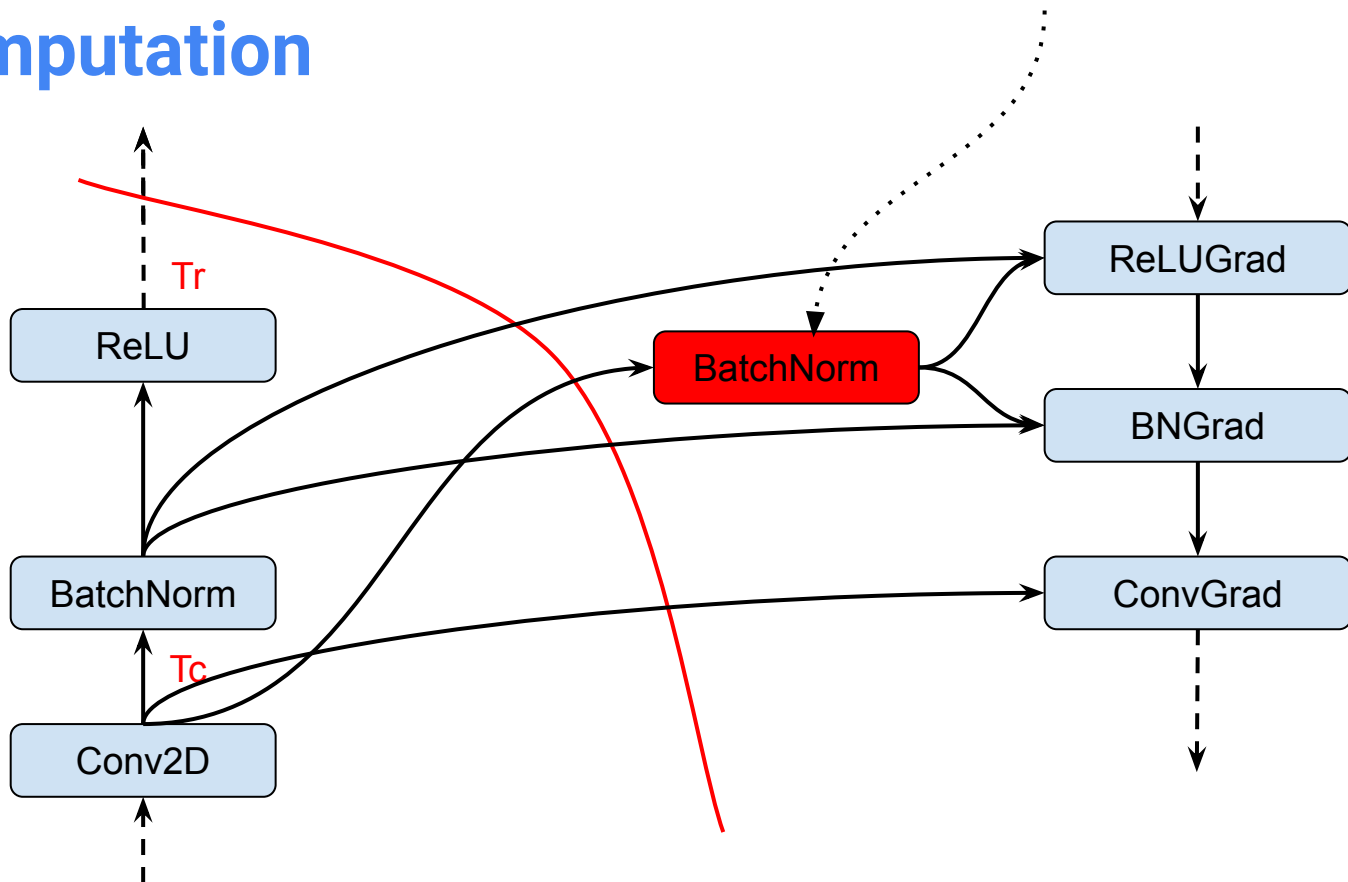
Peak Memory Characterization



Google



Recomputation



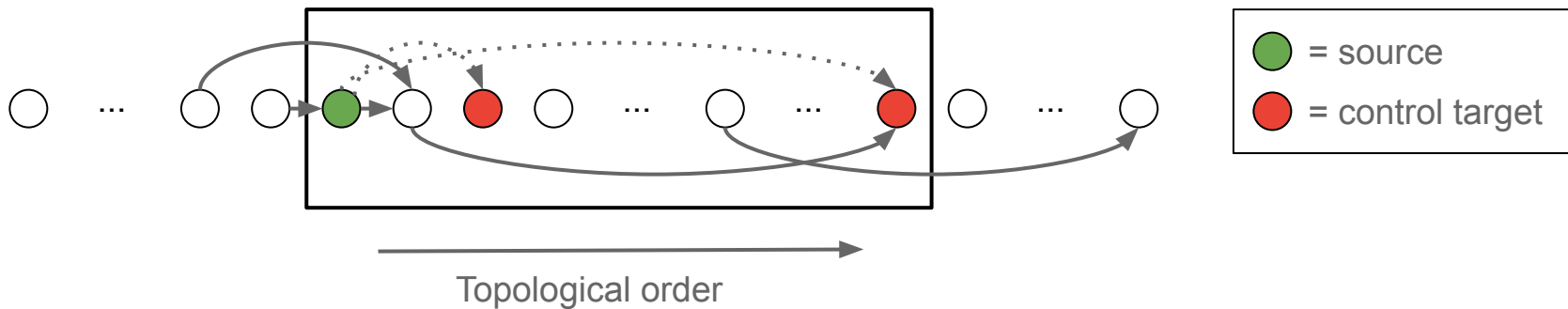
Control Flow Optimizer

- Loop Invariant Node Motion
 - Contributed by Alibaba TensorFlow team
 - Hoists loop-invariant subgraphs out of loops
 - Not enabled by default
- StackPush removal
 - Remove StackPushes without consumers
 - No matching StackPop or matching StackPop with no consumers
- Dead Branch Elimination for Switch with constant predicate
- Deduce loop trip count statically
 - Remove loop for zero trip count
 - Remove control flow nodes for trip count == 1

Dependency Optimizer

1. Whole-graph optimization: Removal of redundant control edges through *transitive reduction*
2. Conversion of nodes bypassed by other optimizations to NoOp
3. Pruning of NoOp and Identity nodes
4. Consolidation of cross-device control edges

Dependency Optimizer: Transitive Reduction



A control edge is redundant iff there exists a path of length > 1 from ● to ●.

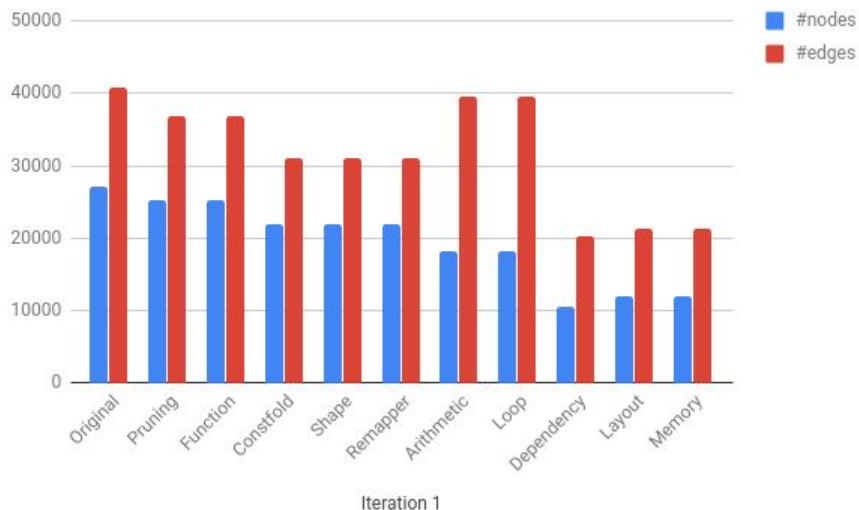
Algorithm:

1. Sort nodes topologically after removing back-edges in loops
2. For each ●:
 - a. Compute longest paths in DAG for nodes up to $\max(\text{topo_index}(\text{red node}))$
 - b. Discard control edges to ● with distance > 1

Step 2 has $O(N \cdot (M + N))$ worst case complexity. Very fast in practice: **26ms** on InceptionV3.

Grappler: Performance Results

Results: InceptionV3

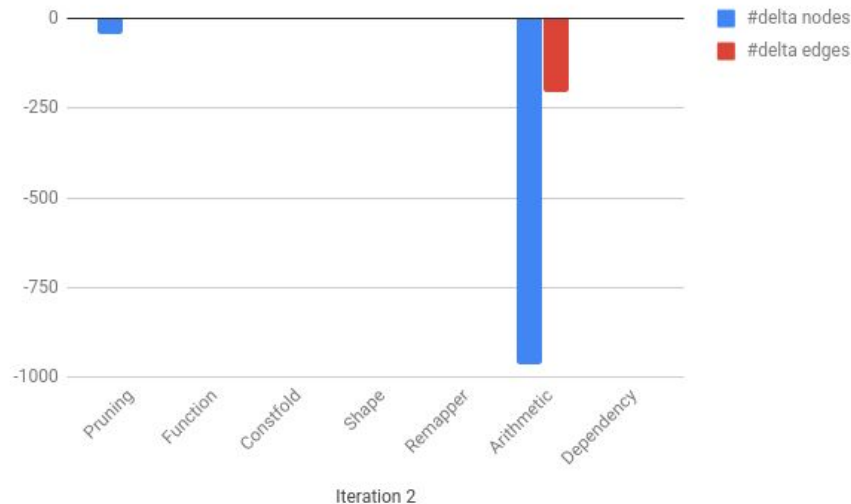
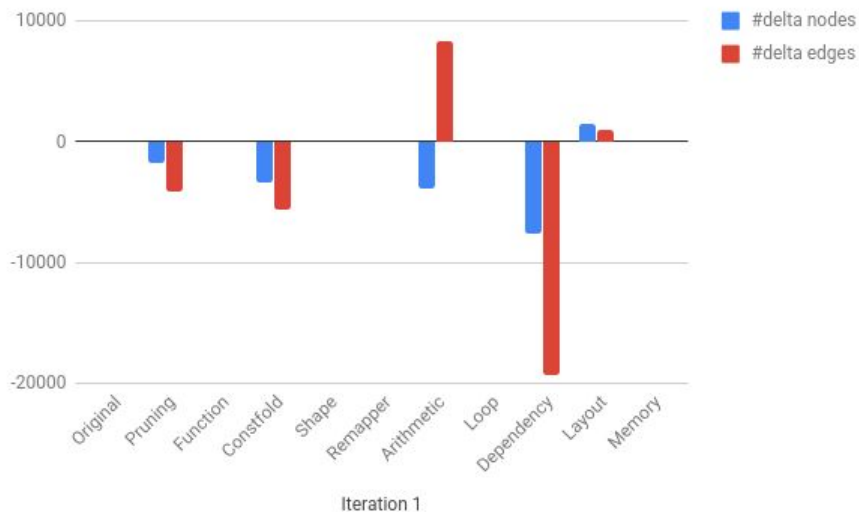


	Nodes	Edges
Iteration 1	-55.9%	-48.0%
Iteration 2	-3.7%	-0.5%
Total	-59.6%	-48.6%

Performance gains:

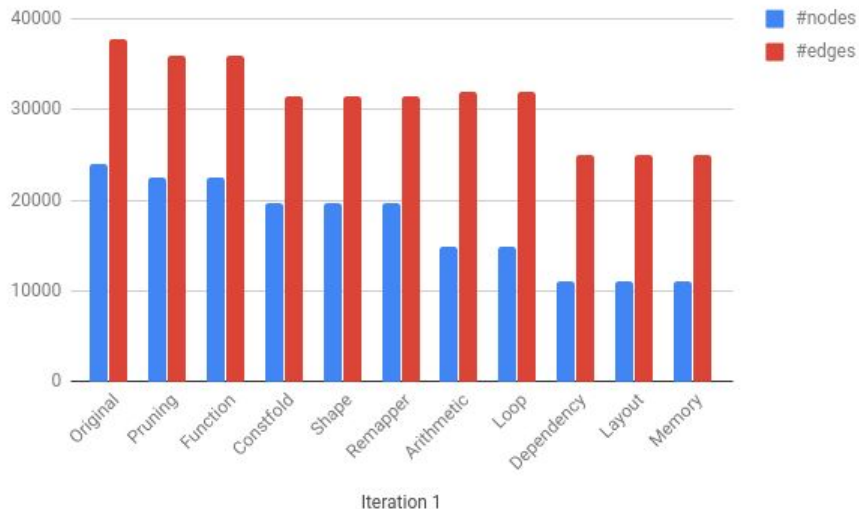
- 43% step time reduction w/o fused batch norm
- 9% step time reduction w/o fused batch norm
- 26% step time reduction w/ fused batch norm
- No significant gains on CPU w/ fused batch norm

Results: InceptionV3 graph size reduction



Note: The arithmetic optimizer often temporarily grows the graph by leaving behind by-passed nodes with only control outputs. They are subsequently pruned by the dependency optimizer.

Results: Transformer seq2seq model

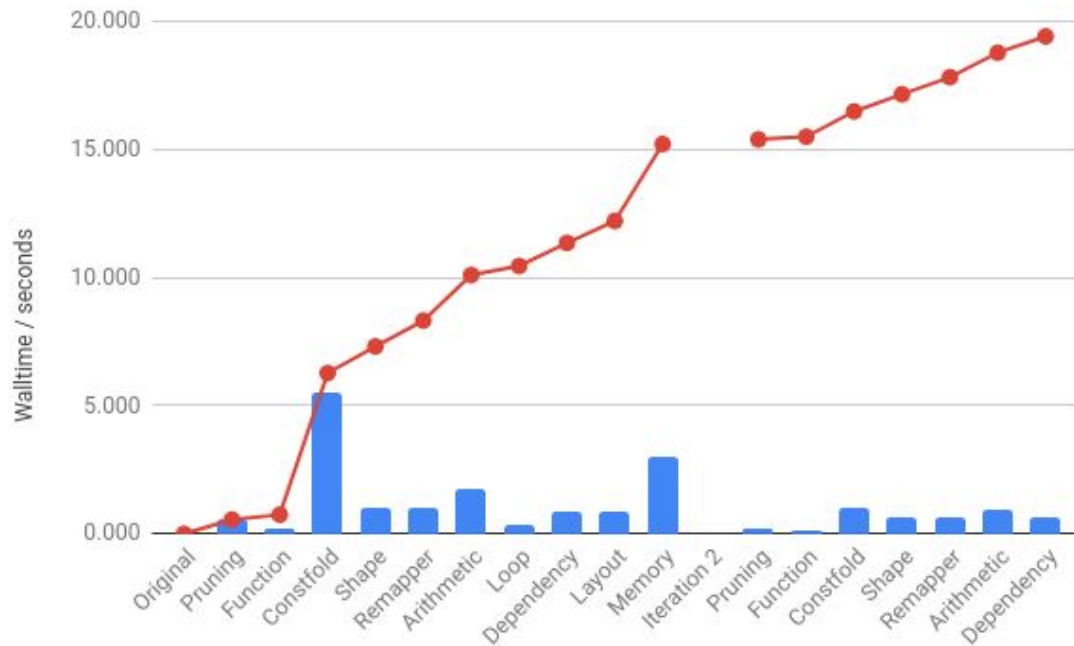


	Nodes	Edges
Iteration 1	-53.5%	-34.0%
Iteration 2	-1.4%	-1.3%
Total	-54.9%	-35.2%

Performance gains:

- 17.5% step time reduction on GPU
- 16.2% step time reduction on CPU

Results: Grappler runtime on Transformer model



	Walltime
Iteration 1	15.6s
Iteration 2	5.9s
Total	21.5s

Results: TensorFlow.js inference

- Inference in Javascript with WebGL acceleration
- Grappler optimizations improve
 - Graph size
 - Inference speed
 - Time needed for kernel compilation

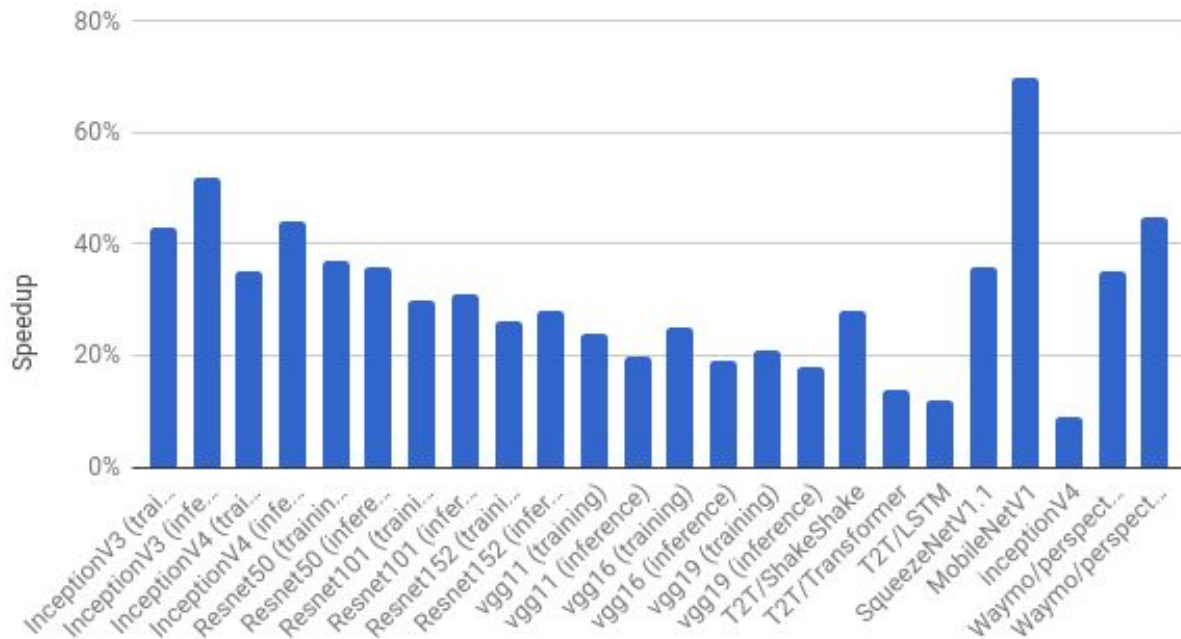
	Size reduction (#nodes)	Compilation time	Inference time
SqueezeNetV1.1	9.6% (177->160)	0.0% (800ms)	26.3% (95ms->70ms)
MobileNetV1	64.1% (555->199)	11.1% (900ms->800ms)	41.2% (85ms->50ms)
InceptionV4	58.1% (2613->1096)	52.0% (5000ms->2400ms)	8.3% (1200ms->1100ms)

Results: Step time improvements

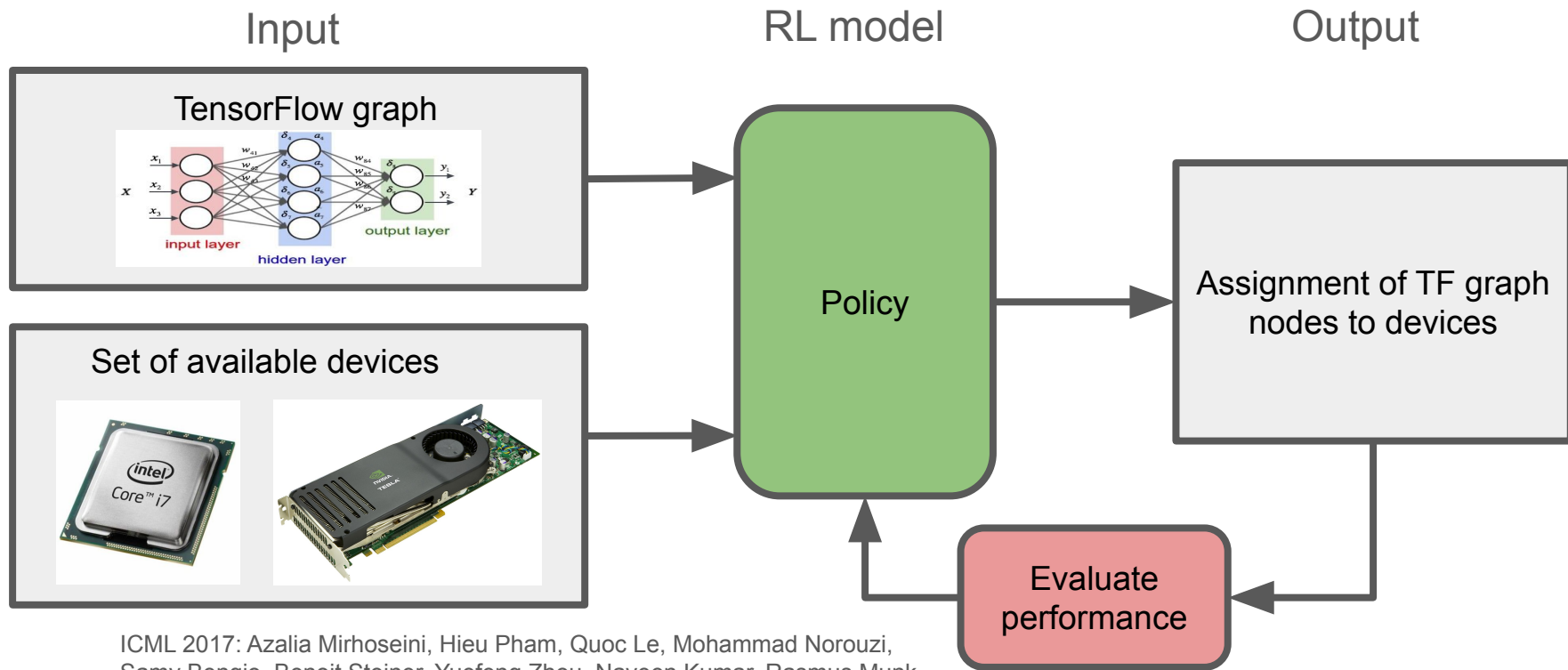
Performance
measured on GPU

Results only include
optimizations that
are turned on by
default

Improvements from Graph Optimizations



Improved HW Utilization Through ML Placement



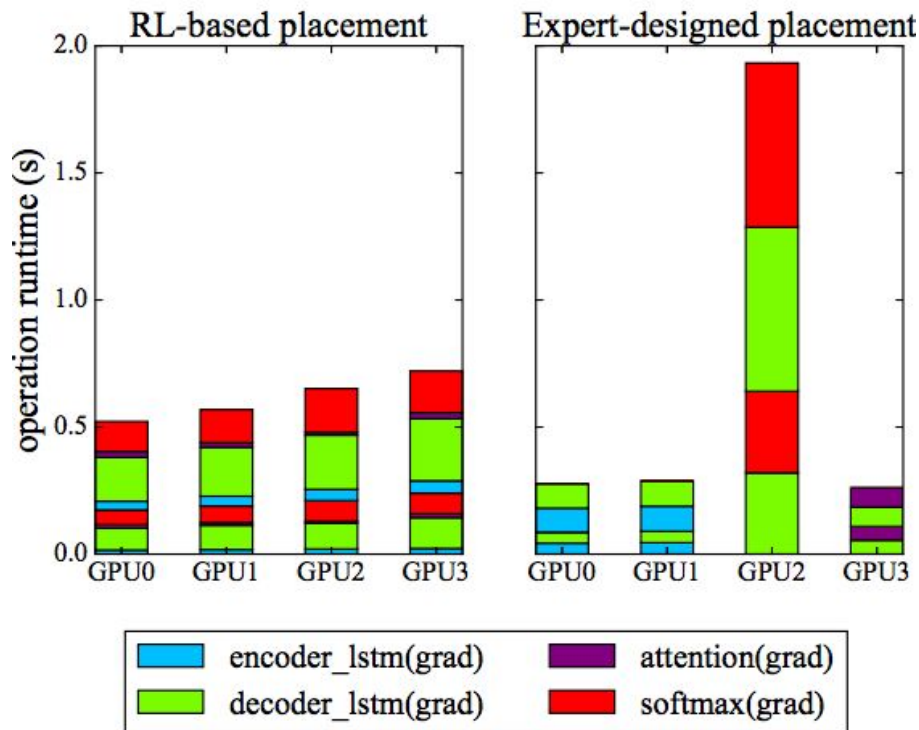
ICML 2017: Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Munk Larsen, Jeff Dean

Placer Spreads The Compute Load

Model: NMT with 4 layers

Hardware: 4 K40 GPUS,
1 Haswell CPU

Performance improved
by **2.4x**

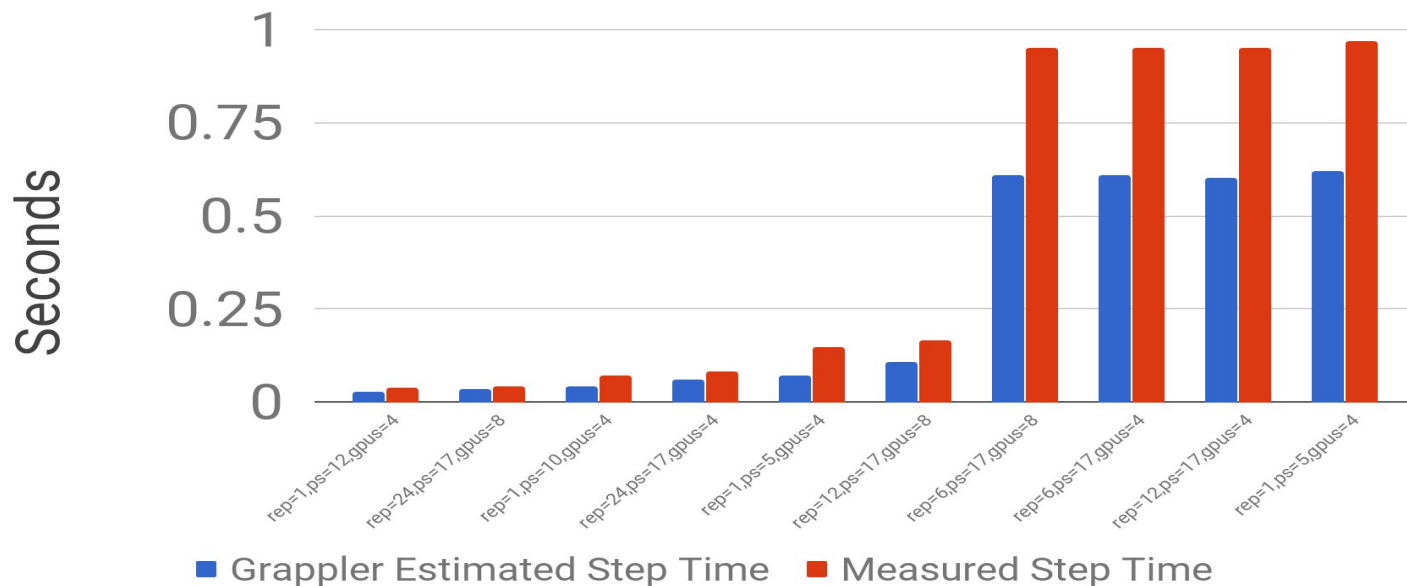


Performance Evaluation Techniques

- Measurement
 - Outliers filtering: warmup + robust statistics
 - Captures all the side effects: memory fragmentation, cache pollution, ...
 - Fairly slow and requires access to input data
- Simulation
 - Per op cost based on roofline estimates
 - Propagation based on plausible schedule
 - Fast but optimistic and requires robust shape inference

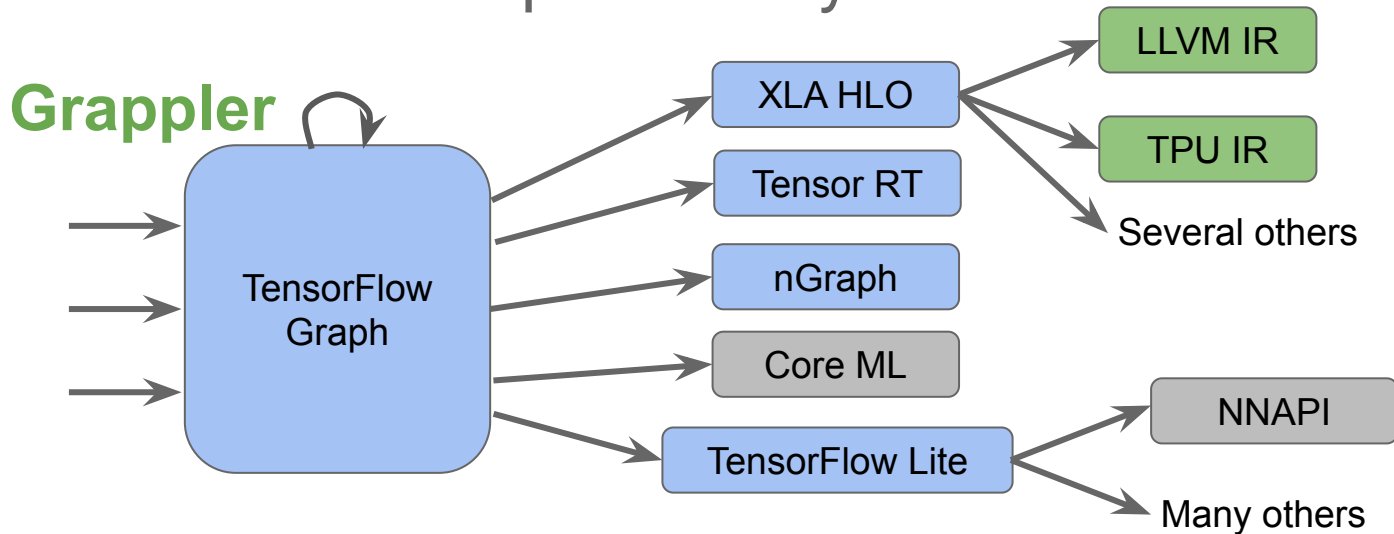
Simulation vs Measurement

Distributed BNMT Step Time Comparison



MLIR: A Compiler Infrastructure for the End of Moore's Law

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 “graphs”:

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

No reasonable existing answers!

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

Quick Tour of MLIR: **Multi-Level** IR

Also: Mid Level,

Moore's Law,

Multidimensional Loop,

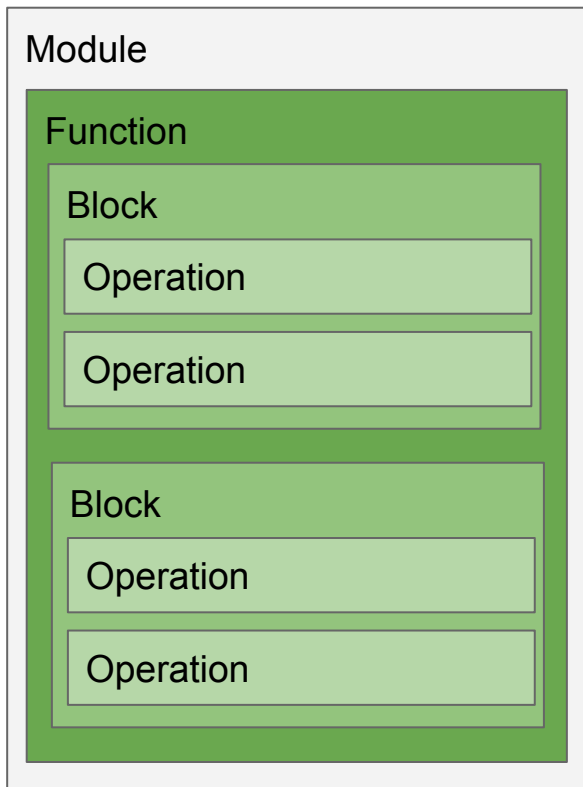
Machine Learning,

...

Many similarities to LLVM

- SSA, typed, three address
- Module/Function/Block/Operation structure
- Round trippable textual form
- Syntactically similar:

```
func @testFunction(%arg0: i32) {  
    %x = call @thingToCall(%arg0) : (i32) -> i32  
    br ^bb1  
^bb1:  
    %y = addi %x, %x : i32  
    return %y : i32  
}
```



MLIR Type System - some examples

Scalars:

- f16, bf16, f32, ... i1, i8, i16, i32, ... i3, i4, i7, i57, ...

Vectors:

- vector<4 x f32> vector<4x4 x f16> etc.

Tensors, including dynamic shape and rank:

- tensor<4x4 x f32> tensor<4x?x?x17x? x f32> tensor<* x f32>

Others: functions, memory buffers, quantized integers, other TensorFlow stuff, ...

Extensible!!

MLIR Operations: an open ecosystem

No fixed / builtin list of globally known operations:

- No “instruction” vs “target-indep intrinsic” vs “target-dep intrinsic” distinction
 - Why is “add” an instruction but “add with overflow” an intrinsic in LLVM? 🤔

Passes are expected to conservatively handle unknown ops:

- just like LLVM does with unknown intrinsics

```
func @testFunction(%arg0: i32) -> i32 {  
    %x = "any_unknown_operation_here"(%arg0, %arg0) : (i32, i32) -> i32  
    %y = "my_increment"(%x) : (i32) -> i32  
    return %y : i32  
}
```

Capabilities of MLIR Operations

Operations always have: **opcode** and source **location** info

Instructions may have:

- Arbitrary number of SSA **results** and **operands**
- **Attributes**: guaranteed constant values
- **Block operands**: e.g. for branch instructions
- **Regions**: discussed in later slide
- Custom printing/parsing - or use the more verbose generic syntax

```
%2 = dim %1, 1 : tensor<1024x? x f32>
```



Dimension to extract is guaranteed integer constant, an “attribute”

```
%x = alloc() : memref<1024x64 x f32>
```

```
%y = load %x[%a, %b] : memref<1024x64 x f32>
```

Complicated TensorFlow Example

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,  
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {  
  
    %0:5 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)  
        {data_format: "NHWC", epsilon: 0.001, is_training: false}  
      : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
      -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
  
    "use"(%0#2, %0#4 ...
```

Complicated TensorFlow Example: Inputs

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,  
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {  
  
  %0:5 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)  
        {data_format: "NHWC", epsilon: 0.001, is_training: false}  
        : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
        -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
  
  "use"(%0#2, %0#4 ...
```

→ Input SSA values and corresponding type info

Complicated TensorFlow Example: Results

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,  
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {
```

```
  %0:5 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)  
    {data_format: "NHWC", epsilon: 0.001, is_training: false}  
    : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
    -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)
```

```
  "use"(%0#2, %0#4 ...
```

- This op produces five results
- Each result can be used independently with # syntax
- No “tuple extracts” get in the way of transformations

Complicated TensorFlow Example: Attributes

```
func @foo(%arg0: tensor<8x?x?x8xf32>, %arg1: tensor<8xf32>,  
         %arg2: tensor<8xf32>, %arg3: tensor<8xf32>, %arg4: tensor<8xf32>) {  
  
  %0:5 = "tf.FusedBatchNorm"(%arg0, %arg1, %arg2, %arg3, %arg4)  
    {data_format: "NHWC", epsilon: 0.001, is_training: false}  
    : (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
    -> (tensor<8x?x?x8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>, tensor<8xf32>)  
  
  "use"(%0#2, %0#4 ...
```

- Named attributes
- “NHWC” is a constant, static entity, not an SSA value
- Similar to “[immarg](#)”, but much richer vocabulary of constants

Extensible Operations Allow Multi-Level IR

Lowering

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
 : !llvm.float`

Also: TF-Lite, Core ML, other frontends, etc ...



Don't we end up with the JSON of compiler IRs????

MLIR “Dialects”: Families of defined operations

Example Dialects:

- TensorFlow, LLVM IR, XLA HLO, TF Lite, Swift SIL...

Dialects can define:

- Sets of defined operations
- Entirely custom type system
- Customization hooks - constant folding, decoding ...

Operation can define:

- Invariants on # operands, results, attributes, etc
- Custom parser, printer, verifier, ...
- Constant folding, canonicalization patterns, ...

Nested Regions

```
%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 ...  
}
```

→ Functional control flow, XLA fusion node, closures/lambda's, parallelism abstractions like OpenMP, etc.

MLIR: Infrastructure

Declarative Op definitions: TensorFlow LeakyRelu

- Specified using TableGen
 - LLVM Data modelling language
- Dialect can create own hierarchies
 - "tf.LeakyRelu" is a "TensorFlow unary op"
- Specify op properties (open ended)
 - e.g. side-effect free, commutative, ...
- Name input and output operands
 - Named accessors created
- Document along with the op
- Define optimization & semantics

```
def TF_LeakyReluOp : TF_UnaryOp<"LeakyRelu",  
                                [NoSideEffect, SameValueType]>,  
                                Results<(outs TF_Tensor:$output)> {  
  let arguments = (ins  
    TF_FloatTensor:$value,  
    DefaultValuedAttr<F32Attr, "0.2">:$alpha  
  );  
  
  let summary = "Leaky ReLU operator";  
  let description = [{  
    The Leaky ReLU operation takes a tensor and returns  
    a new tensor element-wise as follows:  
    LeakyRelu(x) = x          if x >= 0  
                  = alpha*x   else  
  }];  
  
  let constantFolding = ...;  
  let canonicalizer = ...;  
  let referenceImplementation = ...;  
}
```

Generated documentation

tf.LeakyRelu (TF::LeakyReluOp)

Leaky ReLU operator

Description:

The Leaky ReLU operation takes a tensor and returns a new tensor element-wise as follows:

```
LeakyRelu(x) = x      if x >= 0
              = alpha*x else
```

Operands:

1. `value`: tensor of floating-point values

Attributes:

Attribute	MLIR Type	Description
<code>alpha</code>	<code>FloatAttr</code>	32-bit float attribute
<code>T</code>	<code>Attribute</code>	derived attribute

Results:

1. `output`: tensor of tf.dtype values

Generated C++ Code

- C++ class TF::LeakyReluOp
- Typed accessors:
 - value() and alpha()
- IRBuilder constructor
 - builder->create<LeakyReluOp>(loc, ...)
- Verify function
 - Check number of operands, type of operands, compatibility of operands
 - Xforms can assume valid input!

```
namespace TF {  
class LeakyReluOp  
    : public Op<LeakyReluOp,  
                OpTrait::OneResult,  
                OpTrait::HasNoSideEffect,  
                OpTrait::SameOperandsAndResultType,  
                OpTrait::OneOperand> {  
  
public:  
    static StringRef getOperationName() {  
        return "tf.LeakyRelu";  
    };  
    Value *value() { ... }  
    APFloat alpha() { ... }  
    static void build(...) { ... }  
    bool verify() const {  
        if (...) return emitOpError(  
            "requires 32-bit float attribute 'alpha'");  
        return false;  
    }  
    ...  
};  
} // end namespace
```

Specify simple patterns simply

```
def : Pat<(TF_SqueezeOp StaticShapeTensor:$arg), (TFL_ReshapeOp $arg)>;
```

- Support M-N patterns
- Support constraints on Operations, Operands and Attributes
- Support specifying dynamic predicates
 - Similar to "Fast and Flexible Instruction Selection With Constraints", CC18
- Support manually written rewrites in C++
 - Always a long tail, don't make the common case hard for the tail!

Goal: Declarative, reduces boilerplate, easy to express for all

Passes, Walkers, Pattern Matchers

- Additionally module/function passes, function passes, utility matching functions, nested loop matchers ...

```
struct Vectorize : public FunctionPass<Vectorize> {  
    void runOnFunction() override;  
};
```

```
...  
f->walk([&](Operation *op) {  
    process(op);  
});  
...
```

```
...  
if (matchPattern(getOperand(1), m_Zero()))  
    return getOperand(0);  
...
```

mlir-opt

- Similar to LLVM's opt: a tool for testing compiler passes
- Every compiler transformation is unit testable:
 - Including verification logic, without dependence on earlier passes
 - Policy: every behavior changing commit includes a test case

```
// RUN: mlir-opt %s -loop-unroll | FileCheck %s
func @loop_nest_simplest() {
  // CHECK: affine.for %i0 = 0 to 100 step 2 {
  affine.for %i = 0 to 100 step 2 {
    // CHECK: %c1_i32 = constant 1 : i32
    // CHECK-NEXT: %c1_i32_0 = constant 1 : i32
    // CHECK-NEXT: %c1_i32_1 = constant 1 : i32
    affine.for %j = 0 to 3 {
      %x = constant 1 : i32
    }
  }
  return
}
```

Integrated Source Location Tracking

API *requires* location information on each operation:

- File/line/column, op fusion, op fission
- “Unknown” is allowed, but discouraged and must be explicit.

Easy for passes to emit structured diagnostics:

```
$ cat test/Transforms/memref-dependence-check.mlir
```

```
// Actual test is much longer...
func @test() {
  %0 = alloc() : memref<100xf32>
  affine.for %i0 = 0 to 10 {
    %1 = load %0[%i0] : memref<100xf32>
    store %1, %0[%i0] : memref<100xf32>
  }
  return
}
```

```
$ mlir-opt -memref-dependence-check memref-dependence-check.mlir
...
m-d-c.mlir:5:10: note: dependence from 0 to 0 at depth 1 = false
    %1 = load %0[%i0] : memref<100xf32>
            ^
...
m-d-c.mlir:6:5: note: dependence from 1 to 0 at depth 1 = false
  store %1, %0[%i0] : memref<100xf32>
    ^
```

Location Tracking: Great for Testing!

Test suite uses -verify mode just like Clang/Swift diagnostic test:

- Test analysis passes directly, instead of through optimizations that use them!

```
// RUN: mlir-opt %s -memref-dependence-check -verify
func @test() {
  %0 = alloc() : memref<100xf32>
  affine.for %i0 = 0 to 10 {

    // expected-note @+1 {{dependence from 0 to 1 at depth 2 = true}}
    %1 = load %0[%i0] : memref<100xf32>

    store %1, %0[%i0] : memref<100xf32>
  }
}
```

mlir-translate - test data structure translations

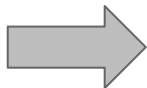
- mlir-translate converts MLIR $\rightleftharpoons$ external format (e.g. LLVM .bc file)
- The actual *lowerings* and abstraction changes happen within MLIR
 - Progressive lowering of ops within same IR!
 - Leverage all the infra built for other transformations
- Decouple function/graph transformations from format change
 - Principle: Keep format transformations simple/direct/trivially testable & correct
 - ~> Target dialect represents external target closely
- But what about codegen via LLVM ... ?

LLVM IR Dialect: Directly use LLVM for CodeGen

- LLVM optimization and codegen is great at the C level abstraction
- Directly uses the LLVM type system:

```
!llvm<"{ i32, double, i32 }">
```

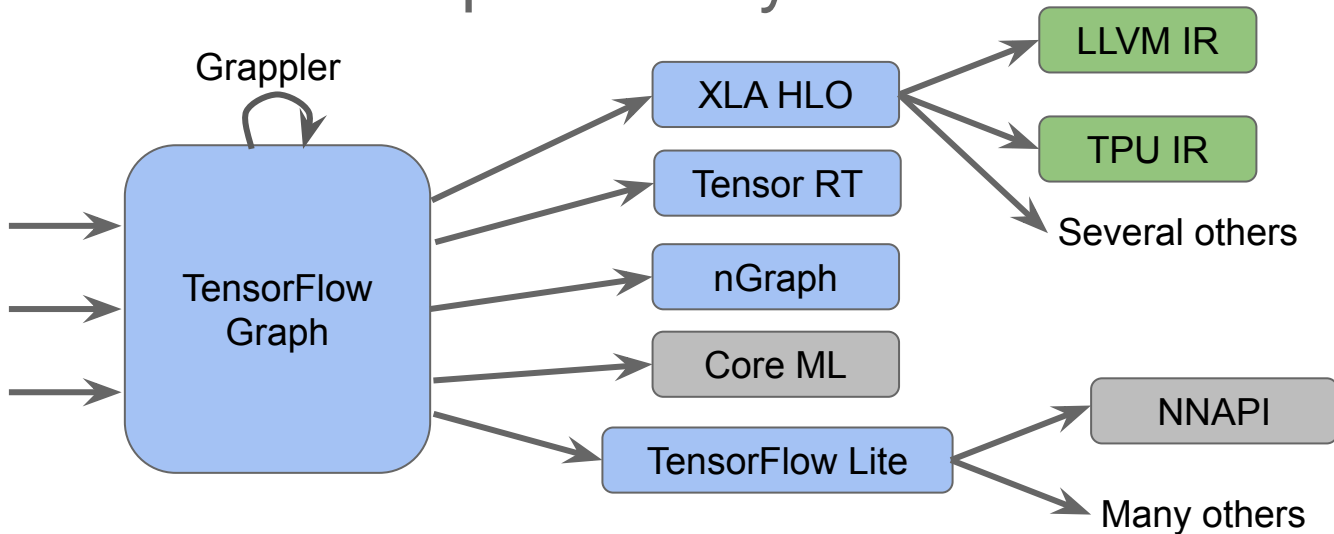
Code lowered to LLVM dialect



```
...  
^bb2: // pred: ^bb1  
  %9 = llvm.constant(10) : !llvm.i64  
  %11 = llvm.mul %2, %9 : !llvm.i64  
  %12 = llvm.add %11, %6 : !llvm.i64  
  %13 = llvm.extractvalue %arg2[0] : !llvm<"{ float* }">  
  %14 = llvm.getelementptr %13[%12] :  
    (!llvm<"float*">, !llvm.i64) -> !llvm<"float*">  
  llvm.store %8, %14 : !llvm<"float*">  
...
```

MLIR: Use within TensorFlow

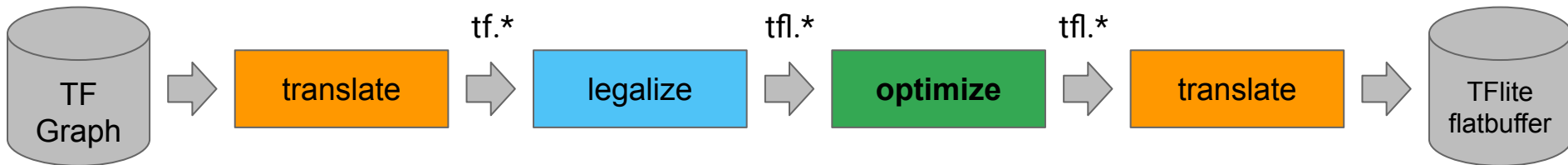
The TensorFlow compiler ecosystem



TensorFlow ecosystem is complicated, TensorFlow team plan:

- Incrementally move graph transformations to MLIR
- Unify interfaces to external code generators
- Provide easier support for first-class integration of codegen algorithms

TensorFlow Lite Converter



- TensorFlow to TensorFlow Lite converter
 - Two different graph representations
 - Different set of ops & types
 - Different constraints/targets
- Overlapping goals with regular compilation
 - Edge devices also can have accelerators (or a multitude of them!)
 - Same lowering path, expressed as rewrite patterns
- MLIR's pluggable type system simplifies transforms & expressibility
 - Quantized types is a first class citizen in dialect

Old “TOCO” User Experience

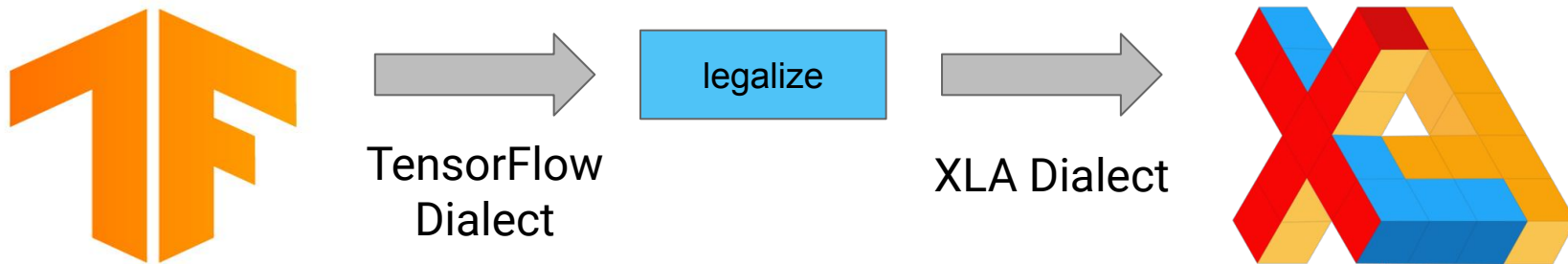
```
F0122 11:20:14.691357    27738 import_tensorflow.cc:2549] Check failed: status.ok()
Unexpected value for attribute 'data_format'. Expected 'NHWC'
*** Check failure stack trace: ***
    @    0x5557b0ac3e78  base_logging::LogMessage::SendToLog()
    @    0x5557b0ac46c2  base_logging::LogMessage::Flush()
    @    0x5557b0ac6665  base_logging::LogMessageFatal::~LogMessageFatal()
    @    0x5557af51e22b  toco::ImportTensorFlowGraphDef()
    @    0x5557af51f60c  toco::ImportTensorFlowGraphDef()
    (...)
    @    0x5557af4ac679  main
    @    0x7f7fa2057bbd  __libc_start_main
    @    0x5557af4ac369  _start
*** SIGABRT received by PID 27738 (TID 27738) from PID 27738; ***
F0122 11:20:14.691357    27738 import_tensorflow.cc:2549] Check failed: status.ok()
Unexpected value for attribute 'data_format'. Expected 'NHWC'
E0122 11:20:14.881460    27738 process_state.cc:689] RAW: Raising signal 6 with default
behavior
Aborted
```

Improved User Experience

Clang-style caret
diagnostics
coming soon!

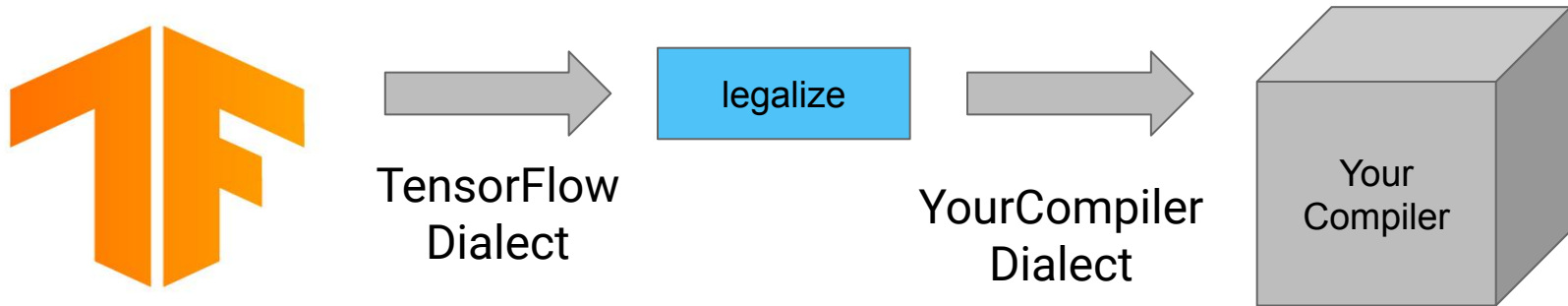
```
node "MobilenetV1/MobilenetV1/Conv2d_0/Conv2D" defined
  at 'convolution2d' (third_party/tensorflow/contrib/layers/python/layers/layers.py:1156):
    conv_dims=2)
  at 'func_with_args' (third_party/tensorflow/contrib/framework/python/ops/arg_scope.py:182):
    return func(*args, **current_args)
  at 'mobilenet_base' (third_party/tensorflow_models/slim/nets/mobilenet/mobilenet.py:278):
    net = opdef.op(net, **params)
  ...
  at 'network_fn' (resnet/nets_factory.py:93):
    return func(images, num_classes, is_training=is_training, **kwargs)
  at 'build_model' (resnet/train_experiment.py:165):
    inputs, depth_multiplier=FLAGS.depth_multiplier)
  ...
error: 'tf.Conv2D' op requires data_format attribute to be either 'NHWC' or 'NCHW'
Failed to run transformation: tf-raise-control-flow
```

New TensorFlow Compiler Bridge



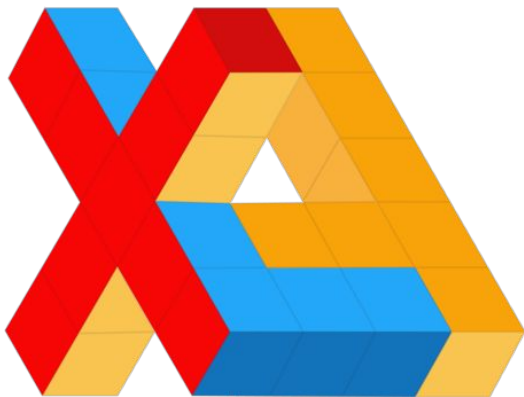
- Interop between TensorFlow and XLA
 - Consists of rewrite passes and transformation to XLA
- Large part is expanding subset of TensorFlow ops to XLA HLO
 - Many 1-M patterns
 - Simple to express as DAG-to-DAG patterns
- XLA targets from multi-node machines to edge devices
 - Not as distinct from TensorFlow Lite

Not just XLA: Custom Compiler Backends

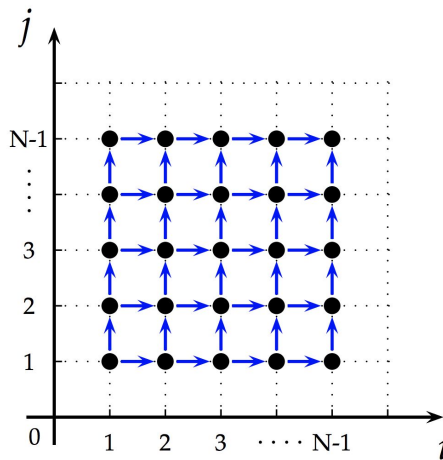


- Other compilers can be integrated using the same framework
 - Dialect defines operations and types
 - Pattern rewrites specify transformation rules
- Custom pipelines can reuse existing components
 - Translate from TensorFlow or XLA dialect
 - Optimize graph before translation

Tensor Codegen: Investing in Two Approaches



XLA Compiler Technology



Polyhedral Compiler Algorithms

MLIR is Open Source!

Visit us at github.com/tensorflow/mlir:

- Code, documentation, examples
- Developer mailing list: mlir@tensorflow.org

Still early days:

- Contributions not accepted yet - still setting up CI, etc.
- Merging TensorFlow-specific pieces into public TensorFlow repo

Thank you to the team!

Questions?

We are hiring interns!
mlir-hiring@google.com

