Google AI

Thoughts on Tensor Code Generation in MLIR

January 23, 2020

Chris Lattner <clattner@google.com>

Sharing ideas from dozens of people

DISCLAIMERS

This is a public version of an internal vision discussion:

- This is Chris' personal opinion and perspective

This isn't a committed design, there are no schedules:

- If you're excited about this, please discuss and contribute! :-)

Shared in case it is interesting and useful to others in the field:

- Explained in TensorFlow specific terms, but is general to other domains
- Many of us in MLIR community are facing similar engineering challenges

This was put together quickly, it is not a polished talk

- Set your expectations low :-)

Talk Overview

Discuss approaches to code generation from tensors to machine instructions:

- Intended to inspire a collaborative and incremental path forward
- Food for thought, not a proscriptive answer

This makes up a bunch of “descriptive” names to reduce bias in the conversation:

- Focus on abstractions, layering, and scale; not existing solutions
- These names should definitely not stick!

Talk is about architecture and project management, *not* codegen algorithm details

- Nothing in this talk is novel, this is just a framework to talk about things
- I am not picking a winner here, I think all the approaches are equally wrong :-) :-)

What is well understood?

Requirements & Goals

Lowering from “ops” to target-specific IR

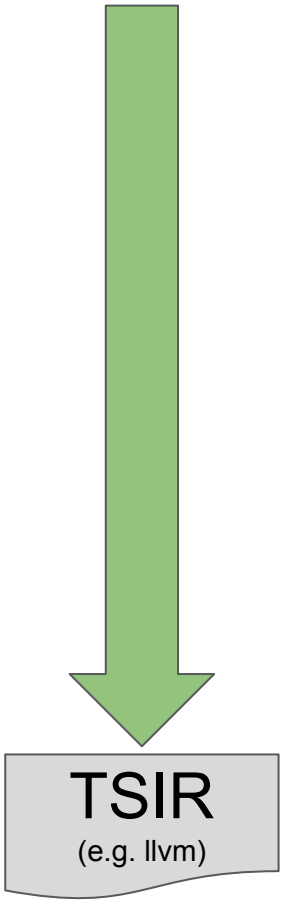
Google’s requirements:

- Support TPUs, GPUs, CPUs, ...
- Provide state of the art code quality, e.g. >= XLA for “xPUs”
- Multiple input dialects: HLO, TFLite, TF, ...

Goals:

- Share as much across targets as possible:
  - Reusable, parameterized algorithms based on target description
  - Reduce amount of proprietary “secret sauce”
  - Make hw/sw co-design much more powerful
- Scalable to support diverse architectures: CGRAs, FPGAs, …
- Declarative is better than imperative
- Simple, extensible and explainable architecture
- Provide framework for custom ops
  - Important for research, hackability, etc
- Go beyond dense linear algebra

Output of this lowering: “Target Specific IR”



Result is a target-specific IR talking to a target-specific code generator:

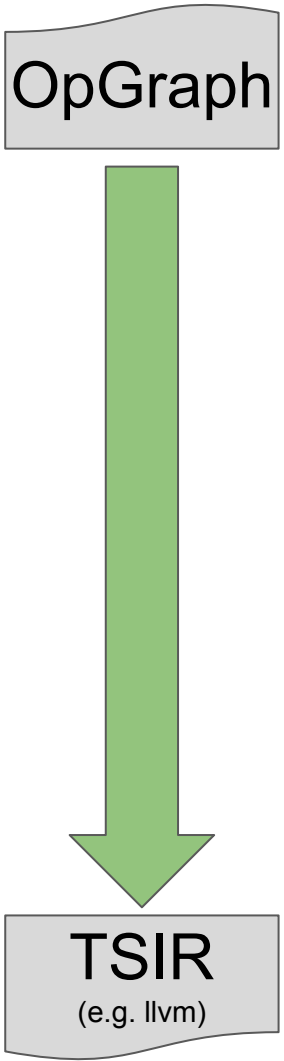
- PTX, AMDIL, SPIRV, MLIR GPU dialect …
- LLVM IR with target-specific intrinsics
- Proprietary code generator for TPUs
- Lots of diversity in the broader accelerator market …
- Not necessarily LLVM IR!

Character of these IRs:

- Completely target specific operations and abstractions
- Handle register allocation, scheduling, machine code emission
- Varied architectures, in every way possible

MLIR provides a standardized framework to talk to these:

- Allows sharing of code between “generators” of the IR



Input: “Graph” of Tensor Ops in MLIR

Can be many different dialects:

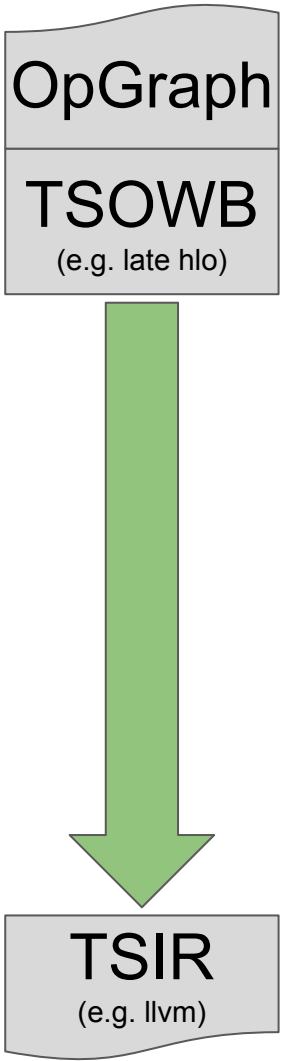
- TensorFlow, XLA HLO, TF-Lite, …
- Community cares about other dialects as well
- Encourage innovation in this space over time

MLIR provides a standard dialect-independent IR + infra:

- Allows optionally capturing known shape info, layout, etc

Future: Would like to move beyond tensors as the only abstraction:

- Many interesting things currently in TensorFlow, but not first class
- TensorArray’s, Ragged Tensors, etc.
- Lots of other interesting ideas to be explored in time



“Target Specific Ops With Buffers”

Produced by New TensorFlow Compiler Bridge + HLO passes

“Compiler Bridge” needs its own MLIR ODM talk!

For Each Device:

- Lower aggregate ops to ops supported by the device compiler, e.g. TF->HLO
  - Using declarative rewrite rules, ABC lowering, etc
- Auto-cluster around unsupported ops and deal with them in the runtime
- Hand written kernels (e.g. cuDNN) handled by the bridge/runtime
- Run “HLO passes” - buffer, layout assignment, etc

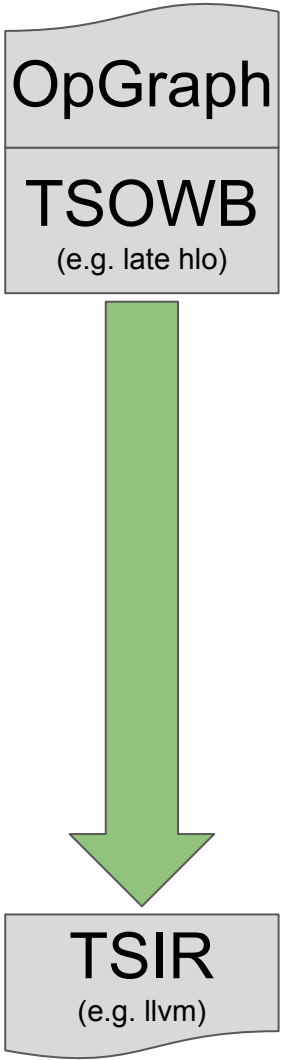
Result is a subgraph, expressing one “kernel”:

- Computation for an accelerator *without* host runtime support
- Resulting ops (e.g. HLO) can be different than input ops (e.g. TF)

Provides incredible flexibility for target authors:

- Specific set of ops can vary based on target: HLO, TFL, “just matmul”
- Target should be able to specify constraints, e.g. “only 256x256 matmuls plz”
  - Compiler Bridge can legalize a lot of things at the graph level

Current Approaches



XLA xPU Emitters

Lowers from Late HLO to xPU-specific instruction set

- Complex hand-crafted C++ code

Proven!

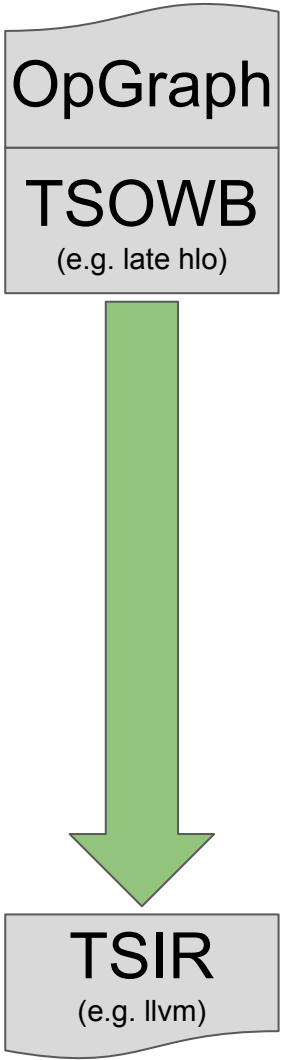
- Works great, high performance, supports a very challenging architecture

Challenges:

- Little reuse/sharing of target-independent algorithms
- Monolithic design - solving many problems in one go
- No IR’s within the emitters makes testing more challenging
- Difficult to understand and evolve

Some implementation limitations: static shapes, custom ops etc

- Not an inherent problem with this approach!



LinAlg / Structured Ops?

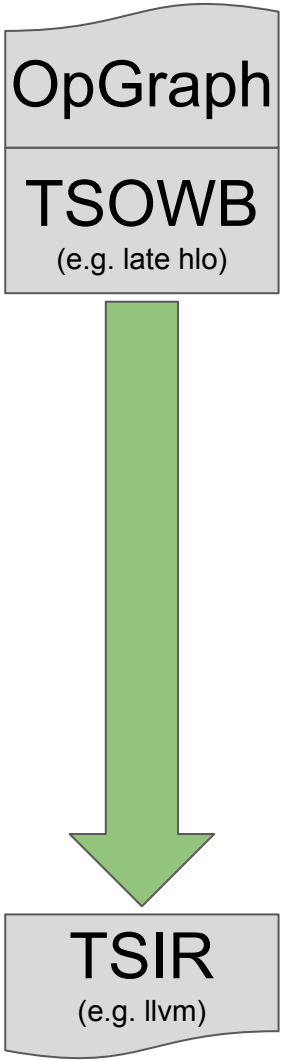
Very useful technologies and implementation approach for many problems:

- Structured Ops are representation of loop nests
- Declarative rewrite rules leveraging SSA and MLIR infra
- EDSLs to help build computations

Learn more: Dec 5, 2019 MLIR Open Design Meeting

Important ingredients, but not really a “framework”:

- Many mechanisms, but not enough imposed policy
- Target authors have too much design freedom / responsibility
- Many best-practices are known
- Some more scaffolding around this would help



Common challenges

Spanning a huge abstraction gap all in one step:

- Huge design space

Monolithic approach encourages conflation of multiple different concerns:

- Complexity, understandability, scalability challenges

Effectively assumes that there is one codegen algorithm/approach per target

⇒ MLIR provides an answer to these sorts of challenges!

- New abstractions, new IRs, better separation of concerns

A xPU* Engineering Management Challenge

XLA/xPU is “really really good” at what it does:

- “Many” people-years of compiler work on xPU-specific code generation
- Extensively performance tuned for internal and external workloads
- Supports multiple generations of xPU
- Runs all production xPU workloads

Also not “general enough”, so we’d like to get to a more flexible design:

- Long wishlists of features to support - e.g. dynamic shapes
- New stack needs must be a superset of old stack
- Cannot regress capabilities or performance!

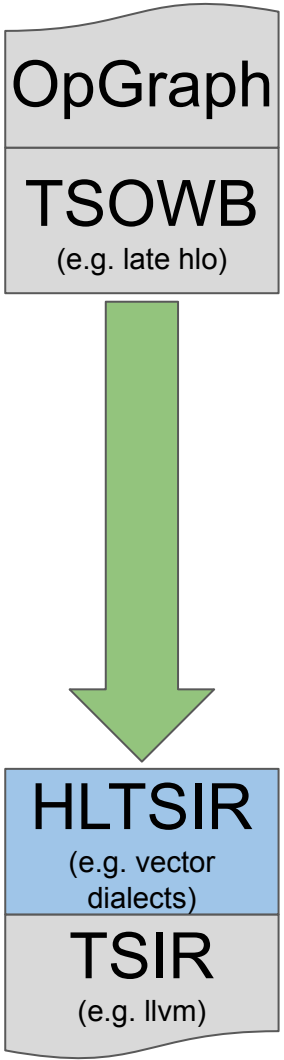
How do we bring up a new codegen architecture?

- Shouldn’t have to do “all the work” before any of the payoff
- “Big bang” changes are very risky

* Certainly not a xPU-specific challenge :-)

Suggested Approach

Incremental and Collaborative



High Level Target Specific IR

Target specific code generators often have a somewhat arbitrary abstraction boundary

- E.g. “whatever LLVM can handle” - single dimensional vectors + target intrinsics

MLIR allows defining higher level IRs:

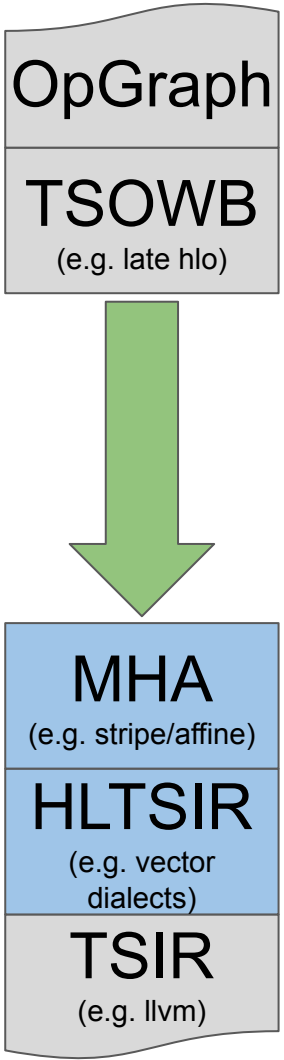
- e.g. the new vector dialects
- “Value semantic” abstractions for matmul on xPUs

Guiding principles designing this layer:

- HLTSIR can always be **predictably** lowered to “TSIR”: simple cost model
- This is a lowering layer; should be a clean abstraction

Why is this useful?

- Reduces the complexity of the arrow, provides better separation of concerns
- Raises the abstraction level for things like “custom ops”



“Memory Hierarchy Abstraction” - worst name in the talk :-)

Input is **valid** “High Level Target Specific IR” embedded into a “loop IR”:

- May be very very slow, e.g. all data is in “global memory”

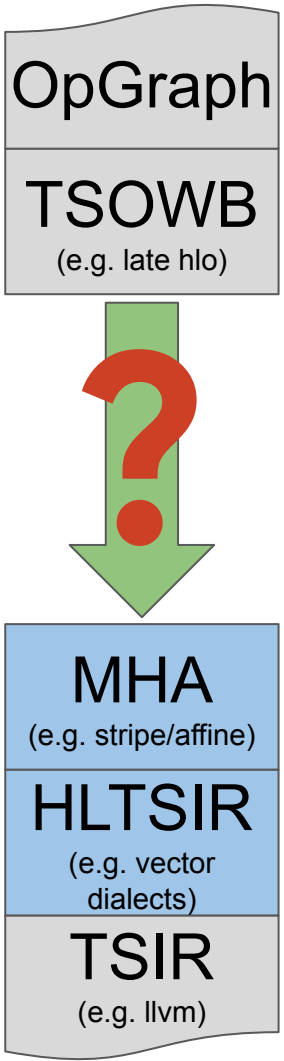
Perform correctness-preserving optimizations:

- Mostly loop level things: Tiling, prefetching, etc
- Utilize smaller/faster memory hierarchies
- Use standardized abstractions for buffers/allocations, etc.

These transformations are well understood:

- The more powerful they are, the less the “green arrow” has to solve
- Generally parameterizable / ~target independent
- Allow target-specific passes as well
- Lots of useful transformations “in the box” for target authors to use/reuse

Uday and Intel’s work here is very exciting!



What is the right green arrow here?

Lots of different algorithms in this space - unlikely that there is “one true answer”:

- We want to support new research directions on tensor codegen
- Draw more people from the C4ML community into a common framework

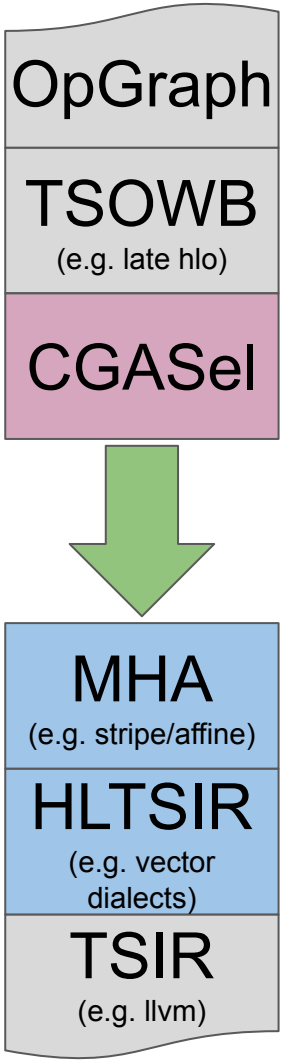
“Ops” are not just linear algebra or dense math:

- Sparsity
- Ragged tensors
- Input pipeline abstractions

Other challenges:

- We want to support “weird” targets like FPGAs, CGRAs, …
- The “XLA codegen for xPUs is very mature” problem

⇒ **Recommendation:** optionality, not “one perfect answer”



CodeGenAlgorithm Selector

Input is a graph of ops that the target can support:
=> But possibly not by **all** code generators for the target

Enables federated / MoE approach with multiple CGA's for a target:

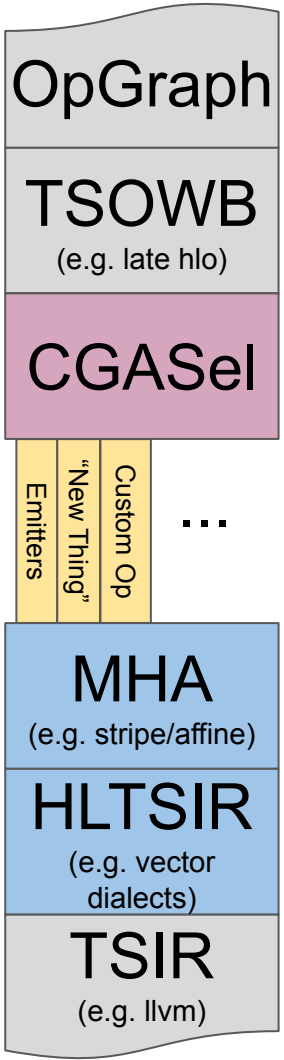
- Different algorithms can have different capabilities
- Can have different performance / implementation quality

"CGASel" is a graph partitioning algorithm:

- Can be written by hand with simple heuristics
- Can use search/RL/… and other algorithms if/when there is a reason to
- We understand this sort of problem pretty well

Key is to make sure they all work on the same input/output abstractions:

- ...and are properly modular, so they work together as libraries



How does this help the "XLA/xPUs is mature" challenge?

Continue incremental improvements to XLA emitters: makes the xPU product better

- Low risk, architecture is well understood by the team

New algorithms can be explored next to them:

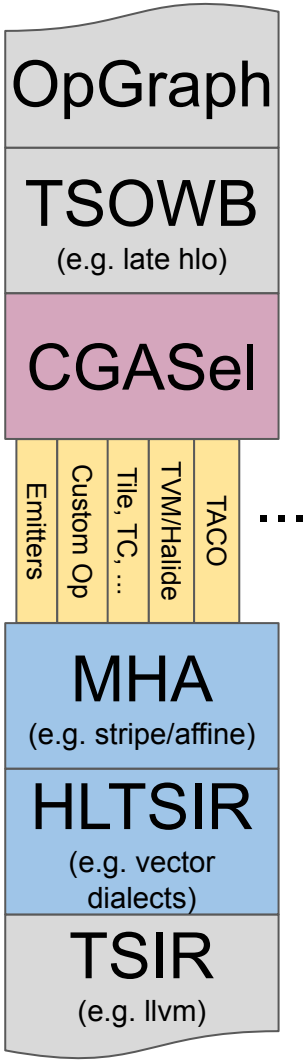
- Prototype and explore with less tuned ops, e.g. FFT or TopK
- Introduce support for currently-unsupported ops
- Explore dynamic shapes without pressure to be optimal for static shapes

Emitters can be progressively simplified as MHO/HLTSIR expands and improves

- Compiler evolution is now nicely incremental

Key requirements:

- Nail down buffer and affine/loop abstractions
- Refactor XLA/xPU to generate "HLTSIR" then "MHO", instead of "TSIR" directly



Why is this nice for MLIR more generally?

Many people care about this problem space!

- Industry, academia, many different groups

There are lots of subdomains with different constraints:

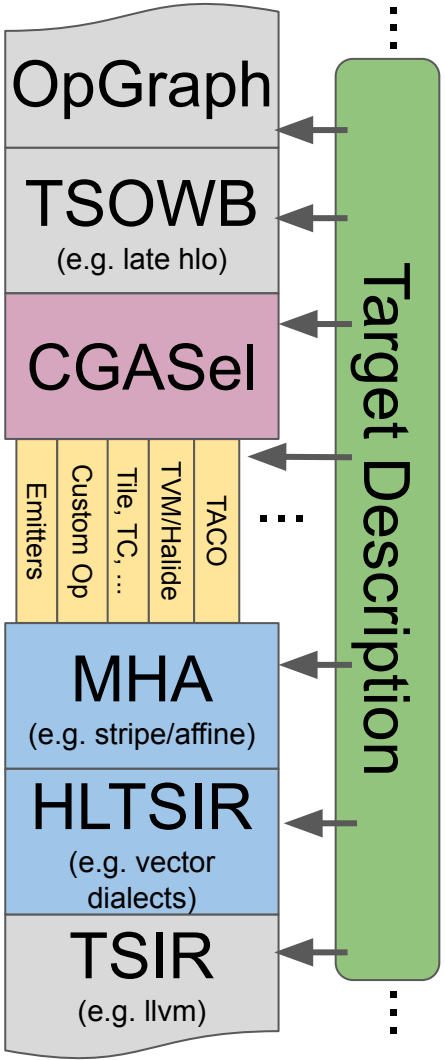
- e.g. Sparse Tensor Algebra Compiler (TACO)
- Many well-known graphics algorithms for GPUs
- Some accelerators are tremendously flexible - e.g. CPUs!

Isn't reasonable to expect a "winner take all" algorithm:

- Users just want fast code, they don't care about the algorithm

This is "**The MLIR approach**":

- Provide a framework, encourage people to collaborate within it
- Things standardize and shake out over time as we gain experience



What about retargetability, co-design, sharing?

Every part of this stack can be written in target-specific or target-indep way:

- Start with target-specific things, and generalize when multiple clients exist

No reason to block on “perfect target descriptions”:

- Generalized architecture can be driven by product requirements
- E.g. new generations within product family, e.g. Pascal vs Volta
- Co-design exploration adds new capabilities

Progressively build this over time:

- How both GCC and LLVM evolved their target description capabilities
- e.g. no LLVM support for VLIW existed until someone needed it
- e.g. first delay slot filler was target specific and generalized later

TLDR: Build infra for optionality on Code Gen Algorithms

Lots of well known algorithms with different tradeoffs
Lots of experts in the field
“Framework” approach will make it easier to experiment and do research
Not every chip can be targeted from every codegen approach
Many interesting things beyond dense tensors

Managing an “existing thing to MLIR” transition

Testing the water

Generic software management advice:

- Prefer incremental and “bulk mechanical” changes; don’t explode/redesign the world
- Pick the right point in the product schedule so you can do it well
- Do not make an ‘old team’ vs ‘new team’: encourage collaboration and joint ownership

A starting point: Keep existing algorithms, use MLIR data structures

- Near zero algorithmic risk
- Phase in massive mechanical change through progressive refactorings
- Emulate your existing “IR builders”
- Refactor your existing APIs to get closer to MLIR concepts
- Use existing MLIR types and other representations

Walk, run, then jump!

Start benefiting from MLIR *incrementally*:

- Take advantage of nice MLIR things: PassMgr, testing tools, DRR, …
- Much easier integration with other systems

Extend your IR:

- Utilize Regions, ease of defining new ops
- Mixing of dialects
- Leverage existing MLIR dialects / passes

Things to consider:

- Leveraging larger infra pieces
- Upstreaming, contributing, improving things: lots of work to do and low hanging fruit!

Questions