Google DeepMind

MLIR Is *Not* an ML Compiler

And Other Common Misconceptions

Alex Zinenko
<zinenko@google.com>

2023 LLVM Developer Meeting
October 12, 2023 - Santa Clara, CA

Using an ML Compiler

Q: What is the performance of an 4x56x56x64 NHWC convolution with a 3x3 window (ResNet-50 conv-2) on an A100 GPU?

XLA: N milliseconds.
TVM: M milliseconds.
TorchInductor: K milliseconds.

MLIR: 

Well, it depends. The question is not well specified.
Which dialects are used?
Which transformations are applied?
How is this lowered?

```
@jax.jit
def myconv(lhs: jnp.Array, rhs: jnp.Array):
  ...

torch_tvm.enable()
@torch.jit.script
def myconv(...):
  torch.nn.Conv2D(...)()

@torch.compile
def myconv(...):
  ...

%timeit myconv(...)
```

Well, all of these are JIT in Python with hidden machinery.

Using a~~n ML~~ Compiler

.cc	clang	.exe
.f90	flang	.exe
.cc	clang -O3	faster .exe
.cc	clang -O3 -mllvm -polly	faster .exe after fancy opts

* assuming environment is configured correctly, eventual (c)make, etc

From test-tensor-e2e.mlir
git@f2f61a99f7f754f3e4

Using MLIR

No optimization

.mlir **mlir-opt** -arith-bufferize
-linalg-bufferize -tensor-bufferize
-func-bufferize -finalizing-bufferize
-buffer-deallocation-pipeline
-convert-bufferization-to-memref
-convert-linalg-to-loops
-convert-arith-to-llvm
-convert-scf-to-cf -convert-cf-to-llvm
-finalize-memref-to-llvm
-convert-func-to-llvm
-reconcile-unrealized-casts
.mlir

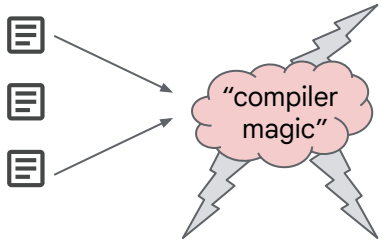
Still MLIR

.mlir **mlir-translate** -mlir-to-llvmir .ll

.ll **clang** -x ir -O3 .exe (not that fast)

* assuming environment is configured correctly, eventual (c)make, etc

Using MLIR



Let’s do transformers instead of ResNet! There is an attention/softmax block that uses math.exp.

.mlir

mlir-opt -arith-bufferize
-linalg-bufferize -tensor-bufferize
-func-bufferize -finalizing-bufferize
-buffer-deallocation-pipeline
-convert-bufferization-to-memref
-convert-linalg-to-loops
-convert-arith-to-llvm
-convert-scf-to-cf -convert-cf-to-llvm
-finalize-memref-to-llvm
-convert-func-to-llvm
-reconcile-unrealized-casts

Math dialect not handled here

MLIR Is Not a Compiler

~~MLIR~~ `mlir-opt` Is Not a Compiler

~~MLIR~~ `mlir-opt` Is Not a Compiler Frontend!

Neither is LLVM `opt`
But there is nothing else!

Towards a Frontend for MLIR

ISO/IEC 14882:2020

.cc → clang → .exe

.mlir → mlirc → .exe

Dialects 💫

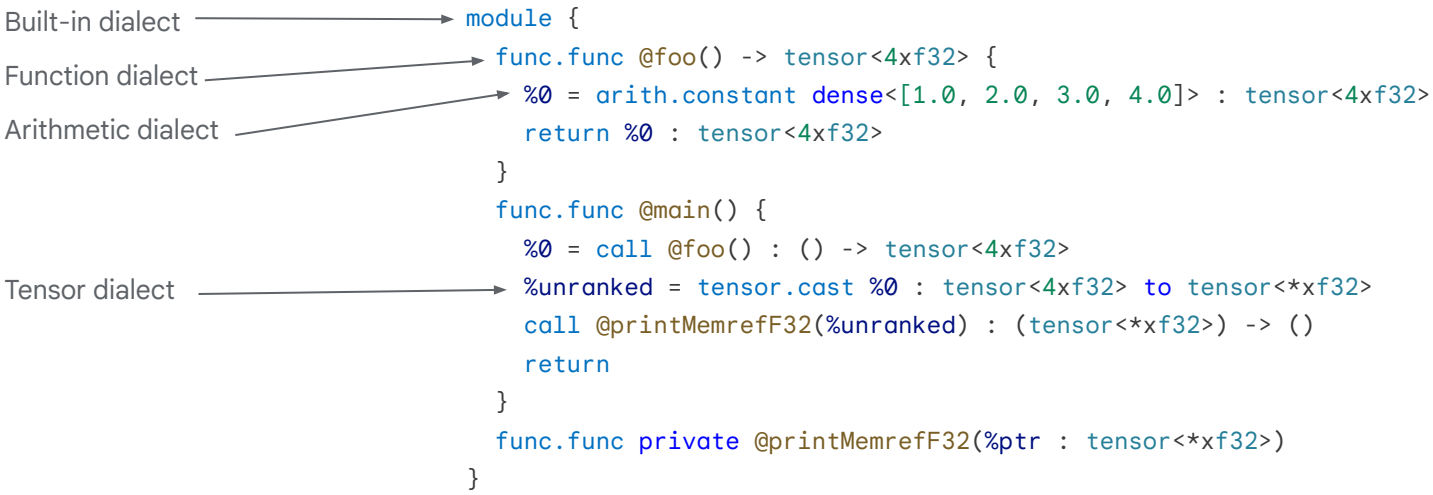
e.g. x86 + Linux ABI

Share target info

Also want: SPIR-V, CIRCT

Misconception: MLIR Can Be In Dialect A, B or C

Unlike human language dialects where we speak one or another,
MLIR dialects are almost always mixed.



Misconception: There Is a Common Set of Instructions

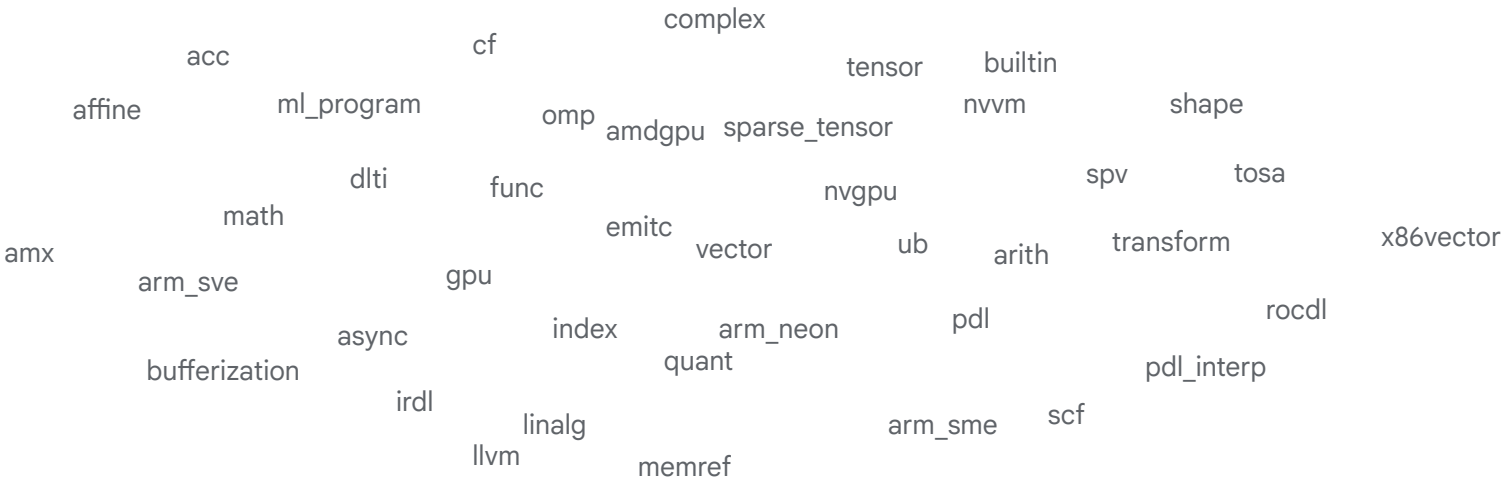
The exhaustive list of built-in MLIR operations:

- `module`,
- `unrealized_conversion_cast`.

There are 10 built-in types though.

Misconception: There Is a Common Set of Instructions

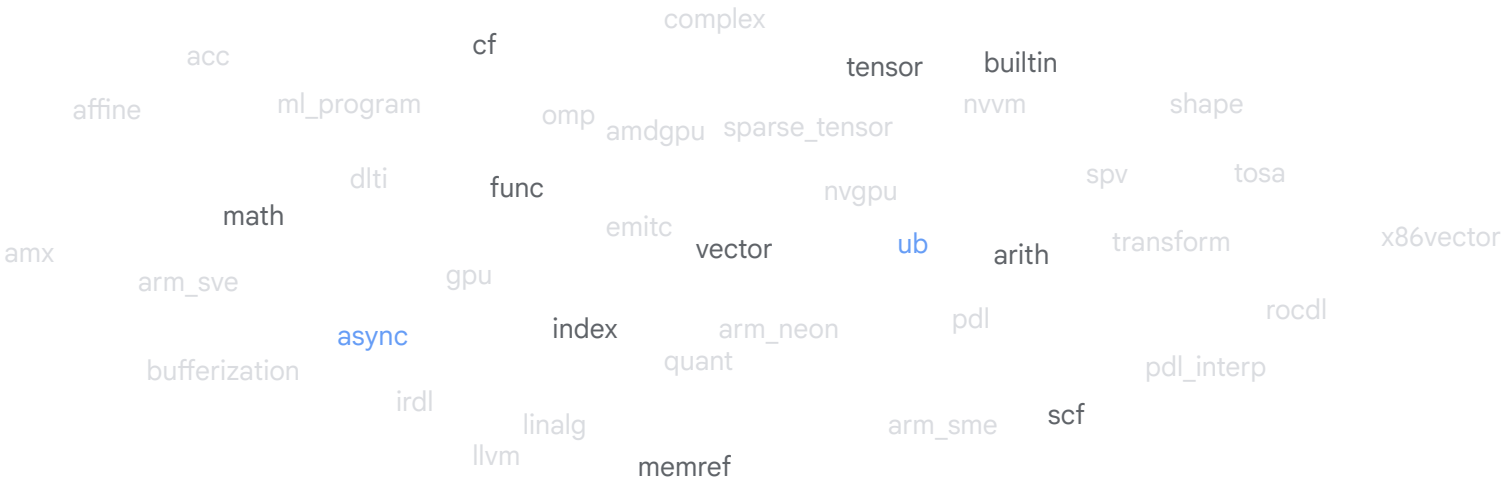
Can’t we just use all upstream dialects?



41 total

Misconception: There Exists a Standard Dialect

It has been split into the following dialects,
with some more that would have been candidates for “standardization”.



Towards a Frontend for MLIR: Stages

mlir-opt -arith-bufferize
-linalg-bufferize -tensor-bufferize
-func-bufferize -finalizing-bufferize
-buffer-deallocation-pipeline

Move from tensors to buffers

-convert-bufferization-to-memref
-convert-linalg-to-loops
-convert-arith-to-llvm
-convert-scf-to-cf -convert-cf-to-llvm
-finalize-memref-to-llvm
-convert-func-to-llvm
-reconcile-unrealized-casts

Convert soup of dialects to the LLVM dialect

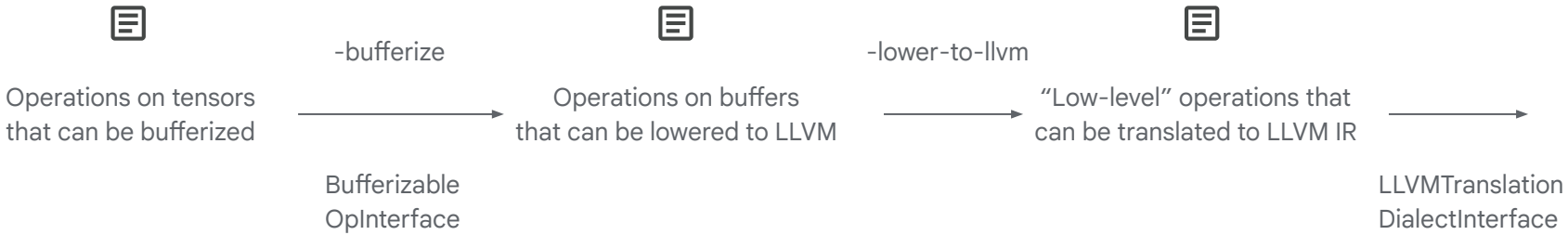
Towards a Frontend for MLIR: Stages

mlirc

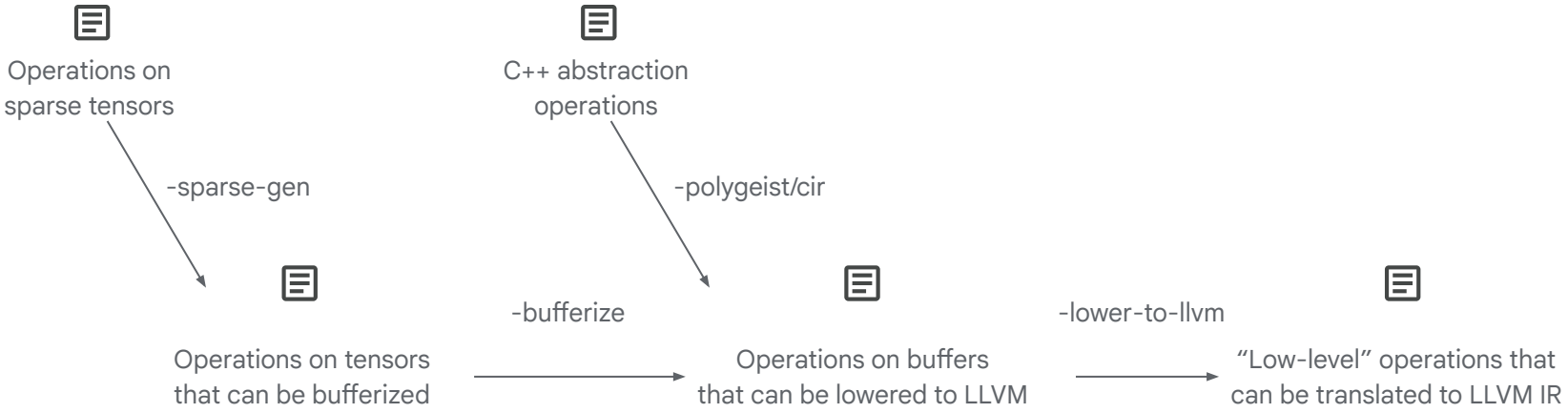
-bufferize

-lower-to-llvm

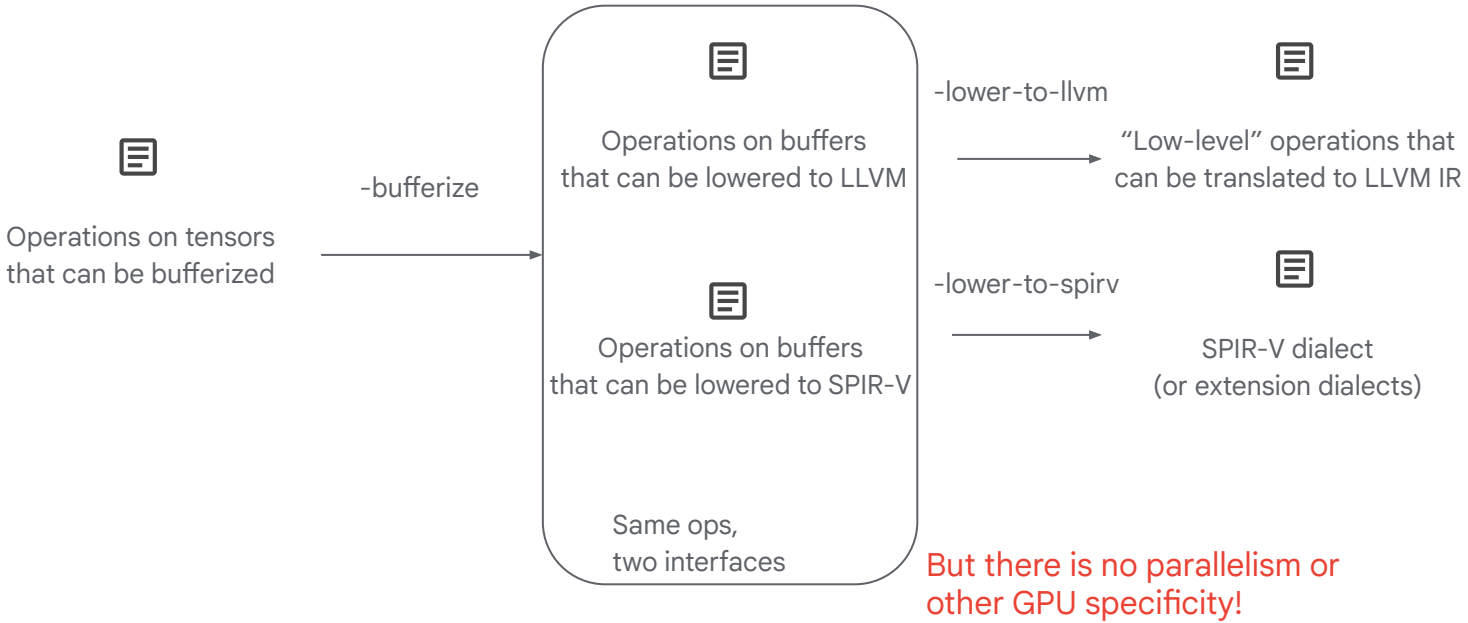
Towards a Frontend for MLIR: Interfaces



Towards a Frontend for MLIR: Stages



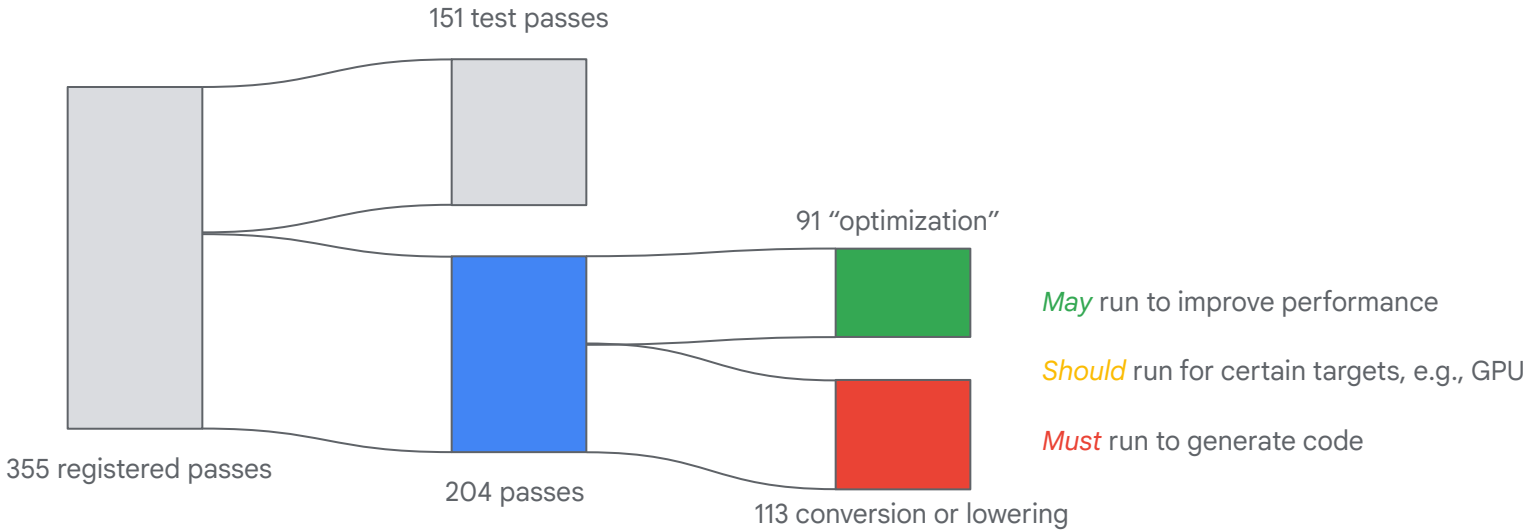
Towards a Frontend for MLIR: Targeting GPUs



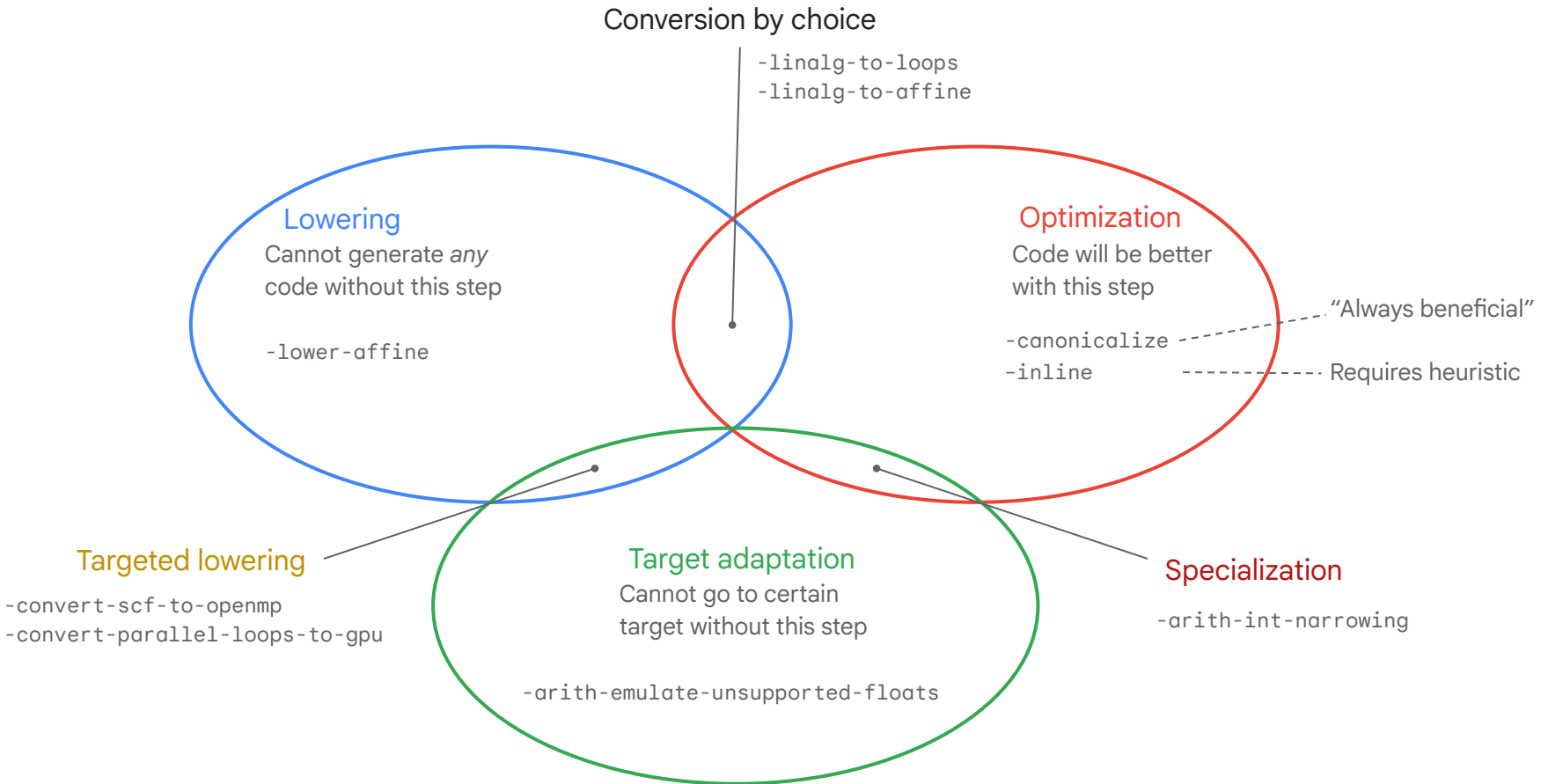
Misconception: `mlir-opt` Is an Optimizer Driver

Documentation

```
$ mlir-opt --help | head -n 1
OVERVIEW: MLIR modular optimizer driver
```

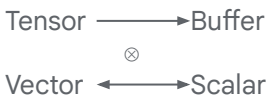


Categorizing `mlir-opt` Passes: Function

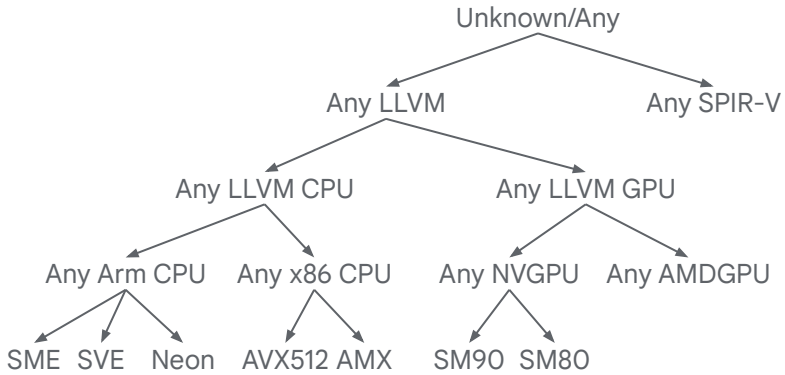


Categorizing `mlir-opt` Passes: Target Abstraction

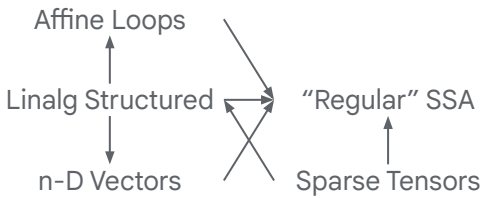
"Type System" ~Lowering



Target ~Targeted Lowering



Optimization substrate ~Conversion by choice

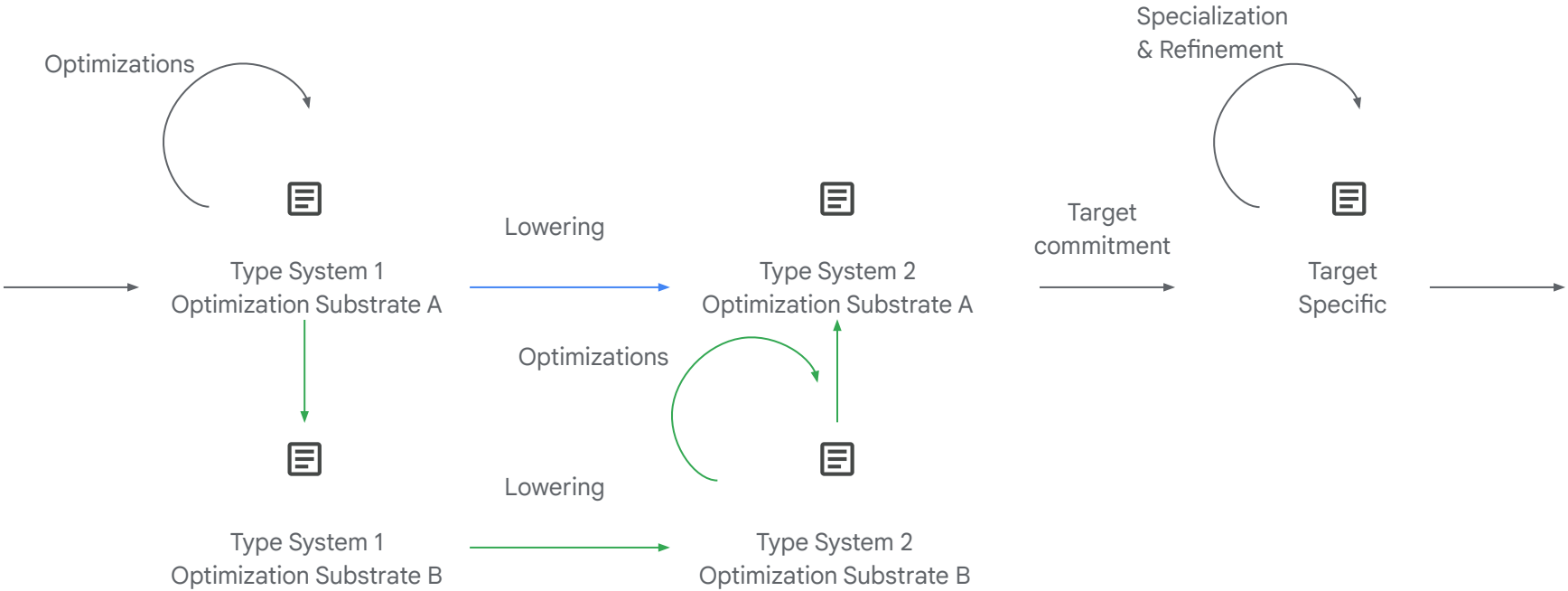


Mostly a DAG.

Constraints: e.g., affine loops substrate requires buffer-of-scalars type system

Refining the target is incompatible with most optimization substrates

Building a Pass Pipeline



Frontend configuration: *order* of type system and substrate changes + target selection
Default optimization pipelines can be informed by the target.

Misconception: MLIR is Primarily for Machine Learning

Doesn't ML in MLIR stand for Machine Learning?

A lot of upstream MLIR has been heavily influenced by ML workloads:

- Built-in tensor type, different from vector.
- No operations on tuple types (ML frameworks provide those).
- "Memory reference" type that is *not a pointer* but a Torch-style sizes/strides/offset buffer descriptor.
- Quantization and `ml_program` dialects.
- The only "frontend-ish" dialect is Tensor Operator Set Architecture (TOSA).

But not entirely:

- Signless integer arithmetic.
- Explicit loops.
- `emitc` dialect.

Sibling projects and downstreams are not ML

- Flang.
- CIR and Polygeist.
- CIRCT.

Misconception: MLIR Compiler Would Be a Sequence of Passes

These are “mostly test” passes.

Internal logic:

- Collect A-to-B rewrite patterns.
- Configure target options.
- `applyConverisonPatterns`

mlir-opt -arith-bufferize
-linalg-bufferize -tensor-bufferize
-func-bufferize -finalizing-bufferize
-buffer-deallocation-pipeline
-convert-bufferization-to-memref
-convert-linalg-to-loops
-convert-arith-to-llvm
-convert-scf-to-cf -convert-cf-to-llvm
-finalize-memref-to-llvm
-convert-func-to-llvm
-reconcile-unrealized-casts

One is supposed to build their own pass:

```
-apply-patterns="
  bufferization-to-memref,
  linalg-to-loops,
  arith-to-llvm,
  ...
  my-dialect-to-llvm
"
```

Should LLVM provide an (MLIR-based) ML compiler?

Should LLVM provide an (MLIR-based) ML compiler?

If not, how do we position the project unambiguously?

MLIR is *not* an ML compiler.

- No defined input format / frontend.
- No established pass pipeline.
- No target information.
- No heuristics.
- No benchmarks.

- It's okay as long as there is an interface.
- *Lowering* can be organized in stages around interfaces.
- We should have one!
- *Optimization* can be driven by flags or meta-dialects (pdl, transform).
- ???

MLIR is a collection of abstractions and transforms to assemble a compiler, for ML or anything else.

Invest in better discoverability of abstractions, especially within the broader LLVM ecosystem.

Google DeepMind

MLIR Is *Not* an ML Compiler, Yet?

Alex Zinenko
<zinenko@google.com>

2023 LLVM Developer Meeting
October 12, 2023 - Santa Clara, CA