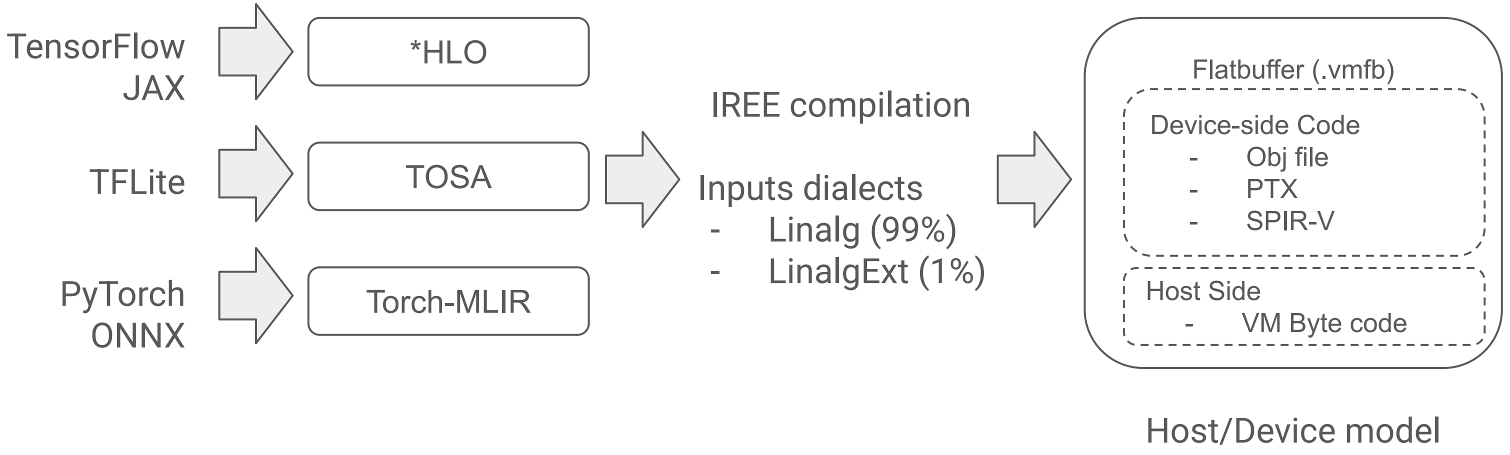
Data-Tiling in IREE: Achieving High Performance Through Compiler Design

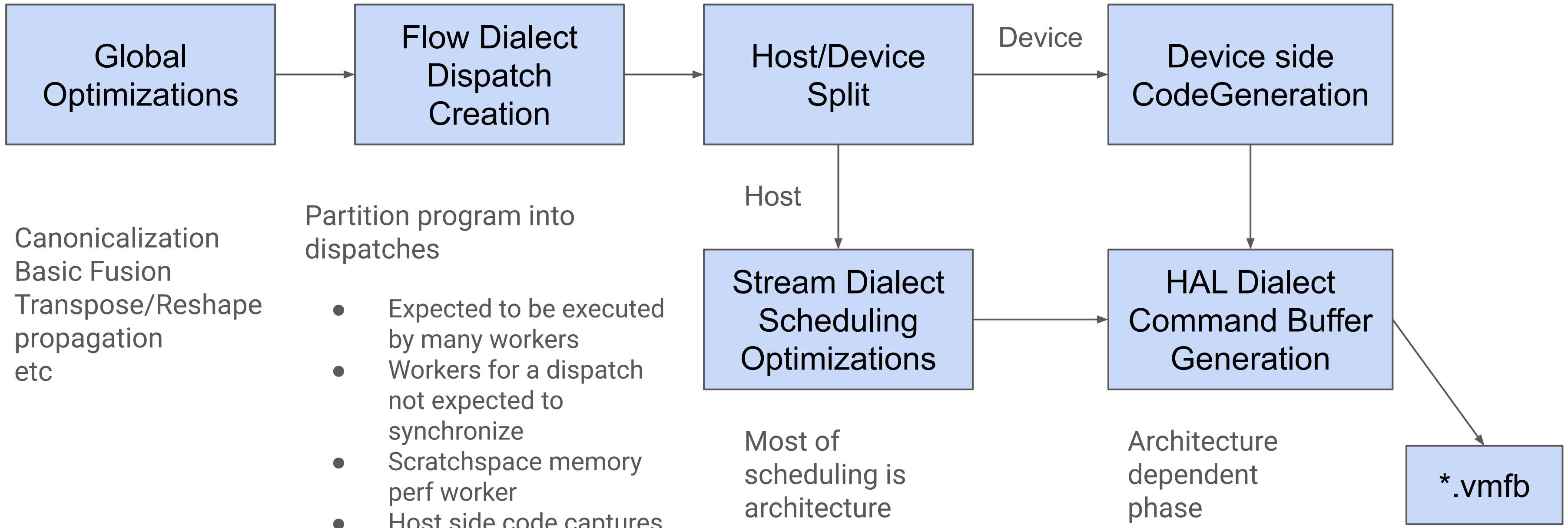
Hanhan Wang, AMD

What is IREE? 👻

- A retargetable MLIR-based compiler for ML programs.
- Take ML workloads from various frontends (PyTorch, Jax, etc.) and execute on different backends (x86, Arm, NVIDIA GPUs, AMD GPUs, etc.)
- https://github.com/iree-org/iree



IREE Compiler Design


Global Optimizations
Flow Dialect Dispatch Creation
Host/Device Split
Device
Device side CodeGeneration
Host
Stream Dialect Scheduling Optimizations
HAL Dialect Command Buffer Generation
*.vmfb

- Canonicalization
- Basic Fusion
- Transpose/Reshape propagation
- etc

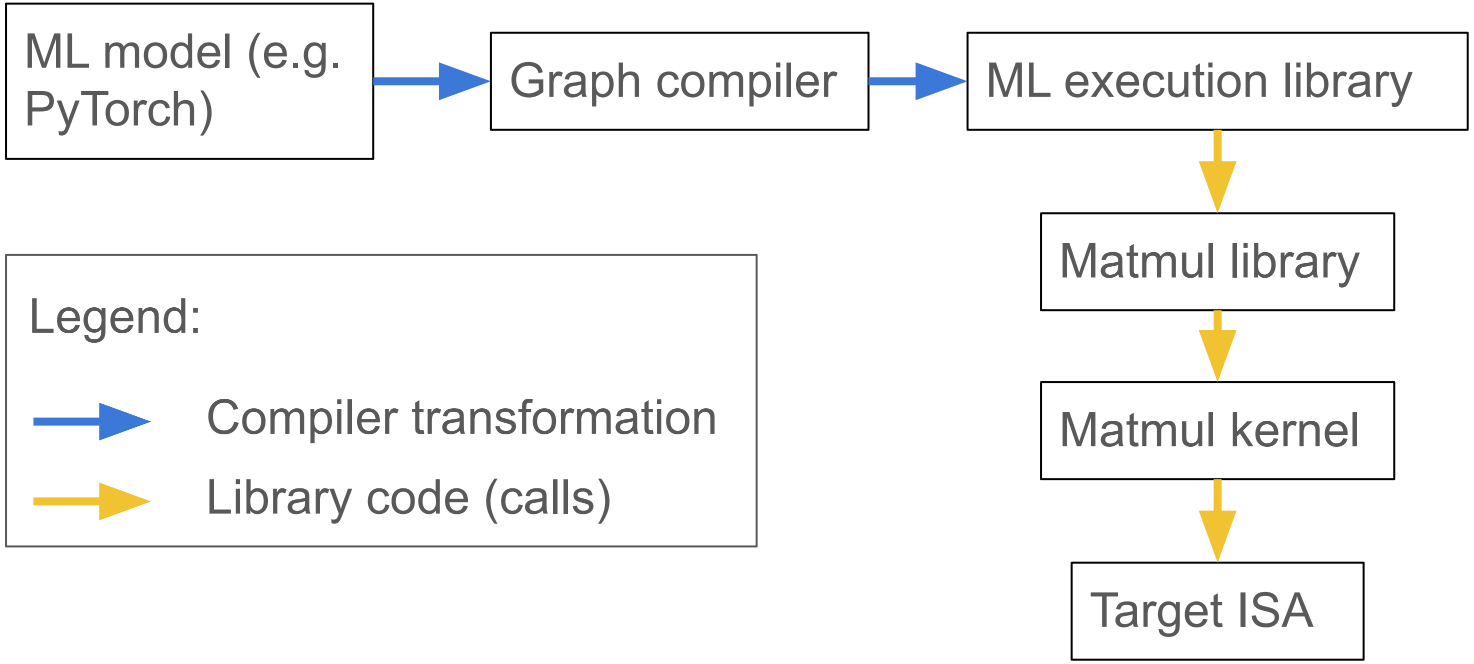
Partition program into dispatches

- Expected to be executed by many workers
- Workers for a dispatch not expected to synchronize
- Scratchspace memory perf worker
- Host side code captures dependences between dispatches

Most of scheduling is architecture agnostic

Architecture dependent phase

ML execution: traditional “library approach” flow



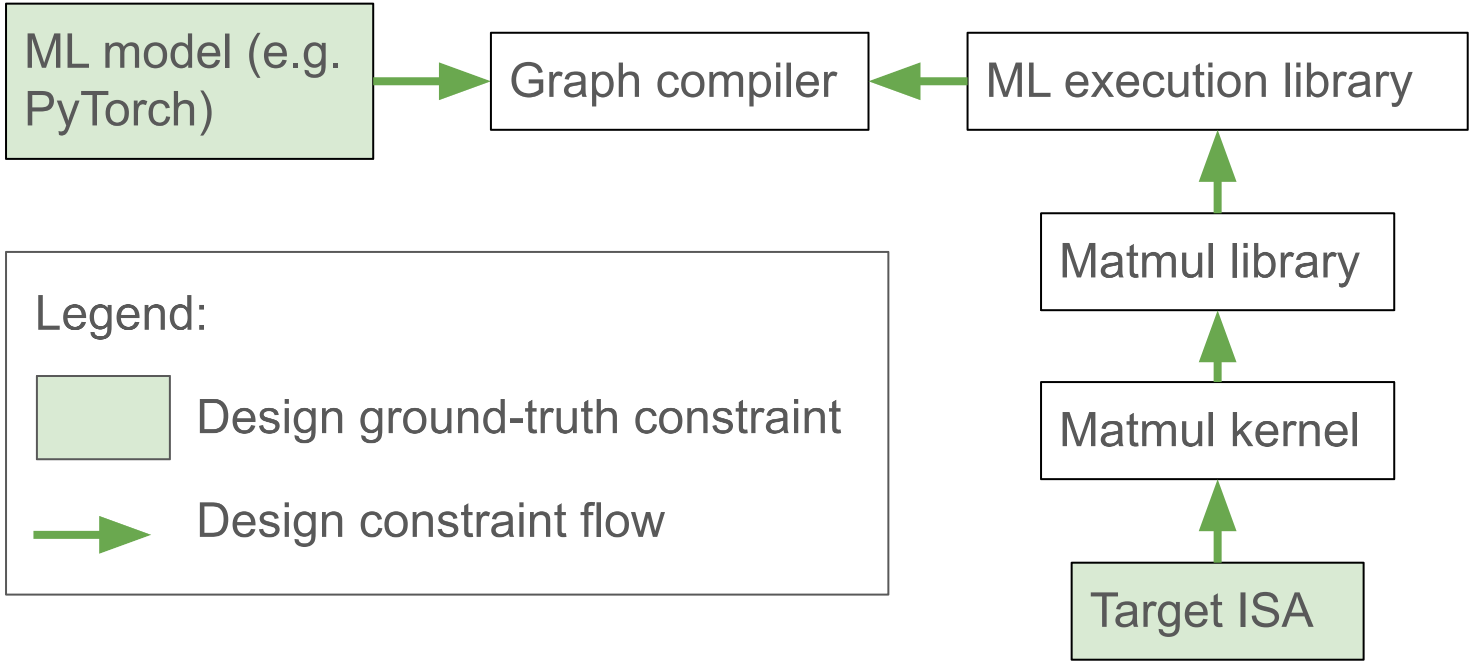
Easy way out

Popular in ML compilers: delegating primitives to libraries.

Result is sub-optimal:

- Usual “mode switch” design caveats.
- Results in a *trade-off*, not a *combination*, of scalability (of the compiler) and performance (of the library).
- Loss of fusion opportunities at library-delegation boundaries.

ML execution: traditional “library approach” flow

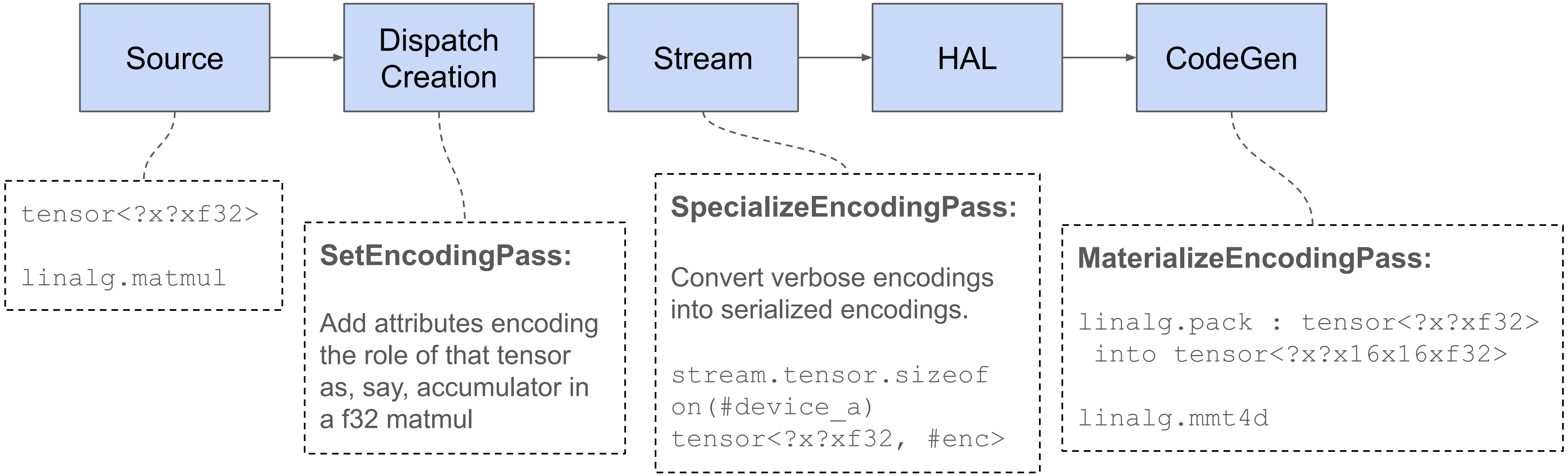


Design constraints flow from the bottom up

Data-Tiling

Introducing layout transformations is not hard in itself.

What is hard is making that fit in the progressive-lowerings design of a retargetable compiler.



```
tensor<?x?xf32>

linalg.matmul
```

SetEncodingPass:

Add attributes encoding the role of that tensor as, say, accumulator in a f32 matmul

SpecializeEncodingPass:

Convert verbose encodings into serialized encodings.

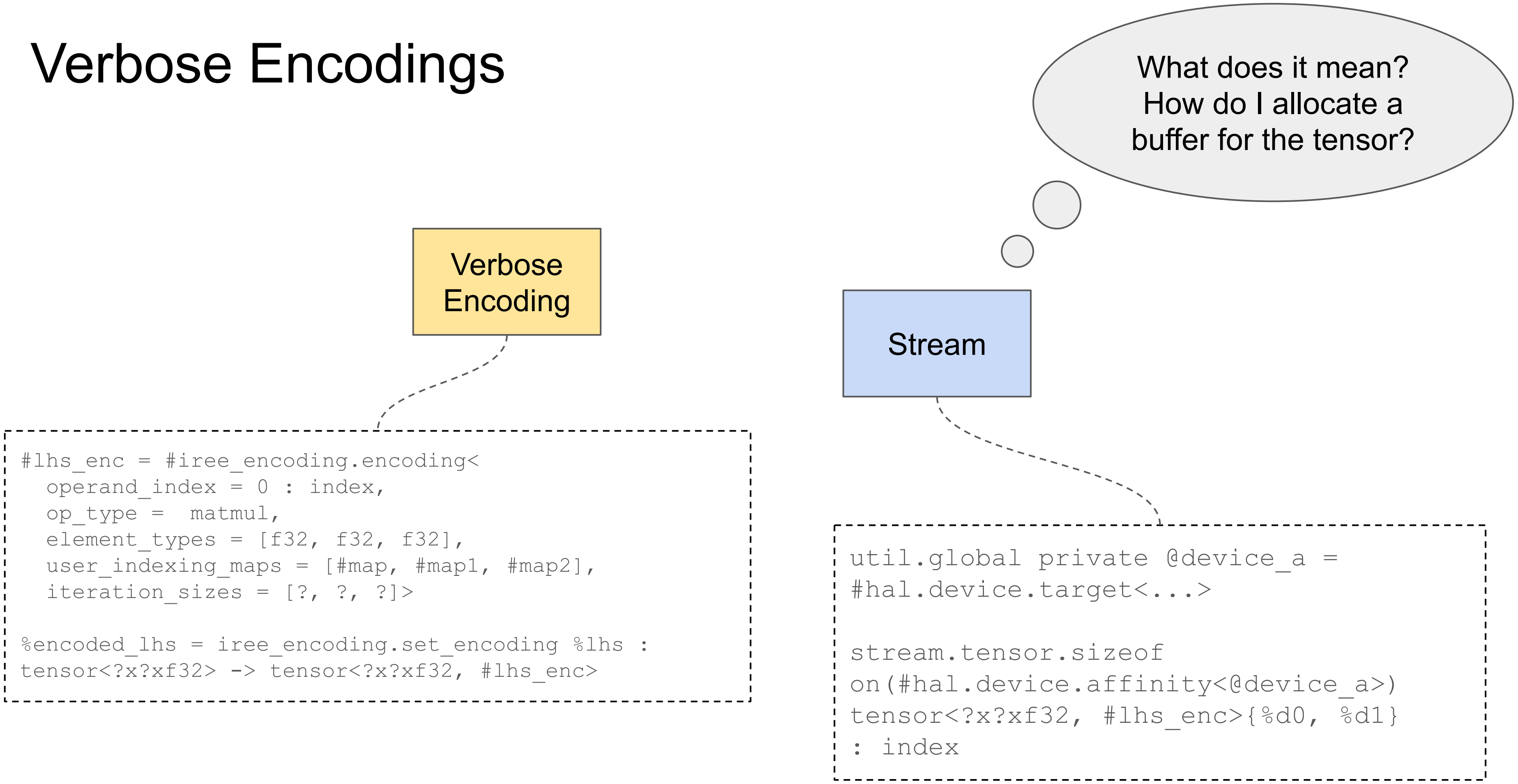
```
stream.tensor.sizeof
on(#device_a)
tensor<?x?xf32, #enc>
```

MaterializeEncodingPass:

```
linalg.pack : tensor<?x?xf32>
 into tensor<?x?x16x16xf32>

linalg.mmt4d
```

Verbose Encodings



```
#lhs_enc = #iree_encoding.encoding<
  operand_index = 0 : index,
  op_type =  matmul,
  element_types = [f32, f32, f32],
  user_indexing_maps = [#map, #map1, #map2],
  iteration_sizes = [?, ?, ?]>

%encoded_lhs = iree_encoding.set_encoding %lhs :
tensor<?x?xf32> -> tensor<?x?xf32, #lhs_enc>
```

```
util.global private @device_a =
#hal.device.target<...>

stream.tensor.sizeof
on(#hal.device.affinity<@device_a>)
tensor<?x?xf32, #lhs_enc>{%d0, %d1}
: index
```

Encoding Resolver

IREE defines executable targets in IR that describes the properties.

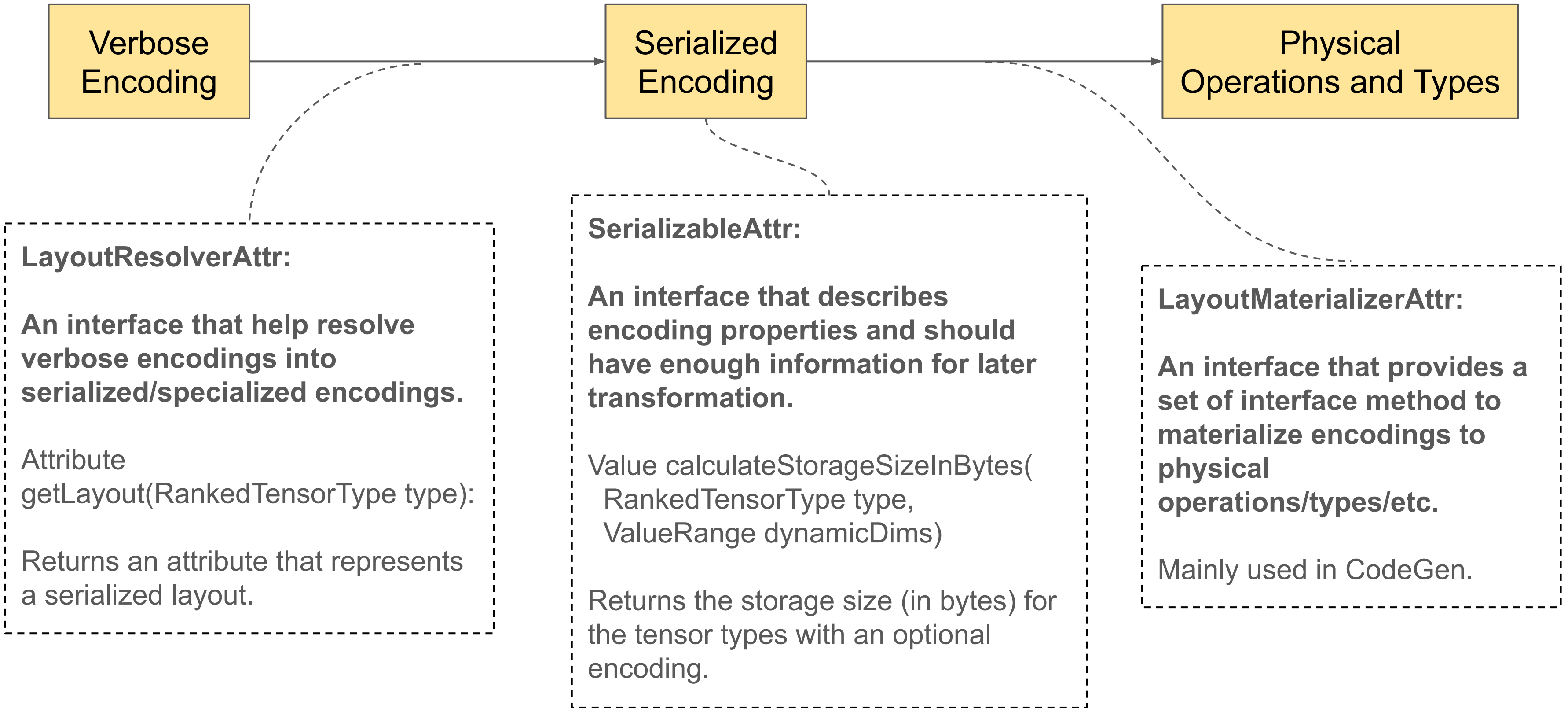
Stream dialect doesn't care about executable target details, but it needs to know affinities for scheduling optimizations, including memory allocation for data-flow between dispatches.

The encoding resolvers implement a couple of interfaces to help lowerings.

```
#executable_target_x86_64 =
  #hal.executable.target<"llvm-cpu", "xyz", {
    iree.encoding.resolver = #iree_cpu.cpu_encoding_layout<>,
    target_triple="x86_64-xyz-xyz",
    cpu_features="+avx512f"}>
util.global private @device_a =
  #hal.device.target<[#executable_target_x86_64]>

util.func {
  stream.tensor.sizeof on(#hal.device.affinity<@device_a>)
    tensor<?x?xf32, #enc>{%d0, %d1} : index
}
```

Encoding States and Interfaces



LayoutResolverAttr:

An interface that help resolve verbose encodings into serialized/specialized encodings.

Attribute getLayout(RankedTensorType type):

Returns an attribute that represents a serialized layout.

SerializableAttr:

An interface that describes encoding properties and should have enough information for later transformation.

```
Value calculateStorageSizeInBytes(
  RankedTensorType type,
  ValueRange dynamicDims)
```

Returns the storage size (in bytes) for the tensor types with an optional encoding.

LayoutMaterializerAttr:

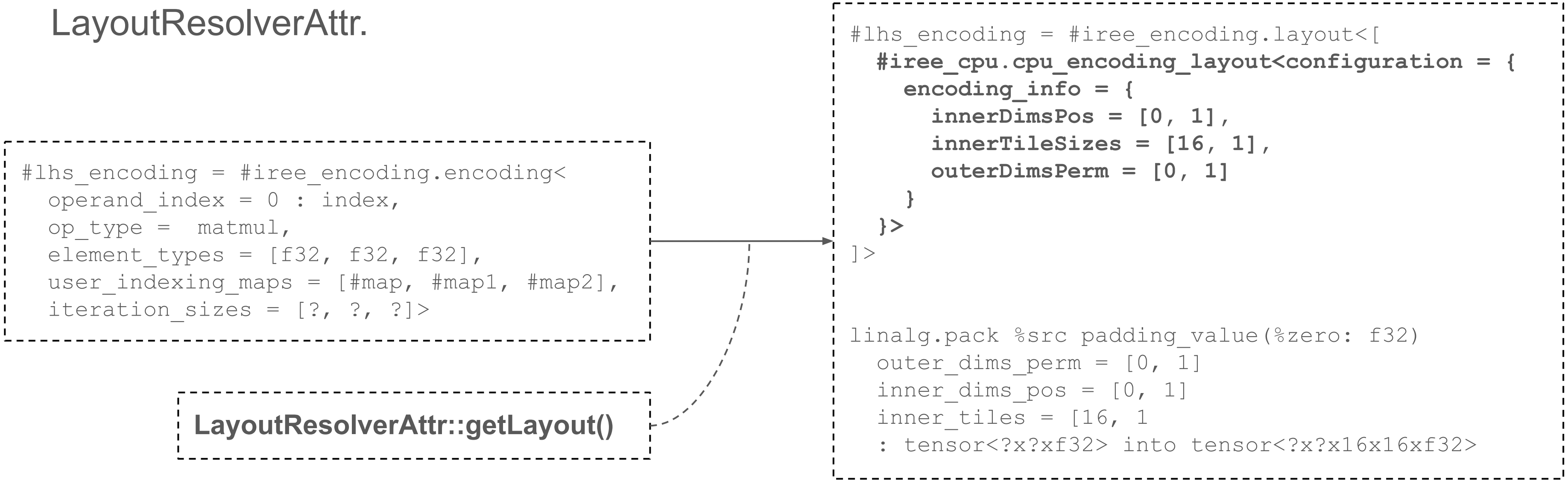
An interface that provides a set of interface method to materialize encodings to physical operations/types/etc.

Mainly used in CodeGen.

Encoding Specialization

IREE runs the SpecializeEncoding pass that converts verbose encodings into serialized encodings using LayoutResolverAttr interface.

The CPU encoding resolver implements LayoutResolverAttr.



```
#lhs_encoding = #iree_encoding.encoding<
  operand_index = 0 : index,
  op_type =  matmul,
  element_types = [f32, f32, f32],
  user_indexing_maps = [#map, #map1, #map2],
  iteration_sizes = [?, ?, ?]>
```

LayoutResolverAttr::getLayout()

```
#lhs_encoding = #iree_encoding.layout<[
  #iree_cpu.cpu_encoding_layout<configuration = {
    encoding_info = {
      innerDimsPos = [0, 1],
      innerTileSizes = [16, 1],
      outerDimsPerm = [0, 1]
    }
  }>
]>


linalg.pack %src padding_value(%zero: f32)
  outer_dims_perm = [0, 1]
  inner_dims_pos = [0, 1]
  inner_tiles = [16, 1
  : tensor<?x?xf32> into tensor<?x?x16x16xf32>
```

Encode Tensors into Storage Formats

The `**stream.tensor.***` ops on tensor-like objects are transformed into encoding-erased asynchronous `**stream.async.***` ops and resolves (if possible) symbolic encoding ops such as `**stream.tensor.sizeof**` into their final values.

```
#lhs_encoding = #iree_encoding.layout<[
  #iree_cpu.cpu_encoding_layout<configuration = {
    encoding_info = {
      innerDimsPos = [0, 1],
      innerTileSizes = [16, 1],
      outerDimsPerm = [0, 1]
    }
  }>
]>

util.func {
  %0 = stream.tensor.sizeof on(#hal.device.affinity<@device_a>)
    tensor<?x?xf32, #lhs_encoding>{%d0, %d1} : index
  util.return %0 : index
}
```

SerializableAttr::
calculateStorageSizeInBytes()

```
util.func {
  %0 = arith.ceildivsi %d0, %c16
  %1 = arith.muli %0, %c16
  %2 = arith.muli %1, %d1
  %3 = arith.muli %2, %c4
  util.return %3 : index
}
```

Encoding Summary

There are categories of encodings:

- **Encoding Type Attribute**: expected to be attached on encoding types. E.g., iree_encoding.encoding<op_type=matmul, …>
- **Encoding Resolver**: used to resolve and transform encoding type attributes.

There are three states of encoding type attribute:

- Verbose encoding
- Serialized encoding
- Physical operations and types

Encoding Summary - Cont

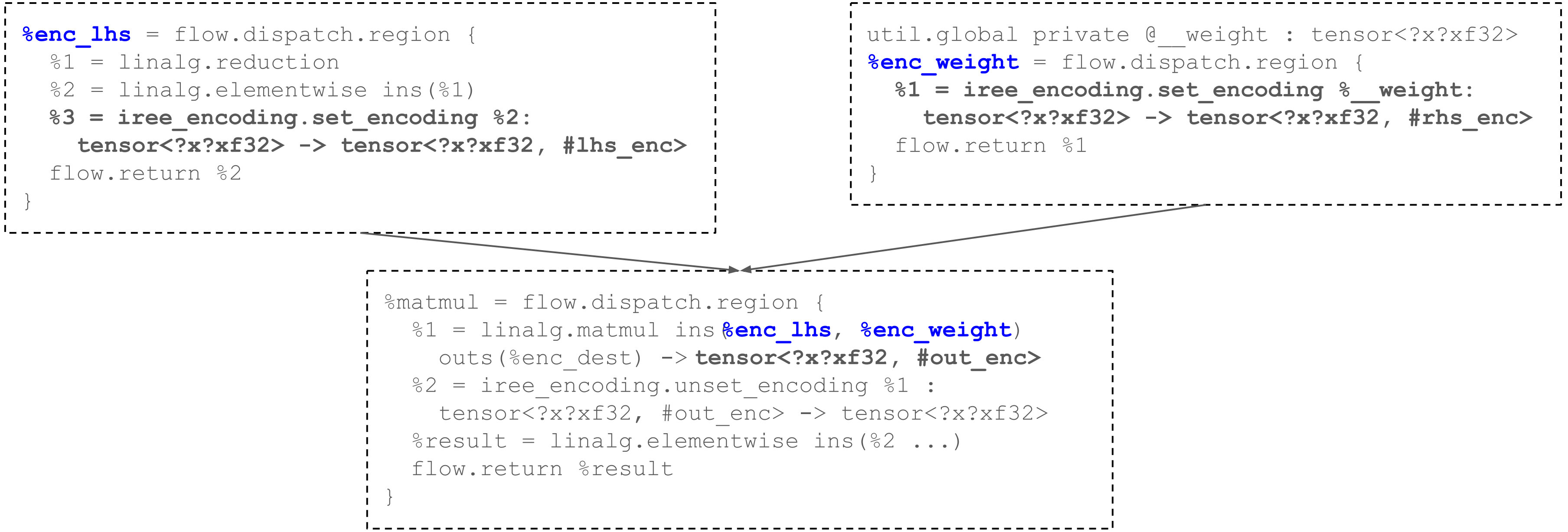
An encoding resolver implements some Encoding interfaces that transform encoding type attributes between the states:

- **LayoutResolverAttr**: resolve verbose encodings into serialized/specialized encodings.
- **SerializableAttr**: describes encoding properties that have enough information for further transformation.
- **LayoutMaterializerAttr**: provides a set of interface methods to materialize encodings to physical operations/types/etc.

Unlocked Optimizations

Fusion, Hoisting, Constant Evaluation

As IREE is a retargetable compiler, it can see the whole program and fuse encoding ops with producers easily; enables hoisting and constant evaluation.



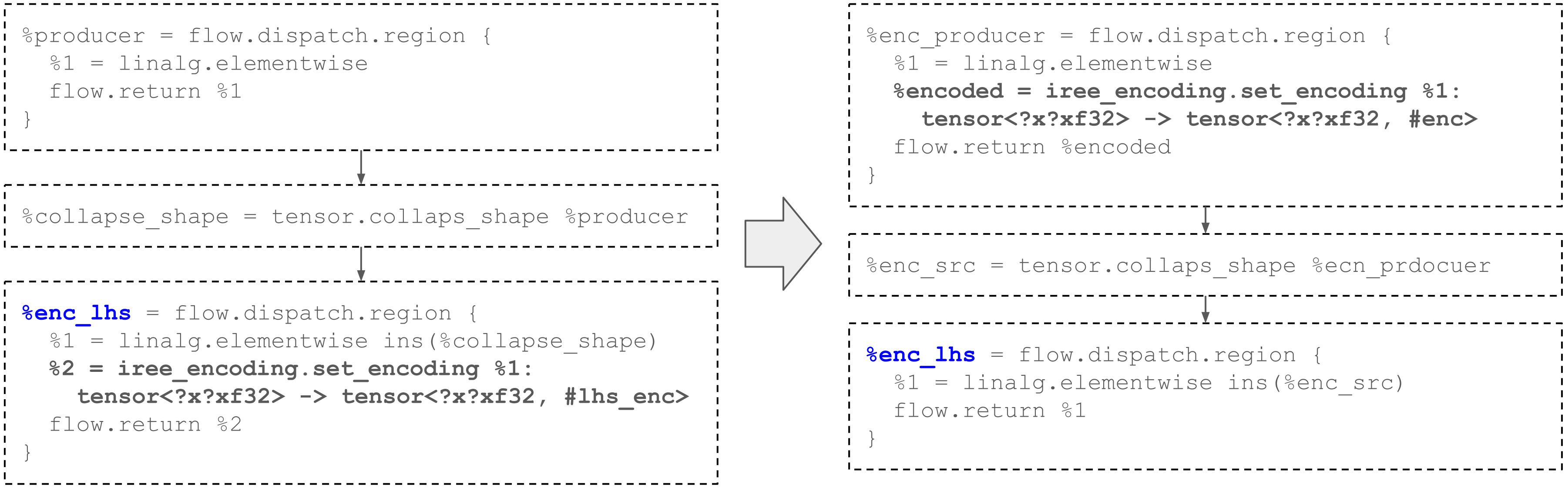
```
%enc_lhs = flow.dispatch.region {
  %1 = linalg.reduction
  %2 = linalg.elementwise ins(%1)
  %3 = iree_encoding.set_encoding %2:
    tensor<?x?xf32> -> tensor<?x?xf32, #lhs_enc>
  flow.return %2
}
```

```
util.global private @__weight : tensor<?x?xf32>
%enc_weight = flow.dispatch.region {
  %1 = iree_encoding.set_encoding %__weight:
    tensor<?x?xf32> -> tensor<?x?xf32, #rhs_enc>
  flow.return %1
}
```

```
%matmul = flow.dispatch.region {
  %1 = linalg.matmul ins(%enc_lhs, %enc_weight)
    outs(%enc_dest) -> tensor<?x?xf32, #out_enc>
  %2 = iree_encoding.unset_encoding %1 :
    tensor<?x?xf32, #out_enc> -> tensor<?x?xf32>
  %result = linalg.elementwise ins(%2 ...)
  flow.return %result
}
```

Encoding Propagation - Working in Progress

Furthermore, we can do encoding propagation globally, which is similar to data-layout propagation.



```
%producer = flow.dispatch.region {
  %1 = linalg.elementwise
  flow.return %1
}
```

```
%collapse_shape = tensor.collaps_shape %producer
```

```
%enc_lhs = flow.dispatch.region {
  %1 = linalg.elementwise ins(%collapse_shape)
  %2 = iree_encoding.set_encoding %1:
    tensor<?x?xf32> -> tensor<?x?xf32, #lhs_enc>
  flow.return %2
}
```

```
%enc_producer = flow.dispatch.region {
  %1 = linalg.elementwise
  %encoded = iree_encoding.set_encoding %1:
    tensor<?x?xf32> -> tensor<?x?xf32, #enc>
  flow.return %encoded
}
```

```
%enc_src = tensor.collaps_shape %ecn_prdocuer
```

```
%enc_lhs = flow.dispatch.region {
  %1 = linalg.elementwise ins(%enc_src)
  flow.return %1
}
```

Multi-Device - Working in Progress

We have been developing the multi-device feature in IREE, including homogeneous computing and heterogeneous computing.

Demo for running matmul chain in heterogeneous computing concept, using data-tiling: https://hackmd.io/@hwPnnvLBTB-JGVMeh-bCEA/rJL0g0JFke

```
func.func @multi_device() -> {
  %matmul_a = linalg.matmul
  %transient_op = flow.tensor.transfer %matmul_a
     : tensor<?x?xf32>{%M, %N} to #hal.device.affinity<@device_b>
  %matmul_b = linalg.matmul
  %add = linalg.elementwise_add ins(%transient_op, matmul_b)
  %result_a = flow.tensor.transfer %add
     : tensor<?x?xf32>{%M, %N} to #hal.device.affinity<@device_a>
  return %result
}
```

Encodings are NOT Data-Tiling Specifics

IREE has the ability to specialize storage size based on logical tensor dimensions with target-specific requirement.

Encoding work is a plus, not a requirement.

We have pad-based encoding strategy that allows IREE allocating larger storage buffer for better cache line fit.

We have been developing split-k and stream-k ideas in IREE compiler concept, that can use the encoding work: https://github.com/iree-org/iree/issues/21012

Future Work

- Move SetEncoding pass from GlobalOptimization phase to DispatchCreation phase.
  - Enable consumer fusion for matmul kernels.
- Add const-evaluation support after DispatchCreation.
- Encoding Propagation.
- Homogeneous Computing + Data-Tiling
- Heterogeneous Computing + Data-Tiling
- https://github.com/iree-org/iree