Google DeepMind

Shardy

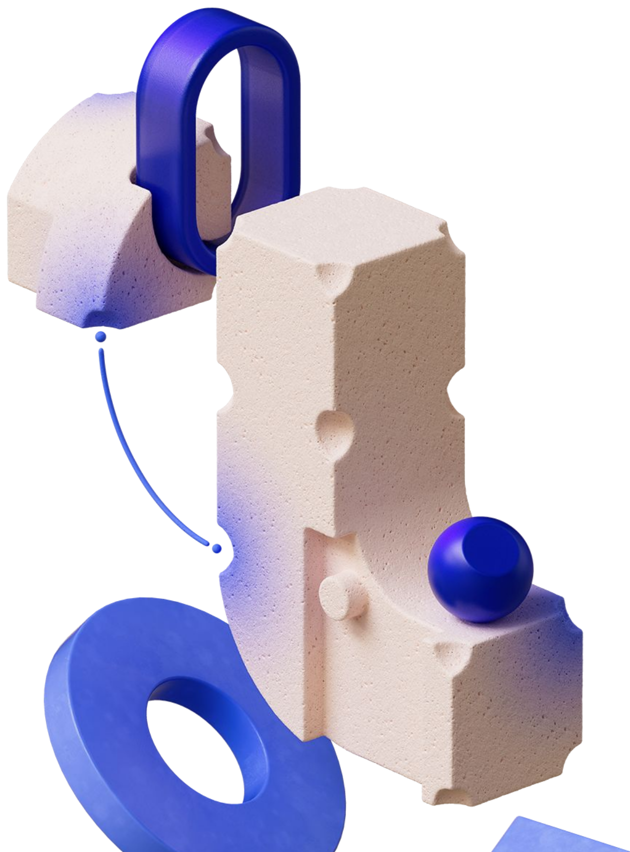
An MLIR-based Tensor Partitioning System for All Dialects

Bart Chrzaszcz
Google DeepMind

Zixuan Jiang
Google Core ML

LLVM Developers' Meeting 2024

Google DeepMind



Background on AI model scaling

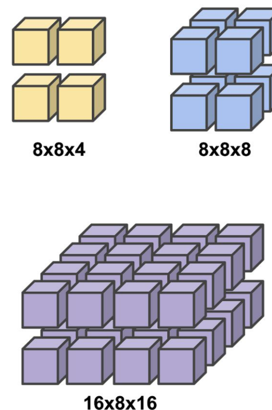
Training Generative AI Models

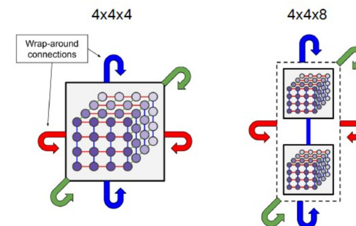
- Generative AI models are large
- They rely on huge matrix multiplications
- They are too large to fit on a single device, let alone host
- Training and serving these models requires distributing them across thousands of devices.
- **But how is this distribution achieved?**

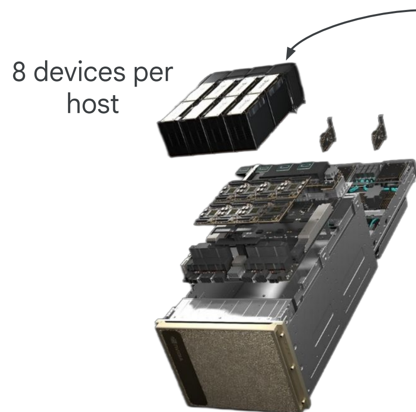
The Mesh: Physical

- Generative AI models are large
- They rely on huge matrix multiplications
- They are too large to fit on a single device, let alone host
- Training and serving these models requires distributing them across thousands of devices.
- **But how is this distribution achieved?**

Physical TPU layout



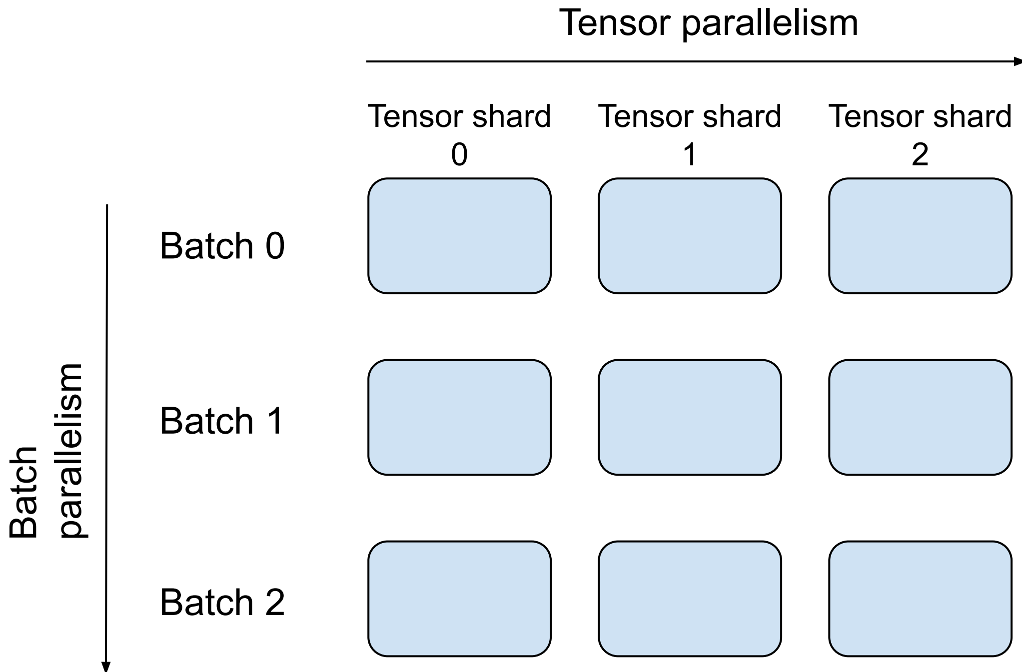




Cross device parallelism

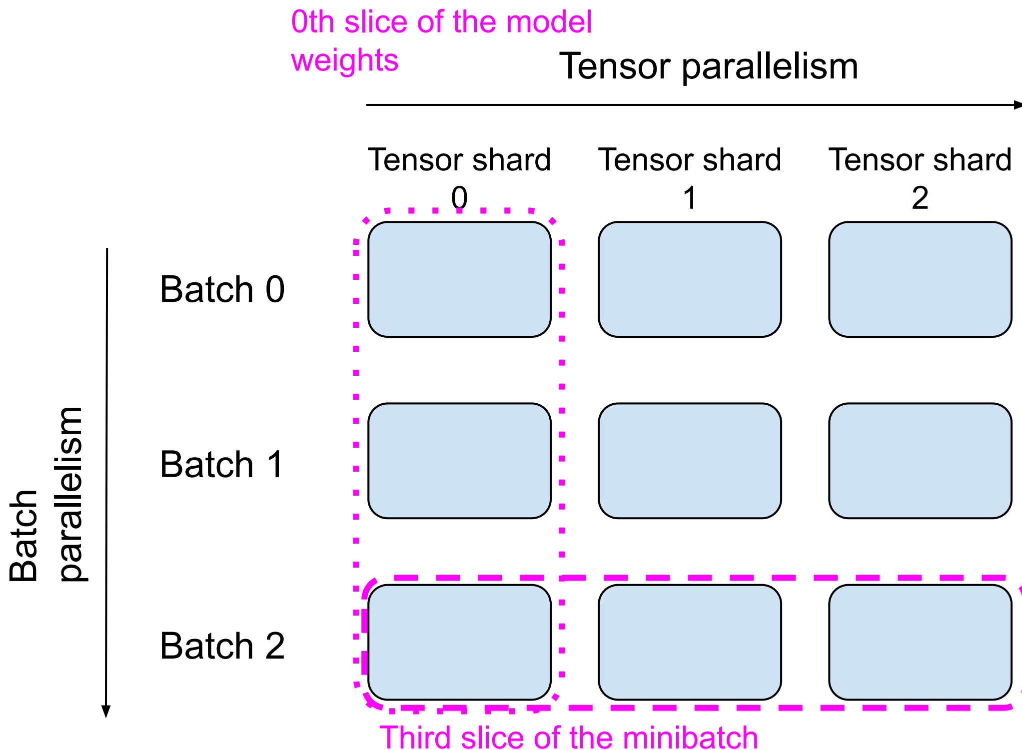
The Mesh: Logical

- Taking the connection speeds across hosts and devices, need to optimize the device order in tensor programs optimally
- Done in the “logical mesh”
- Batch parallelism:
  - Split images/text/examples
  - Can parallelize the predictions
- Tensor parallelism:
  - Size of the model is too big, split tensors across devices
  - parallel matrix multiplications



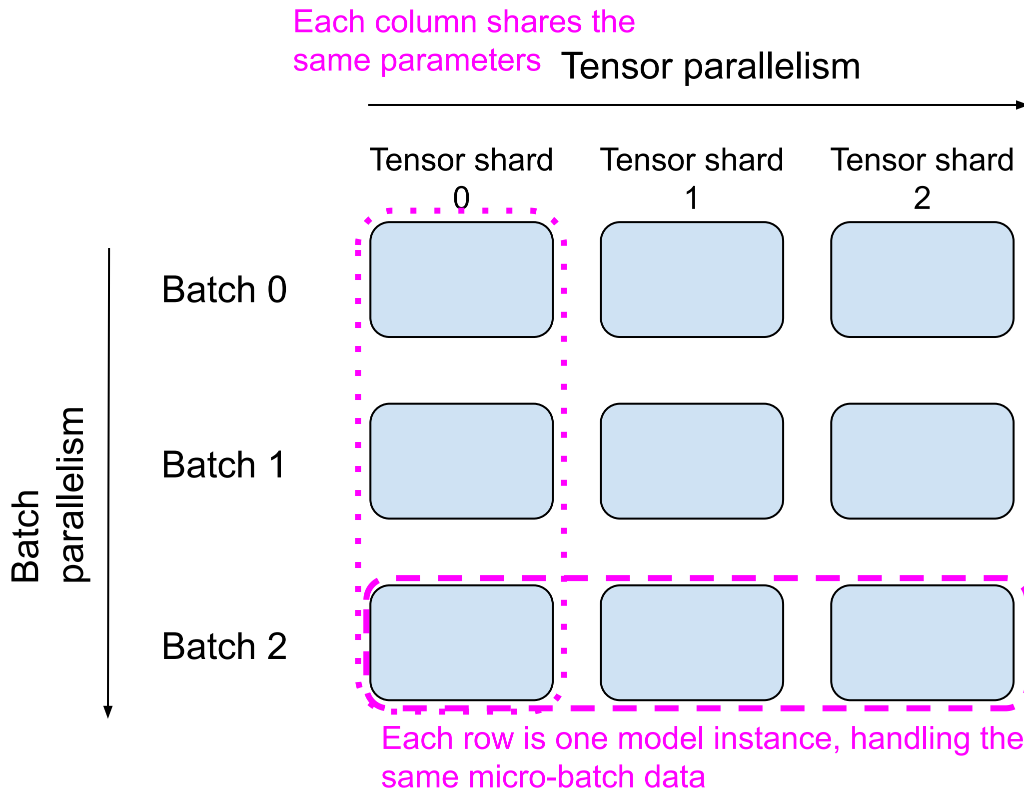
The Mesh: Logical

- Taking the connection speeds across hosts and devices, need to optimize the device order in tensor programs optimally
- Done in the “logical mesh”
- Batch parallelism:
  - Split images/text/examples
  - Can parallelize the predictions
- Tensor parallelism:
  - Size of the model is too big, split tensors across devices
  - parallel matrix multiplications



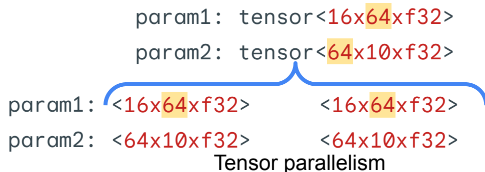
The Mesh: Logical

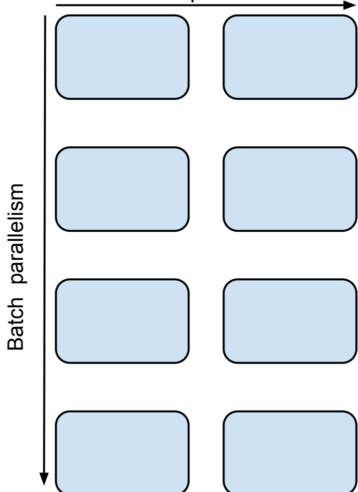
- Taking the connection speeds across hosts and devices, need to optimize the device order in tensor programs optimally
- Done in the “logical mesh”
- Batch parallelism:
  - Split images/text/examples
  - Can parallelize the predictions
- Tensor parallelism:
  - Size of the model is too big, split tensors across devices
  - parallel matrix multiplications



How models are scaled: sharding propagation and partitioning

- Tensor parallelism: calculate 2 matmuls in parallel before all-reducing them together





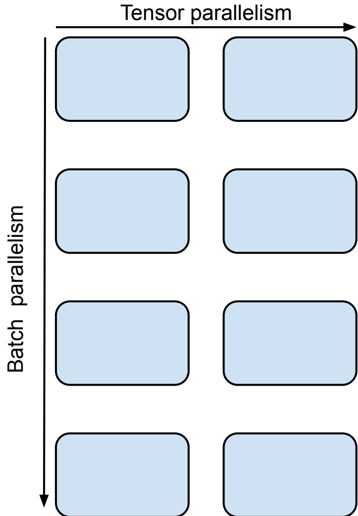

```
mesh @mesh = <"batch"=4, "model"=2>

func.func public @predict(
  %samples: tensor<4x16xf32>,
  %param1: tensor<16x64xf32>,
  %param2: tensor<64x10xf32>) -> tensor<4x10xf32> {
  %0 = stablehlo.dot_general %samples, %param1,
      contracting_dims = [1] x [0]
      : (tensor<4x16xf32>, tensor<16x64xf32>)
         -> tensor<4x64xf32>
  %1 = stablehlo.dot_general %0, %param2,
      contracting_dims = [1] x [0]
      : (tensor<4x64xf32>, tensor<64x10xf32>)
        -> tensor<4x10xf32>
  return %1 : tensor<4x10xf32>
}
```

How models are scaled: sharding propagation and partitioning

- Tensor parallelism: calculate 2 matmuls in parallel before all-reducing them together

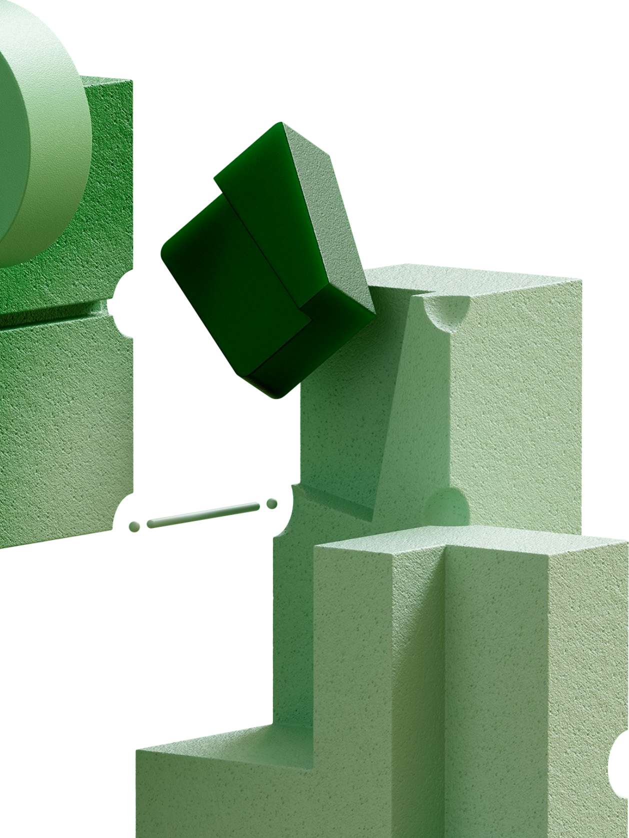
param1: tensor<16x~~64~~32xf32>
param2: tensor<~~64~~32x10xf32>

param1: <16x~~64~~32xf32> <16x~~64~~32xf32>
param2: <64x10xf32> <64x10xf32>




```
mesh @mesh = <"batch"=4, "model"=2>

func.func public @predict(
  %samples: tensor<4x16xf32>,
  %param1: tensor<16x~~64~~32xf32>,
  %param2: tensor<~~64~~32x10xf32>) -> tensor<4x10xf32> {
  %0 = stablehlo.dot_general %samples, %param1,
      contracting_dims = [1] x [0]
      : (tensor<4x16xf32>, tensor<16x~~64~~32xf32>)
        -> tensor<4x~~64~~32xf32>
  %1 = stablehlo.dot_general %0, %param2,
      contracting_dims = [1] x [0]
      : (tensor<4x~~64~~32xf32>, tensor<~~64~~32x10xf32>)
       -> tensor<4x10xf32>
  %2 = stablehlo.all_reduce %1 : tensor<4x10xf32>
  return %2 : tensor<4x10xf32>
}
```

Google DeepMind



Existing Compiler Systems

Existing Tensor Sharding Propagation Systems

xla::GSPMD

- Sharding attribute based propagation
- No concept of a mesh
- No named axes, only sizes
- Op priority propagation
- How to propagate through ops is hardcoded
- Extensive conflict resolution

PartIR (deprecated)

- Loop based propagation
- Top level mesh
- Axis names
- User/round based propagation (propagate certain sharding around in different order)
- C++ data structure that defines how to propagate through an op
- User priorities to resolve conflicts

Mesh Dialect

- Sharding attribute+op based propagation (inserts explicit sharding ops)
- Top level mesh
- No named axes, only sizes
- No order of propagation (all ops/sharding at once)
- `ShardingInterface` defining how to propagate through an op
- No conflict resolution?

Existing Tensor Sharding Propagation Systems

xla::GSPMD

- Sharding attribute based propagation
- No concept of a mesh
- No named axes, only sizes
- Op priority propagation
- How to propagate through ops is hardcoded
- Extensive conflict resolution

PartIR (deprecated)

- Loop based propagation
- Top level mesh
- Axis names
- User/round based propagation (propagate certain sharding around in different order)
- C++ data structure that defines how to propagate through an op
- User priorities to resolve conflicts

Mesh Dialect

- Sharding attribute+op based propagation (inserts explicit sharding ops)
- Top level mesh
- No named axes, only sizes
- No order of propagation (all ops/sharding at once)
- `ShardingInterface` defining how to propagate through an op
- No conflict resolution?

Shardy

- Sharding attribute based propagation **(GSPMD)**
- Top level mesh **(PartIR)**
- Axis names **(PartIR)**
- Op **(GSPMD)** and user **(PartIR)** priority propagation
- Various interfaces for propagation **(Mesh)**
- Conflict resolution **(GSPMD)**

Google DeepMind

Representation and APIs

Sharding Representation: Overview

- An attribute of operation
- It implies that how the results are partitioned.

```
<@mesh, [{"w", "x"}, {}]>
```

- The sharding is bound to the logical mesh with name `@mesh`.
- The 1st tensor dimension is sharded along "w" then further along "x".
- The 2nd tensor dimension is replicated.
- The tensor is replicated along "y" and "z".

```
@mesh = <"x"=2, "y"=4, "z"=2, "w"=2>

// shape on each device (local shape) is tensor<1x8xf32>
%arg0: tensor<4x8xf32> {sdy.sharding = <@mesh, [{"w", "x"}, {}]>}
```

Sharding Representation: Constraining Axes and Dims

- **Explicitly replicated axes** cannot be used to partition the tensor.
- **Implicitly replicated axes** can be used to further partition the tensor.

- **Open dimensions** can be further sharded on available axes.
- **Closed dimensions** are fixed and can't be further sharded.

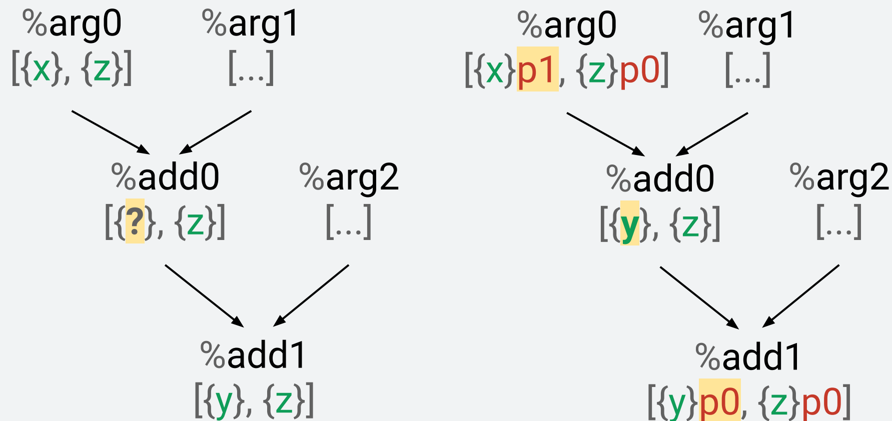
Shardy only propagates **implicitly replicate axes** to **open dimensions**.

```
@mesh = <"x"=2, "y"=4, "z"=2, "w"=2>


tensor<4x8xf32> {sdy.sharding=<@mesh,
    [{"w"}, # The first dim is closed
     {"x", ?}], # The second dim is open
    replicated={"y"}>} # explicitly replicated axes
```

Sharding Representation: User Priorities

- Determine the propagation order -> more user control & better debuggability.
- Example: batch parallelism -> Megatron -> ZeRO.
- Can be attached to dimension shardings.

```
@mesh = <"x"=2, "y"=4, "z"=2>

%arg0 : tensor<4x8xf32>
{sdy.sharding = <@mesh,[{"x"}p1,{"z"}p0]>}

%add1 : tensor<4x8xf32>
{sdy.sharding = <@mesh,[{"y"}p0,{"z"}p0]>}
```



Sharding Rule

- Tell Shardy how an operation should be propagated through
- A dimension can decompose into multiple factors
- We propagate shardings
  - **Forward propagation** (operands -> results)
  - **Backward propagation** (results -> operands)
  - **Sideways propagation** (operand A -> operand B, result A -> result B)

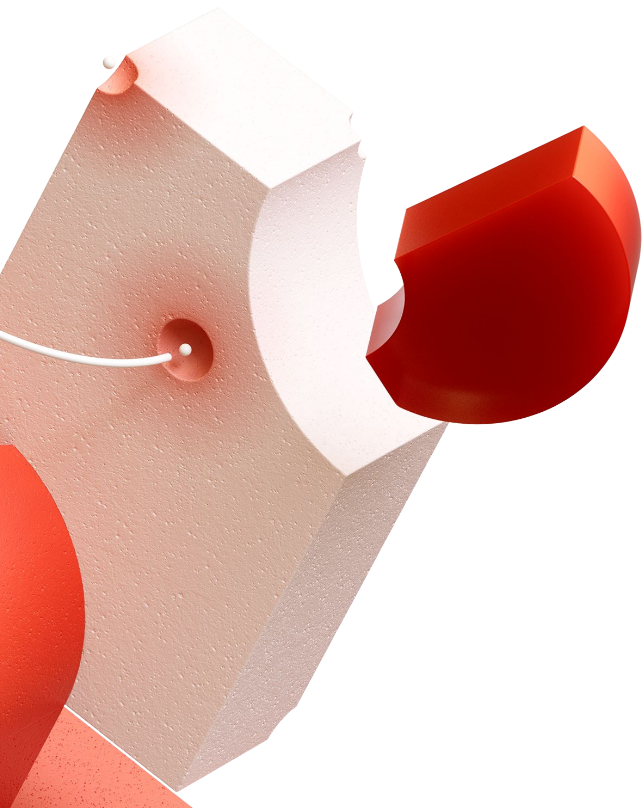
Similar to einsum notation

```
%dot = stablehlo.dot_general %lhs, %rhs,
    batching_dims = [0] x [0], contracting_dims =
[2] x [1],
    {sdy.sharding_rule =
        <([i, j, l], [i, l, k])->([i, j, k])
        {i=4, j=8, k=16, l=32}>} :
    (tensor<4x8x32xf32>, tensor<4x32x16xf32>)
    -> tensor<4x8x16xf32>
```

Compound factors

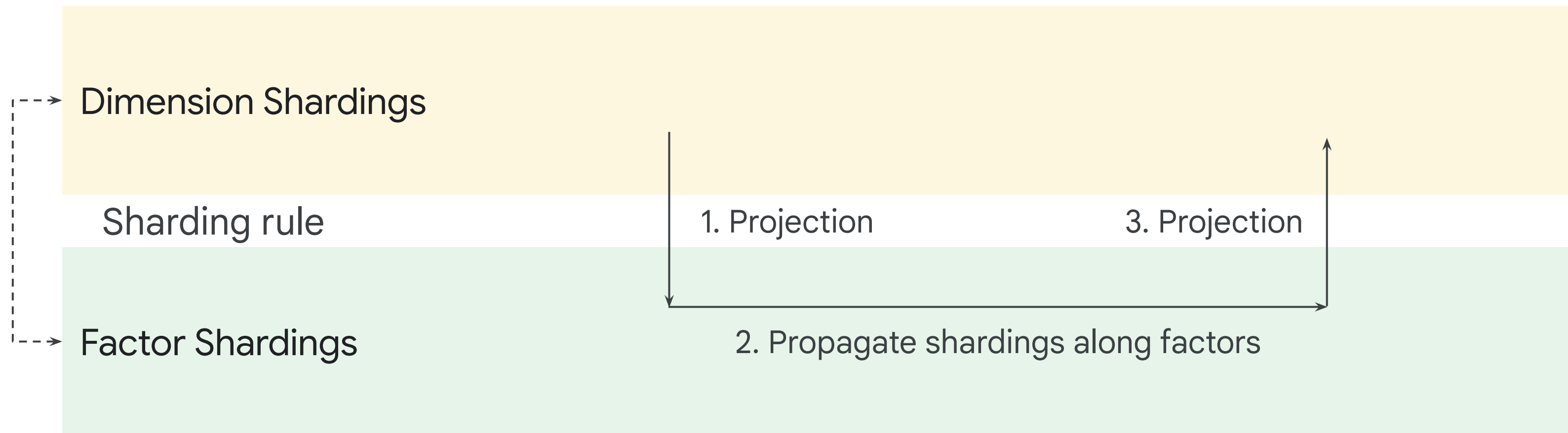
```
%reshape = stablehlo.reshape %arg0,
    {sdy.sharding_rule = <
        ([ij, k, l])->([i, jk, l])
        {i=2, j=4, k=4, l=5}>} :
    (tensor<8x4x5xf32>) -> tensor<2x16x5xf32>
```

Google DeepMind



Propagation Algorithm

Propagate shardings along factors



Propagate shardings in dot

```
@mesh = <"batch"=4, "tensor"=4>

%dot = stablehlo.dot_general %lhs, %rhs, contracting_dims = [1] x [0] :
(tensor<8x32xf32>, tensor<32x16xf32>) -> tensor<8x16xf32>
```

- `%lhs` sharding `[{“batch”, ?}, {“tensor”, ?}]`
- `%rhs` sharding `[{?}, {?}]`
- `%dot` sharding `[{?}, {?}]`

Step 1. Dimensions -> Factors

```
@mesh = <"batch"=4, "tensor"=4>

%dot = stablehlo.dot_general %lhs, %rhs, contracting_dims = [1] x [0] :
(tensor<8x32xf32>, tensor<32x16xf32>) -> tensor<8x16xf32>
```

- `%lhs` sharding `[{“batch”, ?}, {“tensor”, ?}]`
- `%rhs` sharding `[{?}, {?}]`
- `%dot` sharding `[{?}, {?}]`
- **Sharding rule:** `[i, k], [k, j]->[i, j], {i=8, j=16, k=32}`

Step 1. Dimensions -> Factors

```
@mesh = <"batch"=4, "tensor"=4>

%dot = stablehlo.dot_general %lhs, %rhs, contracting_dims = [1] x [0] :
(tensor<8x32xf32>, tensor<32x16xf32>) -> tensor<8x16xf32>
```

- `%lhs` sharding `[{“batch”, ?}, {“tensor”, ?}]`
- `%rhs` sharding `[{?}, {?}]`
- `%dot` sharding `[{?}, {?}]`
- **Sharding rule:** `[i, k], [k, j]->[i, j], {i=8, j=16, k=32}`

	Factor i	Factor j	Factor k
LHS	“batch”	n/a	“tensor”
RHS	n/a		
Result			n/a

Step 2. Propagate shardings along factors

```
@mesh = <"batch"=4, "tensor"=4>

%dot = stablehlo.dot_general %lhs, %rhs, contracting_dims = [1] x [0] :
(tensor<8x32xf32>, tensor<32x16xf32>) -> tensor<8x16xf32>
```

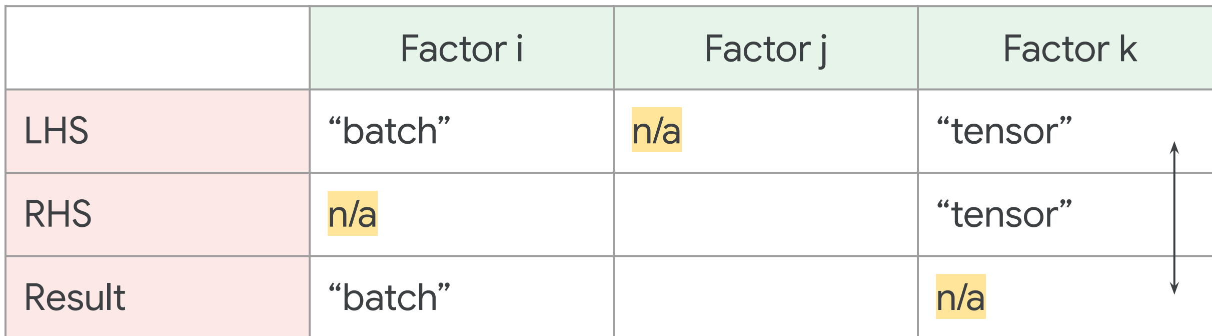
- `%lhs` sharding `[{“batch”, ?}, {“tensor”, ?}]`
- `%rhs` sharding `[{?}, {?}]`
- `%dot` sharding `[{?}, {?}]`
- **Sharding rule:** `[i, k], [k, j]->[i, j], {i=8, j=16, k=32}`

	Factor i	Factor j	Factor k
LHS	“batch”	n/a	“tensor”
RHS	n/a		
Result	“batch”		n/a

Step 2. Propagate shardings along factors

```
@mesh = <"batch"=4, "tensor"=4>

%dot = stablehlo.dot_general %lhs, %rhs, contracting_dims = [1] x [0] :
(tensor<8x32xf32>, tensor<32x16xf32>) -> tensor<8x16xf32>
```

- `%lhs` sharding `[{“batch”, ?}, {“tensor”, ?}]`
- `%rhs` sharding `[{?}, {?}]`
- `%dot` sharding `[{?}, {?}]`
- **Sharding rule:** `[i, k], [k, j]->[i, j], {i=8, j=16, k=32}`



	Factor i	Factor j	Factor k
LHS	“batch”	n/a	“tensor”
RHS	n/a		“tensor”
Result	“batch”		n/a

Step 3. Factors -> Dimensions

- **Sharding rule:** `[i, k], [k, j]->[i, j], {i=8, j=16, k=32}`

	Factor i	Factor j	Factor k
LHS	"batch"	n/a	"tensor"
RHS	n/a		"tensor"
Result	"batch"		n/a

Step 3. Factors -> Dimensions

- **Sharding rule:** `[i, k], [k, j]->[i, j], {i=8, j=16, k=32}`

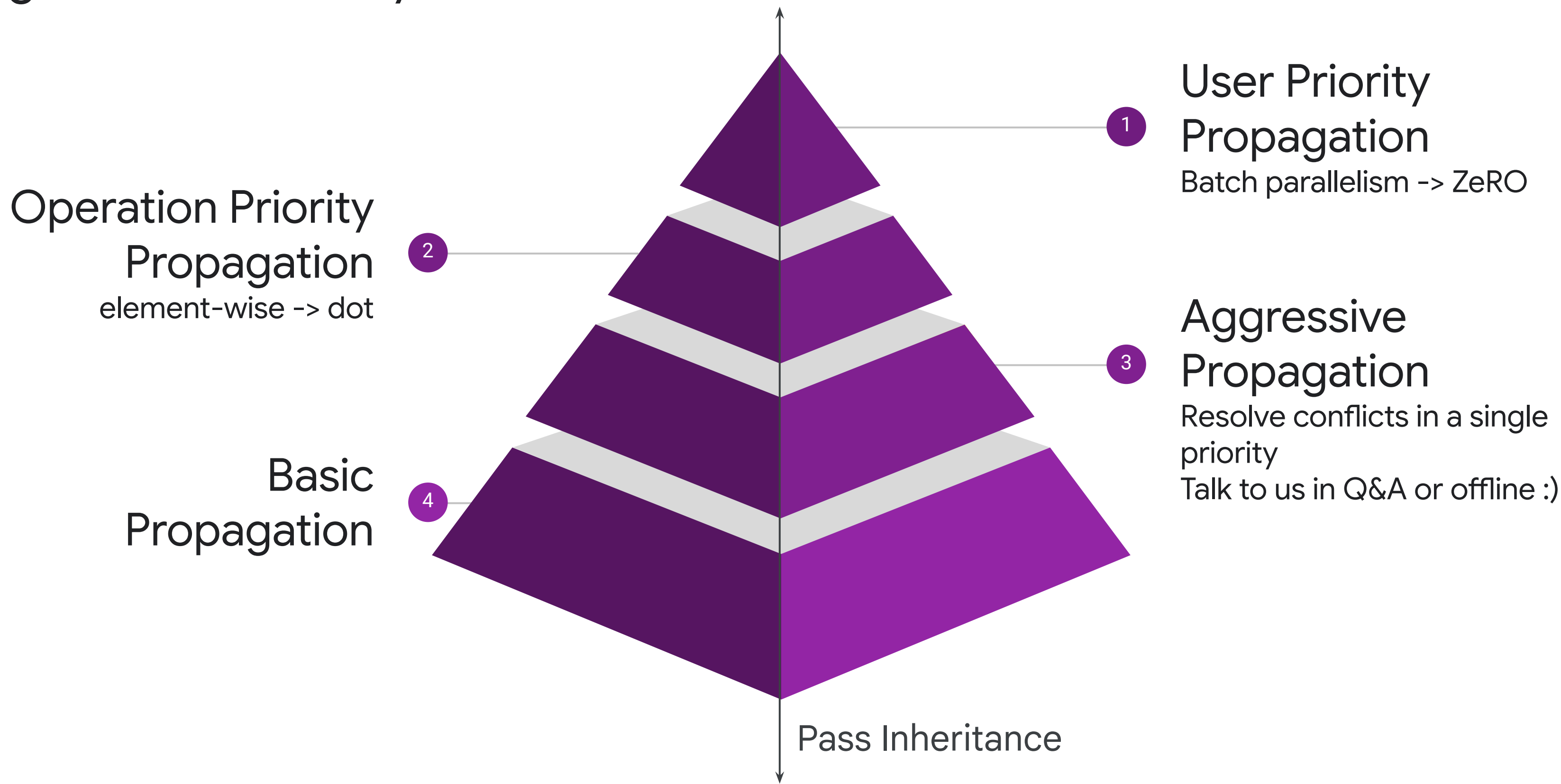
	Factor i	Factor j	Factor k
LHS	“batch”	n/a	“tensor”
RHS	n/a		“tensor”
Result	“batch”		n/a

After propagation

```
%dot = stablehlo.dot_general %lhs, %rhs, contracting_dims = [1] x [0] :
(tensor<8x32xf32>, tensor<32x16xf32>) -> tensor<8x16xf32>
```

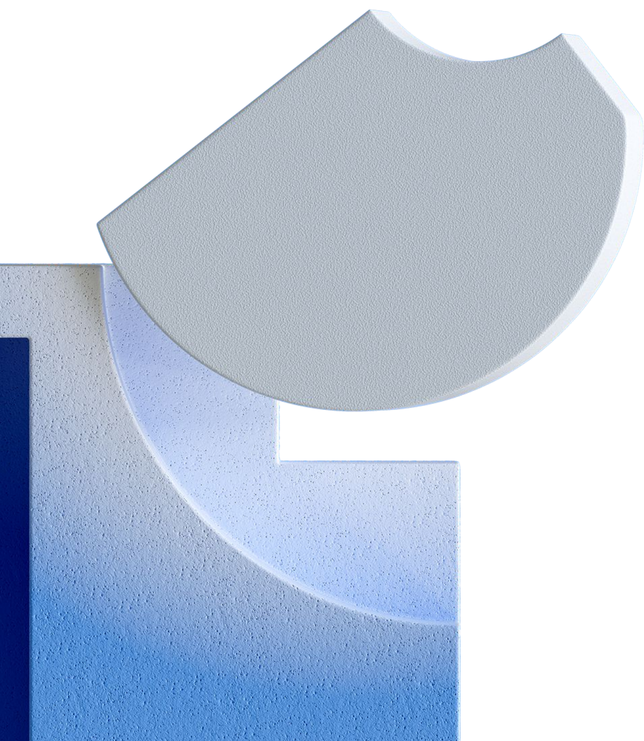
- `%lhs` sharding `[{“batch”, ?}, {“tensor”, ?}]`
- `%rhs` sharding `[{“tensor”, ?}, {?}]`
- `%dot` sharding `[{“batch”, ?}, {?}]`

What if there are conflicts?
Algorithm Hierarchy



Google DeepMind

Being Dialect Agnostic



Being Dialect Agnostic

Let Shardy be used by any MLIR dialect

- This is a long-term goal with a strategic plan in place.
- Currently Shardy depends on StableHLO.
- We aim to eliminate this dependency to maximize Shardy's flexibility.

- Shardy will provide a variety of interfaces and traits.
- Dialect owners can easily integrate these into their own ops.

Sharding Rules

- Currently: Shardy depends on StableHLO and define sharding rules for each op.
- Future: users of Shardy have ops implement this interface to define their own sharding rules

```
def ShardingRuleOpInterface :
OpInterface<"ShardingRuleOpInterface"> {
  let methods = [
    InterfaceMethod<
      /*desc=*/[{
        Returns the sharding rule of the op.
      }],
      /*retType=*/"mlir::sdy::OpShardingRuleAttr",
      /*methodName=*/"getShardingRule"
    >,
  ];
}
```

Region based ops

- Used for: while loops, case, optimization barriers, region based ops, etc.
- Skipping method details for brevity, talk to us offline :)

```
def ShardableDataFlowOpInterface :
    OpInterface<"ShardableDataFlowOpInterface"> {
  (get|set)BlockArgumentEdgeOwnerShardings;
  (get|set)OpResultEdgeOwnerShardings;
  getBlockArgumentEdgeOwners;
  getOpResultEdgeOwners;
  getEdgeSources;
  // ...
}
```

```
%0:2 = stablehlo.while(%iterArg = %arg0, %iterArg_2 = %c)
  : tensor<32x96xf32>, tensor<i32>
  cond {
  // ...
  stablehlo.return %3 : tensor<i1>
} do {
  // ...
  stablehlo.return %4, %3 : tensor<32x96xf32>, tensor<i32>
}
```

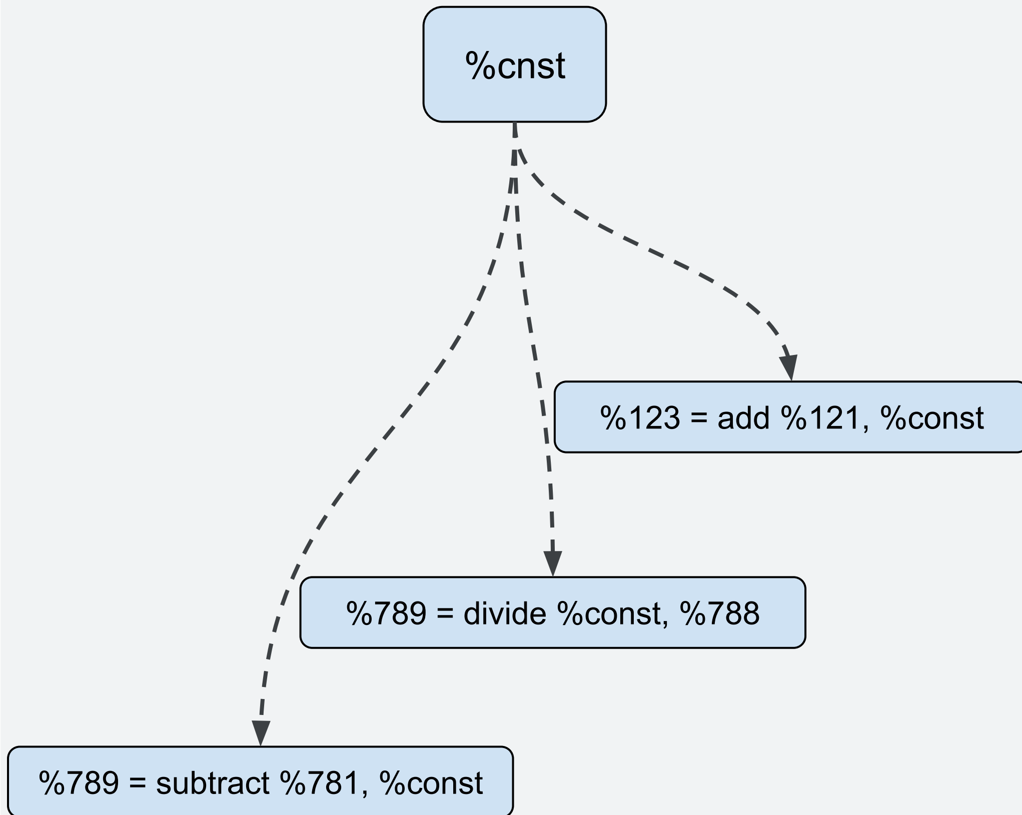
Constant splitting

Want unique constants per use for optimal sharding

- Don't want shardings to propagate through a constant due to multiple uses (false dependency)
- Each use can have a different sharding that can propagate in isolation to its own copy of the constant sub-computation.

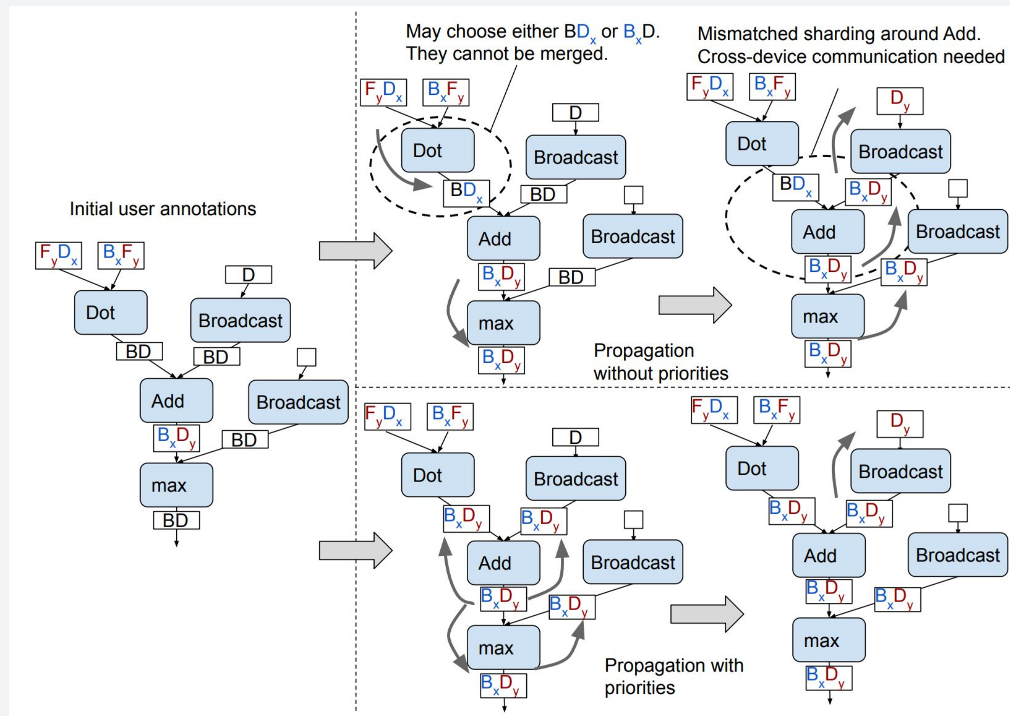
Shardy users need to define:

- `your_dialect.constant` -> `sdy.constant` pass
- `sdy::ConstantLike` trait, such as iota ops
- `mlir::Elementwise` trait for element-wise ops like add and multiply
- `sdy::ConstantFoldable` for ops like slice/broadcast. These ops can technically be calculated at compile time, if all their operands/results are constants.



Op priorities

- GSPMD (and Shardy) defines a pre-registered order of what ops get propagated around first
    - Element-wise -> broadcasts -> matmuls -> …
- Currently hard coded in Shardy on StableHLO ops
- Plan: tell us in what order (and direction*) to propagate ops



See GSPMD paper for why op priorities are important

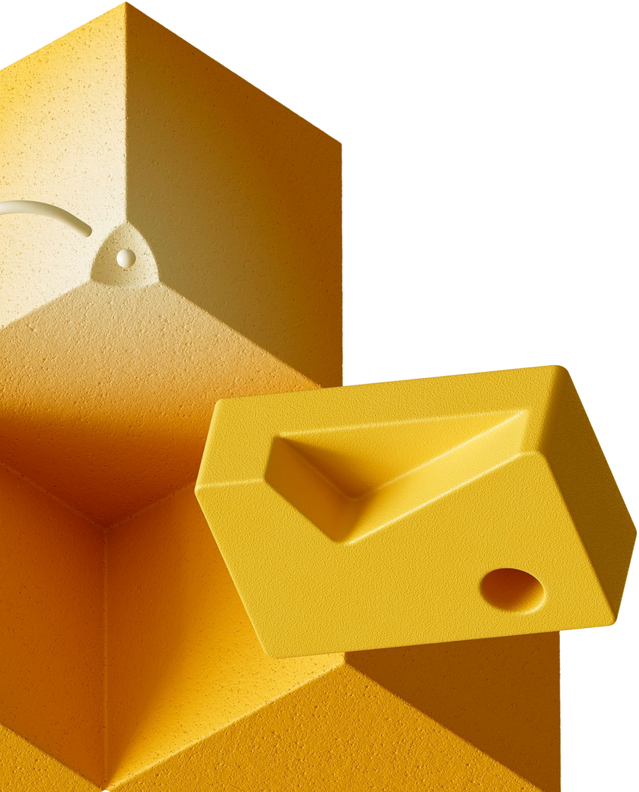
** Direction of propagation is sometimes important as well, see the GSPMD paper!*

Being Dialect Agnostic

As long as you implement the previous interfaces, traits, and pass, **Shardy will be able to work for your dialect!**

Google DeepMind

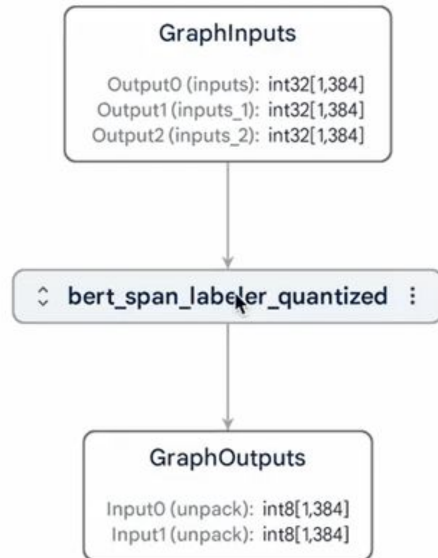
Debugging



Model Explorer

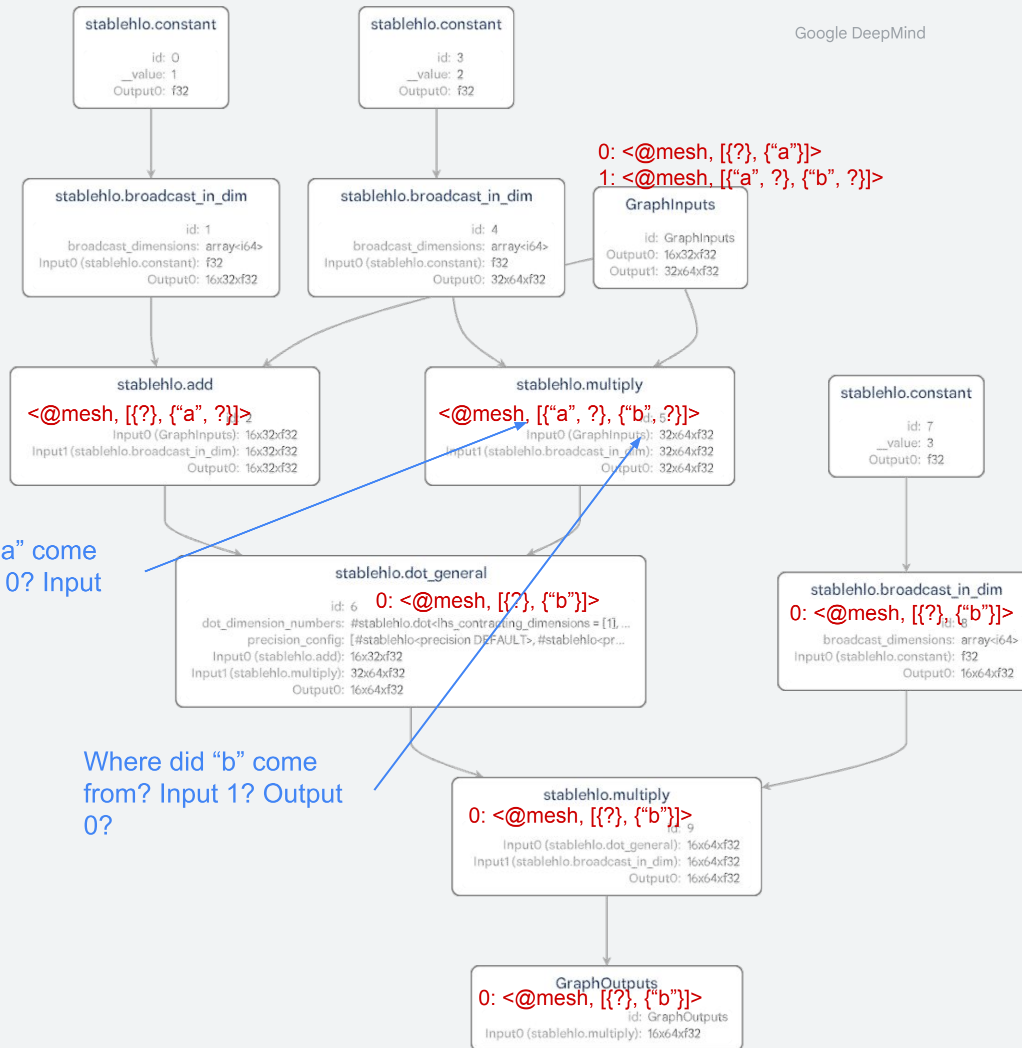
- powerful graph visualization tool that helps one understand, debug, and optimize ML models
- combines graphics techniques used in 3D game and animation production, adapts them for ML graph rendering

See more at
research.google/blog/model-explorer



Trace the shardings

- Given a fully propagated program, how can we determine what caused an SSA value to be sharded?
- Want to know what input/output/intermediate sharding specified by the user caused an op to be sharded a certain way

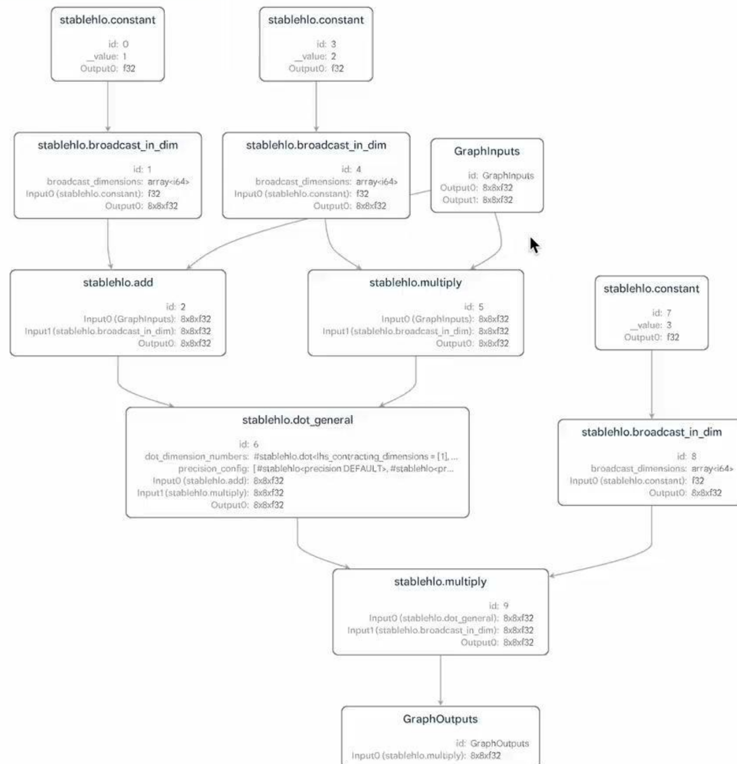


Trace the trajectory

Model Explorer

Add per-node data **demo.mlir** main 12 nodes

Search nodes/layers by regex



GRAPH INFO

op node count 12

layer count 0

Op

Layer

Zoom Ctrl+Scroll

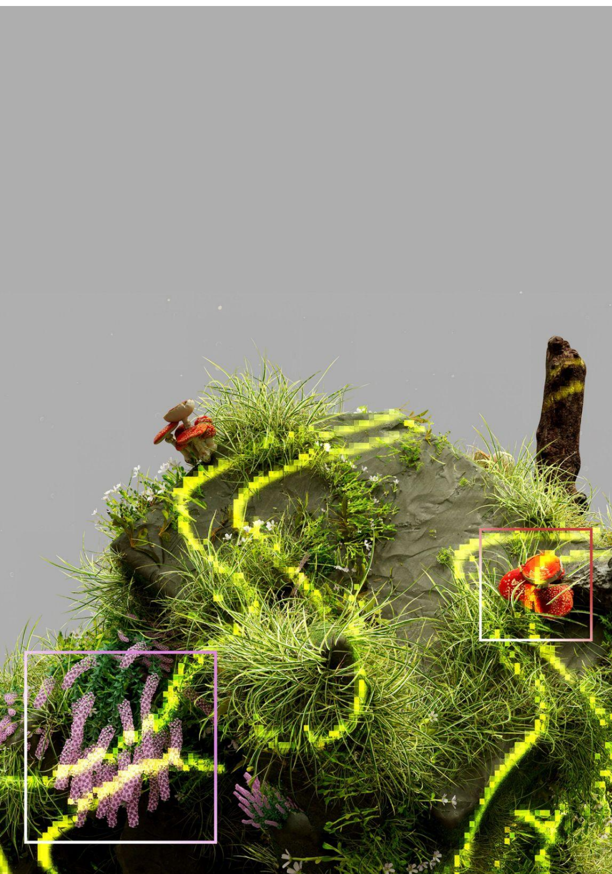
Pan Drag or scroll

Using MLIR Action Tracing

- Execute an action to save metadata about what operand/result caused a Value to be sharded a certain way
- Use `ValueToSourceMap` data structure to build the axis subgraphs.

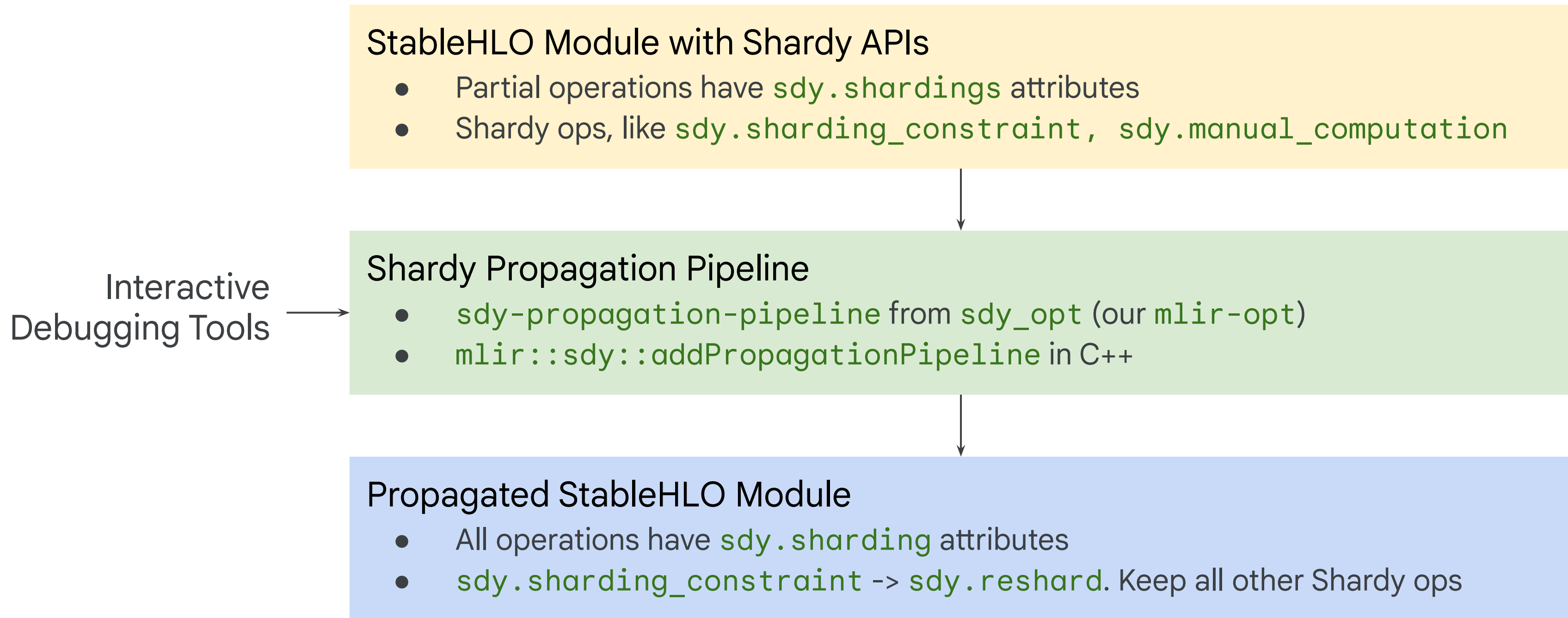
```
class SourceShardingAction : public tracing::ActionImpl<SourceShardingAction>
{
public:
  using Base = tracing::ActionImpl<SourceShardingAction>;

  // Stores a mapping of how an op's result was updated: save which
  // operand/result caused the update on which axis
  ValueToSourceValueMap valueToSourceValue;
};

// ---------------------------------------------------

class SourceShardingHandler {
  // Intercept action and save them per Value for the entire program
  ValueToSourceValueMap valueToSourceValue;
};

// ---------------------------------------------------

LogicalResult Propagate::matchAndRewrite(
    Operation* op, PatternRewriter& rewriter) {
  context->executeAction<SourceShardingAction>(
      [&]() {
        updateShardings(...);
      },
      /*IRUnits=*/{...},
      ...);
}
```

Google DeepMind



Using Shardy Today

Using Shardy Today



Example: JAX -> Shardy -> XLA

JAX

JAX can lower to Shardy representations and APIs with:

```
jax.config.update("jax_use_shardy_partitioner", True)

jax.lax.with_sharding_constraint(x, NamedSharding(jax.sharding.Mesh,
PartitionSpec('data')))
```

Shardy

```
sdy.mesh @mesh = <["data"=4, "model"=2]>
%0 = sdy.sharding_constraint %arg0 <@mesh, [{"data"}]> : tensor<32xf32>
```

Apply these constraints and propagate shardings.

XLA

XLA partitions the exported HLO module and generates machine code.
Talk to us in Q&A or offline :)

github.com/openxla/xla/tree/main/xla/service/spmd/shardy

TPUs/GPUs/CPUs

Future Plans

Shardy Partitioner

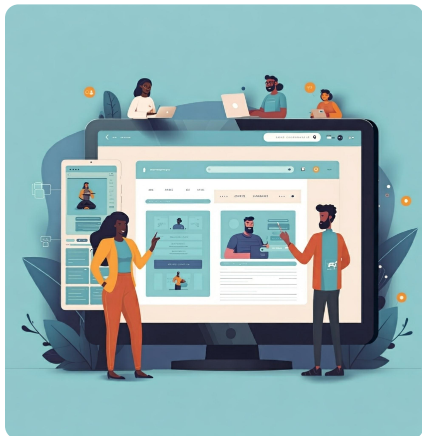
Other ML Frameworks: PyTorch

Bazel + CMake

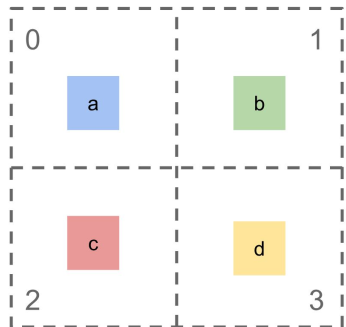
StableHLO -> Dialect Agnostic

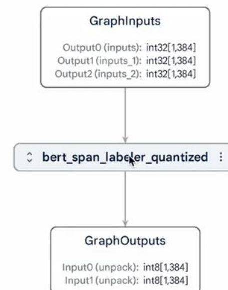
Conclusion

Shardy is a new partitioning system.



```
sharding<@mesh_xy, [{"x"}, {"y"}]>
```





100% open source

New representations and APIs

Dialect agnostic

Interactive debugging

Google DeepMind

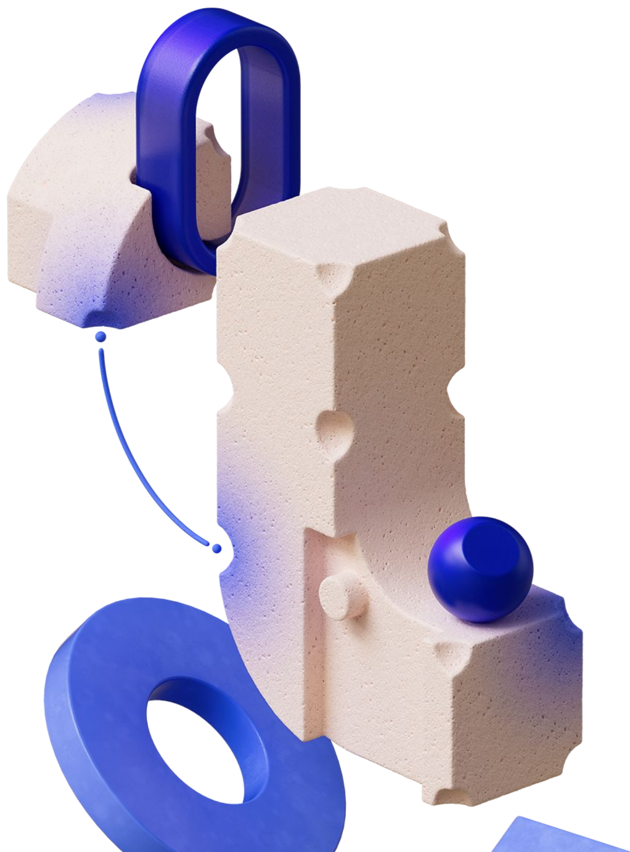
Thank you!

github.com/openxla/shardy

Google DeepMind

Appendix

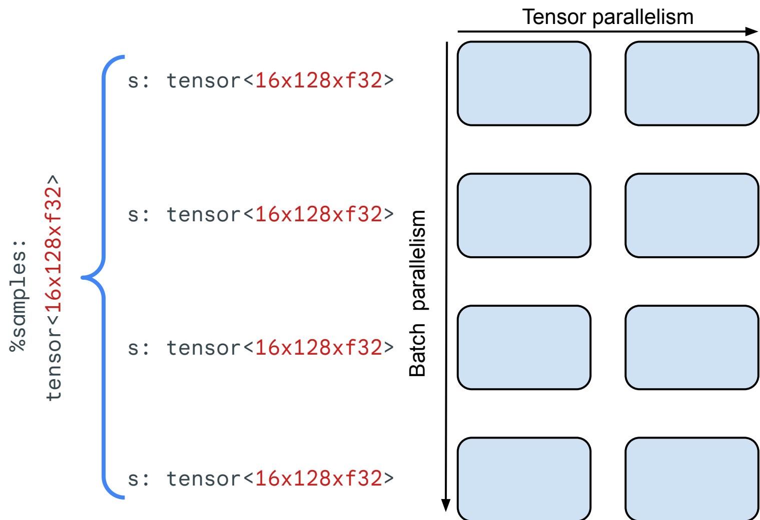
Google DeepMind



Background on AI model scaling

How models are scaled: sharding propagation and partitioning

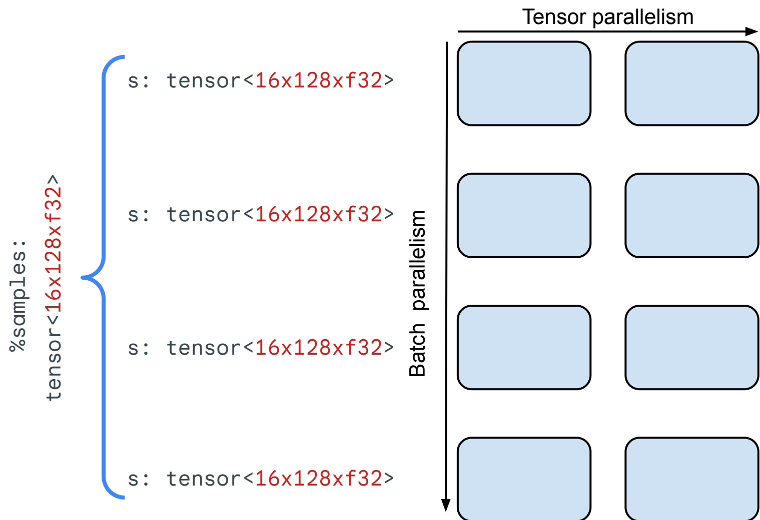
- Global program, nothing partitioned
- 8 total TPUs/GPUs available
  - Reshape into a logical mesh for
    - 4-way data-parallelism
    - 2-way tensor-parallelism




```
mesh @mesh = <"batch"=4, "model"=2>

func.func public @predict(
    %samples: tensor<16x128xf32>,
    %param1: tensor<128x256xf32>,
    %param2: tensor<256x10xf32>) -> tensor<16x10xf32> {
    %0 = stablehlo.dot_general %samples, %param1,
        contracting_dims = [1] x [0]
        : (tensor<16x128xf32>, tensor<128x256xf32>) -> tensor<16x256xf32>
    %1 = stablehlo.dot_general %0, %param2,
        contracting_dims = [1] x [0]
        : (tensor<16x256xf32>, tensor<256x10xf32>) -> tensor<16x10xf32>
    return %1 : tensor<16x10xf32>
  }
}
```

How models are scaled: sharding propagation and partitioning

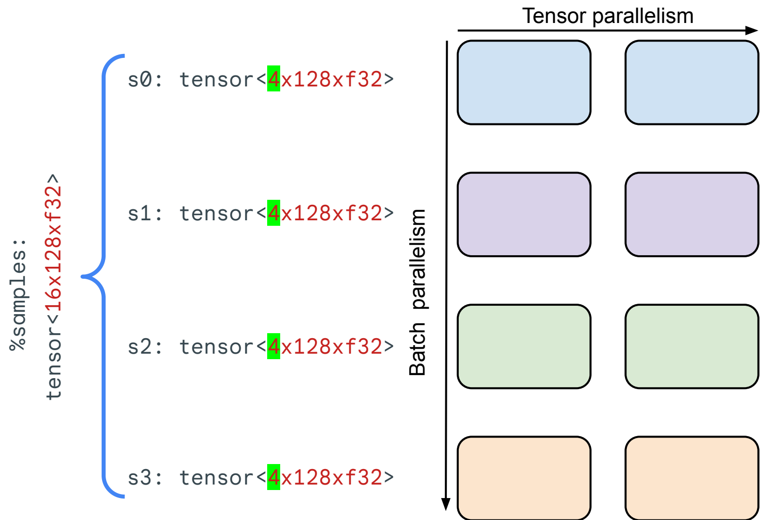
- Batch parallelism: calculate the predictions in parallel




```
mesh @mesh = <"batch"=4, "model"=2>

func.func public @predict(
    %samples: tensor<16x128xf32>,
    %param1: tensor<128x256xf32>,
    %param2: tensor<256x10xf32>) -> tensor<16x10xf32> {
    %0 = stablehlo.dot_general %samples, %param1,
        contracting_dims = [1] x [0]
        : (tensor<16x128xf32>, tensor<128x256xf32>) -> tensor<16x256xf32>
    %1 = stablehlo.dot_general %0, %param2,
        contracting_dims = [1] x [0]
        : (tensor<16x256xf32>, tensor<256x10xf32>) -> tensor<16x10xf32>
    return %1 : tensor<16x10xf32>
  }
}
```

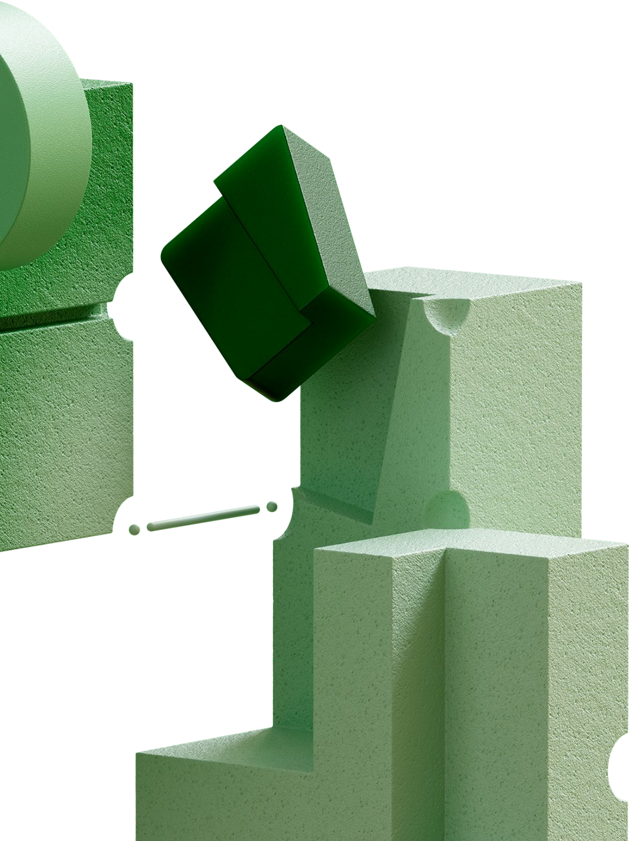
How models are scaled: sharding propagation and partitioning

- Batch parallelism: calculate the predictions in parallel




```
mesh @mesh = <"batch"=4, "model"=2>

func.func public @predict(
    %samples: tensor<4x128xf32>,
    %param1: tensor<128x256xf32>,
    %param2: tensor<256x10xf32>) -> tensor<4x10xf32> {
    %0 = stablehlo.dot_general %samples, %param1,
        contracting_dims = [1] x [0]
        : (tensor<4x128xf32>, tensor<128x256xf32>) -> tensor<4x256xf32>
    %1 = stablehlo.dot_general %0, %param2,
        contracting_dims = [1] x [0]
        : (tensor<4x256xf32>, tensor<256x10xf32>) -> tensor<4x10xf32>
    return %1 : tensor<4x10xf32>
  }
}
```

Google DeepMind



Other Ops/Attributes

Sharding Representation: Axis splitting and sub-axes

- A (full) mesh axis can be split into multiple **sub-axes** that can be individually used to shard a dimension or be explicitly replicated.
- To extract a specific sub-axis of size **k** from a full axis **"x"** of size **n**, we effectively reshape the size **n** (in the mesh) into **[m, k, n/(m*k)]** and use the 2nd dimension as the sub-axis.

```
@mesh_x= <"x"=8>

%arg0 : tensor<16xf32> {sdy.sharding=<@mesh_x, [{"x"}]>}

// axis "x" needs to be split into 2 sub-axes
%0 = reshape %arg0
    {sdy.sharding = <[<@mesh_x, [{"x":(1)4}, {"x":(4)2}]>]>}
    : (tensor<16xf32>) -> tensor<4x4xf32>

// axis "x":(1)4 needs to be further split into 2 sub-axes
%1 = reshape %0
    {sdy.sharding = <[<@mesh_x, [{"x":(1)2}, {"x":(2)2},
                               {"x":(4)2}]>]>}
    : (tensor<4x4xf32>) -> tensor<2x2x4xf32>
```

Ops: Sharding Constraints

- Can shard function inputs/outputs.
  - Via MLIR `FuncOp` arguments and result attributes.
- Can also shard intermediates.
  - Via `sdy.sharding_constraint` op.

```
// GSPMD
%43 = mhlo.custom_call @Sharding(%42) {mhlo.sharding =
"{devices=[8,1,4]<=[32] last_tile_dim_replicate}"} :
(tensor<8x8xf32>) -> tensor<8x8xf32>

// ~~SDY~~>
sdy.mesh @mesh = <"x"=8, "y"=4> %43 = sdy.sharding_constraint
%42 <@mesh, [{"x"}, {}]> : tensor<8x8xf32>
```

Ops: Shard-As / Shard-Like

- Ops sharing the same group id will adopt the same/similar sharding during propagation.

```
@mesh_xy = <"data"=2, "model"=2>



func.func @main(
    %arg0: tensor<8x2xi64>
        {sdy.sharding = #sdy.sharding<@mesh_xy, [{"data"},
                                                 {"model"}]>})
    -> (tensor<8x2xi64>) {
  %0 = sdy.sharding_group %arg0, group_id=0 : tensor<8x2xi64>
  %1 = stablehlo.constant dense<0> : tensor<8x2xi64>
  %2 = sdy.sharding_group %1, group_id=0 : tensor<8x2xi64>
  return %2 : tensor<8x2xi64>

}
```

Ops: Manual Computation

- Enclose a sub-computation that is manually partitioned using a subset of mesh axes.
- The shardings along those manual axes are specified for all inputs and outputs.
- SDY will be allowed to propagate through the body on any non-manual/free, **"data"** in this case.

```
sdy.mesh @mesh = <"data"=4, "model"=2>

%0 = sdy.manual_computation(%arg0, %arg1)

    in_shardings=[<@mesh, [{?}, {"model", ?}]>,
                  <@mesh, [{"model", ?}, {?}]>]

    out_shardings=[<@mesh, [{?}, {?}], replicated={"model"}>]

    manual_axes={"model"}

    (%arg2: tensor<2x8xf32>, %arg3: tensor<8x32xf32>) {

  %1 = stablehlo.dot_general %arg2, %arg3, contracting_dims =
[1] x [0]
      : tensor<2x32xf32>

  %2 = stablehlo.all_reduce(%1)
      {device_groups=...}

  sdy.return %2 : tensor<2x32xf32>

} : (tensor<2x16xf32>, tensor<16x32xf32>) -> tensor<2x32xf32>
```

Free to propagate through on "data" axis

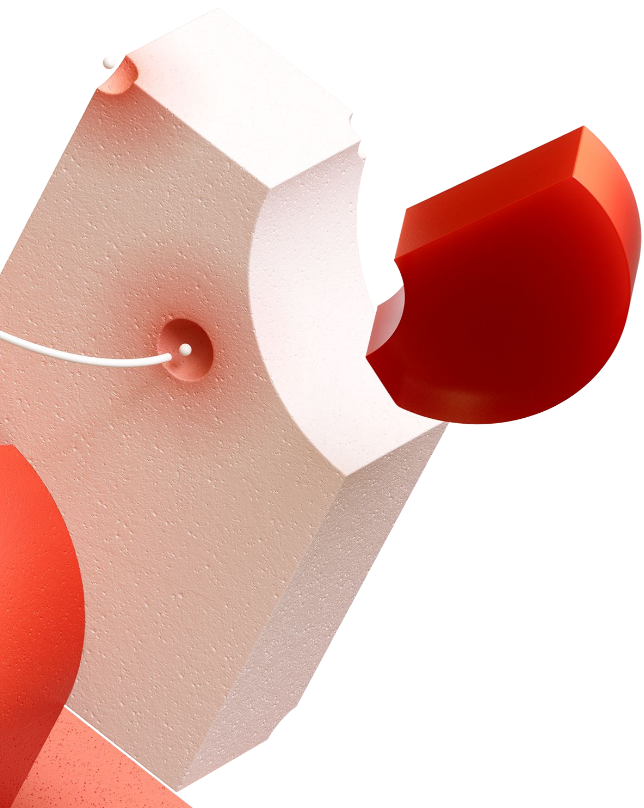
User written collective from original Python program

Ops: Manual Computation

- Enclose a sub-computation that is manually partitioned using a subset of mesh axes.
- The shardings along those manual axes are specified for all inputs and outputs.
- SDY will be allowed to propagate through the body on any non-manual/free, **"data"** in this case.

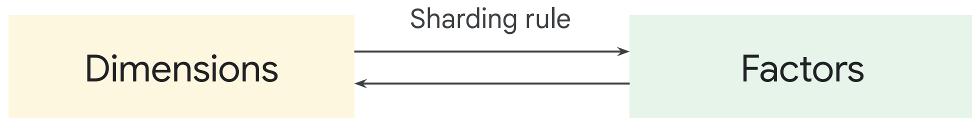
```
sdy.mesh @mesh = <"data"=4, "model"=2>

%0 = sdy.manual_computation(%arg0, %arg1)

    in_shardings=[<@mesh, [{"data", ?}, {"model", ?}]>,
                  <@mesh, [{"model", ?}, {?}]>]

    out_shardings=[<@mesh, [{"data", ?}, {?}],
                    replicated={"model"}>]

    manual_axes={"model"}

    (%arg2: tensor<2x8xf32>, %arg3: tensor<8x32xf32>) {

  %1 = stablehlo.dot_general %arg2, %arg3, contracting_dims =
[1] x [0]
      {sdy.sharding = <@mesh, [{"data", ?}, {?}]>} :
tensor<2x32xf32>

  %2 = stablehlo.all_reduce(%1)
      {device_groups=..., sdy.sharding = <@mesh, [{"data", ?},
{?}]>}

  sdy.return %2 : tensor<2x32xf32>

} : (tensor<2x16xf32>, tensor<16x32xf32>) -> tensor<2x32xf32>
```

Google DeepMind



Example: Propagation on Reshape

Propagate shardings in reshape

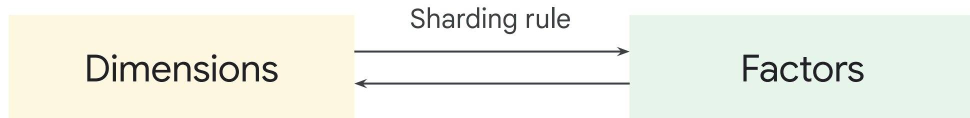


@mesh = <"x"=4, "y"=4>

%b = reshape %a : (tensor<16x4xf32>) -> tensor<8x8xf32>

- **Sharding rule:** `[ij, k] -> [i, jk], i=8, j=2, k=4`
- Sharding of %a: [{"x", "y", ?}, {?}]
- Sharding of %b: [{?}, {?}]

Propagate shardings in reshape



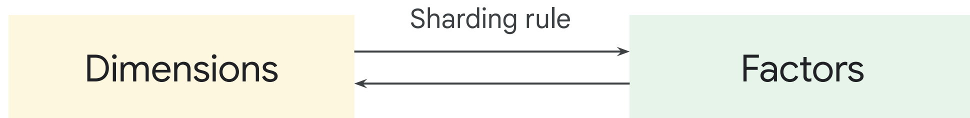
@mesh = <"x"=4, "y"=4>

%b = reshape %a : (tensor<16x4xf32>) -> tensor<8x8xf32>

- **Sharding rule:** `[ij, k] -> [i, jk], i=8, j=2, k=4`
- Sharding of %a: [{"x", "y", ?}, {?}]
- Sharding of %b: [{?}, {?}]

	Factor i, size 8	Factor j, size 2	Factor j, size 4
Tensor A	“x”, “y”:(1)2	“y”:(2)4	
Tensor B			

Propagate shardings in reshape



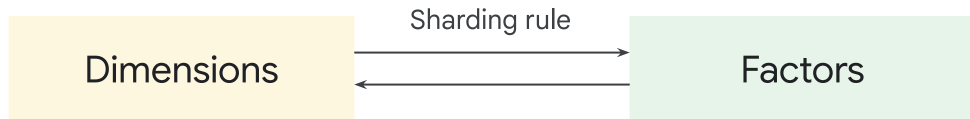
@mesh = <"x"=4, "y"=4>

%b = reshape %a : (tensor<16x4xf32>) -> tensor<8x8xf32>

- **Sharding rule:** `[ij, k] -> [i, jk], i=8, j=2, k=4`
- Sharding of %a: [{"x", "y", ?}, {?}]
- Sharding of %b: [{?}, {?}]

	Factor i, size 8	Factor j, size 2	Factor j, size 4
Tensor A	“x”, “y”:(1)2	“y”:(2)4	
Tensor B	“x”, “y”:(1)2	“y”:(2)4	

Propagate shardings in reshape



@mesh = <"x"=4, "y"=4>

%b = reshape %a : (tensor<16x4xf32>) -> tensor<8x8xf32>

- **Sharding rule:** `[ij, k] -> [i, jk], i=8, j=2, k=4`
- Sharding of %a: [{"x", "y", ?}, {?}]
- Sharding of %b: [{?}, {?}]

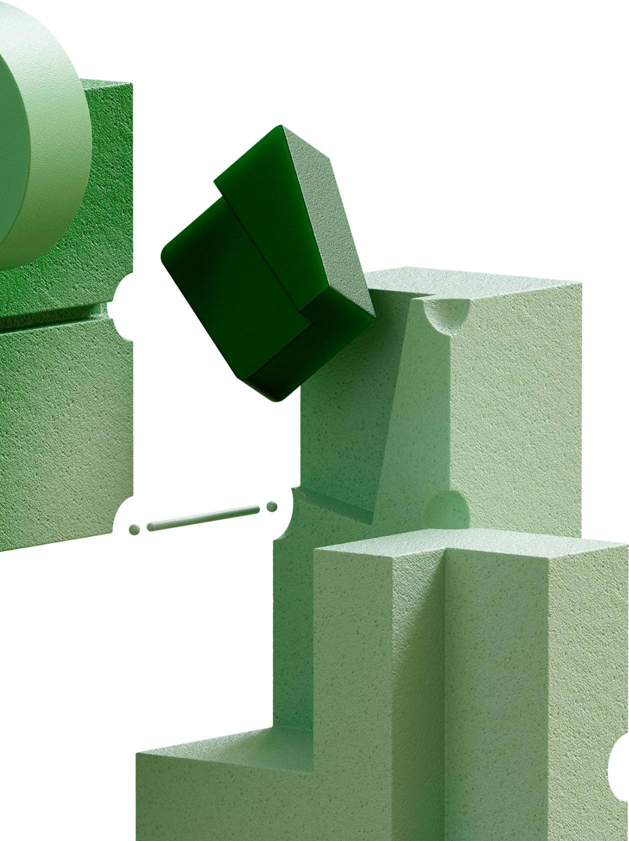
	Factor i, size 8	Factor j, size 2	Factor j, size 4
Tensor A	“x”, “y”:(1)2	“y”:(2)4	
Tensor B	“x”, “y”:(1)2	“y”:(2)4	

After propagation

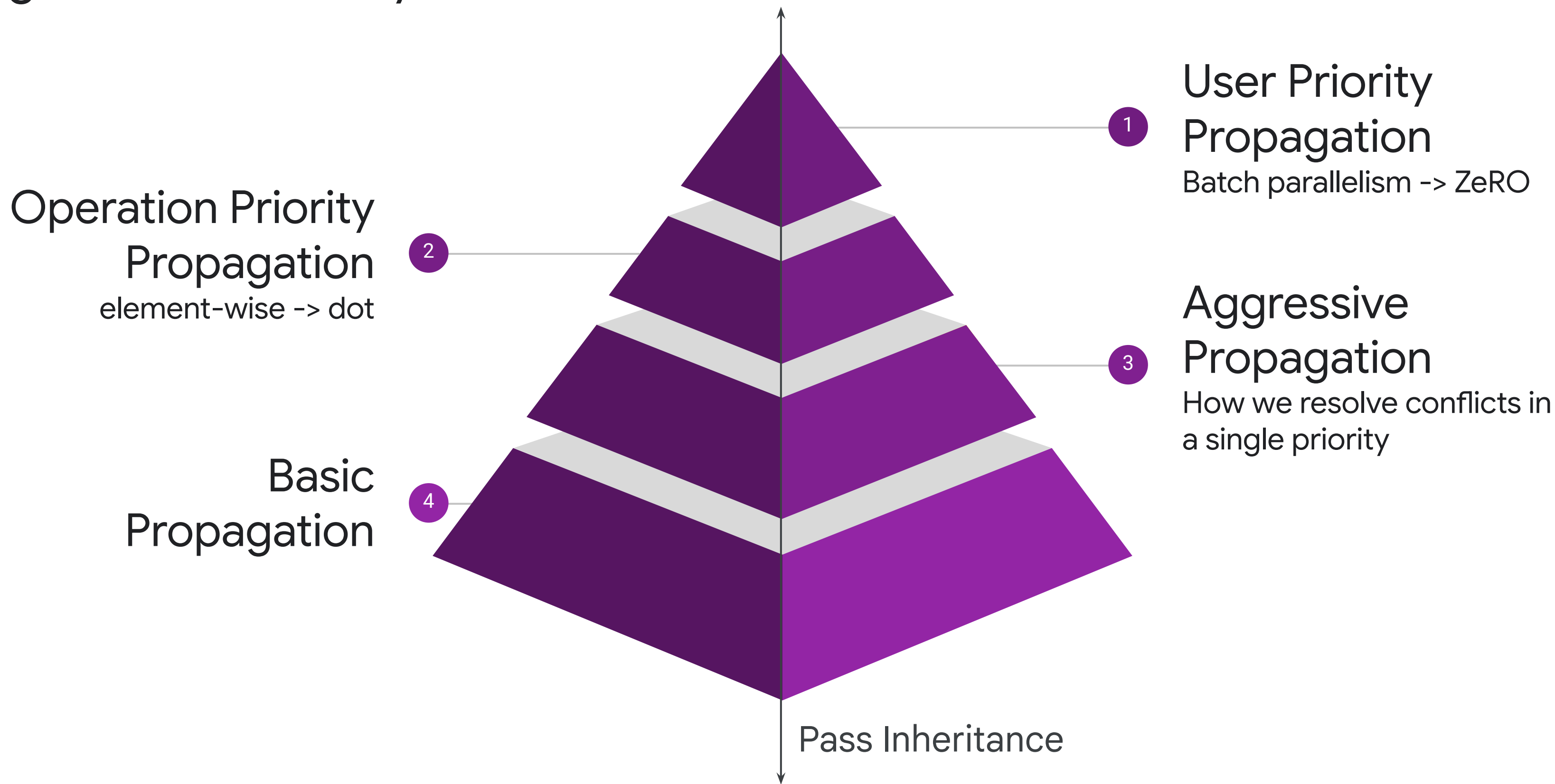
- Sharding of %a: [{"x", "y", ?}, {?}]
- Sharding of %b: **[{“x”, “y”:(1)2, ?}, {“y”:(2)4}]**

Google DeepMind

Conflict Resolution



What if there are conflicts?
Algorithm Hierarchy



Conflict resolution

Resolved by user priority

Resolved by operation priority

Resolved by our strategy in a single priority

- Try more and more aggressive strategy
  - Do not resolve any conflicts.
  - Propagate all potential solutions, which does not introduce conflicts.
  - Resolve conflicts across factors, e.g., batch dims -> contracting dims.
  - Resolve conflicts within a factor.

Step 0, beginning state

	F0	F1	F2	F3	Explicitly replicated
T0	“a”, “b”, “c”		closed		“d”
T1		“b”, “a”		“d”	
T2	closed		“c”, “a”		
T3		closed			
T4	“a”, “b”, “d”				

Step 1, get axes to propagate

	F0	F1	F2	F3	Explicitly replicated
T0	“a”, “b”, “c”		closed		“d”
T1		“b”, “a”		“d”	
T2	closed		“c”, “a”		
T3		closed			
T4	“a”, “b”, “d”				
Axes to propagate	“a”, “b”	“b”, “a”	“c”, “a”	“d”	“a”, “b”

Step 2, propagate the axes

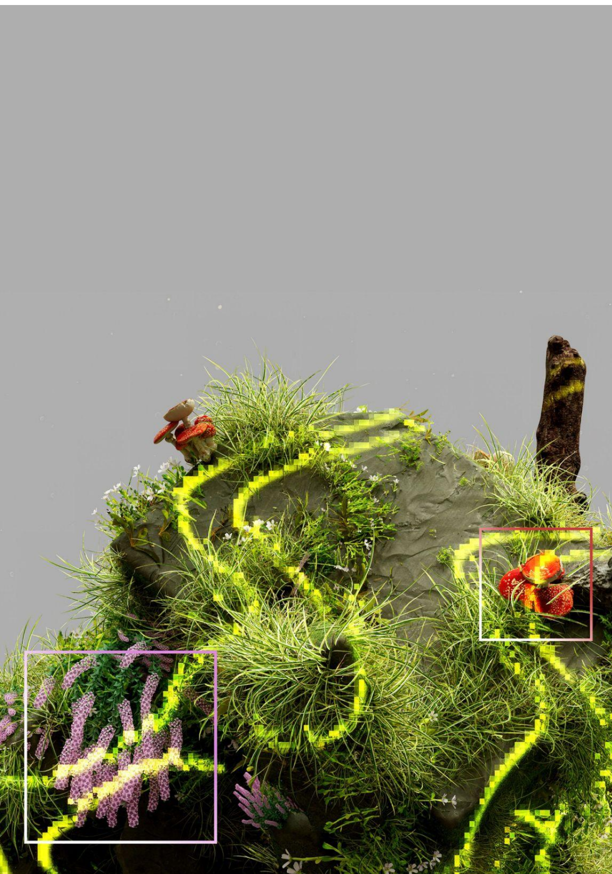
considering conflicts in the factor, ignoring conflicts between factors

	F0	F1	F2	F3	Explicitly replicated
T0	"a", "b", "c"	**"b", "a"**	closed		"d"
T1	**"a", "b"**	"b", "a"	**"c", "a"**	"d"	
T2	closed	**"b", "a"**	"c", "a"	**"d"**	
T3	**"a", "b"**	closed	**"c", "a"**	**"d"**	
T4	"a", "b", "d"	**"b", "a"**	**"c", "a"**	**"d"**	
Axes to propagate	"a", "b"	"b", "a"	"c", "a"	"d"	"a", "b"

Step 3, remove conflicts (overlapping axes) between factors

	F0	F1	F2	F3	Explicitly replicated
T0	“a”, “b”, “c”		closed		“d”
T1		“b”, “a”	**“c”**	“d”	
T2	closed	**“b”**	“c”, “a”	**“d”**	
T3		closed	**“c”**	**“d”**	
T4	“a”, “b”, “d”		**“c”**		
Axes to propagate	“a”, “b”	“b”, “a”	“c”, “a”	“d”	“a”, “b”

Google DeepMind



JAX Lowering

JAX -> Shardy

JAX lowers to Shardy representations and APIs

- jax.sharding.Mesh -> sdy.mesh
- jax.sharding.NamedSharding -> sdy.sharding
- jax.lax.with_sharding_constraint -> sdy.sharding_constraint
- jax.experimental.shard_map -> sdy.manual_computation

```
mesh = jax.sharding.Mesh(
    np.reshape(np.array(jax.devices()), (4, 2)),
    ('data', 'model'))

x = jax.ShapeDtypeStruct((32, 64), jnp.float32)
def f(x):
  return jax.lax.with_sharding_constraint(x,
NamedSharding(mesh, PartitionSpec('data',
PartitionSpec.UNCONSTRAINED)))

jax.config.update("jax_use_shardy_partitioner", True)
print(jax.jit(f).lower(x).as_text())

>
module @jit_f attributes {mhlo.num_partitions = 8 : i32,
mhlo.num_replicas = 1 : i32} {
  sdy.mesh @mesh = <["data"=4, "model"=2]>
  func.func public @main(%arg0: tensor<32x64xf32>
{mhlo.layout_mode = "default"}) -> (tensor<32x64xf32>
{jax.result_info = "", mhlo.layout_mode = "default"}) {
    %0 = sdy.sharding_constraint %arg0 <@mesh, [{"data"},
{?}]> : tensor<32x64xf32>
    return %0 : tensor<32x64xf32>
  }
}
```