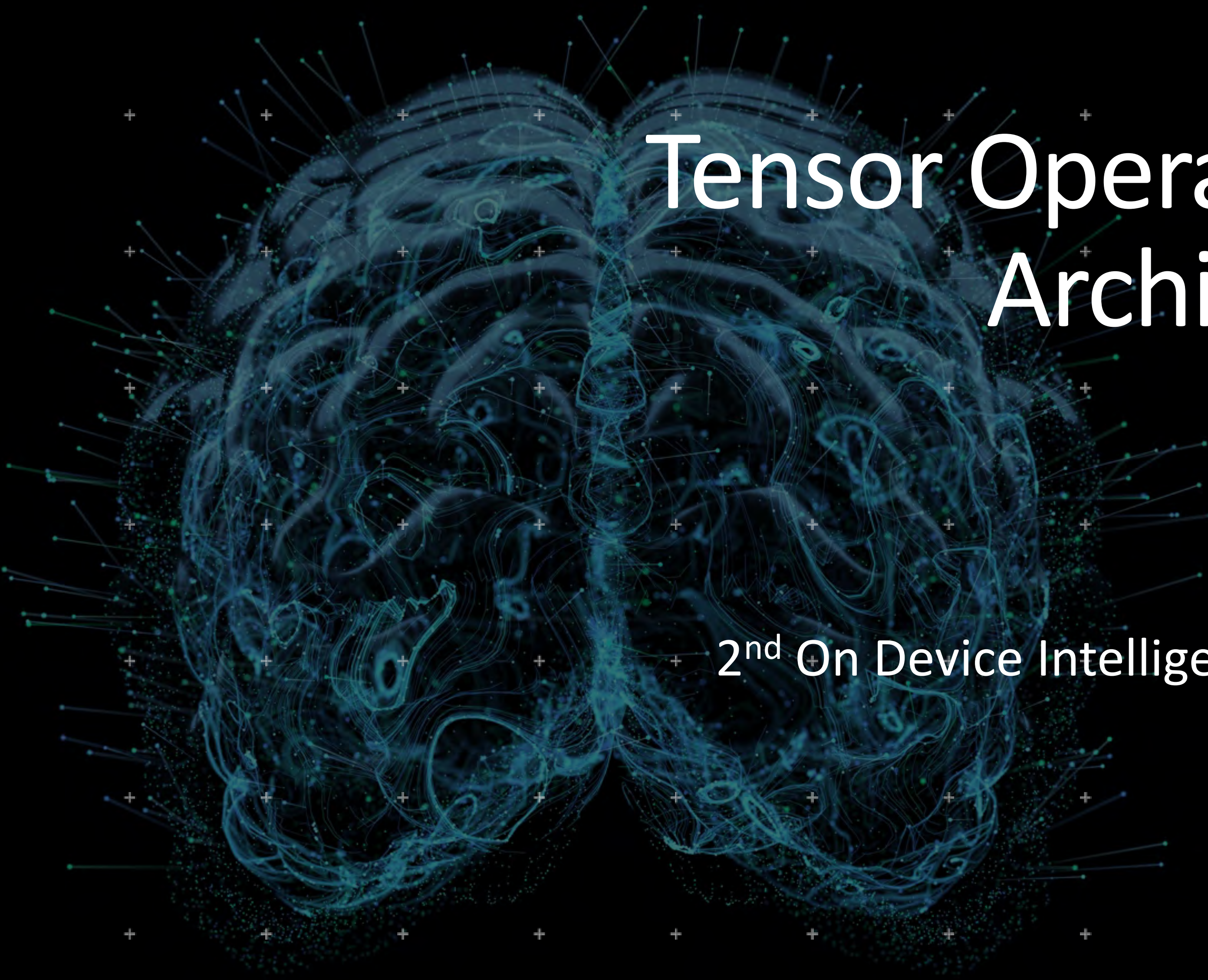


Tensor Operator Set Architecture (TOSA)

2nd On Device Intelligence Workshop

MLSys 2021

Eric Kunze

April 9th 2021

Background

Why TOSA?

Fast moving ecosystem

- ML Frameworks are moving incredibly quickly.
- Hardware and software inference platforms are fragmented.
  - Requires significant work to optimize networks for different inference platforms.
  - This work must be repeated for every new platform.
  - No standards regarding numerical behavior (e.g., quantization) and functionality.

Using the power of the entire system

- ML acceleration is appearing on more devices.
- Without a standard, developers need to choose:
  - Spend significant engineering effort optimizing for each device
  - Go with the lowest common denominator at the cost of performance.
- High end phones now come with NPUs supporting multiple TOPs.
  - Without common operator standards, difficult for a third-party application to use them.

Lowering the support cost

- As ML inference flows into more and more products, support will become an issue.
- In some deployments the devices have a long lifetime, like cars.
- Developers want to bring their latest networks onto all hardware.
- Test the network once for all compliant devices.
- Manage the support burden as more systems are deployed.

TOSA Specification

Tensor Operator Set Architecture (TOSA)

- A set of operators that work on tensors.
- Independent of any software or hardware design.
- Architected precision and numerical operation.
- Rigorous compliance testing.
- Designed for a wide variety of implementations.

Tensor operators only

- TOSA specifies operators only for whole tensors.
- Tensor operations allow for a variety of implementations and optimizations.
  - Operator fusing and tiling.
  - Memory traversal optimizations.
- Tensors are already the core of the frameworks.

Stability and consistency

- Standardized, stable layer between the frameworks and the inference platform.
  - Enable fast evolution of the frameworks, while stabilizing the platform below the layer.
  - Finite set of composable primitives enabling infinite set of operators .
- ML model built using TOSA guaranteed to run on any platform supporting TOSA.

arm

Standardization

- TOSA is an open standard.
- The TOSA standard license grants a license to IP required to implement the specification.
- Contributions to the specification are required to grant similar rights.
- We encourage a wide array of implementations and welcome contributions.

TOSA principles

- Operators should be primitives that cannot be broken down into simpler whole-tensor operations.
- Operators should be building blocks for more complex operations.
- Numerical definition should be consistent between operators.
- Valid input and output ranges for all operands shall be specified.
- Integer operators shall be implementable in bit-exact form with good efficiency.

How to choose the operators?

- Reviewed frameworks comparing the supported operators.
- Iterated over a proposed set of TOSA operators.
- Looked for common building blocks to build framework operators from.
- Created test sequences of TOSA operators matching the original operator.

TOSA Operators

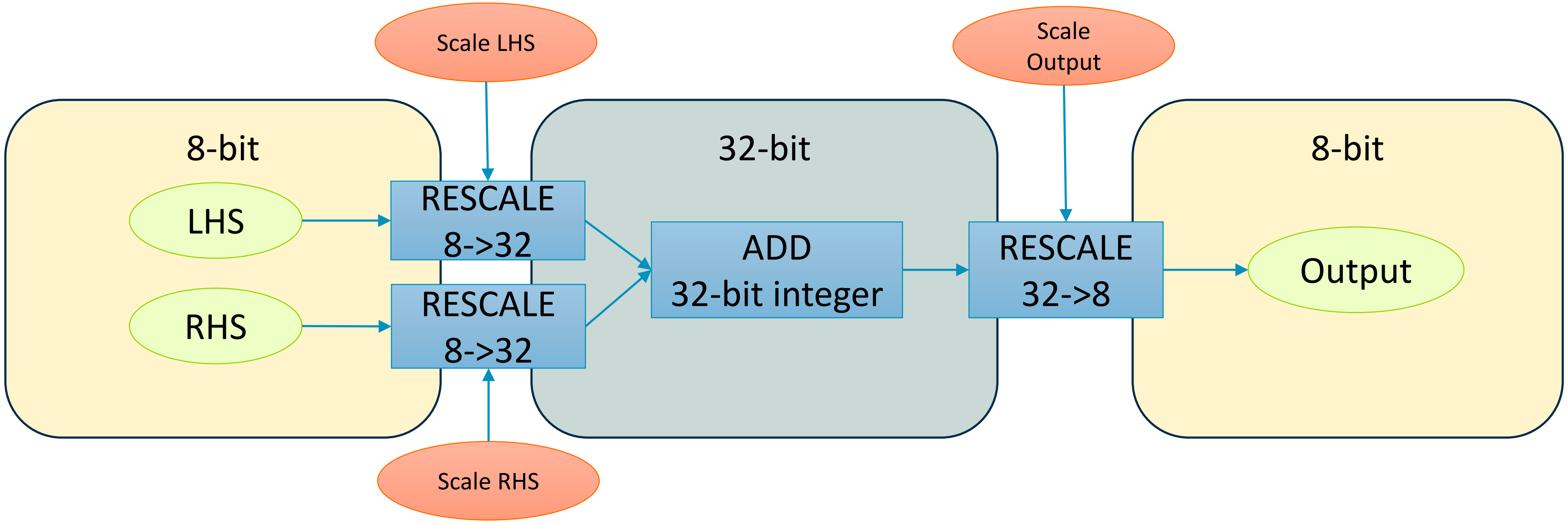
- As of the current version 0.22, TOSA consists of ~70 operators.
- Operator categories
  - Tensor operations (convolve/pool)
  - Elementwise operations (unary/binary/ternary)
  - Activation
  - Comparison
  - Reduction
  - Data transform
  - Scatter/Gather
  - Image
  - Control Flow (if/while)

Quantized integer operation semantics

- Embedded inference platforms often lack floating point hardware.
- The operation of quantized integer operators is not well defined in the frameworks (where it exists at all)
- TOSA makes the semantics explicit by separating scaling out into RESCALE operations.
- RESCALE rescales between different ranges and bit widths using an integer multiply, shift, and round.
- This allows a variety of scale choices, while ensuring the same result for a given sequence of TOSA operations.

Quantized integer example

- Example – Elementwise add of two quantized 8-bit integer tensors.
  - Each tensor may have a different scale, so simple addition doesn't work.
  - We must scale both inputs into a common range.
  - There are multiple valid options for scale LHS/RHS/Output, but for any given choice, the computation must be consistent.



Profiles

- Profiles enable consistent deployment across a class of devices
- 3 profiles defined to cover microcontrollers up through large cores

Profile	Name	Integer Inference	Floating-point Inference	Training	Common use
Base Inference	TOSA-BI	Yes	No	No	Microcontroller deployment
Main Inference	TOSA-MI	Yes	Yes	No	Inference deployment
Main Training	TOSA-MT	Yes	Yes	Yes	Training

Operator specification

- Arguments
  - Inputs - Inputs not known at compile time. Always tensors/lists of tensors.
  - Attributes - Inputs that are known at compile time.
  - Outputs - Operator output values. Always tensors/lists of tensors.
- Supported Data types
  - float/int8/int16/bool.
  - Smaller data types allowed if they give the same numeric result as the same number stored in an 8-bit container.
- Detailed operation code
- Profiles supported
- Quantization parameters (scale, zero point)

Example operator specification

2.9.2. REDUCE_ANY

Reduce a tensor along the given axis with a logical OR operation

Arguments:

Argument	Type	Name	Shape	Description
Input	in_t*	input	shape1	Input tensor with rank from 1 to 4
Attribute	int32_t	axis	-	Axis to reduce, in range from 0 to rank(shape1)-1
Output	in_t*	output	shape	Output tensor. Same rank as the input tensor.

Operation Function:

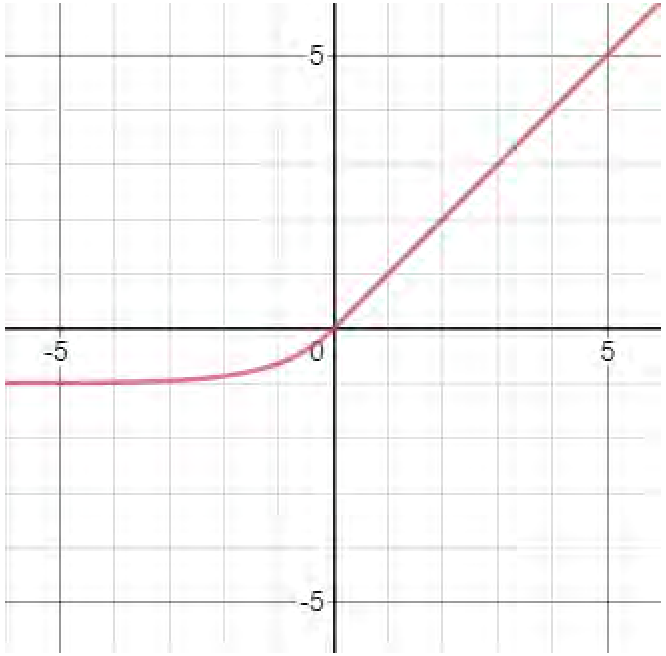
```
assert(0 <= axis && axis < rank(shape1));
assert(shape[axis] == 1);
for_each(index in shape) {
    tensor_write<in_t>(output, shape, index, false);
}
for_each(index in shape1) {
    tmp_index = index;
    tmp_index[axis]=0;
    value = tensor_read<in_t>(input, shape1, index);
    acc   = tensor_read<in_t>(output, shape, tmp_index);
    acc   = acc || value;
    tensor_write<in_t>(output, shape, tmp_index, acc);
}
```

Supported Data Types:

Profile	Mode	in_t
Any	Boolean	bool_t

Composing a new operator with TOSA

- What happens when a new operator comes along?
- Example: ELU activation, not part of TOSA.
  - elu(x) = x if x >= 0, exp(x)-1 otherwise
- TOSA sequence implementing ELU:
  - A = EXP(x)
  - B = SUB(A, 1)
  - C = GREATER_EQUAL(X, 0) // Is X >= 0
  - Output = SELECT(C, X, B) // return X or B based on >= results
- We have sequences of >15 TOSA operators to match one framework operator (quantized SoftMax)

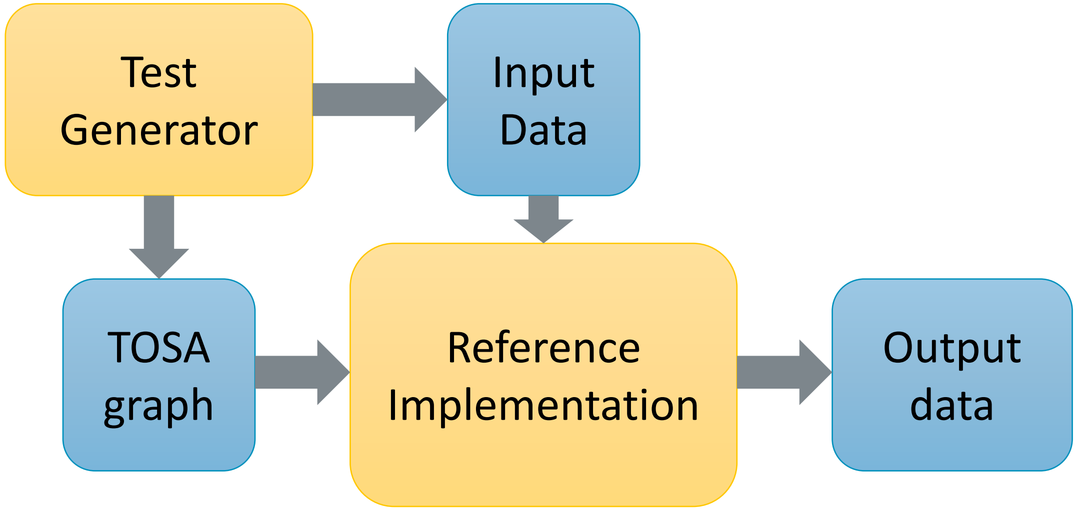


Beyond the specification

Applying TOSA

Reference implementation and test suite

- Reference implementation published along with the specification, which consumes a TOSA graph and input data, and produces output data.
- Reference implementation computations follow the precision in the specification.
- TOSA testcase generator, which creates TOSA graphs and input data.

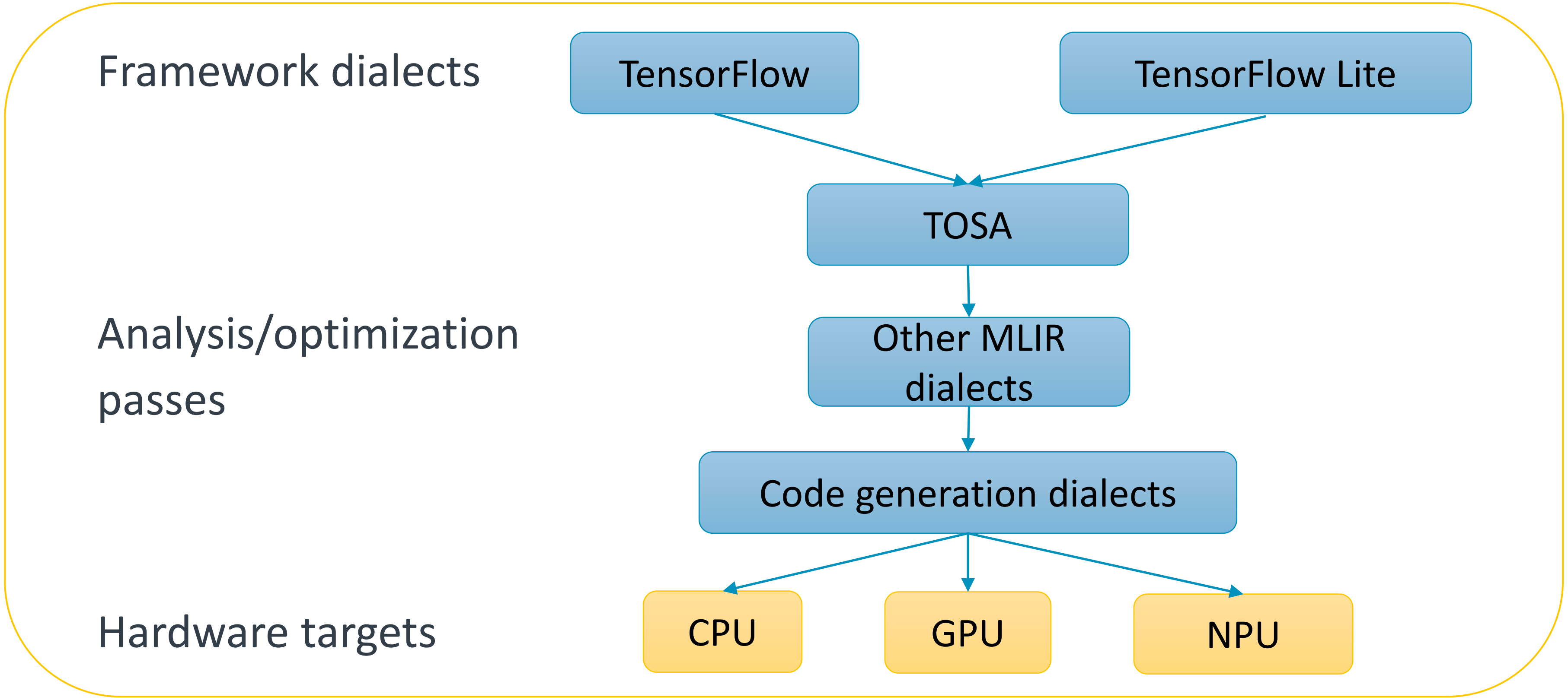


MLIR - Multi Level Intermediate Representation

- MLIR is a compiler toolkit being worked on as part of the LLVM project.
- Provides an infrastructure for representing multiple IRs within a single graph.
- Makes it easy to add new dialects, which represent an abstraction level.
- Passes can provide analysis and optimization of dialects.
- Legalization passes convert from one dialect into another.
- For details on MLIR, see Jacques Pienaar's (Google) talk from the Chips and Compilers Symposium.

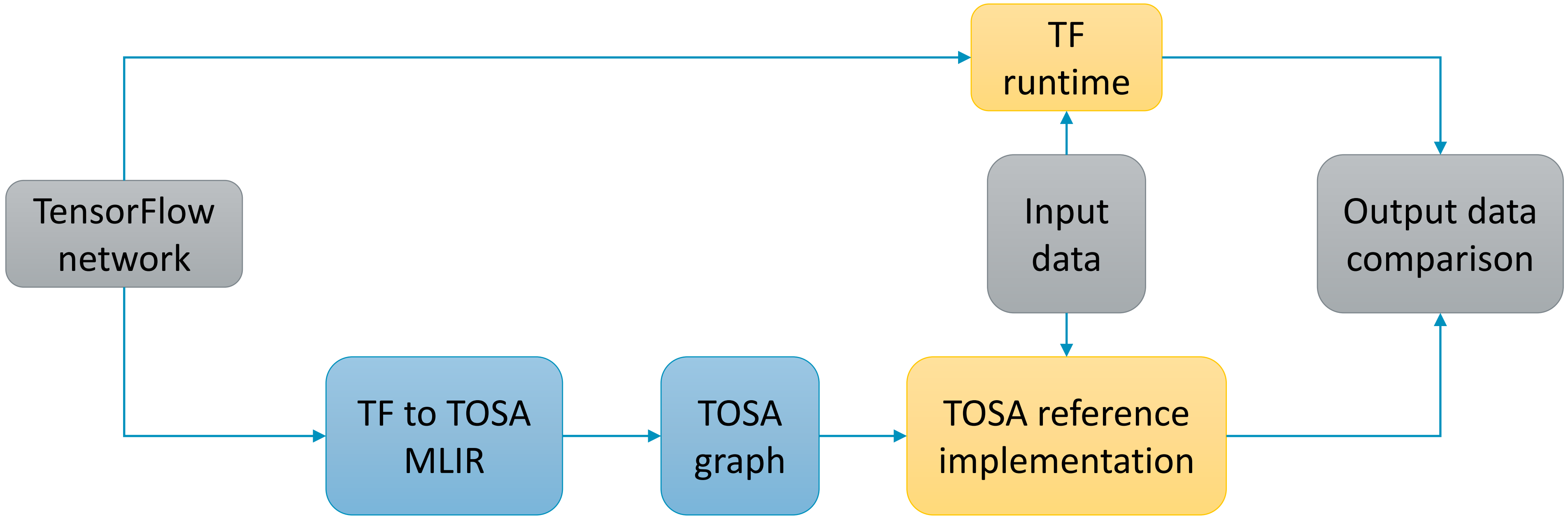
TOSA in MLIR

- We have published a TOSA dialect within the MLIR compiler project.
- TensorFlow and TensorFlow Lite teams have released MLIR dialects.
- MLIR legalization passes take TensorFlow and TensorFlow lite networks and create TOSA graphs from them.



TOSA, TensorFlow, and TensorFlow Lite

- Using the reference implementation and the compiler stack, we can verify that the translation from the framework into TOSA has the same result as the original network.



TOSA in hardware

- Hardware implementation has a stable set of operators to implement.
- Simplify verification by comparing against the reference implementation.
- Public test suite also eases verification effort.
- TOSA abstraction level enables innovative hardware designs.
- Existing TOSA networks port to new hardware designs.

Moving forward

Where does TOSA go from here?

TOSA open-source reference

- TOSA specification published on mlplatform.org
  - https://developer.mlplatform.org/w/tosa/
  - Open for contributions with CLA to enable implementations to avoid IP problems.
- TOSA reference implementation published on mlplatform.org
  - https://git.mlplatform.org/tosa/reference_model.git
  - Includes TOSA test generator.
- TOSA MLIR dialect published in LLVM GitHub repository.
  - https://github.com/llvm/llvm-project/tree/main/mlir/lib/Dialect/Tosa
- TensorFlow and TensorFlow Lite legalizations published in TensorFlow GitHub repository.
  - https://github.com/tensorflow/tensorflow/tree/master/tensorflow/compiler/mlir/tosa

Contribute to TOSA

- Achieving a wide array of implementations benefits application and implementation developers.
- MLPlatform hosts a discourse forum for TOSA discussions.
- Contributions are welcome at all levels
  - Specification
  - Reference implementation
  - MLIR dialect
  - Transformations between frameworks.

Thank you

- The MLIR community has been very helpful as we have worked on the dialect, giving us feedback and assistance to land a very large change.
- Thanks to the TensorFlow and IREE teams at Google for a great deal of advice, code reviews and overall help in bringing the TensorFlow and TensorFlow Lite to TOSA legalizations into the TensorFlow repository.