06/09/2025

# Working with Large Models in ONNX IR

Justin Chu, Microsoft

# Agenda

1. **Challenges** when working with large models in ONNX

2. A **refresher** of onnx-ir concepts

3. How can **onnx-ir** help
   a. Process large weights
   b. Construct more complex model architecture
   c. Optimize model and rewrite patterns

4. **Demos** and examples!

# Challenges: working with LLMs in ONNX

- Larger **weights**
- More complex **architecture** to build
- More complex **patterns** to match

# Part of the ONNX Org

- onnx/ir-py
- Get it via `pip install onnx-ir` right now

# ONNX defines the IR, `onnx/ir-py` is its representation in Python

-> **Native Python** data structures and APIs

-> An alternative to protobuf & APIs

# Concepts

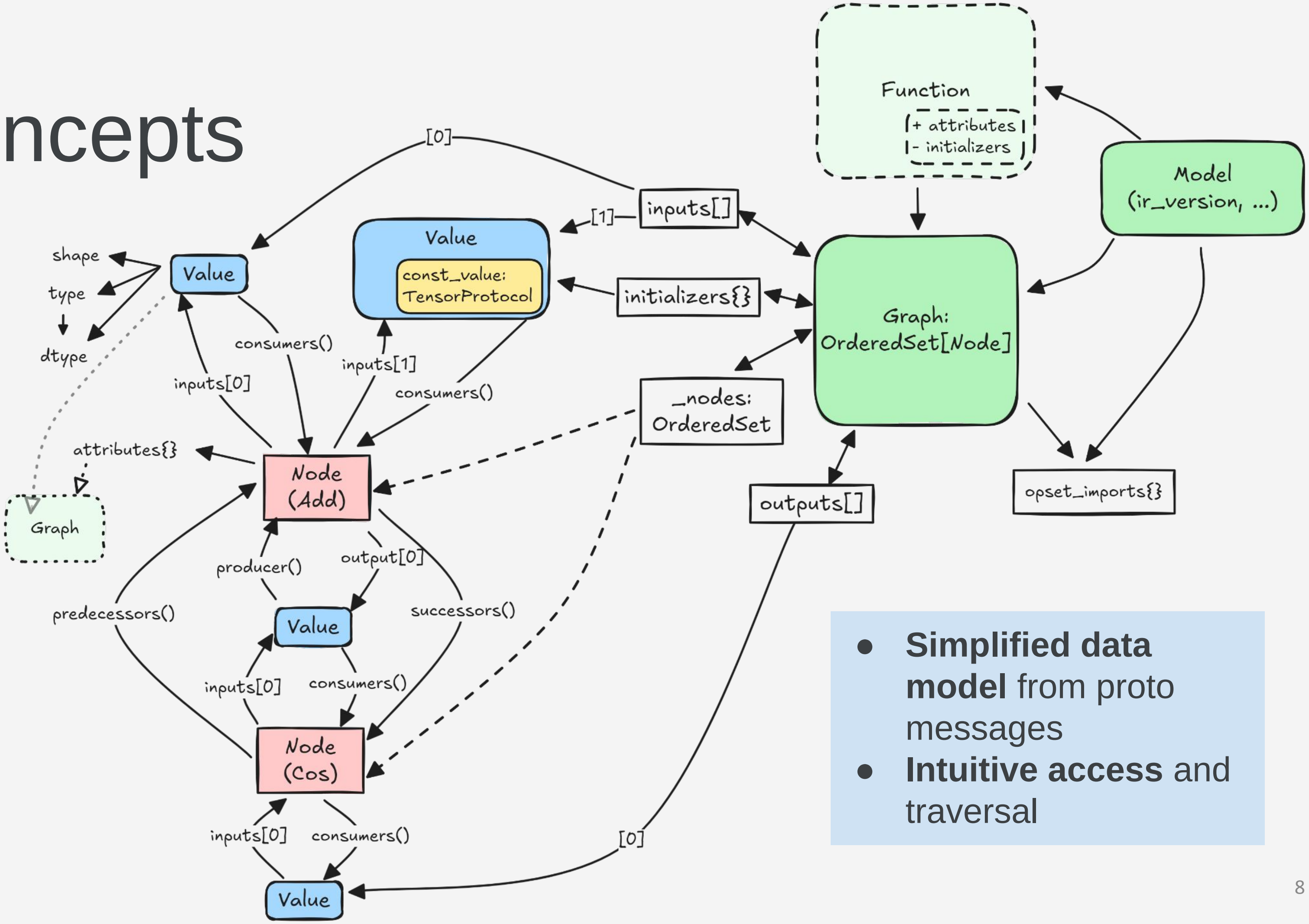
- ModelProto
  - GraphProto
  - NodeProto list
  - **Values as strings**
    - A separate ValueInfoProto list
  - **TensorProto**
  - FunctionProto

- ir.Model
  - ir.Graph (designed to be mutated)
  - ir.Node
    - Allows mixed opset versions
    - predecessor(), successors()
  - **ir.Value**
    - producers and usages
  - **ir.TensorProtocol**: zero copy, unified interface
  - ir.Function (Graph with formal attributes)

# The TensorProtocol

```python
class TensorProtocol(ArrayCompatible, DLPackCompatible, Protocol):
    name: str | None
    shape: ShapeProtocol
    dtype: _enums.DataType
    doc_string: str | None
    raw: Any
    metadata_props: MutableMapping[str, str]
    meta: MutableMapping[str, Any]

    @property
    def size(self) → int: ...

    @property
    def nbytes(self) → int: ...

    def numpy(self) → np.ndarray:
        """Return the tensor as a numpy array."""

    def __array__(self, dtype: Any = None) → np.ndarray:
        """Return the tensor as a numpy array, compatible with np.array."""
    def __dlpack__(self, *, stream: Any = ...) → Any:
        """Return PyCapsule."""
    def __dlpack_device__(self) → Any:
        """Return the device."""

    def tobytes(self) → bytes:
        """Return the tensor as a byte string conformed to the ONNX specification,
in little endian."""
```

# Concepts




- **Simplified data model** from proto messages
- **Intuitive access** and traversal

# LLMs in ONNX

- **Larger *weights***
- More complex architecture to build
- More complex patterns to match

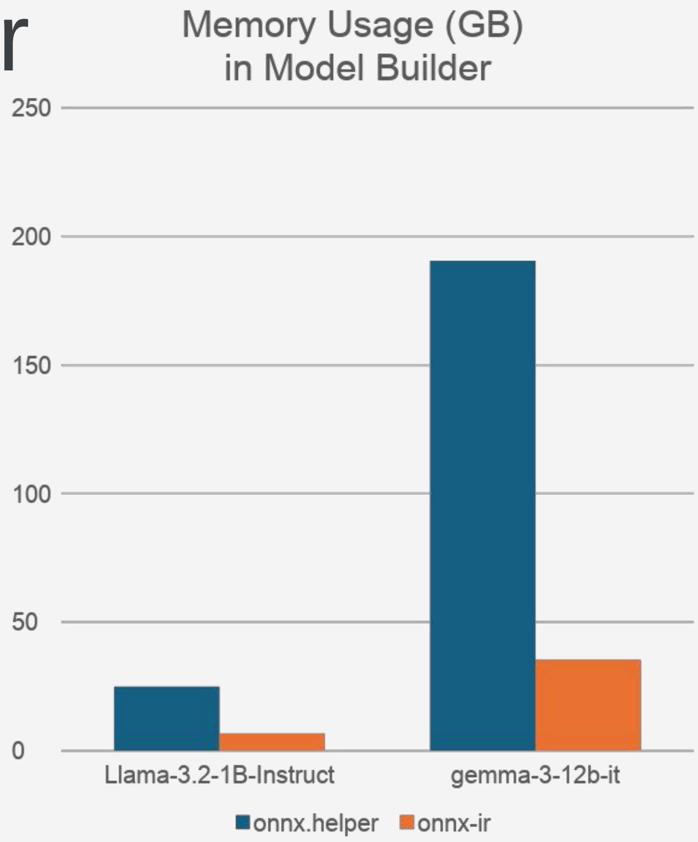
# Weights are large

- ONNX Protobuf:
  - 2GB serialization **size limit** - breaks when crossing the c++ api boundary
  - Have to **copy data** to create TensorProto, or load from disk
  - **Inconsistent APIs** for memory and external tensors

- onnx_ir:
  - **No 2GB limit** from protobuf
  - **Share tensors** from frameworks
  - **Memory mapped** external data files
  - All tensors (external, in-memory, packed int4 tensor) on a **common interface** `ir.TensorProto`

# Example: Model Builder

- `ir.Graph` + `LazyTensor`
- **190.5GB -> 35.3GB memory usage** (**18.5%** of the original) when building the gemma-3-12b-it model
  - 1.x memory used for loading the model with transformers
- **14.5 min -> 2min (13.8%)** model build time



# Demo: Safetensors in ONNX

# LLMs in ONNX

- Larger weights
- **More complex *architecture* to build**
- More complex patterns to match

# Example: PyTorch exporter

- Builds ONNX graph with **rich metadata**
- **Zero copy** of the PyTorch tensors
- Runs **passes on the ONNX model** (much easier than FX for ONNX specific changes like symbolic shapes)

```python
    for name, torch_tensor in itertools.chain(
        exported_program.named_parameters(),
        exported_program.named_buffers(),
        exported_program.constants.items(),
    ):
        initializer = model.graph.initializers.get(name)  # type: ignore[assignmen
        if initializer is None:
            logger.warning("Tensor '%s' is not one of the initializers", name)
            continue
        if not isinstance(torch_tensor, torch.Tensor):
            raise NotImplementedError(
                f"Tensor '{name}' should be a torch.Tensor. Actual type is '{type(
                "This is unexpected and not yet supported."
            )
        ir_tensor = TorchTensor(torch_tensor, name=name)
        initializer.const_value = ir_tensor
        _set_shape_type(
            initializer,
            torch_tensor,
            complex_to_float=lower != "none",
        )
```

# LLMs in ONNX

- Larger weights
- More complex architecture to build
- **More complex *patterns* to match**

# Example: Fusion in onnxscript for ONNX Runtime

# Bonus: `onnx_ir.passes`

- Pass infrastructure for transforming ONNX graphs in Python
- Common passes like DCE, CSE, constant folding, **stable topological sort**, and more

https://onnx.ai/ir-py/api/ir_passes_common.html

1. **Building** a model with the tape module.
2. Load a model and **replace some initializers**
3. Build a pass to **modify the model** (merge QKV weights)
4. Use **rewriter** to do the same thing
5. Show it on model explorer

# Demo: Combining all the above

# Get started

github.com/onnx/ir-py (PRs welcome)

`pip install onnx-ir`

```
```py
import onnx_ir as ir
model = ir.load("model.onnx")
```
```

**Can't wait to see what you build with it! ✨**

# Next steps & Discussion

1. Multi-device representation (IRv11) support
2. Symbolic shape inference with SymPy
3. Graph composition utilities