Polygraphy and ONNX-GraphSurgeon

Pranav Marathe

Background

- I work on the TensorRT team @ NVIDIA[1]
- ONNX is our primary import path, so we've developed lots of tooling for it
- This talk will cover two open-source tools:
  - ONNX-GraphSurgeon: Create and modify ONNX models
  - Polygraphy: Inspect, modify, and debug ONNX models

[1] Just in case the title slide didn't give that away

What is ONNX-GraphSurgeon*?

- Python-based IR for bipartite DAGs consisting of nodes and tensors
- Virtually any modifications are possible using a simple Python API
- Provides some additional conveniences: constant folding, topological sorting, dead layer removal

Source code and examples available here

[*] Not medically licensed

The IR

An Example

- In addition to the fields above, inputs/outputs are also tracked:
  - For tensors, `inputs/outputs` are lists of `Nodes` that consume/produce them
  - For nodes, `inputs/outputs` are lists of `Tensors`
  - Makes graph traversal easy
  - Editing inputs/outputs allows you to restructure the graph
- Everything shown here can be freely edited or constructed manually

Creating A Model The Easy Way

Registering Ops

- Use `Graph.register()` to add methods to Graph
- Methods can be arbitrarily complex and can access the graph via `self`
- Totally reusable

```
@gs.Graph.register()
def leaky_relu(self, inp, alpha=0.01):
    out = self.layer(
        op="LeakyRelu",
        inputs=[inp],
        outputs=["leaky_relu_out"],
        attrs={"alpha": alpha},
    )[0]
    out.dtype = inp.dtype
    return out
```

Creating A Model The Easy Way

Using Registered Ops

- Registered ops can be used directly from graph instances:

```
# Build a graph that computes `out = leaky_relu(input)`
graph = gs.Graph(inputs=[gs.Variable(name="input", dtype=np.float32, shape=(1, 3, 28, 28))])

out = graph.leaky_relu(graph.inputs[0])

graph.outputs = [out]

onnx_model = gs.export_onnx(graph)
```

What is Polygraphy?

A bird? A plane?

- Python API and Command-line Toolkit for debugging DL models
- Does lots of different things, but we'll focus on ONNX tooling

Source code and examples available here

Before We Begin

polygraphy run

- **run** lets you run inference with backends, like ONNX-Runtime, and compare results

```
$ polygraphy run model.onnx --onnxrt
[I] onnxrt-runner  | Activating and starting inference
[I] Creating ONNX-Runtime Inference Session with providers: ['CPUExecutionProvider']
[I] onnxrt-runner
   ---- Inference Input(s) ----
   {input [dtype=float32, shape=(1, 3, 28, 28)]}
[I] onnxrt-runner
   ---- Inference Output(s) ----
   {leaky_relu_out_0 [dtype=float32, shape=(1, 3, 28, 28)]}
[I] onnxrt-runner  | Completed 1 iteration(s) in 0.07 ms | Average inference time: 0.07 ms.
[I] PASSED | Command: polygraphy run model.onnx --onnxrt
```

Inspecting Models

Who needs GUIs?[1]

- **`inspect model`** shows us a text representation of the model
- Display can be configured to show: Initializers, Nodes, and/or Attributes

```
$ polygraphy inspect model model.onnx --show layers attrs weights
[I] ==== ONNX Model ====
    Name: onnx_graphsurgeon_graph | ONNX Opset: 11

    ---- 1 Graph Input(s) ----
    {input [dtype=float32, shape=(1, 3, 28, 28)]}

    ---- 1 Graph Output(s) ----
    {leaky_relu_out_0 [dtype=float32, shape=()]}

    ---- 0 Initializer(s) ----
    {}

    ---- 1 Node(s) ----
    Node 0    | onnx_graphsurgeon_node_1 [Op: LeakyRelu]
        {input [dtype=float32, shape=(1, 3, 28, 28)]}
                -> {leaky_relu_out_0 [dtype=float32, shape=()]}
               ---- Attributes ----
               onnx_graphsurgeon_node_1.alpha = 0.009999999776482582
```

[1] If you need a GUI, I highly recommend Netron

Simplifying Models

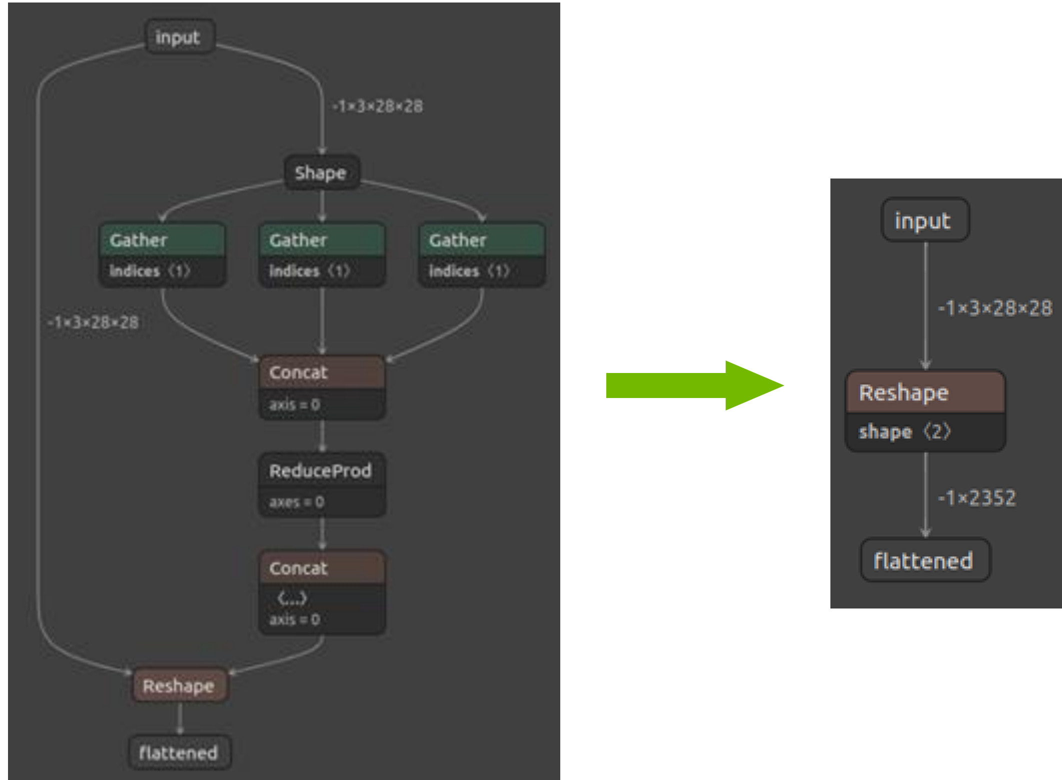
- **`surgeon sanitize`**[1] allows you to fold constants in the model
- Similar to ONNX-Simplifier, but a few key differences:
    - Preserves dynamic shapes while simplifying shape computations
    - Highly fault-tolerant due to partitioning
    - Special optimizations like `If` lowering and `Cast` elision

```
$ polygraphy surgeon sanitize model.onnx -o folded.onnx --fold-constants
[I] Folding Constants | Pass 1
[I]     Total Nodes | Original:     8, After Folding:     1 |     7 Nodes Folded
[I] Folding Constants | Pass 2
[I]     Total Nodes | Original:     1, After Folding:     1 |     0 Nodes Folded
[I] Saving ONNX model to: folded.onnx
```

[1] Side effects include weight loss

Simplifying Models

Eliminates 99.9% of unnecessary nodes and tensors[1]



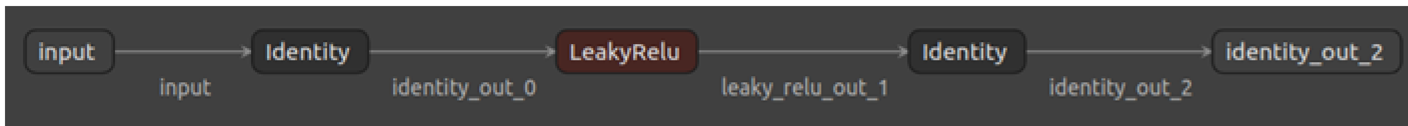
[1] These statements have not been evaluated by the FDA

Extracting Subgraphs

- **surgeon extract** allows you to extract subgraphs from a model
- Use **inspect model** or Netron to figure out input/output tensors
- For inputs, need to provide shapes and data types
- For outputs, need to provide data types
- Format is: **<tensor_name>:<shape>:[<dtype>]**
    - For example: **input0:[1,3,224,224]:float32**
- **auto** indicates shapes/data types should be automatically determined

Extracting Subgraphs

An Example



- Assume we're extracting 'LeakyReLU' - we can see the input/output tensor names in Netron
- We'll use those names and use **`auto`** for shapes and data types:

```
$ polygraphy surgeon extract model.onnx -o subgraph.onnx \
    --inputs identity_out_0:auto:auto \
    --outputs leaky_relu_out_1:auto
```

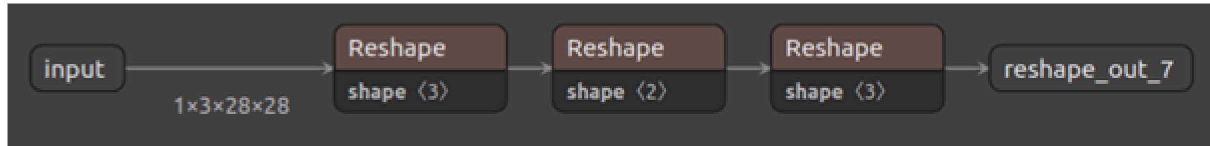
Model Bisection

- Like **`git bisect`**, but for ONNX models!
- Assuming we start with a (failing) **`model.onnx`**, the algorithm is:
    1. Remove **`N`** nodes from the model and generate a new model
    2. If new model fails, **`goto 1`**
    3. If new model passes, add back **`M`** nodes, generate a new model, and **`goto 2`**
    4. Repeat until smallest failing model is found
- 'fail'/'pass' intentionally vague - bisection works for **any** type of failure

Model Bisection: An Example

Setting The Stage

- Imagine we have the following ONNX model which gives us an error when we run it:




```
$ polygraphy run model.onnx --onnxrt
[E:onnxruntime:, sequential_executor.cc:339 Execute] Non-zero status code returned while
running Reshape node. Name:'onnx_graphsurgeon_node_5' Status Message:
/onnxruntime_src/onnxruntime/core/providers/cpu/tensor/reshape_helper.h:41
onnxruntime::ReshapeHelper::ReshapeHelper(const onnxruntime::TensorShape&, std::vector<long
int>&, bool) gsl::narrow_cast<int64_t>(input_shape.Size()) == size was false. The input
tensor cannot be reshaped to the requested shape. Input shape:{1,3,784}, requested
shape:{1,2351}
```

- Reducing the model to something smaller can make this easy to debug[1]

[1] *Reading the error message* would also make this easy to debug, but that doesn't make for a good example

Model Bisection: An Example

Interactive Mode

- In interactive mode, **`debug reduce`** will generate models successively and ask us whether each one passes or fails.
- We'll run each of these models using **`run`** and report what we see
- Our **`debug reduce`** command is quite simple:

```
$ polygraphy debug reduce model.onnx -o reduced.onnx
```

- *Note: Interactive mode may not be available as of this talk, but will be public very soon!*

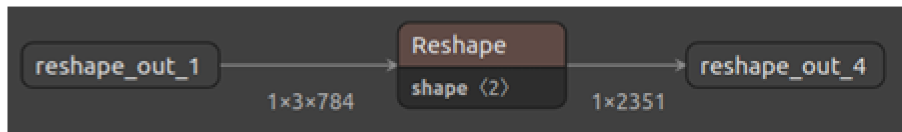
```
pranavm in ~ λ polygraphy debug reduce model.onnx -o reduced.onnx
```

```
pranavm in ~ λ polygraphy run polygraphy_debug.onnx --onnxrt
```

Model Bisection: An Example

Interactive Mode: Results

- Here's what we're left with:



- Now we can clearly see that the Reshape is invalid!

Model Bisection: An Example

Automatic Mode

- We can do the same thing in an automated fashion
- Instead of running a command ourselves, we tell **`debug reduce`** which command to run:

```
$ polygraphy debug reduce model.onnx -o reduced.onnx \
    --check polygraphy run polygraphy_debug.onnx --onnxrt
```

- The resulting model is exactly the same as before

Contact Information

Email: pranavm@nvidia.com

Questions?