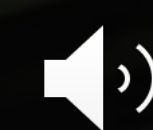


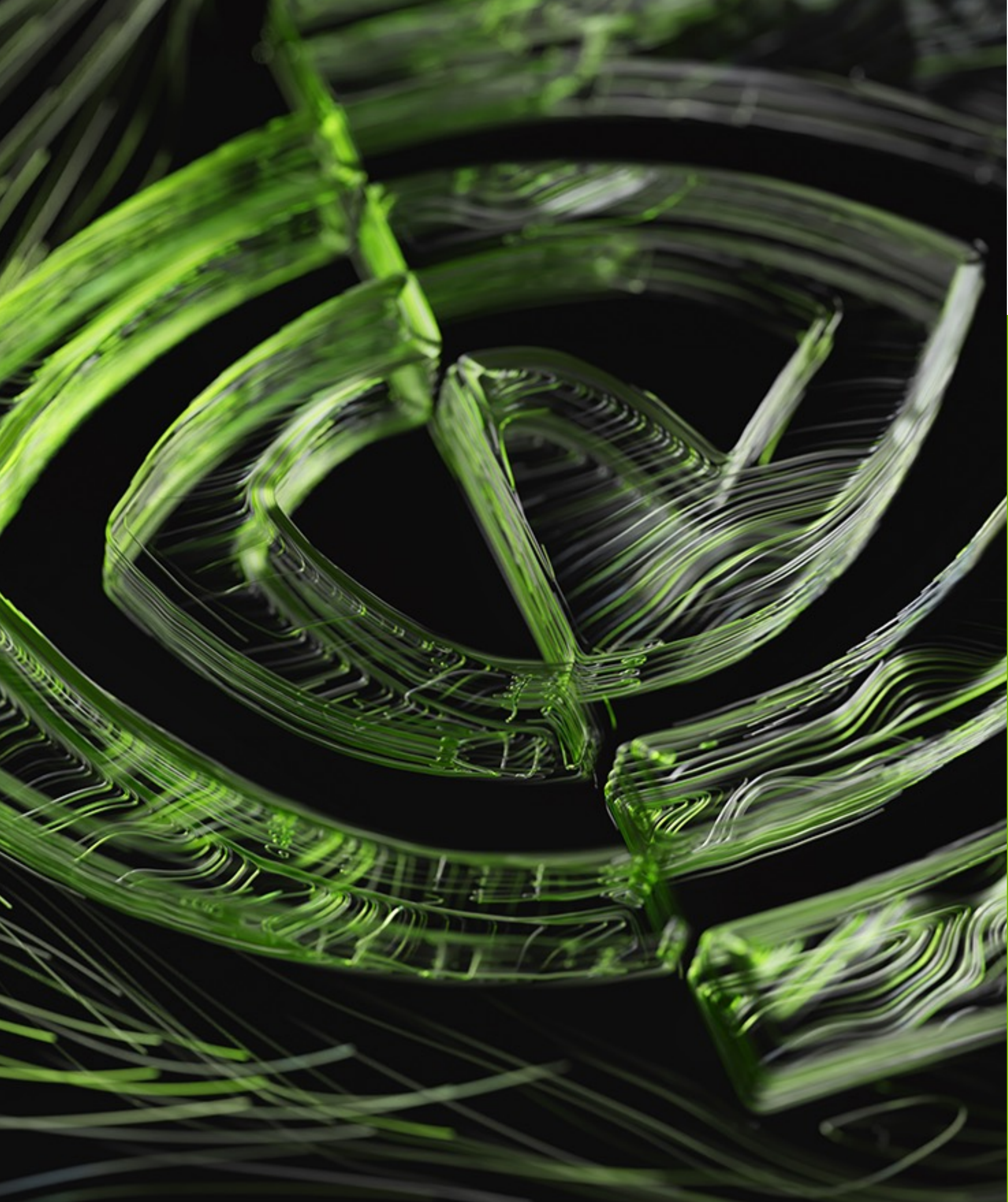


Accelerating Inference in PyTorch 2.0 with TensorRT

Naren Dasan, NVIDIA

Wei Wei, Meta





Agenda

- PyTorch 2.0

- TensorRT

- Torch-TensorRT 2.0

- Transitioning from TorchScript to Dynamo

- Wrap Up

PyTorch 2.0



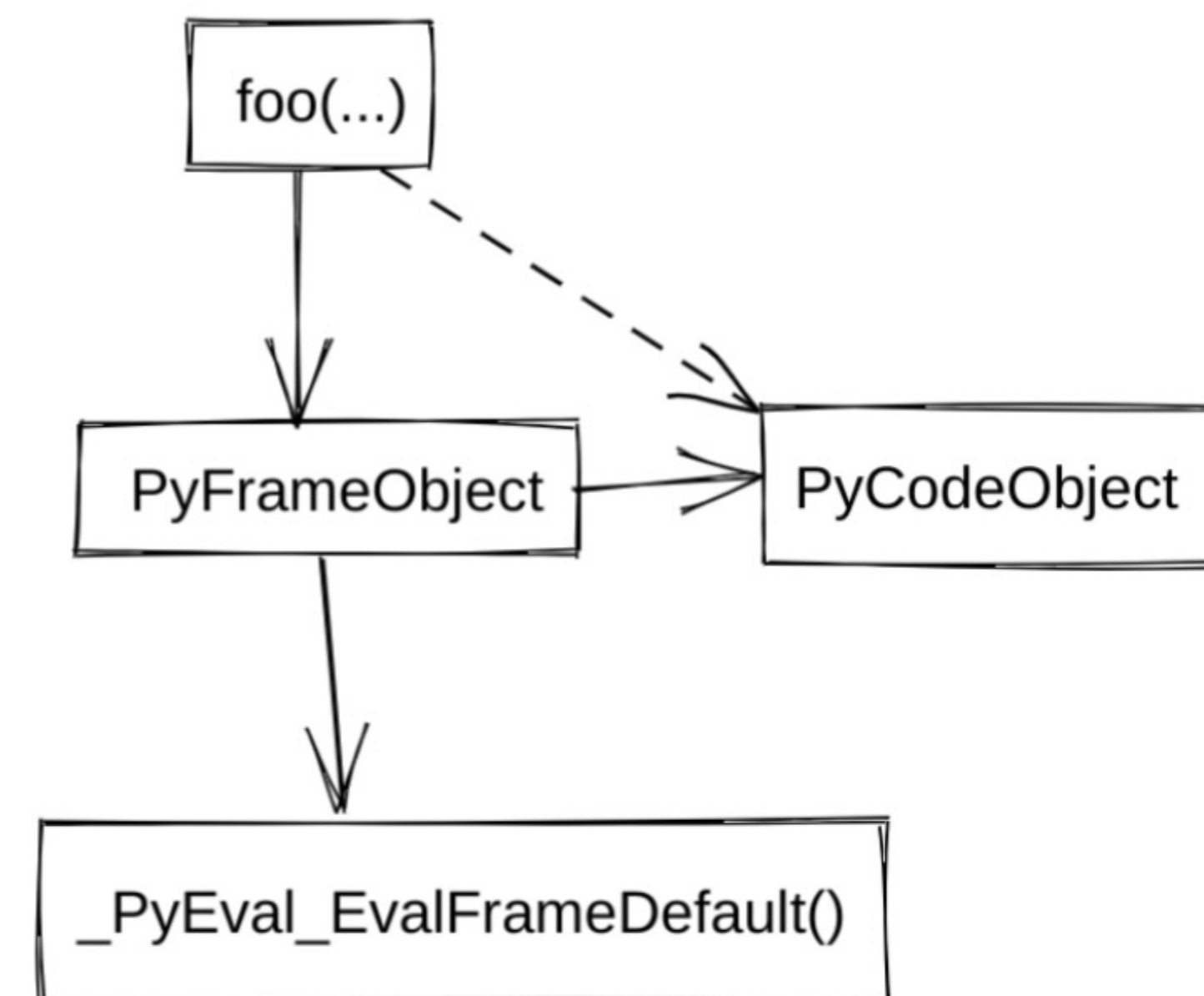
Recap on PT2.0

- PT 2.0 announced in Dec. 2022
- Backward compatibility
- Major components and capabilities
 - Torchdynamo
 - Dynamic shape
 - Export path
 - TorchInductor

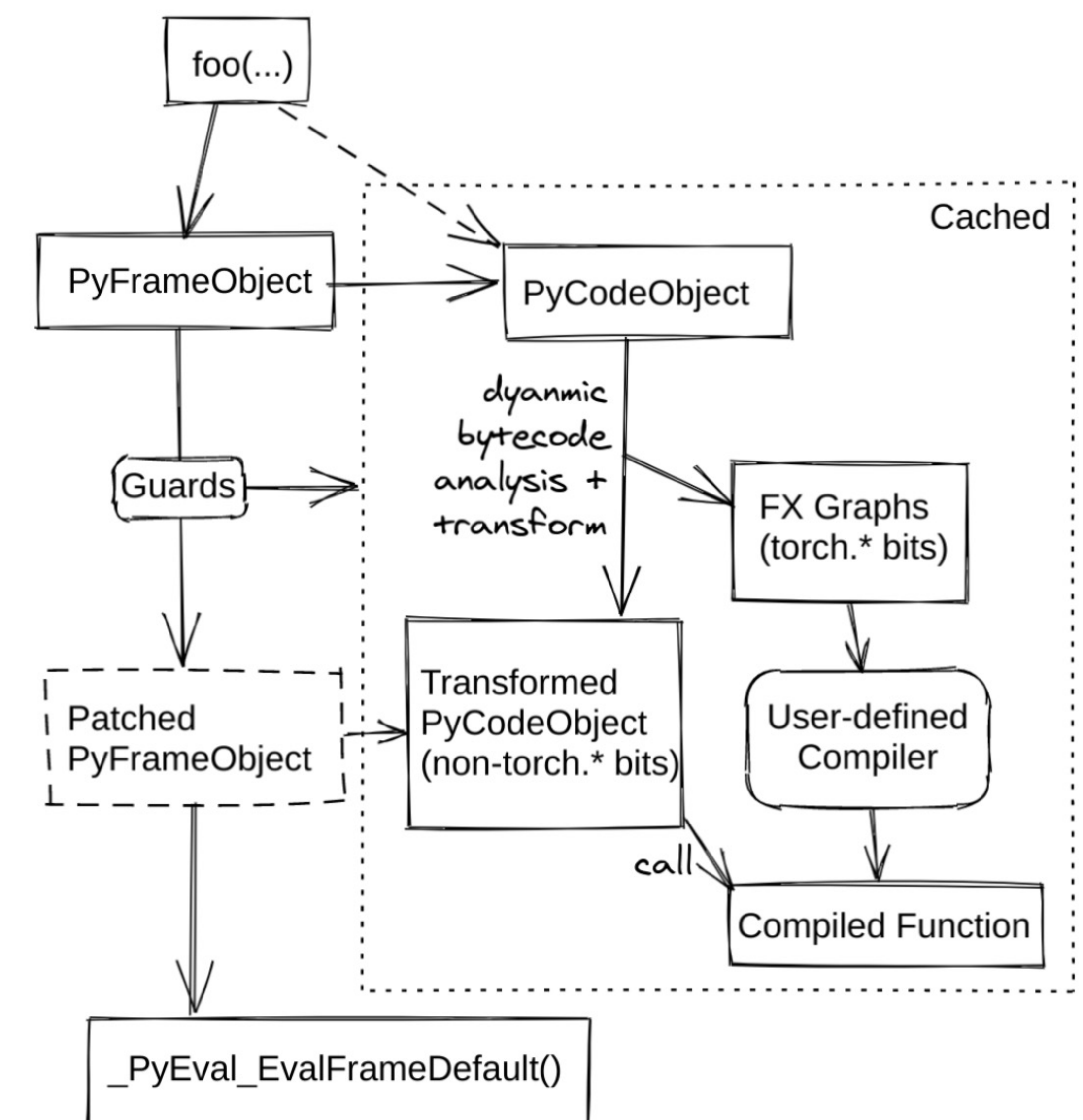
TorchDynamo

- **Control flow graph capture**
 - Cached subgraph capture
- **Guarded graphs**
 - Sound graph capture with checks
- **Just-in-time recapture**
 - Recapture a graph if captured graph is invalid for execution
- **Integrated with various backend**
 - Training
 - Inference

Default Python Behavior



TorchDynamo Behavior



Dynamic Shapes Support

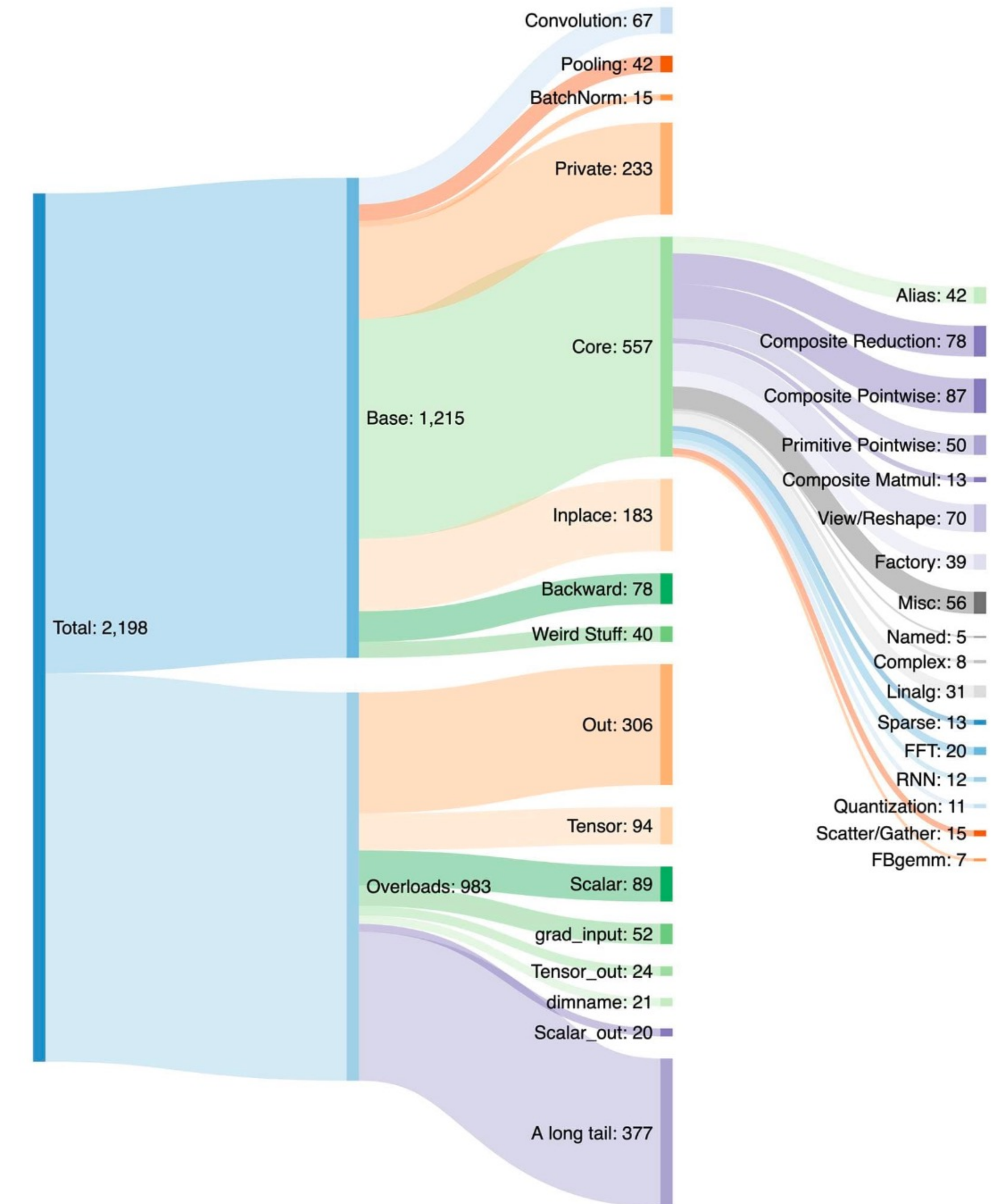
- The trace graph supports dynamic shape export
- Symbolic representations for shapes
 - `torch.size([2, 3]) -> torch.size([s0, s1])`
 - Complicated shape computations are clear
- Tracing with AOTAutograd
 - Leverages PyTorch's `__torch_dispatch__`
 - Trace fwd and bwd
 - Aten IR representation

PT 2.x Export Path

- Targeting whole graph capture and optimization
- Features
 - One graph export API
 - Simplified and core IR and operator set
 - Interface for connecting Dynamo and various backends
- Integrate with backend compiler for ML deployment

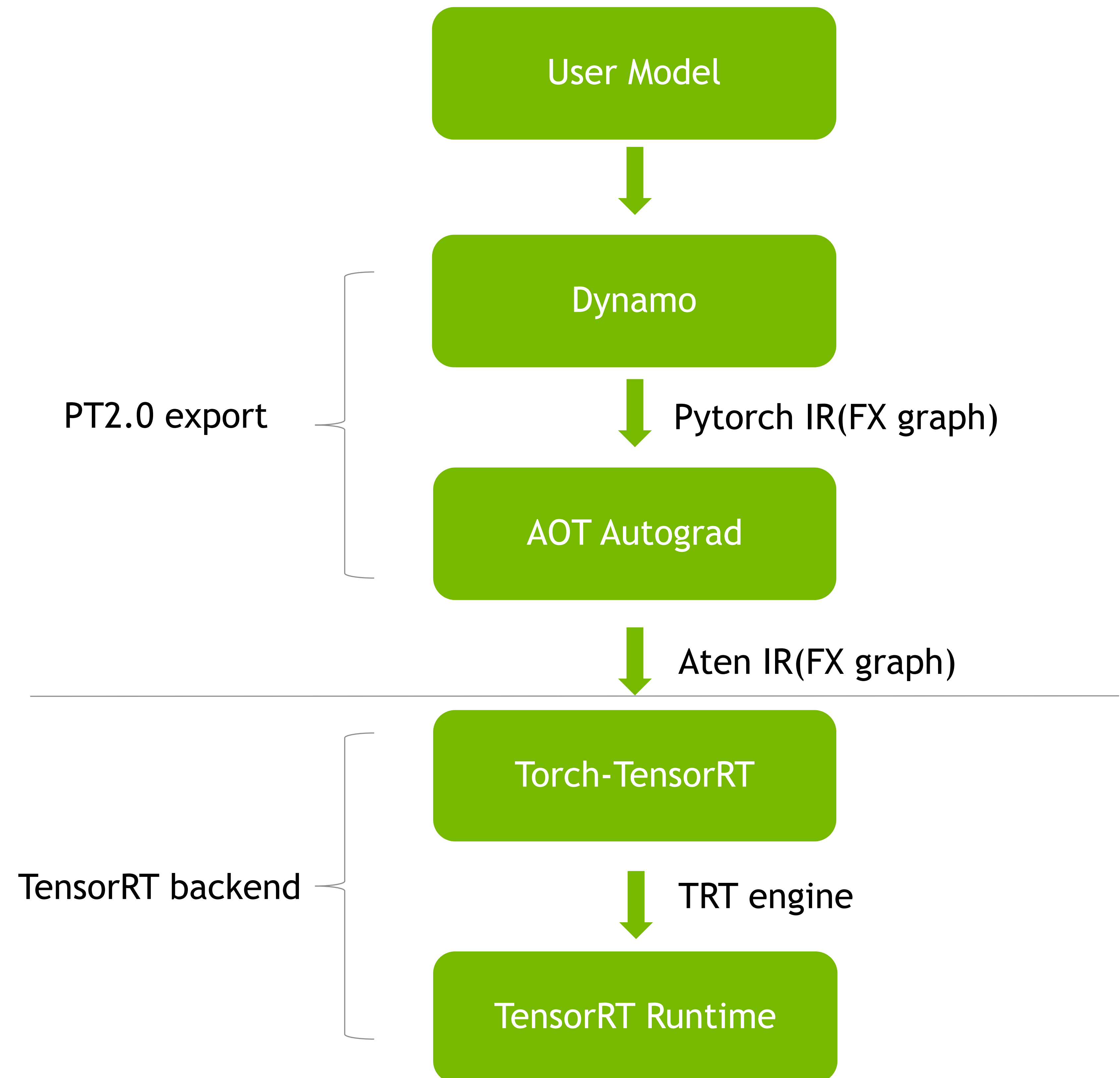
Aten IR

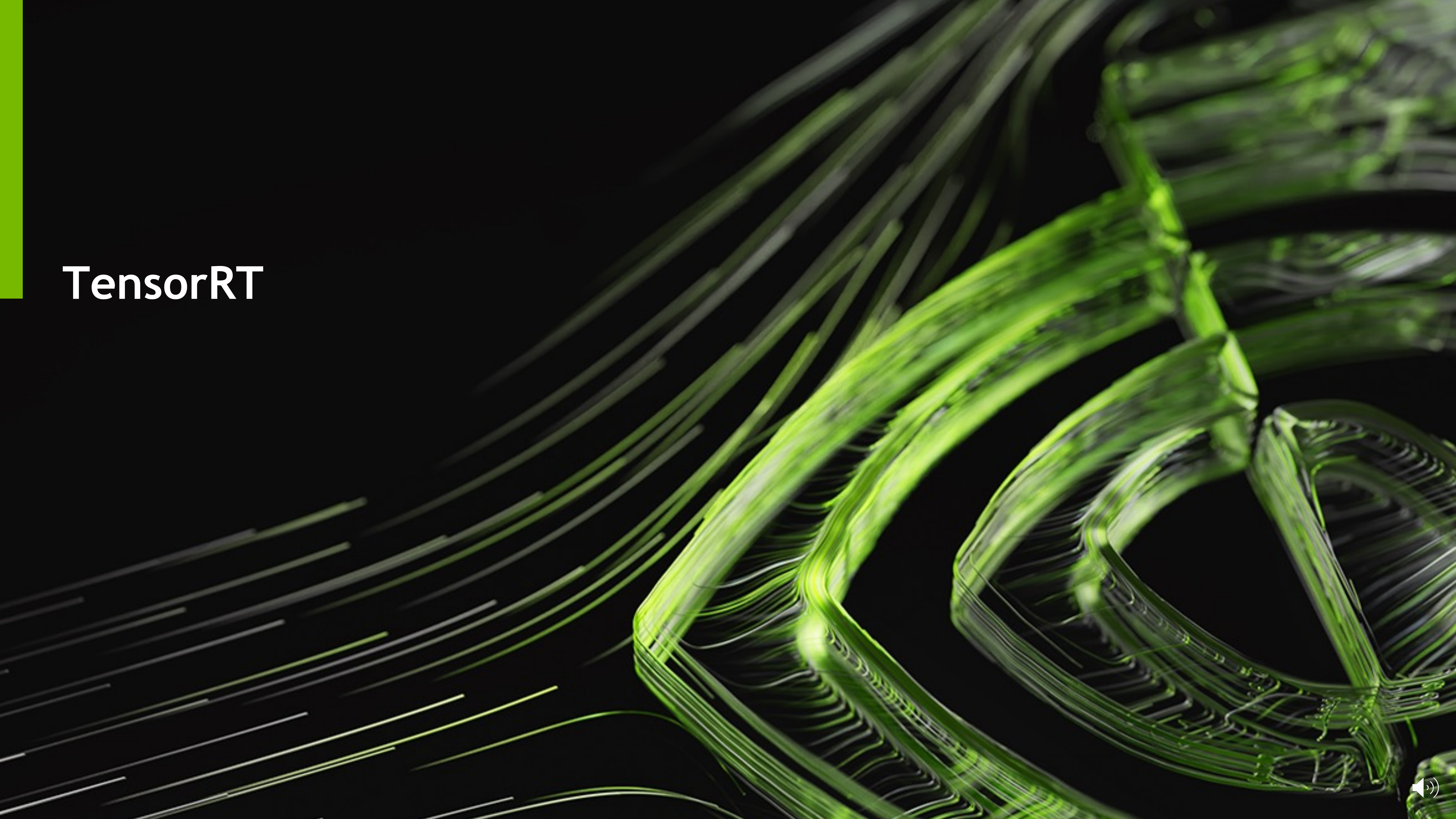
- Pytorch IR vs Core Aten IR
- Reduce the redundancy compared with PT ops
- A strict subset of Aten operators (< 250) after decompositions
- Purely functional (no inputs mutations)
- Guaranteed metadata information, e.g. dtype and shape propagation
- Effectively the output IR of PT2's export path



PT2.0 Integration Interface with TensorRT

- PT2.0 export capture the model graph
- Aten IR layer as integration interface
 - Small op set
 - Dynamic shape
 - Meta data
- Torch-TensorRT will convert Aten IR graph to TRT engine





TensorRT



NVIDIA TensorRT

Optimization Highlights

Reduced Mixed Precision for TF32, FP16, & INT8 execution

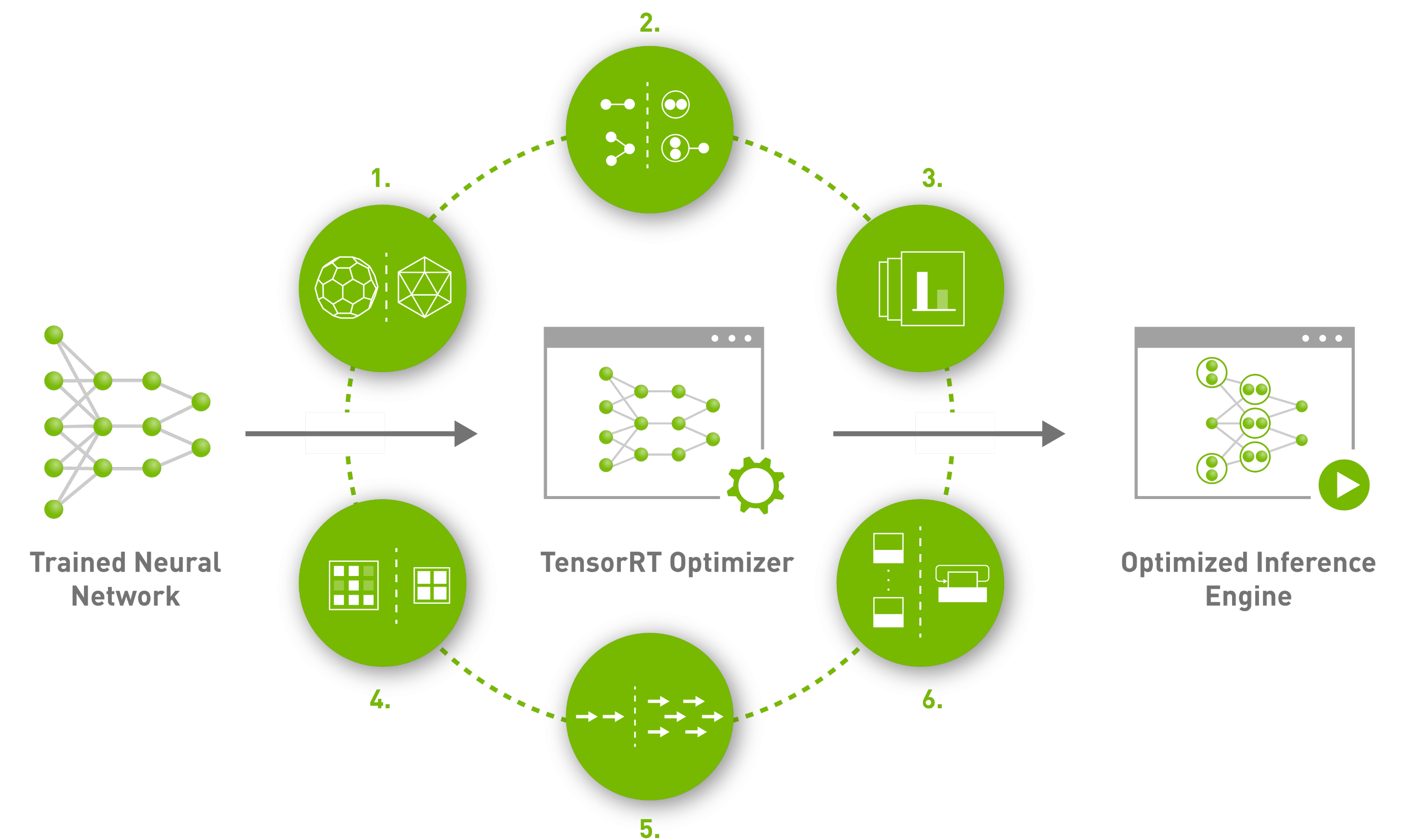
Layer & Tensor Fusion optimizes memory bandwidth

Kernel Auto-Tuning for targeting specific GPU deployment

Dynamic Tensor Memory to deploy memory-efficient apps

Multi-Stream Execution for scalable design

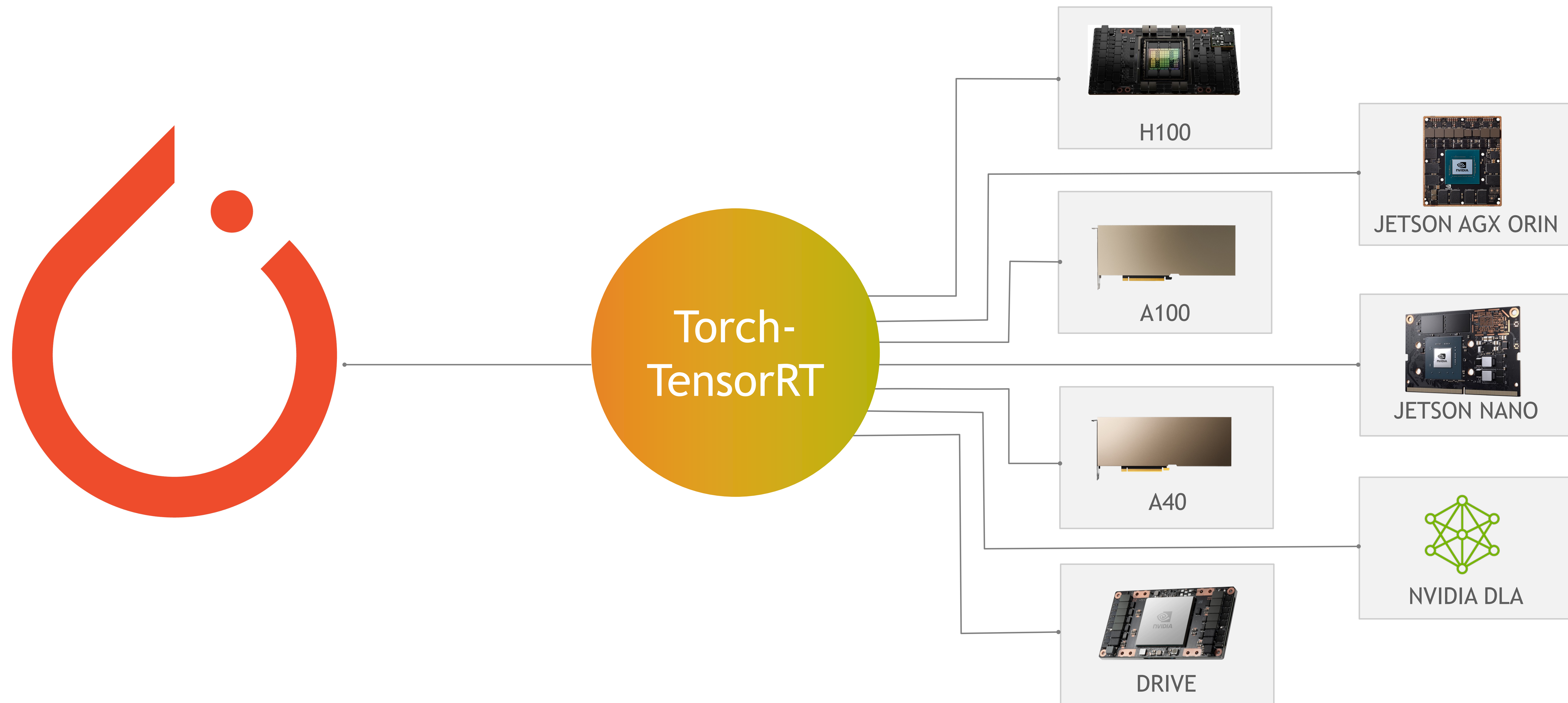
Time Fusion to optimize RNN time steps



<https://developer.nvidia.com/tensorrt>

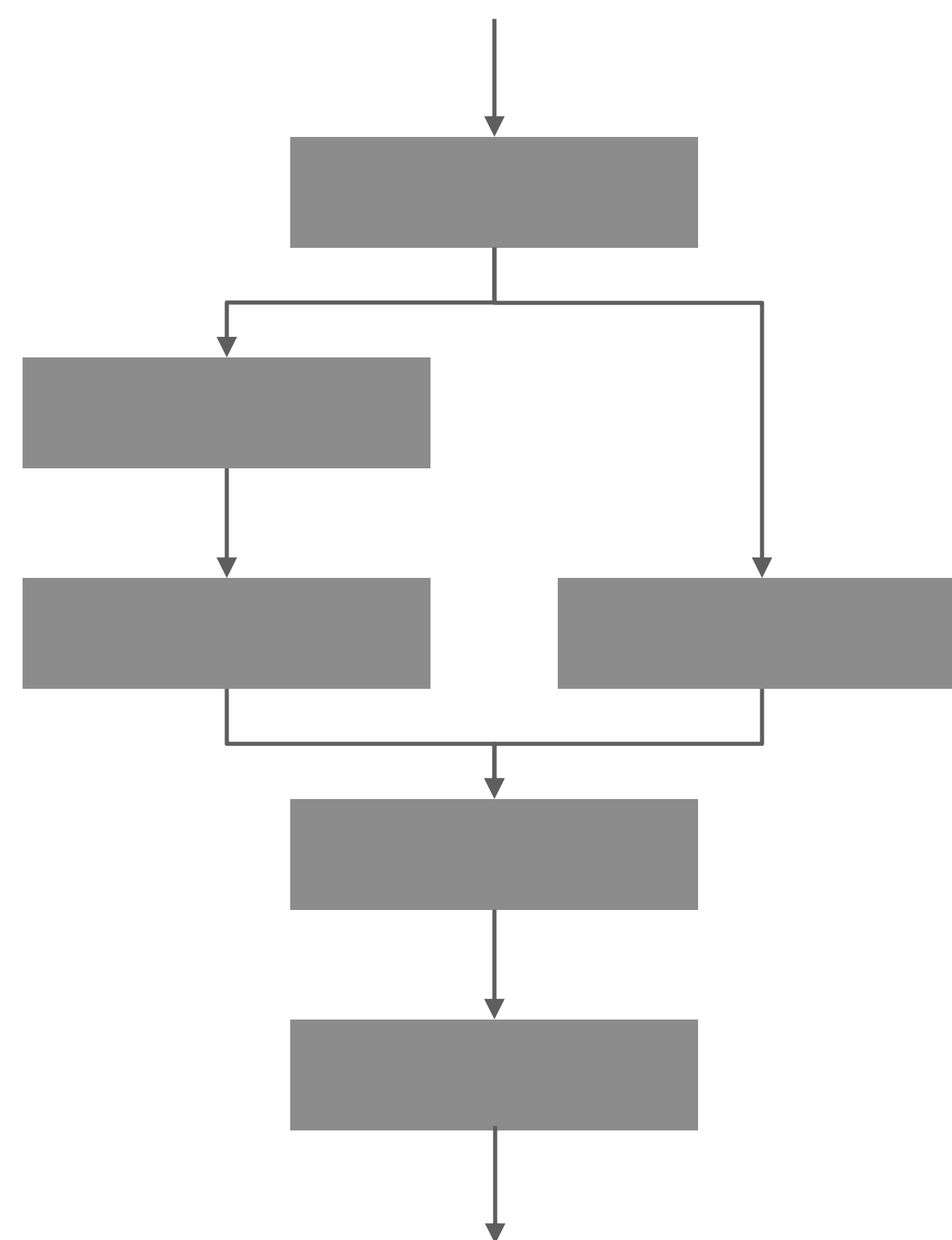
Optimized Deployments

Directly from PyTorch to Optimized Target Platform Deployment

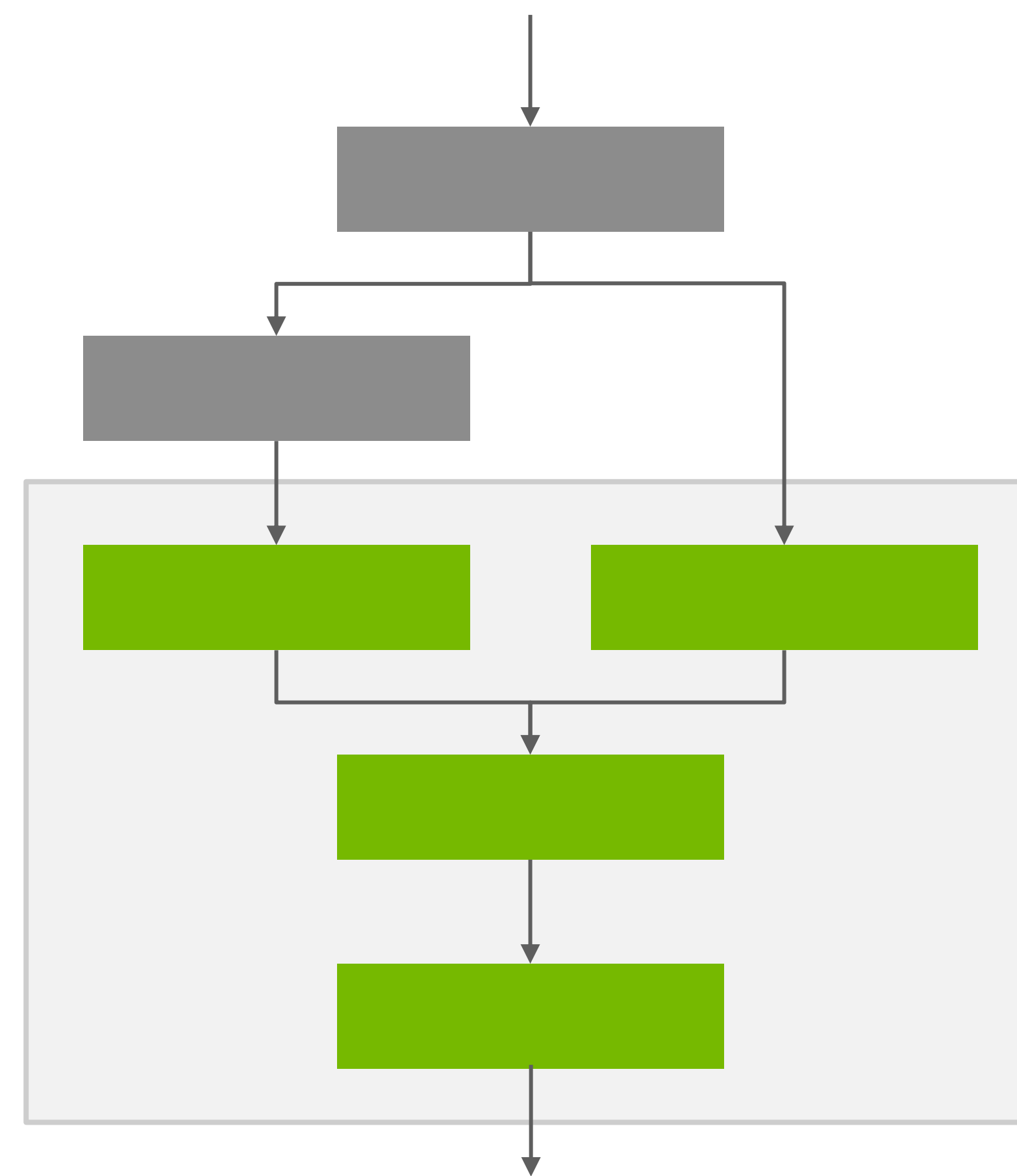


Partial Compilation

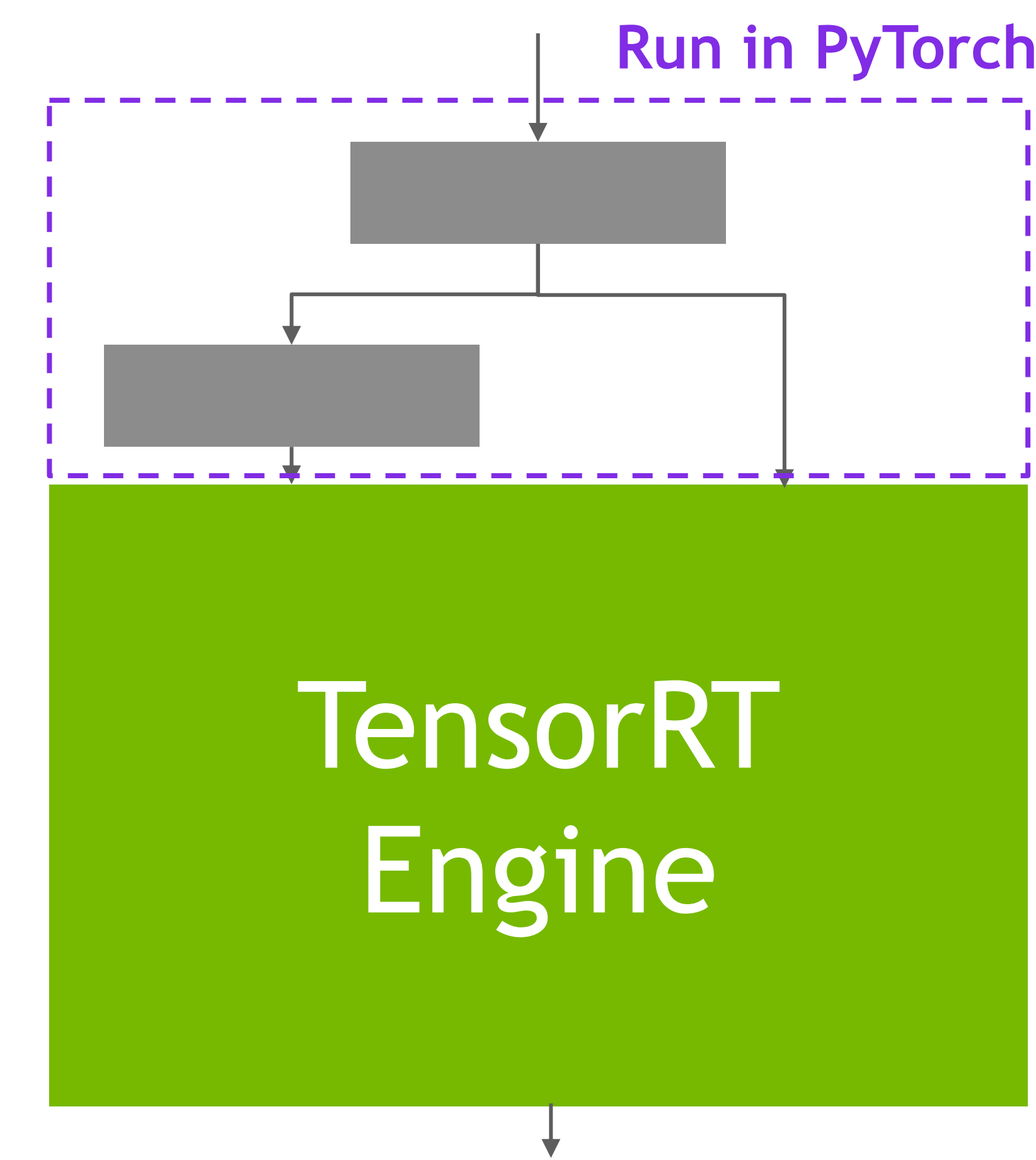
Hybrid PyTorch - TensorRT Execution



PyTorch Model



1. Partition Graph
Identify supported nodes



2. Compile TensorRT
Convert nodes to TensorRT

Torch-TensorRT 2.0



Torch-TensorRT 2.0

- Coming H2Y23
- PyTorch 2.0 native workflow
 - `torch.compile`
 - Python based converter library and workflow for easy extensions and customization
 - Deployment in Python or C++ via TorchScript



Torch-TensorRT 1.x FX SYSTEM ARCHITECTURE

The Four Stages

- Trace
 - Transform the Torch operations into FX graph
- Split
 - Separate graph into subgraphs with TensorRT supported and unsupported nodes
- Convert
 - Take FX Graph and build a TRT engine
- Execute
 - Instantiate the engine with input and engine



Torch-TensorRT 2.0 System Architecture

The New Four Stages

- Trace with PT export (via Dynamo + AOTAutograd)
 - Transform the Torch operations into FX graph
- Split
 - Separate graph into subgraphs with TensorRT supported and unsupported nodes
- Convert with Aten IR
 - Take FX Graph and build a TRT engine
- Execute
 - Instantiate the engine with input and engine



Modifications to Torch-TensorRT

- PT2.0 export
 - Not finalized
 - Symbolic shape
 - Aten IR
 - vs `mod = acc_tracer.trace(mod, inputs)`
- Converter modification
 - Effort is less
 - Most ops conversions are reusable
 - Shape ops simplified
 - Extra cost on shape ops operations

```
with using_config(dynamo_config), setting_python_recursive_limit(2000):  
    torchdynamo.reset()  
    try:  
        return torchdynamo.export(  
            f,  
            *copy.deepcopy(args),  
            aten_graph=aten_graph,  
            tracing_mode=tracing_mode,  
        )
```

```
@tensorrt_converter(torch.ops.aten.mul.Tensor)  
def aten_ops_mul(  
    network: TRTNetwork,  
    target: Target,  
    args: Tuple[Argument, ...],  
    kwargs: Dict[str, Argument],  
    name: str,  
    ) -> Union[TRTTensor, Sequence[TRTTensor]]:  
    kwargs_new = {  
        "input": args[0],  
        "other": args[1],  
    }  
    return acc_ops_converters.acc_ops_mul(network, target, None, kwargs_new, name)
```

Demo of Resnet18

- Support of Aten ops in early stage
 - PT2.0 trace integrated
 - Supported 16 ops in converters
 - More op support is coming
 - OSS contributions are welcome
- Resnet18 model supported
 - E2E runnable
 - Performance on par
 - `is_aten=True` in `compile()`
 - Play with it in (GH: pytorch/tensorrt): `examples/fx/lower_example_aten.py`



Demo of Resnet18

- *python lower_example.py* (PT2.0 not enabled)

```
== Benchmark Result for: Configuration(batch_iter=50, batch_size=128, name='CUDA Eager', trt=False, jit=False, fp16=False, accuracy_rtol=-1)
BS: 128, Time per iter: 13.69ms, QPS: 9350.41, Accuracy: None (rtol=-1)
== Benchmark Result for: Configuration(batch_iter=50, batch_size=128, name='TRT FP16 Eager', trt=True, jit=False, fp16=True, accuracy_rtol=0.01)
BS: 128, Time per iter: 3.50ms, QPS: 36551.82, Accuracy: None (rtol=0.01)
```

```
Supported node types in the model:
acc_ops.conv2d: (C), {'input': torch.float16, 'weight': torch.float16})
acc_ops.batch_norm: (C), {'input': torch.float16, 'running_mean': torch.float16, 'running_var': torch.float16, 'weight': torch.float16, 'bias': torch.float16})
acc_ops.relu: (C), {'input': torch.float16})
acc_ops.max_pool2d: (C), {'input': torch.float16})
acc_ops.add: (C), {'input': torch.float16, 'other': torch.float16})
acc_ops.adaptive_avg_pool2d: (C), {'input': torch.float16})
acc_ops.flatten: (C), {'input': torch.float16})
acc_ops.linear: (C), {'input': torch.float16, 'weight': torch.float16, 'bias': torch.float16})
```

Unsupported node types in the model:

Got 1 acc subgraphs and 0 non-acc subgraphs

- *python lower_example_aten.py* (PT2.0 enabled)

```
== Benchmark Result for: Configuration(batch_iter=50, batch_size=128, name='CUDA Eager', trt=False, jit=False, fp16=False, accuracy_rtol=-1)
BS: 128, Time per iter: 13.72ms, QPS: 9326.74, Accuracy: None (rtol=-1)
== Benchmark Result for: Configuration(batch_iter=50, batch_size=128, name='TRT FP16 Eager', trt=True, jit=False, fp16=True, accuracy_rtol=0.01)
BS: 128, Time per iter: 3.38ms, QPS: 37864.66, Accuracy: None (rtol=0.01)
```

```
Supported node types in the model:
torch.ops.aten.convolution.default: ((torch.float16, torch.float16), {})
torch.ops.aten.batch_norm: ((torch.float16, torch.float16, torch.float16, torch.float16, torch.float16), {})
torch.ops.aten.relu.default: ((torch.float16,), {})
torch.ops.aten.max_pool2d: ((torch.float16,), {})
torch.ops.aten.add.Tensor: ((torch.float16, torch.float16), {})
torch.ops.aten.mean.dim: ((torch.float16,), {})
torch.ops.aten.sym_size: ((torch.float16,), {})
torch.ops.aten.view.default: ((torch.float16,), {})
torch.ops.aten.linear: ((torch.float16, torch.float16, torch.float16), {})
```

Unsupported node types in the model:

Got 1 acc subgraphs and 0 non-acc subgraphs

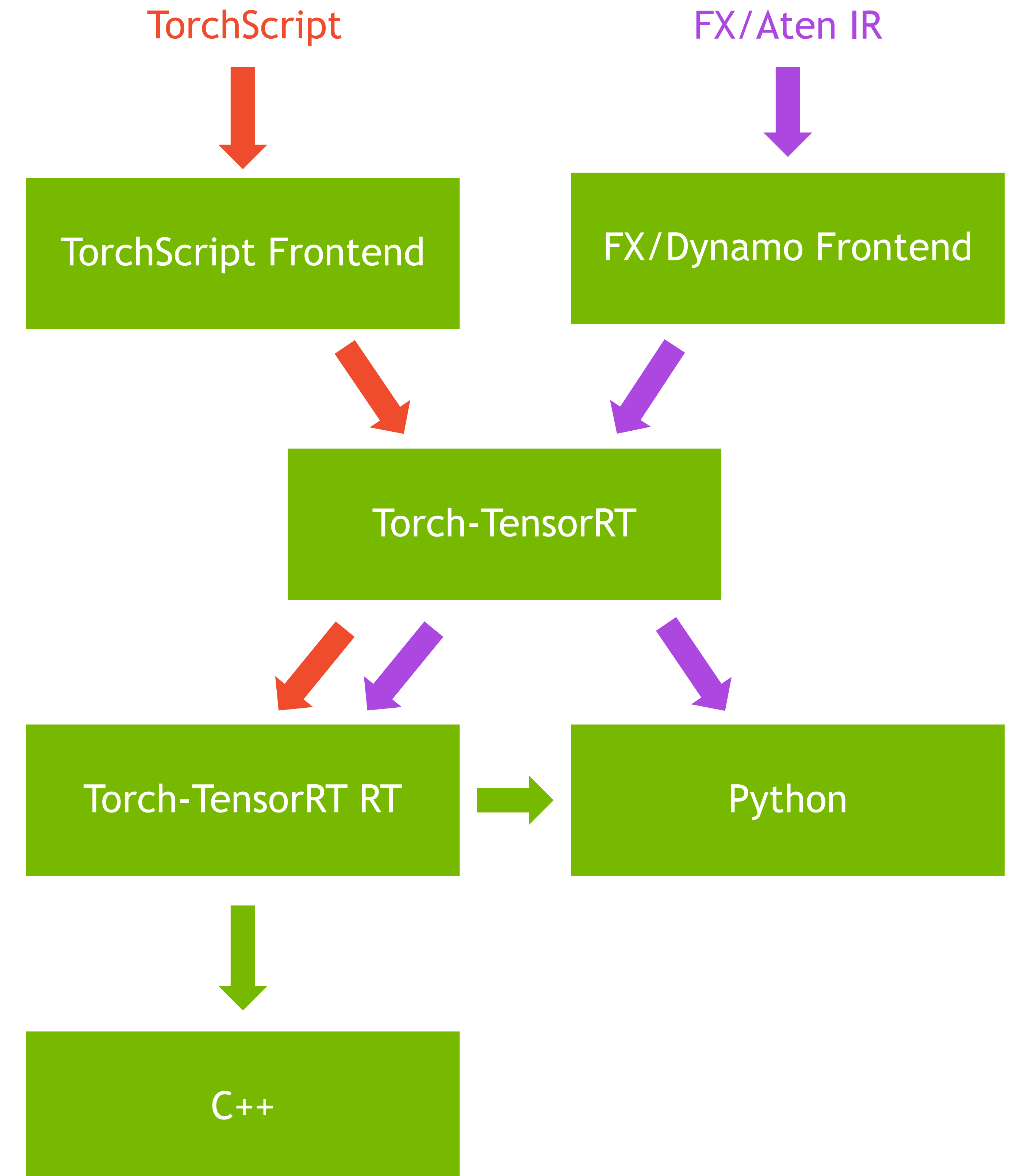
Transitioning from TorchScript to PyTorch 2.0



Transitioning from TorchScript to PyTorch 2.0

Compiling

- `torch_tensorrt.compile` provides a unified interface to both TorchScript and FX/Dynamo frontends
- Controlled via the `ir` flag
- Both frontends standardize on the same APIs for controlling compilation
- Transitioning therefore is just flipping a switch



Transitioning from TorchScript to PyTorch 2.0

Runtime

- Keep using the same Torch-TensorRT runtime with the FX frontend, with its inherent features (portability, low footprint) in C++ or Python
 - Serializable via `torch.save/torch.load`, `state_dict`, `torch.jit.trace`
- For pure python based deployments, TensorRT subgraphs can be wrapped in a fully standard PyTorch module

Using the Torch-TensorRT Runtime with FX/Dynamo

```
model_fx = model_fx.cuda()
inputs_fx = [i.cuda() for i in inputs_fx]
trt_fx_module_f16 = torch_tensorrt.compile(
    model_fx,
    ir="fx",
    inputs=inputs_fx,
    enabled_precisions={torch.float16},
    use_experimental_fx_rt=True,
    explicit_batch_dimension=True
)

# Save model using torch.save

torch.save(trt_fx_module_f16, "trt.pt")
reload_trt_mod = torch.load("trt.pt")

# Trace and save the FX module in TorchScript
scripted_fx_module = torch.jit.trace(trt_fx_module_f16,
example_inputs=inputs_fx)
scripted_fx_module.save("/tmp/scripted_fx_module.ts")

# In new process
import torch_tensorrt
scripted_fx_module = torch.jit.load("/tmp/scripted_fx_module.ts")
```

- Get an model in Aten IR via tracing or dynamo (torch.export)
- Compile using torch_tensorrt FX frontend
- **Target the Torch-TensorRT runtime instead of pure-python runtime**
- Compiled module is savable using traditional PyTorch methods
- Torch-TRT runtime modules are TorchScript traceable in order to export to non Python environments
- Loadable just like TorchScript modules



Wrap Up



Torch-TensorRT 2.0

- Coming H2Y23
- Native PyTorch 2.0 workflow
- Leverages the easier usability and better program acquisition of TorchDynamo to make it easier to get your models compiled by Torch-TensorRT
- Converters and compiler implemented in Python for easier modification and extensibility
- Can still deploy using TorchScript, or in pure Python

