PyTorch 2.0 Backend Integration

Sherlock Huang
Software Engineer, Meta

Agenda

01

PT2 Stack Overview

02

IRs and Integration Points

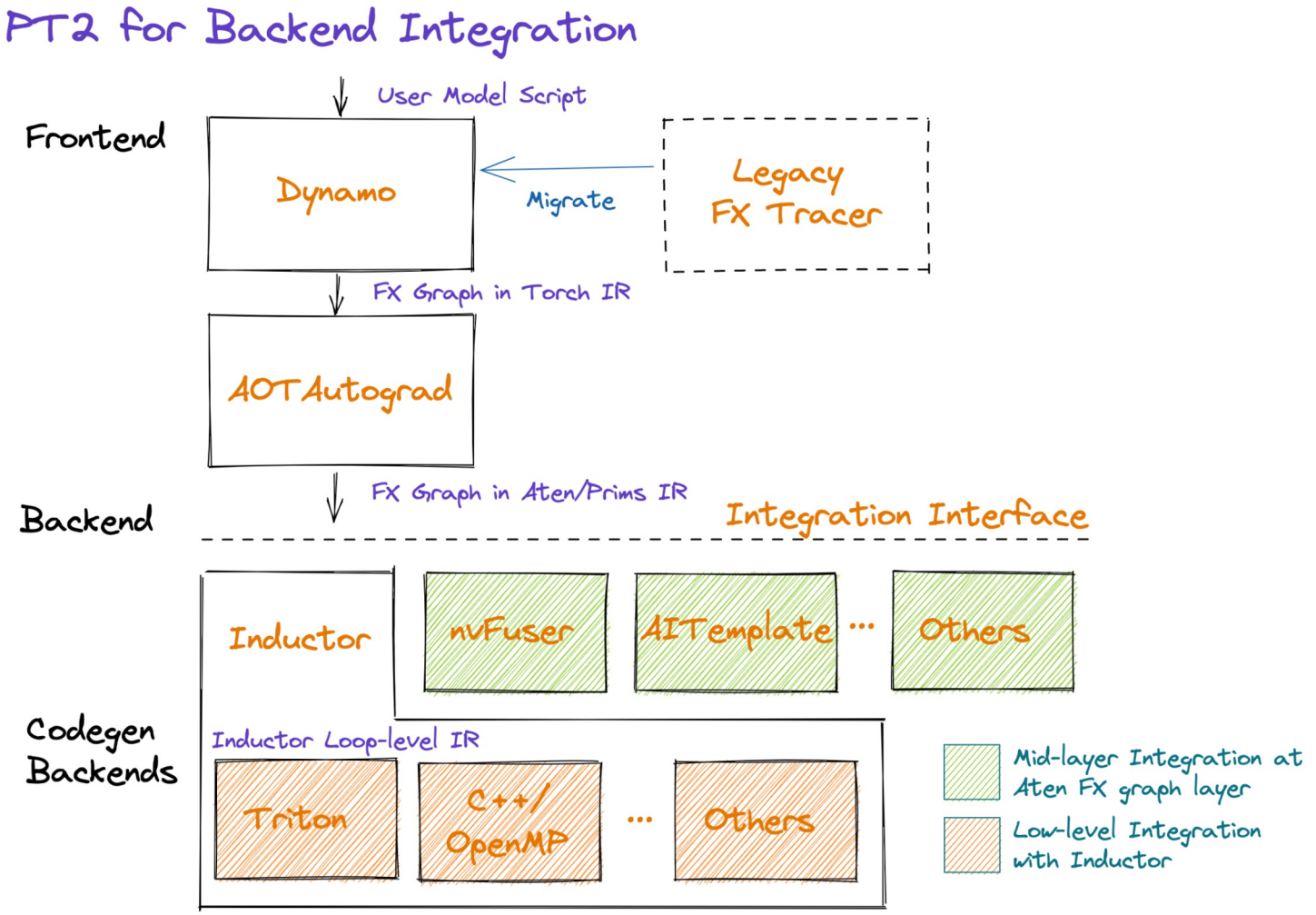
03

Examples and Utilities

Design Considerations

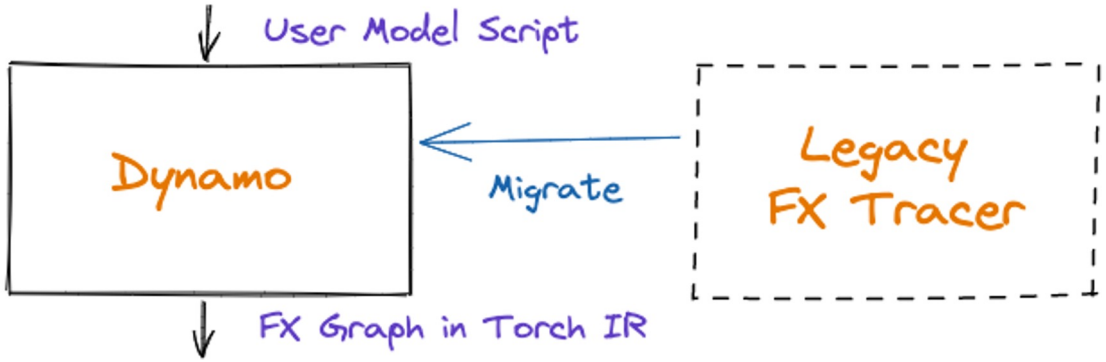
- Diverse backends in pytorch Eco-system
  - AI accelerators, libraries, 3rd party ML frameworks, hand written kernels …
  - Backends need different level of abstraction for operators
  - Backends have different coverage of pytorch operators
- Various scenarios
  - Compiler-accelerated eager, export path …
  - Training and Inference

- Three IRs
  - Torch IR
  - Aten/Prims IR
  - Inductor DBR Loop-level IR
- Two integration points
  - Mid-layer with Aten FX graph
  - Low-level with Inductor



Dynamo: Graph Capturing Frontend

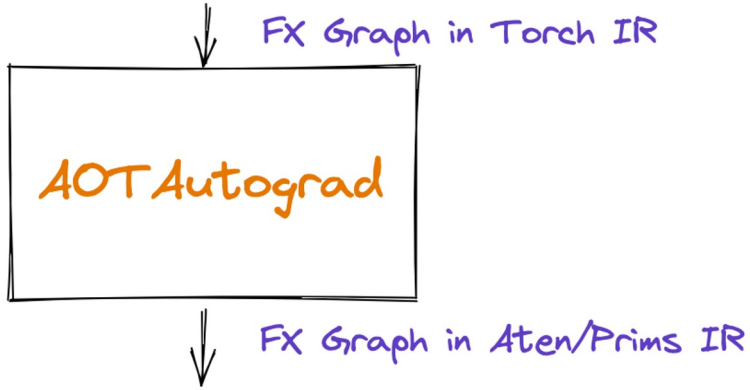
- Soundness with guard
- Supports dynamic shape



Torch IR

- Not ideal for compiler backends
  - FX graph with call_module,call_function, call_method
  - Large operator surface
- Good for nn.module level graph transformations

AOTAutograd



- Captures forward + backwards
- Lowering from Torch IR to Aten/Prims IR
- Cleansed graph via functionalization
- Populated graph with metadata: symbolic shape, dtype…
- Communication collective ops will be traced via AOTAutograd
- Configurable decompositions into a smaller opset

Configurable Decompositions

- Allows backends to fine tune the opset for FX graph
- Writing decompositions as pytorch function for aten ops
- AOTAutograd accepts a user-defined decomposition table
  - Choose from existing decompositions
  - Write your own decompositions

```
@register_decomposition(aten.native_dropout)
def native_dropout(input: Tensor, p: float, train: Optional[bool]):
    if train:
        bool_mask = torch.rand_like(input) > p
        res = bool_mask * input * float(1.0 / (1.0 - p))
        return (res, bool_mask)
    else:
        return (input, torch.ones_like(input, dtype=torch.bool))
```

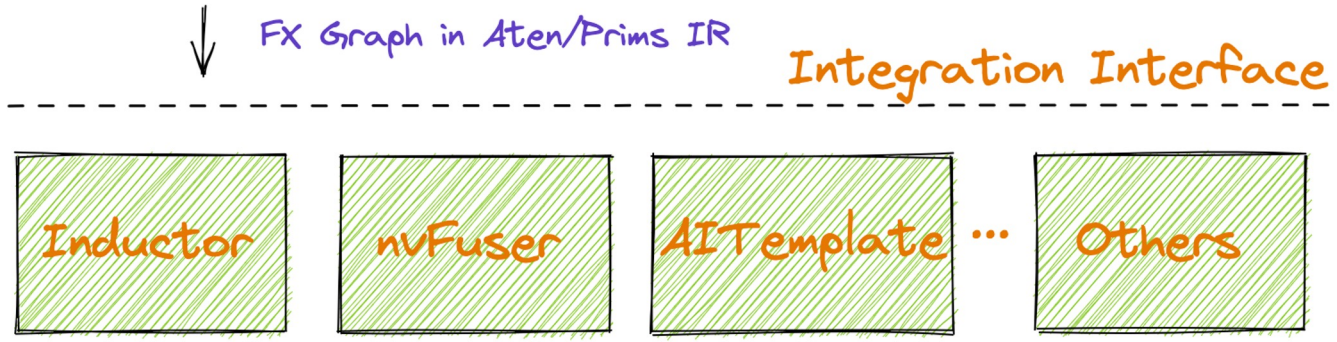
Prims IR

- A primitive opset with ~250 ops
- Explicit type promotion and broadcasting
  - prims.convert_element_type
  - prims.broadcast_in_dim
- For backends with powerful compiler that can reclaim the performance by fusion, e.g. Inductor and nvFuser

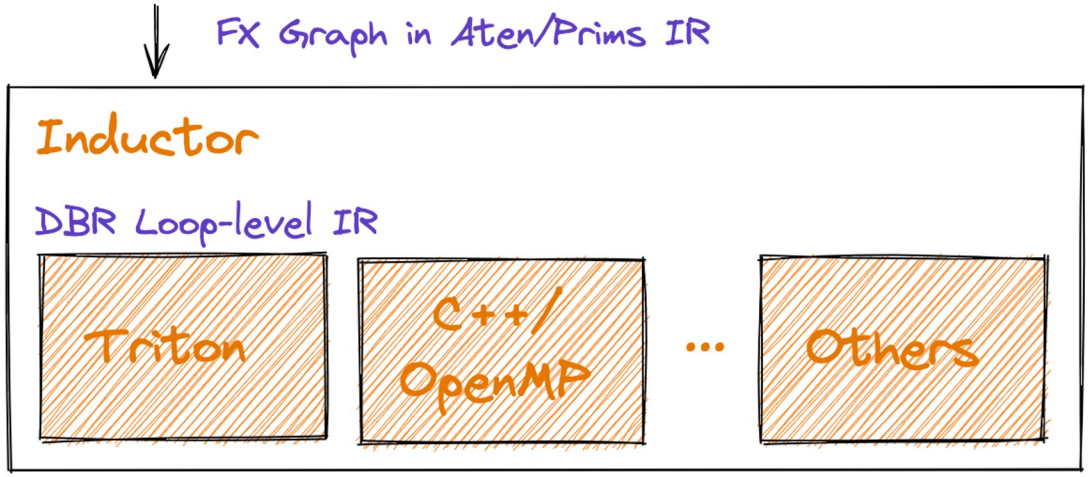
Canonical Aten IR

- A strict subset of existing aten operators after decompositions
- Purely functional (no inputs mutations)
- Guaranteed metadata information, e.g. shape propagation
- Effectively the output IR of PT2's export path
- For backends that wants an eager-like opset, that doesn't have a fusion compiler

Backend Integration Interface (Coming Soon)



- Backend registration
  - Declaring capability: supported ops, specialized patterns
- Resource coordination
  - GPU memory arena, thread pool…
- Work Arbitration
  - Delegate subgraph to the optimal backend



- PyTorch-native compiler
  - Supports dynamic shape and stride
  - Addresses nuanced pytorch semantics, e.g. aliasing/mutation/view…
- Backend configurable optimizations
  - Layout tuning, fusion decisions, tiling, autotuning…
- Codegen backends
  - Triton for CUDA
  - C++/OpenMP for x86 CPU
  - TVM under exploration by OctoML team

Define-by-run (DBR) Loop-level IR

x.permute(1, 0) + x[2, :] becomes:

```
def inner_fn(index: List[sympy.Expr]):
    i1, i0 = index
    tmp0 = ops.load("x", i1 + i0*size1)
    tmp1 = ops.load("x", 2*size1 + i0)
    tmp2 = ops.add(tmp0, tmp1)
    return tmp2
```

```
torchinductor.ir.Pointwise(
        device=torch.device(...),
        dtype=torch.float32,
        inner_fn=inner_fn,
        ranges=[size0, size1],
)
```

Override `ops` to do analysis and backend codegen.

Examples and Utilities

Graph Partitioner

- Given the supported operators, it forms the largest subgraphs supported by a backend.

```
def partition(graph_modue: fx.GraphModule,
                    supported_ops: OperatorSupportBase) -> List[Subgraph]
```

- Partitioned subgraphs can be delegated to a backend
- Remaining graphs can fallback to aten kernel

IR-Agnostic Pattern Matcher

- Finds all the matched subgraph according to the provided pattern
- IR Agnostic: works for Torch IR, Aten IR, Prims IR

```
def match(target: fx.Graph, pattern: fx.Graph) -> List[MatchedSubgraph]
```

- Replace matched subgraph with another graph

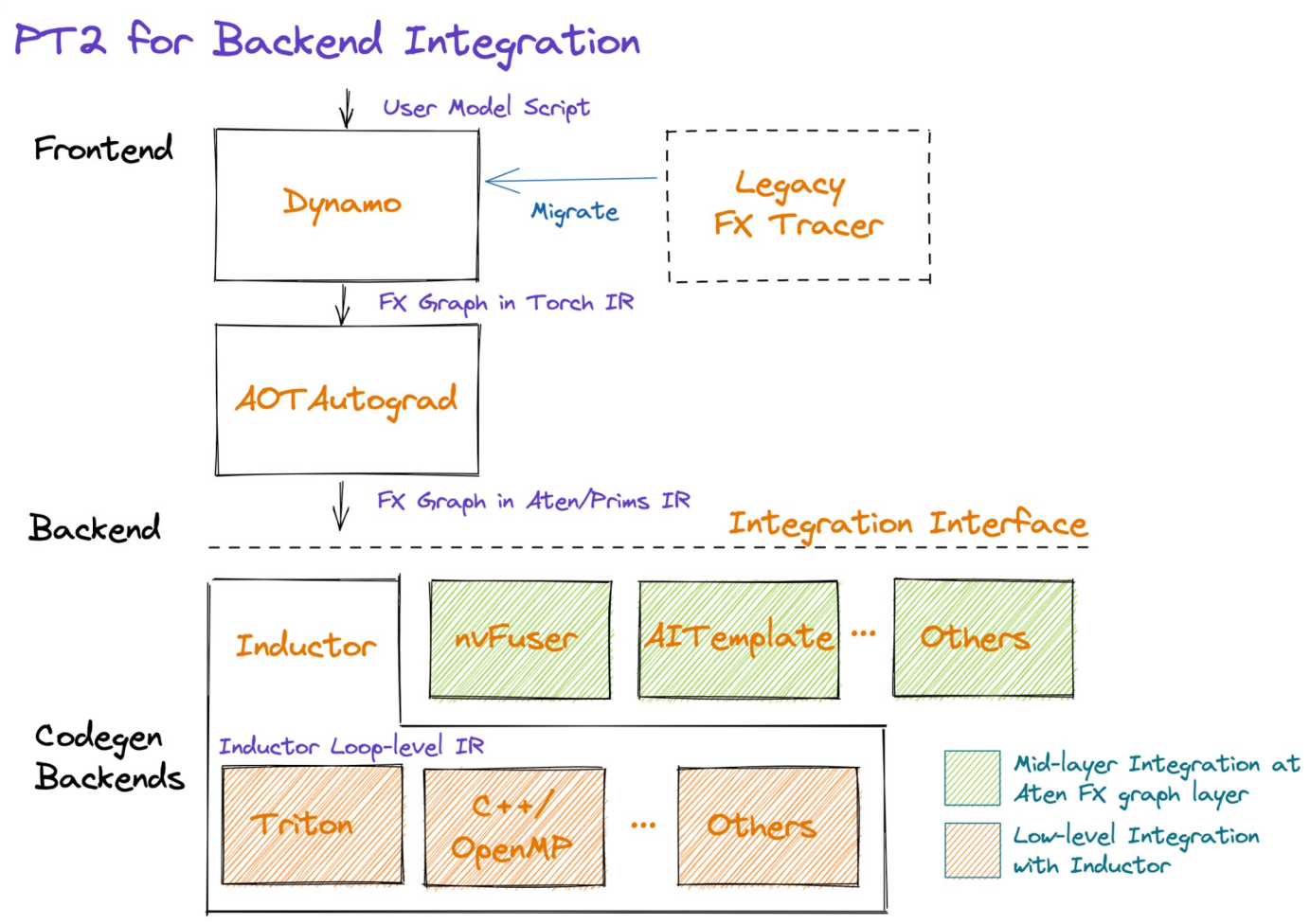
```
def replace_pattern(target: fx.Graph, pattern: fx.Graph, replacement: fx.Graph)
```

Examples of backends

- General full graph compiler backend + Aten fallback kernel
- Pattern specialized CustomOps + Inductor
- Codegen backends

Takeaway

- Three IRs
  - Torch IR
  - Aten/Prims IR
  - Inductor DBR Loop-level IR
- Two integration points
  - Mid-layer with Aten FX graph
  - Low-level with Inductor



Thank you