Sym Shapes, Control Flow in TorchDynamo - RFC + Plan of Record

TLDR:

1) Add shape expressions to dynamo via fake tensor
2) Install shape expression guards in dynamo
3) Add an AOTAutograd level fw and bw cache
4) Add a mode that either writes to the cache in (3) or bubbles up guards to the frontend cache in (2) and we can play around with which tuning is more performant.

Context:

There are a couple of overlapping problems we need to solve around TorchDynamo, AOTAutograd, and TorchInductor when it comes to symbolic shape evaluation.

Specifically, they are:

1) Caching code and subsequently invalidating the cache based on guards created in AOTAutograd and TorchInductor
2) Handling shape based control flow in TorchDynamo

Not all solutions to 1 solve 2, and not all solutions to 2 solve 1. They can be handled in a somewhat orthogonal way, but due to the overlap in existing logic, implementation, and concerns, it easier to address this as a single problem statement, articulated here:

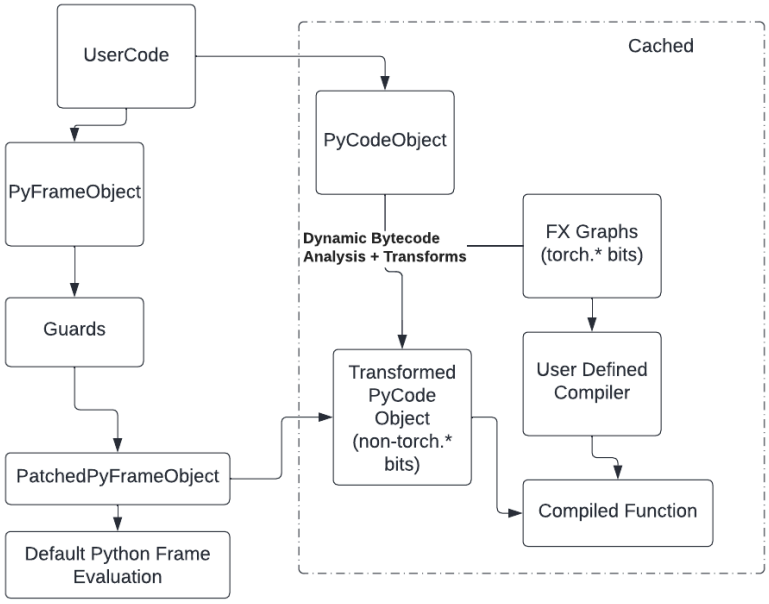
“How do we efficiently cache, and intelligently guard code, including shaped based control flow, in the PyTorch compiler stack?”

Current Landscape

In the current landscape, the rough flow is as follows:

Dynamo hooks into the interpreter, gets frames, and reads code out of it. As it does so, it pulls out and creates relevant tracking objects, which in turn use some properties of the object to determine how and when to guard. At this point, guards are being collected. Once dynamo has created its transformed code, it stores that to save recompilation time. In order to know if the value in the cache is valid, the set of guards collected above will be turned into a function ready for evaluation (check_fn). On a subsequent entry, for the same frame, dynamo will evaluate the

check_fn to determine if we have a cache miss or cache hit. For more information, check out: https://github.com/pytorch/torchdynamo/blob/main/documentation/GuardsOverviewPt1.md



Today's guards are:

- Frontend only
- Primarily for type/id matching
- Trigger a recompile which has us reprocess the whole frame
- Ex: `___guarded_code.valid and ___check_type_id(y, 94367738391392) and y == 2 and ___check_tensors(x)`

Beyond the frontend, we actually have another kind of guard. These guards are produced in AOTAutograd.

These guards are a little different, because they do not guard on types and ids as the Dynamo guards do - rather, they collect data during expression evaluation when running ops against FakeTensor, and produce these symbolic expressions.

For example:

```
torch.cat((a, a)) + b
```

Will produce an expression guard of `Eq(s0 * 1, s1)`

This guard then states that the shapes above will always conform to that relationship. There can be a large number of these guards, and guards can be based on other guards.

These guards are not used anywhere today.

Solution space aka solving *“How do we efficiently cache, and intelligently guard code, including shaped based control flow, in the PyTorch compiler stack?”*

There are a couple of solutions available to us, but the primary one we plan to pursue is around:

1) Utilizing the fake tensors created in TorchDynamo for the purposes of tracing to do shape expression capture and evaluation. This means:

a) That our frontend will be symbolic shape aware,
b) Our frontend will create a shape_env. which it will subsequently pass down to our backends.

2) For the expressions captured in Dynamo in (1) we create new guards and compile them into check_fn (either in python, or in a c++ object a la guards.cpp).

These symbolic shape evaluations and guards will allow us to progress past control flow without sacrificing correctness. Today, Dynamo graph breaks on any control flow on shapes (when running in dynamic shape mode). Instead, we would usually prefer to have “symbolic specialization” on that control flow. In order to perform “symbolic specialization” on control flow, we must symbolically reason about shapes, and use the same symbolic shape system that AOTDispatch uses.

Note: Importantly, there may be cases where we have to raise an assert (like export) instead of graph breaking when we do not have sufficient information, for a user to provide a user driven constraint to help us keep tracing.

3) For expressions captured downstream of TorchDynamo, we have a few options before us:

3.option1) **No Backend Cache.** AOTAutograd forward pass just writes to the shape_env, which in turn bubbles up guards to TorchDynamo. AOTAutograd backward pass is either not cached at all, or we take an assumption that backwards pass cannot introduce new guards and is protected as is. TorchDynamo adds these and handles them exactly as it does its own expression guards in (1) and (2) above, albeit with an expanded infrastructure to match tensor to shape name expression (id<>s*)

Pros:

(a) A single cache. Since we don’t cache break very often on shape stuff today (as shown by us just running with static shapes and doing fine), this means less cache lookups, and simpler cache invalidation logic.

(b) Simpler

Cons:

(a) A cache miss means a total frame recompile.

(b) If the axiom of “an assumption that backwards pass cannot introduce new guards and is protected as is” is invalid, we will potentially need a backend cache anyway, so we didn’t really reduce that much complexity.

3.option2) **Full Backend Cache.** AOTAutograd forward pass and AOTAutograd backward pass have their own cache. We write expression guards for this cache only if they are not already present in the shape_env (presence in the shape_env means TorchDynamo already is caching and protecting on these guards). TorchDynamo does not know about these guards, but the backend is aware of the TorchDynamo guarded on expressions via shape_env, and installs only guards not already considered above it.

Pros:

(a) Smaller blast radius for an AOTAutograd only cache miss.

(b) Might need anyway if backwards pass can introduce new sources of expressions

(c) Code reuse between the forward and backward cache

Cons:

(a) More complex invalidation story

(b) Two caches means two cache lookups.

(c) Duplicate-ish cache infrastructure

3.option1 vs 3.option2:

We are going to proceed with a variation of 3option2: **Full Backend Cache.** This is due to the fact that (a) backwards in inductor can produce new guards and (b) multiple backwards calls:

out1 = f(arg1),
out2 = f(arg2)

and then call

out1.backward() and out2.backward()

Will require it.

There is still an open question of bubbling or not bubbling up the forward guards from AOTAutograd into the Dynamo guards, which we see as an optimization as long as the entire layer hierarchy described below can understand shape expressions or bubble up to a layer that does:

1) Dynamo level guards
2a.) AOTAutograd/PrimTorch/Ref guards for forwards pass
2b.) inductor guards for forwards graph
3a.) AOTAutograd/PrimTorch/Ref guards for backwards pass
3b.) Inductor guards for backwards graph.
4a). Eventually, plausibly, double backwards, triple backwards, etc.