The dynamic shapes manual

Glossary

Error cookbook
- RuntimeError: Cannot call sizes() on tensor with symbolic sizes/strides
- NotImplementedError: could not find kernel for aten.func.overload at dispatch key DispatchKey.Meta
- Expected a value of type 'int' for argument ‘x’ but instead found type 'SymInt'.
- torch._subclasses.fake_tensor.DynamicOutputShapeException: aten.func.overload

Understanding C++ exceptions
- Finding the important part of an exception

Typical situations
- How to SymInt’ify an operator schema
- How to SymInt’ify C++ operator
- How to write a C++ meta function
- How to write a Python meta function
- How to write a Python decomposition for non-composite operator
- How to write a Python decomposition for a composite operator (CompositeImplicitAutograd)

Handling data-dependent shapes

Handling Dynamo problems

More obscure situations
- How to SymInt’ify a TensorImpl method
- How to SymInt’ify a C++ utility function (non-SymInt)
- How to add a new SymInt operation
- How to add a new type of symbolic quantity (e.g., SymComplex)

How things work under the hood

Topics to cover wishlist

Glossary

- Specialize - given a tensor whose dimension is dynamic (may vary across runs), specializing that dimension means we assume that the size is fixed and may no longer vary

- Guard - whenever we specialize, we must ensure that the size does not actually vary in the future. A guard is a conditional expression we test before running a model to ensure that this invariant is upheld
- SymInt - short for symbolic integer, this represents an abstract integer that may vary and we want to trace symbolically (as opposed to a normal int)

Error cookbook

RuntimeError: Cannot call sizes() on tensor with symbolic sizes/strides

Typically, this means there is a C++ composite operator that isn't written to support symbolic shapes (specifically, it is calling tensor.sizes() when it should call tensor.sym_sizes()). However, the sizes() call doesn't have to occur inside an operator; it can also occur in framework code in PyTorch. The general recipe for dealing with this situation is to:

1. Find what C++ is making the failing call
2. Make the C++ support symbolic shapes, either by How to SymInt'ify C++ operator or possibly How to write a Python composite (CompositeImplicitAutograd) if the C++ code in question is an operator.

For step 1, there are a few strategies you can use:

- You can use gdb to find the C++ call stack. This can give useful insights even if you didn't compile with debug symbols (in fact, Edward prefers it this way, as gdb runs quite a bit faster without debug symbols.)
    - Open source: Run gdb --args python repro.py, catch throw, run. If the first exception thrown doesn't look related, use the ignore trick: https://fb.workplace.com/groups/1144215345733672/?multi_permalinks=2333644346790760&hoisted_section_header_type=recently_seen
        - Throws of TypeError typically are spurious and should be ignored
    - fbcode: As above, but follow the instructions at https://www.internalfb.com/intern/wiki/Python/Debugging/Debugging_with_GDB/#debugging-python-unit-te for how to actually run your PAR file with gdb
    - Remember that on an optimized build many functions will have gotten inlined away, but you should still be able to see the dispatcher calls in your stack frame
    - You can also manually induce a segfault with `import ctypes; ctypes.string_at(0)`
- You can use torch dispatch tracing to see what operators were called before the error. With a DEBUG=1 build of PyTorch (@mode/dev-nosan in fbcode), run your program with TORCH_SHOW_DISPATCH_TRACE=1 environment variable

- You can use Python dispatcher to see what operators were called before your error. This is a more complicated workflow, but a notable benefit over the previous bullet is you don't have compile with DEBUG=1 for this to be available.
    - Turn on python dispatcher: torch._dispatch.python.enable_python_dispatcher() context manager (to apply it globally just manually call .__enter__() on it).
    - Uncomment # print(self, key, final_key) in torch/_ops.py

For step 2, there are a few rules of thumb for making the decision:

- If the C++ call site is relatively simple, just update the C++ (e.g., it's a one or two line PR)
- If the C++ call site is not in an operator, you don't have a choice: update the C++. (Most common examples are derivatives.yaml and manual backwards functions.)
- If the C++ call site is from TensorIterator, make a PrimTorch decomposition in torch/_refs using the preexisting elementwise/reduction helpers
- If the C++ call site is complicated and a composite function, you may consider rewriting the decomposition in Python instead of refactoring the C++ to permit SymInts. Refer to How to write a Python composite (CompositeImplicitAutograd) or How to write a Python decomposition for non-composite operator depending on if it has an explicit autograd formula (you can check by seeing if it is registered as CompositeImplicitAutograd or CompositeExplicitAutograd in derivatives.yaml)

NotImplementedError: could not find kernel for aten.func.overload at dispatch key DispatchKey.Meta

This means we don't have a meta function for the operator. This is pretty simple go to How to write a Python meta function

Expected a value of type 'int' for argument 'x' but instead found type 'SymInt'.

This means that the operator doesn't take SymInts as arguments: it's schema says it only supports ints! Go to How to SymInt'ify an operator schema

torch._subclasses.fake_tensor.DynamicOutputShapeException: aten.func.overload

This means that you've hit an operator that has data-dependent output size, e.g., nonzero. Go to Handling data-dependent shapes

Understanding C++ exceptions

PyTorch's C++ exceptions can be difficult to understand. There are two reasons for this:

1. Exceptions often get rethrown multiple times (e.g, a c10::Error gets rethrown as python_error to propagate itself as a Python error across Python-C++ boundaries), so it is not always obvious if you are looking at the right stack.
2. PyTorch throws exceptions as part of its normal operation, so 'catch throw' may give you the wrong exception.
3. The exceptions are quite long and very metaprogrammed.

Finding the important part of an exception

In these examples, I've highlighted the important part of the exception.

In this backtrace, we've hit a C++ meta function that calls into TensorIterator which doesn't support symbolic sizes. See also https://github.com/pytorch/pytorch/pull/92914

```
#0  __cxxabiv1::__cxa_throw (obj=0x376b6960, tinfo=0x7fffa705b220 <typeinfo for
c10::Error>, dest=0x7ffff73834f0 <c10::Error::~Error()>)
    at
/opt/conda/conda-bld/gcc-compiler_1654084175708/work/gcc/libstdc++-v3/libsupc++
/eh_throw.cc:77
#1  0x00007ffff737e941 in c10::detail::torchCheckFail(char const*, char const*,
unsigned int, char const*) [clone .cold.75] ()
   from /data/users/ezyang/b/pytorch/torch/lib/libc10.so
#2  0x00007ffff739258f in c10::TensorImpl::sizes_custom() const () from
/data/users/ezyang/b/pytorch/torch/lib/libc10.so
#3  0x00007fff9d697576 in
at::TensorIteratorBase::compute_shape(at::TensorIteratorConfig const&) ()
   from /data/users/ezyang/b/pytorch/torch/lib/libtorch_cpu.so
#4  0x00007fff9d698439 in
at::TensorIteratorBase::build(at::TensorIteratorConfig&) ()
   from /data/users/ezyang/b/pytorch/torch/lib/libtorch_cpu.so
#5  0x00007fff9d699790 in
at::TensorIteratorBase::build_borrowing_unary_float_op(at::TensorBase const&,
at::TensorBase const&) ()
   from /data/users/ezyang/b/pytorch/torch/lib/libtorch_cpu.so
#6  0x00007fff9ec6cf09 in at::(anonymous
namespace)::wrapper_Meta_polygamma(long, at::Tensor const&) ()
   from /data/users/ezyang/b/pytorch/torch/lib/libtorch_cpu.so
#7  0x00007fff9ecb55f9 in
c10::impl::make_boxed_from_unboxed_functor<c10::impl::detail::WrapFunctionIntoF
```

```
unctor_<c10::CompileTimeFunctionPointer<at::Tensor (long, at::Tensor const&),
&at::(anonymous namespace)::wrapper_Meta_polygamma>, at::Tensor,
c10::guts::typelist::typelist<long, at::Tensor const&> >,
false>::call(c10::OperatorKernel*, c10::OperatorHandle const&,
c10::DispatchKeySet, std::vector<c10::IValue, std::allocator<c10::IValue> >*)
() from /data/users/ezyang/b/pytorch/torch/lib/libtorch_cpu.so
#8  0x00007fffa6877dc8 in (anonymous
namespace)::ConcretePyInterpreterVTable::python_dispatcher(c10::OperatorHandle
const&, c10::DispatchKeySet, std::vector<c10::IValue,
std::allocator<c10::IValue> >*) const () from
/data/users/ezyang/b/pytorch/torch/lib/libtorch_python.so
...
```

Sometimes, it is not a good idea to stop looking at the backtrace when you start seeing Python frames. There may be more C++ frames after the Python frames: this will occur if you have a Python->C++->Python->C++ transition.

```
#0  __cxxabiv1::__cxa_throw (obj=0x36f47c00, tinfo=0x7fffa705ad28 <typeinfo
for python_error>,
    dest=0x7fffa66801c0 <python_error::~python_error()>)
    at
/opt/conda/conda-bld/gcc-compiler_1654084175708/work/gcc/libstdc++-v3/libsu
pc++/eh_throw.cc:77
...
#10 0x00007fffa65adbb3 in pybind11::cpp_function::dispatcher(_object*,
_object*, _object*) ()
   from /data/users/ezyang/b/pytorch/torch/lib/libtorch_python.so
#11 0x0000000000507357 in cfunction_call (func=0x7fff25996400,
args=<optimized out>, kwargs=<optimized out>)
    at /usr/local/src/conda/python-3.9.16/Objects/methodobject.c:543
...
#47 0x000000000059489d in PyObject_CallMethod (obj=<optimized out>,
name=<optimized out>, format=0x7fffa6eba02d "OOOO")
    at /usr/local/src/conda/python-3.9.16/Objects/call.c:635
#48 0x00007fffa6bf0c73 in
torch::handle_torch_function_no_python_arg_parser(c10::ArrayRef<pybind11::h
andle>, _object*, _object*, char const*, _object*, char const*,
torch::TorchFunctionName) () from
/data/users/ezyang/b/pytorch/torch/lib/libtorch_python.so
...
#73 0x00007fffa67a79eb in
torch::autograd::THPVariable_special_polygamma(_object*, _object*,
```

```
_object*) ()
   from /data/users/ezyang/b/pytorch/torch/lib/libtorch_python.so
#74 0x0000000000507357 in cfunction_call (func=0x7fffbd4b12c0,
args=<optimized out>, kwargs=<optimized out>)
    at /usr/local/src/conda/python-3.9.16/Objects/methodobject.c:543
...
```

Typical situations

How to SymInt'ify an operator schema

Given an operator schema which takes int/int[] arguments, change the type to SymInt/SymInt[] so that it accepts SymInt.

Before:

```
- func: tensor_split.sections(Tensor(a -> *) self, int sections, int dim=0) ->
Tensor(a)[]
```

After:

```
- func: tensor_split.sections(Tensor(a -> *) self, SymInt sections, int dim=0)
-> Tensor(a)[]
```

There may be some side effects of the schema change, which should become clear when you attempt to build PyTorch / run CI. However, this are for edge cases, and we believe we have already ported all schemas that hit these cases. Things to watch out for:

- If it's a view op, FunctionalInverses.cpp may need updating
- If the op is supported by LTC, you may need to update LTC's bindings
- If it is supported by functorch, you may need to update the functorch batching rule
- If the kernel has Metal support, you may need to update this registration, e.g., https://www.internalfb.com/diff/D39084762 (fortunately, OSS CI will catch this)

One thing you do not have to do: SymInt'ifying a signature is backwards compatible as far as TorchScript is concerned (under the hood, we translate SymInt back into int for interaction with TorchScript and other legacy systems.) In particular, it is NOT strictly necessary to change any of the C++ kernels (although you probably want to know How to write a Python meta function; or

if there is a preexisting CompositeImplicitAutograd and Meta, either How to SymInt'ify C++ operator or How to write a Python composite (CompositeImplicitAutograd)).

Notes:

- More obscure types that are supported: SymInt? (translates into c10::optional<c10::SymInt>), SymInt[]? (translates into at::OptionalSymIntArrayRef). The canonical list of supported C++ SymInt operator types can be found in the has_symint template in aten/src/ATen/core/boxing/KernelFunction.h. If you need support for another C++ type, you must add support for it; see https://github.com/pytorch/pytorch/pull/86088
- If an operator accepts Scalar, it already automatically accepts SymInt/SymFloat/etc, no schema modifications necessary.
- Functions that return SymInt are special; see https://github.com/pytorch/pytorch/pull/92402
- If Executorch already has created an out-version of the operator, the Executorch schema needs to be updated too. See https://github.com/pytorch/pytorch/pull/87604

How to SymInt'ify C++ operator

You only need to SymInt'ify a C++ operator if it is CompositeImplicitAutograd, CompositeExplicitAutograd or Meta; it is not useful to SymInt'ify a CPU/CUDA kernel as it is expected that if you are going to do the actual CPU/CUDA compute, you always will have concrete (non-symbolic) integers.

native_functions.yaml changes: A C++ operator will be referenced in a dispatch table in native_functions.yaml. To indicate that the C++ implementation can directly accept SymInts, you must suffix the name of the kernel with _symint. If you are creating an out-of-tree custom operator registration, you will instead change the schema string in the m.def(...) call in your TORCH_LIBRARY block.

Before:

```
- func: empty.memory_format(SymInt[] size, *, ScalarType? dtype=None,
Layout? layout=None, Device? device=None, bool? pin_memory=None,
MemoryFormat? memory_format=None) -> Tensor
  dispatch:
    Meta: empty_meta
```

After:

```
- func: empty.memory_format(SymInt[] size, *, ScalarType? dtype=None,
Layout? layout=None, Device? device=None, bool? pin_memory=None,
MemoryFormat? memory_format=None) -> Tensor
  dispatch:
    Meta: empty_meta_symint
```

If the schema has no dispatch table **AND IT DOESN'T HAVE A structured_delegate entry**, that is equivalent to it having a dispatch table with a single entry CompositeImplicitAutograd: func_name. To SymIntify it, you have to explicitly add the previously implicit dispatch table and then add the suffix.

Before:

```
- func: broadcast_to(Tensor(a) self, SymInt[] size) -> Tensor(a)
  variants: function, method
```

After:

```
- func: broadcast_to(Tensor(a) self, SymInt[] size) -> Tensor(a)
  variants: function, method
  dispatch:
    CompositeImplicitAutograd: broadcast_to_symint
```

If the schema has a structured_delegate entry, that means it is a structured kernel. We do not support directly SymInt'ifying C++ kernels that are structured; you should do the Python meta instructions instead.

C++ changes: Anywhere the C++ operator previously handled a int64_t/at::IntArrayRef at a SymInt/SymInt[] function schema type, it now must handle a c10::SymInt/at::SymIntArrayRef instead. The sections below give common situations; however, it's also a good idea to grep for SymInt in aten/src/ATen which will give you ideas about idioms how the code is currently working.

The prototype for the native function is different.

Before:

```
Tensor broadcast_to(const Tensor& self, IntArrayRef size) {
```

After:

```
Tensor broadcast_to_symint(const Tensor & self, SymIntArrayRef size) {
```

If you are an out-of-tree custom function, just directly change the signature of your function; the operator registration API will automatically detect that you are providing a SymInt aware function.

If you would retrieve sizes/strides/storage_offset/numel from a Tensor, you must use the sym_ prefixed variant of the method.

Before:

```
int64_t batch = self.size(0);
```

After:

```
c10::SymInt batch = self.sym_size(0);
```

If you would call a function/method with a SymInt/SymIntArrayRef/... C++ type, you must use the _symint prefixed variant of the function/method.

Before:

```
  return self.expand(size);
```

After:

```
  return self.expand_symint(size);
```

If you define an integer literal / integer initializer list in a context where it needs to be sent to a function/method expecting a SymInt, you must explicitly specify its type.

Before:

```
  auto values = grad.reshape({-1, num_features});
```

After:

```
  auto values = grad.reshape_symint({c10::SymInt(-1),
std::move(num_features)});
```

If you have a SymInt and you need an int64_t (because you need to call a function that isn't SymInt'ified yet, or there is some fundamental reason we cannot treat size symbolically--e.g., the size will be used to provide the size of an output list or the dimension of a tensor), call guard_int(__FILE__, __LINE__) to induce a guard on the integer size, eliminating dynamic support for that particular shape.

Before:

```
  std::vector<Tensor> splits(sections);
```

After:

```
  // NB: intentional, sections specifies number of output tensors, which
  // cannot be polymorphic
  int64_t sections = sym_sections.guard_int(__FILE__, __LINE__);
  std::vector<Tensor> splits(sections);
```

If you are modifying a performance critical function, you may want to create both an int64_t and a SymInt specialization of the function and dispatch them separately. In that case, use the at::symint:: namespace of function calls, which use a (mandatory) template argument to specify if they need SymInt or not.

Before:

```
std::vector<Tensor> tensor_split(const Tensor& self, IntArrayRef indices,
int64_t dim) {
```

```
  ...
  for (const auto split_idx : c10::irange(num_indices)) {
    int64_t end_idx = indices[split_idx];
    splits[split_idx] = at::slice(self, dim_, start_idx, end_idx);
    ...
```

After:

```
template <typename T>
std::vector<Tensor> _tensor_split_indices(const Tensor& self, ArrayRef<T>
indices, int64_t dim) {
  ...
  for (const auto split_idx : c10::irange(num_indices)) {
    auto end_idx = indices[split_idx];
    splits[split_idx] = at::symint::slice<T>(self, dim_, start_idx,
end_idx);
    ...
```

In similar spirit, we also have at::symint::size<T>, sizes<T>, stride<T>, etc to support size/stride queries in templated code like this.

How to write a C++ meta function

To support dynamic shapes, an operator must have a SymInt aware meta function that can propagate symbolic shapes through it. In C++, what you need to do is similar to How to SymInt'ify C++ operator. However, we don't really recommend this for in-tree operators; it's usually easier and quicker to write a Python meta function instead.

In-tree: Add a Meta entry to the dispatch table, give the function name a _symint suffix, and then implement the function as you would a normal meta function, following the C++ rules at How to SymInt'ify C++ operator. A meta function looks like a normal operator implementation, except all you do is allocate the outputs and return it; you do not do any compute (if you have a preexisting C++ operator implementation, it is usually fairly easy to make a meta version: copy paste the old implementation, and then delete all the code that actually does data manipulation.)

You should seriously consider writing a Python meta function instead; you skip recompile cycles and the resulting meta is easier to debug. The only exception is for extremely low level operations (like torch.empty), where there are not enough public Python APIs to actually write the Meta function in Python.

Out-of-tree (e.g., custom operators): Write a meta function as per How to SymInt'ify C++ operator and then register it at the dispatch key Meta. C++ meta function can be convenient for custom operators which aren't a logical place to place a Python meta registration (as the Python meta registration must be somehow imported to be made available.)

Example:

```
TORCH_LIBRARY_IMPL(fbgemm, Meta, m) {
  m.impl(
      "jagged_to_padded_dense_forward",
      TORCH_FN(fbgemm_gpu::jagged_to_padded_dense_forward_meta));
}
```

How to write a Python meta function

To support dynamic shapes, you need a meta function for an operator. Python meta function is the **recommended** way to add symbolic shape support (to give a sense for how much better Python meta functions are than the other options, after we added support for Python meta functions, our symbolic shape productivity tripled.) If you are not in a rush, writing a Python decomposition for non-composite operator subsumes writing a meta function (and has the added benefit of producing a compiler lowering for the function), but decomps are considerably more complicated so there is not much wasted work in writing the (typically simple) meta function first, and upgrading it later if necessary.

In-tree: Add your meta function to torch/_meta_registrations.py. There are plenty of examples in this file, but there are a few important points:

- The meta function must be registered with @register_meta decorator (which takes aten.funcname.overloadname as argument).
- You cannot register a meta function if an operator is CompositeImplicitAutograd. You will get an error message if you try to do this; make sure to check on this *before* you start writing the meta function.
- Consider using @out_wrapper() decorator so that your function can also have out= support, and then you can register the implementation for both the functional and out variant; e.g., @register_meta([aten._fft_c2c.default, aten._fft_c2c.out])
- Use check(...) to test for input validity
- Unlike C++ SymInt support, you can generally write Python meta functions for symbolic shapes the same way you write meta functions for regular shapes, as Python is duck-typed and int/SymInt can generally be treated interchangeably. (One major caveat are Python builtins like int()/float() which are not overloadable. You should get a clear error message if you mess this up.)

Out-of-tree: The main difference is you don't use the register_meta decorator; instead, use Python operator registration API.

```
from torch.library import impl_abstract

@impl_abstract("fbgemm::jagged_to_padded_dense_forward")
def jagged_to_padded_dense_forward_meta(values, offsets, max_lengths,
padding_value=0):
    return torch.empty(...)
```

Note that you must actually import the Python file that defines this Python registration for it to be available. If there is not an obvious Python module to trigger this import, consider looking at How to write a C++ meta function instead.

Please see the docstring or The C++ Custom Operators Manual for more details on how to use this API.

TODO: pt2_compliant c++ tag

How to write a Python decomposition for non-composite operator

A Python decomposition for a non-composite operator is an alternative to writing a meta function. The added benefit of writing a decomposition is that it can be helpful for compilers: the decomposition can be used not only for dynamic shape propagation, but also to add compilation support for the operator if the operations it decomposes to are supported by the compiler. It can be quite a lot more difficult to write a Python decomposition though, so no harm if you write a Python reference instead! Be careful to distinguish this from How to write a Python decomposition for a composite operator (CompositeImplicitAutograd), where typically the C++ implementation is already a composite and it's mostly a question if you want it to be in Python or C++ (and also, the recipe described here WILL NOT WORK; your decomposition will be silently ignored if you use the API here).

There are two places where these decompositions can live: torch/_decomp or torch/_ref. The rule for which directory to place it in, is if the operator is directly exposed in the Python torch.* API with exactly the same API as the ATen operator, place it in torch/_ref; otherwise, put it in torch/_decomp (which is for all the "internal" ATen operators that aren't publicly exposed.) In other words, torch/_refs is intended to be a replication of the public Python torch API, so if you need to write an operator that isn't in the public Python API, you can't put it there.

In-tree for torch/_decomp (ATen decomposition): Add your decomposition to torch/_decomp/decompositions.py. There are plenty of examples in this file; the main points:

- Use @register_decomposition to register the function as a decomposition
- It's easy to forget handling type promotion behavior. There are some helper decorators like pw_cast_for_opmath
- See also How to write a Python meta function for more tips about writing operator implementations in Python

In-tree for torch/_refs (Torch decomposition): If the public function is available at torch.foo module, then place your decomposition in torch/_ref/foo.py . Feel free to write your decomposition in the same style as torch/_decomp (in particular, you should feel free to call other torch.* functions); however, in torch/_ref you also have the option to decompose to PrimTorch prims. Some backends (e.g., Inductor) prefer decomposition to ATen rather than Prims, so if you can write your decomposition going entirely to torch.*, prefer that. There are also plenty of examples in this directory. You still must use the @register_decomposition decorator to register the function as a decomposition.

Out-of-tree: This is currently un-tested, but it should work to just use @register_decomposition decorator out of tree.

Implementation note: in-tree decompositions don't register to CompositeExplicitAutograd dispatch key. Maybe they should, but in eager mode we prefer the C++ implementation if it exists (this is different from Meta, where we made the call that we should always prefer the Python implementation for consistency, since Meta invocations are not performance critical.)

How to write a Python decomposition for a composite operator (CompositeImplicitAutograd)

CompositeImplicitAutograd decompositions are special because they cannot be overridden in the 'normal' way; e.g., if you use @register_decomposition, it will not do anything. This is because autograd decompositions happen before we apply other decompositions/meta functions, so a layer like make_fx never actually sees the operator before it is decomposed. If you have chosen to reimplement these operators in Python (as opposed to modifying the C++ implementation aka per How to SymInt'ify C++ operator).

The place to put the operator (torch/_decomp or torch/_refs) is the same choice per How to write a Python decomposition for non-composite operator.

To register a composite, use py_impl method on the operator:

```
@aten.matmul.default.py_impl(DispatchKey.CompositeImplicitAutograd)
```

```
def matmul(tensor1, tensor2):
    ...
```

The internals of this function are the same as non-composite Python decompositions, but every operator you call must be differentiable (this typically excludes prims from PrimTorch).

Warning: this WILL NOT work if the operator in question is not actually composite (e.g., if it is a regular operator that has an autograd formula).

Handling data-dependent shapes

See Dealing with GuardOnDataDependentSymNode errors

Handling guards on unbacked SymInts

An unbacked SymInt is a SymInt which cannot be guarded on, either because we do not know what the real value of it is (because it's data dependent) or because we are enforcing that a dimension must be dynamic and rejecting guards on it. When working with unbacked SymInts, you will frequently run into code that *should* be guard-free, but actually isn't, resulting in this error:

> GuardOnDataDependentSymNode: It appears that you're trying to get a value out of symbolic int/float whose value is data-dependent (and thus we do not know the true value.) The expression we were trying to evaluate is i0 >= 16. Scroll up to see where each of these data-dependent accesses originally occurred.

There are a few different strategies for handling this sort of situation. In general, however, removing guards from PyTorch library code requires some understanding what semantically the original code did, and whether or not you can remove the guard in a semantics preserving way. The strictest form of semantics preservation is to match the original behavior exactly, but it may also be acceptable to remove fast paths (given that the slow path has the same semantics, and the fast path existed purely for eager PyTorch) or slightly change the striding behavior of a function (given that changing the stride doesn't affect downstream semantics, which is *usually* true and will be formally true with stride agnostic PyTorch).

Remove a fast path

Sometimes the guard is coming from special case logic in PyTorch, which is testing if it can run some fast path, but the slow path is exactly equivalent. Avoiding the fast path check when there are unbacked SymInts can save you a guard.

Example: https://github.com/pytorch/pytorch/pull/94399 Avoid guarding zeroness on meta tensors

Before:

```
  // Default TensorImpl has size [0]
  if (size.size() != 1 || size[0] != 0) {
tensor.unsafeGetTensorImpl()->generic_set_sizes_contiguous(size);
  }
```

After:

```
  // NB: test for meta dispatch key to avoid guarding on zero-ness
  if (ks.has(c10::DispatchKey::Meta) || size.size() != 1 || size[0] != 0) {
    tensor.unsafeGetTensorImpl()->generic_set_sizes_contiguous(size);
  }
```

Originally, the code had a fast-path to avoid generic_set_sizes_contiguous when the Tensor was already sized correctly. However, this logic results in a guard on size[0] != 0. It is always safe to generic_set_sizes_contiguous, so we can skip the guard check (or in this case, we keep it for eager mode but suppress it when symbolic shapes may be involved.)

Example: https://github.com/pytorch/pytorch/pull/95218 Disable matrix multiply folding

Before:

```
    elif should_fold(tensor1, dim_tensor2) or should_fold(tensor2,
dim_tensor1):
        # fast path code
```

After:

```
    # The folding logic ends up triggering a lot of guards in a hard to
    # understand way.  Suppress folding if you have unbacked SymInts.  It's
```

```
    # possible you can get the folding code to work with unbacked SymInts
    # but I couldn't figure out how to do it.
    elif (
        tensor_has_hints(tensor1)
        and tensor_has_hints(tensor2)
        and (should_fold(tensor1, dim_tensor2) or should_fold(tensor2,
dim_tensor1))
    ):
        # fast path code
```

Here we use the tensor_has_hints helper to test if the tensor has backed SymInts everywhere, and only go into the fast path if we do.

Add a fast path / short circuit

Conversely, sometimes code triggering guards is written in a very general way, and if you add a short circuit if the input is, e.g., contiguous, you can avoid the guards / guard on something that is in fact hinted.

Examples: https://github.com/pytorch/pytorch/pull/95216

```
Tensor reshape_symint(const Tensor& self, c10::SymIntArrayRef
proposed_shape) {
  if (self.is_sparse()) {
    AT_ERROR("reshape is not implemented for sparse tensors");
  }

  if (self.is_contiguous() && !self.is_mkldnn()) {
    return self.view_symint(proposed_shape);
  }
```

By testing if the tensor is contiguous, we can avoid performing a heavy correctness test in computeStride later in reshape.

NB: if is_contiguous() is unbacked, adding this special case can introduce more guards! So it is better to do this as definitely_true(), but that requires a sym_is_contiguous which doesn't exist yet.

Handle things at a higher level, making the guard is unnecessary

Examples:

- https://github.com/pytorch/pytorch/pull/95069 Introduce torch.empty_permuted and then https://github.com/pytorch/pytorch/pull/94523
- https://github.com/pytorch/pytorch/pull/95004 Remove unnecessary TensorMeta rewrap
- https://github.com/pytorch/pytorch/pull/95003 Hard code known true contiguity settings
- https://github.com/pytorch/pytorch/pull/94681 Don't use PrimTorch decomposition for empty

Use definitely_true to take advantage of symmetry, e.g., broadcasting

Example: https://github.com/pytorch/pytorch/pull/95217

Allow for differing stride behavior when unbacked

Example:

- https://github.com/pytorch/pytorch/pull/95218 the expand implementation
- https://github.com/pytorch/pytorch/pull/95222 the conv implementation

Handling Dynamo problems

TODO: I don't know how to write this section

More obscure situations

How to SymInt'ify a TensorImpl method

This is similar to the template section in How to SymInt'ify C++ operator but you only have TensorImpl methods available. The templated ones are typically prefixed generic_

How to SymInt'ify a C++ utility function (non-SymInt)

This is the same playbook as the templated case in How to SymInt'ify C++ operator. One thing to be careful about is that if you have a pre-existing C++ utility function, turning it into a template may cause overload resolution to fail (as a template does not induce implicit conversions.) The

typical workaround for this is to keep the old overloads, and have an internal template function that does the actual implementation.

Example:

```
template <typename T>
inline bool is_channels_last_strides_3d(
    const ArrayRef<T> sizes,
    const ArrayRef<T> strides) {
  switch (sizes.size()) {
    case 5:
      return is_channels_last_strides_3d_s5(sizes, strides);
    case 4:
      // TODO dim == 4 case will be enabled once it is fully tested
      return false;
    default:
      return false;
  }
}

inline bool is_channels_last_strides_3d(
    const IntArrayRef sizes,
    const IntArrayRef strides) {
  return is_channels_last_strides_3d<int64_t>(sizes, strides);
}
```

How to add a new SymInt operation

Grep for an existing SymInt operation to see what you need to update: at time of writing sym_float was a pretty good example. You'll expect to have to update:

- c10/core/SymInt.h, cpp
- c10/core/SymNodeImpl.h
- torch/csrc/jit/python/init.cpp
- torch/csrc/utils/python_symnode.h
- torch/fx/experimental/symbolic_shapes.py

How to add a new type of symbolic quantity (e.g., SymComplex)

Look at https://github.com/pytorch/pytorch/pull/92149

How things work under the hood

UNDER CONSTRUCTION

The way to think about the underlying implementation of our system is that we have a number of extension points in PyTorch, and the various ways of adding dynamic shape support are using the various extension points. These extension points also get run “in an order”; e.g., outer extension points apply before inner extension points.

Going outside-in (higher priority extension points come first):

1. TorchFunctionMode. We do not use this in our current stack.
2. Autograd$BACKEND key in Python dispatcher
    a. This can be
3. Meta key in C++ dispatcher (How to write a C++ meta function).
4. Just directly modify PyTorch C++ code (How to SymInt’ify C++ operator). This is not a “real” extension point, but this is a valid way to add symbolic support and is the starting point.

Topics to cover wishlist

- (add here)
- Which systems use the python global decomp table? (probably covered by how things work under the hood)
- What’s the flow for the system (how do you know what is going to get hit)