Adapted from: http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/

Why care about a dispatcher?

- PyTorch has a lot of systems: autograd, tracing, vmap
  - and a lot of backend devices: XLA, CUDA, CPU, ...
- We could write a single at::add function that handles all of the above
  - It would probably have a big fat switch statement with a lot of code in it.
  - Think about packing the VariableType (autogenerated) code, the CUDA code, and the CPU code all into one function!

What is the dispatcher?

- Each operator has a *dispatch table*, a table of function pointers for each key.
- The dispatch keys are sorted by *priority*
- When you call an operator, the dispatcher looks at the current set of DispatchKeys to figure out which function pointer to call.

Each Tensor has a DispatchKeySet

- To figure out which function pointer to call, we:
  - Union all the dispatch keys in the Tensor
  - Union some global dispatch keys (just BackendSelect*)
  - Union a set of "Local Include" keys. These are usually set in a thread-local way
  - Remove a set of "Local Exclude keys". This is usually set in a thread-local way
- The code for that lives here.
- Once we have our final key set, we pick the first dispatch key.

Let's go through an example of what happens when we add two Tensors together.

```
x = torch.randn(3, device='cuda')
y = torch.randn(1, device='cuda')
torch.add(x, y)

# Tensor dispatch keys:
# x has the AutogradCUDA and CUDA dispatch key
# y has the AutogradCUDA and CUDA dispatch key.

# Global dispatch keys:
# [BackendSelect]

# Local include: []
```

```
# Local exclude: []

# The final set ordered by priority, is [AutogradCUDA, BackendSelect, CUDA]
```

Ultimately, what happens is that we will make it to `native::add`. native::add is registered for at::add on the CPU and CUDA keys

So here's what happens:

- at::add(x, y) invokes the dispatcher, which combines the dispatch keys into a DispatchKeySet as described above. In this scenario, the highest priority key is the `AutogradCUDA` key.
  - The file that this function lives in is actually codegen'd, so if you want to view it in source you'll need to build pytorch. Then you can view it at `build/aten/src/ATen/Functions.h`.
- at::add(x, y) dispatches to the Autograd implementation of add. That's the below function.
  - Don't worry too much about the `Autograd` vs. `AutogradCUDA` distinction; in 99% of cases you can treat them as identical. If you're curious though, `Autograd` is an alias dispatch key.
  - This function is also codegen'd - after building, you can view it at `torch/csrc/autograd/generated/VariableTypeEverything.cpp`.

```
// In VariableTypeEverything.cpp

  Tensor add_Tensor(const Tensor & self, const Tensor & other, Scalar alpha) {
    auto& self_ = unpack(self, "self", 0);
    auto& other_ = unpack(other, "other", 1);
    std::shared_ptr<AddBackward0> grad_fn;
    if (compute_requires_grad( self, other )) {
      grad_fn = std::shared_ptr<AddBackward0>(new AddBackward0(), deleteNode);
      grad_fn->set_next_edges(collect_next_edges( self, other ));
      grad_fn->alpha = alpha;
    }
    #ifndef NDEBUG
    c10::optional<Storage> self__storage_saved =
      self_.has_storage() ? c10::optional<Storage>(self_.storage()) : c10::nullopt;
    c10::intrusive_ptr<TensorImpl> self__impl_saved;
    if (self_.defined()) self__impl_saved = self_.getIntrusivePtr();
    c10::optional<Storage> other__storage_saved =
      other_.has_storage() ? c10::optional<Storage>(other_.storage()) : c10::nullopt;
    c10::intrusive_ptr<TensorImpl> other__impl_saved;
    if (other_.defined()) other__impl_saved = other_.getIntrusivePtr();
    #endif
    auto tmp = ([&]() {
      at::AutoNonVariableTypeMode non_var_type_mode(true);
      return at::add(self_, other_, alpha);
    })();
```

```
  auto result = std::move(tmp);
  #ifndef NDEBUG
  if (self__storage_saved.has_value())
    AT_ASSERT(self__storage_saved.value().is_alias_of(self_.storage()));
  if (self__impl_saved) AT_ASSERT(self__impl_saved == self_.getIntrusivePtr());
  if (other__storage_saved.has_value())
    AT_ASSERT(other__storage_saved.value().is_alias_of(other_.storage()));
  if (other__impl_saved) AT_ASSERT(other__impl_saved == other_.getIntrusivePtr());
  #endif
  if (grad_fn) {
      set_history(flatten_tensor_args( result ), grad_fn);
  }
  return result;
}
```

- None of the tensors require grad, so none of the autograd specific logic actually happens
  - fun fact: we're planning on changing this behavior in the near-to-mid future! Eventually, we'd like it if the autograd kernel doesn't ever get called unless the input tensors actually require gradients (specifying `requires_grad=True`)

```
auto tmp = ([&]() {
  at::AutoNonVariableTypeMode non_var_type_mode(true);
  return at::add(self_, other_, alpha);
})();
```

- Now, the above code: creates a Local Exclude set of [Autograd]
- Inside at::add(self_, other_, alpha);, the computation happens again:

```
# Tensor dispatch keys:
# x has the AutogradCUDA and CUDA dispatch key
# y has the AutogradCUDA and CUDA dispatch key.

# Global include dispatch keys:
# BackendSelect

# Local include: []
# Local exclude: [AutogradCPU, AutogradCUDA, AutogradXLA]

# The final set is [BackendSelect, CUDA]
```

- Cool! So now the dispatcher looks up BackendSelect's implementation for add.
- Checking `build/aten/src/ATen/BackendSelectRegister.cpp`, there is no BackendSelect implementation for add. vBackendSelect's add implementation is a special *fallback kernel*that says "there is nothing here, instead, hop over to the next dispatch key". More on fallback kernels later.
  - In fact, the Dispatcher actually has an optimization to avoid calling the fallthrough kernel at all. It figures out which kernels are "fallthrough"

kernels at static initialization time, and adds them to a bitset mask to skip them entirely. If you're curious, the logic for that lives here.

```
TORCH_LIBRARY_IMPL(_, BackendSelect, m) {
  m.fallback(torch::CppFunction::makeFallthrough());
}
```

- So the dispatcher goes and picks the next key down the list, which is CUDA. We now invoke at::native::add. We've reached the end!

Example number 2: factory function

```
x = torch.randn(3, 3, device='cuda')

# Upon calling at::randn, the dispatch keys are:
# Global include set: [BackendSelect]
# Local include set: []
# Local exclude set: []

# So we select the BackendSelect version of randn.
```

- Factory functions like randn are treated specially, and do not get Fallthrough kernels registered to the dispatcher. Again, you can see the kernel for randn that's registered to the `BackendSelect` key in `build/aten/src/ATen/BackendSelectRegister.cpp`.
- BackendSelect version of randn:

```
C10_ALWAYS_INLINE
at::Tensor randn(at::IntArrayRef size, c10::optional<at::ScalarType> dtype, c10::opt
  static auto op = c10::Dispatcher::singleton()
    .findSchemaOrThrow("aten::randn", "")
    .typed<at::Tensor (at::IntArrayRef, c10::optional<at::ScalarType>, c10::optional
  DispatchKeySet _dk = c10::DispatchKeySet(c10::computeDispatchKey(dtype, layout, de
  return op.redispatch(_dk, size, dtype, layout, device, pin_memory);
}
```

- It computes a dispatch key based on the dtype, layout, and device. In our case, the computed dispatch key is CUDA, so it straight up calls native::randn: https://github.com/pytorch/pytorch/blob/2c554266108f1b556dd49f7c3c06c08f2bbd3cbe/aten/ L623

```
Tensor randn(IntArrayRef size, c10::optional<Generator> generator, const TensorOptions&
  auto result = at::empty(size, options);
  return result.normal_(0, 1, generator);
}
```

- But we're not done! at::empty got invoked. at::empty also invokesthe CUDA version

```
- func: empty.memory_format(int[] size, *, ScalarType? dtype=None, Layout? layout=None
  #use_c10_dispatcher: full
```

```
dispatch:
  CPU: empty_cpu
  CUDA: empty_cuda
```

- Great. `result` is a Tensor with dispatch keys [AutogradCUDA, CUDA].
- `result.normal_(...)` dispatches to the Autograd implementation of normal_

```
// See VariableTypeEverything.cpp
Tensor & normal_(Tensor & self, double mean, double std, c10::optional<Generator> genera
  auto& self_ = unpack(self, "self", 0);
  check_inplace(self);
  std::shared_ptr<NormalBackward0> grad_fn;
  if (compute_requires_grad( self )) {
    grad_fn = std::shared_ptr<NormalBackward0>(new NormalBackward0(), deleteNode);
    grad_fn->set_next_edges(collect_next_edges( self ));
  }
  #ifndef NDEBUG
  c10::optional<Storage> self__storage_saved =
    self_.has_storage() ? c10::optional<Storage>(self_.storage()) : c10::nullopt;
  c10::intrusive_ptr<TensorImpl> self__impl_saved;
  if (self_.defined()) self__impl_saved = self_.getIntrusivePtr();
  #endif
  {
    at::AutoNonVariableTypeMode non_var_type_mode(true);
    self_.normal_(mean, std, generator);
  }
  #ifndef NDEBUG
  if (self__storage_saved.has_value())
    AT_ASSERT(self__storage_saved.value().is_alias_of(self_.storage()));
  if (self__impl_saved) AT_ASSERT(self__impl_saved == self_.getIntrusivePtr());
  #endif
  increment_version(self);
  if (grad_fn) {
      rebase_history(flatten_tensor_args( self ), grad_fn);
  }
  return self;
}
```

- Which then goes to

```
{
  at::AutoNonVariableTypeMode non_var_type_mode(true);
  self_.normal_(mean, std, generator);
}
```

- `self_` is a Tensor with dispatch keys [AutogradCUDA, CUDA]. The AutoNonVariableTypeMode adds a Local Exclude of [AutogradCUDA].

So the final set is [BackendSelect, CUDA] and the dispatcher selects the CUDA implementation of normal_

- Which straight up goes tonative::normal_

```
- func: normal_(Tensor(a!) self, float mean=0, float std=1, *, Generator? generato
variants: method
dispatch:
  CPU, CUDA: normal_
```

How do we populate the dispatch table?

The dispatcher has a registration API

- see "Operator Registration" in http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/
- Our codegen pipeline takes care of the work of calling the registration API, and registering all of our different kernels to most of the important dispatch keys:
  - CPU
  - CUDA
  - Autograd
  - BackendSelect
- The API includes the ability to define a Fallback kernel (that does nothing), which you saw used by BackendSelect

Boxing vs unboxing

Helpful resources:

- See "Unboxing" in http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/
- See this wiki page, which has a really useful diagram: https://github.com/pytorch/pytorch/wiki/Boxing-and-Unboxing-in-the-PyTorch-Operator-Library

Understanding boxed vs. unboxed representations

Unboxed representation

- Objects have a different layout depending on the data in question
- What you expect from C++: each struct is a different size depending on its type
- This is great for efficiency - your data is packed together tightly, only takes up as much space as it needs!

An unboxed data representation has a downside though: you can't write a single function that works over all of your different objects!

Well, you sort of can with templates in C++:

```cpp
template<typename T>
void foo(T obj) {...}
```

In the above, `void foo(T)` is a function template that you call with different types. But if I call `foo("a"); foo(123);`, the compiler generates and stamps out two completely different implementations of foo() - one that looks like `void foo(const char*)`, and another that looks like `void foo(int)`! Templates are handy for avoiding code duplication, but we still end up producing a new specialized function for every different type that's passed into the template.

Contrast that to a boxed representation:

Boxed representation

- Objects have a unified layout.
- In general: Different programming languages may choose to use a boxed layout by default for all of their types, e.g. Java.
- In PyTorch: We have our own boxed layout implemented in C++: Some of our APIs shove values into these things called IValuesand toss them onto a stack.
  - IValue is a union between Tensor, int64, float64, etc…

Having a boxed representation for types lets us write boxed functions in PyTorch:

- Boxed functions can be written once, and (if implemented correctly) work for all operators
- Boxed functions in PyTorch have a very specific schema:
  - `void my_boxed_f(const OperatorHandle& op, std::vector<c10::IValue> stack)`
  - Defined here: https://github.com/pytorch/pytorch/blob/8216da1f23b893c074e76e8a9aa7127efbda428
  - `OperatorHandle` is a class that represents an operator (e.g. `torch.add`, `torch.mul`), and the `std::vector<c10::IValue>` is a stack of IValues that are the inputs to that operator
- The general idea when writing a boxed function: Pop the inputs off of the stack, compute some output(s), and push them back onto the stack.

Example where boxing is used: Batched Fallback kernel.

- https://github.com/pytorch/pytorch/blob/2c554266108f1b556dd49f7c3c06c08f2bbd3cbe/aten/src/ATen/B
- void`` ``batchedTensorForLoopFallback``(``const`` c10::OperatorHandle& op, torch::jit::Stack* stack) {...}
- Essentially, all of the arguments are IValues on the stack, and we have a handle to an operator in the DispatchTable. A fallback tells us what to do to handle this
- What BatchedFallback does is the following:
  - For each sample in the batch, call op(sample)
  - For example:

```python
x = torch.randn(N, 3)
y = torch.randn(N, 3)
```

`torch.add(x, y)`

- There can be multiple tensors present in `stack`.
- BatchedFallback takes all the IValues, converts some to Tensor, slices them in the batch dimension as necessary, and calls `op` multiple times.
  - It ends up doing: `torch.stack([x[0] + y[0], x[1] + y[1], x[2] + y[2], ...)`

Benefits of boxing in PyTorch:

There are some benefits to writing batching logic as a boxed fallback like the above:

- Complexity decrease. This is kind of subjective, but arguably a single boxed kernel like the above is easier to maintain compared to the alternatives. If you want to write some functionality that works for every operator in PyTorch, some alternatives to writing a boxed fallback are:
  - Manually write 1000+ versions of your code, one for each operator (ouch)
  - Write fancy template metaprogramming logic to templatize over the operator and argument types (ouch)
  - Write codegen that generates all of the different kernels for you.
    * We actually do this in some cases: for autograd, and (currently) for tracing. This code is faster than a boxed fallback, but also requires work and careful design to make it maintainable.
- Binary size: We only have one function, instead of having separate specialized functions for every operator (and there are 1000+ operators).
  - This is especially important for the mobile use case: mobile cares a lot about having a small binary size!
  - Mobile internally also uses the Lite Interpreter, which executes ops in a boxed format. I'm not an expert on what this looks like though.

How is this related to the Dispatcher?

So, how is this whole notion of boxed vs. unboxed kernels relevant to the dispatcher?

Well, suppose we have a boxed kernel for Batching like we described above, and with batching turned on, I call `torch.sin(x)` on a cpu tensor. We expect batching logic to run before we eventually hit the sin() kernel.

The dispatcher is responsible for going from

- `at::sin()` (normal, unboxed frontend C++ entry point)
- to `void batchedTensorForLoopFallback(const c10::OperatorHandle&, torch::jit::Stack*)` (BOXED kernel that performs batching. Somehow all of the arguments to sin() need to be wrapped into a Stack!)
- to `at::native::sin()` (normal, UNBOXED cpu kernel for sin(). Somehow, the arguments need to be unboxed again so we can call this unboxed

function!)

Where does all of this boxing and unboxing conversion logic happen? Since the dispatcher provides API's for registering both unboxed and boxed kernels, then it also needs to know how to convert arguments and operators between the unboxed and boxed world between invocations.

- If you're curious, some of the template magic that does that lives around here: https://github.com/pytorch/pytorch/blob/8216da1f23b893c074e76e8a9aa7127efbda4287/aten/src/A