Writing a Toy Backend Compiler for PyTorch

This tutorial assumes familiarity with C++11 and uses bleeding-edge features from PyTorch. API semantics are likely to change over time, but the general approach should remain a useful guide for understanding PyTorch JIT internals. All of the code can be found here.

Why?

PyTorch is a machine learning framework that focuses on providing flexibility to users and has received praise for its simplicity, transparency and debuggability. PyTorch's natural Pythonic API makes it easy for new users to quickly become productive. It avoids restricting the language and provides an easy to understand eager execution model.

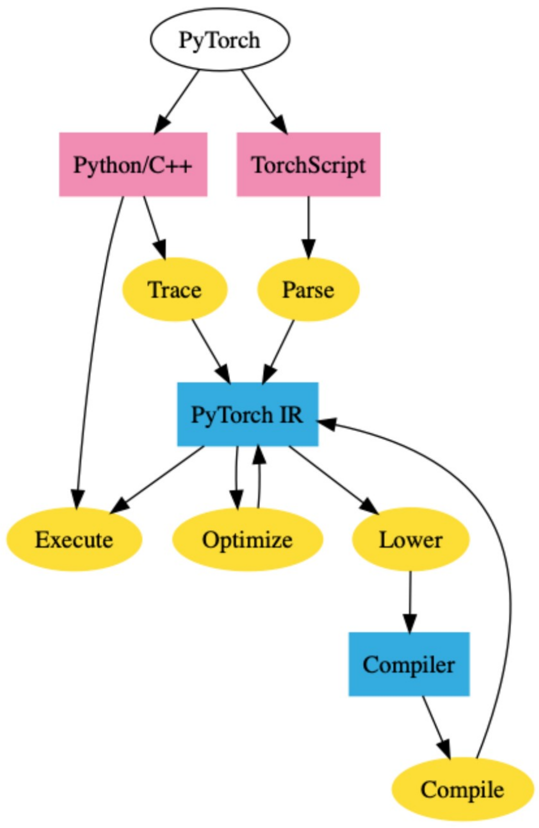
However, JIT compiling the dynamic Python code used to write programs with PyTorch is not easy. The PyTorch JIT team has gone to great lengths to make the compilation experience painless. The result is PyTorch IR, a convenient graph representation of PyTorch programs.

Often, custom backends and hardware require specialized compilation technqiues. Consistent with PyTorch's frontend API design philosophy, hooking up a new compiler should be a pleasant experience for all involved. This tutorial is designed as an end-to-end walkthrough detailing all that is necessary for building and integrating a compiler into PyTorch's JIT.

Approach

PyTorch's JIT can currently be invoked in two ways: via tracing and TorchScript. In this tutorial we will be transparently overriding some JIT functionality for these frontends by registering our own backend compiler. Our compiler will generate x64 assembly at runtime using asmjit. Ultimately, we hope to improve the performance for a small subset of operations.

The structure of PyTorch looks a bit like this:



We will be implementing the blue "Compiler" square at the bottom. For simplicity, our compiler will need static shape information, which PyTorch doesn't always provide. This means our code will need to JIT compile when we see new input shapes.

The blue "PyTorch IR" square is super important to understand, and there is a nice document detailing all of its inner workings. This will probably be a good reference to have open but is not required reading.

Code

We'll be implementing a "`PointwiseCompiler`" that can handle a couple of pointwise operations (well, just one: `mul`). We're not going to concern ourselves with the details of operator implementations as they are well studied in many other contexts.

A couple of restrictions are necessary to keep our compiler small and readable. The subset of PyTorch we're targeting will have

- No control flow
- No aliasing or in-place mutation

Although PyTorch exposes these concepts in the IR, they are not trivial to work with correctly and fall outside the scope of this tutorial. Dealing with them is left as an exercise to the reader.

Throughout the following section we will construct the interface to `PointwiseCompiler` and will then implement it in the section after. `PointwiseCompiler` will contain both compilation and runtime code. If you are eager to test the code as you go, all the code and build instructions can be found in the repo.

Registering a New Compiler (`register.cpp`)

To integrate a backend compiler, PyTorch requires static registration of two components.

- A transformation pass to partition the graph
- A custom operator executor that will compile/run the partition we care about

We will need to use a name to associate the two together, which PyTorch expects in the form of a `Symbol`. `Symbol`s are nothing more than interned strings, and there are utilities to help with their creation.

```
const  pointwise_compiler_symbol = Symbol::fromQualString("pw::CompilationGroup");
```

Transformation Pass [code]

First, we register a pass that will coalesce operators we can handle into a single operator containing a subgraph.

```
RegisterPass pass([pointwise_compiler_symbol](std::shared_ptr<Graph>& g) {
    CustomFuseGraph(g, PointwiseCompiler::supported, pointwise_compiler_symbol);
});
```

The `RegisterPass` object is really a sneaky static registration mechanism that you will see used throughout the PyTorch JIT codebase. The idea is to have the object's constructor tap into a global list of compiler passes and register a new one. If we compile the variable into our code the pass is enabled. Very useful!

Note the signature of the pass that we registered: `void f(std::shared_ptr<Graph>&)`. It is simply fed a PyTorch IR graph and is expected to mutate the graph in place. Currently, this pass will be called during JIT optimization on subsets of the graph after differentiation has already happened.

Although the pass interface is extremely flexible, manipulating the IR graph can be a hassle. A convenient helper function to use is

```
void  CustomFuseGraph   (std::shared_ptr<Graph>& g, bool (*)(Node*) callback, Symbol s)
```

This function deals with the graph and repeatedly invokes the callback (second argument) to determine which `Node*`s should be added to the subgraph of an operator labeled by the third argument. Importantly, it restricts the manipulations to pure functions and avoids control flow. In our code, the result of a call to this function will be a PyTorch IR graph that (potentially) includes operators called `"pw::CompilationGroup"`. These operators will contain subgraphs with nodes that we know our compiler can handle. Keep in mind that subgraphs are just `Graph`s as well.

We'll let `PointwiseCompiler::supported` deal with the operator support callback.

Custom Operator Execution [code]

If you were to attempt to execute any JIT compiled PyTorch program that was modified to include a `"pw::CompilationGroup"` by our above pass, the program would fail.

We need to teach PyTorch how to actually execute this operator.

```
// We are only dealing with pure operations (no aliasing or in place mutation),
// so our subgraph will always be pure.
auto  options = c10::OperatorOptions();
options.setAliasAnalysis(AliasAnalysisKind::PURE);

RegisterOperators op ({Operator(
    pointwise_compiler_symbol,
    [](const  Node* node) {
      auto  compiler = std::make_shared<PointwiseCompiler>(node);
      return  [compiler](Stack& stack) {
        compiler->run(stack);
        return  0;
      };
    },
    options)});
```

Similar to `RegisterPass` object, we will be using a `RegisterOperators` object to register a single `Operator` implementation. The `Operator` API used here takes the name of the operator as the first argument and options associated with the operator as the third. Note that we need to specify our operator as pure to allow JIT optimizations like dead code elimination to work properly.

The second argument is a function that processes the `const Node*` containing the operator and returns another function that can execute a `Stack&`. The function returned will be called on every invocation. Thus, we will instantiate a compiler while processing the `const Node*` and have it manipulate the stack. A `Stack` is just a `std::vector<IValue>`, which is kind of like `std::stack<std::any>`.

We will let `PointwiseCompiler::run` deal with the stack.

Test

To test that our half-baked code is at least working structurally, it is easiest to jump into Python. In the repo, we use `pybind11` for running the above code. This is convenient because the only required change to typical code is the `import pointwise_compiler` statement seen below. Everything else is handled transparently by PyTorch's JIT in the background.

```
import torch
import pointwise_compiler

A = torch.randn(1024)
B = torch.randn(1024)

@torch.jit.script
def foo(a, b):
  c = a.mul(b)
  a = c.mul(c)
  a = c.mul(a)
  return a

print(foo.graph_for(A,B))
```

The useful bit of code is `foo.graph_for(t)`, which will show us exactly what the graph looks like after our transformation pass. If `PointwiseCompiler::supported` returns `true` for a `Node*` with a `kind()` of `aten::mul`, we should see:

```
graph(%a.1 : Float(*),
      %b : Float(*)):
  %a : Float(*) = pw::CompilationGroup_0(%a.1, %b)
  return  (%a)
with pw::CompilationGroup_0 = graph(%4 : Float(*),
      %5 : Float(*)):
  %c : Float(*) = aten::mul(%4, %5) # test.py:33:7
  %a.2 : Float(*) = aten::mul(%c, %c) # test.py:34:7
  %a : Float(*) = aten::mul(%c, %a.2) # test.py:35:7
  return  (%a)
```

The above IR dump shows us that we have a `pw::CompilationGroup` containing three `aten::mul` nodes.

Now we need to write the compiler.

Writing the Compiler (`compiler.cpp`)

The strategy we'll employ here is simple. Because we are dealing with pure pointwise operations, any operation we support can be emitted on each scalar element within a tensor rather than on the entire tensor. Theoretically, this will save us operations that write to memory and speed up the program.

The above code (without compilation) would execute somewhat like this:

```
for i in N:
  tensor_c[i] = tensor_a[i] * tensor_b[i]
for i in N:
  tensor_a[i] = tensor_c[i] * tensor_c[i]
for i in N:
  tensor_a[i] = tensor_c[i] * tensor_a[i]
return tensor_a
```

We are going to compile that down to this:

```
for i in N:
  c_reg = tensor_a[i] * tensor_b[i]
  a_reg = c_reg * c_reg
  tensor_a[i] = c_reg * a_reg
return tensor_a
```

This simple change can save us `2N` writes to memory by keeping the values in registers inside the loop. For memory bound pointwise operations, should result in a sizable speedup. This is conventionally referred to as loop fusion.

In the following section we'll start to implement the interface we derived above, keeping these ideas in mind.

`bool PointwiseCompiler::supported(const torch::jit::Node* node)` [code]

This function should have an obvious implementation: return `true` for all operations the compiler can handle. The semantics of PyTorch operators are sometimes a bit tricky, so the function may want to do additional checks on the inputs to the node to ensure it truly can handle the code.

PyTorch's `mul` is not tricky, so we'll just deal with that.

```
bool PointwiseCompiler::supported(const torch::jit::Node* node) {
  switch (node->kind()) {
    case aten::mul:
      return true;
    default:
      return false;
  }
  return false;
}
```

The above code could probably be made simpler.

`void PointwiseCompiler::run(torch::jit::Stack& stack)` [code]

This function will serve the dual purpose of dispatching to a JIT compiler and running the code.

```
void PointwiseCompiler::run(torch::jit::Stack& stack) {
  // Get the number of expected inputs to the graph we are compiling
  const at::ArrayRef<Value*>& graph_inputs = subgraph_->inputs();
  const auto num_inputs = graph_inputs.size();

  // Pop these inputs from the stack.
  at::ArrayRef<IValue> inputs = last(stack, num_inputs);

  // If we haven't compiled for the shape/device of these inputs before,
  // do so now.
  CompleteArgumentSpec spec{false, ArrayRef<IValue>(inputs)};
  if (cache_.find(spec) == cache_.end()) {
    cache_[spec] = compile(inputs);
  }

  // Run the compiled function!
  auto outputs = cache_[spec](inputs);

  drop(stack, num_inputs);
  for (auto& output : outputs) {
    auto var = torch::autograd::make_variable(output.toTensor());
    stack.push_back(IValue(var));
  }
}
```

An important API used for the implementation of this function is `CompleteArgumentSpec`, which is a hashable, comparable value derived from the inputs and their types (including shapes). In the above code we use the `spec` to dispatch to code we've already compiled.

The heavy lifter in the above code is the call to `compile(inputs)`, which returns a function takes in `IValue`s and returns `IValue`s. `IValues` are the variant types used for all JIT execution. They can contain many things, including `Tensor`s, which is what we care about.

The rest of the code above is effectively a wrapper to use PyTorch's `Stack`-based execution.

Note that this code assumes the compiler can handle everything, which may not always be true. PyTorch provides a convenient way to fall back to the interpreter in the worst case:

```
#include <torch/csrc/jit/interpreter.h>

void runOnFailure(torch::jit::Stack& stack) {
  torch::jit::InterpreterState(torch::jit::Code(subgraph_)).run(stack);
}
```

Code Generation [code]

At this point we've covered the entirety of PyTorch's registration APIs. Now we will dive into how to interact with PyTorch IR itself for code generation.

We will be using `asmjit` heavily going forward with only relevant code snippets shown and explained. For a more detailed overview, see the repo. To keep things simple, the compiler will generate scalar code. Extending the code-gen to use vector instructions is left as an exercise to the reader. Surprisingly, even scalar code yields a slight speedup over non-compiled code on the machine used to test this tutorial.

We start by initializing some code generation utilities.

```
CompiledCode PointwiseCompiler::compile(
    at::ArrayRef<torch::jit::IValue>& inputs) {

  // ... Checks here ...

  auto reg_manager = RegisterManager();
  asmjit::CodeHolder code;
  code.init(jit_runtime_.getCodeInfo());
  asmjit::X86Assembler assembler(&code);
```

`RegisterManager` is a simple object containing register maps for both general purpose and floating point registers. We will use this object to map `Value*` to registers containing either addresses or values (if they've been loaded from the address).

Load Addresses

The function we are generating code for will be passed an array of data pointers (`void fn(void** data)`), each of which corresponds to an input or output `Value*` in the PyTorch IR. `Value*s` represent the data flowing through a program. They are entirely symbolic, so we will map them to real data.

The first code generated will map all the input `Value*`s to their corresponding data pointers stored in general purpose registers.

```
  asmjit::X86Gp pointers;

  // Move all the input Tensor addresses into registers
  for (auto i = 0; i < inputs.size(); ++i) {
    auto reg = reg_manager.getFreeAddrReg();
    auto mem_ptr = asmjit::x86::ptr(pointers, i * sizeof(void *));
    reg_manager.mapReg(subgraph_->inputs()[i], reg);
    assembler.mov(reg, mem_ptr);
  }
```

The same thing is done for output data pointers. After that, we're set to start computation.

Create a Loop

Because we're dealing with pointwise code, we will just emit all of our floating point instructions inside a for-loop. In assembly this is most easily expressed as a label we can jump to and a register keeping track of iterations.

```
  // Setup a label for looping
  auto iter = reg_manager.getFreeAddrReg();
  assembler.mov(iter, 0);
  auto loop_label = assembler.newLabel();
  assembler.bind(loop_label);
```

For simplicity, we will insert the bound check and jump instructions after emitting the inner loop.

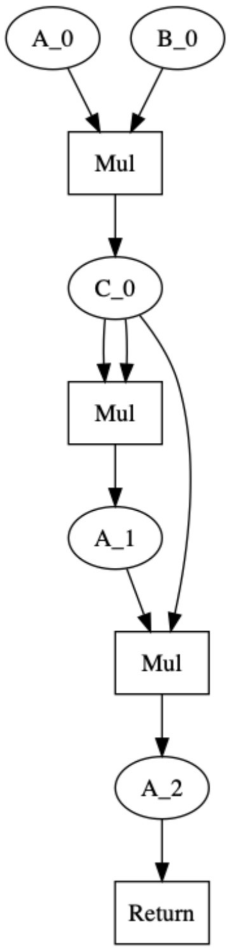
Load Values

Similar to the address mapping we did for inputs, we will use the `RegisterManager` to map live `Value*`s to `xmm` floating point registers. We will also use `RegisterManager` to free the registers for reuse. We first load input values from memory and use the `iter` register to get the offset.

```
for (auto input : subgraph_->inputs()) {
  auto reg = reg_manager.getFreeValueReg();
  assembler.movd(
      reg, asmjit::x86::ptr(reg_manager.getAddrReg(input), iter, 2));
  reg_manager.mapReg(input, reg);
}
```

Traverse PyTorch IR

We will now need to inspect the actual graph PyTorch has handed us. Everything in PyTorch IR is represented in SSA. Below is a visualization of the PyTorch IR for the example program we are dealing with:



The strategy we will take is to traverse the frontier of the graph, attemping to reuse floating point registers when we can (if the values they are storing aren't needed anymore).

We can walk the PyTorch graph using `nodes()`, which is conveniently topologically sorted.

```
// Iterating over graph nodes is guaranteed to be topologically sorted
for (auto node : subgraph_->nodes()) {
  seen.insert(node);
  emitOperation(node, seen, assembler, reg_manager);
}
```

The `emitOperation` [code] function needs to do two things:

- Emit the x86 instructions corresponding to the current PyTorch node (including using any new registers for outputs)
- Check to see if any input to the node has been entirely consumed with the `seen` variable and free the corresponding register if so

Store Outputs

Just as we loaded values from memory, we will store them back using the `iter` register for the offset.

```
// Store all the output values into memory.
for (auto output : subgraph_->outputs()) {
  assembler.movd(
      asmjit::x86::ptr(reg_manager.getAddrReg(output), iter, 2),
      reg_manager.getValueReg(output));
}
```

Close Loop and Exit

Finally, we will insert the loop branching logic and return from the function.

```
assembler.add(iter, 1);
assembler.cmp(iter, size);
assembler.jb(loop_label);

assembler.ret();
```

Bind Generated Code [code]

As a final step we need to bind the generated code to the `compile` interface we designed above.

```
  std::vector<void *> args;
  for (auto input : inputs) {
    TORCH_CHECK(input.isTensor());
    TORCH_CHECK(input.toTensor().is_contiguous());
    TORCH_CHECK(input.toTensor().device().is_cpu());
    args.emplace_back(input.toTensor().data_ptr());
  }
  std::vector<IValue> outputs;
  for (auto output : subgraph_->outputs()) {
    outputs.emplace_back(at::empty({size}));
  }
  for (auto output : outputs) {
    args.emplace_back(output.toTensor().data_ptr());
  }
```

Using `at::empty` we can allocate output tensors. The raw `void*` values can be extracted from `IValue`s with `ival.toTensor().data_ptr()`.

Results

It is important to test our compiler with "real" examples. Check out the `test.py` file in the repository. It uses `torch.randn(n)` to generate example inputs and `torch.allclose(t1, t2)` to compare output tensors.

There is also a small benchmark script:

```
def benchmark(f):
  A_ = torch.randn(1024)
  B_ = torch.randn(1024)
  # Warmup
  for _ in range(10):
    _ = f(A_,B_)
  t = time.time()
  for _ in range(100):
    _ = f(A_,B_)
  return time.time() - t
```

On the device used to test this tutorial our compiler yields a **2x speed up** over the default JIT backend.

Followup Exercises

The compiler built in this tutorial is missing many features that might be nice exercises for the curious reader.

Handle more pointwise operations

Try adding `div`, `relu` or other operations using the `mul` implementation as an example. Note: PyTorch has strange semantics for `add`.

Vectorize the code

The generated code loads only single values at a time, which isn't efficient on modern CPUs. Using vector loads and instructions, the code can be sped up substantially.

Tile the code

Unroll the loops!

Implement an auto-tuning mechanism

Parameterize the above optimizations and search for the best set of parameters for performance.

Handle non-pointwise operations

This change requires a sizable refactor of the `compile` function, which presupposes the code generated can exist within a single for-loop.

Implement register spilling

For large programs this compiler will saturate all the registers it knows about. A fix is to spill registers into memory.