How Pytorch 2.0 Accelerates Deep Learning with Operator Fusion and CPU/GPU Code-Generation

A primer on deep learning compiler technologies in PyTorch for graph capture, intermediate representations, operator fusion, and optimized C++ and GPU code generation

Shashank Prasanna · Follow

Published in Towards Data Science · 17 min read · Apr 20, 2023

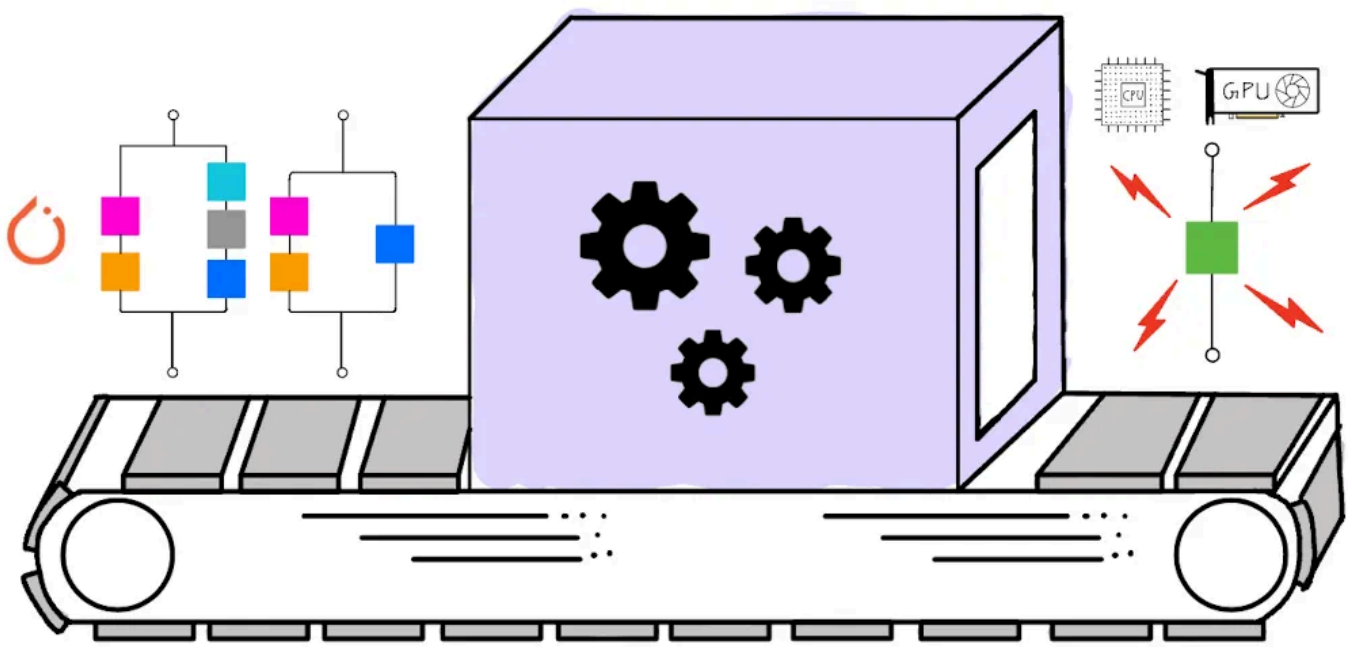


illustration by author

Computer programming is magical. We write code in human readable languages, and as though by magic, it gets translated into electric currents through silicon transistors making them behave like switches and allowing them to implement complex logic — just so we can enjoy cat videos on the internet. Between the programming language and hardware processors that run it, is an important piece of technology — the compiler. A compiler's job is to translate and simplify our human readable language code into instructions that a processor understands.

Compilers play a very important role in deep learning to improve training and inference performance, improve energy efficiency, and target diverse AI accelerator hardware. In this blog post I'm going to discuss deep learning compiler technologies that powers PyTorch 2.0. I'll walk you through the different phases of the compilation process and discuss various underlying technologies with code examples and visualizations.

What is a deep learning compiler?

A deep learning compiler translates high-level code written in deep learning frameworks into optimized lower level hardware specific code to accelerate training and inference. It finds opportunities in deep learning models to optimize for performance by performing layer and operator fusion, better memory planning, and generating target specific optimized fused kernels to reduce function call overhead.

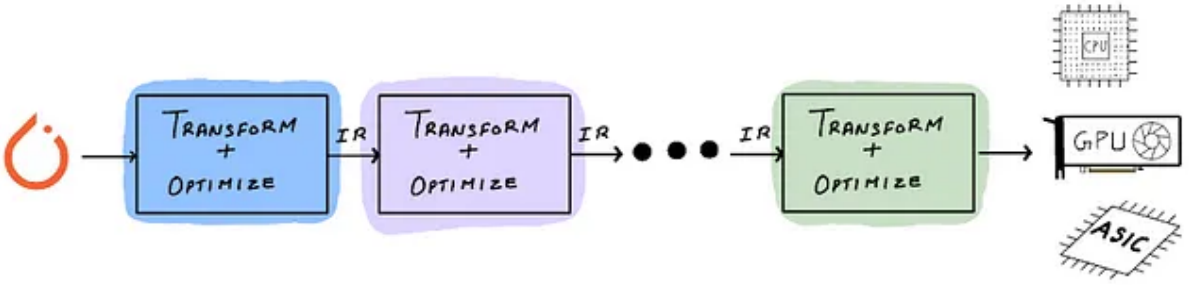


illustration by author

Unlike traditional software compilers, deep learning compilers have to work with highly-parallelizable code often accelerated on specialized AI accelerator hardware (GPUs, TPUs, AWS Trainium/Inferentia, Intel Habana Gaudi etc.). To improve performance, a deep learning compiler has to take advantage of hardware specific features such as mixed precision support, performance optimized kernels and minimize communication between host (CPU) and AI accelerator.

While deep learning algorithms are continuing to advance at a rapid pace, hardware AI accelerators have also been evolving alongside to keep up with

deep learning algorithm performance and efficiency needs. I discuss the co-evolution of algorithms and AI accelerators in an earlier blog post:

In this blog post I'll focus on the software side of things, and particularly the subset of software closer to the hardware — deep learning compilers. First, let's start by taking a look at different functions in a deep learning compiler.

Deep learning compiler in PyTorch 2.0

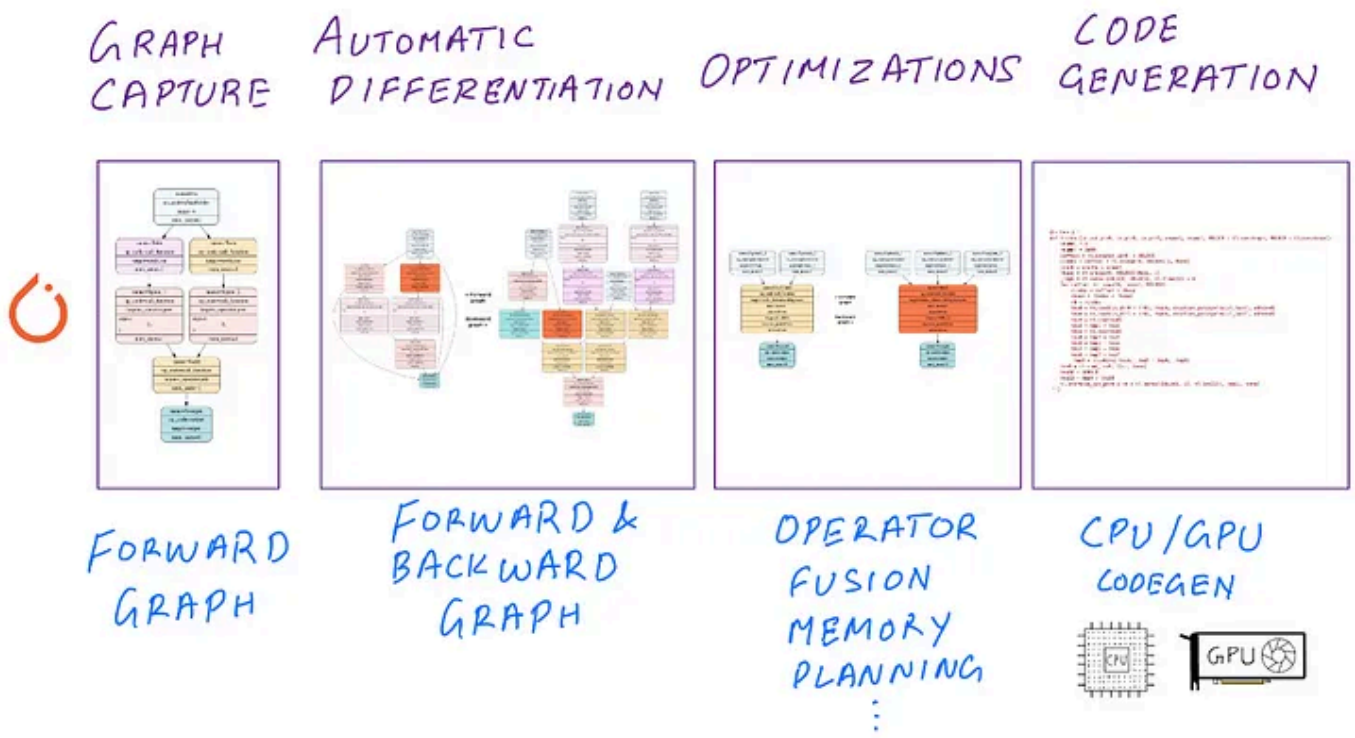
PyTorch 2.0 includes new compiler technologies to improve model performance and runtime efficiency and target diverse hardware backends with a simple API: torch.compile(). While other blog posts and articles have discussed performance benefits of PyTorch 2.0 in detail, here I'm going to focus on what happens under the hood when you invoke the PyTorch 2.0 compiler. If you're looking for quantified performance benefits, you can find a performance dashboard of different models from huggingface, timm and torchbench.

At a high-level the default options for PyTorch 2.0 deep learning compiler performs the following key tasks:

1. **Graph capture**: Computational graph representation for your models and functions. PyTorch technologies: TorchDynamo, Torch FX, FX IR

2. **Automatic differentiation**: Backward graph tracing using automatic differentiation and lowering to primitives operators. PyTorch technologies: AOTAutograd, Aten IR
3. **Optimizations**: Forward and backward graph-level optimizations and operator fusion. PyTorch technologies: TorchInductor (default) or other compilers
4. **Code generation**: Generating hardware specific C++/GPU Code. PyTorch technologies: TorchInductor, OpenAI Triton (default) other compilers

Through these steps, the compiler transforms your code and generates intermediate representations (IRs) that are progressively “lowered”. Lowering is a term in the compiler lexicon that refers to mapping a broad set of operations (such as supported by PyTorch API) to a narrow set of operations (such as supported by hardware) through automatic transformation and re-writing by the compiler. The PyTorch 2.0 compiler flow:



If you are new to compiler terminology don't let all of this scare you yet. I'm not a compiler engineer either. Keep reading and things will become clear as I'll break the process down using a simple example and visualizations.

A walk through the `torch.compile()` compiler process

Note: This whole walkthrough is in a Jupyter Notebook hosted here

For the sake of simplicity, I'll define a very simple function and run it through the PyTorch 2.0 compiler process. You can replace this function with a deep neural network model or an nn.Module subclass, but this example should help you appreciate what's going on under the hood much better than a complex multi-million parameter model.

$$y = f(x) = sin^2(x) + cos^2(x)$$

PyTorch code for that function:

```
def f(x):
  return torch.sin(x)**2 + torch.cos(x)**2
```

If you paid attention in high-school trigonometry class, you know that the value of our function is always going to be 1 for all real valued x. Which means it's derivative, a derivative of a constant, and must be equal to zero. This will come in handy to verify what the function and its derivatives are doing.

Now, it's time to call `torch.compile()` . First let's convince ourselves that compiling this function doesn't change its output. For the same 1x1000 random vector the mean squared error between the output of our function and a vector of 1s should be zero for both the compiled and the uncompiled function (under some error tolerance).

```
torch.manual_seed(0)
x = torch.rand(1000, requires_grad=True).to(device)
torch.nn.functional.mse_loss(f(x),torch.ones_like(x)) < 1e-10
```

```
tensor(True, device='cuda:0')
```

```
torch.manual_seed(0)
x = torch.rand(1000, requires_grad=True).to(device)

compiled_f = torch.compile(f)
torch.nn.functional.mse_loss(compiled_f(x),torch.ones_like(x)) < 1e-10
```

```
tensor(True, device='cuda:0')
```

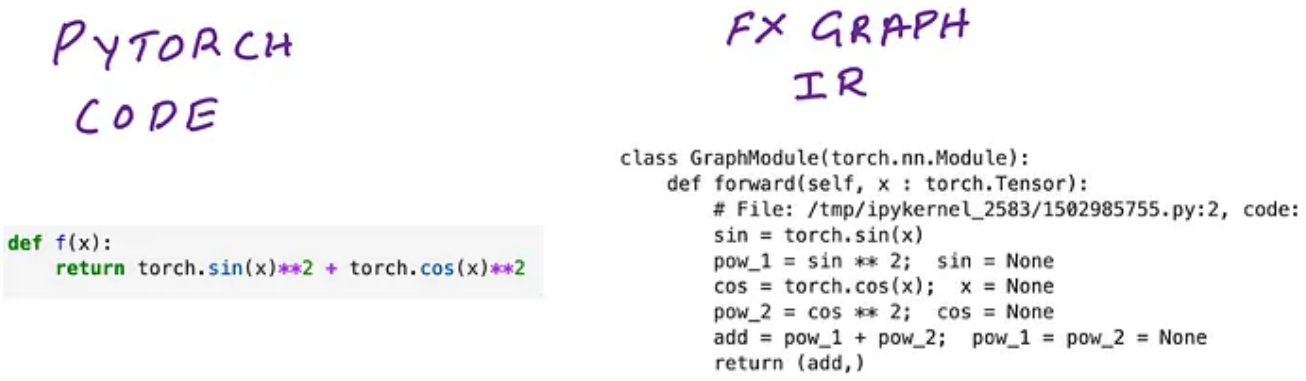
screenshot by author

All we did was add a single line of extra code `torch.compile()` to invoke our compiler. Let's now take a look at what's happening under the hood at each stage.

Graph capture: Computational graph representation for your PyTorch models or functions

PyTorch technologies: TorchDynamo, FX Graphs, FX IR

The first step for the compiler is to determine what to compile. Enter TorchDynamo. TorchDynamo intercepts the execution of your Python code and transforms it into FX intermediate representation (IR), and stores it in a special data structure called FX Graph. What does this look like you ask? Glad you asked. Below, we'll take a looks the code we use to generate this, but here is the transformation and output:



screenshot by author

It's important to note that Torch FX graphs are just containers for IR and don't really specify what operators it should hold. In the next section we'll see the FX graph container come up again with a different set of IRs. If you compare the function code and FX IR there's very little difference between

the two. In fact, it's the same PyTorch code you wrote, but laid out in a format that the FX graph data structure expects. They both will provide the same result when executed.

If you call torch.compile() without any arguments, it'll use the default settings which runs the entire compiler stack which includes the default hardware backend compiler called TorchInductor. But we'd be jumping ahead if we discussed TorchInductor now, so let's park that topic for now, and we'll come back to it when we're ready. First we need to discuss graph capture and we can do that by intercepting the calls from torch.compile(). Here's how we'll do that: torch.compile() allows you to provide your own compiler too, but because I'm not a compiler engineer, and I don't have the slightest clue how to write a compiler, I'll provide a fake compiler function to capture the FX graph IR that TorchDynamo generates.

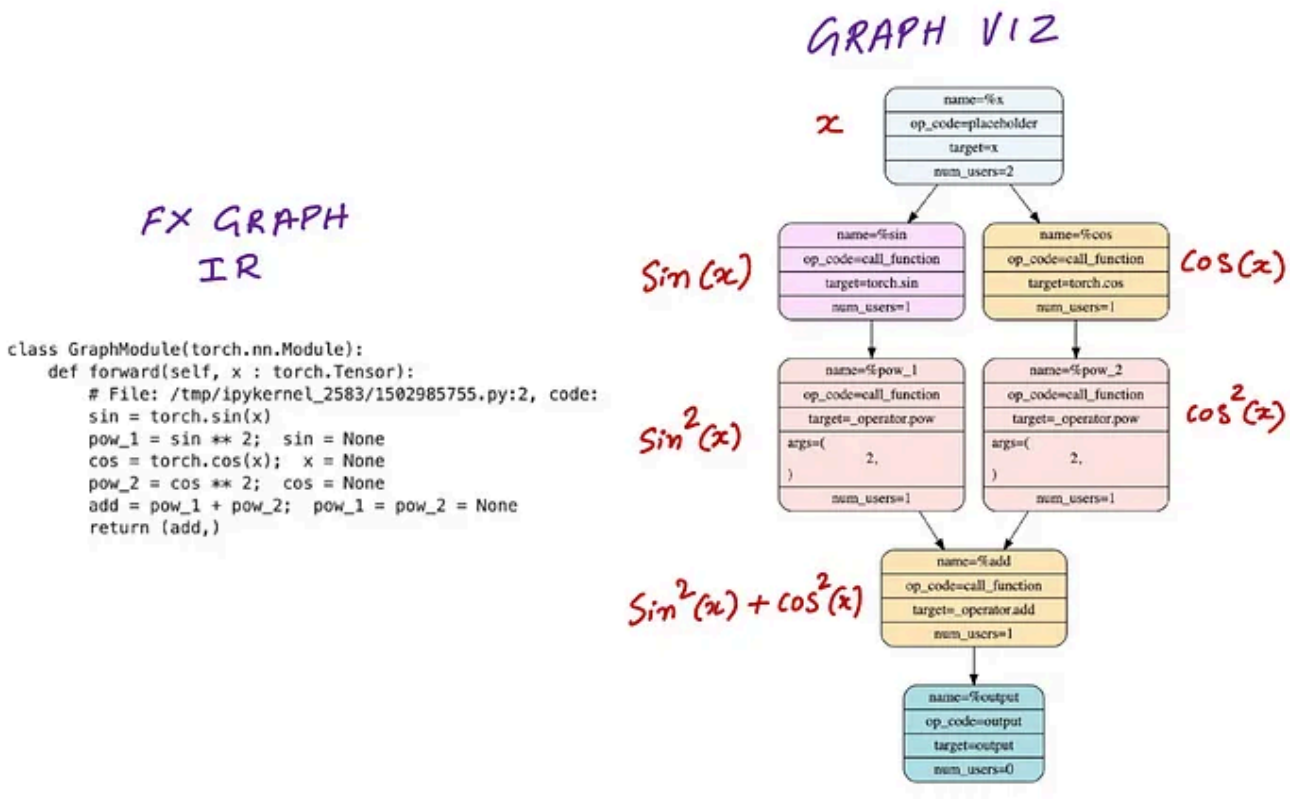
Below is our fake compiler backend function called inspect_backend to torch.compile() and within that function I do two things:

1. Print the FX IR code that was captured by TorchDynamo
2. Save the FX graph visualization

```python
def inspect_backend(gm, sample_inputs):
    code = gm.print_readable()
    with open("forward.svg", "wb") as file:
        file.write(FxGraphDrawer(gm,'f').get_dot_graph().create_svg())
    return gm.forward

torch._dynamo.reset()
compiled_f = torch.compile(f, backend=inspect_backend)

x = torch.rand(1000, requires_grad=True).to(device)
out = compiled_f(x)
```

torch_compile1.py hosted with ❤️ by GitHub view raw

The output of that above code snippet are the FX IR code and the graph diagram showing our function `sin^2(x)+cos^2(x)`



screenshot by author

Note that our fake compiler inspect_backend function is only invoked when we call the compiled function with some data i.e. when we call `compiled_model(x)`. In the above code snippet, we're only evaluating the function or in deep learning terminology, doing a "forward-pass". In the next section we'll take advantage of the PyTorch's automatic differentiation engine called torch.autograd to compute the derivative and the "backward-pass" graph.

Automatic differentiation: Forward and backward computational graphs

PyTorch technologies: AOTAutograd, Core Aten IR

TorchDynamo gave us the forward pass function evaluation as an FX graph, but what about the backward pass? For the sake of completeness, I'm going to digress from our primary topic and talk a bit about why we need to evaluate the gradients of a function with respect to its weights. If you're already familiar with how mathematical optimization works skip this immediate section.

What is backward pass and backward graph?

The "learning" part of deep learning and machine learning is a mathematical optimization problem which is simply stated as: Find the value of a variable w that yields the lowest value of some function of w. Or more succinctly:

$$\text{find } w^* \text{ that } \min_{w} f(w)$$

In machine learning f(w) is the loss function parametrized by weights. f(w) can be more clearly represented as some measure of error between the training labels and the model's prediction labels based on the training data:

$$loss(w) : loss(model(w, batch_{inputs}), batch_{outputs})$$

Turns out, if we can calculate the "rate of reduction" of loss with respect to weights, we can update our weights to move one step closer to a smaller and smaller loss f(w). In other words, we must move closer to a model that better fits our training dataset. We can find next values of weights by calculating the steepest slope of the loss f(w) at a given w and perturb w to head in that direction. The slope of a function with respect to the weights, is its derivative with respect to the weights. Since there are more than one weight values, the derivative becomes a vector quantity called the gradient which is a vector of partial derivatives with components for each weight. The weights w are perturbed at each iteration by some function g() of the gradients as follows:

$$w_{i+1} = w_i - g(\nabla f(w))$$

Where the function g(.) depends on the optimizer (e.g. sgd, sgdm, rmsprop, adam etc.).

For SGD the weight update step becomes:

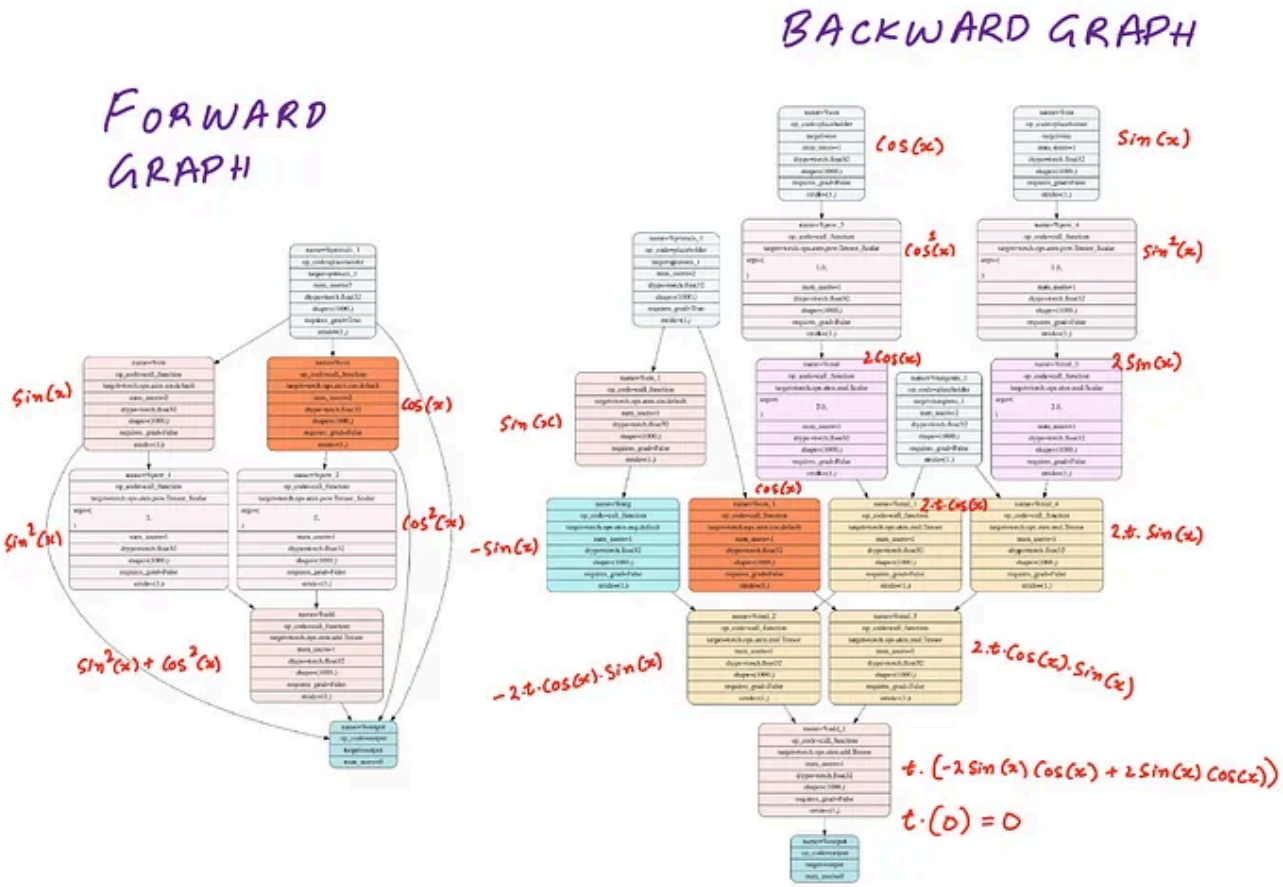
How does PyTorch 2.0 trace the backward pass graph?

First let's calculate what we expect the backward pass graph should look like and then compare it with what PyTorch generates. For our simple function, the forward graph and the backward graph should implement the following function. If sin and cos bother you, you can imagine f(x) being the loss function applied to a neural network.

Forward graph: $f(x) = sin^2(x) + cos^2(x)$

Backward graph: $\frac{df(x)}{d\vec{w}} = f'(x) = 2sin(x)cos(x) + 2cos(x)(-sin(x))$

PyTorch uses reverse-mode automatic differentiation to compute the gradients, and PyTorch's implementation of automation differentiation is called Autograd. PyTorch 2.0 introduces AOTAutograd which traces the forward and backward graph ahead of time, i.e. prior to execution, and generates a joint forward and backward graph. It then partitions the forward and the backward graph into two separate graphs. Both the forward and the backward graphs are stored in the FX graph data structure and can be visualized as shown below.



screenshot by author

You can verify that the math checks out by working through the nodes on the graph. AOTAutograd generated backward pass indeed computes the derivative shown in the equation I shared earlier, which should equal zero since the original function only produces the identity.

We'll now run AOTAutograd by extending our fake compiler function inspect_backend to call AOTAutograd and generate our backward graph. The updated inspect_backed defines a forward (fw) and backward (bw) compiler capture function that reads the forward and backward graph from AOTAutograd and prints the lowered ATen IR and saves the FX graph for the forward and backward graphs.

```python
import torch._dynamo
from torch.fx.passes.graph_drawer import FxGraphDrawer
from functorch.compile import make_boxed_func
from torch._functorch.aot_autograd import aot_module_simplified

def f(x):
    return torch.sin(x)**2 + torch.cos(x)**2

def inspect_backend(gm, sample_inputs):
    # Forward compiler capture
    def fw(gm, sample_inputs):
        gm.print_readable()
        g = FxGraphDrawer(gm, 'fn')
        with open("forward_aot.svg", "wb") as file:
            file.write(g.get_dot_graph().create_svg())
        return make_boxed_func(gm.forward)

    # Backward compiler capture
    def bw(gm, sample_inputs):
        gm.print_readable()
        g = FxGraphDrawer(gm, 'fn')
        with open("backward_aot.svg", "wb") as file:
            file.write(g.get_dot_graph().create_svg())
        return make_boxed_func(gm.forward)

    # Call AOTAutograd
    gm_forward = aot_module_simplified(gm,sample_inputs,
                                       fw_compiler=fw,
                                       bw_compiler=bw)

    return gm_forward

torch.manual_seed(0)
x = torch.rand(1000, requires_grad=True).to(device)
y = torch.ones_like(x)

torch._dynamo.reset()
compiled_f = torch.compile(f, backend=inspect_backend)
out = torch.nn.functional.mse_loss(compiled_f(x), y).backward()
```

This will generate the following forward AND backward graphs. Notice that the forward graph also looks slightly different from what we saw earlier in Figure x. For example torch.sin(x) in the FX graph IR and in our original code has been replaced by torch.ops.aten.sin.default(). What's this funny thing called aten, you might ask, if you're not already familiar with it. ATen stands for A Tensor library, which is a very creatively named low level library with a C++ interface that implements many of the fundamental operations that run on CPU and GPU.

In eager mode operation, your PyTorch operations are routed to this library which then calls the appropriate CPU or GPU implementation. AOTAutograd automatically generates code that replaces the higher level PyTorch API with ATen IR for the forward and backward graph which you can see in the output below:

```
class GraphModule(torch.nn.Module):
    def forward(self, primals_1: f32[1000]):
        # File: /tmp/ipykernel_2583/2663716573.py:7, code: return torch.sin(x)**2 + torch.cos(x)**2
        sin: f32[1000] = torch.ops.aten.sin.default(primals_1)
        pow_1: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(sin, 2)
        cos: f32[1000] = torch.ops.aten.cos.default(primals_1)
        pow_2: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(cos, 2)
        add: f32[1000] = torch.ops.aten.add.Tensor(pow_1, pow_2);  pow_1 = pow_2 = None
        return [add, sin, cos, primals_1]

class GraphModule(torch.nn.Module):
    def forward(self, sin: f32[1000], cos: f32[1000], primals_1: f32[1000], tangents_1: f32[1000]):
        # File: /tmp/ipykernel_2583/2663716573.py:7, code: return torch.sin(x)**2 + torch.cos(x)**2
        pow_3: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(cos, 1.0);  cos = None
        mul: f32[1000] = torch.ops.aten.mul.Scalar(pow_3, 2.0);  pow_3 = None
        mul_1: f32[1000] = torch.ops.aten.mul.Tensor(tangents_1, mul);  mul = None
        sin_1: f32[1000] = torch.ops.aten.sin.default(primals_1)
        neg: f32[1000] = torch.ops.aten.neg.default(sin_1);  sin_1 = None
        mul_2: f32[1000] = torch.ops.aten.mul.Tensor(mul_1, neg);  mul_1 = neg = None
        pow_4: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(sin, 1.0);  sin = None
        mul_3: f32[1000] = torch.ops.aten.mul.Scalar(pow_4, 2.0);  pow_4 = None
        mul_4: f32[1000] = torch.ops.aten.mul.Tensor(tangents_1, mul_3);  tangents_1 = mul_3 = None
        cos_1: f32[1000] = torch.ops.aten.cos.default(primals_1);  primals_1 = None
        mul_5: f32[1000] = torch.ops.aten.mul.Tensor(mul_4, cos_1);  mul_4 = cos_1 = None

        # File: /tmp/ipykernel_2583/2663716573.py:7, code: return torch.sin(x)**2 + torch.cos(x)**2
        add_1: f32[1000] = torch.ops.aten.add.Tensor(mul_2, mul_5);  mul_2 = mul_5 = None
        return [add_1]
```

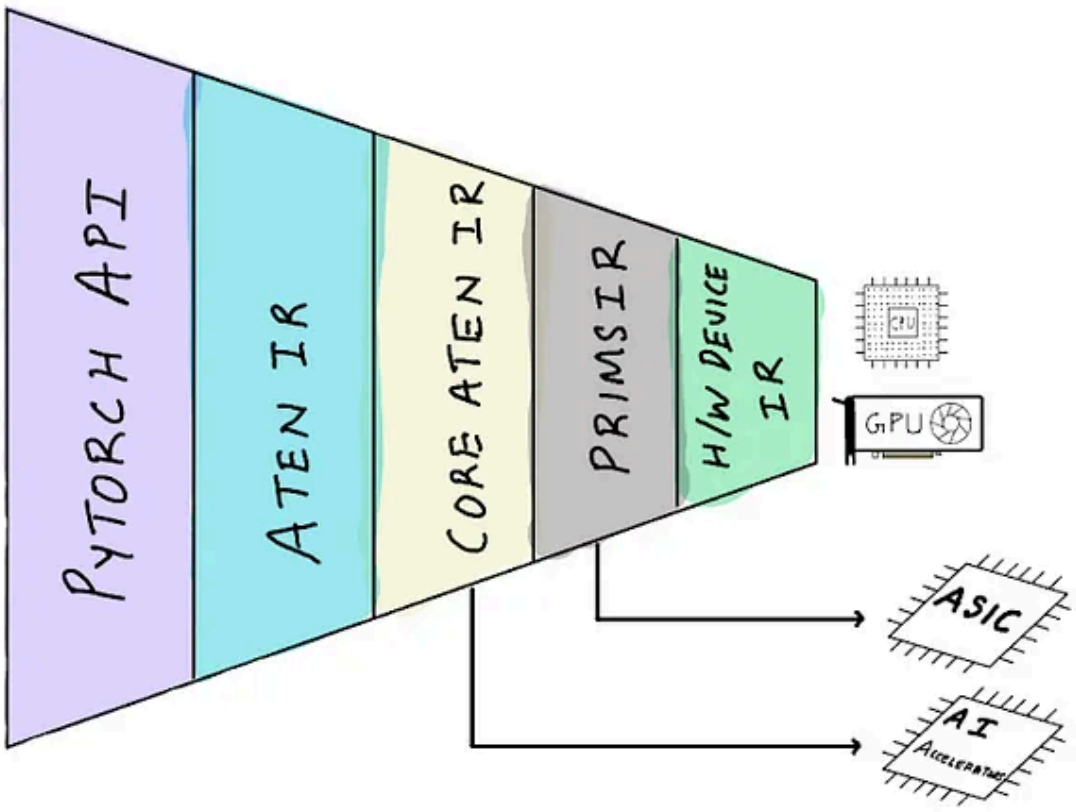
screenshot by author

You can also see that in addition to the output of the forward pass, the forward graph outputs some additional tensors [add, sin, cos, primals_1] . These tensors are saved for the backward pass for gradient calculation. You can also see this in the computational graphs for the forward and backward pass in the figure shared earlier.

What are the different types of IR in PyTorch?

ATen IR is a list of operators supported by the ATen library as we discussed in the previous section, and you can see the full list of operations implemented in ATen library here. There are two other IR concepts in PyTorch you should be aware of: 1/ Core Aten IR 2/ Prims IR. Core Aten IR is a subset of the broader Aten IR and Prims IR and an even smaller subset of Core Aten IR. Let's say you are designing a processor and want to support PyTorch code acceleration on your hardware. It'd be near impossible to support the full list of PyTorch API in hardware, so what you can do is build a compiler that only supports the smaller subset of fundamental operators defined in Core Aten IR or Prims IR, and let AOTAutograd decompose compound operators into the core operators as we'll see in the next section.

What's the difference between ATen IR, Core ATen IR, Prims IR?



screenshot by author

Core Aten IR (formerly canonical Aten IR) is a subset of the Aten IR that can be used to compose all other operators in the Aten IR. Compilers that target specific hardware accelerators can focus on supporting only the Core Aten IR and mapping it to their low level hardware API. This makes it easier to add hardware support to PyTorch since they don't have to implement support for the full PyTorch API which will continue to grow with more and more abstractions.

Prims IR is an even smaller subset of the Core Aten IR that further decomposes Core Aten IR ops into fundamental operations making it even easier for compilers that target specific hardware to support PyTorch. But decomposing operators into lower and lower operations will most definitely lead to performance degradation due to excess memory writes and function

call overhead. But the expectation is that hardware compilers can take these operators and fuse them back together to support hardware API to get back performance.

While we don't need to further decompose our function into Core Aten IR and Prims IR I'll demonstrate how below.

(Optional topic) Decomposition to Core Aten IR and Prims IR

If you're designing hardware or hardware compilers, it'd be near impossible to support the full list of PyTorch API in hardware, especially given the pace at which deep learning and AI are advancing. But the advantage for a hardware designer is that most deep learning functionality can be mapped into very few basic mathematical operations and the most computationally intensive ones are matrix-matrix and matrix-vector operations. Compound operators like those supported by PyTorch API can be decomposed into these fundamental operations using AOTAutograd as we'll discuss in this section. If you don't deal with low level hardware, you can skip this section.

You can update the AOTAutograd function to pass in a dictionary of decompositions that can lower the Aten IR into Core Aten IR and Prims IR. I'll only share the relevant code snippet and output here since you can find the full notebook on GitHub. By default operators are not decomposed into Core Aten IR or Prims IR, but you can pass a dictionary of decompositions.

In the code snippet below, I've converted our function f into a loss function f_loss by including the computation of mean squared error (MSE) into our function. I'm doing this to demonstrate how AOTAutograd can decompose MSE into its fundamental operators.

```
import torch._dynamo
from torch.fx.passes.graph_drawer import FxGraphDrawer
from functorch.compile import make_boxed_func
from torch._functorch.aot_autograd import aot_module_simplified
from torch._decomp import core_aten_decompositions

def f_loss(x, y):
    f_x = torch.sin(x)**2 + torch.cos(x)**2
    return torch.nn.functional.mse_loss(f_x, y)

decompositions = core_aten_decompositions() # Use decomposition to Core Aten IR
# decompositions = {} # Don't use decomposition to Core Aten IR
```

```
    return aot_module_simplified(
        gm,
        sample_inputs,
        fw_compiler=fw,
        bw_compiler=bw,
        decompositions=decompositions
    )
```

screenshot by author

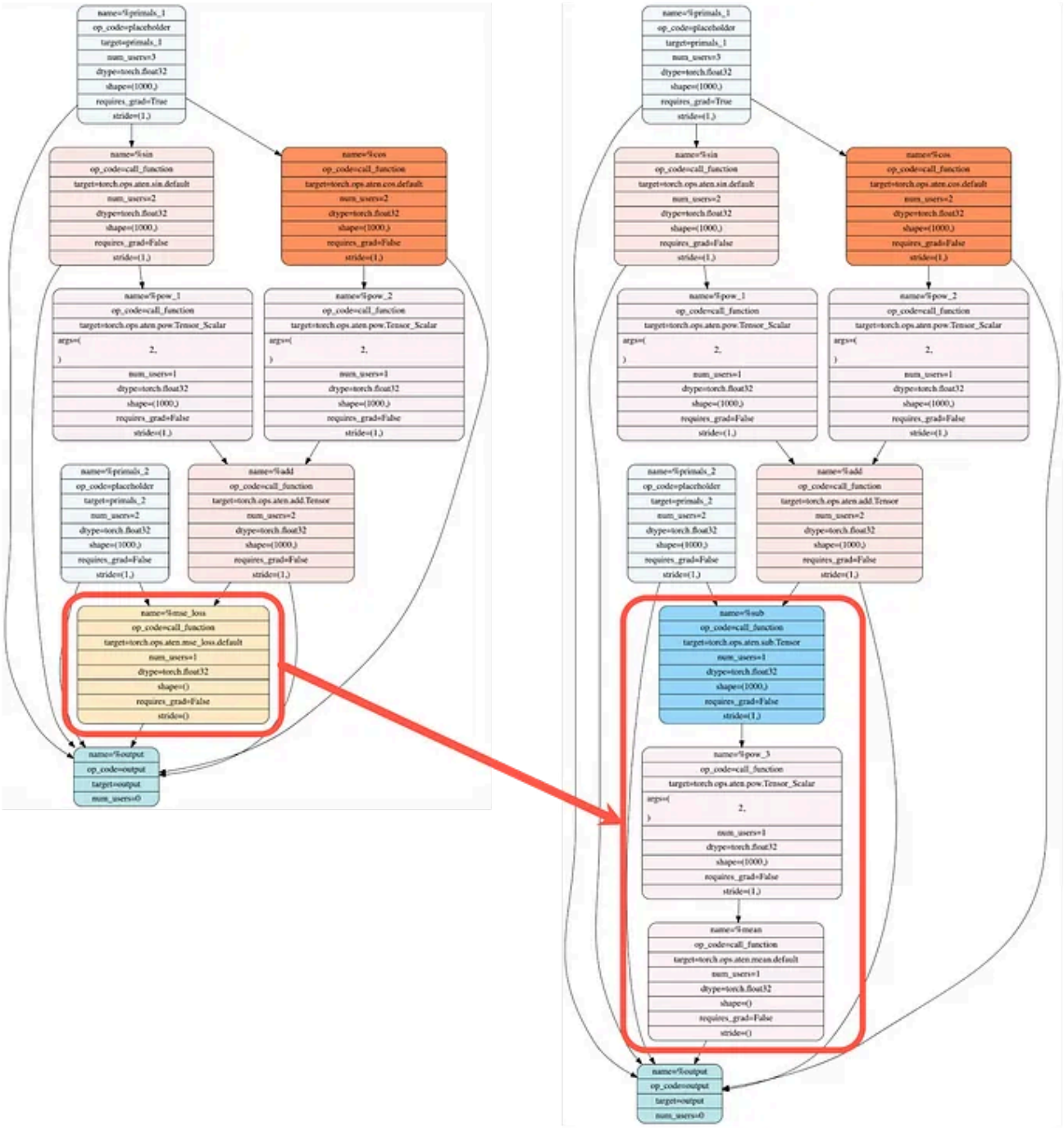
The output of the decomposition is that mse_loss gets decomposed into more fundamental operations: subtract , power(2) , mean.

screenshot by author

This is because MSE or mean square error between two vectors x and y is defined as the following which only needs those 3 operations subtract, where power is an element-wise operation. If you write a compiler for your hardware, you likely already support these 3 operations and by decomposition your PyTorch code would run without further modifications.

$$MSE = (\frac{1}{n})(\vec{y} - \vec{x})^2$$

You can also see this reflected in the FX graph visualization



screenshot by author

Now let's decompose it further into Prims IR which is a much smaller subset of ~250 operators. Again, I'll only share the relevant code snippet and output here since you can find the full notebook on GitHub.

```
import torch._dynamo
from torch.fx.passes.graph_drawer import FxGraphDrawer
from functorch.compile import make_boxed_func
from torch._functorch.aot_autograd import aot_module_simplified
from torch._decomp import core_aten_decompositions

def f_loss(x, y):
    f_x = torch.sin(x)**2 + torch.cos(x)**2
    return torch.nn.functional.mse_loss(f_x, y)

decompositions = core_aten_decompositions()
decompositions.update(
    torch._decomp.get_decompositions([
        torch.ops.aten.sin,
        torch.ops.aten.cos,
        torch.ops.aten.add,
        torch.ops.aten.sub,
        torch.ops.aten.mul,
        torch.ops.aten.sum,
        torch.ops.aten.mean,
        torch.ops.aten.pow.Tensor_Scalar,
    ])
)
```

```
        gm,
        sample_inputs,
        fw_compiler=fw,
        bw_compiler=bw,
        decompositions=decompositions
    )
```

screenshot by author

The output of the prim IR decomposition is below. All the aten ops in RED are replaced or decomposed to use prim operators in green.

```
class GraphModule(torch.nn.Module):
    def forward(self, primals_1: f32[1000], primals_2: f32[1000]):
        # File: /tmp/ipykernel_2583/3442173555.py:8, code: f_x = torch.sin(x)**2 + torch.cos(x)**2
        sin: f32[1000] = torch.ops.aten.sin.default(primals_1)
        pow_1: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(sin, 2)
        cos: f32[1000] = torch.ops.aten.cos.default(primals_1)
        pow_2: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(cos, 2)
        add: f32[1000] = torch.ops.aten.add.Tensor(pow_1, pow_2);  pow_1 = pow_2 = None

        # File: /tmp/ipykernel_2583/3442173555.py:9, code: return torch.nn.functional.mse_loss(f_x, y)
        sub: f32[1000] = torch.ops.aten.sub.Tensor(add, primals_2)
        pow_3: f32[1000] = torch.ops.aten.pow.Tensor_Scalar(sub, 2);  sub = None
        mean: f32[] = torch.ops.aten.mean.default(pow_3);  pow_3 = None
        return [mean, primals_2, cos, add, sin, primals_1]
```

```
class GraphModule(torch.nn.Module):
    def forward(self, primals_1: f32[1000], primals_2: f32[1000]):
        # File: /tmp/ipykernel_2583/2598708872.py:8, code: f_x = torch.sin(x)**2 + torch.cos(x)**2
        sin: f32[1000] = torch.ops.prims.sin.default(primals_1)
        mul: f32[1000] = torch.ops.prims.mul.default(sin, sin)
        cos: f32[1000] = torch.ops.prims.cos.default(primals_1)
        mul_1: f32[1000] = torch.ops.prims.mul.default(cos, cos)
        add: f32[1000] = torch.ops.prims.add.default(mul, mul_1);  mul = mul_1 = None

        # File: /tmp/ipykernel_2583/2598708872.py:9, code: return torch.nn.functional.mse_loss(f_x, y)
        sub: f32[1000] = torch.ops.prims.sub.default(add, primals_2)
        mul_2: f32[1000] = torch.ops.prims.mul.default(sub, sub);  sub = None
        sum_1: f32[] = torch.ops.prims.sum.default(mul_2, [0]);  mul_2 = None
        div: f32[] = torch.ops.prims.div.default(sum_1, 1000.0);  sum_1 = None
        return [div, sin, add, primals_1, primals_2, cos]
```

screenshot by author

Graph optimization: Layer and operator fusion and C++/GPU code generation

PyTorch technologies discussed: TorchInductor, OpenAI Triton (default) other compilers

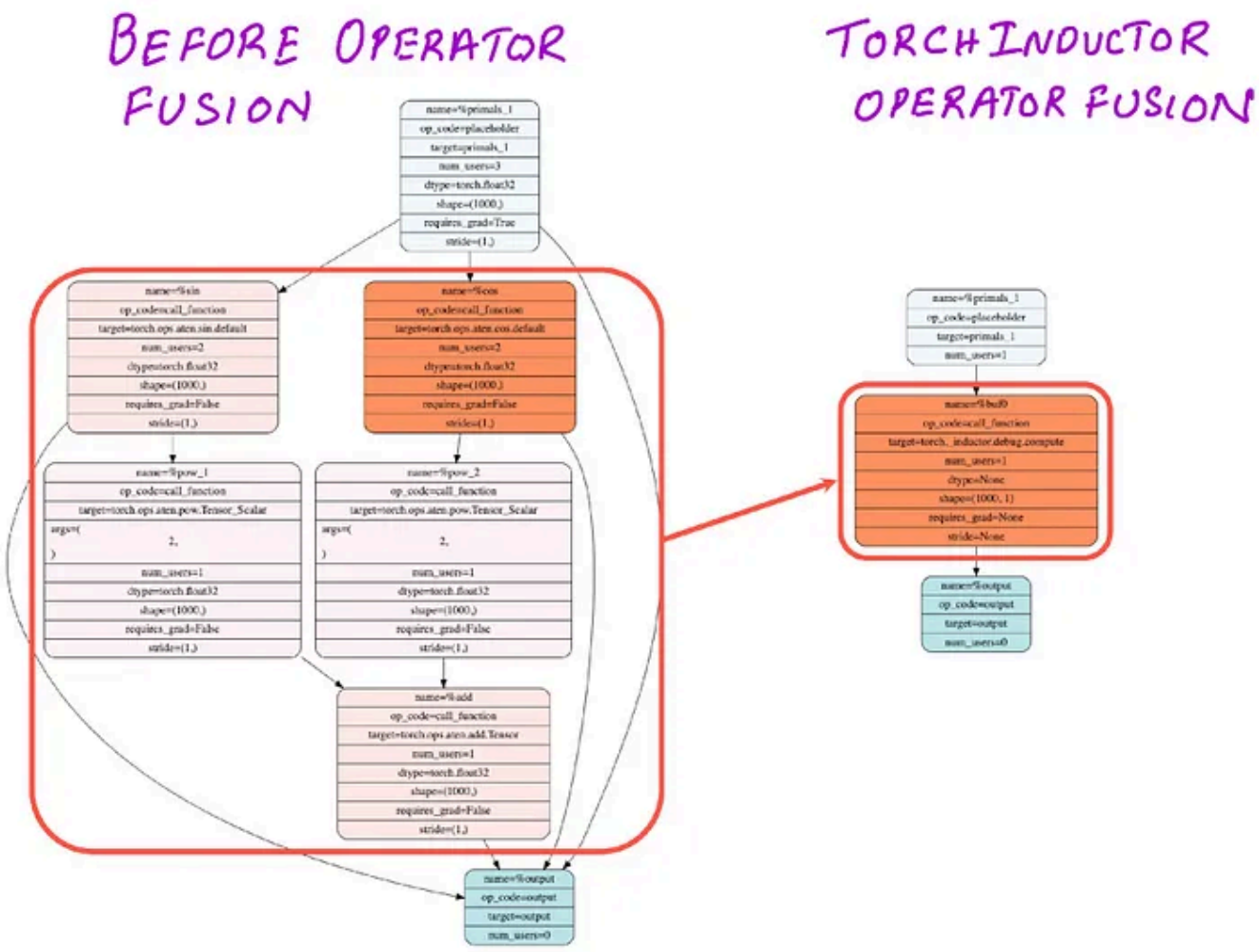
In this final section of the blog post we'll discuss operator fusion and automatic code generation for CPUs and GPUs using TorchInductor. First some basics:

What is a deep learning optimizing compiler?

An optimizing compiler for deep learning is good at finding performance gaps in code and addressing them by transforming the code to reduce code attributes such as memory access, kernel launches, data layout optimizations for a target backend. TorchInductor is the default optimizing compiler with torch.compile() that can generate optimized kernels for GPUs using OpenAI Triton and CPUs using OpenMP pragma directives.

What is operator fusion in deep learning?

Deep learning is composed of many fundamental operations such as matrix-matrix and matrix-vector multiplications. In PyTorch eager mode of execution each operation will result in separate function calls or kernel launches on hardware. This leads to CPU overhead of launching kernels and results in more memory reads and writes between kernel launches. A deep learning optimizing compiler like TorchInductor can fuse multiple operations into a single compound operator in python and generate low-level GPU kernels or C++/OpenMP code for it. This results in faster computation due to fewer kernel launches and fewer memory read/writes.



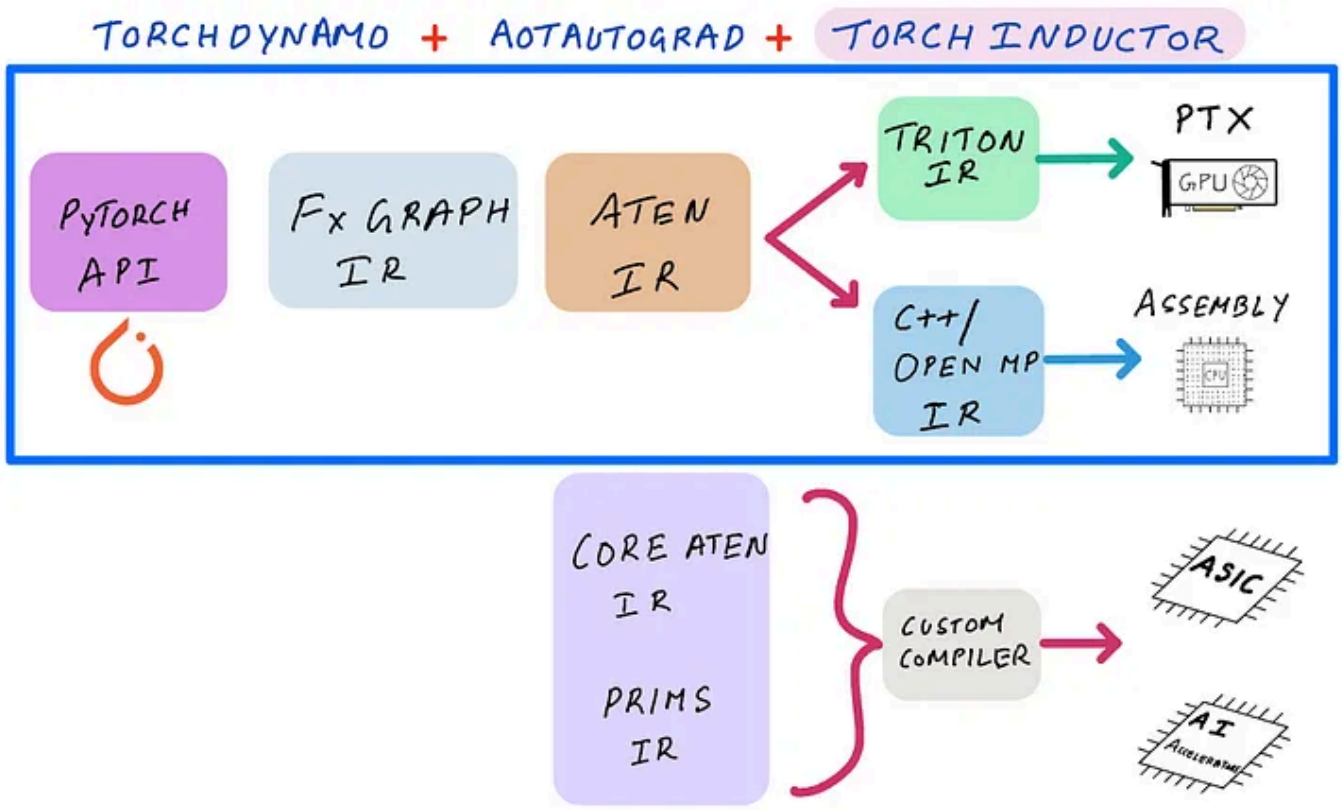
screenshot by author

The computational graph from the output of AOTAutograd in the previous section is composed of many Aten operators represented in an FX graph. TorchInductor optimizations doesn't change the underlying computation in the graph but merely restructures it with operator and layer fusion, and generates CPU or GPU code for it. Since TorchInductor can see the full forward and backward computational graph ahead of time, it can take decisions on out-of-order execution of operations that don't have dependence on each other, and maximize hardware resource utilization.

Under the hood, for GPU targets, TorchInductor uses OpenAI's Triton to generate fused GPU kernels. Triton itself is a separate Python based framework and compiler for writing optimized low-level GPU code which is

otherwise written in CUDA C/C++. But the only difference is that TorchInductor will generate Triton code which is compiled into low level PTX code.

For multi-core CPU targets, TorchInductor generates C++ code and injects OpenMP pragma directives to generate parallel kernels. From the PyTorch user level world view, this is the IR transformation flow:



Of course this being the high-level view, I'm omitting some details here and I encourage you to read the TorchInductor forum post and OpenAI's triton blog post for triton.

We'll now throw away our fake compiler which we used in our previous section, and use the full PyTorch compiler stack that uses TorchInductor.

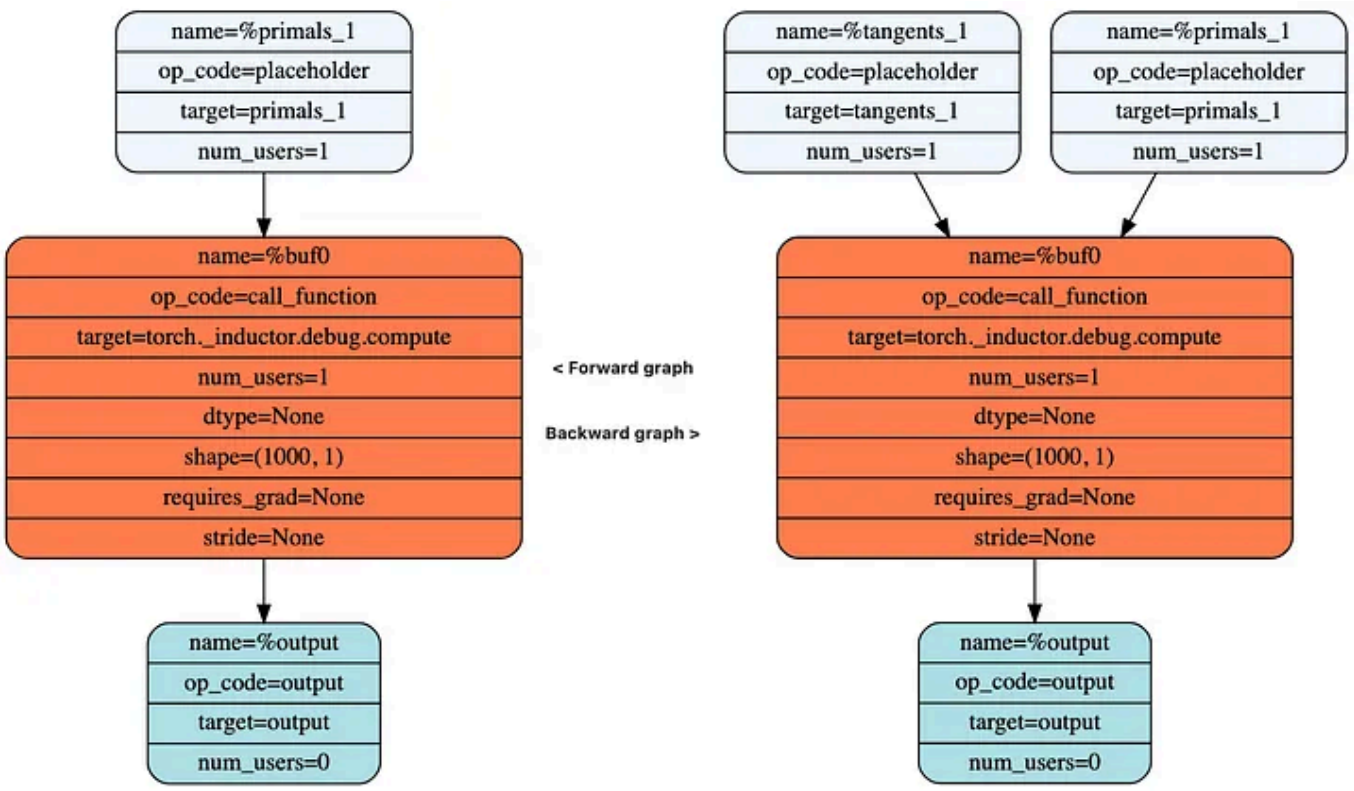
```python
def f(x):
    return torch.sin(x)**2 + torch.cos(x)**2

torch._dynamo.reset()
compiled_f = torch.compile(f, backend='inductor',
                           options={'trace.enabled':True,
                                    'trace.graph_diagram':True})

# device = 'cpu'
device = 'cuda'

torch.manual_seed(0)
x = torch.rand(1000, requires_grad=True).to(device)
y = torch.ones_like(x)

out = torch.nn.functional.mse_loss(compiled_f(x),y).backward()
```

torch_compile3.py hosted with ❤️ by GitHub view raw

Notice that I've passed optional argument that enables two debug features:

- `trace.enabled` : Generates intermediate code to inspect code generated by TorchInductor
- `trace.graph_enabled` : Generates the optimized computational graph visualization after operator fusion

For our simple example TorchInductor is able to fuse all intermediate operations in our function into a single custom operator, and you can see below how that simplifies the forward and backward computational graphs.



screenshot by author

You must surely be wondering what this fused operator looks like in code. The code for the fused operator is automatically generated by TorchInductor and it's in C++ or Triton based on the target device — CPU or GPU. You don't need to explicitly specify to TorchInductor which device to target, it can infer it from the data and model device type.

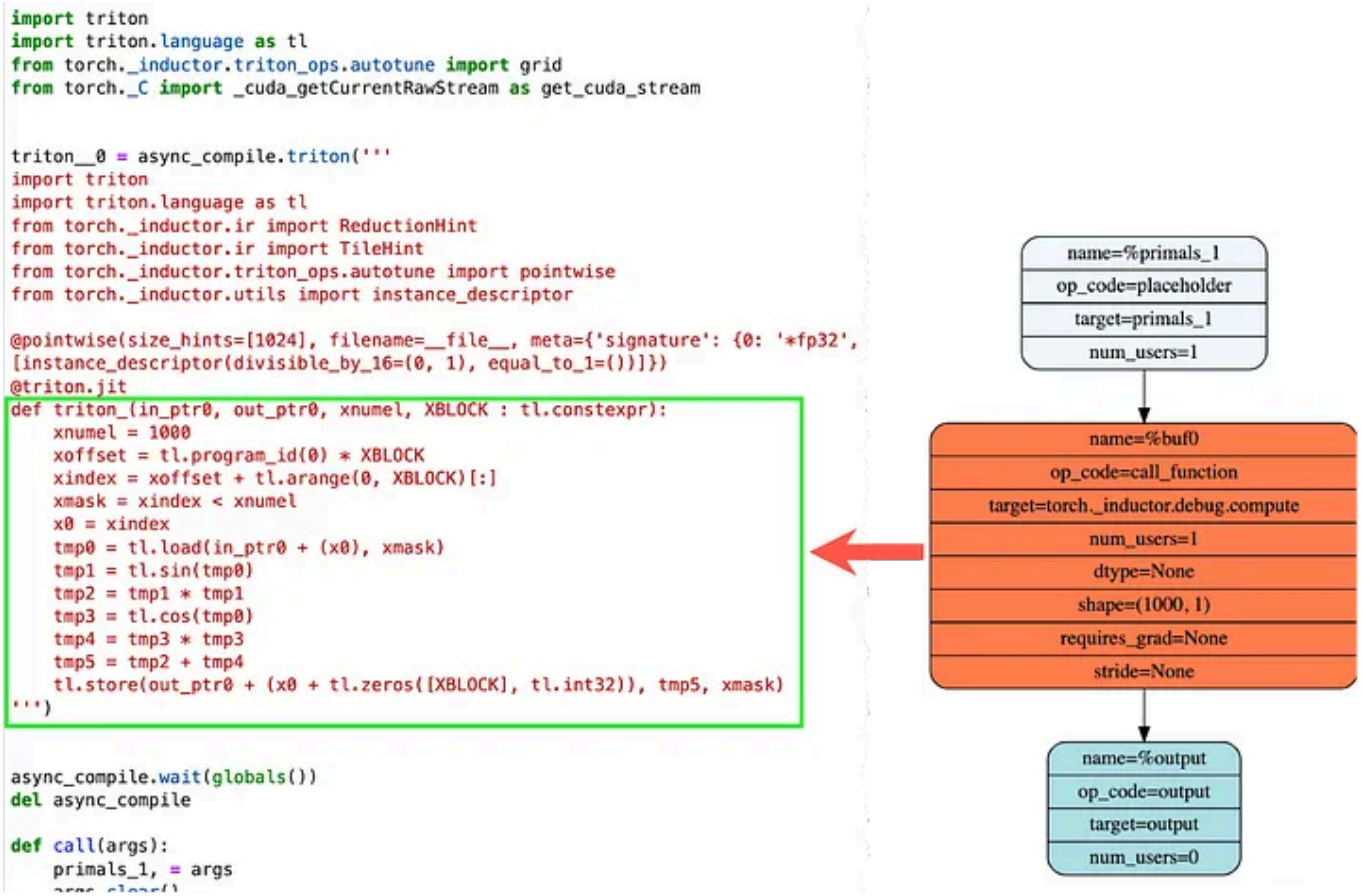
To view the generated code, you have to enable debugging using trace.enabled=True and this creates a director called torch_compile_debug with debug information.

The full path to forward and backward graph code are:

- `torch_compile_debug/run_<DATE_TIME_PID>/aot_torchinductor/model__XX_forward_XX/output_code.py`

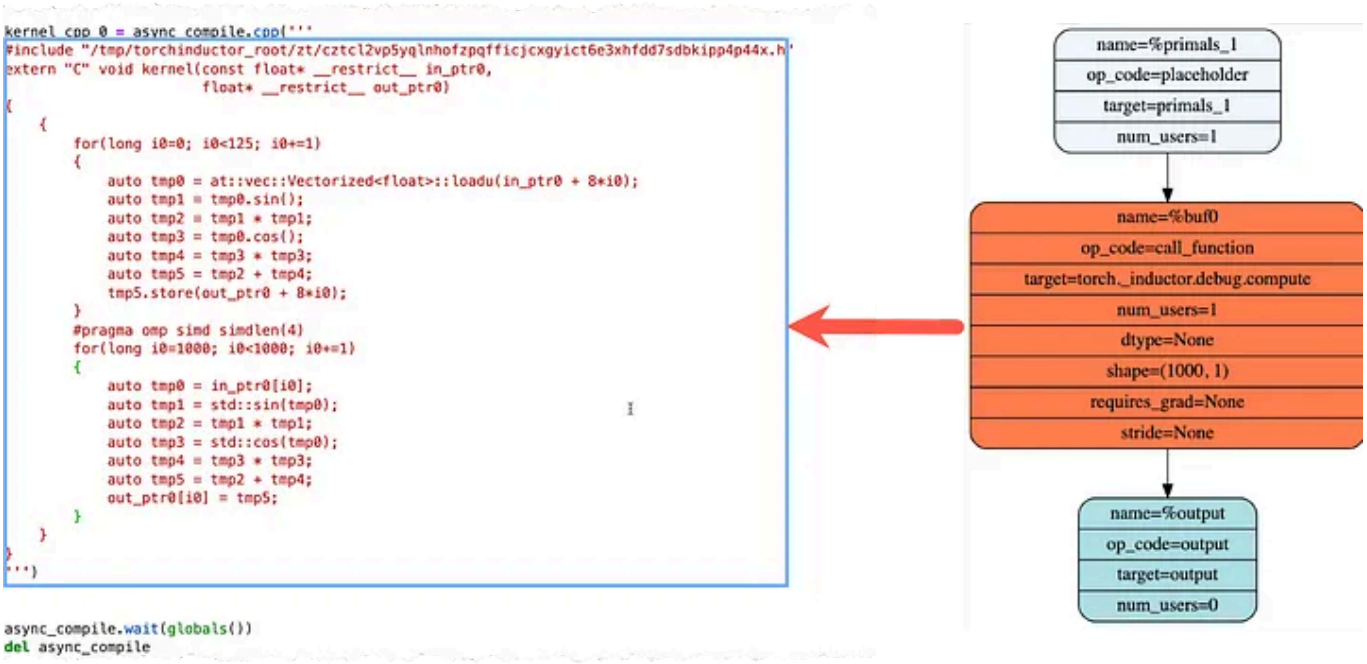
- `torch_compile_debug/run_<DATE_TIME_PID>/aot_torchinductor/model__XX_backward_XX/output_code.py`

If you set device = ‘cuda’ (assuming your computer has a GPU device) then the generated code in the forward folder is in Open AI Triton



screenshot by author

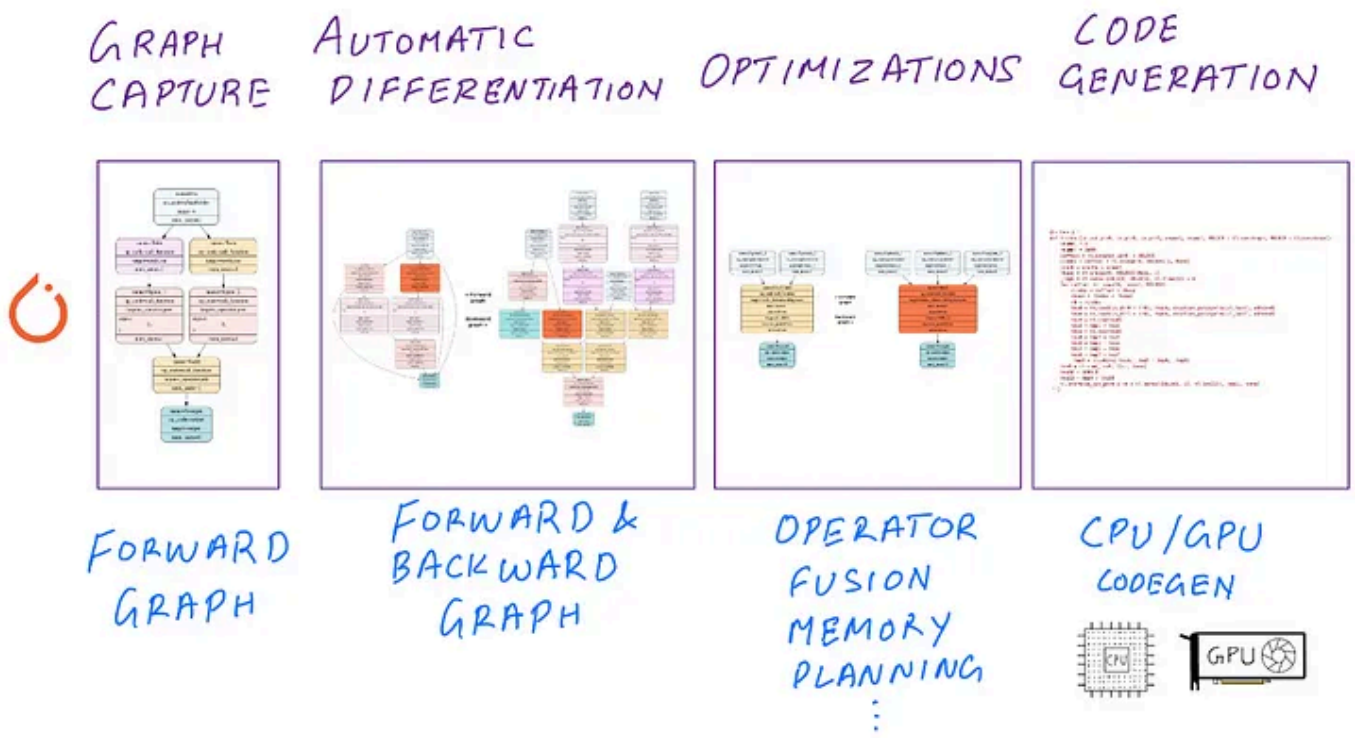
If you set device = ‘CPU’ then the generated code is in C++ with OpenMP pragmas



screenshot by author

Recap

Deep learning compilers are complex with intricate inner workings that rival a swiss watch. In this blog post, I hope I provided you with a gentle and easy to follow primer on this topic and how these technologies power PyTorch 2.0.



screenshot by author

1. I started with a simple PyTorch function
2. I showed how TorchDynamo captures the graph and represents it in FX IR
3. I showed how AOTAutograd generates the backward pass graph, lowers PyTorch operators into Aten operators and represents it in an FX graph container.
4. I discussed how Aten operators can be further decomposed into Core Aten IR and Prims IR that reduces the number of operators that other compilers can support without supporting the full PyTorch API list.
5. I showed how TorchInductor performs operator fusion and generates optimized code for CPU and GPU targets

If you followed along you should be able to provide a high-level response to the following questions:

- What is a deep learning compiler?
- What does a PyTorch 2.0 compiler do when you call torch.compile()?
- Why do we need a forward and backward pass graph ahead of time?
- What are the different intermediate representations (IR) in PyTorch
- What is the difference between ATen IR, Core ATen IR, Prims IR?
- What is operator fusion and why is it important?

Thank you for reading all the way to the end!

If you found this article interesting, consider following me on medium to be notified when I publish new articles. Please also check out my other blog posts on medium or follow me on twitter (@shshnkp), LinkedIn or leave a comment below. Want me to write on a specific topic I'd love to hear from you!

Pytorch | Deep Learning | Machine Learning | Gpu | Deep Dives

Written by Shashank Prasanna

885 Followers · Writer for Towards Data Science

Talking Engineer. Runner. Coffee Connoisseur. ML @ Modular. Formerly ML @Meta, AWS, NVIDIA, MATLAB, posts are my own opinions. website: shashankprasanna.com

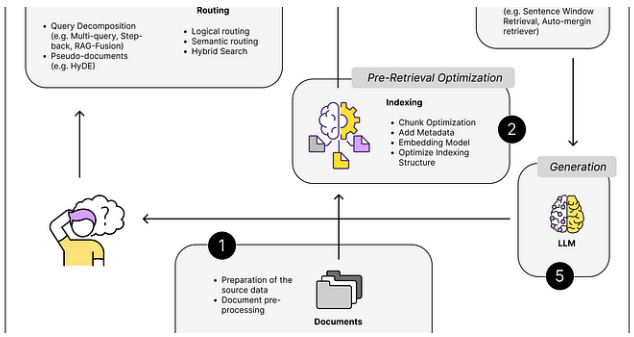
More from Shashank Prasanna and Towards Data Science

Shashank Prasanna in Towards Data Science

Choosing the right GPU for deep learning on AWS

How to choose the right Amazon EC2 GPU instance for deep learning training and...

Jul 25, 2020 · 1.4K · 16



Dominik Polzer in Towards Data Science

17 (Advanced) RAG Techniques to Turn Your LLM App Prototype into...

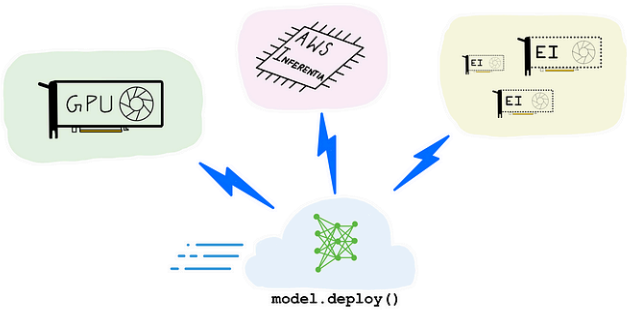
A collection of RAG techniques to help you develop your RAG app into something robus...

Jun 26 · 1.8K · 20

Mauro Di Pietro in Towards Data Science

GenAI with Python: RAG with LLM (Complete Tutorial)

Build your own ChatGPT with multimodal data and run it on your laptop without GPU



Shashank Prasanna in Towards Data Science

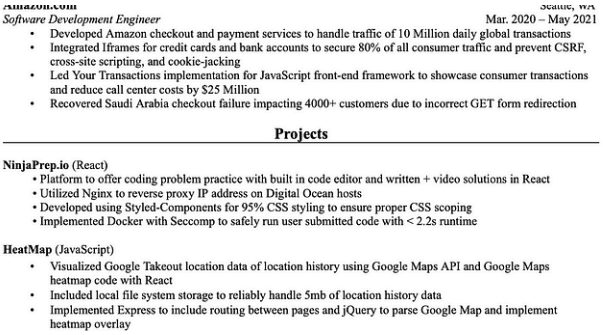
A complete guide to AI accelerators for deep learning...

Learn about CPUs, GPUs, AWS Inferentia, and Amazon Elastic Inference and how to choose...

Jun 28 705 12 Oct 21, 2020 541 4

See all from Shashank Prasanna See all from Towards Data Science

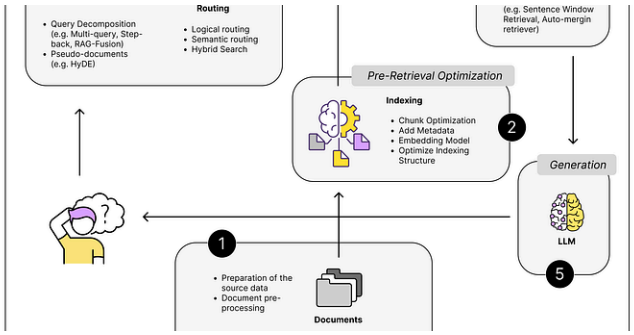
Recommended from Medium


Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021
- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay



Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

May 31 13.5K 199

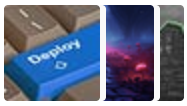
Dominik Polzer in Towards Data Science

17 (Advanced) RAG Techniques to Turn Your LLM App Prototype into...

A collection of RAG techniques to help you develop your RAG app into something robus...

Jun 26 1.8K 20

Lists

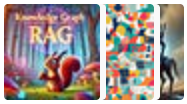


Predictive Modeling w/ Python

20 stories · 1381 saves

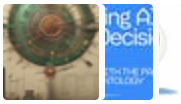
Practical Guides to Machine Learning

10 stories · 1669 saves



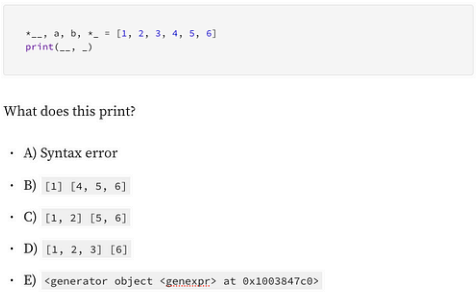
Natural Language Processing

1587 stories · 1133 saves



data science and AI

40 stories · 200 saves

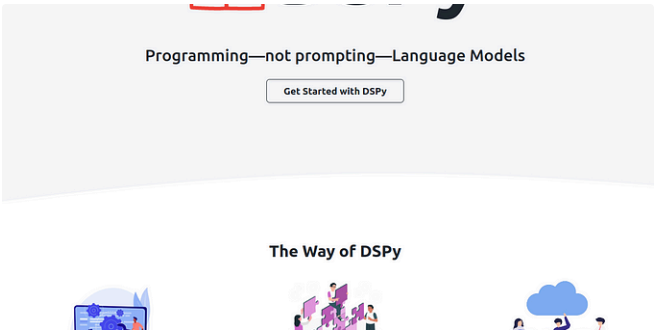


Liu Zuo Lin

You're Decent At Python If You Can Answer These 7 Questions...

No cheating pls!!

Mar 5 · 6.2K · 30

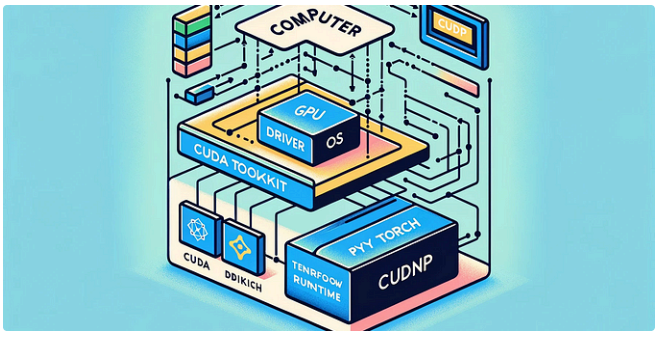


Vishal Rajput in AIGuys

Prompt Engineering Is Dead: DSPy Is New Paradigm For Prompting

DSPy Paradigm: Let's program—not prompt—LLMs

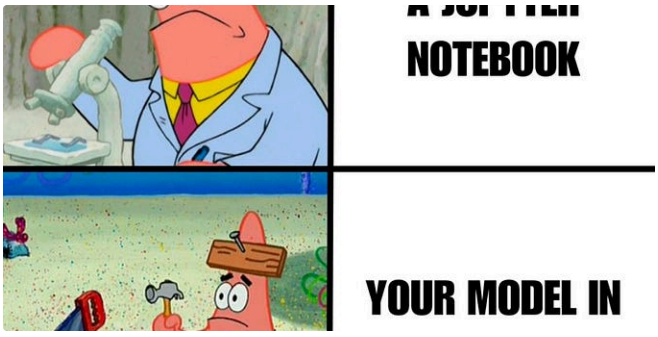
May 28 · 4K · 40



RickyYang

CUDA, CUDA Toolkit, and cuDNN: Clarifying the Confusion in One Go

If you're venturing into the world of Deep Learning, you'll inevitably come across...



Mandar Karhade, MD. PhD. in Towards AI

Why RAG Applications Fail in Production

It worked as a prototype; then all went down!

Jan 17 11

Mar 19 2.1K 23

See more recommendations