Unlocking faster models with torch.compile

Gaurav Jain

Who Am I?

- Sr. SWE AI/ML Kernels @ d-matrix.ai
- Leading the efforts on building the kernel SW stack for d-matrix HW
  - Unique in-memory compute data-flow architecture
- Experience across the HW/SW stack
  - Previously worked at Google as a TPU Silicon Architect
  - And before that as a Scientist at the Indian Space Research Organisation
- Outside work, I like to climb, run, and ski!

You can connect with me on LinkedIn @ gauravjain14

Outline

- Pytorch 2.0
- Eager vs Compiled-Mode
- Eager Mode
- Compiled-Mode - torch.compile
- Graph-Compilation is Hard

PyTorch 2.0

- Same Eager-mode development as PyTorch 1.x
- Fundamentally different at compiler level under the hood
  - Faster performance
  - Support for Dynamic Shapes and Distributed
- `**torch.compile**` - main API for PyTorch 2.0
- Many optimizations beyond `torch.compile` including
  - Support for training and inference using a custom kernel architecture for scaled dot product attention (SPDA) [To read more]

Eager vs Compiled-Mode

- One of the greatest debates among ML practitioners

Eager Mode

- Preferred by Users
- Easier to use programming model, debug, and quick experimentation

Compiled-Mode *Our focus today!*

- Graph Mode - Preferred by Framework and Backend Builders
- Easier to Optimize with a Compiler and Transformer frameworks

Eager Mode

- Easier to understand
  - More intuitive and interactive programming style
  - Allows for quick prototyping
- Code is executed immediately
- No Graph-level optimizations - execution has visibility only into immediate operators
  - No operator fusion, memory optimizations, etc.

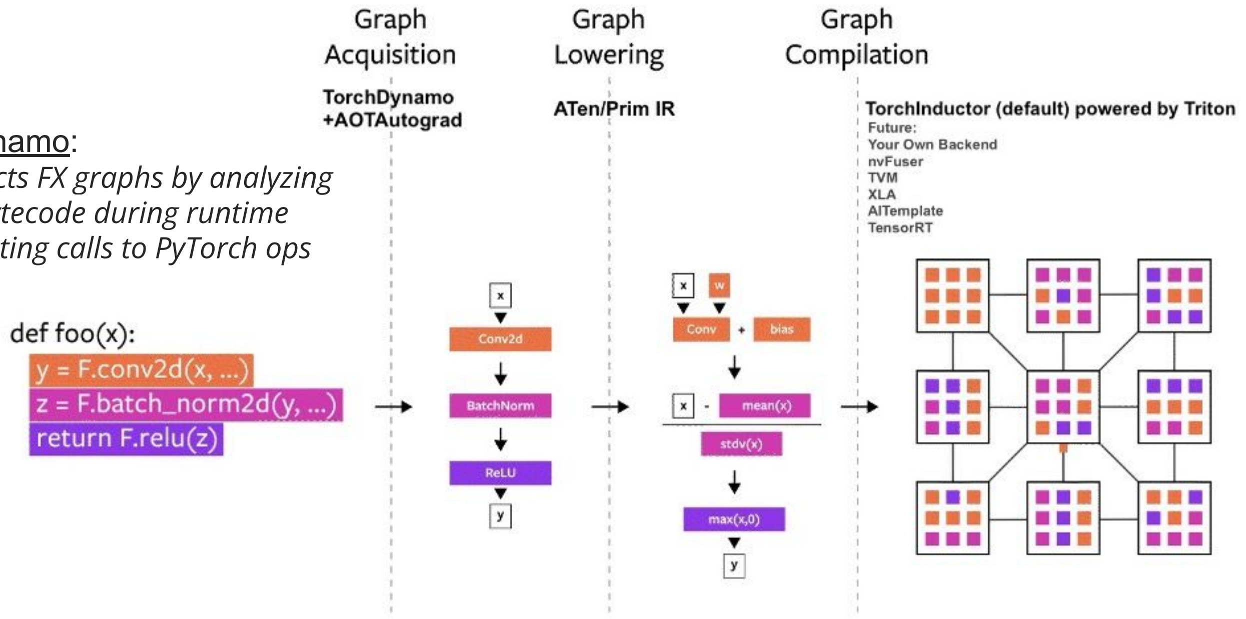
And most importantly

PyTorch users don’t want to build Static Graphs

Compiled Mode - *torch.compile*

- Speed up model execution using Just-in-time compilation

TorchDynamo:
(JIT) extracts FX graphs by analyzing Python bytecode during runtime and detecting calls to PyTorch ops

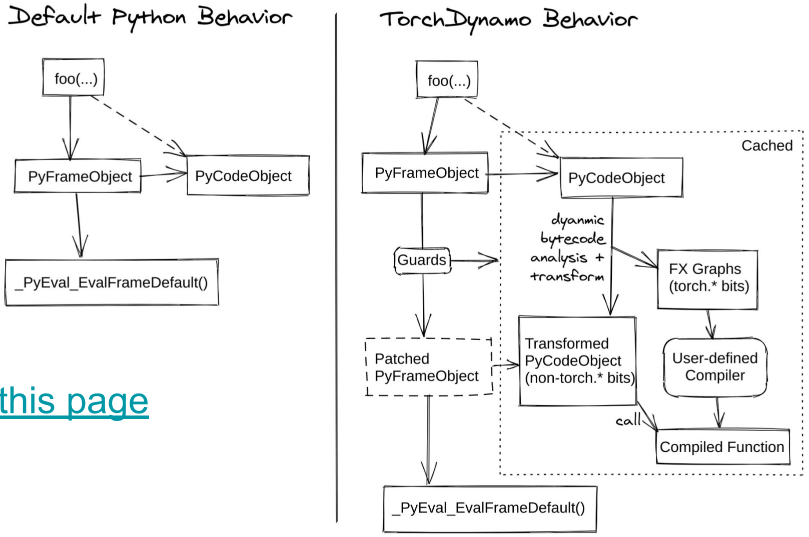


TorchInductor:
compiles the FX graphs into optimized kernels (for GPU backend).

src: https://pytorch.org/get-started/pytorch-2.0/

TorchDynamo - A deep-dive video

- Python-level JIT compiler to make PyTorch programs run faster
  - Without needing any code changes
- Traces a Python function with given inputs and records the operations that are executed



More details on this page

TorchInductor

- Compiles FX graphs into optimized C++/Triton Kernels
- Implemented in Python - increased developer velocity and productivity
- Core Infrastructure supports majority of PyTorch functions, including
  - Aliasing/Views
  - Scatter/Gather
  - Pooling/Windows/Reduction
  - Masked/Conditional Execution
  - Tiling
- TorchInductor Mode - [‘default’, ‘reduce-overhead’, max-autotune’]

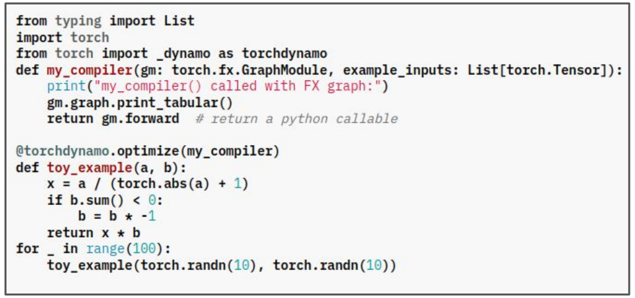
TorchInductor uses Define-By-Run Loop-Level IR

But Graph-compilation is Hard!

- Tensor Data structure manipulation - x.item(), x.tolist(), int(x)
- Code consists of various 3rd-party libraries - *numpy*, *xarray*
- Data-Dependent Python Control Flow
- Plenty use of Exceptions, Generators, Classes, etc.
- Dynamic Shapes - frequently changing tensor dimensions

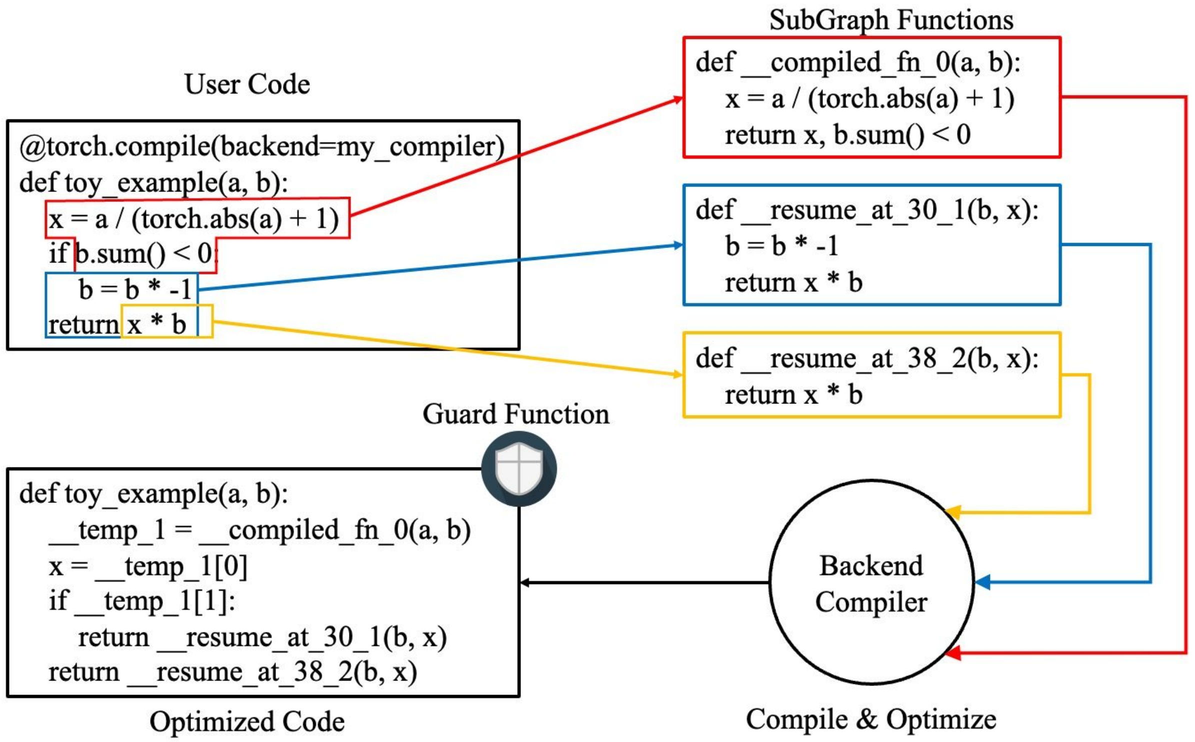
torch.compile can work through this all (almost!)

The Magic Behind!



```
from typing import List
import torch
from torch import _dynamo as torchdynamo
def my_compiler(gm: torch.fx.GraphModule, example_inputs: List[torch.Tensor]):
    print("my_compiler() called with FX graph:")
    gm.graph.print_tabular()
    return gm.forward  # return a python callable

@torchdynamo.optimize(my_compiler)
def toy_example(a, b):
    x = a / (torch.abs(a) + 1)
    if b.sum() < 0:
        b = b * -1
    return x * b
for _ in range(100):
    toy_example(torch.randn(10), torch.randn(10))
```

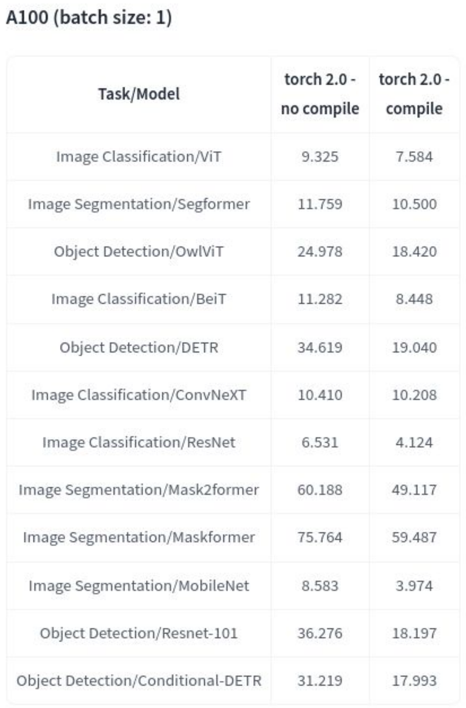


User Code
@torch.compile(backend=my_compiler)
def toy_example(a, b):
x = a / (torch.abs(a) + 1)
if b.sum() < 0:
b = b * -1
return x * b
SubGraph Functions
def __compiled_fn_0(a, b):
x = a / (torch.abs(a) + 1)
return x, b.sum() < 0
def __resume_at_30_1(b, x):
b = b * -1
return x * b
def __resume_at_38_2(b, x):
return x * b
Guard Function
def toy_example(a, b):
__temp_1 = __compiled_fn_0(a, b)
x = __temp_1[0]
if __temp_1[1]:
return __resume_at_30_1(b, x)
return __resume_at_38_2(b, x)
Optimized Code
Backend Compiler
Compile & Optimize

What about Training and Backprop?

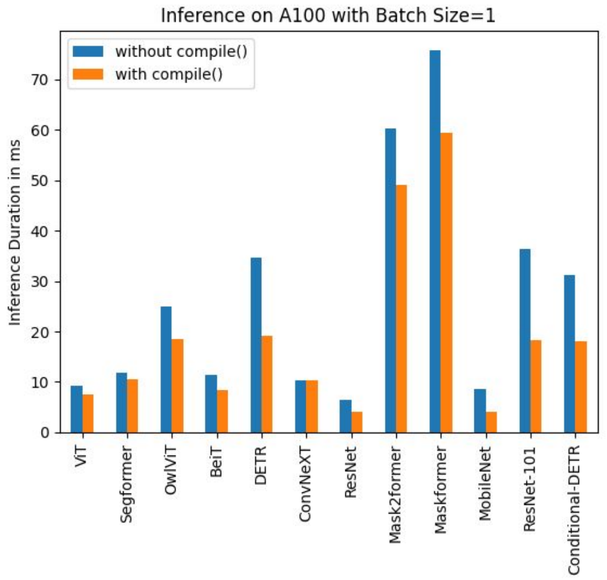
- Backward Propagation involves capturing the gradient
- **AutoGrad**: Captures backward graph for torch.compile
- Graph Breaks in forward → Graph Breaks in backward
- AOTDispatcher - calls autograd engine to compute gradients
- Loss calculation (loss = model(x).sum()) is executed by Python Interpreter
- loss.backward() calls into Compiled Autograd

Some Industry Benchmarking

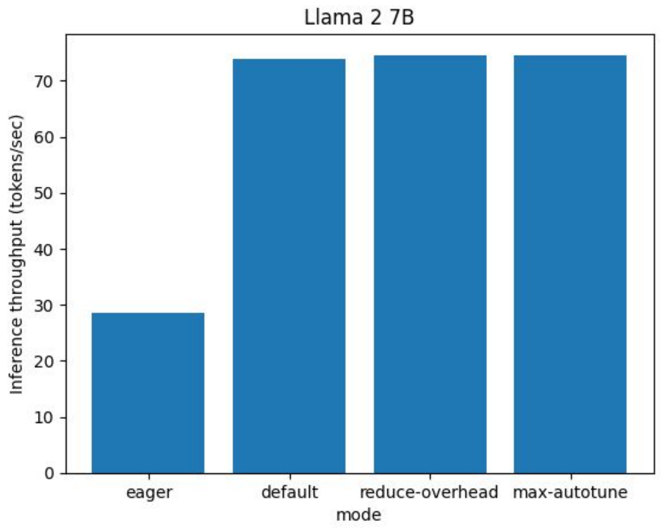


A100 (batch size: 1)

Task/Model	torch 2.0 - no compile	torch 2.0 - compile
Image Classification/ViT	9.325	7.584
Image Segmentation/Segformer	11.759	10.500
Object Detection/OwlViT	24.978	18.420
Image Classification/BeiT	11.282	8.448
Object Detection/DETR	34.619	19.040
Image Classification/ConvNeXT	10.410	10.208
Image Classification/ResNet	6.531	4.124
Image Segmentation/Mask2former	60.188	49.117
Image Segmentation/Maskformer	75.764	59.487
Image Segmentation/MobileNet	8.583	3.974
Object Detection/Resnet-101	36.276	18.197
Object Detection/Conditional-DETR	31.219	17.993



HuggingFace Perf Benchmarking



Speed-up gained for Llama 2 7B - https://rocm.blogs.amd.com/artificial-intelligence/torch_compile/README.html

Referencesw

- TorchInductor
- TorchCompiler
- Torch.compile - An Introduction
- PyTorch 2.0 - ASPLOS 2024 Tutorial
- PyTorch Autograd Explained
- Torch.compile, the missing manual
- Huggingface torch.compile Benchmarking
- Torch_Compiler_Profling

Thank You

Questions?

Backup

Why was TorchDynamo needed?

Record and Replay

Works for simple programs but not for complex semantics, data flow dependency,

Lazy evaluation - high run-time overhead, latency to kernel launches

Come in TorchDynamo, TorchInductor

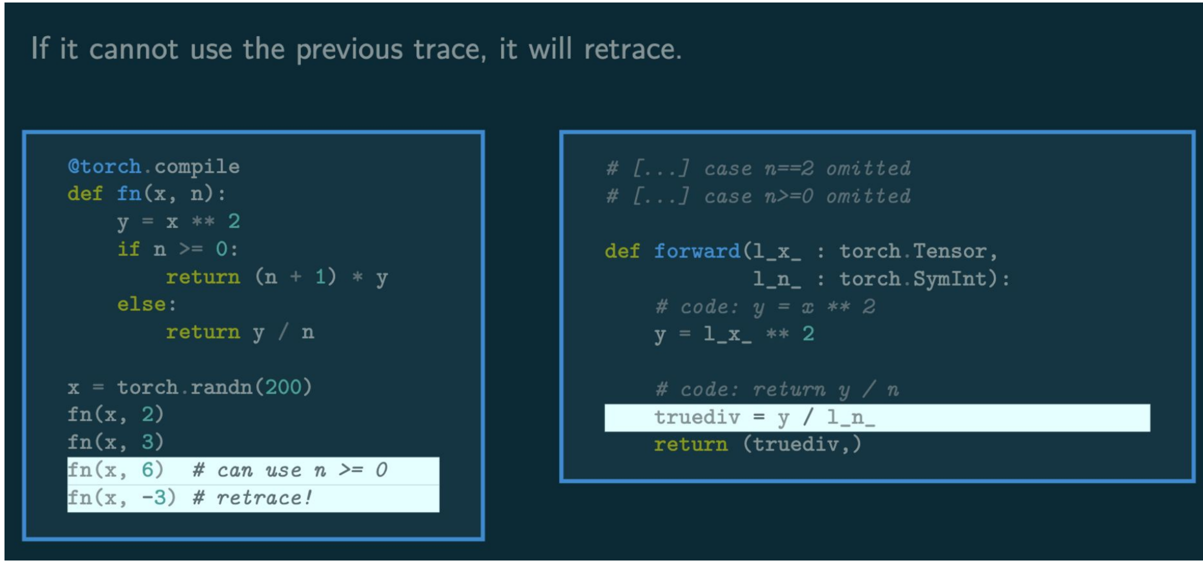
Why was TorchDynamo needed?

- Previously existing stack worked for simple programs
  - Not for complex semantics, data flow dependency, etc.
- Lazy Evaluation for Dynamic Shapes/Data-dependent Control Flow
  - High Run-time overhead
  - Huge latency for kernel launches

TorchDynamo

VERY IMPORTANT -

If Dynamo Cannot use the previous trace, it will retrace → Data-dependent control flow



If it cannot use the previous trace, it will retrace.

```
@torch.compile
def fn(x, n):
    y = x ** 2
    if n >= 0:
        return (n + 1) * y
    else:
        return y / n

x = torch.randn(200)
fn(x, 2)
fn(x, 3)
fn(x, 6)  # can use n >= 0
fn(x, -3) # retrace!
```

```
# [...] case n==2 omitted
# [...] case n>=0 omitted

def forward(l_x_ : torch.Tensor,
            l_n_ : torch.SymInt):
    # code: y = x ** 2
    y = l_x_ ** 2

    # code: return y / n
    truediv = y / l_n_
    return (truediv,)
```

TorchDynamo and TorchInductor

- TorchDynamo captures PyTorch programs safely using Python Frame Evaluation Hook
- TorchInductor - compiler that generates fast code for multiple accelerators and backends
  - Uses OpenAI Triton as a key building block for Nvidia and AMD GPUs
- More details available at https://pytorch.org/blog/pytorch-2.0-release/

TorchDynamo and TorchInductor

two open source extensions to PyTorch: TorchDynamo and TorchInductor. These extensions are behind the torch.compile feature introduced in PyTorch 2 and officially released in March 2023. TorchDynamo is a Python-level JIT compiler designed to allow graph compilation in PyTorch programs while retaining the full flexibility of Python. TorchDynamo hooks into the Python frame evaluation API [9] in CPython to dynamically modify Python bytecode right before it is executed. It rewrites Python bytecode in order to extract sequences of PyTorch operations into an FX graph [34] which is then just-in-time compiled with many extensible backends. It creates this FX graph through bytecode analysis and is designed to generate smaller graph fragments that can be mixed with Python execution to get the best of both worlds: usability and performance

TorchInductor is the compiler backend for TorchDynamo. Once you have the FX graph, it uses OpenAI's Triton to generate the GPU kernels

Very simple to invoke the compiler

torch.compile

TorchDynamo

TorchDynamo → creates a PyFrameObject

Checks if the frame has been previous compiled, if yes, execute the guard function

If guard is false, caching doesn’t help → recompile

Generate new Python bytecode. This new bytecode will: 1) call the compiled FX graph; 2) store and reconstruct the local/stack state; 3) perform side effects the original function should have had, see Section 3.7; 4) either return or implement a graph break by falling back to the original bytecode and calling the generated continuation function(s). • Install the generated Python bytecode and guard function in the cache, run the generated bytecode with _PyEval_EvalFrameDefault, and return.

Guards

Used to check dynamic properties

Checking tensor properties, attributes, instances, etc.

Symbolic evaluation of the functions

  Passing inputs in symbolic states and recording any changes

If graph breaks or change in behavior, TorchDynamo rolls back to the symbolic state and generates a graph break

For control flow based on the type, size, and shape of tensors, TorchDynamo will guard on those properties and remove the control flow. In less common cases where there is control flow that cannot be removed (for example, branching on the value of a tensor rather than the metadata), TorchDynamo will generate a graph break that will trigger the branch bytecode to run in CPython, and analysis will resume after the jump. Anoth

TorchInductor Scheduler

Measurements show TorchInductor produces faster code on average than six other PyTorch compiler backends. Performance comparisons include both training and inference, CPU and GPU, float32 and float16, and three large benchmark suites containing 180+ full-sized models taken from real-world applications.

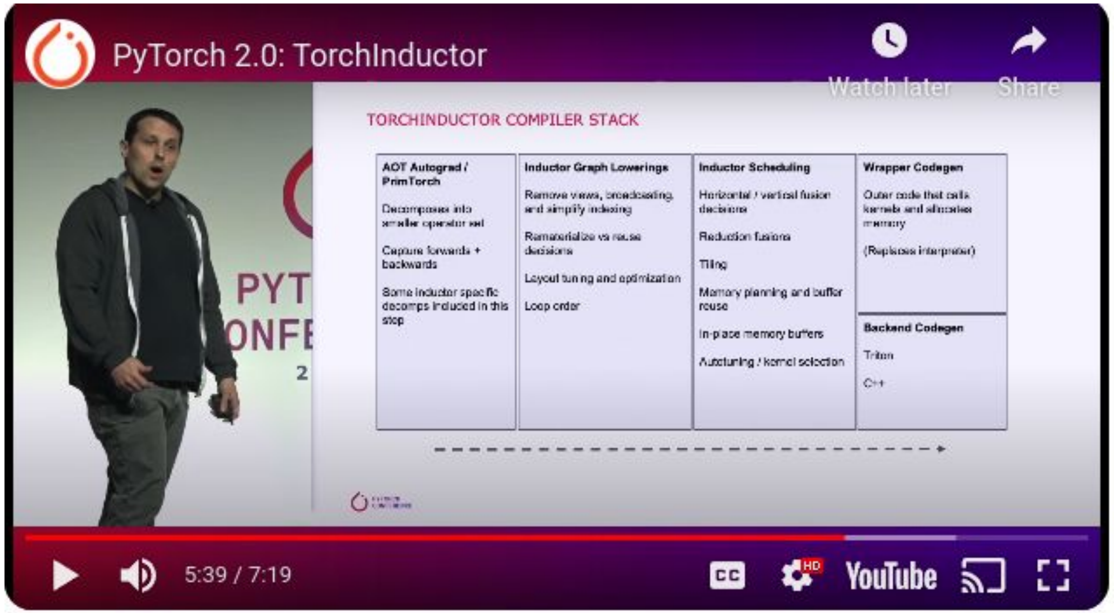
4.3 Lowerings and Define-By-Run Loop-Level IR The next phase of compilation is lowering from an FX graph of PyTorch operations into TorchInductor's define-by-run IR. A define-by-run IR means the IR uses executable Python code to define the bodies of loops, giving TorchInductor's IR much of the power of full Python, removing the need for a large amount of boilerplate, and allowing lowerings to be written concisely

TorchInductor Scheduler

The scheduling phase of TorchInductor determines which operators get fused, what order kernels run in, and does memory planning for buffer removal and/or reuse. Scheduling starts by converting every buffer in the IR into a subclass of BaseSchedulerNode. SchedulerNode represents a standard kernel that TorchInductor will codegen the body of. ExternKernelSchedulerNode represents calls to library code or user-defined kernels. Additionally, NopKernelSchedulerNode maps to nothing, but is used to add dependency edges to ensure the ordering of kernels (for example, a concatenate kernel which has been handled by making producers write directly to the combined buffer). Finally, a FusedSchedulerNode represents a set of two or more SchedulerNodes fused into a single kernel. Next, the scheduler converts the memory read/write sets of each kernel into dependency edges between nodes. Dependency edges are annotated with the symbolic memory address being read. Symbolic memory addresses are important in determining which fusions are legal. For example, if one kernel writes buf0 in forwards order, but a consumer reads in reverse order (using ops.load("buf0", s0 - 1 - i0)), then those nodes cannot be fused.

Scheduler.can_fuse(node1, node2) returns True if two nodes can be fused together. This checks dependency edges, and also checks many other properties to ensure correctness of a fusion. There are some heuristics here as well, for example, if config.aggressive_fusion=False, then can_fuse will prevent fusion of nodes that do not share any common memory accesses. There is also backend specific logic here, for example, TorchInductor supports reduction-broadcast-reduction fusions for Triton but not C++. • Scheduler.score_fusion(node1, node2) is used to order different fusion possibilities. Some fusions are mutually exclusive, so TorchInductor chooses the one with the higher score. The fusion score orders fusions by: 1) the category of the fusion (e.g. pointwise/reduction/template); 2) estimated bytes of memory traffic saved by the fusion; and 3) sh

Performance comparison



TORCHINDUCTOR COMPILER STACK

AOT Autograd / PrimTorch

Decomposes into smaller operator set

Capture forwards + backwards

Some inductor specific decomps included in this step

Inductor Graph Lowerings

Remove views, broadcasting, and simplify indexing

Rematerialize vs reuse decisions

Layout tuning and optimization

Loop order

Inductor Scheduling

Horizontal / vertical fusion decisions

Reduction fusions

Tiling

Memory planning and buffer reuse

In-place memory buffers

Autotuning / kernel selection

Wrapper Codegen

Outer code that calls kernels and allocates memory

(Replaces interpreter)

Backend Codegen

Triton

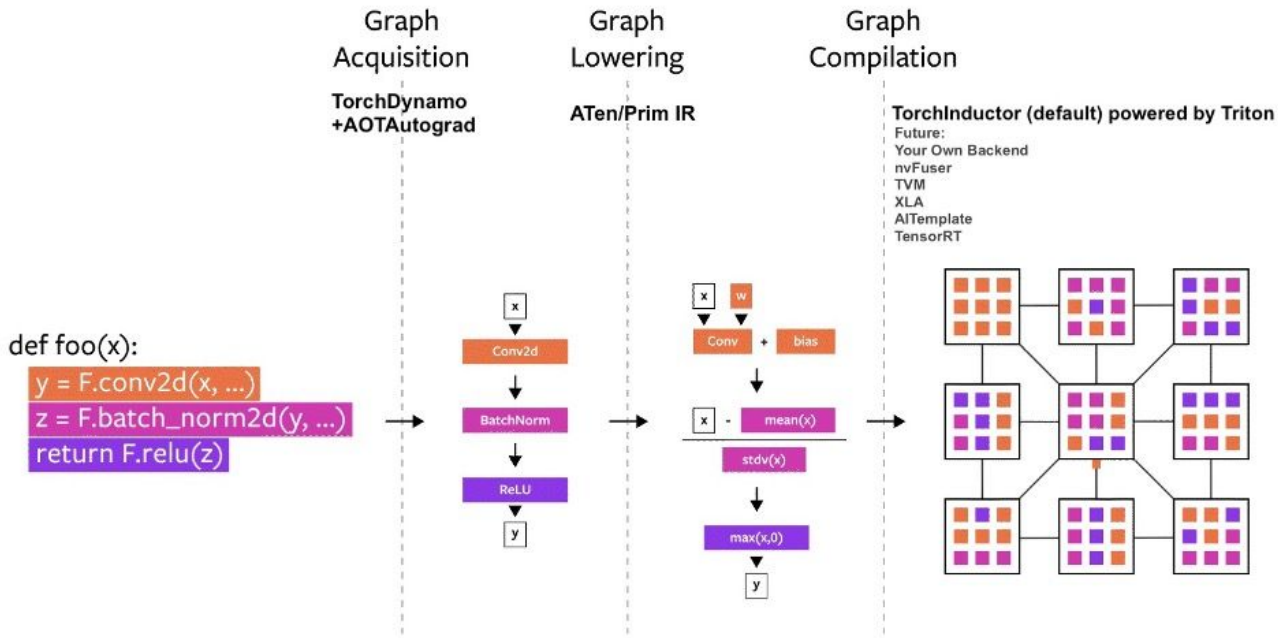
C++

Compiler parts

Graph acquisition

Graph lowering

Graph compilation



Graph acquisition - then and now

Torch.jit.trace, torchscript, fx tracing, lazy tensors

Can graph acquisition be made as little intrusive as possible? - **TorchDynamo**

Uses CPython - negligible overhead, no changes to the code

TorchInductor

Pythonic define-by-run IR → automatically map PyTorch models to generated Triton Code

Inductor core loop level IR contains 50 operators → implemented in Python

TORCH COMPILE

```
def torch.compile(model: Callable,

  *,

  mode: Optional[str] = "default",

  dynamic: bool = False,

  fullgraph:bool = False,

  backend: Union[str, Callable] = "inductor",

  # advanced backend options go here as kwargs

  **kwargs

) ->
```

Define-by-Loop IR:

- **Computation Graph**: In define-by-loop IR, although the model is still defined using Python control flow (like `for` loops or `if` conditions), **a computation graph is built dynamically at runtime**. This graph represents the operations and data flow across the model. While the model is being executed, the graph is constructed incrementally, which gives it a hybrid nature—some aspects of dynamic execution combined with the benefits of graph-based execution.
- **Graph Construction**: As the model's forward pass runs, the framework builds a graph under the hood, tracking operations and dependencies between tensors. This means that, while it might feel like eager execution, there is actually a graph being constructed that can be used for optimizations later.
- **Optimizations**: Once the graph is built, optimizations like **operator fusion**, **constant folding**, or other low-level transformations can be applied. This is what frameworks like PyTorch 2.0 (with `torch.compile`) or TensorFlow's XLA aim to do—**compile** the dynamically built graph into a more optimized, static representation that runs efficiently. This means that define-by-loop IR offers an opportunity for graph-level optimizations after the initial dynamic graph has been constructed.
- **Balance of Flexibility and Performance**: Define-by-loop strikes a balance by allowing the flexibility of dynamic graph construction (based on control flow) while still giving the framework the chance to optimize the execution. You get the best of both worlds: dynamic graph creation and optimizations typically associated with static graphs.

Handling Dynamic Shapes

Fake Tensor

run tensor operations to understand what output sizes/dtypes/devices are, without actually running those operations (or trashing preexisting tensors), which would be slower (if you’re doing a lot of compute) and take a lot of memory (it’s bad if your compiler needs to use GPU memory while you are compiling the program).