PyTorch 2.0

2023-08-27

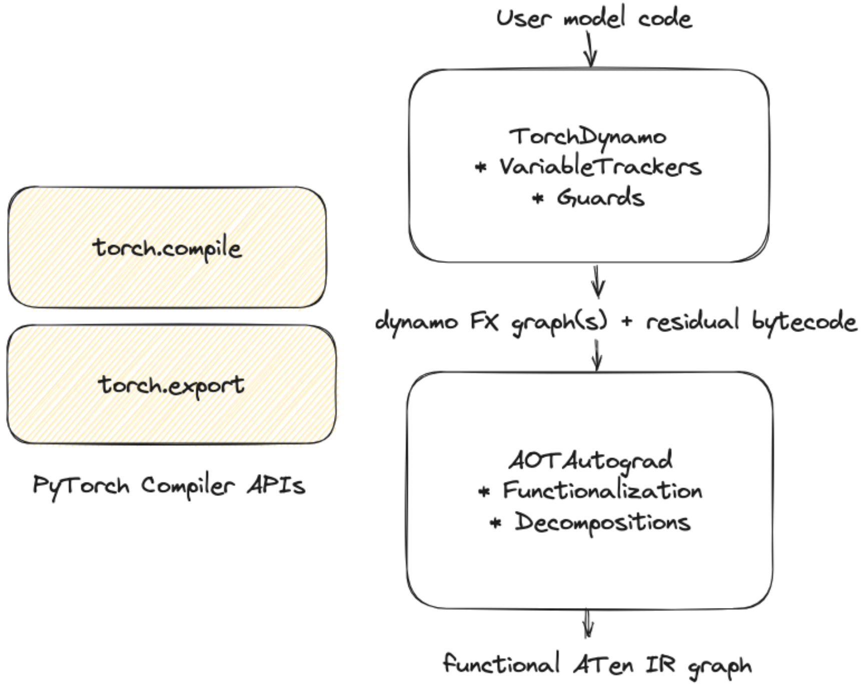
Agenda

1. Compiler frontend UX
2. Deep dive
   a. TorchDynamo
   b. AOTAutograd
   c. TorchInductor
3. PT2 for New Hardware Backends

Compiler Frontend UX

- torch.compile
  - Goal - Make PyTorch faster w/o sacrificing eager user experience
  - UX - add torch.compile(model) to your model script
  - Captures **accelerable regions/graphs** in your program (TorchDynamo, AOTAutograd)
  - Generates high performance machine code (TorchInductor)
- torch.export
  - Goal - Maximize performance and portability by extracting full graph from a model with some trade-off in user experience.
  - UX - Call torch.export(model) **Ahead-Of-Time** to generate a stand-alone artifact that can be executed without Python.
  - Captures full graph in your program through TorchDynamo and AOTAutograd too

PyTorch Compiler Frontend



TorchDynamo

The dream: “Just add torch.compile!”

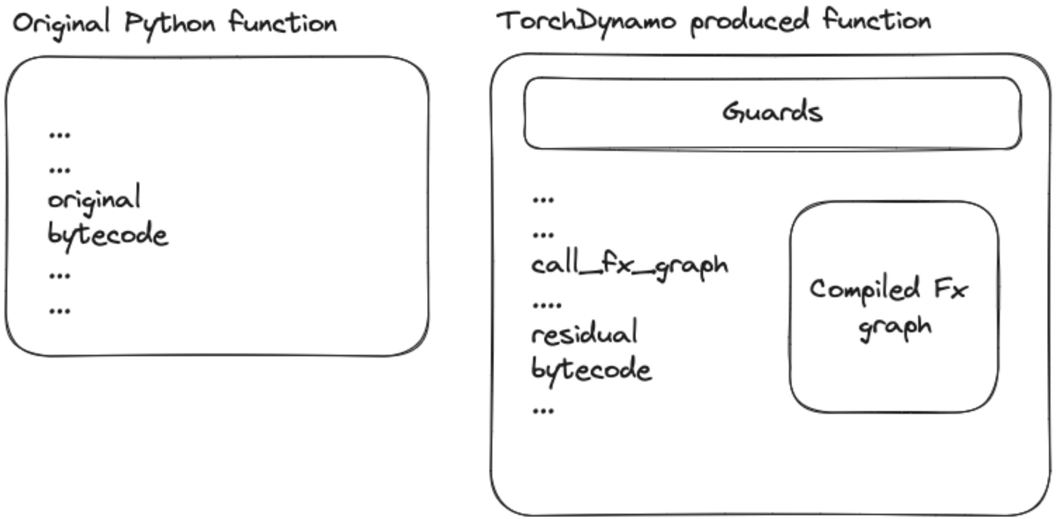
Compare w/ TorchScript: you must TSify your model to run it

PyTorch eager’s charm is its flexibility:

- Converting tensor into native Python types (x.item(), x.tolist())
- Using other frameworks (numpy/xarray etc)
- Exceptions, closures, generators etc Python constructs

TorchDynamo is designed so that we don’t need 100% feature coverage: unsupported features can transparently fallback to eager (graph break)

TorchDynamo Overview



TorchDynamo interposes on frame evaluation in an *observationally equivalent* way

- Extracts FX graph(s)
- Optimized bytecode - calls extracted graphs and bytecode for stuff outside graphs
- Guards - checks on the input conditions for which the graph is valid to use

TorchDynamo Bytecode Analysis - VariableTrackers

- TorchDynamo symbolically evaluates Python bytecode
- Each Python object is tracked by a Variable Tracker
  - `torch.* ops - TorchVariable`
  - `torch.tensor - TensorVariable`
  - `Python builtin variables - BuiltInVariable`
  - `Python lists/dicts - ListVariable, DictVariable`
- Operations on a TensorVariable adds a FX node in the graph

```
LOAD_GLOBAL torch []
LOAD_ATTR clamp_min [TorchVariable(<module 'torch' from '/scratch/anijain/work/pytorch/torch/__init__.py'>)]
LOAD_FAST a [TorchVariable(<built-in method clamp_min of type object at 0x7f258f1d2b80>)]
LOAD_FAST b [TorchVariable(<built-in method clamp_min of type object at 0x7f258f1d2b80>), TensorVariable()]
CALL_FUNCTION 2 [TorchVariable(<built-in method clamp_min of type object at 0x7f258f1d2b80>), TensorVariable(), ConstantVariable(int)]
LOAD_CONST 3 [TensorVariable()]
BINARY_MULTIPLY None [TensorVariable(), ConstantVariable(int)]
RETURN_VALUE None [TensorVariable()]
```

TorchDynamo Symbolic Evaluation of Python Bytecode

TorchDynamo Guards and Cache

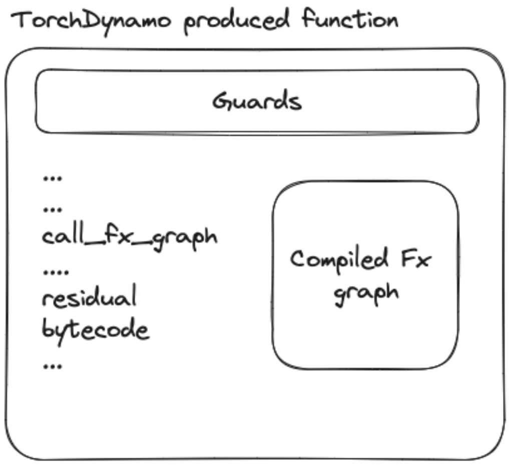
- Guards - set of conditions observed during JIT compilation
  - TorchDynamo produced graph is specialized for these conditions
- Compilation unit - Optimized bytecode (including compiled graph) and associated guards
- TorchDynamo cache - Linked list of compilation units per frame object
- Recompilation happens if none of cached guards satisfy the new input conditions

Graph Capture Example

```python
@torch.compile()
def func(a, b):
    return torch.clamp_min(a, b) * 3

p = torch.tensor([0.4, -0.2], requires_grad=True, device='cuda')
loss = func(p, 0).sum()
loss.backward()
print(p.grad)
```

TorchDynamo Extracted Graph



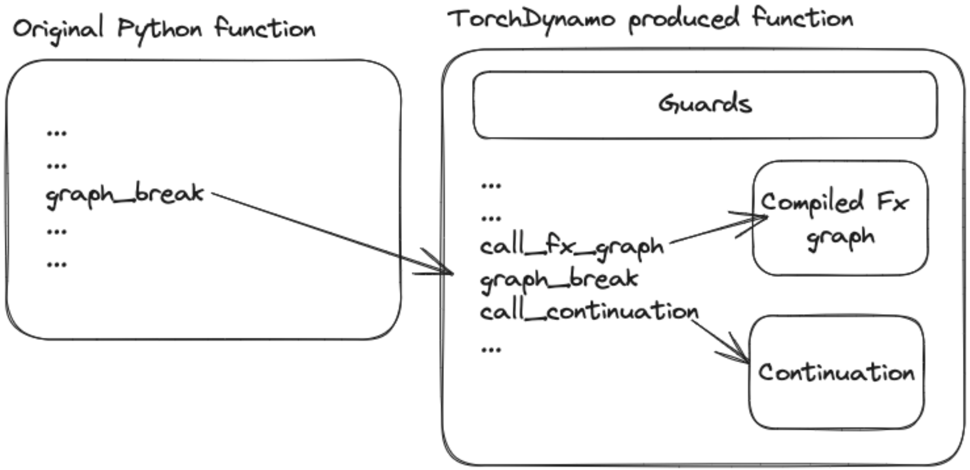

```
Guards:
  ___check_tensor(L['a'])
  ___check_type_id(L['b'], 7596096)
  L['b'] == 0
```

```
Optimized bytecode:
0 LOAD_GLOBAL              2 (__compiled_fn_0)
2 LOAD_FAST                0 (a)
4 CALL_FUNCTION            1
6 UNPACK_SEQUENCE          1
8 RETURN_VALUE
```

```
Compiled Fx Graph:
def forward(self, L_a_ : torch.Tensor):
  l_a_ = L_a_
  clamp_min = torch.clamp_min(l_a_, 0)
  mul = clamp_min * 3
  return (mul,)
```

TorchDynamo Bytecode Analysis - Graph Break

- On encountering an unsupported Python construct
  - Pre-graph-break-bytecode - Convert it to FX graph call + residual bytecode
  - Post-graph-break-bytecode - Wrap it in a new function object
  - New bytecode - Call FX graph + residual bytecode + continuation
- TorchDynamo is called again on the continuation function at its invocation
- Graph breaks can be expensive because of guards



AOTAutograd

Why AOTAutograd? Calling convention

Naive picture:

- FX graph -> Compiler -> Compiled callable
- FX graph takes a bunch of Tensors and produces Tensors

Problem:

- PyTorch eager’s calling convention is much more complicated!
- Aliasing: Input tensors may be aliased, mutations must reflect to all aliases
- Autograd: Produced tensors must have grad_fn, which can be backward()’ed through
- Subclass: Input/output tensors might be subclasses

Solution: AOTAutograd deals with the complicated interactions, so inner backends don’t have to

Why AOTAutograd? Graph normalization

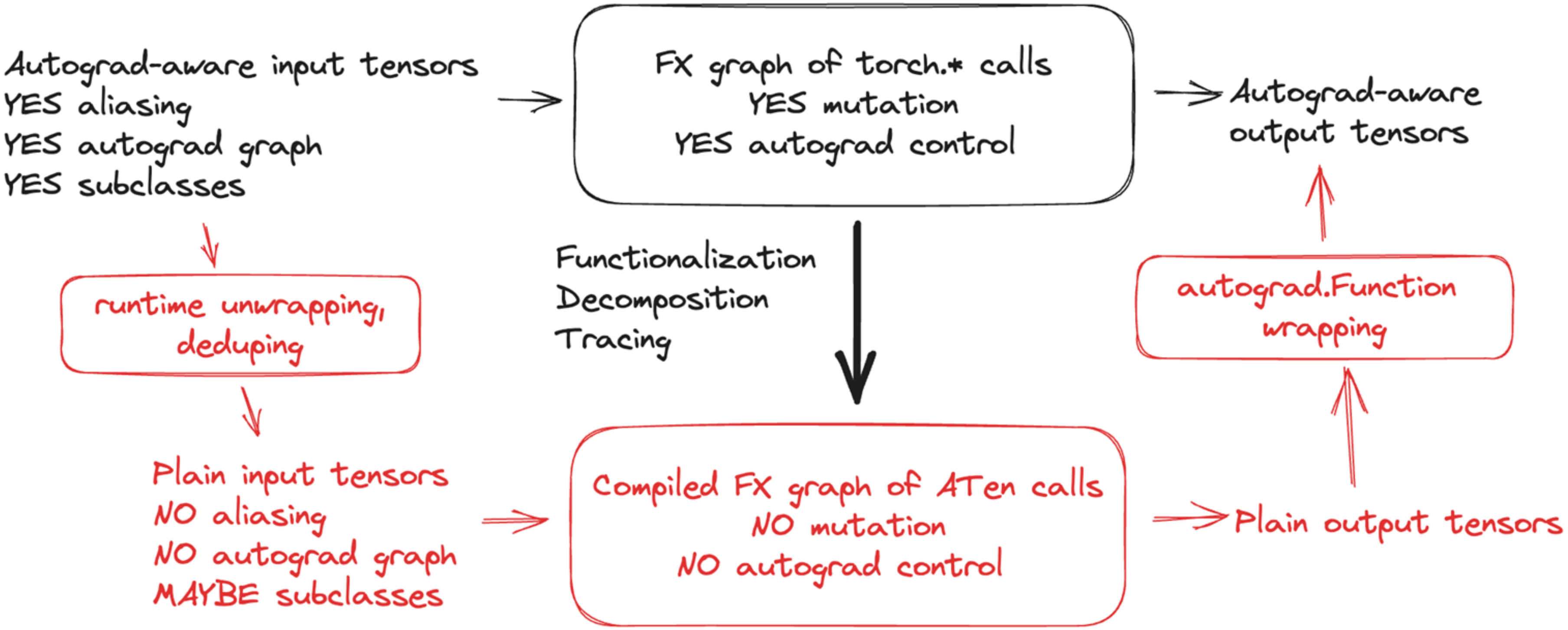
Dynamo produced FX graph: torch.* calls, looks like Python

Problem:

- torch.* argument resolution is nontrivial (default arguments, overload matching)
- Can have mutating operations (e.g., add_); mutating operations difficult to work with in compiler (code motion no longer always valid)

Solution: AOTAutograd canonicalizes all IR nodes into functional, ATen operators which are easy to deal with (NB: input mutation)

AOTAutograd architecture



AOTAutograd example

```python
@torch.compile()
def func(a, b):
    return torch.clamp_min(a, b) * 3

p = torch.tensor([0.4, -0.2], requires_grad=True, device='cuda')
loss = func(p, 0).sum()
loss.backward()
print(p.grad)
```

forward / backward

AOTAutograd example: TORCH_LOGS=aot_joint_graph

```
def forward(self, primals, tangents):
    primals_1, tangents_1, = fx_pytree.tree_flatten_spec([primals, tangents], self._in_spec)
    _tensor_constant0 = self._tensor_constant0
    maximum_default = torch.ops.aten.maximum.default(primals_1, _tensor_constant0)
    mul_tensor = torch.ops.aten.mul.Tensor(maximum_default, 3)
    is_same_size_default = torch.ops.aten.is_same_size.default(mul_tensor, tangents_1)
    mul_tensor_1 = torch.ops.aten.mul.Tensor(tangents_1, 3);  tangents_1 = None
    scalar_tensor = torch.ops.aten.scalar_tensor.default(0.0, dtype = torch.float32, layout = torch.strided, device = device(type='cuda', index=0))
    ge_scalar = torch.ops.aten.ge.Scalar(primals_1, 0)
    where_self = torch.ops.aten.where.self(ge_scalar, mul_tensor_1, scalar_tensor)
    return pytree.tree_unflatten([mul_tensor, where_self], self._out_spec)
```

```
torch.clamp_min(a, b) * 3
```

```
- name: clamp_min(Tensor self, Scalar min) -> Tensor
  self: where(self >= min, grad, at::scalar_tensor(0.))
```

forward / backward

AOTAutograd example: TORCH_LOGS=aot_graphs

```
def forward(self, primals_1):
    _tensor_constant0 = self._tensor_constant0
    maximum_default = torch.ops.aten.maximum.default(primals_1, _tensor_constant0)
    mul_tensor = torch.ops.aten.mul.Tensor(maximum_default, 3)
    ge_scalar = torch.ops.aten.ge.Scalar(primals_1, 0)
    return [mul_tensor, ge_scalar]
```

Partitioner moved backward compute to forwards!

```
def backward(self, ge_scalar, tangents_1):
    mul_tensor_1 = torch.ops.aten.mul.Tensor(tangents_1, 3)
    scalar_tensor = torch.ops.aten.scalar_tensor.default(0.0, dtype = torch.float32, 
layout = torch.strided, device = device(type='cuda', index=0))
    where_self = torch.ops.aten.where.self(ge_scalar, mul_tensor_1, scalar_tensor)
    return [where_self]
```

```
torch.clamp_min(a, b) * 3
```

```
- name: clamp_min(Tensor self, Scalar min) -> Tensor
  self: where(self >= min, grad, at::scalar_tensor(0.))
```

AOTAutograd example: lowering through eager mode

```
def forward(self, primals_1):
    _tensor_constant0 = self._tensor_constant0
    maximum_default = torch.ops.aten.maximum.default(primals_1, _tensor_constant0)
    mul_tensor = torch.ops.aten.mul.Tensor(maximum_default, 3)
    ge_scalar = torch.ops.aten.ge.Scalar(primals_1, 0)
    return [mul_tensor, ge_scalar]

def backward(self, ge_scalar, tangents_1):
    mul_tensor_1 = torch.ops.aten.mul.Tensor(tangents_1, 3)
    scalar_tensor = torch.ops.aten.scalar_tensor.default(0.0, dtype = torch.float32,
layout = torch.strided, device = device(type='cuda', index=0))
    where_self = torch.ops.aten.where.self(ge_scalar, mul_tensor_1, scalar_tensor)
    return [where_self]
```

```
torch.clamp_min(a, b) * 3
```

PrimTorch decompositions

```
# torch/_refs/__init__.py
@register_decomposition(torch.ops.aten.clamp_min)
@out_wrapper()
def clamp_min(
    self: TensorLikeType,
    min: TensorOrNumberLikeType = None,
) -> TensorLikeType:
    return torch.clamp(self, min=min)
```

torch.clamp_min is just syntax sugar around torch.clamp...

...torch.clamp internally dispatches to aten.clamp...

```
# torchinductor/decomposition.py
@register_decomposition([aten.clamp])
def clamp(x, min=None, max=None):
    if min is not None:
        x = torch.maximum(x, torch.tensor(min, dtype=x.dtype, device=x.device))
    if max is not None:
        x = torch.minimum(x, torch.tensor(max, dtype=x.dtype, device=x.device))
    return x
```

...which dispatches to torch.maximum (aka aten.maximum)

Functionalization

If you have: **`x.add_(y)`**, convert into **`x_new = x.add(y)`**

What if you have an alias? **`x2 = x[0]; x.add_(y)`**

Must update all aliases! Functionalization knows to do this:

```
x2_new = x2.add(y[0])
x_new = x.add(y)
```

Note: must know if operators mutate or not! Captured by JIT schema

TorchInductor

TorchInductor Principles

- PyTorch Native
  - Similar abstractions to PyTorch eager to allow support for nearly all of PyTorch, with a thin translation layer.
- Python First
  - A pure python compiler makes TorchInductor easy to understand and hackable by users. Generates Triton and C++.
- Breadth First
  - Early focus on supporting a wide variety of operators, hardware, and optimization. A general purpose compiler, that can scale.

TorchInductor Technologies

- Define-By-Run Loop-Level IR
  - Direct use of Python functions in IR definitions allows for rapidly defining lowering with little boilerplate.
- Dynamic Shapes & Strides
  - Uses SymPy to reason about shapes, indexing, and managing guards. Symbolic shapes from the ground up.
- Reuse State-Of-The-Art Languages
  - Generates output code in languages popular for writing handwritten kernels:
    - Triton for GPUs
    - C++/OpenMP for CPUs

What is Triton?

A new programming language for highly performant GPU kernels

- Higher level than CUDA
- Lower level than preexisting DSLs
- Allows non-experts to write fast custom kernels

Users define tensors (i.e., blocks of data) in SRAM, and modify them using torch-like operators

PYTHONIC INTERFACE

Like in Numba, kernels are defined in Python using the triton.jit decorator

LOW-LEVEL MEMORY CONTROL

Users can construct tensors of pointers and dereference them element-wise

Optimizing Compiler

Blocked program representation allows the Triton compiler to generate extremely efficient code

https://triton-lang.org
https://github.com/openai/triton
by Philippe Tillet @ OpenAI

Triton: an intermediate language and compiler for tiled neural network computations

Philippe Tillet, H. T. Kung, David Cox

In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)

https://doi.org/10.1145/3315508.3329973

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++

TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	Output Triton	Output Wrapper

```python
import torch

@torch.compile(dynamic=True)
def toy_example(x):
    y = x.sin()
    z = y.cos()
    return y, z

toy_example(torch.randn([8192, 1024], device="cuda"))
```

Run with:
TORCH_COMPILE_DEBUG=1 python inductor_demo.py

TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	Output Triton	Output Wrapper

```python
def forward(self, arg0_1: f32[s0, s1]):
    # File: inductor_demo.py:6, code: y = x.sin()
    sin: f32[s0, s1] = torch.ops.aten.sin.default(arg0_1)

    # File: inductor_demo.py:7, code: z = y.cos()
    cos: f32[s0, s1] = torch.ops.aten.cos.default(sin)
    return (sin, cos)
```

TorchInductor Example

Input Code	ATen FX Graph	**Define-by-run IR**	Scheduling/Fusion	Output Triton	Output Wrapper

```
def inner_fn_buf0(index):
  i0, i1 = index
  tmp0 = ops.load(arg0_1, i1 + i0 * s1)
  tmp1 = ops.sin(tmp0)
  return tmp1

def inner_fn_buf1(index):
  i0, i1 = index
  tmp0 = ops.load(buf0, i1 + i0 * s1)
  tmp1 = ops.cos(tmp0)
  return tmp1
```

```
buf0_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf0',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf0,
                 ranges=[s0, s1], ...))))

buf1_ir = TensorBox(StorageBox(ComputedBuffer(
  name='buf1',
  layout=FixedLayout('cuda', torch.float32,
                     size=[s0, s1], stride=[s1, 1]),
  data=Pointwise(inner_fn=inner_fn_buf1,
                 ranges=[s0, s1], ...))))
```

TorchInductor Example

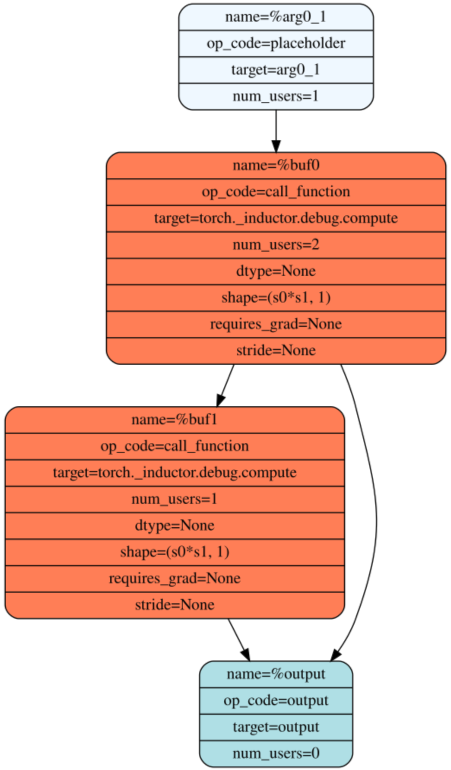
Input Code	ATen FX Graph	Define-by-run IR	**Scheduling/Fusion**	Output Triton	Output Wrapper

torch/_inductor/scheduler.py

Scheduler.can_fuse(buf0, buf1):
True

Scheduler.score_fusion(buf0, buf1):
(True, True, 33554432, -1)

- True/True is category of fusion (pointwise/pointwise)
- 33554432 is estimated memory bandwidth saved by fusion: 8192*1024*4
- -1 is distance in graph



TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	**Output Triton**	Output Wrapper

```
@triton.jit
def triton__0(in_ptr0, out_ptr0, out_ptr1, xnumel, XBLOCK : tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), None)
    tmp1 = tl.sin(tmp0)
    tmp2 = tl.cos(tmp1)
    tl.store(out_ptr0 + (x0 + tl.zeros([XBLOCK], tl.int32)), tmp1, None)
    tl.store(out_ptr1 + (x0 + tl.zeros([XBLOCK], tl.int32)), tmp2, None)
```

TorchInductor Example

Input Code	ATen FX Graph	Define-by-run IR	Scheduling/Fusion	Output Triton	**Output Wrapper**

```
def call(args):
    arg0_1, = args
    args.clear()
    arg0_1_size = arg0_1.size()
    s0 = arg0_1_size[0]
    s1 = arg0_1_size[1]
    buf0 = empty_strided((s0, s1), (s1, 1), device='cuda', dtype=torch.float32)
    buf1 = empty_strided((s0, s1), (s1, 1), device='cuda', dtype=torch.float32)
    triton__0_xnumel = s0*s1
    triton__0.run(arg0_1, buf0, buf1, triton__0_xnumel, grid=grid(triton__0_xnumel))
    return (buf0, buf1, )
```

TorchInductor Example: C++ Output

Change device='cuda' to device='cpu'

```
extern "C" void kernel(const float* __restrict__ in_ptr0,
                       float* __restrict__ out_ptr0,
                       float* __restrict__ out_ptr1,
                       const long ks0,
                       const long ks1)
{
    #pragma omp parallel num_threads(8)
    {
        {
            #pragma omp for
            for(long i0=0; i0<((ks0*ks1) / 16); i0+=1)
            {
                auto tmp0 = at::vec::Vectorized<float>::loadu(in_ptr0 + 16*i0);
                auto tmp1 = tmp0.sin();
                auto tmp2 = tmp1.cos();
                tmp1.store(out_ptr0 + 16*i0);
                tmp2.store(out_ptr1 + 16*i0);
            }
            #pragma omp for simd simdlen(8)
            for(long i0=16*(((ks0*ks1) / 16)); i0<ks0*ks1; i0+=1)
            {
                auto tmp0 = in_ptr0[i0];
                auto tmp1 = std::sin(tmp0);
                auto tmp2 = std::cos(tmp1);
                out_ptr0[i0] = tmp1;
                out_ptr1[i0] = tmp2;
            }
        }
    }
}
```

GPU RESULTS

	TorchBench	Models from **HuggingFace**	Models from **TIMM**
AMP Training Speedup vs Eager+AMP Passing models count	**1.68 gmean speedup** 64 of 64 models	**1.89 gmean speedup** 45 of 46 models	**1.66 gmean speedup** 60 of 61 models
BFloat16 Inference Speedup vs Eager+AMP Passing models count	**1.67 gmean speedup** 64 of 72 models	**1.91x gmean speedup** 46 of 46 models	**1.78x gmean speedup** 61 of 61 models

- NVIDIA A100 GPU
- TorchBench: github.com/pytorch/benchmark
- HuggingFace: github.com/huggingface/transformers
- TIMM: github.com/rwightman/pytorch-image-models
- Note: using mode=”reduce-overhead”

PT2 For Hardware Backends

Extension Points For torch.compile

1. Pre-Grad Fx Graph
2. Post-grad, functionalized
   a. ONNX has integration here
3. Torchinductor
   a. Recent PR merged adds new extensibility point for 3rd party backend
4. Triton - recommended path
   a. Intel XPU, AMD GPUs have integrated this way
   b. Takes advantage of triton templates for compute heavy operators

PT2 For Pytorch Eager

1. Primtorch/PyTorch Decompositions
    a. Vastly reduce the 2000+ PyTorch Operators
        i. Inductor has ~140 unique lowerings
    b. Explicitly model type promotion and broadcasting behavior
2. We are looking into reusing Triton codegen for eager kernels
    a. Write a compiler backend, get eager support

PT2 For Non-Python Deployment

1. torch.export
    a. Full model graph capture
2. aot_inductor
    a. Export inductor runtime to c++
3. ExecuTorch
    a. Newly released runtime targeting edge devices

Stay in Touch

1. Follow https://dev-discuss.pytorch.org/ for updates
2. PyTorch Slack
3. Github
4. email eellison (at) meta.com

Meta