PyTorch Custom C++ and CUDA Operators

Complete Example: `mymuladd`

Yuke Wang

January 30, 2026

Motivation

PyTorch provides many built-in operators, but advanced workloads require:

- Custom kernels for performance and fusion
- Native C++ or CUDA implementations
- Integration with dispatcher, autograd, and `torch.compile`

Example Operator: `mymuladd`

We implement a fused multiply-add operator:

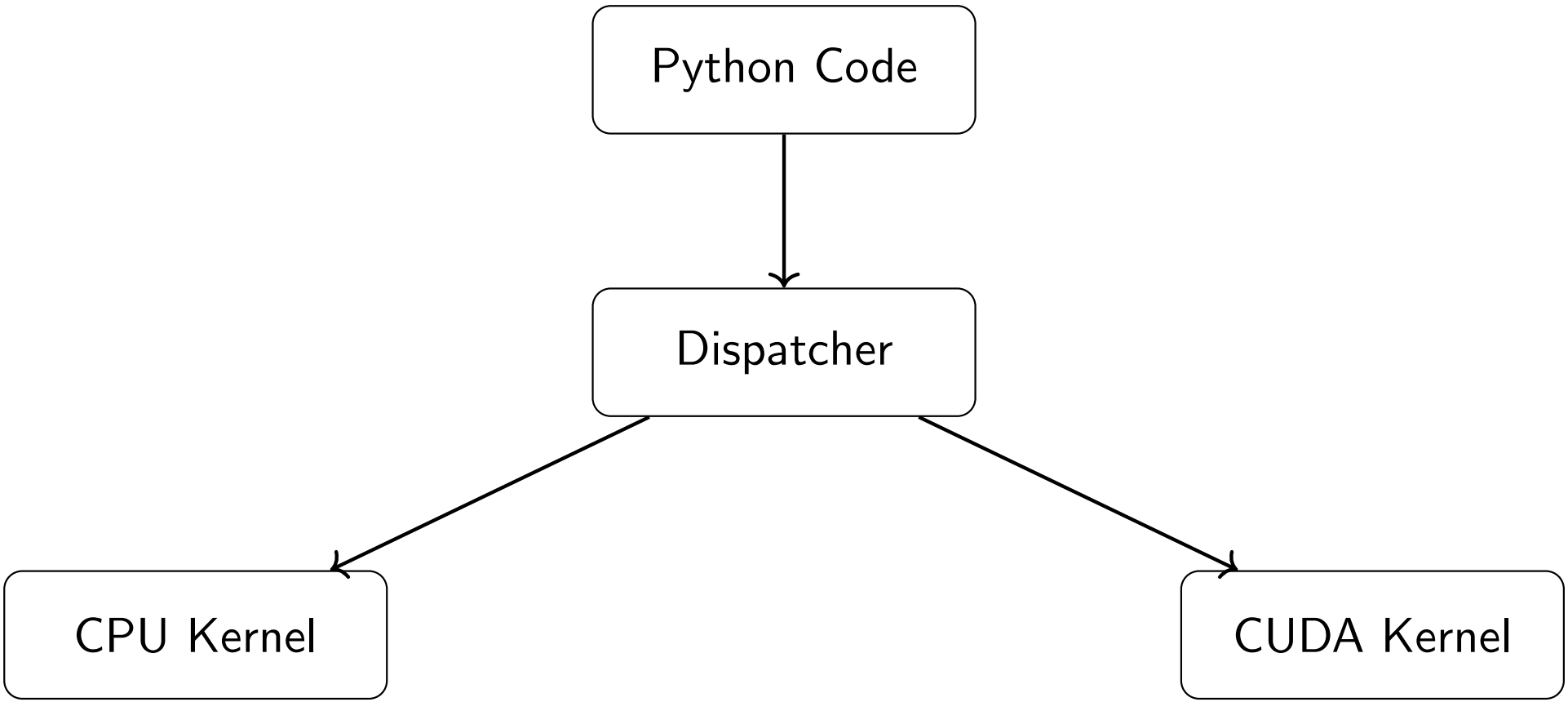
$$\texttt{mymuladd}(a, b, c) = a \cdot b + c$$

- a, b are tensors of the same shape
- c is a scalar
- Output shape matches inputs

High-Level Workflow

1. Write C++ / CUDA kernel
2. Define operator schema
3. Register operator with dispatcher
4. Build extension
5. Load and call from Python
6. Support autograd and `torch.compile`

Custom Operator Integration Overview



Custom operators integrate through the dispatcher, like native ops.

Setup: setup.py

Build the Extension

```
from setuptools import setup
from torch.utils import cpp_extension

setup(
    name='extension_cpp',
    ext_modules=[cpp_extension.CppExtension(
        'extension_cpp',
        ['mymuladd.cpp', 'mymuladd_kernel.cu'],
    )],
    cmdclass={'build_ext': cpp_extension.BuildExtension}
)
```

C++ CPU Kernel

CPU Implementation (mymuladd.cpp)

```
#include <torch/extension.h>

torch::Tensor mymuladd_cpu(
    const torch::Tensor& a,
    const torch::Tensor& b,
    double c) {
    return a * b + c;
}
```

CUDA Kernel

CUDA Implementation (mymuladd$_k$ernel.cu)

```
#include <torch/extension.h>

__global__ void muladd_kernel(float* a, float* b, float c, float* out, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) out[i] = a[i] * b[i] + c;
}

torch::Tensor mymuladd_cuda(torch::Tensor a, torch::Tensor b, float c) {
    auto out = torch::empty_like(a);
    int threads = 256;
    int blocks = (a.numel() + threads - 1) / threads;
    muladd_kernel<<<blocks, threads>>>(
        a.data_ptr<float>(),
        b.data_ptr<float>(),
        c,
        out.data_ptr<float>(),
        a.numel()
    );
    return out;
}
```

Operator Registration

Register the Operator

```
TORCH_LIBRARY(extension_cpp, m) {
    m.def("mymuladd(Tensor a, Tensor b, float c) -> Tensor");
}

TORCH_LIBRARY_IMPL(extension_cpp, CPU, m) {
    m.impl("mymuladd", TORCH_FN(mymuladd_cpu));
}

TORCH_LIBRARY_IMPL(extension_cpp, CUDA, m) {
    m.impl("mymuladd", TORCH_FN(mymuladd_cuda));
}
```

Dispatcher Model

- Dispatcher selects kernel based on:
  - Device (CPU / CUDA)
  - Dtype
  - Layout
- Single schema $\rightarrow$ multiple implementations

Python Usage

Call the Custom Operator

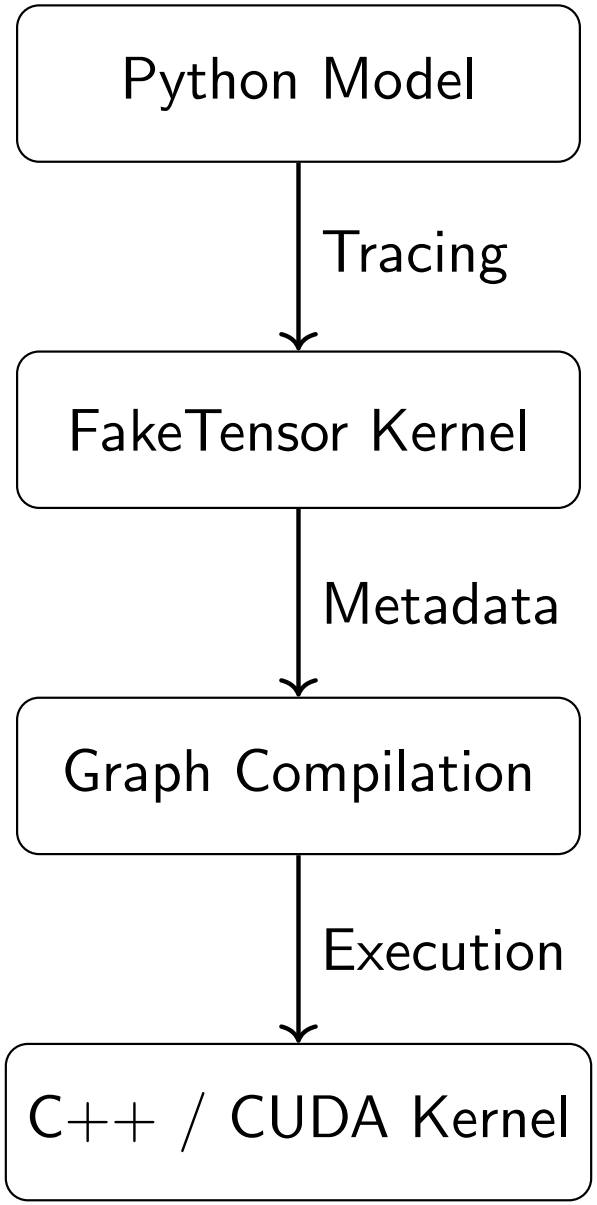
```
import torch
import extension_cpp

a = torch.rand(4)
b = torch.rand(4)
c = 2.0

y = torch.ops.extension_cpp.mymuladd(a, b, c)
```

FakeTensor Registration for torch.compile

FakeTensor Kernel

```
@torch.library.register_fake("extension_cpp::mymuladd")
def fake_mymuladd(a, b, c):
    return a
```

torch.compile Flow



Testing with opcheck

Validate Operator

```
torch.library.opcheck(
    torch.ops.extension_cpp.mymuladd.default,
    samples
)
```

Key Takeaways

- Custom C++ / CUDA operators extend PyTorch efficiently
- Dispatcher enables schema $\rightarrow$ multiple backends
- ABI-stable builds simplify deployment
- Integration supports:
  - Autograd
  - torch.compile / FakeTensor