PyTorch CONFERENCE EUROPE 2026

How to Write C++ Extensions in 2026

Jane Xu & Mikayla Gawarecki
PyTorch @ Meta

How to Write C++ Extensions in 2026

Why write a C++ extension at all?

- you have a special op (none of our 2k+ ones suffice) AND want to plug into the PyTorch system
  - autograd, PT2
  - you still want torch ops
- you could extend PyTorch with a python extension...
  - Triton, cuteDSL, etc, etc, torch ops
- ...so why C++?
  - you want to target specific hardware
  - for perf optimizations
  - you love writing C++

Say you plan to write a fused multiply-add in CUDA C++

Semantically, in Python:

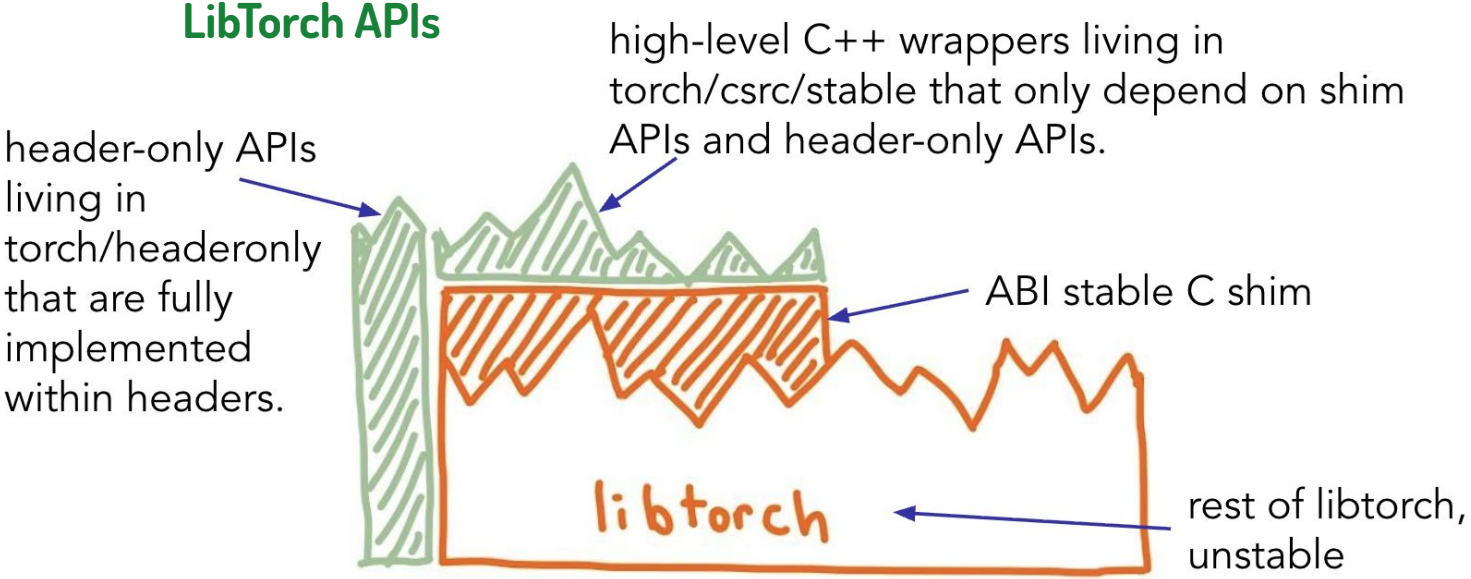
```
def mymuladd(a: Tensor, b: Tensor, c: float):
    return a * b + c
```

For simplicity, our op will return an **out-of-place**, **contiguous** result, even if the inputs are not contiguous.

(note, this is for illustration purposes! we know `std::fma` exists)

Which APIs should you use?

green = ABI-Stable LibTorch APIs



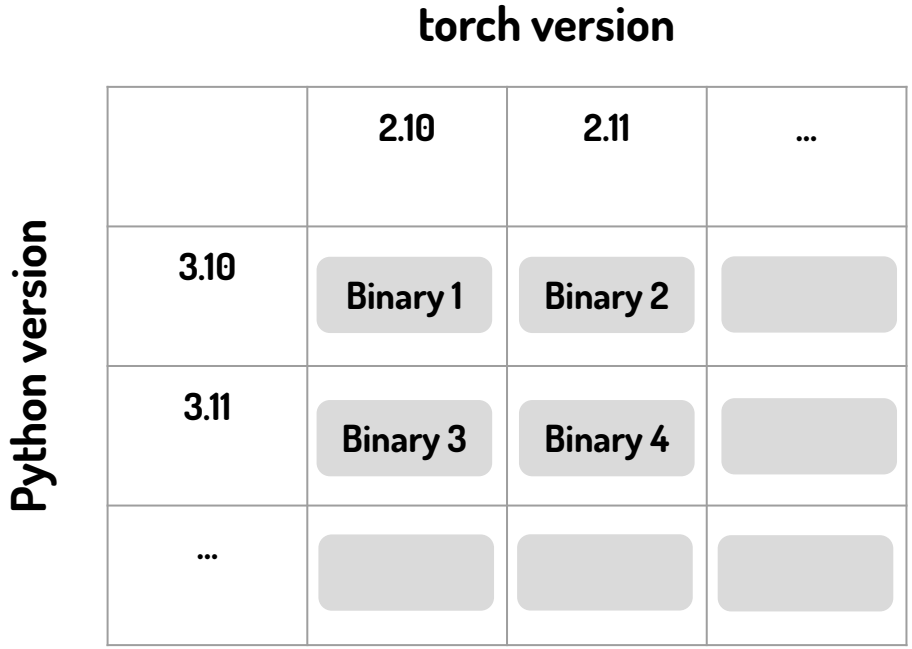
Drawing taken from Stable ABI talk at PTC 2025

Which API should you use?

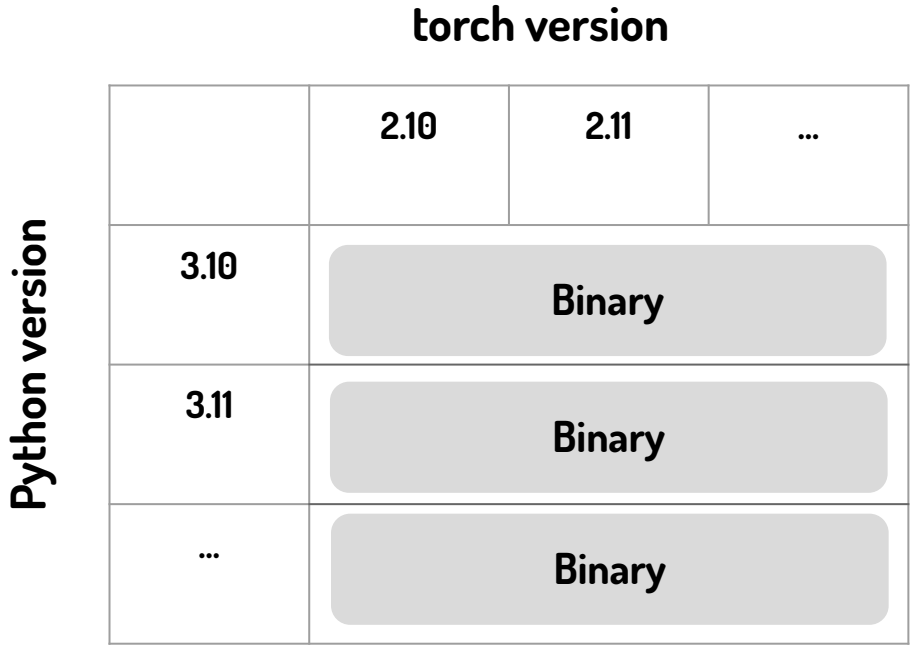
- Use the ✨ ABI-Stable LibTorch API ✨!
  - enables a single wheel that works across multiple PyTorch versions (2.10, 2.11, 2.12, etc.)
  - 2.10+ tho
  - note that this API is intentionally a limited subset of LibTorch
- You can always fall back to the non-ABI-Stable LibTorch API, but you may feel FOMO

If you need APIs not yet available in the stable ABI, file an issue requesting the API be added!

Why ABI stability?



Why ABI stability?



LibTorch ABI stable extensions can only include these headers:

- **torch/csrc/stable**: `torch::stable::Tensor`, `torch::stable::Device`
- **torch/headeronly**: utilities like `torch::headeronly::ScalarType`
- **torch/csrc/stable/c/shim.h** or **torch/csrc/inductor/aoti_torch/c/shim.h**

```
#include <ATen/Operators.h>
#include <torch/all.h>
#include <torch/library.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <ATen/cuda/CUDAContext.h>
```

```
#include <torch/csrc/stable/ops.h>
#include <torch/csrc/stable/library.h>
#include <torch/csrc/stable/tensor.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <torch/csrc/stable/c/shim.h>
```

This is enforced when the `TORCH_TARGET_VERSION` compiler flag is set.

Let's write code together!

Remember our reference in Python:

```
def mymuladd(a: Tensor, b: Tensor, c: float):
    return a * b + c
```

Let's write code together: our C++ schema

Remember our reference in Python:

```python
def mymuladd(a: Tensor, b: Tensor, c: float):
    return a * b + c
```

In C++, we'll be taking in two `Tensor`s, a `float`, and then returning a new `Tensor`:

```cpp
torch::stable::Tensor mymuladd_cuda(const torch::stable::Tensor &a,
                                    const torch::stable::Tensor &b, double c) {
    // the compute ...

    return result;
}
```

see https://docs.pytorch.org/cppdocs/stable.html#tensor-class

Let's write code together: add some checks

```
torch::stable::Tensor mymuladd_cuda(const torch::stable::Tensor &a,
                                    const torch::stable::Tensor &b, double c) {

    STD_TORCH_CHECK(a.sizes().equals(b.sizes()));
    STD_TORCH_CHECK(a.scalar_type() == torch::headeronly::ScalarType::Float);
    STD_TORCH_CHECK(b.scalar_type() == torch::headeronly::ScalarType::Float);
    STD_TORCH_CHECK(a.device().type() == torch::headeronly::DeviceType::CUDA);
    STD_TORCH_CHECK(b.device().type() == torch::headeronly::DeviceType::CUDA);

    // the compute ...

    return result;
}
```

https://docs.pytorch.org/cppdocs/stable.html#error-checking
https://docs.pytorch.org/cppdocs/stable.html#core-types
https://docs.pytorch.org/cppdocs/stable.html#device-class

Let's write code together: the CUDA kernel compute

Remember our reference in Python:

```python
def mymuladd(a: Tensor, b: Tensor, c: float):
    return a * b + c
```

In CUDA (assuming all contiguous Tensors):

```
__global__ void muladd_kernel(int numel, const float* a, const float* b, float c,
                              float* result) {

    result[idx] = a[idx] * b[idx] + c;

}
```

Let's write code together: the CUDA kernel compute

Remember our reference in Python:

```python
def mymuladd(a: Tensor, b: Tensor, c: float):
    return a * b + c
```

In CUDA (assuming all contiguous Tensors):

```
__global__ void muladd_kernel(int numel, const float* a, const float* b, float c,
                              float* result) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < numel) result[idx] = a[idx] * b[idx] + c;
}
```

Let's write code together: "call" the CUDA kernel

Our kernel signature:

```
__global__ void muladd_kernel(int numel, const float* a,
                              const float* b, float c, float* result)
```

Launch the kernel assuming block size 256:

```
torch::stable::Tensor mymuladd_cuda(const torch::stable::Tensor &a,
                                    const torch::stable::Tensor &b, double c) {
    …
    muladd_kernel<<<(numel + 255) / 256, 256, 0, stream>>>(numel, a_ptr, b_ptr,
                                                           c, result_ptr);
    return result;

}
```

CUDA Launch params: `<<<gridSize, blockSize, smemBytes, stream>>>`

Let's write code together: prepare inputs...

```
torch::stable::Tensor mymuladd_cuda(const torch::stable::Tensor &a,
                                    const torch::stable::Tensor &b, double c) {
    ...
    // we need stream, numel, a_ptr, b_ptr, result_ptr
    torch::stable::Tensor a_contig = torch::stable::contiguous(a);
    torch::stable::Tensor b_contig = torch::stable::contiguous(b);
    torch::stable::Tensor result = torch::stable::empty_like(a_contig);

    const float *a_ptr = a_contig.const_data_ptr<float>();
    const float *b_ptr = b_contig.const_data_ptr<float>();
    float *result_ptr = result.mutable_data_ptr<float>();
    ...
}
```

https://docs.pytorch.org/cppdocs/stable.html#stable-operators

Let's write code together: prepare more inputs...

```
torch::stable::Tensor mymuladd_cuda(const torch::stable::Tensor &a,
                                    const torch::stable::Tensor &b, double c) {
    ...
    // lastly, we need stream, numel
    int numel = a_contig.numel();

    void *stream_ptr = nullptr;
    TORCH_ERROR_CODE_CHECK(      // this is a peek into our shim heh
        aoti_torch_get_current_cuda_stream(a.get_device_index(), &stream_ptr));
    cudaStream_t stream = static_cast<cudaStream_t>(stream_ptr);

    ...
}
```

https://docs.pytorch.org/cppdocs/stable.html#tensor-class

And we're done!

```
torch::stable::Tensor mymuladd_cuda(const torch::stable::Tensor &a, const torch::stable::Tensor &b, double c) {
      STD_TORCH_CHECK(a.sizes().equals(b.sizes()));
      STD_TORCH_CHECK(a.scalar_type() == torch::headeronly::ScalarType::Float);
      STD_TORCH_CHECK(b.scalar_type() == torch::headeronly::ScalarType::Float);
      STD_TORCH_CHECK(a.device().type() == torch::headeronly::DeviceType::CUDA);
      STD_TORCH_CHECK(b.device().type() == torch::headeronly::DeviceType::CUDA);

      torch::stable::Tensor a_contig = torch::stable::contiguous(a);
      torch::stable::Tensor b_contig = torch::stable::contiguous(b);
      torch::stable::Tensor result = torch::stable::empty_like(a_contig);
      const float *a_ptr = a_contig.const_data_ptr<float>();
      const float *b_ptr = b_contig.const_data_ptr<float>();
      float *result_ptr = result.mutable_data_ptr<float>();

      int numel = a_contig.numel();
      void *stream_ptr = nullptr;
      TORCH_ERROR_CODE_CHECK(aoti_torch_get_current_cuda_stream(a.get_device_index(), &stream_ptr));
      cudaStream_t stream = static_cast<cudaStream_t>(stream_ptr);

      muladd_kernel<<<(numel + 255) / 256, 256, 0, stream>>>(numel, a_ptr, b_ptr, c, result_ptr);
      return result;
}
```

https://github.com/pytorch/extension-cpp/blob/master/extension_cpp_stable/extension_cpp_stable/csrc/cuda/muladd.cu

But how do we expose it to Python?

Use torch.Library APIs to define and register `mymuladd_cuda` to the PyTorch dispatcher:

```
STABLE_TORCH_LIBRARY(extension_cpp_stable, m) {
  m.def("mymuladd(Tensor a, Tensor b, float c) -> Tensor");
}
STABLE_TORCH_LIBRARY_IMPL(extension_cpp_stable, CUDA, m) {
  m.impl("mymuladd", TORCH_BOX(&mymuladd_cuda));
}
```

You can then call `mymuladd` from Python with Python Tensors and floats!

```
torch.ops.extension_cpp_stable.mymuladd.default(a, b, c)
```

But how do we expose it to Python?

Use torch.Library APIs to define and register `mymuladd_cuda` to the PyTorch dispatcher:

```
STABLE_TORCH_LIBRARY(extension_cpp_stable, m) {
  m.def("mymuladd(Tensor a, Tensor b, float c) -> Tensor");
}
STABLE_TORCH_LIBRARY_IMPL(extension_cpp_stable, CUDA, m) {
  m.impl("mymuladd", TORCH_BOX(&mymuladd_cuda));
}
```

You can then call `mymuladd` from Python with Python Tensors and floats!

~~`torch.ops.extension_cpp_stable.mymuladd.default(a, b, c)`~~

Well...not yet. But we will get back to this!

Why do we need TORCH_BOX?

TORCH_BOX converts a function pointer to a kernel to a boxed kernel:

```
void boxed_fn(StableIValue* stack, uint64_t num_args, uint64_t num_outputs) {
    STD_TORCH_CHECK(num_args == 3, ...); // Check arity matches the schema
    STD_TORCH_CHECK(num_outputs == 1, ...);

    std::tuple<Tensor, Tensor, float> args = std::make_tuple(
        to<Tensor>(stack[0]), to<Tensor>(stack[1]), to<float>(stack[2]));

    auto res = std::apply(&mymuladd_cuda, args); // Call the actual function

    stack[0] = from<Tensor>(res); // Box the result back into a StableIValue
}
```

Why do we need TORCH_BOX?

`to` and `from` convert objects ⇔ StableIValue, the ABI stable representation of the object.

```
void boxed_fn(StableIValue* stack, uint64_t num_args, uint64_t num_outputs) {
    STD_TORCH_CHECK(num_args == 3, ...); // Check arity matches the schema
    STD_TORCH_CHECK(num_outputs == 1, ...);

    std::tuple<Tensor, Tensor, float> args = std::make_tuple(
        to<Tensor>(stack[0]), to<Tensor>(stack[1]), to<float>(stack[2]));

    auto res = std::apply(&mymuladd_cuda, args); // Call the actual function

    stack[0] = from<Tensor>(res); // Box the result back into a StableIValue
}
```

It must get compiled: use torch.utils.cpp_extension

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

setup(name="extension_cpp_stable",
      ext_modules=[
          cpp_extension.CUDAExtension(
              "extension_cpp_stable",
              ["muladd.cu"],
              extra_compile_args={
                  "cxx": [
                      # define Py_LIMITED_API with min version 3.9 to expose only the stable
                      # limited API subset from Python.h
                      "-DPy_LIMITED_API=0x03090000",
                   ]
                  "nvcc": [
                      # define TORCH_TARGET_VERSION with min version 2.10 to expose only the
                      # stable API subset from torch
                      "-DTORCH_TARGET_VERSION=0x020a000000000000",
                      # USE_CUDA is currently needed for some of the CUDA shims
                      "-DUSE_CUDA",
                  ]
              },
              py_limited_api=True)],  # Build 1 wheel across multiple Python versions
      cmdclass={'build_ext': cpp_extension.BuildExtension},
      options={"bdist_wheel": {"py_limited_api": "cp39"}}  # 3.9 is minimum supported Python version
```

TORCH_TARGET_VERSION

Build a LibTorch agnostic wheel

ABI tag is 0, reserved for future use

```
[ byte ][ byte ][ byte ][ byte ][ byte ][ byte ][ byte ][ byte ]
[MAJ   ][ MIN  ][PATCH ][                ABI TAG              ]
```

Use compiler flag, **`-DTORCH_TARGET_VERSION=0x020a000000000000`** for 2.10

TORCH_TARGET_VERSION

Purpose #1: Bans extension from including libtorch-unstable headers

```
#include <ATen/cuda/CUDAContext.h>

torch::stable::Tensor mymuladd_cuda(
    const torch::stable::Tensor& a,
    const torch::stable::Tensor& b,
    double c) {

  ...

  cudaStream_t stream = at::cuda::getCurrentCUDAStream();

  ...

}
```

Compilation error:
"This file should not be included when either TORCH_STABLE_ONLY or TORCH_TARGET_VERSION is defined.

TORCH_TARGET_VERSION

Purpose #2: Select a minimum torch version the compiled extension runs with

- Set 2.9 <= `TORCH_TARGET_VERSION` <= LibTorch version at build time
- For example
  - Download a 2.12 torch nightly
  - Build extension with **`-DTORCH_TARGET_VERSION=0x020a000000000000`**
  - Compiled extension is guaranteed to run with any torch nightly/stable release >= torch 2.10

How TORCH_TARGET_VERSION works?

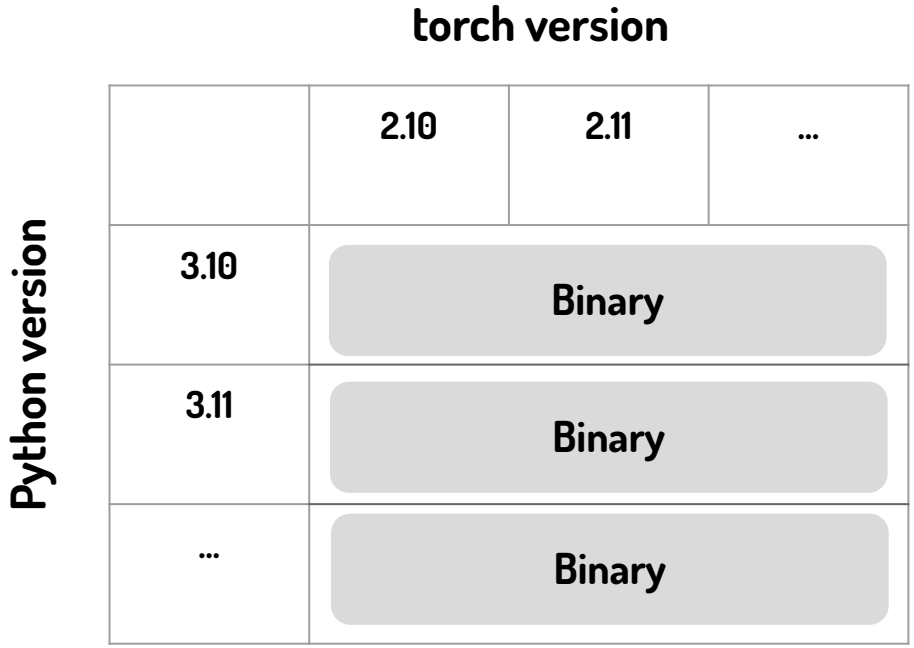
C-shims in shim.h are versioned

```
#if TORCH_FEATURE_VERSION >= TORCH_VERSION_2_11_0
...
AOTI_TORCH_EXPORT AOTITorchError torch_from_blob(
    void* data,
    int64_t ndim,
    const int64_t* sizes_ptr,
    const int64_t* strides_ptr,
    int64_t storage_offset,
    int32_t dtype,
    int32_t device_type,
    int32_t device_index,
    AtenTensorHandle* ret, // returns new reference
    int32_t layout,
    const uint8_t* opaque_metadata,
    int64_t opaque_metadata_size,
    void (*deleter)(void* data, void* ctx),
    void* deleter_ctx);

#endif // TORCH_FEATURE_VERSION >= TORCH_VERSION_2_11_0
```

APIs in torch/csrc/stable are versioned based on the max version of any shim it uses

```
#if TORCH_FEATURE_VERSION >= TORCH_VERSION_2_11_0

inline torch::stable::Tensor from_blob(
    void* data,
    torch::headeronly::IntHeaderOnlyArrayRef sizes,
    ...) {
  ...
  if constexpr (std::is_convertible_v<F, DeleterFnPtr>) {
    ...
    TORCH_ERROR_CODE_CHECK(torch_from_blob(...)
  } else {
    ...
  return torch::stable::Tensor(ath);
}
#endif // TORCH_FEATURE_VERSION >= TORCH_VERSION_2_11_0
```

Why ABI stability?



Py_LIMITED_API

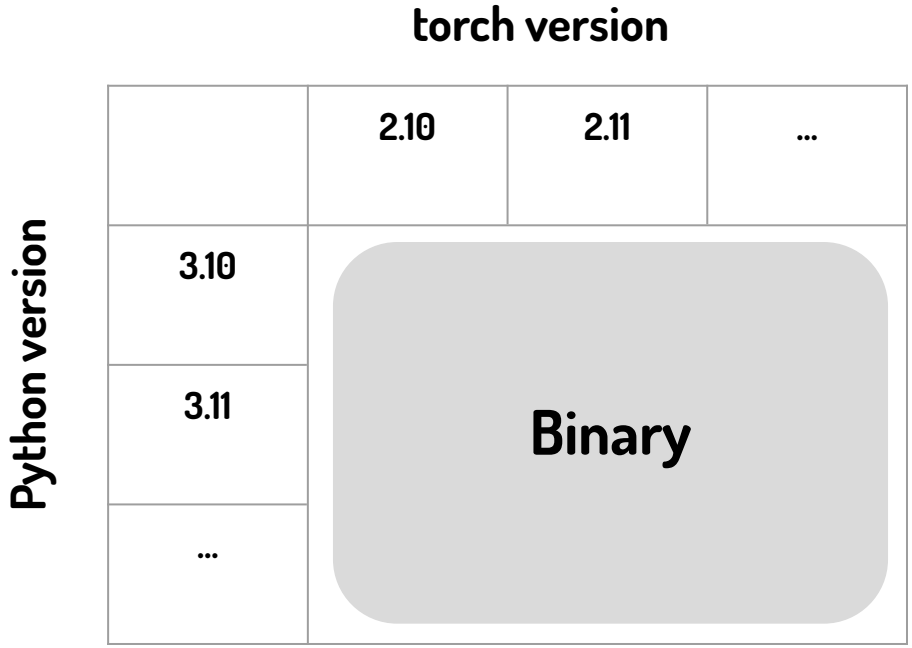
Build a CPython agnostic wheel

help verify extension only uses CPython Stable Limited API

```
setup(name="extension_cpp_stable",
      ext_modules=[
          cpp_extension.CUDAExtension(
              ...
              extra_compile_args={
                  "cxx": [
                      "-DPy_LIMITED_API=0x03090000",
                      ...
                  ]
              },
              py_limited_api=True)],
    cmdclass={'build_ext': cpp_extension.BuildExtension},
    options={"bdist_wheel": {"py_limited_api": "cp39"}}  )
```

inform setuptools that you plan to build a CPython agnostic wheel

Why ABI stability? For the goal of achieving a single wheel!



Revisiting our Compatibility Matrix

torch version

Python version	2.10	2.12	...
3.10	Binary 1	Binary 2	
3.11	Binary 3	Binary 4	
...			

DPy_LIMITED_API=0x030a0000

DTORCH_TARGET_VERSION=0x020a000000000000

	2.10	2.11	...
3.10	Binary		
3.11			
...			

Back to calling your op in Python

We must first load the C++ library that holds the custom op definition:

```
# In __init__.py
import torch
from pathlib import Path

so_files = list(Path(__file__).parent.glob("_C*.so"))
assert len(so_files) == 1, f"Expected one _C*.so file, found {len(so_files)}"
torch.ops.load_library(so_files[0])
```

the challenge often is figuring out where the .so is situated

The op registered is now available from Python as `torch.ops.extension_cpp_stable.mymuladd`

Back to calling your op in Python

There are other ways to expose C++ ops, but we recommend this to achieve **CPython ABI stability** as mentioned before!

```
# In __init__.py
import torch
from pathlib import Path

so_files = list(Path(__file__).parent.glob("_C*.so"))
assert len(so_files) == 1, f"Expected one _C*.so file, found {len(so_files)}"
torch.ops.load_library(so_files[0])
```

Adding torch.compile support

You'd have to give us some deets about your op through a meta kernel. It's easiest to do it through Python:

```
@torch.library.register_fake("extension_cpp_stable::mymuladd")
def _(a, b, c):
    torch._check(a.shape == b.shape)  # add any checks for your inputs
    torch._check(a.dtype == torch.float)
    torch._check(b.dtype == torch.float)
    torch._check(a.device == b.device)

    # return what the output shape and metadata should be
    return torch.empty_like(a)
```

Make sure everything works accordingly with testing!

```
def reference_muladd ...
samples = ...
for args in samples:
    # Correctness test
    result = torch.ops.extension_cpp_stable.mymuladd(*args)
    expected = reference_muladd(*args)
    torch.testing.assert_close(result, expected)

    # Use opcheck to check for incorrect usage of operator registration APIs
    torch.library.opcheck(torch.ops.extension_cpp_stable.mymuladd.default,
                          args)
```

Note that opcheck does not check everything! You should always verify via tests!

Go off and write a real good PyTorch C++ Extension!

This presentation walked through a simple muladd example:
https://github.com/pytorch/extension-cpp/tree/master/extension_cpp_stable

There are legit use cases in the wild!

- FlashAttention-3: https://github.com/Dao-AILab/flash-attention/blob/main/hopper
- xformers CUDA: https://github.com/facebookresearch/xformers
- torchaudio: https://github.com/pytorch/audio
- torchao CUDA: https://github.com/pytorch/ao
- vLLM CUDA - in progress, https://github.com/vllm-project/vllm

Now it's your turn to follow suit!


PyTorch
CONFERENCE
EUROPE 2026