# Accelerating PyTorch Models With torch.compile's Cpp-Wrapper Mode

Bin Bao, Meta

PYTORCH CONFERENCE EUROPE 2026

# GPU model optimization falls into three bottleneck categories

## Compute Bound

- Faster hardware (next-gen GPUs)
- Lower precision (FP16, BF16, INT8, etc.)

## Memory Bound

- Kernel fusion (reduce memory traffic)
- Lower precision (smaller data footprint)

## CPU Bound

- CUDAGraphs: works well when applicable, but limited coverage
- ***Cpp-Wrapper: replace Python execution with C++ compiled binary***

# torch.compile architecture

- **Inductor generates two artifacts:**
  - Compute kernels (Triton) — GPU code that does the math
  - Wrapper code — orchestrates kernel launches, memory allocation, and synchronization
- **Cpp-Wrapper replaces the wrapper from Python to C++**
  - Eliminates Python interpreter overhead
  - Compute kernels remain unchanged (same Triton code)

**TorchDynamo**
Python Bytecode Tracing

↓

**AOT Autograd**
Graph Capture & Decomposition

↓

**TorchInductor**
Optimizing Compiler

↙ ↘

**Compute Kernels**
(Triton)

**Execution Wrapper**
(Python → C++)

# Cpp-Wrapper vs. AOTInductor — same C++ backend, different deployment targets

| Aspect | Cpp-Wrapper | AOTInductor |
|---|---|---|
| **Use Case** | torch.compile | Ahead-of-time export & deploy |
| **Workflow** | JIT compile at first call | Export → compile → load artifact |
| **Python Dependency** | Still runs within Python process | Fully standalone C++ binary |
| **Best For** | Training and inference | Inference |
| | | |
| | | |

**Key Takeaway:** Both share the same Inductor C++ code generation backend. Cpp-Wrapper brings C++ speed to the interactive torch.compile workflow without requiring model export.

# Enabling Cpp-Wrapper is a single configuration change

## Default Python Wrapper

```
import torch

torch.compile(my_model)(x, y)
```

## With Cpp-Wrapper Enabled

```
import torch
import torch._inductor.config as inductor_config
inductor_config.cpp_wrapper = True

torch.compile(my_model)(x, y)
```

**Only 2 lines of code to switch from Python to C++ wrapper execution,**
**or run with environment variable TORCHINDUCTOR_CPP_WRAPPER=1**

**** C++ compiler required***

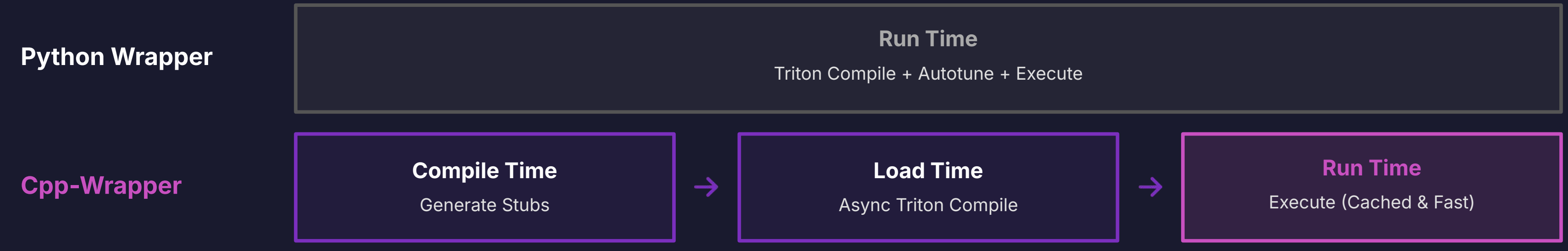
# Lazy kernel compilation separates compile time from run time

## Python Wrapper (Default)

- Triton kernel compilation and autotuning happen at run time

## Cpp-Wrapper (Optimized)

- Clear separation of compile time vs. run time
- Compile time: generate Python call stubs for Triton kernels
- Load time: invoke async Triton kernel compilation in background
- Run time: autotune on the first run and results cached
- Result: same Triton kernel, faster on warm run



# Cpp-Wrapper supports Tensor Memory Accelerator (TMA) descriptors natively

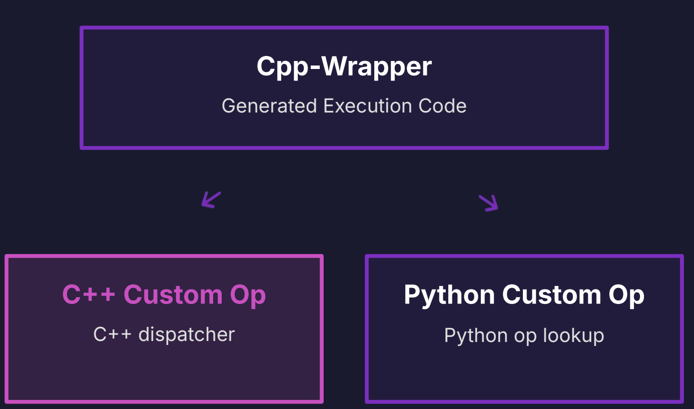
- TMA is a hardware feature on NVIDIA Hopper+ GPUs for efficient tensor data movement
- Requires creating TMA descriptors that specify tensor size and layout

**Cpp-Wrapper Support:**

- Construct TensorDescriptor in Python on the first run
- Cache the result in StableTensorDescriptor

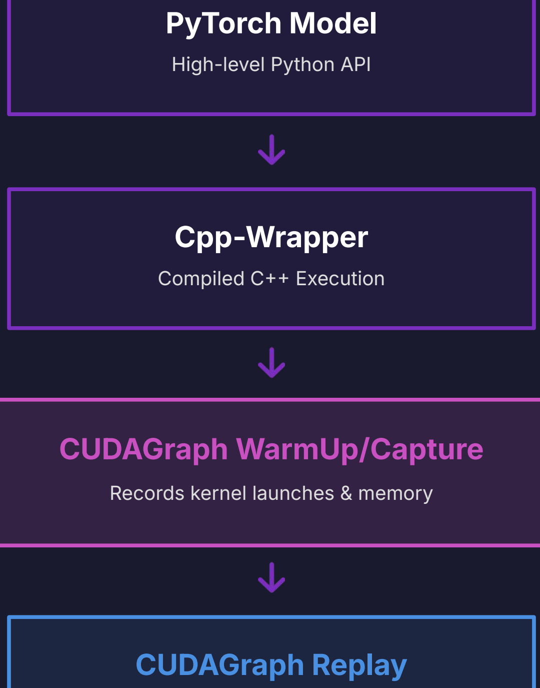
# Custom ops integrate seamlessly through the C++ fallback mechanism

- **Users register custom ops via torch.library or C++ extensions**
- **Cpp-Wrapper handles custom ops through:**
  - Generating C++ call into PyTorch dispatcher for registered C++ custom ops
  - Falling back to Python dispatch for pure-Python custom ops

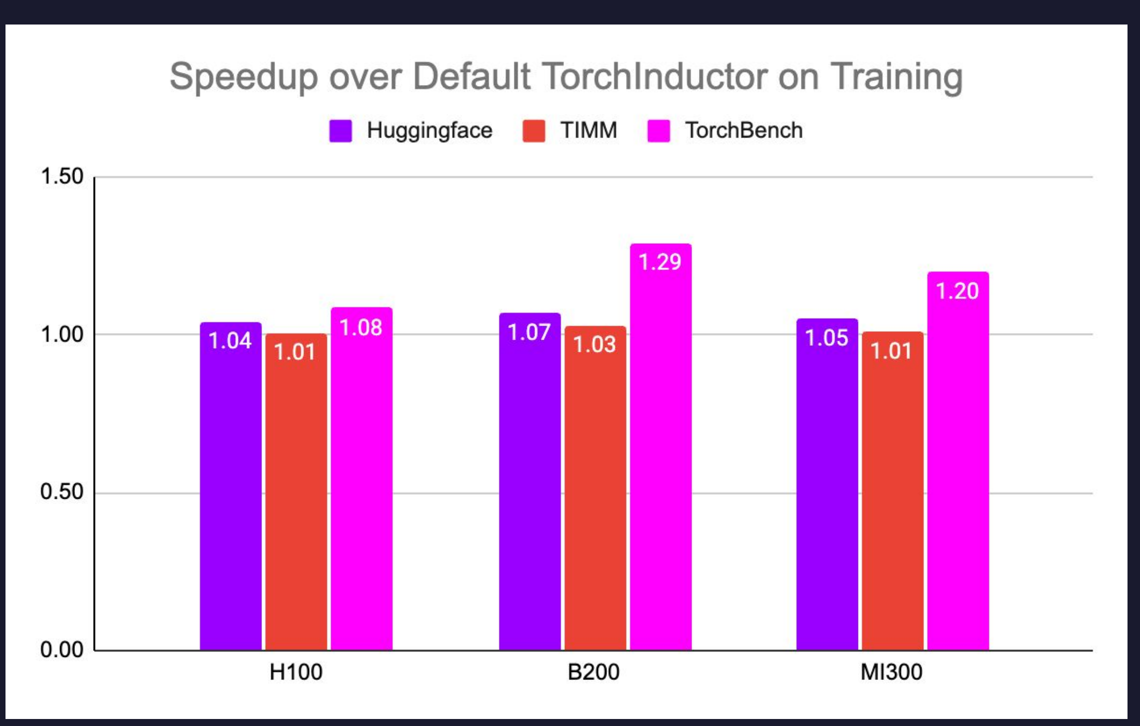


# CUDAGraphs composability eliminates remaining kernel launch overhead

- Inductor reduce-overhead mode uses CUDAGraphs as an additional optimization layer on top of generated wrapper code
- **Current status with Cpp-Wrapper:**
  - Compatible when NOT using graph partition
  - The warmup stage triggers cpp-wrapper first run and later capture/replay just works
- **Coming soon:**
  - Support graph partition



# Cpp-Wrapper delivers consistent speedup over default Inductor across hardware

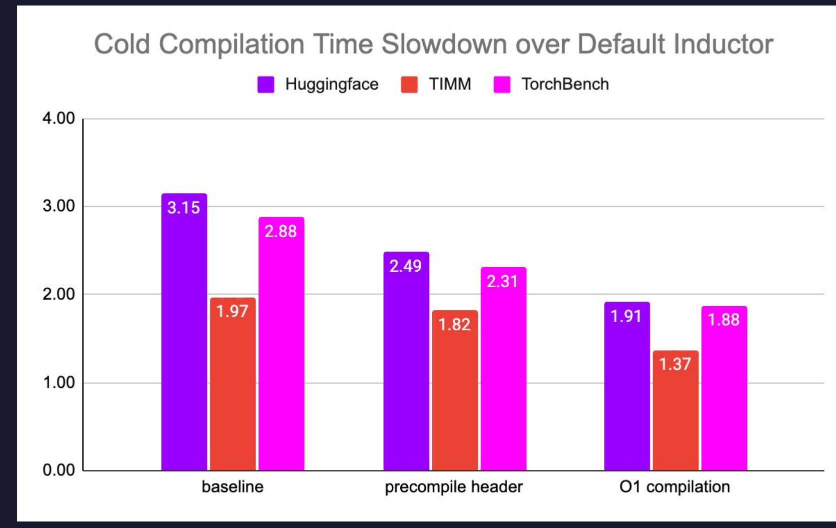


# C++ compilation overhead is mitigated through multiple optimization techniques

**Challenge:** C++ compilation adds extra time for cold (first-time) compilation compared to the default Python wrapper.

## Optimization Techniques:

- **Precompiled headers (PCH):** Cache common header compilations across runs
- **Code splitting:** Avoid large function body and use __noinline__ attribute when needed
- **Lower optimization level :** -O1 achieves the same performance as -O3, but much faster to compile
- **Warm cache:** After first compilation, subsequent runs use cached artifacts



# Status

- Available on nightly
- Some relevant PRs
  - *https://github.com/pytorch/pytorch/pull/175416*
  - *https://github.com/pytorch/pytorch/pull/178166*

**Future work:**

- Make it composable with more non-default-on optimizations
- Further reduce cold compilation time

# Thank You

PYTORCH CONFERENCE EUROPE 2026

- **Special thanks to Benjamin Glass from OpenTeams for collaborating on this work**