

 PyTorch

CONFERENCE

—NORTH AMERICA 2026—

PyTorch Generalization: A Journey Toward Write Once, Run Anywhere



Guangye Yu
Intel



Eikan Wang
Intel

API Fragmentation Issue

```
6  if device_type == "cuda":
7      device_count = torch.cuda.device_count()
8      torch.cuda.set_device(0)
9      stream = torch.cuda.Stream()
10     model = model.to("cuda")
11     with torch.cuda.stream(stream):
12         input = torch.randn(64, 3, 224, 224, device="cuda")
13         output = model(input)
14     torch.cuda.synchronize()
15 elif device_type == "xpu":
16     device_count = torch.xpu.device_count()
17     torch.xpu.set_device(0)
18     stream = torch.xpu.Stream()
19     model = model.to("xpu")
20     with torch.xpu.stream(stream):
21         input = torch.randn(64, 3, 224, 224, device="xpu")
22         output = model(input)
23     torch.xpu.synchronize()
24 elif ...:
25
```

- Device-specific APIs lead to API fragmentation and increased maintenance overhead.
- Adding a new backend requires extending existing if-else branches, resulting in increasingly complex code.

Device Interface Solution

```
6  if device_type == "cuda":
7      device_count = torch.cuda.device_count()
8      torch.cuda.set_device(0)
9      stream = torch.cuda.Stream()
10     model = model.to("cuda")
11     with torch.cuda.stream(stream):
12         input = torch.randn(64, 3, 224, 224, device="cuda")
13         output = model(input)
14     torch.cuda.synchronize()
15 elif device_type == "xpu":
16     device_count = torch.xpu.device_count()
17     torch.xpu.set_device(0)
18     stream = torch.xpu.Stream()
19     model = model.to("xpu")
20     with torch.xpu.stream(stream):
21         input = torch.randn(64, 3, 224, 224, device="xpu")
22         output = model(input)
23     torch.xpu.synchronize()
24 elif ...:
25
```

```
27 class DeviceInterface:
28     def device_count(self):
29         raise NotImplementedError
30
31     def set_device(self, device):
32         raise NotImplementedError
33
34     def synchronize(self):
35         ...
36
37 def register_interface_for_device(device, device_interface):
38     ...
39
40 def get_interface_for_device(device) -> type[DeviceInterface]:
41     ...
42
43 device_interface = get_interface_for_device(device_type)
44 device_count = device_interface.device_count()
45 device_interface.set_device(0)
46 stream = device_interface.Stream()
47 model = model.to(device_type)
48 with device_interface.stream(stream):
49     input = torch.randn(64, 3, 224, 224, device=device_type)
50     output = model(input)
51 device_interface.synchronize()
52
```

Device Interface Pros & Cons

```
27 class DeviceInterface:
28     def device_count(self):
29         raise NotImplementedError
30
31     def set_device(self, device):
32         raise NotImplementedError
33
34     def synchronize(self):
35         ...
36
37 def register_interface_for_device(device, device_interface):
38     ...
39
40 def get_interface_for_device(device) -> type[DeviceInterface]:
41     ...
42
```

Pros:

- Eliminates if-else branches and scales to new backend.
- Decouple core logic from backend implementations through interface registration.

Cons:

- Module-specific and difficult to share across components or repositories.
- Can lead to duplicated interface definitions and implementation efforts.

Unified API torch.accelerator

```
6  if device_type == "cuda":
7      device_count = torch.cuda.device_count()
8      torch.cuda.set_device(0)
9      stream = torch.cuda.Stream()
10     model = model.to("cuda")
11     with torch.cuda.stream(stream):
12         input = torch.randn(64, 3, 224, 224, device="cuda")
13         output = model(input)
14     torch.cuda.synchronize()
15 elif device_type == "xpu":
16     device_count = torch.xpu.device_count()
17     torch.xpu.set_device(0)
18     stream = torch.xpu.Stream()
19     model = model.to("xpu")
20     with torch.xpu.stream(stream):
21         input = torch.randn(64, 3, 224, 224, device="xpu")
22         output = model(input)
23     torch.xpu.synchronize()
24 elif ...:
25
54 device_type = torch.accelerator.current_accelerator()
55
56 device_count = torch.accelerator.device_count()
57 torch.accelerator.set_device_index(0)
58 stream = torch.Stream()
59 model = model.to(device_type)
60 with stream:
61     input = torch.randn(64, 3, 224, 224, device=device_type)
62     output = model(input)
63 torch.accelerator.synchronize()
64
```

Write Once, Run Anywhere – One API, Any Accelerator

- One consistent and well-defined API across all accelerators.
- Seamless integration of new accelerator.
- Out-of-the-box PyTorch ecosystem support for both in-tree and out-of-tree accelerators.

Design Principle

- Only one kind of accelerator can be enabled at a time on a host, determined at compile time.
- Currently supported accelerator backends including CUDA, XPU, MPS, MTIA, HPU, and PrivateUser1.
- CPU is not considered an accelerator in PyTorch
- Device-specific API `torch.device_type.foo` and `torch.accelerator.foo` are designed to interoperate seamlessly.

```
66 torch.cuda.set_device(1)
67 torch.accelerator.current_device_index() # returns 1
```

Design Scope

- PyTorch runtime components include 6 parties: **Device**, **Stream**, **Event**, **Allocator**, **Generator**, and others (such as **Graph**).
- **torch.accelerator.foo** generally maps 1:1 to torch.<device_type>.foo
- **torch.Stream** provides equivalent functionality to torch.<device_type>.Stream
- **torch.Event** provides equivalent functionality to torch.<device_type>.Event

Accelerator Integration

Device, Stream, Event

```
struct DeviceGuardImplInterface {  
    virtual Device getDevice() const = 0;  
    virtual Stream exchangeStream(Stream) const = 0;  
    ...  
};
```

Allocator

```
struct DeviceAllocator {  
    virtual void emptyCache(MempoolId_t) = 0;  
    virtual void recordStream(const DataPtr&, Stream) = 0;  
    ...  
};  
struct HostAllocator {  
    virtual void empty_cache() = 0;  
    virtual void record_event(void*, void*, Stream) = 0;  
    ...  
};
```

Accelerator
Integration

Generator

```
struct XXXHooksInterface {  
    virtual Generator* getNewGenerator(Device) = 0;  
    virtual const Generator& getDefaultGenerator(Device) = 0;  
    ...  
};
```

Graph

```
struct GraphImplInterface {  
    virtual void capture_begin(MempoolId_t, GraphCaptureMode) = 0;  
    virtual void capture_end() = 0;  
    ...  
};
```

Generalization Usage

- PyTorch module: deliver a consistent experience across all accelerators.
- Test: share tests across all accelerators, expand test coverage, reduce validation effort, and improve backend quality.
- Ecosystem: enable new accelerators to integrate with less repo-specific code.

Acknowledgments

A special thank you to the PyTorch community maintainers, including

Alban(@albanD), **Jason**(@jansel), **Elias**(@eellison) and many others.

Thank you for your valuable insights, thoughtful feedback, and continued guidance.

Thank You!



Guangye Yu
@guangyey

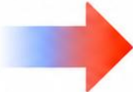


Eikan Wang
@EikanWang

Appendix: torch.accelerator API List

One interface for device management across accelerators

BEFORE		
torch.xxx.*		
Device-specific (e.g., torch.cuda.*)		
	Availability	<code>torch.xxx.is_available()</code>
	Device Count	<code>torch.xxx.device_count()</code>
	Current Device	<code>torch.xxx.current_device()</code>
	Set Device	<code>torch.xxx.set_device(i)</code>
	Device Context	<code>torch.xxx.device(i)</code>
	Synchronization	<code>torch.xxx.synchronize()</code>


UNIFY
From device-specific
to any accelerator

UNIFIED API		
torch.accelerator.*		
Unified interface for all accelerators		
	Availability	<code>torch.accelerator.is_available()</code>
	Device Count	<code>torch.accelerator.device_count()</code>
	Current Accelerator	<code>torch.accelerator.current_accelerator()</code>
	Current Device	<code>torch.accelerator.current_device_index()</code>
	Set Device	<code>torch.accelerator.set_device_index(i)</code>
	Device Context	<code>torch.accelerator.device_index(i)</code>
	Synchronization	<code>torch.accelerator.synchronize()</code>

Type
+
Index

Appendix: torch.accelerator API List

One interface for stream management across accelerators

BEFORE		
torch.xxx.*		
Device-specific (e.g., torch.cuda.*)		
≈	Get Current Stream	<code>torch.xxx.current_stream(device=None)</code>
⊕	Create Stream	<code>torch.xxx.Stream(device=None, priority=0)</code>
↔	Set Current Stream	<code>torch.xxx.set_stream(stream)</code>
C	Stream Context	<code>torch.xxx.stream(stream)</code>



UNIFY

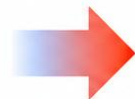
From device-specific
to any accelerator

UNIFIED API		
torch.accelerator.*		
Unified interface for all accelerators		
≈	Get Current Stream	<code>torch.accelerator.current_stream(device=None)</code>
⊕	Create Stream	<code>torch.Stream(device=None, priority=0)</code>
↔	Set Current Stream	<code>torch.accelerator.set_stream(stream)</code>
C	Stream Context	<code>stream</code>

Appendix: torch.accelerator API List

One interface for memory management across accelerators

BEFORE		
torch.xxx.*		
Device-specific (e.g., torch.cuda.*)		
	Release Cache	<code>torch.xxx.empty_cache()</code>
	Release Host Cache	<code>torch.xxx.empty_host_cache()</code>
	Memory Info	<code>torch.xxx.mem_get_info()</code>
	Memory Allocated	<code>torch.xxx.memory_allocated()</code>
	Peak Allocated	<code>torch.xxx.max_memory_allocated()</code>
	Memory Reserved	<code>torch.xxx.memory_reserved()</code>
	Peak Reserved	<code>torch.xxx.max_memory_reserved()</code>
	Memory Statistics	<code>torch.xxx.memory_stats()</code>
	Reset Accumulated Stats	<code>torch.xxx.reset_accumulated_memory_stats()</code>
	Reset Peak Stats	<code>torch.xxx.reset_peak_memory_stats()</code>



UNIFY
From device-specific
to any accelerator

UNIFIED API		
torch.accelerator.*		
Unified interface for all accelerators		
	Release Cache	<code>torch.accelerator.empty_cache()</code>
	Release Host Cache	<code>torch.accelerator.empty_host_cache()</code>
	Memory Info	<code>torch.accelerator.get_memory_info()</code>
	Memory Allocated	<code>torch.accelerator.memory_allocated()</code>
	Peak Allocated	<code>torch.accelerator.max_memory_allocated()</code>
	Memory Reserved	<code>torch.accelerator.memory_reserved()</code>
	Peak Reserved	<code>torch.accelerator.max_memory_reserved()</code>
	Memory Statistics	<code>torch.accelerator.memory_stats()</code>
	Reset Accumulated Stats	<code>torch.accelerator.reset_accumulated_memory_stats()</code>
	Reset Peak Stats	<code>torch.accelerator.reset_peak_memory_stats()</code>

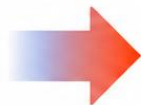
Appendix: torch.accelerator API List

One interface for generator management across accelerators

BEFORE

torch.xxx.*
Device-specific (e.g., torch.cuda.*)

	Initial Seed	<code>torch.xxx.random.initial_seed()</code>
	RNG State	<code>torch.xxx.random.get_rng_state()</code>
	RNG State All	<code>torch.xxx.random.get_rng_state_all()</code>



UNIFY
From device-specific
to any accelerator

UNIFIED API

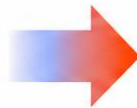
torch.accelerator.*
Unified interface for all accelerators

	Initial Seed	<code>torch.accelerator.random.initial_seed()</code>
	RNG State	<code>torch.accelerator.random.get_rng_state()</code>
	RNG State All	<code>torch.accelerator.random.get_rng_state_all()</code>

Appendix: torch.accelerator API List

One API for Stream across accelerators

BEFORE	
torch.xxx.Stream	
Device-specific (e.g., torch.cuda.Stream)	
 Create Stream	<code>torch.xxx.Stream(device=None, priority=0)</code>
 Methods	<code>wait_stream(other_stream) wait_event(event) record_event(event) synchronize() is_capturing() query()</code>
 Stream Context	<code>with stream: # operations on the stream ...</code>
 Underlying Handle	<code>stream.cuda_stream</code> # CUDA example



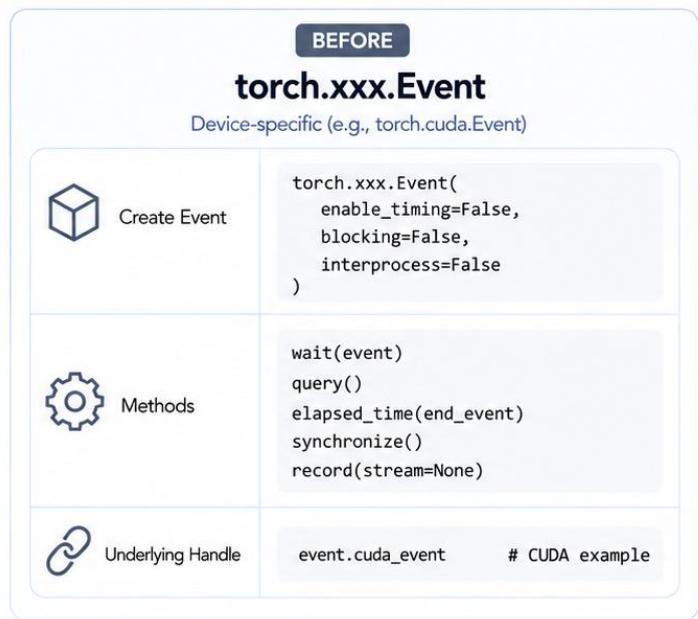
UNIFY

From device-specific
to any accelerator

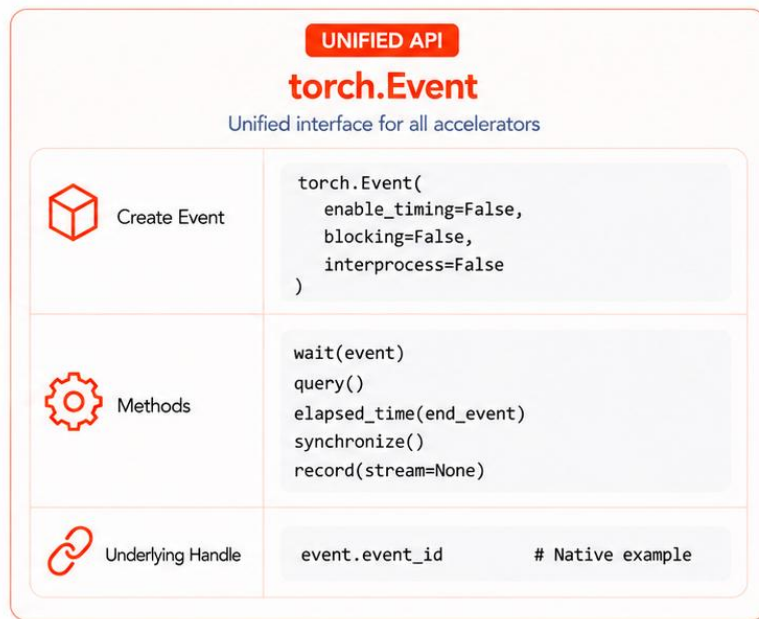
UNIFIED API	
torch.Stream	
Unified interface for all accelerators	
 Create Stream	<code>torch.Stream(device=None, priority=0)</code>
 Methods	<code>wait_stream(other_stream) wait_event(event) record_event(event) synchronize() is_capturing() query()</code>
 Stream Context	<code>with stream: # operations on the stream ...</code>
 Underlying Handle	<code>stream.native_handle</code> # Native example

Appendix: torch.accelerator API List

One API for Event across accelerators



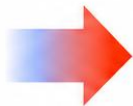

UNIFY
From device-specific
to any accelerator



Appendix: torch.accelerator API List

One API for Graph across accelerators

BEFORE	
torch.xxx.Graph	
Device-specific (e.g., torch.cuda.Graph)	
 Create Graph	<code>torch.xxx.Graph(keep_graph=False, ...)</code>
 Methods	<code>capture_begin() capture_end() replay() reset() instantiate() pool() debug_dump()</code>
 Graph Context	<code>with torch.xxx.graph(graph): # operations captured by the graph ...</code>



UNIFY

From device-specific
to any accelerator

UNIFIED API	
torch.accelerator.Graph	
Unified interface for all accelerators	
 Create Graph	<code>torch.accelerator.Graph(keep_graph=False, ...)</code>
 Methods	<code>capture_begin() capture_end() replay() reset() instantiate() pool() debug_dump()</code>
 Graph Context	<code>with graph: # operations captured by the graph ...</code>