Pluggable PyTorch LLM Inference Architecture with vLLM and AWS Neuron Backends

PyTorch 2026, Paris

Yahav Biran; yahavb@amazon.com

From Monolithic to Modular: Enabling Hardware Diversity in LLM Serving

- LLM serving has matured beyond single-vendor solutions
- Hardware accelerator diversity is increasing rapidly
- Challenge: How do we integrate new hardware without fragmenting codebases?
- Solution: Modular plugin architectures with standardized interfaces

TorchNeuron: Native PyTorch Backend for Trainium

The Foundation for Pluggable Inference

- TorchNeuron integrates via PyTorch's PrivateUse1 device backend mechanism
- Registers Trainium as torch.device('neuron') alongside CUDA, XPU, MPS
- Operations automatically dispatched through PyTorch's standard dispatcher
- Open source: github.com/aws-neuron/torch-neuronx

The Native PyTorch Experience

Frictionless Device Switching

GPU

```
model = MyModel().to('cuda')
optimizer = torch.optim.AdamW(model.parameters())
for batch in dataloader:
    output = model(batch)
    loss.backward()
    optimizer.step()
```

Trainium

```
model = MyModel().to('neuron')  # <-- Only change needed
optimizer = torch.optim.AdamW(model.parameters())
for batch in dataloader:
    output = model(batch)
    loss.backward()
    optimizer.step()
```

The Native PyTorch Experience

Two Execution Modes on Trainium

Eager Mode and torch.compile

Mode	Best For	Characteristics	NKI Kernel Usage
Eager	Research, prototyping, debugging	Imperative execution, intuitive debugging	Novel research, custom ops
torch.compile	Production training/inference	JIT compilation, optimizations	Maximize performance

Eager mode: Operations execute immediately when called

torch.compile: @torch.compile(backend="neuron") for production performance

Both modes support NKI custom kernels

The Native PyTorch Experience

Neuron Kernel Interface (NKI)

```
# PyTorch (compiler does everything)
import torch
c = torch.add(a, b) # Compiler might
spill c back to HBM between ops
```

```
# NKI (you control structure)
import nki
import nki.language as nl
@nki.jit
def nki_add(a, b):
    a = nl.load(a) #HBM->SBUF
    b = nl.load(b) #HBM->SBUF
    c = nl.add(a, b)
    nl.store(...) #SBUF->HBM
```

```
# ISA (you control HOW)
import nki
import nki.isa as nisa
@nki.jit
def isa_add(a, b):
    nisa.dma_copy(...)
    nisa.tensor_tensor(..., op=nl.add)
    nisa.dma_copy(...)
```

The Native PyTorch Experience

torch.compile with Neuron Backend

Compile specific parts or entire model:

```
@torch.compile(backend="neuron")
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b

# Same API as CUDA - just specify backend="neuron"
print(foo(torch.randn(3, 3), torch.randn(3, 3)))
```

The Native PyTorch Experience

torch.compile with Neuron Backend

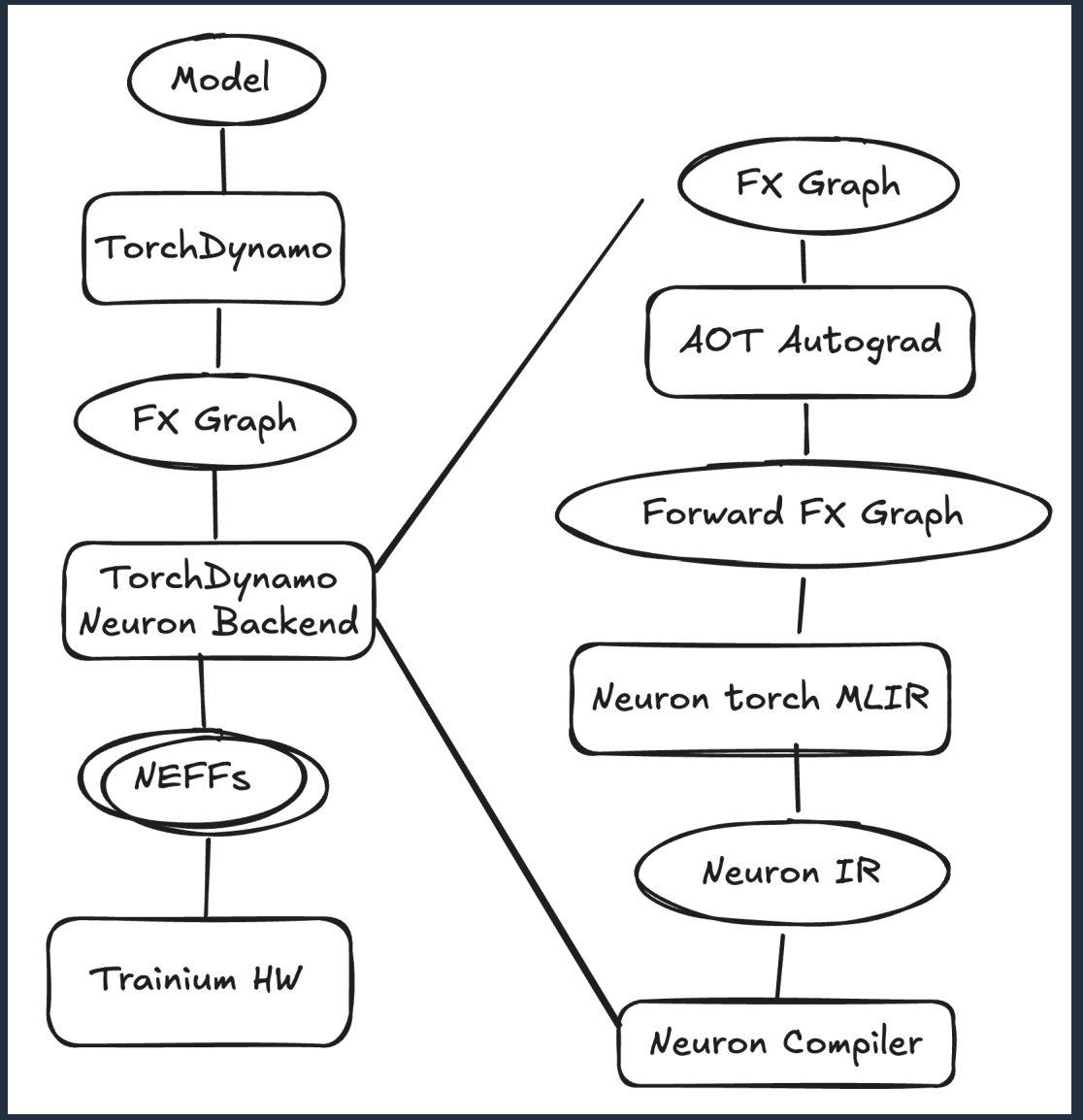
Compilation Flow:

TorchDynamo/Python bytecode → FX Graph

Generates forward and backward graphs

FX graphs → lowers to Neuron IR

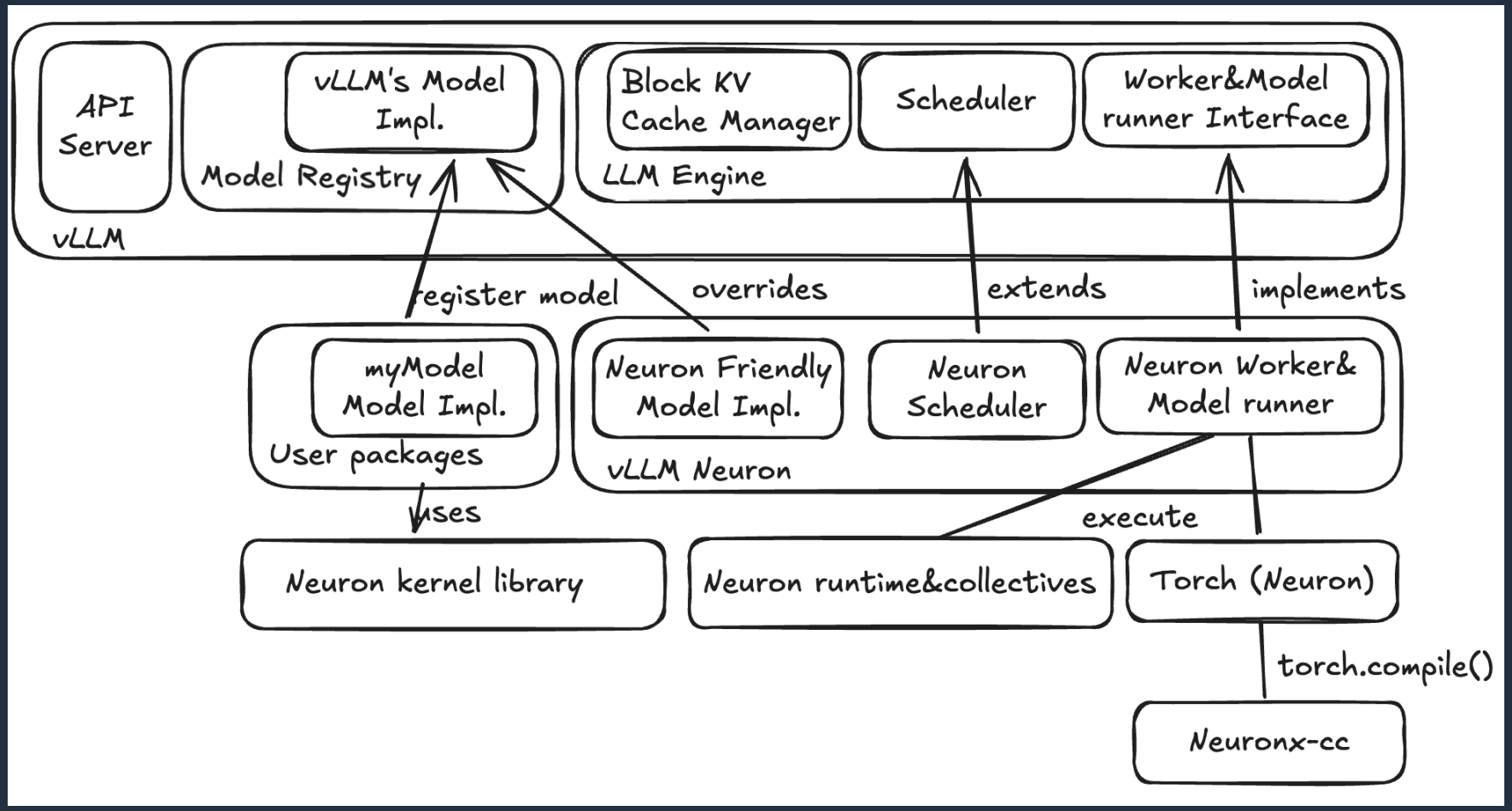
Compiler generates Trainium instructions



vLLM Hardware Plugin Architecture

- Python Entry Points Enable Automatic Platform Detection
- Hardware vendors extend PyTorch inference without forking the codebase
- entry_points mechanism for automatic device detection
- Modular feature development
- Seamless integration with PyTorch's model loading patterns

Architecture Overview



Key Takeaway

Pluggable Architecture Matters

Performance Portability, Not Just Portability

- **Same model definition, hardware-optimal execution**
- **Fusion strategies matched to hardware profiles**

Clean Abstraction Boundaries

- **Bucketing in scheduler, not model code**
- **Vendors extend, don't fork**
- **Standard workflows preserved**

Ecosystem Growth

- **Hardware vendors can participate in PyTorch inference**
- **Users benefit from hardware diversity**
- **Innovation without fragmentation**