



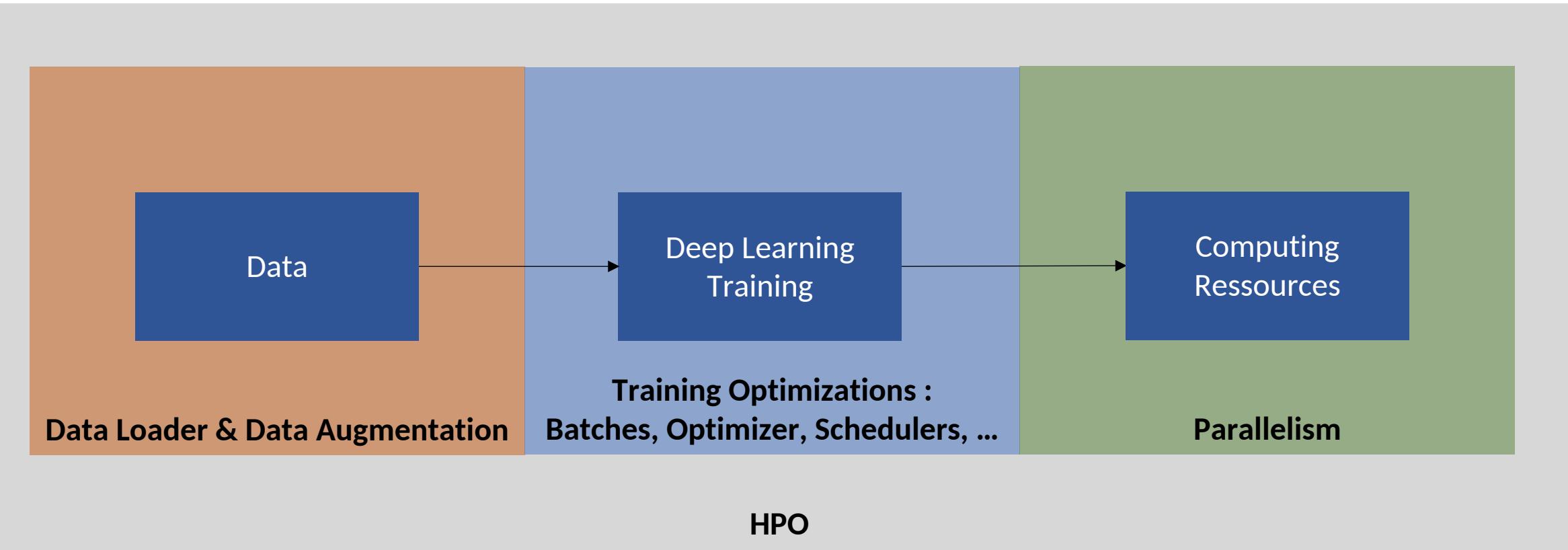
# Optimised Deep Learning on Jean Zay (DLO-JZ)

## Torch & Compilation



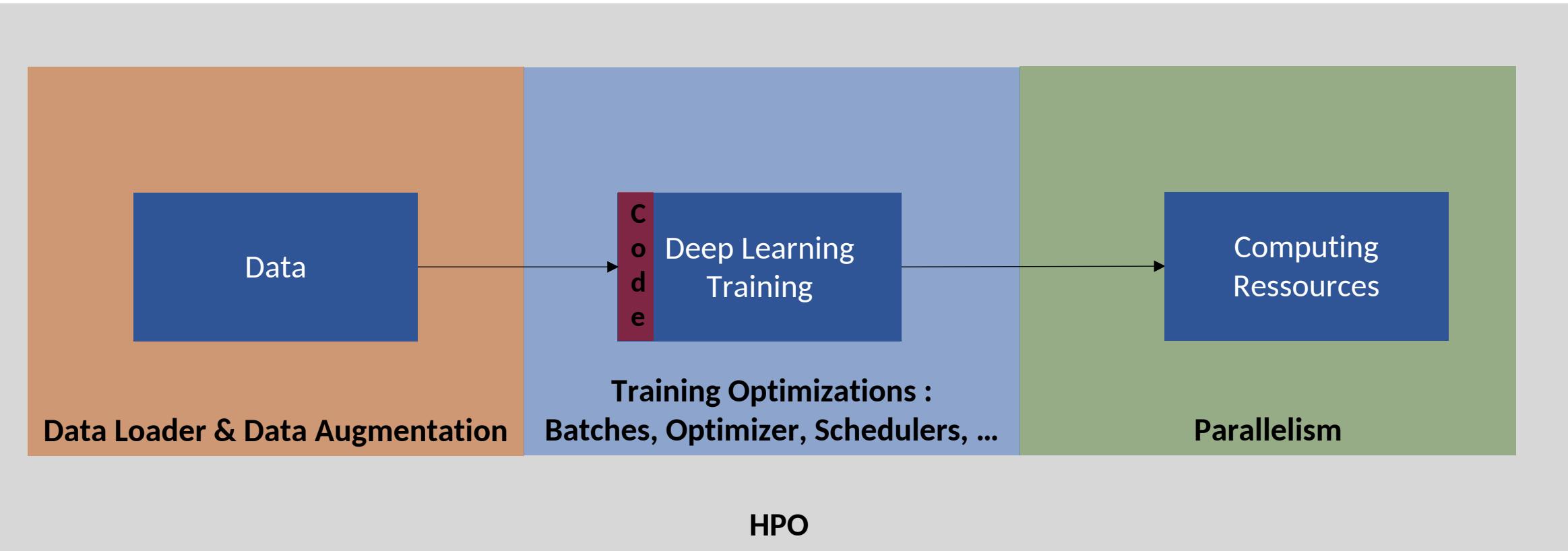
INSTITUT DU  
DÉVELOPPEMENT ET DES  
RESSOURCES EN  
INFORMATIQUE  
SCIENTIFIQUE





Optimize everything ?



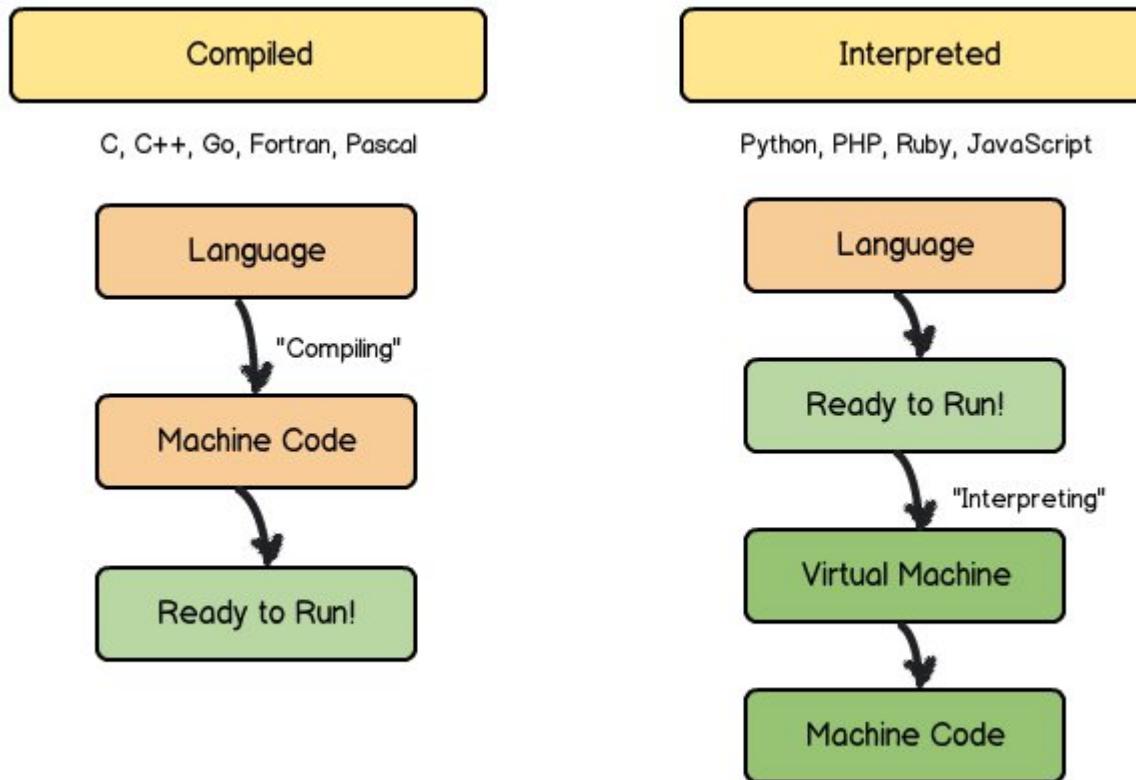


Optimize everything !



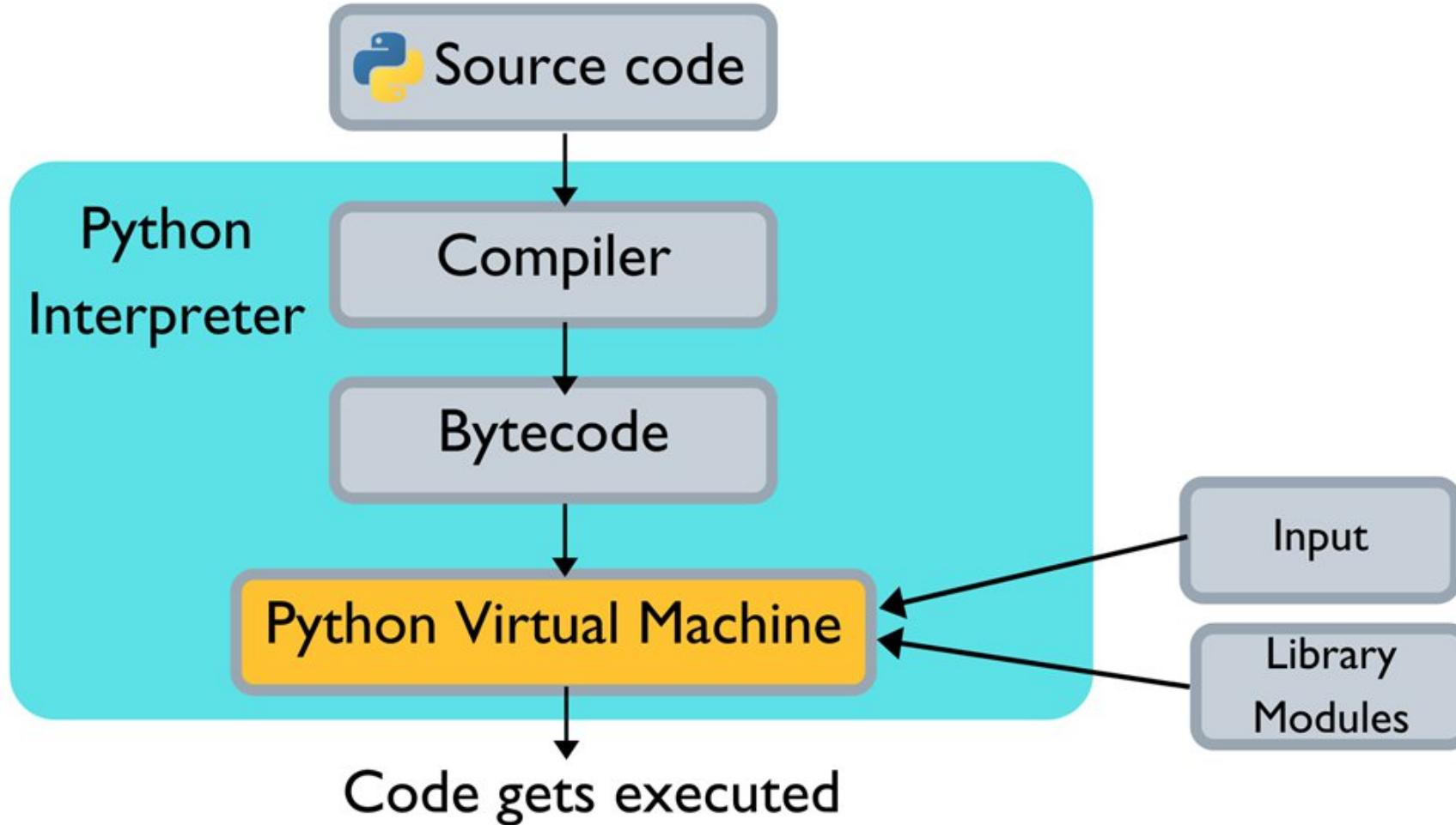
```
helloworld.c
1 #include <stdio.h>
2
3 int main (void) {
4     printf ("Hello, World!\n");
5
6     return 0;
7 }
```

```
hello.py
1 msg = "Hello World"
2 print(msg)
```



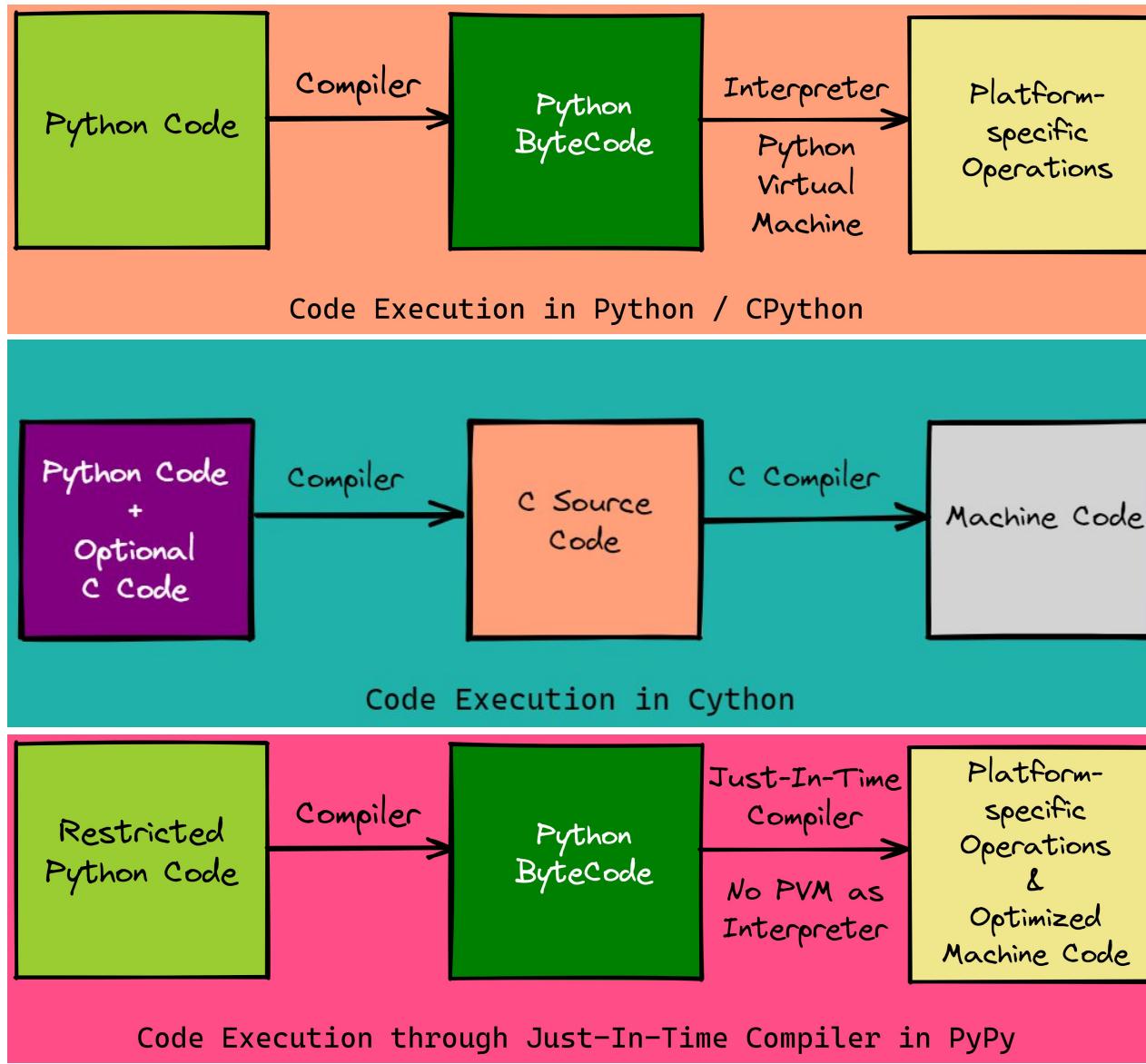
Code?



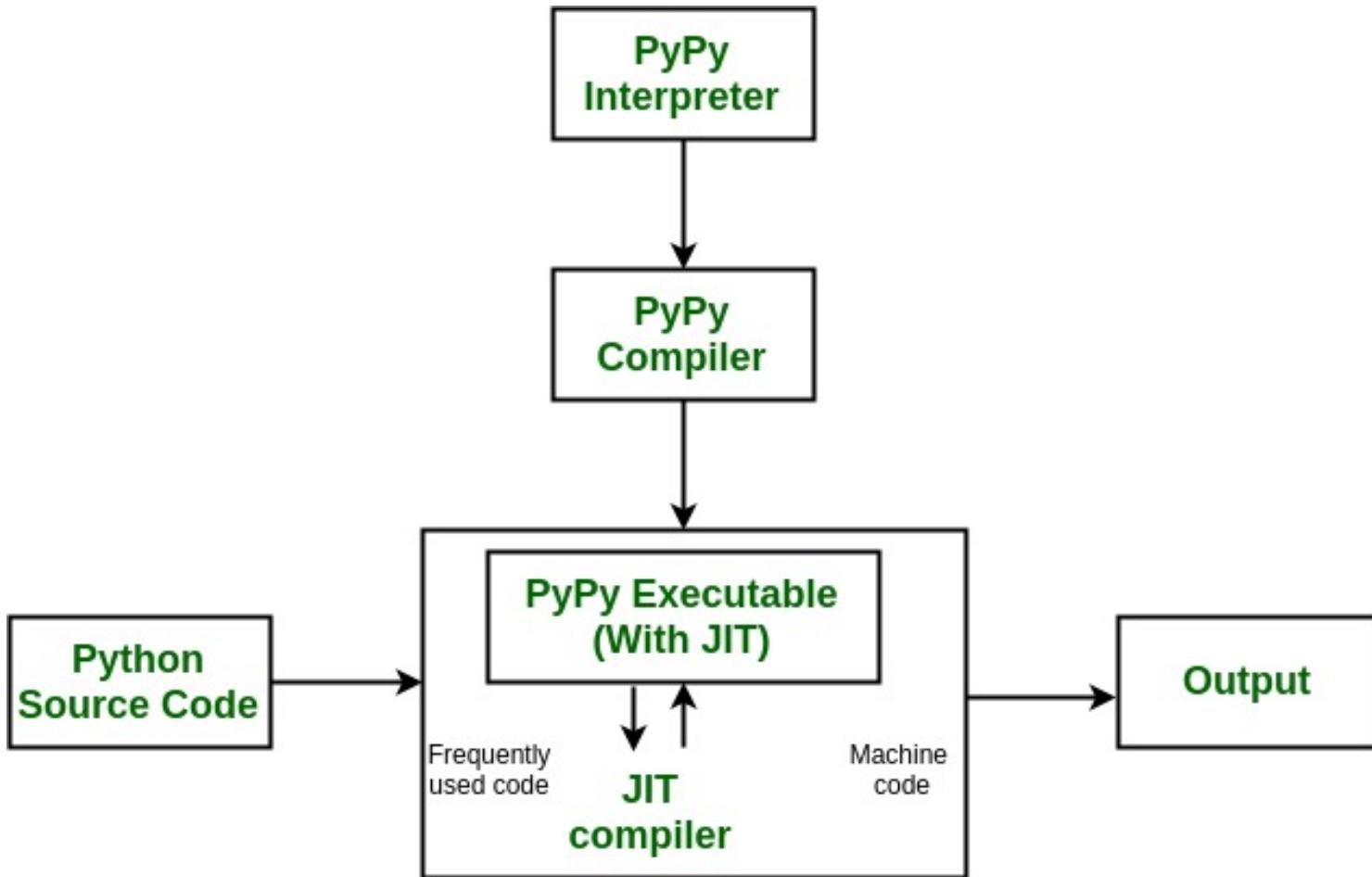


Python ? CPython !

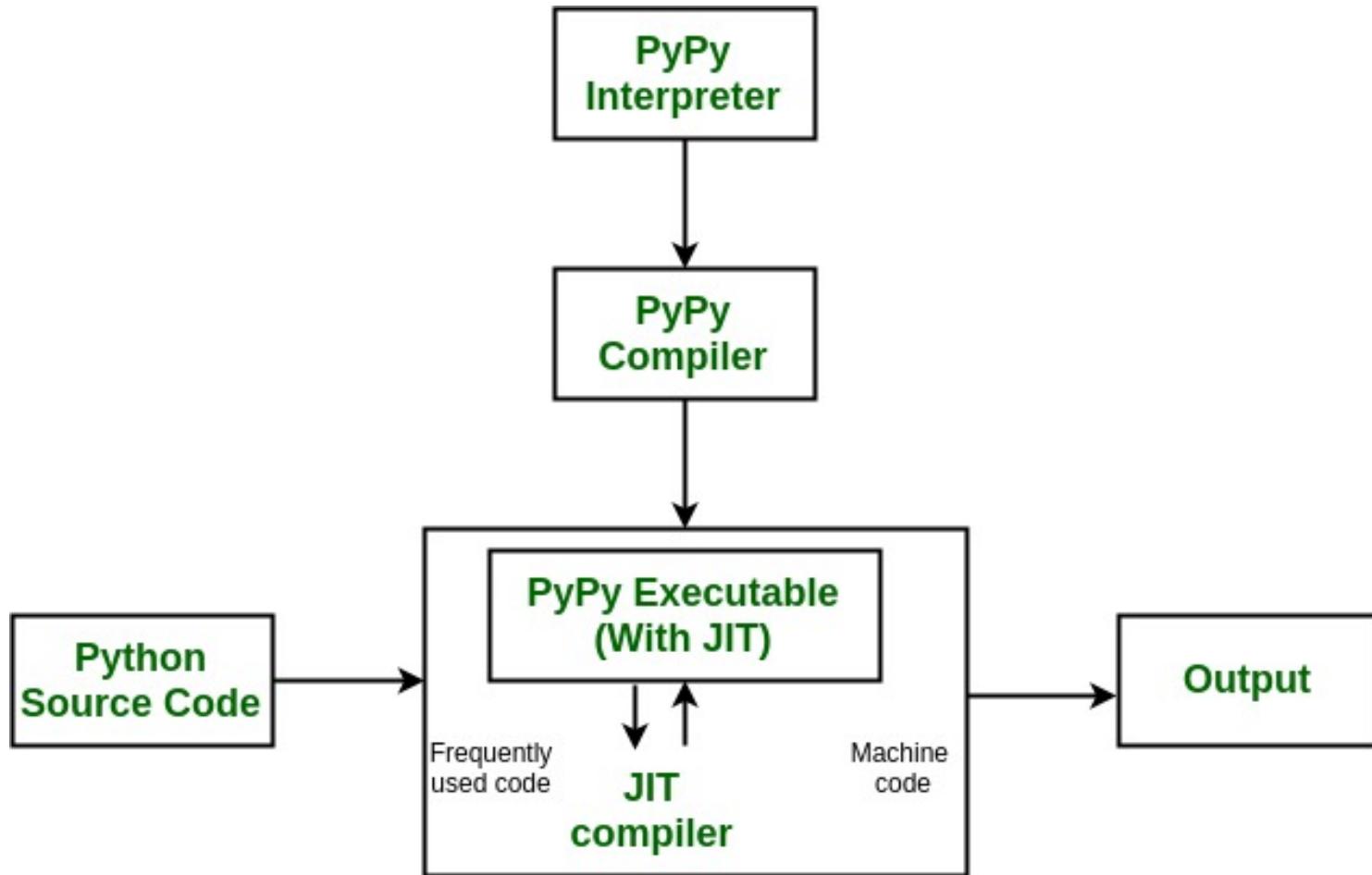




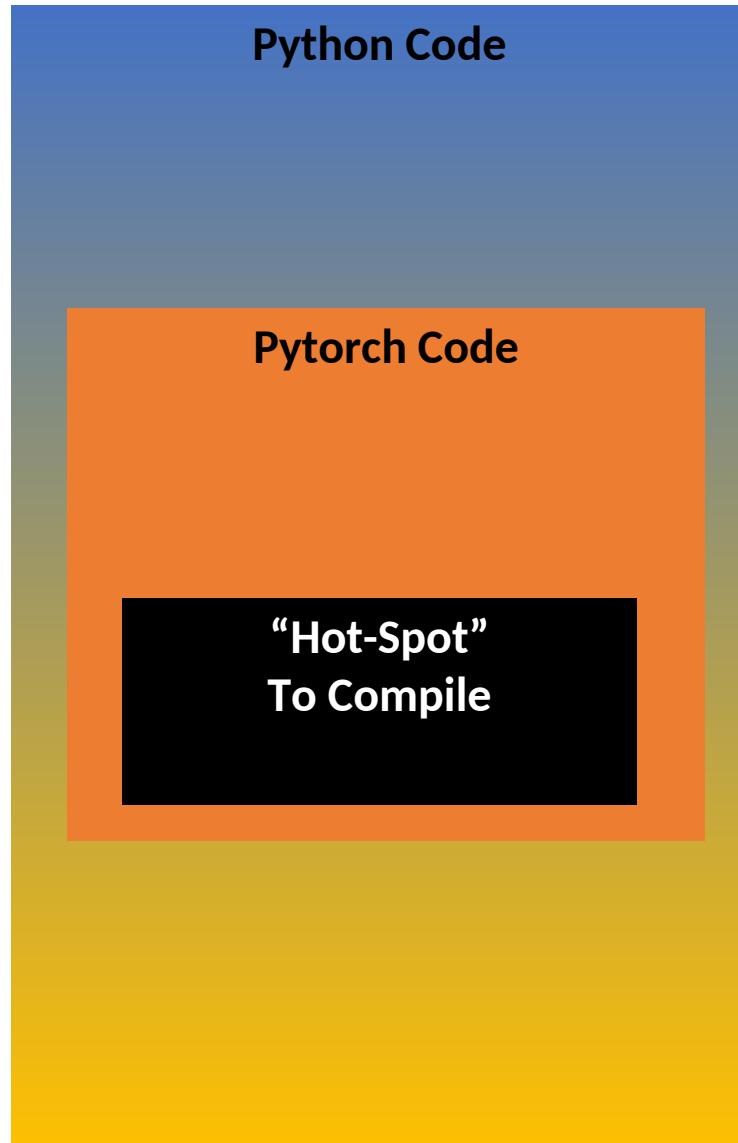
# Cpython, Jython, Cython, PyPy



## Just-In-Time Compilation



## Just-In-Time Compilation – Python Level



**Internal Pytorch JIT:**

**TorchScript**

**Torch Compile**

**Just-In-Time Compilation – Pytorch Level**

```
import torch
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.linear(x) + h)
        return new_h, new_h

x, h = torch.rand(3, 4), torch.rand(3, 4)
my_cell = MyCell()
my_cell(x, h)
```

## Tracing

```
traced_cell = torch.jit.trace(my_cell, (x, h))
traced_cell(x, h)
print(traced_cell.code)

def forward(self,
            x: Tensor,
            h: Tensor) -> Tuple[Tensor, Tensor]:
    linear = self.linear
    _θ = torch.tanh(torch.add((linear).forward(x, ), h))
    return (_θ, _θ)
```

# TorchScript - Tracing



```
class MyDecisionGate(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x
        else:
            return -x

class MyCell(torch.nn.Module):
    def __init__(self, dg):
        super(MyCell, self).__init__()
        self.dg = dg
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.dg(self.linear(x)) + h)
        return new_h, new_h

my_cell = MyCell(MyDecisionGate())
traced_cell = torch.jit.trace(my_cell, (x, h))
```

*TracerWarning: Converting a tensor to a Python boolean might cause the trace to be incorrect. We can't record the data flow of Python values, so this value will be treated as a constant in the future. This means that the trace might not generalize to other*

## Scripting

```
scripted_gate = torch.jit.script(MyDecisionGate())

my_cell = MyCell(scripted_gate)
scripted_cell = torch.jit.script(my_cell)
```



# TorchScript – Scripting

```

print(f"Non-traced average time: {non_traced_time:.6f} seconds")
print(f"Traced average time: {traced_time:.6f} seconds")
print(f"Scripted average time: {traced_time:.6f} seconds")

Non-traced average time: 0.000030 seconds
Traced average time: 0.000038 seconds
Scripted average time: 0.000038 seconds

```

	Latency on CPU (ms)	Latency on GPU(ms)
PyTorch	86.23	16.49
TorchScript	81.57	10.54

PyTorch vs TorchScript for BERT

	Latency on CPU (ms)	Latency on GPU(ms)
PyTorch	25.96	4.02
TorchScript	23.01	2.41

PyTorch vs TorchScript for ResNet

## Save in Python

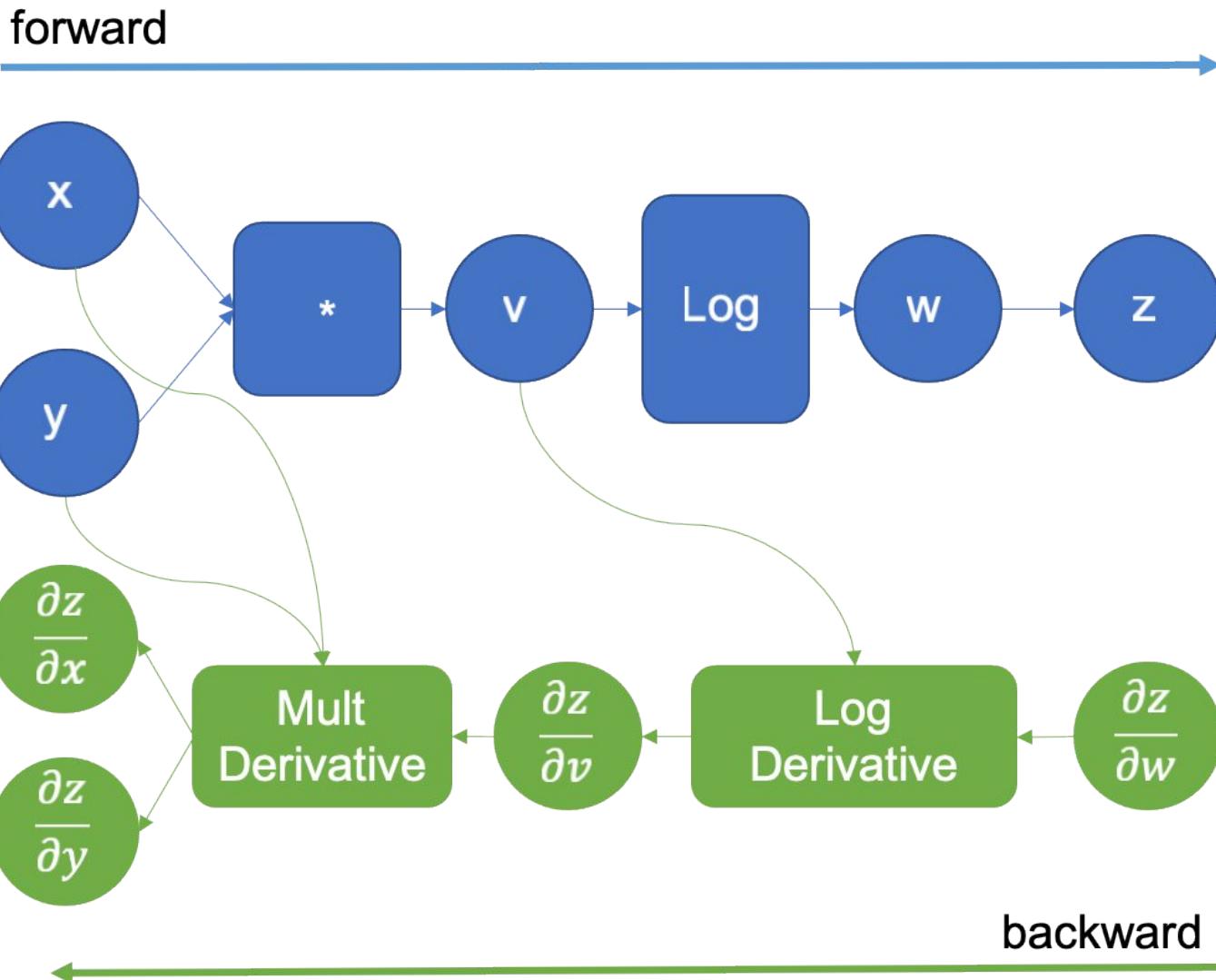
```
traced_cell.save('wrapped_cell.pt')
```

## Load in C++

```
#include <torch/script.h>
...
torch::jit::script::Module module;
...
module = torch::jit::load(model_path);
...
```

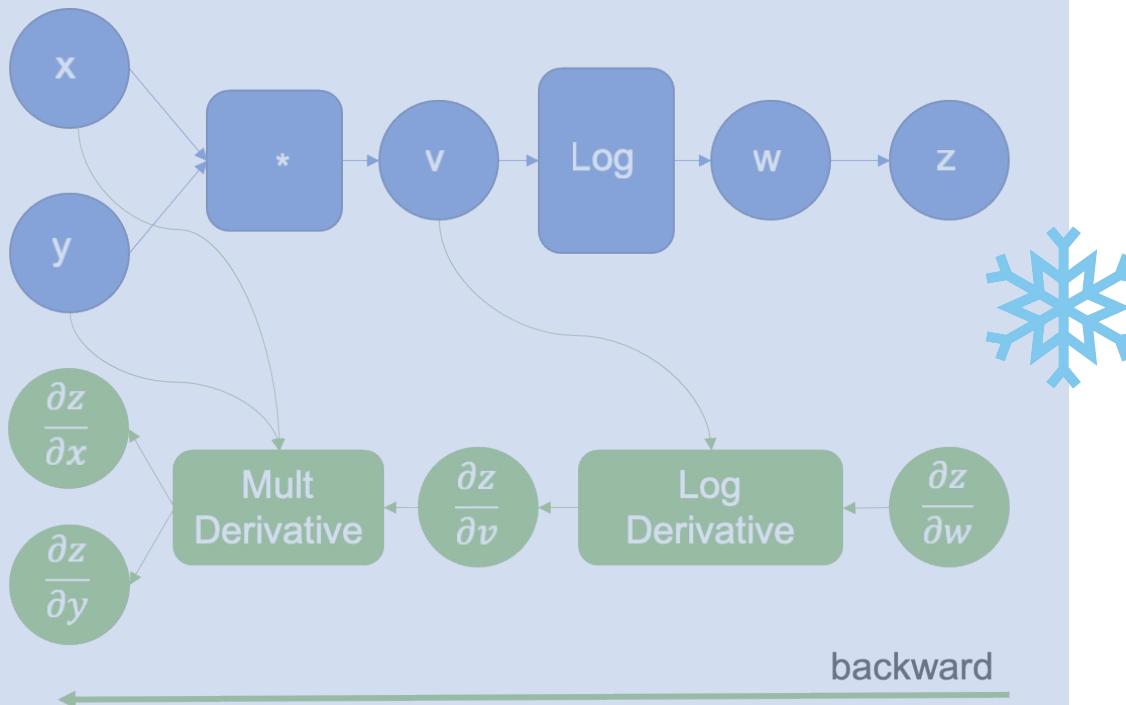
# TorchScript – Save & Load





Torch AutoGrad

forward



```
import torch
def fn(a, b, c, d):
    x = a + b + c + d
    return x.cos().cos()
a, b, c, d = [torch.randn(2, 4, requires_grad=True) for _ in range(4)]
ref = fn(a, b, c, d)
loss = ref.sum()
loss.backward()
```

```
from functorch.compile import aot_function

def compiler_fn(fx_module: torch.fx.GraphModule, _):
    print(fx_module.code)
    return fx_module

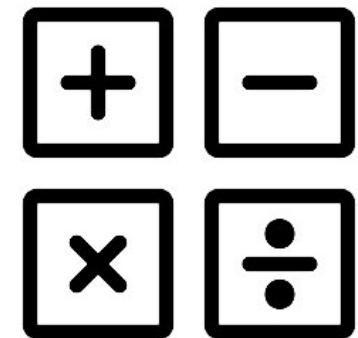
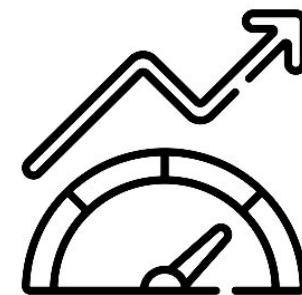
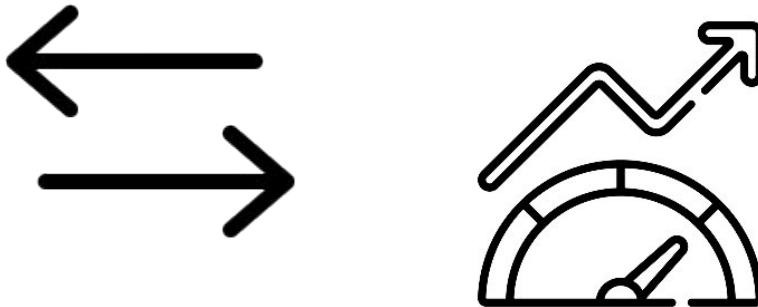
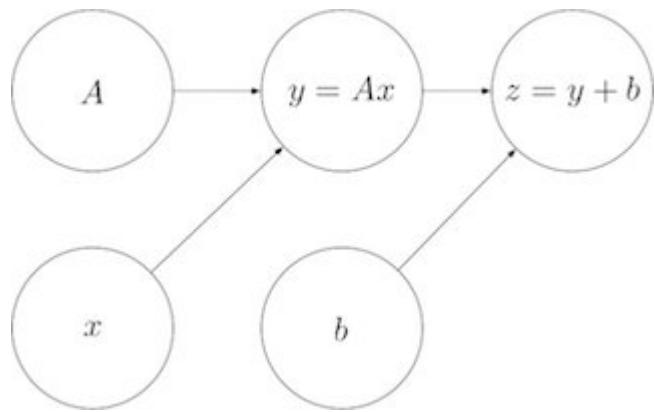
aot_print_fn = aot_function(fn, fw_compiler=compiler_fn, bw_compiler=compiler_fn)

# To trigger the compilation and see the result:
res = aot_print_fn(a, b, c, d)
```

```
def forward(self, primals_1, primals_2, primals_3, primals_4):
    add = torch.ops.aten.add.Tensor(primals_1, primals_2); primals_1 = primals_2 = None
    add_1 = torch.ops.aten.add.Tensor(add, primals_3); add = primals_3 = None
    add_2 = torch.ops.aten.add.Tensor(add_1, primals_4); add_1 = primals_4 = None
    cos = torch.ops.aten.cos.default(add_2)
    cos_1 = torch.ops.aten.cos.default(cos)
    return [cos_1, add_2, cos]
```

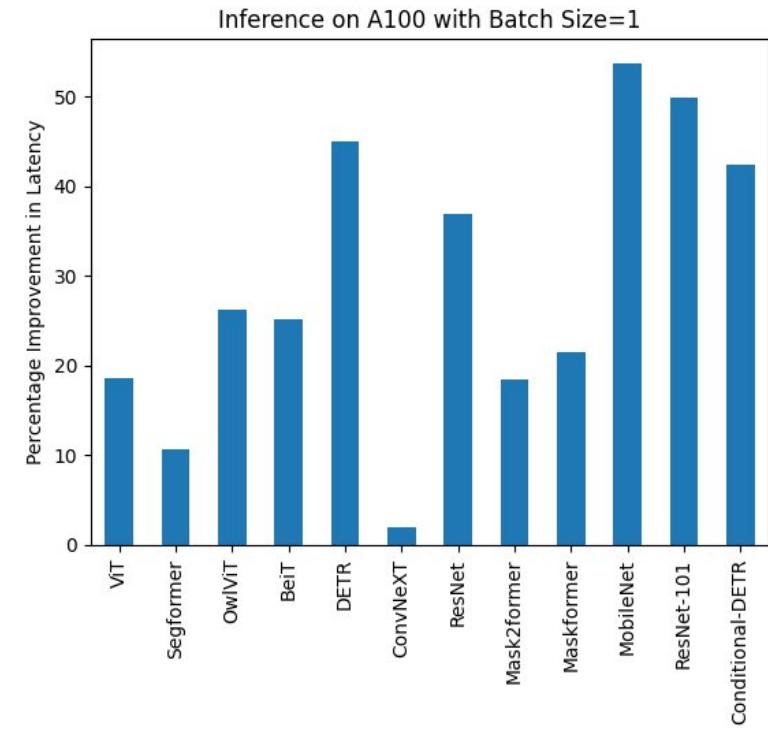
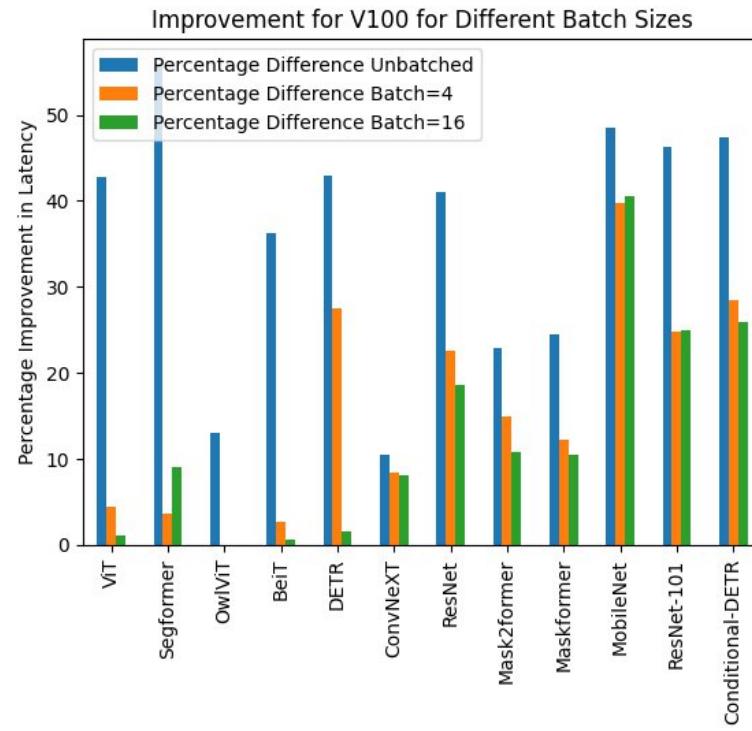
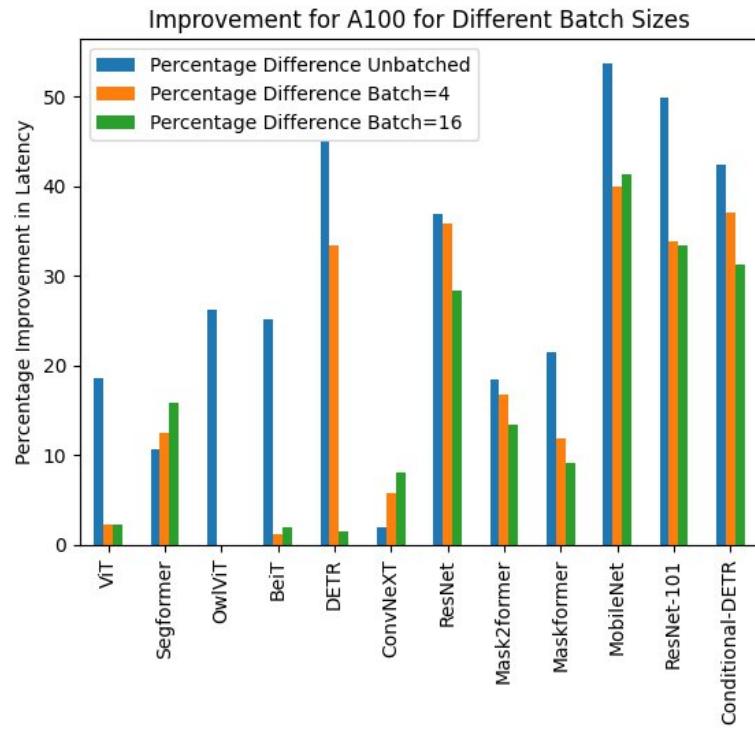
```
torch.compile(model=None, *, fullgraph=False, dynamic=None, backend='inductor', mode=None,  
options=None, disable=False) [SOURCE]
```

## TorchDynamo + AOT AutoGrad + TorchInductor + PrimTorch



### Torch Compile - Mechanisms

# Torch Compile - Performances



Compilation mode	Initial step time [s]	Initial step memory [MiB]	Training time / iter [ms]	Inference time / iter [ms]
N/A (eager)	1	3675	57	<b>18</b>
default	29	3277	34	30
reduce-overhead	29	5736	34	28
max-autotune	35	5736	<b>32</b>	30

## Torch Compile - Performances



```
import numpy as np

def kmeans(X, means):
    return np.argmin(np.linalg.norm(X - means[:, None], axis=2), axis=0)

npts = 10_000_000
X = np.repeat([[5, 5], [10, 10]], [npts, npts], axis=0)
X = X + np.random.randn(*X.shape) # 2 distinct "blobs"
means = np.array([[5, 5], [10, 10]])
np_pred = kmeans(X, means)
```

## Torch Compile Wrapping

```
import torch

compiled_fn = torch.compile(kmeans)
compiled_pred = compiled_fn(X, means)
assert np.allclose(np_pred, compiled_pred)
```

## Torch Compile Performance

```
print(f"Non-compiled average time: {non_compiled_time:.6f} seconds")
print(f"Compiled average time: {compiled_time:.6f} seconds")

Non-compiled average time: 2.521456 seconds
Compiled average time: 0.233836 seconds
```



# Torch Compile - numpy

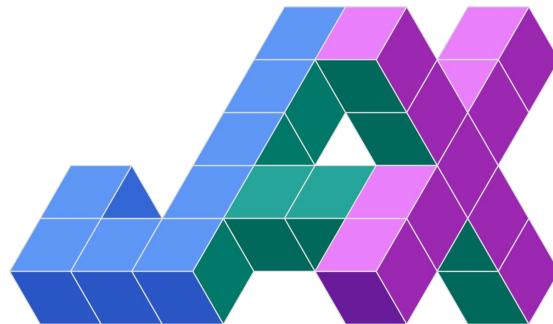


RAPIDS

Accelerated with  
 NVIDIA.



CuPy



« Numpy » on gpus



# Flax Linen

*Neural networks with JAX*

Flax Linen delivers an **end-to-end and flexible user experience for researchers who use JAX with neural networks**. Flax exposes the full power of [JAX](#). It is made up of loosely coupled libraries, which are showcased with end-to-end integrated [guides](#) and [examples](#).

## jax.jit

```
jax.jit(fun, in_shardings=UnspecifiedValue, out_shardings=UnspecifiedValue,  
static_argnums=None, static_argnames=None, donate_argnums=None, donate_argnames=None,  
keep_unused=False, device=None, backend=None, inline=False, abstracted_axes=None)  
Sets up fun for just-in-time compilation with XLA.
```

[\[source\]](#)

## GPU memory allocation

JAX will preallocate 75% of the total GPU memory when the first JAX operation is run. Preallocating minimizes allocation overhead and memory fragmentation, but can sometimes cause out-of-memory (OOM) errors. If your JAX process fails with OOM, the following environment variables can be used to override the default behavior:

**XLA\_PYTHON\_CLIENT\_PREALLOCATE=false**

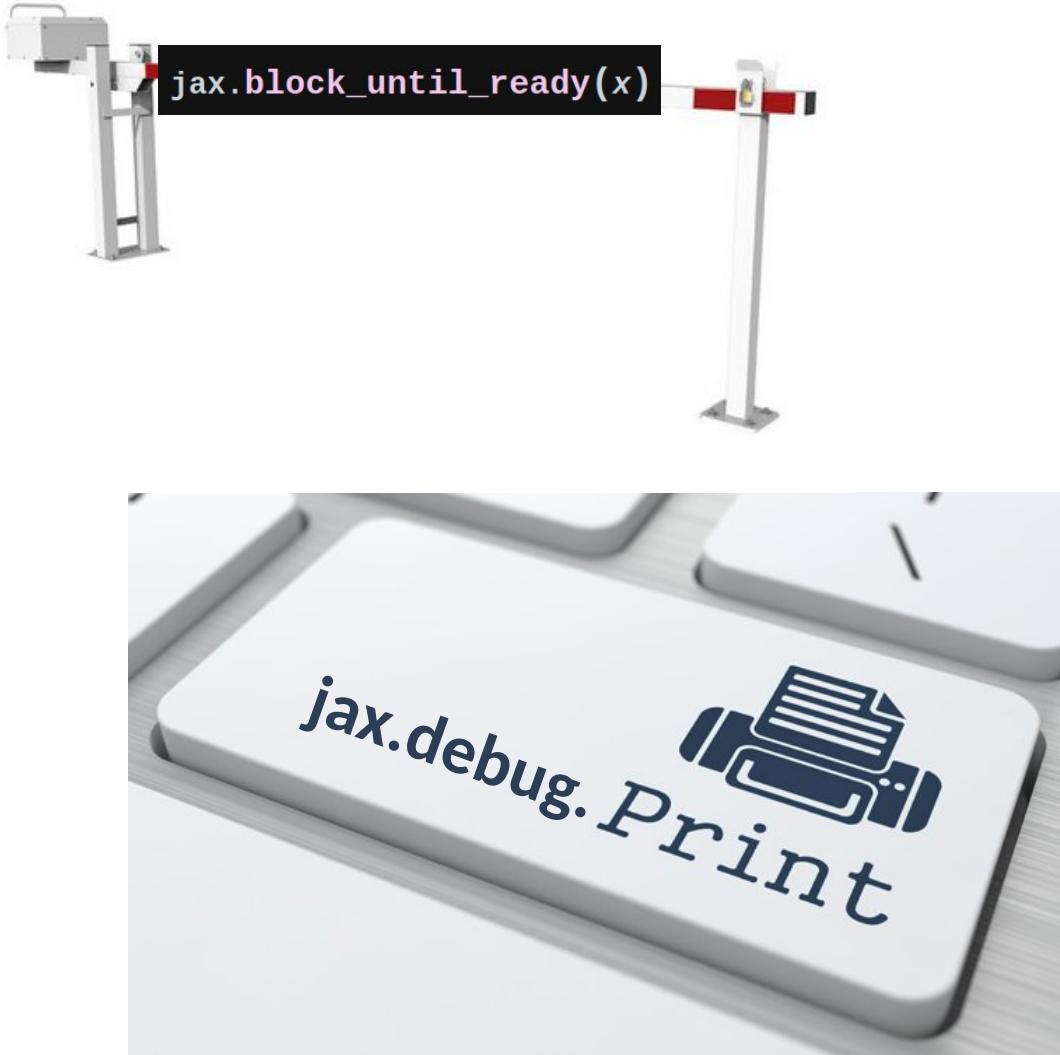
This disables the preallocation behavior. JAX will instead allocate GPU memory as needed, potentially decreasing the overall memory usage. However, this behavior is more prone to GPU memory fragmentation, meaning a JAX program that uses most of the available GPU memory may OOM with preallocation disabled.

JAX tips



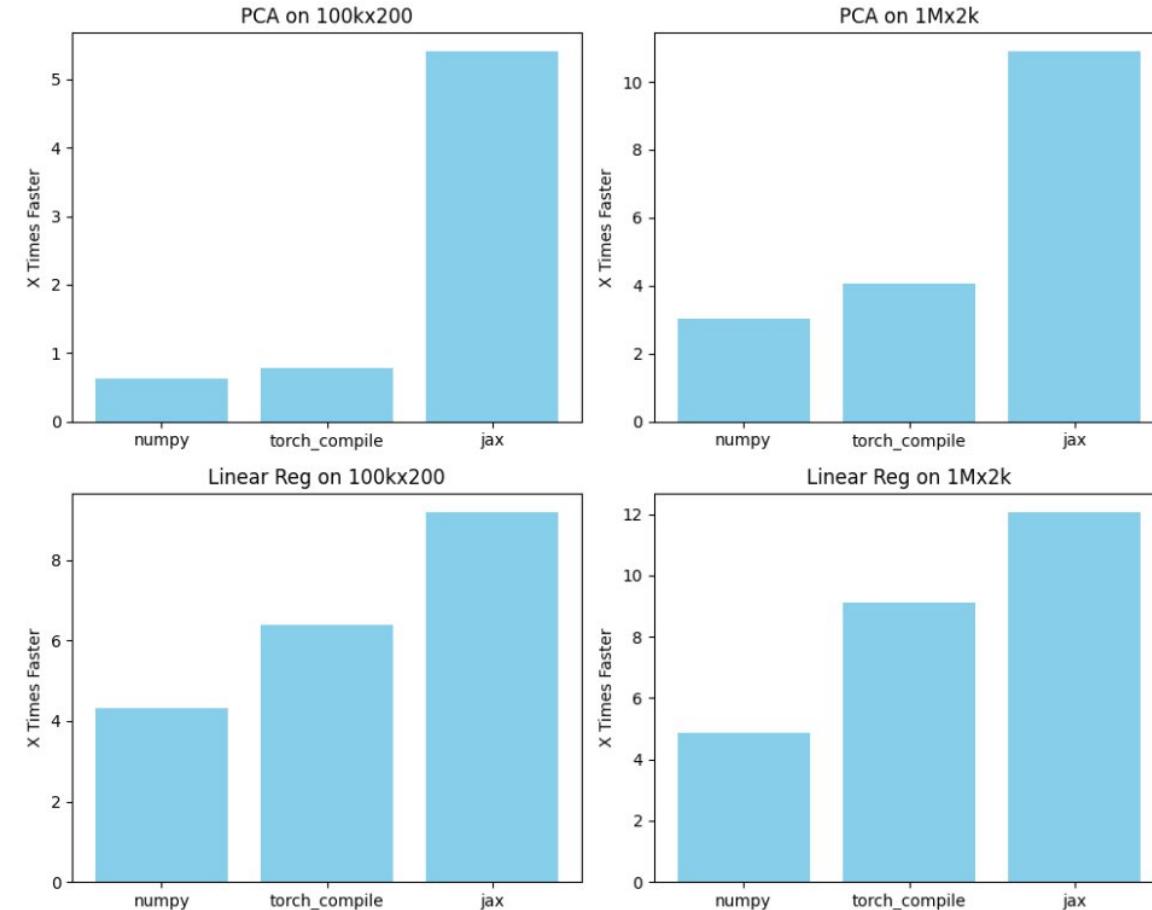


## BENCHMARKING

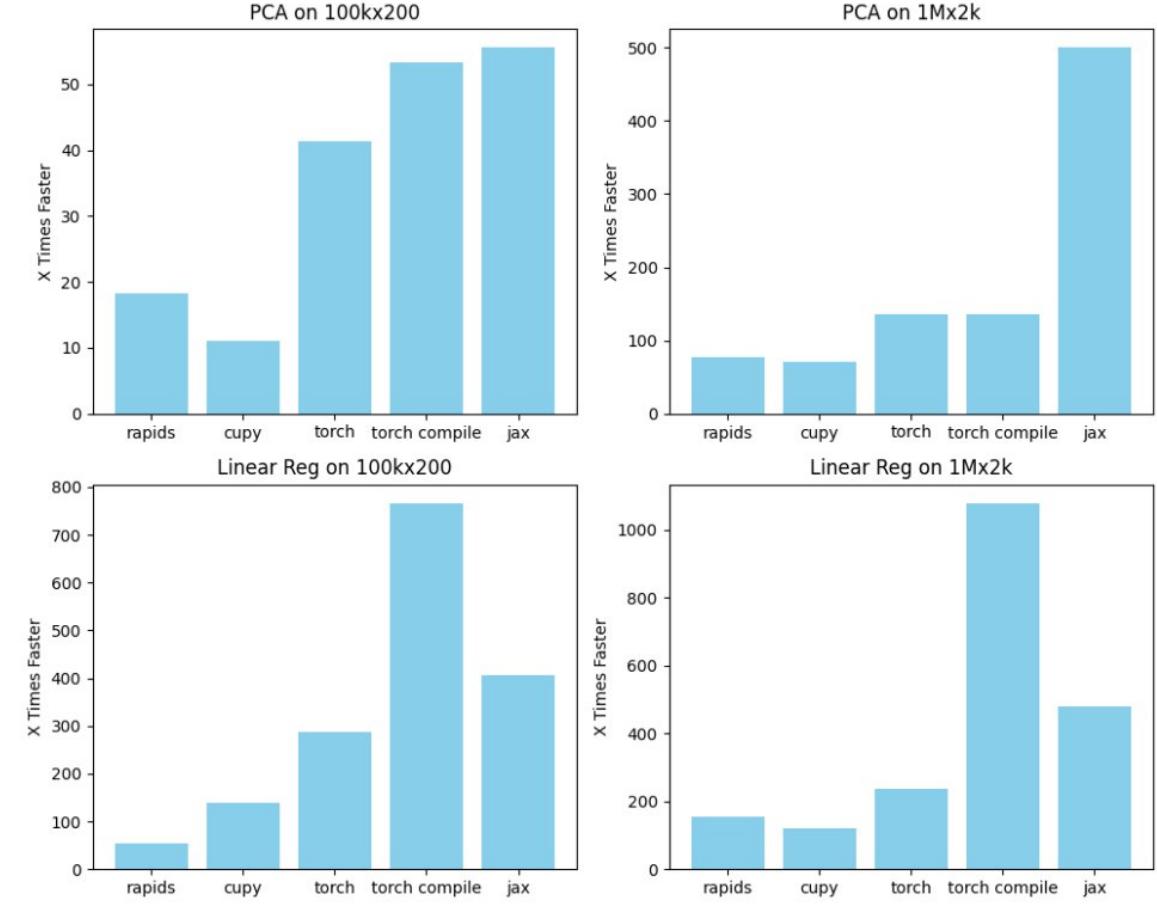


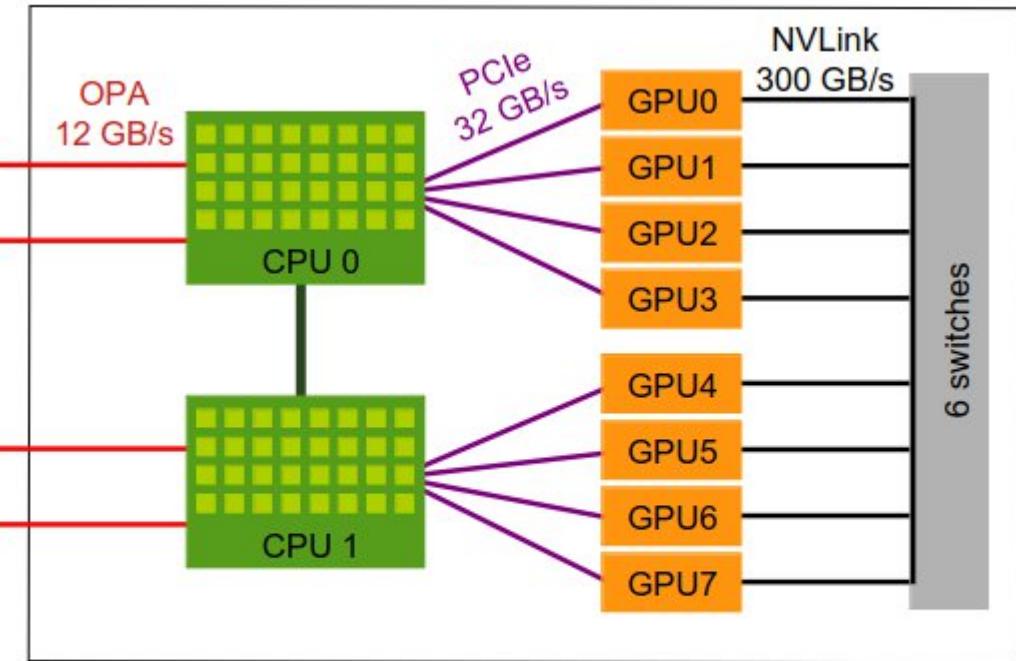
JAX warnings

## On CPU



## On GPU





Node  $8 \times$  A100 80Go

Really worth it ?

