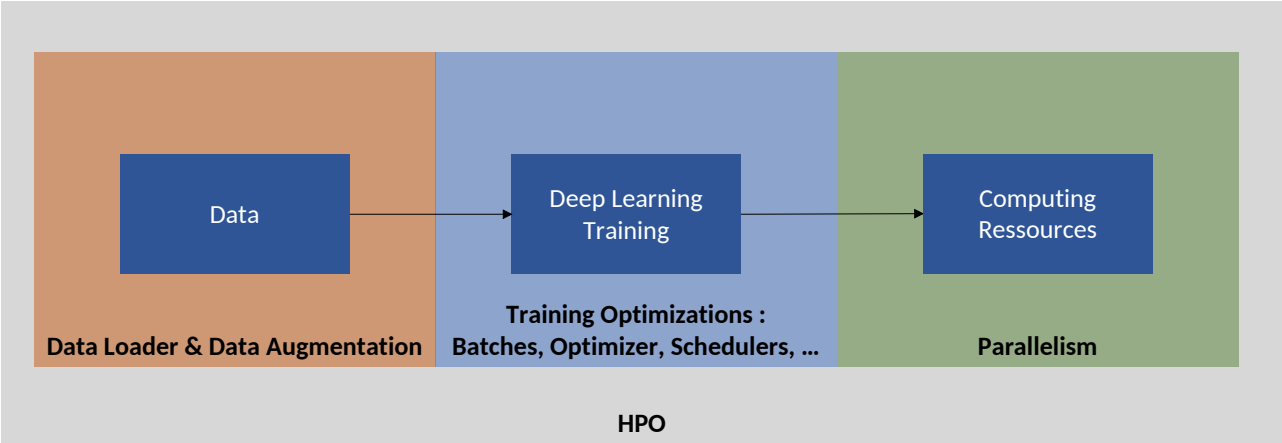


Optimised Deep Learning on Jean Zay (DLO-JZ)

Torch & Compilation

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

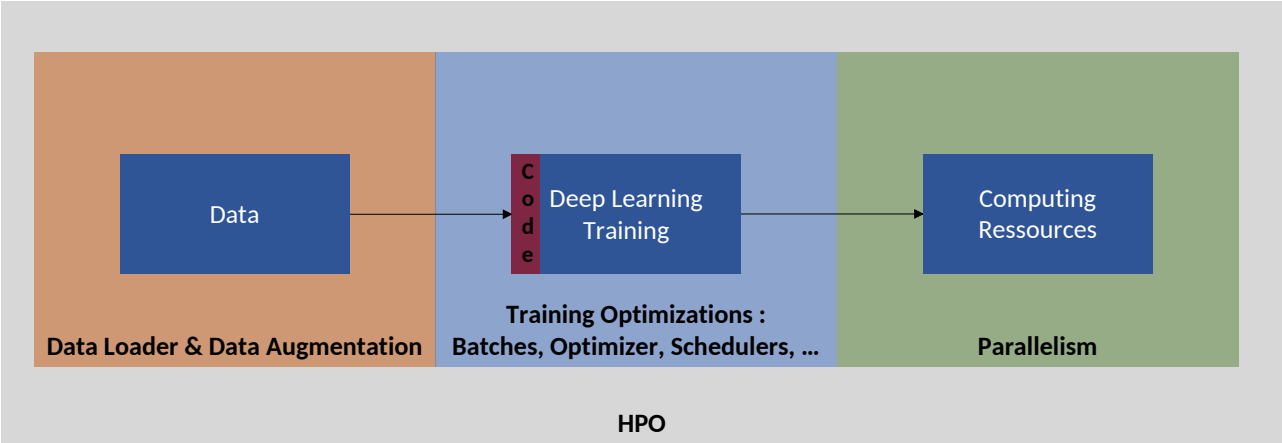
BY NC SA



Optimize everything ?

We have seen that there are a number of ways in which we can optimise our training pipeline.

– Optimising CPU usage via DataLoader settings
– Optimising the quality of the training loop by improving the network's ability to learn
– Making the best possible use of the computational resources at our disposal



Optimize everything !

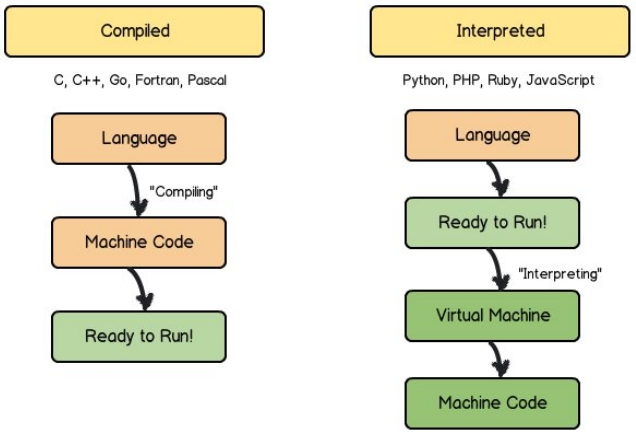
However, we can go a step further and optimise the actual execution of the code at a lower level.

```c
#include <stdio.h>

int main (void) {
    printf ("Hello, World!\n");

    return 0;
}
```

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

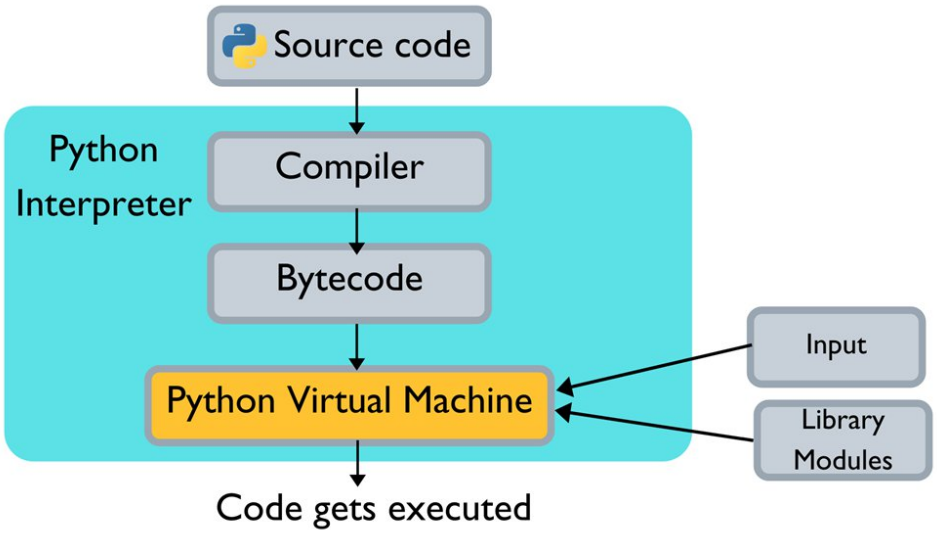


Code?

Difference between compiled and interpreted languages:

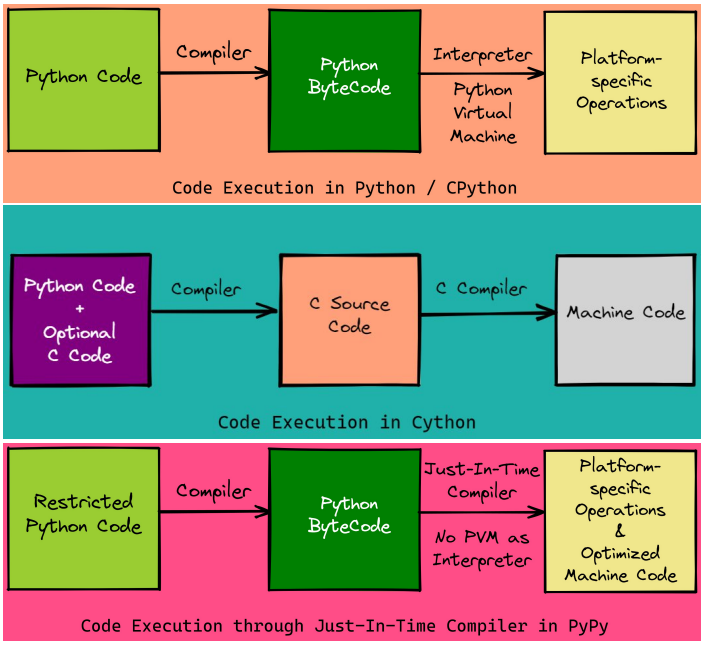
– A compiled language is a programming language that is generally compiled rather than interpreted. The compiled programme is expressed in machine code, which is unintelligible to humans.

– An interpreted language is a programming language that is executed without first being compiled into machine code. The instructions are not executed directly by the target machine, but are read and executed by another programme.



Python ? CPython !

The most widely used implementation of Python is ‘CPython’. CPython acts as both a compiler and an interpreter, as it first compiles Python code into an intermediate ‘bytecode’ format, also known as ‘portable code’.

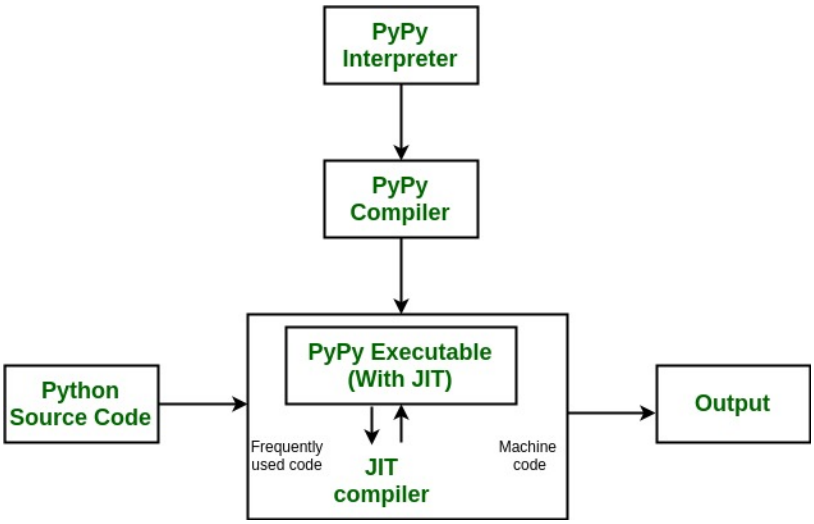


Cpython, Jython, Cython, PyPy

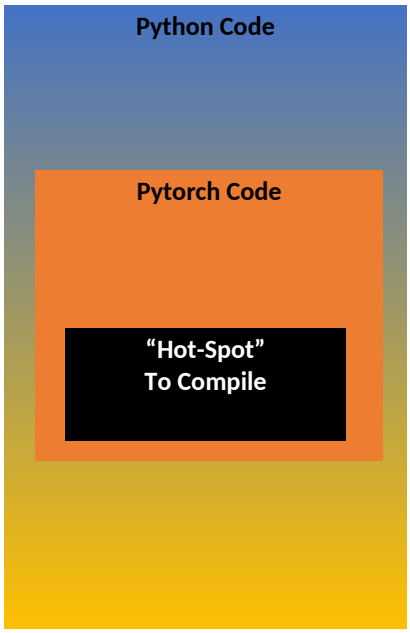
Bytecode is a set of instructions that can either be executed as is by an interpreter, or further compiled into machine code for improved performance.

Certain systems known as ‘Just-In-Time’ compilers (or ‘on-the-fly’ compilers) compile the instructions at runtime.

A system implementing a JIT compiler generally analyses the code being executed on an ongoing basis and identifies sections of code where the speed-up gained during compilation would outweigh the overhead of that compilation.



PyPy is a Python interpreter equipped with a JIT compiler. It is an alternative Python executable (an alternative to CPython, in fact).



Internal Pytorch JIT:

TorchScript

Torch Compile

Just-In-Time Compilation – Pytorch Level

The advantage of PyTorch is that there is no need to compile everything on the fly: we know which parts will be executed repeatedly: our model

PyTorch therefore incorporates ‘JIT’ compilation, in the form of TorchScript and `torch.compile`.

```python
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

```python
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
  _0 = torch.tanh(torch.add((linear).forward(x, ), h))
  return (_0, _0)
```

TorchScript – Tracing

La première manière de faire de torchscript est le tracing.

Torch.jit.trace prend en argument un code à tracer et le lire ligne par ligne pour le traduire.

```python
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

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

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

TorchScript – Scripting

La deuxième manière de faire de torchscript et le scripting.

On exécute un code pour créer un « graphe de calcul » qui remplacera l'exécution classique de la fonction.

Le problème est que ce graphe est statique et ne permet pas d'intégrer par exemple des conditions booléennes. L'exécution initiale déterminera le graphe.

```
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

TorchScript – Performance

Ces deux méthodes améliorent quelques peu les performances, mais avec pas mal de variance dans le temps gagné.

Save in Python

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

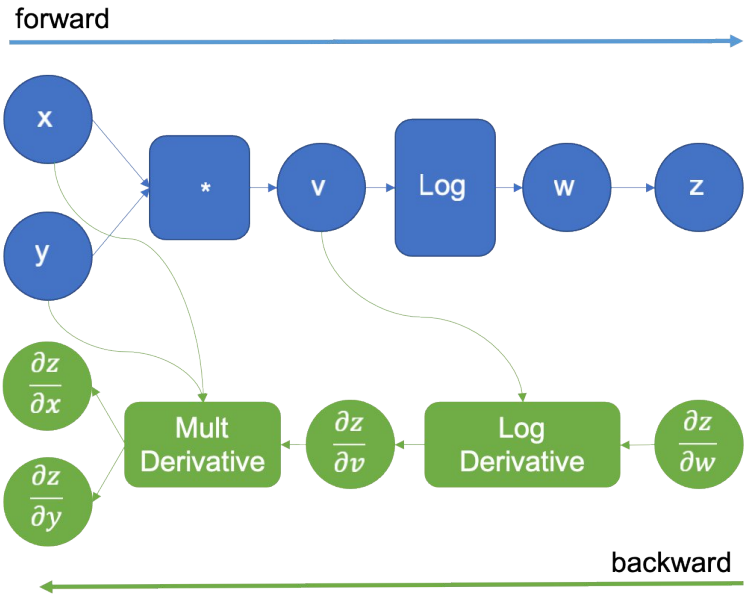
Load in C++

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



TorchScript – Save & Load

L’autre avantage est de pouvoir sauvegarder les graphes ainsi créés de manière pour les charger dans d’autres programmes ou d’autres systèmes (pratique pour l’embarqué)

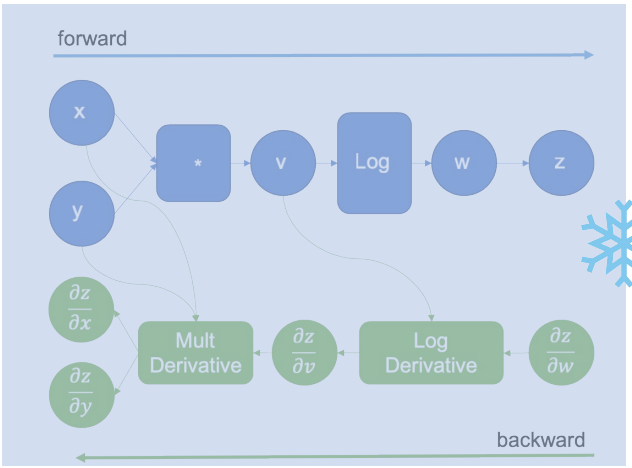


Torch AutoGrad

Dans le torch.autograd classique, chaque opération sur tensor est enregistrée à la volée. Chaque tensor de sortie qui nécessite des gradients est lié à une fonction grad_fn, qui fait référence à l'opération qui l'a produit. Et lors de la backward, pytorch remonte cette chaine de grad_fn pour calculer les gradients. Après quoi le graphe est supprimé.

Avantages : flexible, dynamique

Limites : Pas analysable, pas optimisable en compilant




```python
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()
```

```python
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)
```

```python
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 Ahead of Time AutoGrad

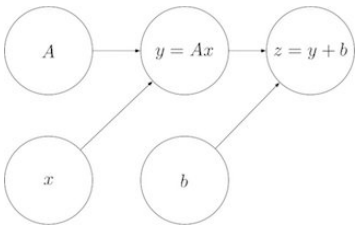
Torch « AOT » (Ahead of Time) Autograd permet de capturer le graphe de calcul forward/backward à l'avance, similairement à ce que fait torchscript.

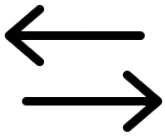
Ce graphe est fait d'objets pythons et est donc analysable et réutilisable facilement.

Il est ensuite passé dans un backend (comme TorchInductor) qui le compile et performe des optimisations là où cela est possible.

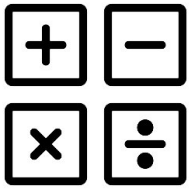
```
torch.compile(model=None, *, fullgraph=False, dynamic=None, backend='inductor', mode=None,
options=None, disable=False) [SOURCE]
```

TorchDynamo + AOT AutoGrad + TorchInductor + PrimTorch





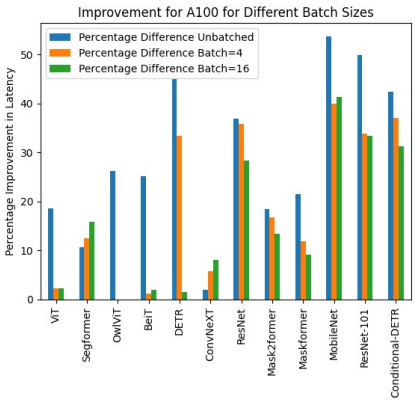


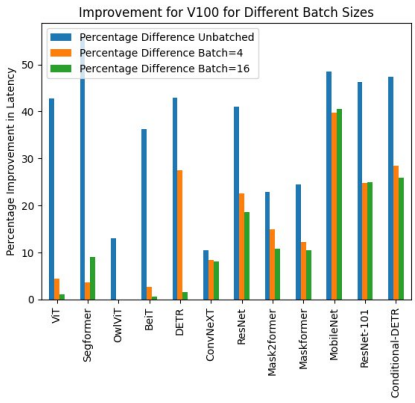


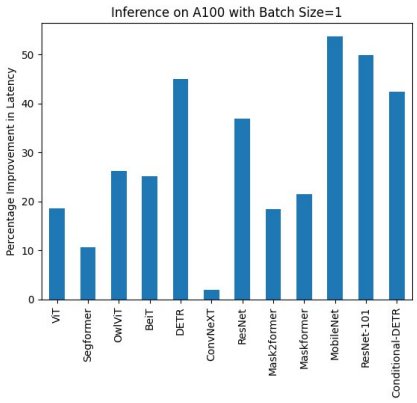
Torch Compile – Mechanisms

On en arrive donc à torch compile qui fonctionne comme suit :

1. TorchDynamo : capture du graphe à partir du bytecode Python.
2. AOTAutograd : construction/optimisation conjointe du forward et du backward (ahead-of-time).
3. Backend : choix de la stratégie de compilation et de génération de code.
4. TorchInductor : backend par défaut, qui génère du code très optimisé CPU/GPU (souvent via Triton pour GPU).







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	**18**
default	29	3277	34	30
reduce-overhead	29	5736	34	28
max-autotune	35	5736	**32**	30



Torch Compile – Performances

Toutefois cette optimisation n'est pas sans coût supplémentaire.

Le temps d'initialisation durant la première step par exemple, et le coût mémoire dû à de la mise en cache



```python
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

```python
import torch

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

Torch Compile Performance

```python
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

Toch Compile permet aussi le wrapping de code numpy

ATTENTION les temps affichés ici ne sont pas exactement a ressources égales : compile arrive a paralléliser le code sur les 10 cœurs CPU à sa disposition, ce qui n'est pas le cas de la fonction native. Ce qui explique le facteur 10.

(Restez critiques face à ce que les gens disent avec leurs benchs)



RAPIDS

Accelerated with NVIDIA



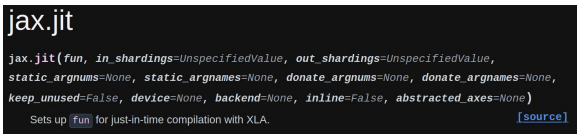


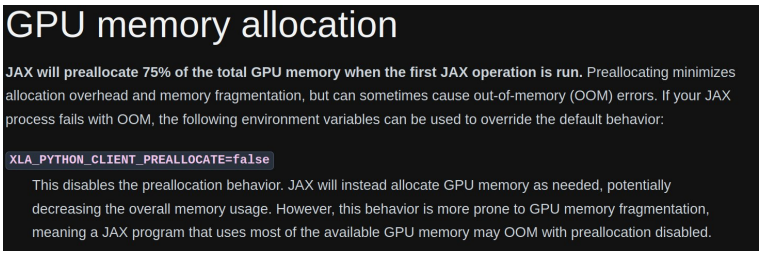


« Numpy » on gpus

Alternatives contenant aussi du JIT : JAX





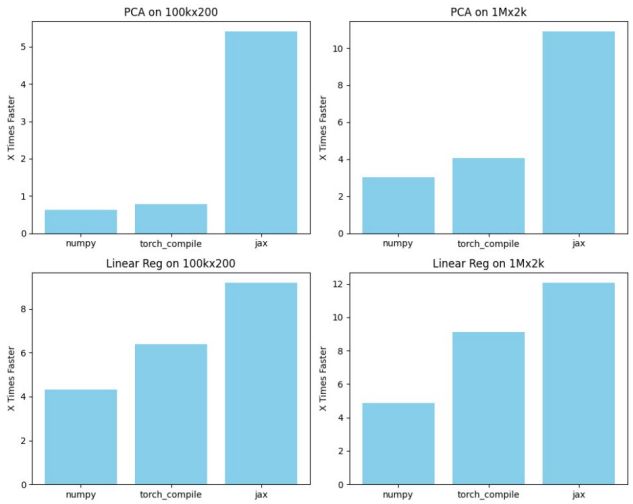


JAX tips

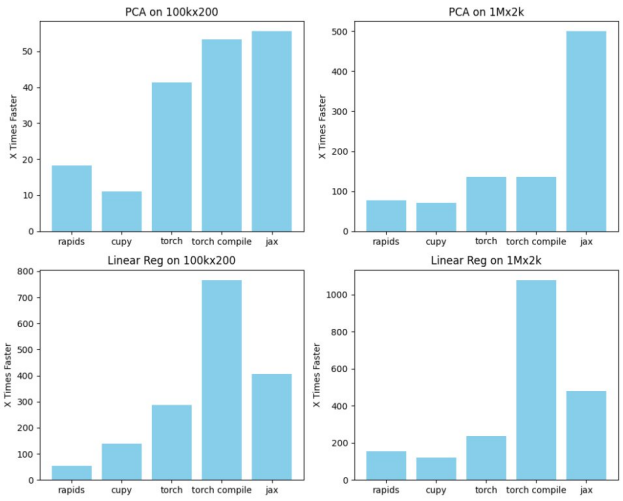
JAX : compliqué à benchmarker car beaucoup d'optimisations automatiques et obscures.

JAX réserve aussi automatiquement 75% de la mémoire à l'initialisation pour éviter les overhead d'allocation à la vollée (mais cela peut causer des OOM errors).

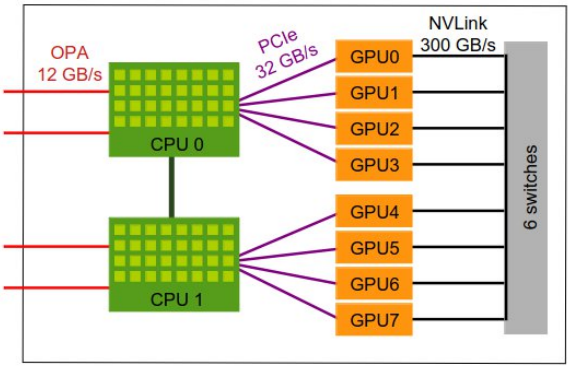
On CPU



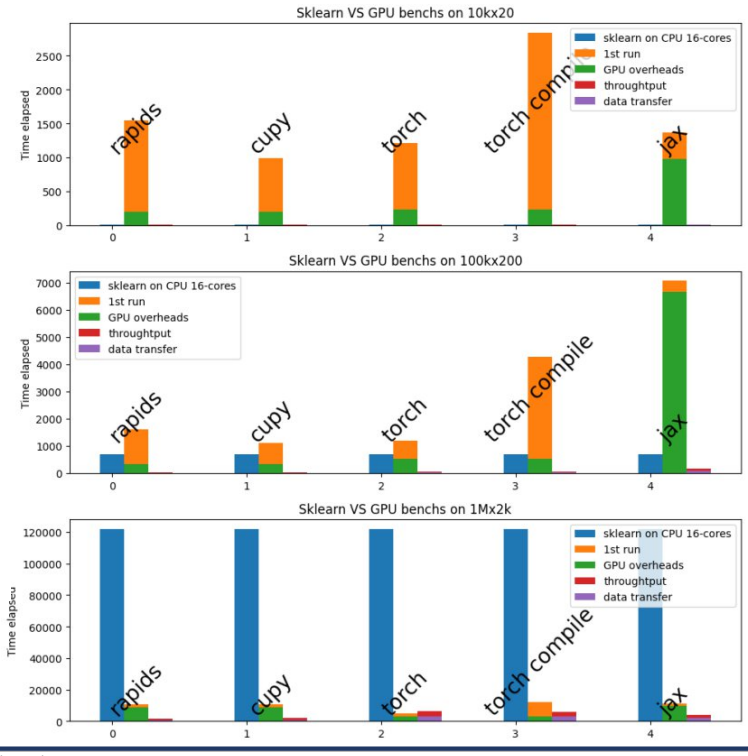
On GPU



Permet aussi d'améliorer des perfs en Machine Learning



Node 8 × A100 80Go



Really worth it ?

Est-ce vraiment utile de passer sur GPU pour des problèmes de Machine Learning ?