Python pour GPU

Jean-Marc Colley, Nabil Garroum, Alice Faure

Introduction

En Python, de nombreuses bibliothèques permettent de programmer sur GPU, à des niveaux d'intégration différents.
Haut niveau : cuNumeric, Intel DPNP, JAX...
Niveau intermédiaire : CuPy, PyTorch...
Bas niveau : PyCUDA, Numba, ...

En rose : celles que nous allons présenter dans ce webinaire !

Profilage et benchmarking

Pour profiler un programme : utiliser Nsight.

Pour chronométrer le temps d'exécution d'un programme :

- Le module timeit recommandé pour les CPU n'est plus adapté car l'exécution sur le GPU est asynchrone !
- Il faut donc utiliser les fonctions dédiées de chaque bibliothèque.

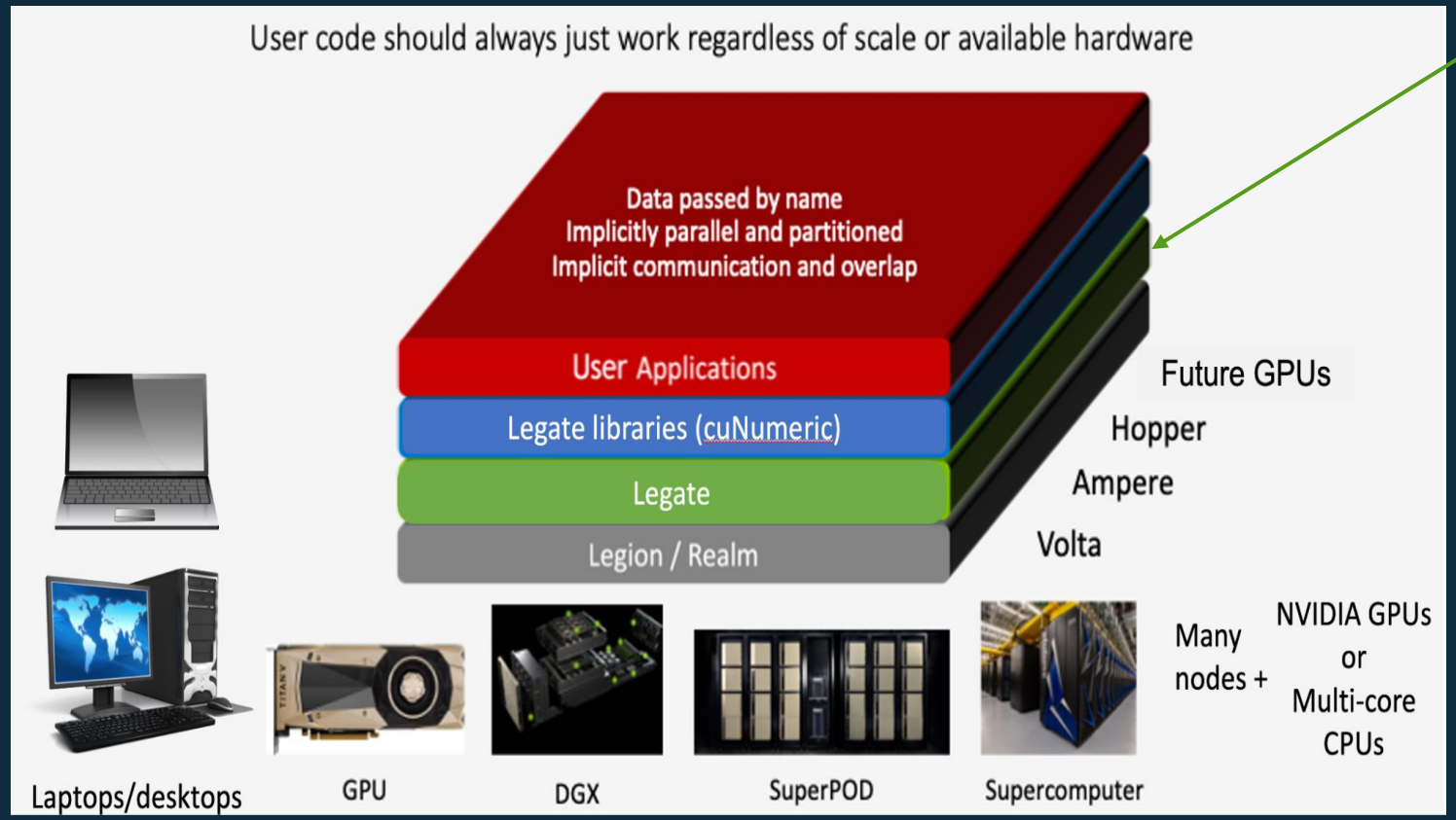
```
import cunumeric as np
from legate.timing import time

init() # Initialization step

# Do few warm-up iterations
for i in range(n_warmup_iters):
    compute()

start = time()
for i in range(niters):
    compute()
end = time()

elapsed_millisecs = (end - start)/1000.0

dump_data() # I/O
```

Tiré de la doc de cuNumeric

```
>>> from cupyx.profiler import benchmark
>>>
>>> def my_func(a):
...     return cp.sqrt(cp.sum(a**2, axis=-1))
...
>>> a = cp.random.random((256, 1024))
>>> print(benchmark(my_func, (a,), n_repeat=20))
my_func             :    CPU:   44.407 us   +/- 2.428 (min:   42.516 / max:
```

Tiré de la doc de CuPy

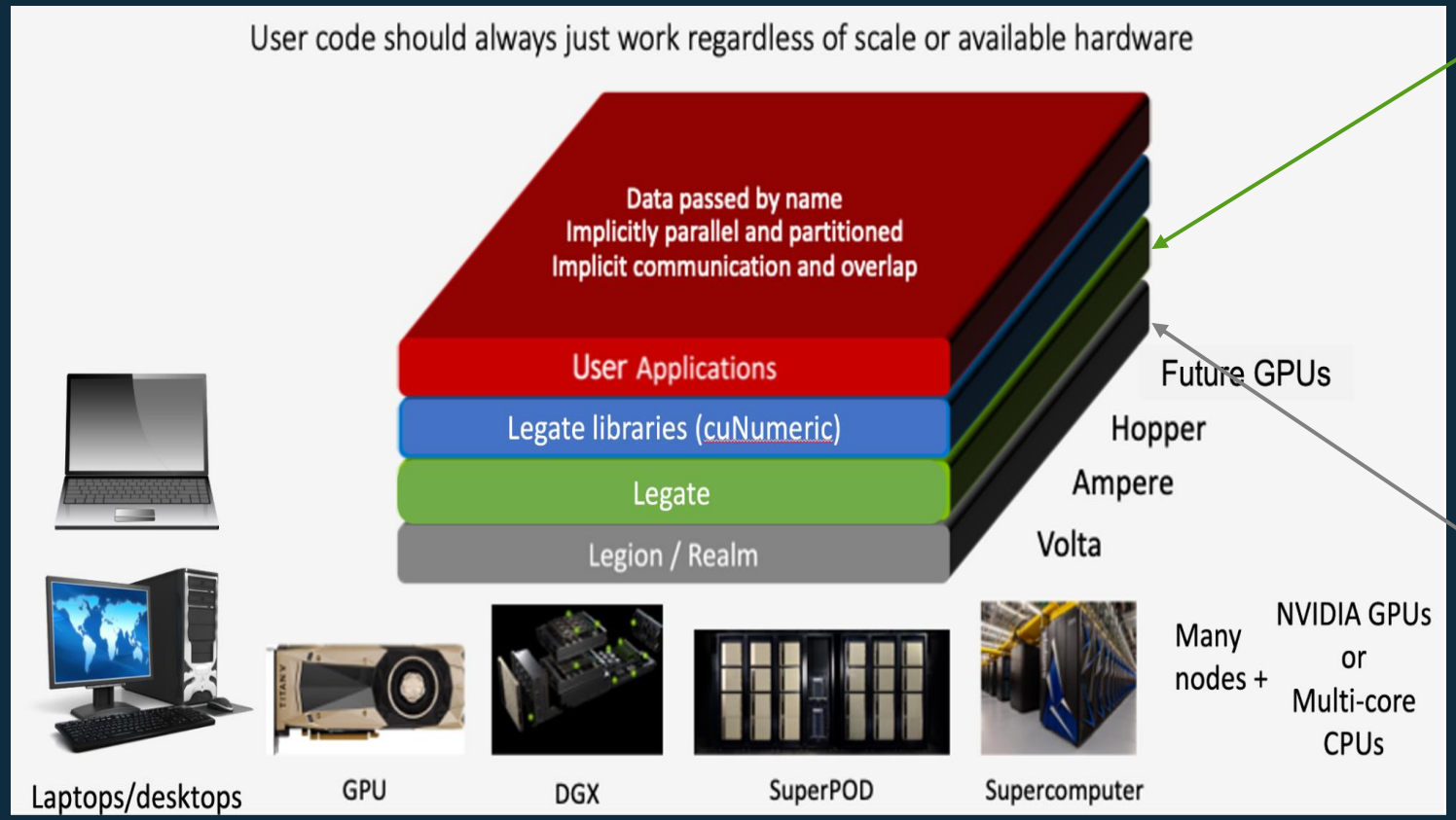
L’écosystème Python de NVIDIA



Legate : projet pour démocratiser le calcul distribué et hétérogène sur CPU ou GPU. La parallélisation est implicite, la personne qui code n’a pas à s’en soucier.

Tiré de Nvidia Developer

L’écosystème Python de NVIDIA

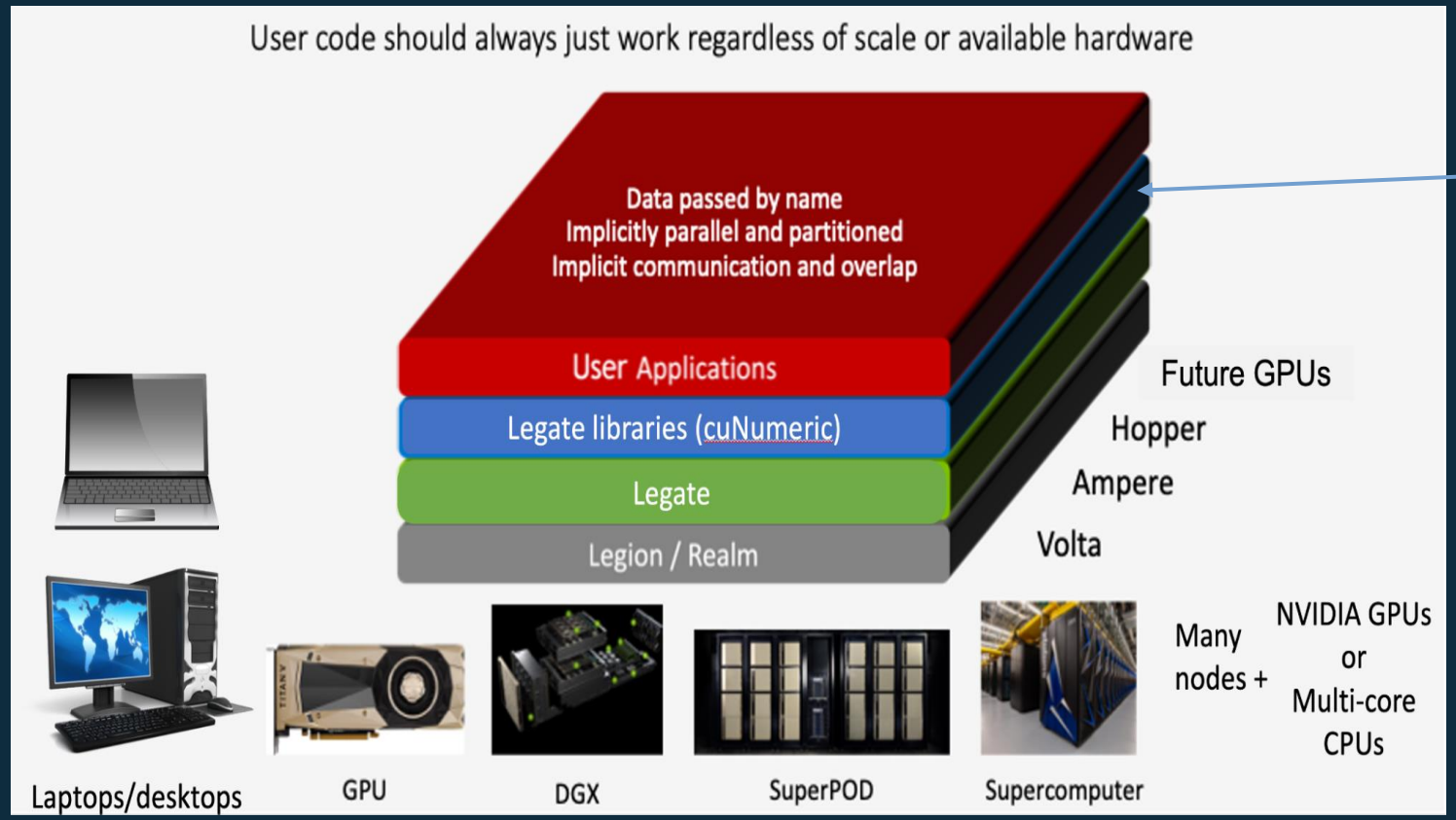


Tiré de Nvidia Developer

Legate : projet pour démocratiser le calcul distribué et hétérogène sur CPU ou GPU. La parallélisation est implicite, la personne qui code n’a pas à s’en soucier.

Basé sur le système de programmation Legion et l’interface d’exécution de bas niveau Realm (basée elle-même sur CUDA).

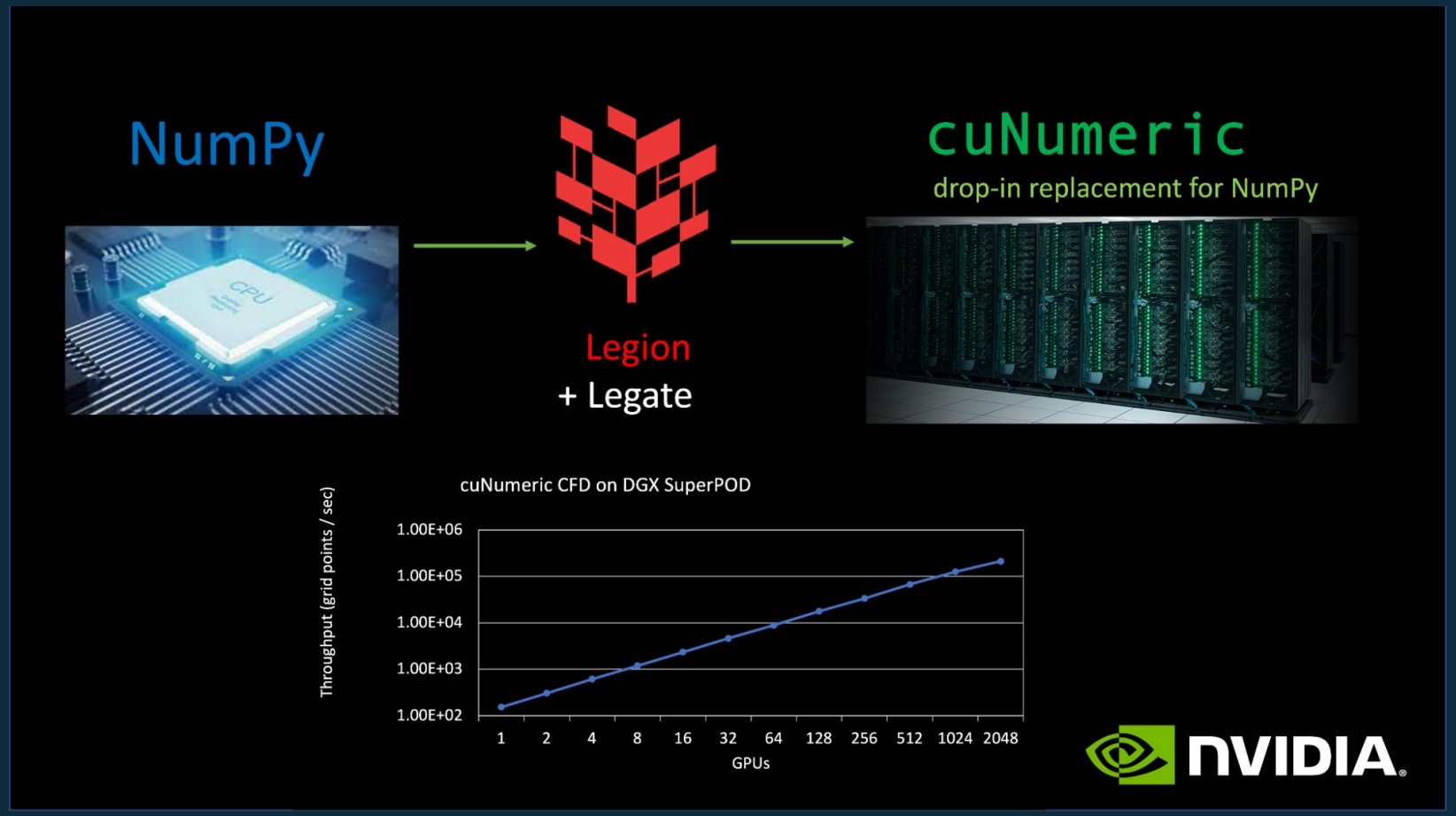
L'écosystème Python de NVIDIA



Des librairies basées sur Legate sont disponibles pour Python : cuNumerics (pour Numpy), legate.pandas (pour Pandas) et permettent de calculer sur CPU ou GPU.

Tiré de Nvidia Developer

cuNumeric



Tiré de [Nvidia Developer]

cuNumeric

cuNumeric est un clone de Numpy. Pour l'instant, 68% des fonctions Numpy ont un équivalent cuNumeric.

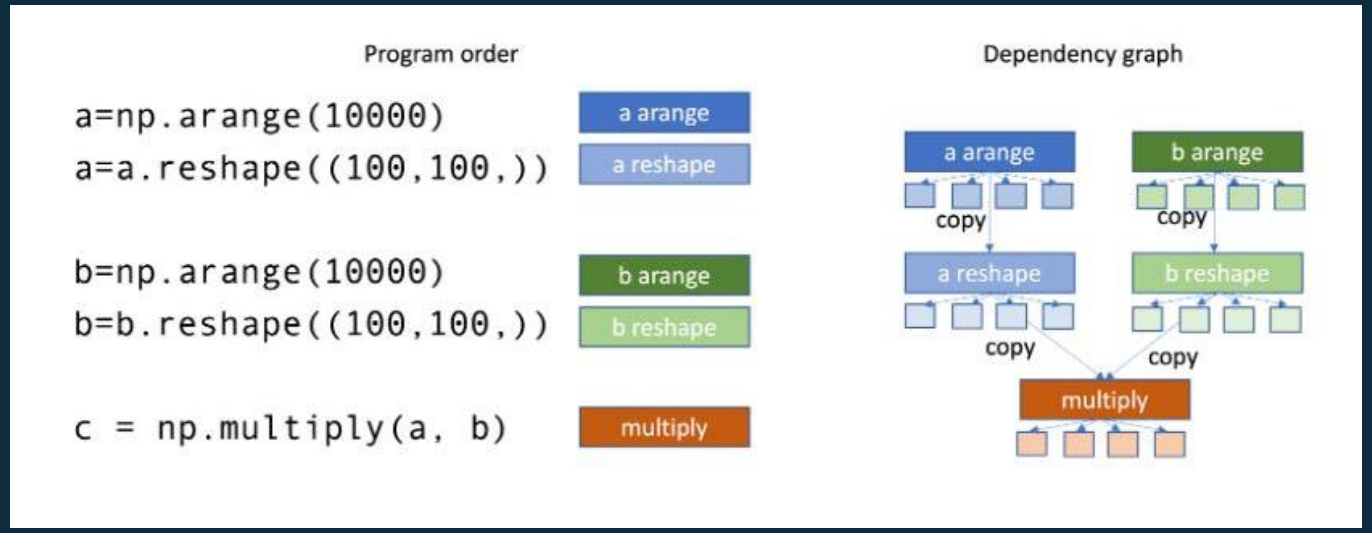
- Installation : `>>> conda install -c nvidia -c conda-forge -c legate cunumeric`
- Import : `import cunumeric as np`
- Exécution : avec le driver legate (recommar `legate --cpus 2 --gpus 2 program.py`

Comme avec les bibliothèques d'optimisation pour CPU, l'utilisation de cuNumeric sera d'autant plus efficace que le code utilise des tableaux et fonctions Numpy.

cuNumeric

Parallélisation implicite :

- sur les données : cuNumeric partitionne les données, et Legion les répartit aux processeurs
- sur les tâches : cuNumeric crée des tâches à exécuter, Legion produit un graphe de dépendances et gère l'exécution asynchrone des différentes tâches



Tiré de Nvidia Developer

cuNumeric : un exemple qui marche

Le fichier stencil.py :

```
import cunumeric as np
```

```
def initialize(N):
    print("Initializing stencil grid...")
    grid = np.zeros((N + 2, N + 2))
    grid[:, 0] = -273.15
    grid[:, -1] = -273.15
    grid[-1, :] = -273.15
    grid[0, :] = 40.0
    return grid
```

```
def run_stencil(N, I, warmup, timing):
    grid = initialize(N)
    print("Running Jacobi stencil...")
    center = grid[1:-1, 1:-1]
    north = grid[0:-2, 1:-1]
    east = grid[1:-1, 2:]
    west = grid[1:-1, 0:-2]
    south = grid[2:, 1:-1]

    timer.start()
    for i in range(I + warmup):
        if i == warmup:
            timer.start()
        average = center + north + east + west +
south
        work = 0.2 * average
        center[:] = work
    total = timer.stop()

    if timing:
        print(f"Elapsed Time: {total} ms")
```

```
if __name__ == "__main__":
    run_stencil(N=100, I=100, warmup=5,
timing=True)
```

cuNumeric : un exemple qui marche

Le fichier stencil.py :

```
import cunumeric as np
```

```
def initialize(N):
    print("Initializing stencil grid...")
    grid = np.zeros((N + 2, N + 2))
    grid[:, 0] = -273.15
    grid[:, -1] = -273.15
    grid[-1, :] = -273.15
    grid[0, :] = 40.0
    return grid
```

```
def run_stencil(N, I, warmup, timing):
    grid = initialize(N)
    print("Running Jacobi stencil...")
    center = grid[1:-1, 1:-1]
    north = grid[0:-2, 1:-1]
    east = grid[1:-1, 2:]
    west = grid[1:-1, 0:-2]
    south = grid[2:, 1:-1]

    timer.start()
    for i in range(I + warmup):
        if i == warmup:
            timer.start()
        average = center + north + east + west +
south
        work = 0.2 * average
        center[:] = work
    total = timer.stop()

    if timing:
        print(f"Elapsed Time: {total} ms")
```

```
if __name__ == "__main__":
    run_stencil(N=100, I=100, warmup=5,
timing=True)
```

L'exécution :

```
>>> legate --cpus 1 --gpus 1 stencil.py
```

```
Initializing stencil grid...
Running Jacobi stencil...
Elapsed Time: 85.053 ms
```

```
>>> legate --cpus 1 stencil.py
```

```
Initializing stencil grid...
Running Jacobi stencil...
Elapsed Time: 525.713 ms
```

cuNumeric : un exemple qui ne marche pas

Le fichier array.py :

```
import cunumeric as np

nimages = 2
nrow = 1080
ncol = 1920

image_array = np.zeros((nimages, nrow,
ncol))
image_array[0,:,:]=1
```

L'exécution :

```
>>> legate --cpus 1 --gpus 1 array.py
```

cuNumeric : un exemple qui ne marche pas

```
Signal 4 received by node 0, process 24501 (thread 2b8c331a5700) - obtaining backtrace
Signal 4 received by process 24501 (thread 2b8c331a5700) at: stack trace: 14 frames
  [0] = /lib64/libc.so.6(+0x36400) [0x2b8c2fbef400]
  [1] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/lib/libcunumeric.so(void Realm::AffineAccessor<double, 2, long long>::reset<3, long long>(Realm::RegionInstance, Realm::Matrix<3, 2, long long> const&, Realm::Point<3, long long> const&, int, Realm::Rect<2, long long> const&, unsigned long)+0x8c) [0x2b8d882dd54c]
  [2] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/lib/libcunumeric.so(Legion::FieldAccessor<(legion_privilege_mode_t)268435463, double, 2, long long, Realm::AffineAccessor<double, 2, long long>, false> legate::Store::write_accessor<double, 2>(Realm::Rect<2, long long> const&) const+0x5f4) [0x2b8d882de264]
  [3] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/lib/libcunumeric.so(+0x1afea62) [0x2b8d89afea62]
  [4] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/lib/libcunumeric.so(+0x193fca8) [0x2b8d8993fca8]
  [5] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/lib/libcunumeric.so(void legate::LegateTask<cunumeric::UnaryOpTask>::legate_task_wrapper<&cunumeric::UnaryOpTask::gpu_variant>(void const*, unsigned long, void const*, unsigned long, Realm::Processor)+0xab) [0x2b8d885c203b]
  [6] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/bin/../lib/librealm.so.1(+0x4c9de1) [0x2b8c2eebade1]
  [7] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/bin/../lib/librealm.so.1(+0x4c9e56) [0x2b8c2eebae56]
  [8] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/bin/../lib/librealm.so.1(+0x4fe388) [0x2b8c2eeef388]
  [9] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/bin/../lib/librealm.so.1(+0x4cca5a) [0x2b8c2eebda5a]
  [10] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/bin/../lib/librealm.so.1(+0x4ceed2) [0x2b8c2eebfed2]
  [11] = /mustfs/CONTAINERS/conda/formation/afaure/envs/myenv/bin/../lib/librealm.so.1(+0x4d217f) [0x2b8c2eec317f]
  [12] = /lib64/libpthread.so.0(+0x7ea5) [0x2b8c31b76ea5]
  [13] = /lib64/libc.so.6(clone+0x6d) [0x2b8c2fcb7b0d]
```

cuNumeric

Limites :

- l'analyse dynamique du code pour répartir les données et déterminer les dépendances génère un overhead. Il est supposé être compensé par le pipelining de l'analyse avec les calculs effectués par l'application → Pour cela il faut que l'application fasse assez de calculs !
- La version beta est sortie il y a un an. Depuis, cuNumeric est encore en développement. Le projet est prometteur mais malheureusement il y a encore beaucoup de bugs (array slicing notamment) et les messages d'erreur n'aident pas vraiment...

En tout cas, à tester sur votre code si vous pouvez l'installer ! Si vous rencontrez des bugs, il vaut peut-être mieux utiliser une autre librairie pour l'instant.

Présentation de JAX

- JAX est une bibliothèque python pour le calcul orienté tableau du type Numpy
- JAX contient plusieurs sous-bibliothèques et a un « écosystème » orienté pour le Deep Learning
- JAX est conçu pour le calcul haute performance et supporte plusieurs types d'accélérateur matériel (device) sans modification de code
- JAX introduit un pseudo-langage « jax expression » pour décrire les fonctions numériques facilitant le travail des « transformations »
- JAX est aussi conçu dans le but de faire du calcul à grande échelle en gérant plusieurs devices

JAX et les tableaux

- JAX cohabite avec Numpy mais il a sa propre représentation de tableau les « jax array » avec quasiment toutes les fonctionnalités de Numpy
- Cependant les jax array ont quelques particularités
  - ils sont **immutables**
  - Ce sont des instances de la classe `jax.core.Tracer`, qui joue un rôle important dans l'optimisation des calculs
  - Ils sont associés à un device
  - là où Numpy utilise des vues, JAX emploie des copies
  - la « politique » de type de défaut est différente

- Conséquence le portage d'un programme Numpy vers JAX n'est pas immédiat

Les bibliothèques internes de JAX

- Il y en a une vingtaine de sous-bibliothèques dont bien sûr `jax.numpy`
- `jax.scipy`
  - Contient 10 des 14 sous bibliothèques de scipy
- Toutes les fonctions ne sont pas présentes, mais elles peuvent être présentes dans des annexes de JAX, comme pour l'optimmisation avec JAXOpt.
- Biblothèques liées aux calculs : `jax.random`, `jax.image`, `jax.lax` (numpy bas niveau/interface avec XLA), `jax.nn` (neural network)
- Bibliothèques liées au développement : `jax.debbug`, `jax.profiler`
- Bibliothèques liées au parallélisme et gestion matériel : `jax.sharding`, `jax.distributed`
- Bibliothèques liées à la représentation des données : `jax.tree`, `jax.typing`, `jax.dlpack`

JAX et le calcul haute performance

De nombreux device pris en charge

Ainsi que les 3 OS

	Linux, x86_64	Linux, aarch64	macOS, Intel x86_64, AMD GPU	macOS, Apple Silicon, ARM-based	Windows, x86_64	Windows WSL2, x86_64
CPU	yes	yes	yes	yes	yes	yes
NVIDIA GPU	yes	yes	no	n/a	no	experimental
Google Cloud TPU	yes	n/a	n/a	n/a	n/a	n/a
AMD GPU	experimental	no	no	n/a	no	no
Apple GPU	n/a	no	experimental	experimental	n/a	n/a

JAX notion de « transformation »

- Une « transformation » est une fonction qui transforme une fonction en une autre fonction
- L'exemple typique est jax.grad(f) qui retourne l'application dérivée de f
- Les fonctions suivantes sont les principales "transformations" à connaître
  - Jax.grad()
  - jax.jit(app_f) : compilation à la volée pour le device
  - jax.vmap(app_f) : vectorise automatiquement une fonction
- Les transformations peuvent se combiner

JAX expression

- Pour facilter le travail des « transformations» les fonctions ont une réprésentation en pseudo-code que l'on peut visualiser avec make_jaxpr()

```
import jax

In [19]: def carre(x)
    ...:     return x*x
    ...:

In [20]: a=5.0
```

```
In [21]: print(jax.make_jaxpr(jax.grad(carre))(a))
{ lambda ; a:f32[]. let
    _:f32[] = mul a a
    b:f32[] = mul a 1.0
    c:f32[] = mul 1.0 a
    d:f32[] = add_any c b
  in (d,) }

In [22]: jax.grad(carre)(a)

Out[22]: Array(10., dtype=float32, weak_type=True)
```

JAX et le calcul haute performance

- JAX utilise le backend OpenXLA pour la compilaton juste à temps
- Open XLA s'adapte au device, voir site OpenXLA
- Contrainte pour utiliser le jax.jit() :
  - La taille et les dimensions des données d'entrées et de sortie sont connues au moment de la compilation

JAX et le calcul à grand échelle

- JAX permet de paralléliser une fonction et ses données sur plusieurs unités de calcul, modèle de calcul : Single-Program Multi-Data (SPMD) avec la fonction **jax.pmap()**
- jax.pmap(func) applique par defaut jax.jit sur func

JAX, code exemple, erreur

```
from jax import jit
import jax.numpy as jnp

@jit
def func(x, axis):
   return x.min(axis)

func(jnp.arange(4), 0)
Traceback (most recent call last):
    ...
ConcretizationTypeError: Abstract tracer
value encountered where concrete
value is expected: axis argument to
jnp.min().
```

Correction:

```
from functools import partial

@partial(jit, static_argnums=1)
def func(x, axis):
   return x.min(axis)

func(jnp.arange(4), 0)
Array(0, dtype=int32)
```

JAX conclusion

- Bibliothèque très complète, performante mais la prise en main demande du travail
- Bibliothèque conçue pour des algorithmes nécessitant beaucoup de calcul, passage à l'échelle entre le développement et la production est a priori facile
- Le changement de device est aussi facile
- c'est une bibliothèque de haut niveau
- JAX introduit des notions (sans équivalent dans les autres bibliothèques ?) indispensables à maîtriser.
- Les noms des articles de la documentation « How to think in JAX » et « JAX Errors » illustrent l'effort nécessaire pour utiliser cette bibliothèque, à lire absolument !
- Sans ses connaissances, les messages d'erreur sont énigmatiques

Cupy présentation

Objectif du projet

L'objectif du projet CuPy est de fournir aux utilisateurs de Python des capacités d'accélération GPU, sans une connaissance approfondie des technologies GPU sous-jacentes.

L'équipe CuPy se concentre sur la fourniture d’une couverture complète des API NumPy et SciPy pour devenir un remplacement complet, ainsi que des fonctionnalités CUDA avancées pour maximiser les performances.

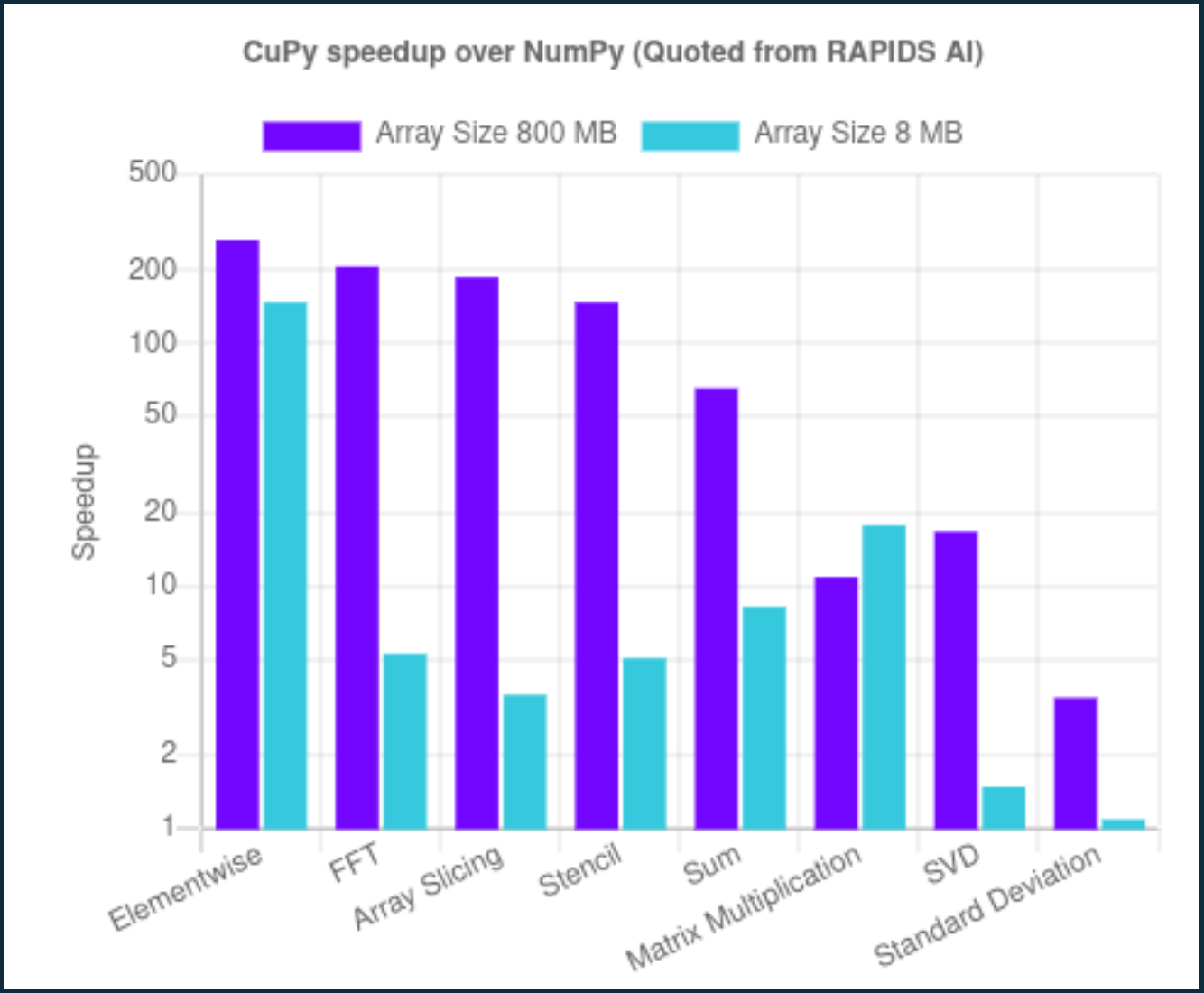
Support matériel

- GPU Nvidia et AMD Roc(experimental)

Backend :

- Bibliothèques Nvidia : cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL

Cupy performance



Cupy Numpy/Scipy

Numpy pour GPU : module cupy

- Les tableaux cupy sont dans la mémoire du « device » , ie GPU
- Copie tableau CPU vers GPU : **a_gpu = cupy.array(my_array_numpy)**
- Copie tableau GPU vers CPU : **b_cpu= a_gpu.get()**

Très bonne couverture de l'API Numpy version GPU, table de comparaison disponible :

- https://docs.cupy.dev/en/stable/reference/comparison.html#numpy-cupy-apis

Scipy pour GPU : module cupyx.scipy

Table de comparaison disponible aussi, moins bien couvert que Numpy

- https://docs.cupy.dev/en/stable/reference/comparison.html#scipy-cupy-apis

Cupy plusieurs types de Kernel

- **Elementwise** : instance de la classe `ElementwiseKernel`, défini par des chaines de caractètres : type entrée, Type sortie, formule/code, Nom du kernel
- **Reduction** : instance de la classe `ReductionKernel`, défini par des chaines de caractètres aussi. Réduction dans le sens array vers scalaire, par exemple: une norme, une moyenne.
- **Raw** : instance de la classe `RawKernel`, identique à un kernel CUDA dans une chaine de caractère
- **Fusion** : avec le décorateur `@cupy.fuse`, défini avec al syntaxe python comme une fonction « normale ». Fusion car il supporte ElementWise et Reduction

Cupy gestion de la mémoire

- Possibilité de spécifier le type de mémoire pour gagner en performance de transfert :
  - Device memory pool (GPU device memory), which is used for GPU memory allocations.
    - `cupy.get_default_memory_pool()`
  - Pinned memory pool (non-swappable CPU memory), which is used during CPU-to-GPU data transfer.
    - `cupy.get_default_pinned_memory_pool()`
- On peut sélectionner son « device » si plusieurs GPU

```
with cp.cuda.Device(1):
   x = cp.array([1, 2, 3,
4, 5])
x.device
```

```
<CUDA Device 1>
```

Cupy exemple de code : Gray-Scott

Le kernel fusion écrit en python

```
import cupy as cp

@cp.fuse()
def grayscott_kernel(Lu, Lv, u, v, Du, Dv, F, k, delta_t):
    uvv = u * v * v
    u += delta_t * ((Du * Lu - uvv) + F * (1 - u))
    v += delta_t * ((Dv * Lv + uvv) - v * (F + k))
```

Cupy exemple de code : Gray-Scott

Initialisation du stencil et mise en mémoire GPU

```
# Laplacian stencil
stencil = np.array([[0, 1, 0],
                    [1, -4, 1],
                    [0, 1, 0]],
                    dtype=np.float32)

# Load in GPU memory
stl_gpu = cp.array(stencil)
u_gpu = cp.asarray(U)
v_gpu = cp.asarray(V)
```

Cupy exemple de code : Gray-Scott

Calcul du laplacien par convolution, appel au kernel GPU , récupération d'un frame sur "step_frame" vers le CPU.

```
from cupyx.scipy.signal import convolve2d as conv2d_gpu

frames_V = np.empty((nb_frame, n_x, n_y), dtype=np.float32)
for idx_fr  in range(nb_frame):
        for _ in range(step_frame):
                # compute laplacians with convolution provided by c
                Lu = conv2d_gpu(u_gpu, stl_gpu, 'same', 'fill', 0)
                Lv = conv2d_gpu(v_gpu, stl_gpu, 'same', 'fill', 0)
                grayscott_kernel(Lu, Lv, u_gpu, v_gpu, Du, Dv, F, k, delta_t)
        frames_V[idx_fr ,:,:] = v_gpu.get()
```

X 80 par rapport à la version CPU

Cupy conclusions

Retour d'expérience très positif :

- pas de problème d'installation
- Le mécanisme de base du placement mémoire CPU/GPU est très facile à utiliser
- Pas de surprise dans l'utilisation des versions GPU de numpy/scipy
- Le kernel GPU peut s' écrire en python
- L'adaptation Gray-Scott a été rapide et le gain de vitesse pour une adaptation naïve est déjà important
  - Sur cette exemple l'objectif de la bibliothèque est atteint.
- Bonne documentation

Python et le GPU/Nvidia bas niveau

Connaissance de 3 bibliothèques proposant une programmation bas niveau des GPUs en python, pour avoir par exemple une gestion fine de la mémoire ou de la répartition des calculs , mais pas de retour pour l'instant :

- Numba : voir doc **Numba for CUDA GPUs**
- pyCuda : voir doc pycuda
- Cupy : voir doc low-level-cuda-support

Pytorch : intermediaire

Pytorch est une librairie crée par Meta (Facebook) pour le deep learning.

Pytorch combine plusieurs fonctionnalités

- Opérations arithmétiques pour les tenseurs sur le modèle numpy
- Opérations de deep-learning (réseaux de neurones et graphes)
- Opérations de dérivation (autograd) de tenseurs ou de fonctions analytiques.
  - Installation : `import` torch

Pytorch : BackEnds

Les operation arithmétiques et matricielles sous Pytorch sont faites avec la librarie cuBLAS, librairie propriétaire, le code source n'est pas disponible

https://pytorch.org/docs/stable/backends.html

Efforts en cours pour utiliser CUTLASS pour le calcul matriciel (GEMM : GEneral Matrix Multiplication) pour le calcul matriciel :

https://research.colfax-intl.com/tutorial-python-binding-for-cuda-libraries-in-pytorch/

Intégration d'OpenXLA en cours pour l'utilisation TPUs.

https://pytorch.org/blog/pytorch-2.0-xla-path-forward/

https://pytorch.org/xla/release/2.3/index.html

- torch.backends.cpu
- torch.backends.cuda
- torch.backends.cudnn
- torch.backends.mha
- torch.backends.mps
- torch.backends.mkl
- torch.backends.mkldnn
- torch.backends.nnpack
- torch.backends.openmp
- torch.backends.opt_einsum
- torch.backends.xeon

https://pytorch.org/docs/stable/cuda.html

device(type='cpu')

Pytorch : tensors

Pytorch : Exécution sur CPU

```
import torch
a = torch.FloatTensor([1.,2.])
```

Sans spécification, par défaut le CPU est utilisé

```
a.device
device(type='cpu')
```

Détection simple et numérotation des GPUs :

```
import torch
torch.cuda.is_available() # True
torch.cuda.get_device_name(0) # Tesla T4
```

Pytorch : tensors

Le choix de l'allocation est fait à la déclaration de la variable

```
a = torch.FloatTensor([1., 2.]).cuda(0)

a.device
# device(type='cuda', index=0)

torch.cuda.memory_allocated()  # 512
```

Contrôle du transfert des variables à la volée :

Gestion Multi-GPUs:

De CPU à CPU, et de GPU à GPU

```
Tensor.to("cuda:0")
Tensor.to("cuda:1")
Tensor.to("cpu")
```

Pytorch : tensors

Fonctions arithmétiques et matricielles habituelles

Arithmetic		
OPERATION	FUNCTION	DESCRIPTION
a + b	a.add(b)	element wise addition
a - b	a.sub(b)	subtraction
a * b	a.mul(b)	multiplication
a / b	a.div(b)	division
a % b	a.fmod(b)	modulo (remainder after division)
a^b	a.pow(b)	power

Monomial Operations		
OPERATION	FUNCTION	DESCRIPTION
|a|	torch.abs(a)	absolute value
1/a	torch.reciprocal(a)	reciprocal
$\sqrt{a}$	torch.sqrt(a)	square root
log(a)	torch.log(a)	natural log
e^a	torch.exp(a)	exponential
12.34 ==> 12.	torch.trunc(a)	truncated integer
12.34 ==> 0.34	torch.frac(a)	fractional component

Opérations sur les tenseurs sur le modèle numpy

Plusieurs fonctions comme view() , reshape()

Pytorch : tensors

Exemple d’opérations sur les tenseurs GPU

Produit matriciel ou elementwise

```
a = torch.tensor([1,2,3], dtype=torch.float).cuda(0)
b = torch.tensor([4,5,6], dtype=torch.float).cuda(0)
print(a.mul(b)) # for reference
print()
print(a.dot(b))
```

```
tensor([ 4., 10., 18.])

tensor(32.)
```

Brodcasting sur le modèle numpy

```
[ ]  t1 = torch.randn(2, 3, 4)
     t2 = torch.randn(4, 5)

     print(torch.matmul(t1, t2).size())
```

```
torch.Size([2, 3, 5])
```

Pytorch : Vectorisation des fonctions

Pytorch permet d'automatiser la vectorisation de fonction ET leur dérivation

$$\begin{aligned} Function: \quad & y = 2x^4 + x^3 + 3x^2 + 5x + 1 \\ Derivative: \quad & y' = 8x^3 + 3x^2 + 6x + 5 \end{aligned}$$

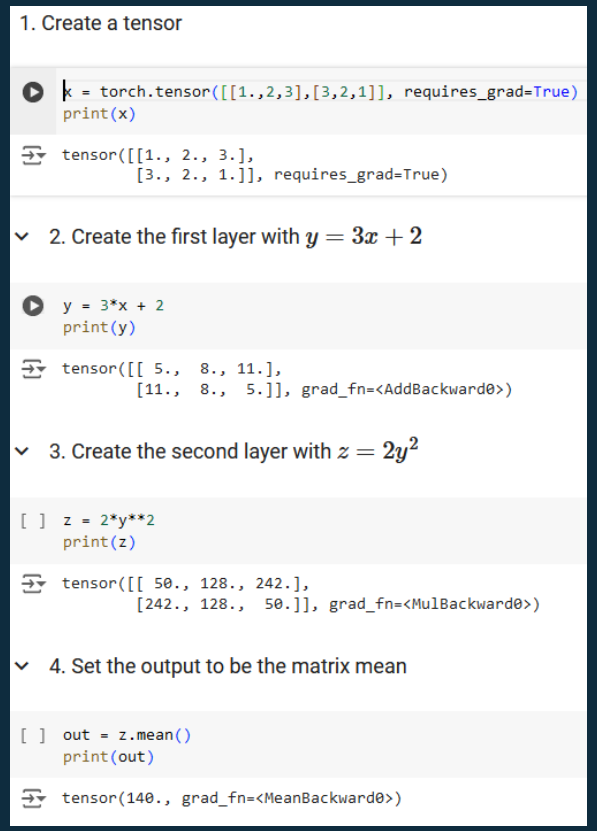
```
x = torch.tensor(2.0, requires_grad=True)
y = 2*x**4 + x**3 + 3*x**2 + 5*x + 1
Type (y) # torch.Tensor
y.backward()
print(x.grad) # tensor(93.)
```

Pytorch : Dérivation des fonctions

Vectorisation

$$y = 3x + 2$$
$$z = 2y^2$$

```
x = torch.tensor([[1.,2,3],[3,2,1]], requires_grad=True)
y = 3*x + 2
z = 2*y**2
z.backward()
print(x.grad) tensor([[10., 16., 22.],
      [22., 16., 10.]])
```



Calcul automatique des derivées, et des dérivées composées sous format tensoriel.

Rq : Différenciation au sens des différences finies est possible :

magnum.np: a PyTorch based GPU enhanced finite difference : https://www.nature.com/articles/s41598-023-39192-5

Pytorch : Profilage

Profileur intégré dans la librairie, dédié au deep learning

with torch.profiler.profile()

…

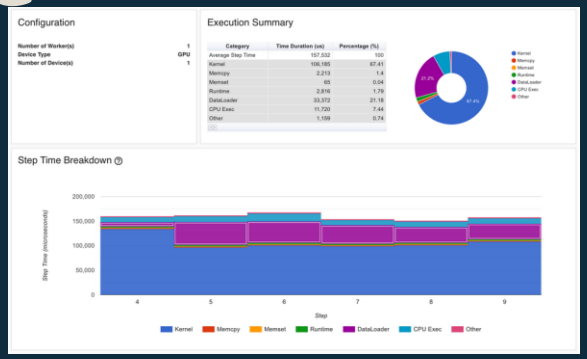
optimizer.zero_grad()

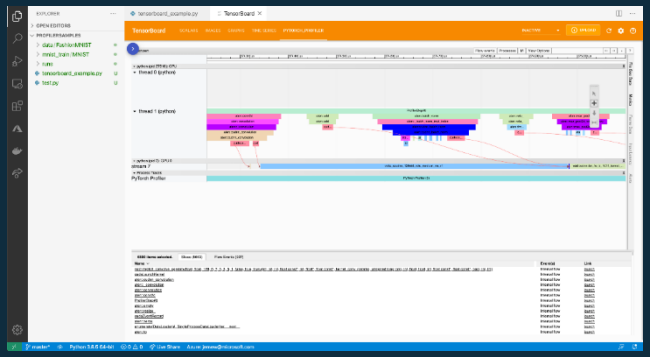
loss.backward()

optimizer.step()

profiler.step()

Intégration dans VS Code

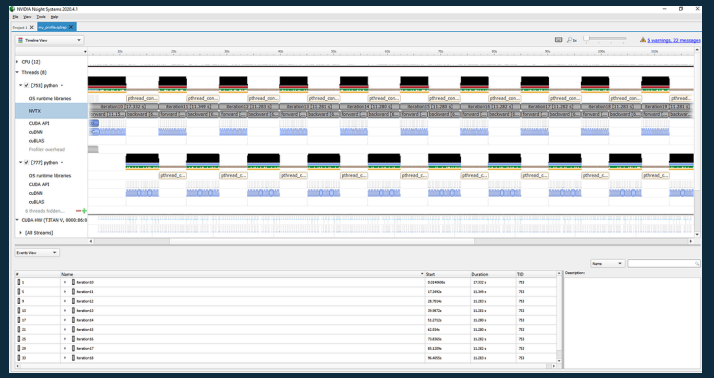




Possibilité d'utilisation de nsight pour le profilage sur GPUs nvidia

import torch

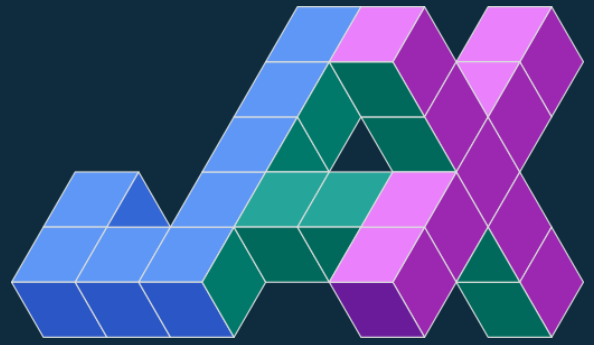
import torch.cuda.profiler as profiler



Résumé

Librairie	cuNumeric	JAX	CuPy	Pytorch
Niveau	Haut	Haut	Intermédiaire	Intermédiaire
Calcul Matriciel Combinatoire	X	X	X	X
Deep learning		X		X
Dérivation		X		X
Multi-accélérateurs	X	X	_	_
Profilage	X	X	X	X

Conclusion



CuPy