Pascal ORTIZ

Numba

Version du 18 septembre 2021

Table des matières

Chapitre I

Présentation, installation

Qu'est-ce que Numba ?

Numba est un module Python utilisé pour du calcul intensif et qui permet, sous réserve de travailler avec des tableaux Numpy :

— d'améliorer considérablement les performances de certains programmes en gardant une syntaxe presque identique au programme n'utilisant pas Numba

— de donner accès, avec une syntaxe légère, à des boucles parallèles sur un processeur multicœur,

— de donner accès, sous Python, à l'environnement Cuda de programmation sur GPU, avec bien sûr, d'importantes améliorations de performances par rapport au calcul sur un simple CPU, même multicœur.

Il y a toutefois de nombreuses contreparties à ces excellentes performances.

Guide de lecture

Ce document sur le module Numba est divisé est 5 parties mais si vous voulez aller à l'essentiel, voici ce que je vous recommande :

— Commencez par lire : Utilisation de Numba sur Google Colab

— Ensuite, il faut avoir parcouru le paragraphe : Les deux modes de fonctionnement de Numba

— Vous pouvez alors directement lire la partie Programmation Numba sur CPU. L'essentiel se trouve dans les paragraphes suivants :

  — L'efficacité de Numba sur le tri par sélection

  — Vectorisation avec Numba

  — Parallélisme sur CPU

— Lire le paragraphe Utiliser Numba avec un GPU sous Google Colab et, éventuellement, les deux paragraphes de généralités sur le GPGPU avec Numba et les cartes graphiques.

— Lire la dernière partie Programmation Numba sur GPU.

— La partie sur les structures de données Numba a un intérêt limité dans la mesure où elles sont expérimentales et d'une efficacité variable.

— Les autres parties sont consacrées à l'installation et à des informations diverses sur Numba.

Ce document n'a pas un caractère approfondi et propose essentiellement un panorama des fonctionnalités de Numba.

Je n'explique pas comment utiliser des feuilles Jupyter Notebook. Sous Linux ou macOS, je suppose que vous avez l'habitude de travailler en ligne de commande. Je suppose aussi que vous avez utilisé pip (appelé pip3 sur mon système Linux).

Pouvoir utiliser Numba

Accès à Numba

Pour pouvoir utiliser *pleinement* Numba (sous-entendu : avec GPU si disponible), vous avez les possibilités suivantes :

— utiliser Google Colab. Avantages :
  — fontionne d'emblée, rien à installer
  — on peut utiliser des GPU presque immédiatement
— utiliser Numba via le toolkit Anaconda 3 Personal Edition (Linux, macOS, fortement recommandé pour Windows)
— installer Numba via pip (linux, macOS)
— utiliser Numba en ligne avec mybinder, cf. la section dédiée ci-dessous.

Version

Pour connaître la version de Numba que vous utilisez, il suffit d'écrire le code suivant :

```
import numba

print(numba.__version__)
```

qui affiche chez moi (mai 2021) :

```
0.53.1
```

Le problème des GPU

L'installation **par défaut** d'Anaconda ou l'installation via pip ne suffisent pas à exploiter votre (éventuel) GPU sous Numba. Si vous tentez de l'utiliser, vous aurez différents types de message d'erreur et qui dépendent de l'état de votre système.

Par exemple, sous Anaconda ou avec pip, le simple fichier suivant

```
from numba import cuda
cuda.detect()
```

pourrait renvoyer l'erreur suivante :

```
CudaSupportError: Error at driver init:

CUDA driver library cannot be found.
If you are sure that a CUDA driver is installed,
```

```
try setting environment variable NUMBA_CUDA_DRIVER
with the file path of the CUDA driver shared library
```

(ici, le driver matériel n'a pas été installé).

Utilisation de Numba sur Google Colab

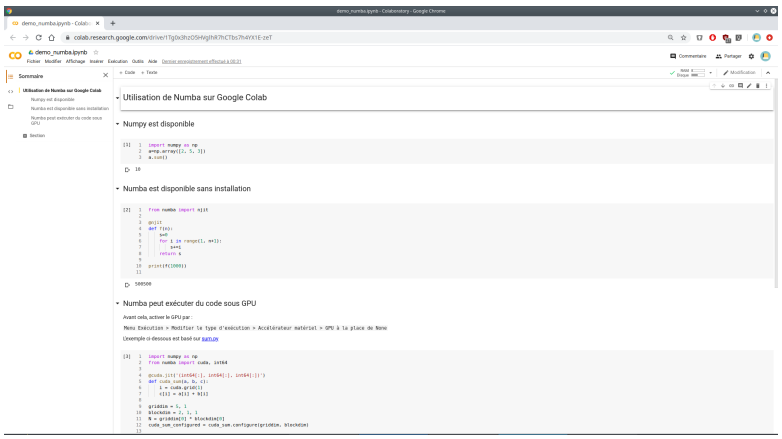
Google Colab est un service gratuit mis à la disposition d'un utilisateur ayant un compte Google et permettant de programmer dans des feuilles Jupyter Notebook et exécutées sur l'infrastructure de Google.

Je suppose que

— vous avez installé l'application Google Colaboratory sur votre Google-Drive

— vous avez déjà codé en Python dans des feuilles Jupyter Notebook.

La feuille que nous allons écrire ressemblera à ça :



et est disponible et exécutable en ligne ICI.

Modules disponibles

Les feuilles Google Colab sont destinées à faire des data sciences. En particulier, par défaut, les modules Python indispensables, comme Numpy ou SciPy, sont disponibles sans besoin de les installer. Par exemple, vous pouvez directement écrire dans une cellule du code comme

```
import numpy as np
a=np.array([2, 5, 3])
a.sum()
```

qui affichera

```
10
```

Mais, mieux que cela, **Numba est disponible** par défaut, autrement dit, déjà installé. Donc vous pouvez directement écrire un code tel que

```
from numba import njit
```

```
@njit
def f(n):
    s=0
    for i in range(1, n+1):
        s+=i
    return s

print(f(1000))
```

et qui affichera

```
500500
```

Version disponible

Testons la version utilisée :

```
import numba
numba.__version__
```

```
0.51.2
```

Pour le coup, ce n'est pas la dernière disponible qui est, en mai 2021, la version 0.53. On essaye de mettre à jour avec une commande spéciale

```
!pip install --upgrade numba
```

(sans omettre le point d'exclamation) qui affiche

Après redémarrage de l'environnement d'exécution effectué en cliquant sur le bouton qui est apparu (**Restart Runtime**), si on teste :

```
import numba
numba.__version__
```

on voit que Numba a été mis à jour :

```
0.53.1
```

Pour l'usage des GPU sous Google Colab, voir le paragraphe dédié dans la partie correspondante.

En outre, il est possible, avec des commandes spécifiques d'installer la plupart des bibliothèques d'apprentissage non déjà installées (voir un exemple avec Cupy dans la dernière partie).

Installation de Numba avec pip

Ce qui suit s'applique à Windows, Linux et masOS. Utiliser pip pour installer Numba est la méthode la plus simple. Bien que cette installation ne suffise pas à utiliser un éventuel GPU, elle délivre déjà une bonne partie de la puissance de Numba.

Il suffit d'utiliser pip en ligne de commande de la manière suivante (ici sous Linux) :

```
po@po-18:~$ pip3 install numba
Collecting numba
  Downloading https://files.pythonhosted.org/packages/73/c1/
  7e8bf88ee00aa89270c364c22c3f21a1e23243cbcaf3f0714fd5af953
  a34/numba-0.50.1-cp36-cp36m-manylinux1_x86_64.whl (2.6MB)
    100% |████████████████████████████████| 2.6MB 731kB/s
Collecting llvmlite<0.34,>=0.33.0.dev0 (from numba)
  Downloading https://files.pythonhosted.org/packages/d5/37/
  10d2c4ba0c131cf112d0a8fc97af7a77ca4a769aff626b2bfa2e57e679
  cf/llvmlite-0.33.0-cp36-cp36m-manylinux1_x86_64.whl (18.3MB)
    100% |████████████████████████████████| 18.3MB 104kB/s
Collecting setuptools (from numba)
  Downloading https://files.pythonhosted.org/packages/8e/11/
  9e10f1cad4518cb307b484c255cae61e97f05b82f6d536932b1714e01b
  47/setuptools-49.2.0-py3-none-any.whl (789kB)
    100% |████████████████████████████████| 798kB 2.4MB/s
Collecting numpy>=1.15 (from numba)
  Downloading https://files.pythonhosted.org/packages/22/e7
  /4b2bdddb99f5f631d8c1de259897c2b7d65dcfcc1e0a6fd17a7f62923
  500/numpy-1.19.1-cp36-cp36m-manylinux1_x86_64.whl (13.4MB)
    100% |████████████████████████████████| 13.4MB 143kB/s
Installing collected packages: llvmlite, setuptools, numpy, numba
Successfully installed llvmlite-0.33.0 numba-0.50.1 numpy-1.19.1
 setuptools-49.2.0
```

Vous noterez que l'installation de Numba entraîne l'installation de Numpy.

Une fois Numba installé, testez en exécutant sous Python le programme suivant

tester_numba.py

```
from numba import njit

@njit
def f(n):
    s=0
    for i in range(1, n+1):
        s+=i
    return s

print(f(1000))
```

et qui doit afficher

```
500500
```

On peut vérifier la version :

```
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numba
>>> numba.__version__
'0.50.1'
>>>
```

Mise-à-jour

Pour mettre à jour une version de Numba, utiliser en ligne de commande (testé sous Linux seulement) :

```
$ pip3 install --upgrade numba
WARNING: pip is being invoked by an old script wrapper. This will fail in a future ver
Please see https://github.com/pypa/pip/issues/5599 for advice on fixing the underlying
To avoid this problem you can invoke Python with '-m pip' instead of running pip direc
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: numba in /home/po/.local/lib/python3.6/site-packages (0
Collecting numba
  Downloading numba-0.53.1-cp36-cp36m-manylinux2014_x86_64.whl (3.4 MB)
     |████████████████████████████████████████████████████████████████| 3.4
Requirement already satisfied: numpy>=1.15 in /home/po/.local/lib/python3.6/site-packa
Requirement already satisfied: setuptools in /home/po/.local/lib/python3.6/site-packag
Collecting llvmlite<0.37,>=0.36.0rc1
  Downloading llvmlite-0.36.0-cp36-cp36m-manylinux2010_x86_64.whl (25.3 MB)
     |████████████████████████████████████████████████████████████████| 25
Installing collected packages: llvmlite, numba
  Attempting uninstall: llvmlite
    Found existing installation: llvmlite 0.33.0
    Uninstalling llvmlite-0.33.0:
      Successfully uninstalled llvmlite-0.33.0
  Attempting uninstall: numba
    Found existing installation: numba 0.50.0
    Uninstalling numba-0.50.0:
      Successfully uninstalled numba-0.50.0
Successfully installed llvmlite-0.36.0 numba-0.53.1
```

Utilisation d'Anaconda

Anaconda Individual Edition est une suite gratuite et open-source d'outils (je dirai parfois *toolkit*) permettant de faire des data sciences en Python. Elle facilite en particulier l'installation et la maintenance des outils utilisés.

En installant Anaconda 3, vous pourrez utiliser pleinement Numba, y compris en faisant appel à un GPU si votre machine en dispose.

Se rendre sur la page de téléchargement du toolkit et téléchargez la version en fonction de votre OS (le fichier a une taille autour de 500 Mo). Pour macOS, choisir le version en ligne de

commande.

L'installation est expliquée ci-dessous selon l'OS que vous utilisez.

Dès qu'Anaconda est installé, Numba est accessible dans sa version CPU. Pour utiliser le GPU (si disponible sur votre machine), il faudra compléter l'installation (4e partie du document).

Installation d'Anaconda sous Linux ou macOS

La version d'Anaconda que vous aurez téléchargée est un gros script shell ayant un nom tel que :

```
Anaconda3-2020.07-Linux-x86_64.sh
```

(pour macOS, il faut avoir téléchargé l'installeur en ligne de commande).

Ce qui suit a été réalisé sous Ubuntu versions 18 et 20, c'est analogue sous macOS. L'installation dure autour de 5 minutes en principe. En ligne de commande, se placer dans le répertoire où se trouve le fichier ci-dessus et lancer la commande

```
$ bash Anaconda3-2020.07-Linux-x86_64.sh
```

Cela va installer Anaconda 3 dans le répertoire type `/home/moi/anaconda3`. Il installe **énormément** d'outils (des dizaines voire des centaines), en particulier toutes les bibliothèques utilisées en apprentissage automatique. Toute l'installation se fait en mode texte mais vous n'aurez qu'à répondre à des questions en tapant **yes** (si vous êtes d'accord).

A la dernière étape de l'installation, l'installeur va vous demander s'il doit initialiser Anaconda avec la commande `conda init`. Si vous répondez **yes**, Anaconda modifiera votre fichier caché bashrc avec les conséquences suivantes :

— Anaconda deviendra votre installation de Python par défaut (au lieu de l'installation système)

— le prompt de votre terminal sera modifié dans toutes ses utilisations (il sera précédé de **(base)**, voir ci-dessous)

— vous aurez accès à tous les outils d'Anaconda directement depuis votre console.

Plus précisément, votre bash aura la forme suivante :

```
(base) po@po-ta:~$
```

Personnellement, je réponds **no** car on peut accéder aux possibilités d'Anaconda en passant par leur navigateur maison.

Au passage, il faut savoir que par défaut, Anaconda installe Python 3.8 et pour le lancer on tape juste **python**, sans numéro (y compris sur les versions d'Ubuntu antérieures à la 20.04).

Pour faire des essais ou si vous savez ce que vous faites, vous pouvez répondre **yes**. De toutes façons, vous pouvez toujours revenir en arrière en supprimant du fichier bashrc ce qu'Anaconda a rajouté, à savoir le code (équivalent au) suivant :

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!

__conda_setup="$('/home/po/anaconda3/bin/conda' \
                 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
```

```
else
    if [ -f "/home/po/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/home/po/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/home/po/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup

# <<< conda initialize <<<
```

Pour revenir en arrière, il vous suffit de le supprimer du fichier bashrc ou de le commenter en mettant des caractères dièse devant chaque ligne, comme ceci :

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!

# __conda_setup="$('/home/po/anaconda3/bin/conda' \
#                  'shell.bash' 'hook' 2> /dev/null)"
# if [ $? -eq 0 ]; then
#     eval "$__conda_setup"
# else
#     if [ -f "/home/po/anaconda3/etc/profile.d/conda.sh" ]; then
#         . "/home/po/anaconda3/etc/profile.d/conda.sh"
#     else
#         export PATH="/home/po/anaconda3/bin:$PATH"
#     fi
# fi
# unset __conda_setup

# <<< conda initialize <<<
```

Plus simple encore, vous pouvez désactiver le shell conda par :

```
(base) po@po-ta:~$ conda deactivate
po@po-ta:~$
```

mais cela n'aura un effet que dans le terminal où vous aurez tapé la commande.

Pour lancer le navigateur d'Anaconda, normalement vous tapez

```
$ anaconda-navigator
```

Si comme moi, vous n'avez pas demandé à l'installeur de changer votre bashrc, il faut écrire en console le nom **complet** de l'exécutable, à savoir :

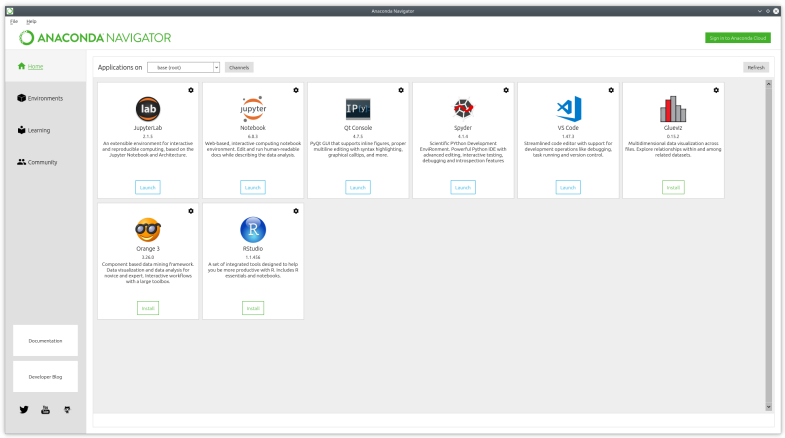
```
$ ~/anaconda3/bin/anaconda-navigator
```

Si vous êtes amené à répéter ce genre de commande souvent, autant écrire dans votre bashrc un alias, par exemple :

```
alias anav="~/anaconda3/bin/anaconda-navigator"
```

et vous n'aurez plus qu'à taper `anav` pour lancer la navigateur.

Quand vous lancez le navigateur (temps de chargement assez long), il a l'allure suivante :



Vous pouvez alors utiliser différents éditeurs (Jupyter Notebook, Spyder, VSC) en cliquant sur le bouton **Launch** pour écrire, exécuter, sauvegarder votre code Python et qui sera exécuté par le toolkit.

Installation d'Anaconda sous Windows 10

Pour disposer de Numba, je vais utiliser la suite Anaconda 3 qu'il faut avant tout, installer.

Installation d'Anaconda 3 Individual Edition

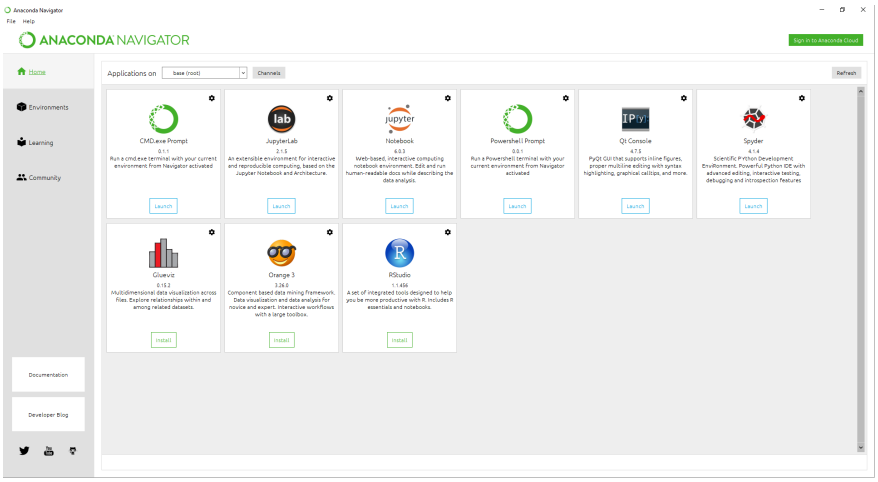
Se rendre ICI et télécharger la version pour Windows. Installez. C'est assez long (compter 10 bonnes minutes), la barre de progression n'est pas pertinente.

Il installe **énormément** d'outils (des dizaines voire des centaines), en particulier toutes les bibliothèques utilisées en apprentissage automatique. Vous n'avez quasiment rien à faire et je vous suggère d'accepter les options par défaut, en particulier qu'Anaconda devienne votre installation de Python par défaut, car cela simplifie beaucoup les installations. A la fin de l'installation, Anaconda aura installé dans mon répertoire `C:\Users\po\anaconda3` quelques `130 000` fichiers sur 5 Go !

Au passage, il faut savoir que par défaut, Anaconda installe Python 3.8 et pour le lancer en ligne de commande (si cela vous arrive) on tape juste **python**, sans numéro.

A la fin de l'installation, une entrée est crée dans le menu Démarrer. Pour utiliser Anaconda, le plus simple est de lancer leur navigateur, il s'appelle Anaconda-Navigator.

Quand vous lancez le navigateur (temps de chargement assez long), il a l'allure suivante :



Vous pouvez alors utiliser différents éditeurs (Jupyter Notebook, Spyder, VSC) en cliquant sur le bouton **Launch** pour écrire, exécuter, sauvegarder votre code Python et qui sera exécuté par le toolkit.

La désinstallation d'Anaconda est très très longue (tant il installe de fichiers).

Utiliser Numba sous Windows avec Anaconda

Quand vous installez Anaconda, bonne nouvelle : Numba est déjà installé (mais cette installation permet uniquement un usage sans GPU). Pour tester, il suffit d'ouvrir une feuille Jupyter Notebook (qu'Anaconda a installé aussi) :

```
Menu Démarrer > Anaconda3 > Jupyter Notebook
```

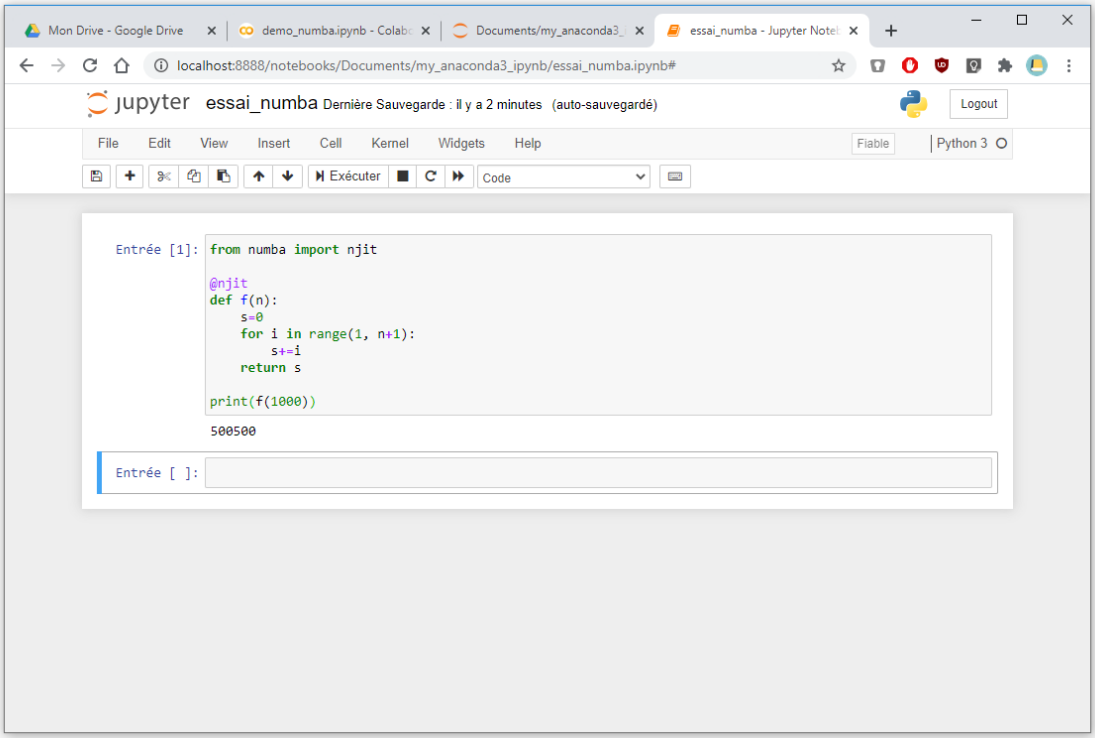
Créez une feuille de travail (cliquer sur le bouton **New**), dans mon cas `essai_numba.ipynb`, et dans une cellule de code, placez le code :

```
1 from numba import njit
2
3 @njit
4 def f(n):
5     s=0
6     for i in range(1, n+1):
7         s+=i
8     return s
9
10 print(f(1000))
```

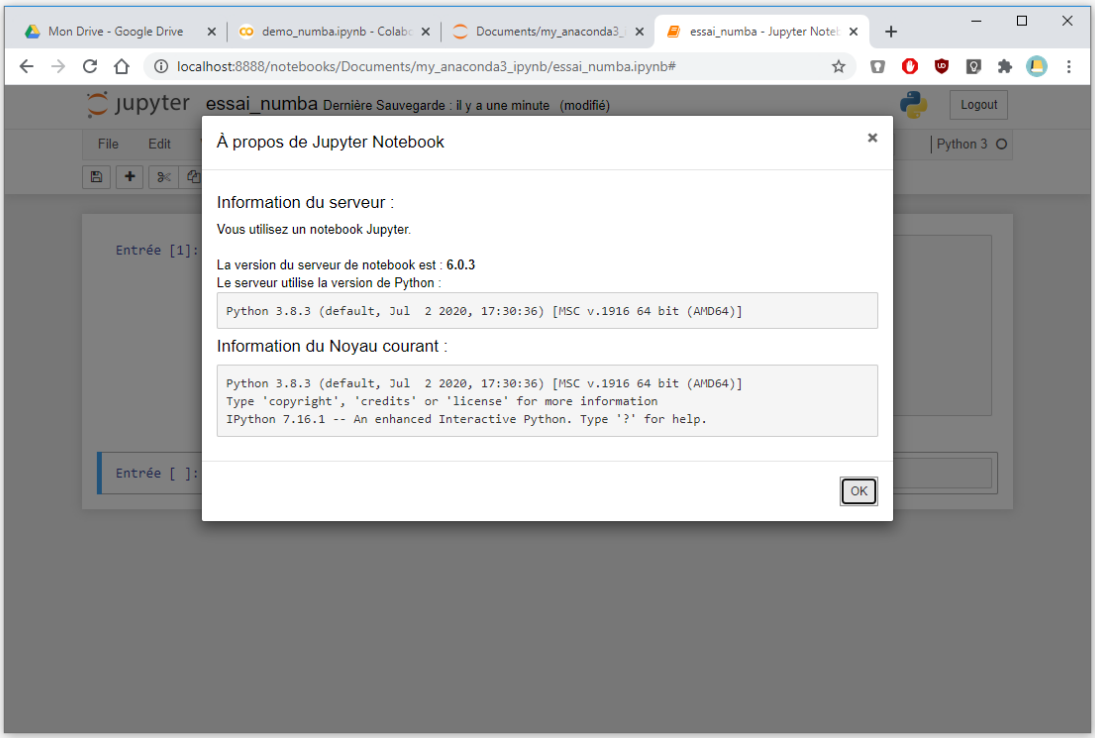
Il utilise Numba (cf. ligne 1). Exécutez le code en cliquant sur le triangle dans la barre d'outils et vérifiez que la sortie est :

```
500500
```

La feuille apparaît sous cette forme :



Au passage, on peut vérifier la version de Jupyter Notebook, de python et de IPython :



Utiliser Numba via Anaconda sous Linux

Anaconda 3 installe par défaut Numba (mais cette installation permet uniquement un usage sans GPU). Nous allons faire un test avec le fichier suivant :

tester_numba.py

```
from numba import njit

@njit
def f(n):
    s=0
    for i in range(1, n+1):
        s+=i
    return s

print(f(1000))
```

Pour comprendre ce qui suit, il faut avoir lu la partie sur l'installation d'Anaconda.

Prompt sous Anaconda

Si votre bashrc a été modifié par Anaconda au moment de l'installation, vous faites comme vous faisiez pour éxécuter un fichier Python avant l'installation d'Anaconda. Ce qui donne par exemple :

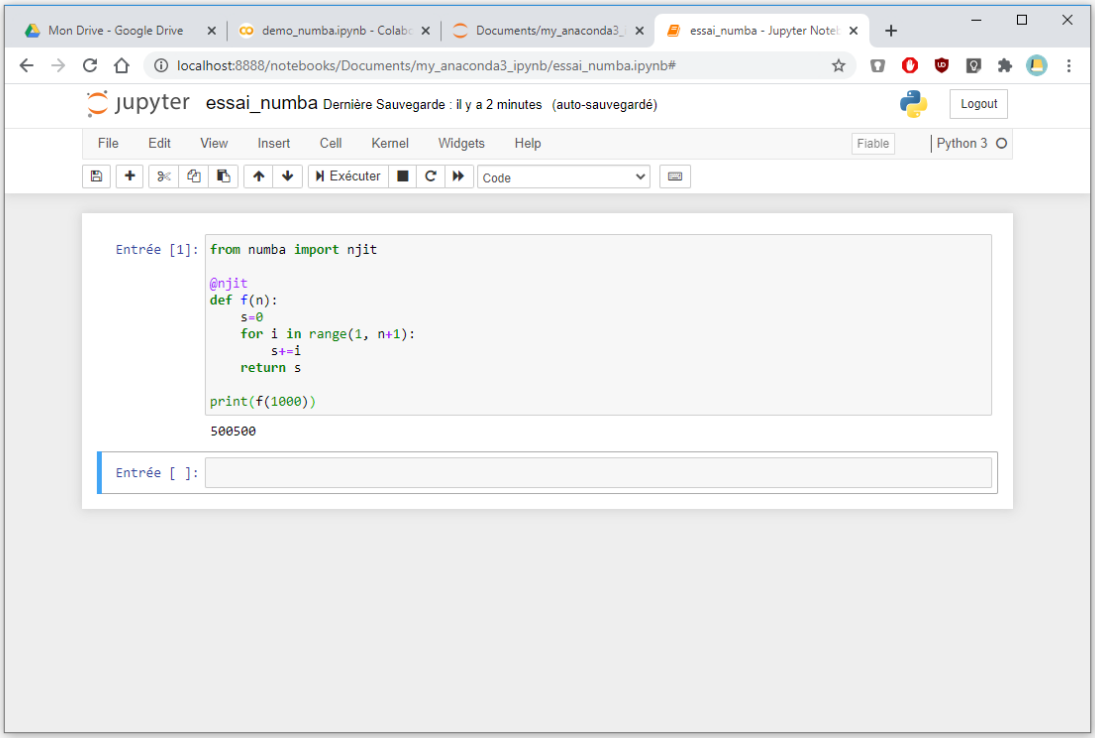
```
(base) po@po-ta:~$ python tester_numba.py
500500
(base) po@po-ta:~$
```

En passant par le Navigator

Vous ouvrez Anaconda Navigator, puis vous crééz et ouvrez une feuille Jupyter Notebook (cliquer sur le bouton *Launch*) et y tapez le code dans une cellule. Vous exécutez la cellule, Numba devrait être reconnu et le code s'exécuter en affichant

```
500500
```

En visuel :



Les deux modes de fonctionnement de Numba

Le code auquel Numba s'applique s'exécute sous deux modes :

— soit le mode *object*,

— soit le mode *nopython*.

C'est uniquement le second mode qui est susceptible d'apporter des gains de performances et donc le premier mode ne sera **pas du tout utilisé** dans ce document.

En outre, l'efficacité du mode nopython sur des structures de données tient à l'utilisation quasi-obligatoire de tableaux Numpy même s'il faut parfois nuancer quand on utilise des structures de données propres à Numba.

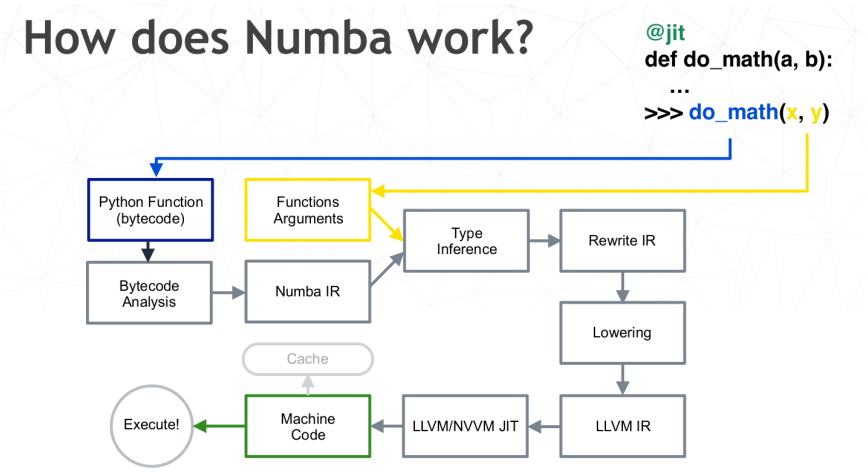
Lorsque le programme s'y prête, Numba produit un code qui s'exécute à la vitesse du C++, parfois même plus rapide. Et le code reste écrit en un Python très lisible, surtout si on est habitué à utiliser Numpy. Il en est de même du parallélisme sur processeur multicœur qui est beaucoup plus simple à écrire que si on utilise le module standard multiprocessing.

Concernant la programmation Cuda (si on dispose d'une carte graphique Nvidia), l'intérêt est qu'on a accès à la puissance de calcul d'un GPU tout en écrivant un code Python beaucoup plus simple et de beaucoup plus haut niveau que son équivalent Cuda C++.

Fonctionnement interne de Numba

Techniquement, Numba en mode nopython est un compilateur JIT (compilation en code natif au moment de l'exécution du code). Il N'utilise PAS l'API C de Python (d'où le terme de *nopython*), autrement dit il ne crée pas une extension Python comme le sont ou le font la plupart des programmes qui accélèrent Python (comme Numpy, SciPy, pybind11, Cython, etc).

Numba crée en 8 étapes (et plusieurs sous étapes) son propre binaire exécutable :



— la première étape est l'analyse du bytecode Python,
— d'autres étapes servent, à partir de la connaissance ou l'inférence des types des données utilisées par la fonction à compiler, à générer une représentation intermédiaire (IR) du code, propre à Numba
— les dernières étapes consistent à envoyer cette représentation à l'infrastructure de compilateur LLVM qui en génère une représentation intermédiaire puis qui génère du code assembleur optimisé propre à la plateforme qui exécutera le code.

Ici, je me limite à utiliser Numba sous l'interpréteur CPython (le plus courant) mais il semblerait que Numba soit potentiellement utilisable sous Pypy : Running Numba on PyPy.

Limitations, inconvénients de Numba

L'utilisation de Numba en mode nopython entraîne de nombreuses contraintes par rapport à la programmation traditionnelle en Python. Ci-dessous, les principales que je relève

— Numba n'apporte un gain de performance que sur un type limité de programmes qui utilisent des tableaux Numpy et/ou des boucles et des fonctions ou algorithmes mathématiques. Les tableaux Numpy doivent, pour être utilisables par Numba, être d'un type, d'une part, numérique (des double, des int64, etc) et, d'autre part, unique (pas de panachage de types comme une structure en C où on pourrait assembler un flottant et un int par exemple). En outre, on doit accepter la contrainte qu'un tableau Numpy ne peut contenir des tableaux Numpy de tailles différentes (ce qu'on appelle parfois des ragged array, des « tableaux en escalier »).
— De très nombreuses fonctionnalités Python (et même de Numpy) ne sont pas prises en charge
— Toutes les données doivent être typées. Le moindre défaut, la moindre incohérence ou ambiguïté de typage entraîne un rejet du programme avec des messages d'erreur assez cryptiques
— Efficacité très inégale des structures de données de Numba autres que les tableaux Numpy, les listes Numba étant souvent lentes
— La conséquence de l'inefficacité (variable) des Listes Numba est qu'on ne dispose pas de listes extensibles en mode nopython (comme les listes habituelless de Python ou les *vector* de la STL du C++), ce qui est très pénalisant dans de nombreuses formes d'algorithmes

— Attente de la phase de compilation avant d'obtenir la bonne performance
— Pas mal de bugs même si la communauté est très réactive
— Évolution rapide des versions donc nécessité de souvent mettre à jour (pour suivre l'évolution des versions, consulter ICI)
— Communauté d'utilisateurs assez réduite : en mai 2021, sur Stack Overflow, je compte
  — 1541 questions Numba
  — 4523 pour Cython,
  — 18122 pour SciPy
  — 89949 pour Numpy.
— Me semble assez peu utilisé en production, peut-être utilisé en prototypage.

Ressources pour Numba

— La documentation de la dernière version au format Readthedocs
— La documentation de la dernière version au format pdf
— Description des nouveautés et améliorations de chaque version, y compris la dernière disponible.
— Présentations pdf de Numba d'avril 2018 mais riche en information et toujours d'actualité, par Stanley Seibert, ex-manager de Numba.
— Dépôt Github de Numba.
— Questions Numba sur Stack Overflow
— Portail de la mailing list Numba sur Google groups

Présentations vidéo de Numba

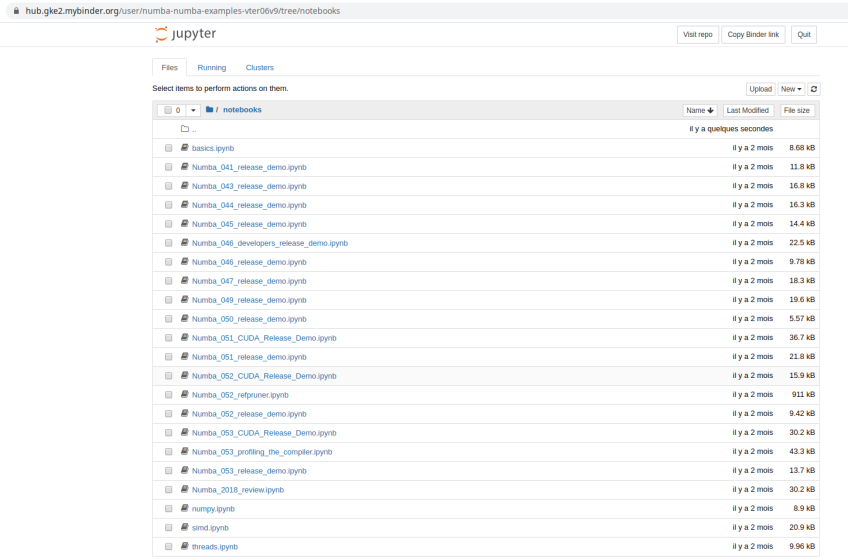
(Vérifié en mai 2021)

— Numba - Stanley Seibert, mai 2019, par le directeur de l'innovation à Anaconda (qui développe Numba)
— How to Accelerate an Existing Codebase with Numba par Stanley Seibert, SciPy 2019
— Accelerating Scientific Workloads with Numba par Siu Kwan Lam, un des principaux développeurs de Numba (avril 2018).
— Vidéo (2019) très pédagogique sur la programmation GPU : High-Performance Computing with Python: Numba and GPUs

Essayer Numba sur le site mybinder

Pour découvrir et essayer Numba, vous pouvez le faire en ligne sur le site mybinder. Il y a une limitation : les GPU ne sont pas disponibles (pas en libre accès en tous cas).

Après avoir cliqué sur le lien, attendez que l'environnement Jupyter Notebook se charge. Vous pouvez alors examiner les feuilles d'exemples proposés :



ou créer votre propre feuille Jupyter Notebook en cliquant sur **New**. Vous pouvez aussi uploader votre propre feuille Jupyter.

La dernière version de Numba est proposée. Les temps d'exécution sont relativement lents par rapport à ce que donne ma propre machine munie d'un processeur i5 de 2015.

L'exécutable numba

Quand le module Numba est installé un exécutable en ligne de commande du nom de **numba** est également installé. Il est destiné à des utilisateurs avancés et qui cherchent à examiner les différentes phases lors du processus de compilation (annotation du bytecode, code assembleur LLVM généré, etc).

L'exécutable numba dispose d'une option généraliste, l'option -s, qui donne des informations sur le système et qui sont utilisées par Numba. Voici un extrait (mai 2021) :

```
$ numba -s
System info:
--------------------------------------------------------------------------------
...
__Hardware Information__
Machine                                         : x86_64
CPU Name                                        : haswell
CPU Count                                       : 4
...

CPU Features                                    : 64bit aes avx avx2 bmi bmi2 cmov
                                                  cx16 cx8 f16c fma fsgsbase fxsr
                                                  invpcid lzcnt mmx movbe pclmul
                                                  popcnt rdrnd sahf sse sse2 sse3
                                                  sse4.1 sse4.2 ssse3 xsave xsaveopt
```

```
...
Memory Available (MB)                          : 3625

__OS Information__
Platform Name                                  : Linux-4.15.0-117-generic-x86_64-with-U
...
Libc Version                                   : glibc 2.25

__Python Information__
Python Compiler                                : GCC 8.4.0
Python Implementation                          : CPython
Python Version                                 : 3.6.9
Python Locale                                  : fr_FR.UTF-8

__LLVM Information__
LLVM Version                                   : 10.0.1

__CUDA Information__
CUDA Device Initialized                        : True
CUDA Driver Version                            : 10020
CUDA Detect Output:
Found 1 CUDA devices
id 0      b'GeForce GTX 970'                              [SUPPORTED]
                      compute capability: 5.2
                           pci device id: 0
                              pci bus id: 1


CUDA Libraries Test Output:
Finding cublas from System
        named  libcublas.so
        trying to open library...       ok

...

__Installed Packages__

...
```

Informations diverses

— Le développement du package Numba est réalisé sous l'égide de l'entreprise Anaconda.

— Numba a été créé en 2012 par Travis Oliphant, qui est le fondateur de la bibliothèque Numpy et le créateur de Continuum Analytics, devenu par la suite Anaconda.

— Le code source de Numpy est essentiellement écrit en C et en Python. Le package llvmlite fait la liaison entre Numba et une petite partie, enveloppée en C, de l'API C++ de la bibliothèque LLVM. La liaison avec Python est assurée par le module ctypes.

— On lit de-ci de-là, que PyCuda serait plus rapide que Numba. Cela semble basé sur d'anciens

benchmarks comparant NumbaPro (ancêtre de Numba) et PyCuda.

— Un article de Lena Olden en 2020 semblerait indiquer que Numba serait moins performant que d'utiliser directement Cuda en C. L'article n'est pas disponible mais l'autrice expose ses résultats dans cette vidéo : Lessons learned from comparing Numba-CUDA and C-CUDA.

Chapitre II

Programmation en Numba sur le CPU

L'efficacité de Numba sur le tri par sélection

Soit un code Python d'un simple tri par sélection :

```
def selection(L):
    n=len(L)
    for i in range(n-1):
        jMin=i
        mini=L[i]
        for j in range(i+1, n):
            if L[j] < mini:
                jMin=j
                mini=L[j]
        if mini!=L[i]:
            L[i], L[jMin]= L[jMin], L[i]

L = [19, 16, 18, 14, 18, 17, 17, 18, 14, 15]
print(L)
selection(L)
print(L)
```

qui affiche :

```
[19, 16, 18, 14, 18, 17, 17, 18, 14, 15]
[14, 14, 15, 16, 17, 17, 18, 18, 18, 19]
```

Testons son temps d'exécution sur une liste de 20000 entiers :

```
from time import perf_counter
from random import randrange

def selection(L):
    n=len(L)
    for i in range(n-1):
        jMin=i
        mini=L[i]
```

```
        for j in range(i+1, n):
            if L[j] < mini:
                jMin=j
                mini=L[j]
        if mini!=L[i]:
            L[i], L[jMin]= L[jMin], L[i]

N=20000
L=[randrange(10**6) for _ in range(N)]

print("Sorting ...")

begin_perf = perf_counter()

selection(L)

delta_py = perf_counter() - begin_perf

print(f"Python : {delta_py:.2f}s")
```

qui affiche

```
Sorting ...
Python : 7.65s
```

Maintenant, dans un autre fichier, on va soumettre le même test au package Numba (qui doit être disponible sur votre installation). Le nouveau code est :

```
1  from time import perf_counter
2  from random import randrange
3  from numba import njit
4  import numpy as np
5
6  @njit
7  def selection_nb(L):
8      n=len(L)
9      for i in range(n-1):
10         jMin=i
11         mini=L[i]
12         for j in range(i+1, n):
13             if L[j] < mini:
14                 jMin=j
15                 mini=L[j]
16         if mini!=L[i]:
17             L[i], L[jMin]= L[jMin], L[i]
18
19 N=20000
20 L=[randrange(10**6) for _ in range(N)]
21 M=np.array(L, dtype=np.int64)
22 print("Sorting ...")
```

```
23
24 # warmup
25 selection_nb(np.array([42], dtype=np.int64))
26
27 begin_perf = perf_counter()
28
29 selection_nb(M)
30
31 delta = perf_counter() - begin_perf
32
33 print(f"Numba : {delta:.2f}s")
```

Voici les points communs entre les deux codes :

— le corps de la fonction `selection` reste totalement inchangé (lignes 7-17)

— on fait un appel à la fonction de la même façon (ligne 29).

Voici les différences :

— on doit importer du module Numba le nom `njit` (ligne 3)

— la fonction `selection` est décorée par le décorateur `njit` (ligne 6)

— on doit importer le module Numpy (ligne 4)

— la liste `L` donnée dans le test initial doit être convertie en un tableau Numpy (ligne 21).

Pour comprendre ce qui suit, vous n'avez pas besoin de savoir ce qu'est un décorateur (cf. ligne 6), il vous suffit d'admettre qu'il faut l'écrire au-dessus du nom de la fonction avec la syntaxe indiquée pour qu'elle soit exécutée en mode nopython.

La conversion de la liste Python en un tableau Numpy est un point essentiel (ligne 21).

Et quand on teste ce nouveau code, il affiche un joli :

```
Sorting ...
Numba : 0.21s
```

On voit donc que le code Numba traite le même problème en étant **36 fois plus rapide**. On est très proche des performances du C++ : le même code mais récrit en C++ demande un temps de 0.18s.

Phase de compilation

Il y a un point très important à préciser et qui correspond aux lignes 24-25. Numba est un compilateur *Just in time*, donc, quand il rencontre la fonction `selection` pour la première fois, il agit en suivant les étapes ci-dessous :

— il compile la fonction `selection` autrement dit Numba produit le code natif correspondant (comprendre : le code-machine de l'architecture-hôte) à la fonction

— le code natif est placé en cache,

— le code natif est exécuté.

Ensuite, tout appel à la fonction `selection` lance en fait le code natif placé en cache.

Or, la compilation peut durer un certain temps, même parfois très important en C++ par exemple. Donc **avant** de mesurer des performances, on génère le code natif en faisant un (petit) essai ce

qui permet en particulier de donner à Numba le **type** des données utilisées afin qu'il puisse le compiler ; j'ai qualifié cette phase de *warmup*.

Retenons, comme l'indique la documentation :

A really common mistake when measuring performance is to not account for the above behaviour and to time code once with a simple timer that includes the time taken to compile your function in the execution time

Commentaire de code

— Ligne 6 : le décorateur `@njit` a été placé au-dessus de la fonction ce qui fait référence à :

  — *just in time* pour *jit*

  — *nopython* mode pour le *n*

  C'est ce mode qui permet d'obtenir souvent d'excellentes performances.

— Ligne 3 : le décorateur `njit` est importé de Numba

— La liste Python d'entiers ligne 20 est convertie en un tableau Numpy de `int64` (entiers sur 64 bits, équivalent du long long en C++)

— On doit importer Numpy (ligne 4).

— Ligne 25 : la phase de compilation en code en natif, en général sur un exemple trivial et qui permet à Numba d'identifier le type des données utilisées, car il en a besoin pour compiler la fonction.

— Le reste du code est inchangé.

Il existe un décorateur Numba nommé `jit`, à paramètre et qui est plus général que le décorateur `njit`. Dans ce document, je n'utiliserai quasiment que la version `njit` puisque je souhaite exécuter le code exclusivement en mode nopython pour obtenir la meilleure performance d'exécution.

En conclusion, Numba est capable d'accélérer radicalement les performances d'un code Python au prix de la conversion des données traitées dans le format Numpy.

Numba ou Numpy ?

On pourrait se demander si dans le cas du tri par sélection l'amélioration de performance ne serait pas due à l'utilisation de Numpy. Il n'est est rien. Si on exécute la fonction `selection` sans recours à Numba, le code est **plus lent** que du code Python pur :

```
from time import perf_counter
from random import randrange
import numpy as np

def selection_np(L):
    n=len(L)
    for i in range(n-1):
        jMin=i
        mini=L[i]
        for j in range(i+1, n):
            if L[j] < mini:
```

```
                jMin=j
                mini=L[j]
        if mini!=L[i]:
            L[i], L[jMin]= L[jMin], L[i]

N=20000
L=[randrange(10**6) for _ in range(N)]
M=np.array(L, dtype=np.int64)
print("Sorting ...")


begin_perf = perf_counter()

selection_np(M)

delta = perf_counter() - begin_perf

print(f"Numpy : {delta:.2f}s")
```

```
Sorting ...
Numpy : 27.54s
```

Rappelons que le temps en Python pur était de l'ordre de 7.65s donc le code est devenu plus de trois fois plus lent. Deux conclusions :

— Numpy ne rend pas votre code Python plus rapide ;

— c'est l'association de Numpy et Numba qui produit un code très efficace.

Problèmes de types ou de non-reconnaissance

Si vous vous décidez à utiliser fréquemment Numba, vous serez souvent confronté à des problèmes de typage. Ci-dessous quelques exemples variés.

Voici un problème dû à un typage hybride :

```
import numpy as np
from numba import njit

@njit
def f(t,x):
    t[0]=x

wedges = np.array([(3.14, 1), (2.71, 4)], dtype='float32,int32')
wedge=(0.47, 42)
f(wedges, wedge)
```

qui affiche (sortie tronquée) :

```
Traceback (most recent call last):

...
```

```
No implementation of function Function(<built-in function setitem>)
found for signature:

 >>> setitem(unaligned array(Record(f0[type=float32;offset=0],
 f1[type=int32;offset=4];8;False), 1d, C), Literal[int](0), Tuple(float64, int64))

There are 10 candidate implementations:

...
```

Le code suivant ne reconnait pas la fonction Python `sum` :

```
from numba import njit
import numpy as np

@njit
def f(L):
    s = sum(L)
    return s

L=np.array([5])
f(L)
```

```
Traceback (most recent call last):

...

Untyped global name 'sum': cannot determine Numba type
of <class 'builtin_function_or_method'>
```

Le code suivant ne reconnait pas la fonction Python str :

```
from numba import njit
import numpy as np

@njit
def f(L):
    n=L[0]
    s = str(n)
    return s

L=np.array([5])
f(L)
```

```
Traceback (most recent call last):

...

No implementation of function Function(<class 'str'>)
found for signature:
```

```
...
```

Le code suivant est rejeté car le type de H n'est pas connu :

```
from numba import njit
import numpy as np

@njit
def f(L):
    n=len(L)
    H=[]

f(np.array([42]))
```

Débordement d'entiers

Par ailleurs, les entiers de Python, qui peuvent être de longueur arbitraire, ne sont pas reconnus par Numba en mode `jit`, comme expliqué ICI. Par exemple, la somme des carrés des `n` premiers entiers calculée sous ce mode sera fausse dès que `n` sera assez grand :

```
from numba import jit

@jit
def nbsum2(n):
    s=0
    for i in range(n+1):
        s+=i*i

    return s

for n in [10**6, 5*10**6]:
    s=nbsum2(n)
    S=sum(i*i for i in range(n+1))
    print(f"n = {n} : {s==S}")
```

```
n = 1000000 : True
n = 5000000 : False
```

On voit que pour $n = 10^6$, la calcul est correct mais qu'il ne l'est plus pour $n = 5 \times 10^6$.

Numba appliqué au produit matriciel

Le produit matriciel est une situation typique où Numba peut montrer toute son efficacité : l'opération comprend trois boucles imbriquées, et lit des tableaux statiques de données numériques homogènes.

Dans ce qui suit, on va utiliser une matrice `A` de taille 1000 x 1000, à coefficients aléatoires parmi 0 ou 1 et on va tester le temps d'exécution du produit $A \times A$.

Python pur

Voici le code en Python pur :

```
from time import perf_counter
from random import randrange

def product_direct(A, B):
    n, p=len(A), len(A[0])
    q=len(B[0])

    P=[[0]*q for _ in range(n)]
    for i in range(n):
        for j in range(q):
            for k in range(p):
                P[i][j]+=A[i][k]*B[k][j]
    return P

def build_matrix(n):
    return [[randrange(2) for _ in range(n)] for _ in range(n)]

n=1000
A=build_matrix(n)

begin_perf = perf_counter()

P=product_direct(A, A)

delta = perf_counter() - begin_perf

print(f"Python pur : {delta :.2f}s")
```

```
Python pur : 133.11s
```

Numpy pur

Essayons Numpy seul, le calcul matriciel étant un de ses domaines de prédilection. Numpy dispose d'une implémentation du produit matriciel via l'opérateur `@` :

```
from time import perf_counter
from random import randrange
import numpy as np

def product_np(A, B):
    return A@B

def build_matrix(n):
    return [[randrange(2) for _ in range(n)] for _ in range(n)]

n=1000

A=build_matrix(n)
```

```
begin_perf = perf_counter()

#  ---- Début de code à exécuter ----
Anp=np.matrix(A)
prod_np=product_np(Anp, Anp)
#  ---- Fin de code à exécuter ----

delta_np = perf_counter() - begin_perf

print(f"Numpy : {delta_np:.2f}s")
```

qui affiche un joli :

```
Numpy : 1.80s
```

Noter que le temps de conversion de la liste Python vers le tableau Numpy a été compté mais en pratique, quand on travaille sous environnement Numpy, les objets sont déjà au format Numpy. Quoi qu'il en soit, l'amélioration est d'un facteur 74, ce qui est très important.

Numba pur

On va reprendre le code en Python pur et l'adapter aux besoins de Numba. Pour cela, il suffit essentiellement

— de convertir la liste en tableau Numpy

— d'adapter la syntaxe.

Comme les temps mesurés me semblaient assez variables, le code a été placé dans une cellule d'une feuille Jupyter Notebook afin de pouvoir utiliser facilement `%timeit` (qui fait une moyenne des temps relevés) :

```
from time import perf_counter
from random import randrange
import numpy as np
from numba import njit
import numpy as np

@njit
def product_nb(A, B):
    n, p=A.shape
    p, q=B.shape

    P=np.zeros((n,q), dtype=np.int64)
    for i in range(n):
        for j in range(q):
            for k in range(p):
                P[i,j]+=A[i,k]*B[k,j]
    return P

def build_matrix(n):
    return [[randrange(2) for _ in range(n)] for _ in range(n)]
```

```
n=1000
A=build_matrix(n)

# warmup
B=np.matrix([[0, 1], [0, 0]], dtype=np.int64)
product_nb(B, B)

begin_perf = perf_counter()

%timeit -r10 Anp=np.matrix(A, dtype=np.int64);P=product_nb(Anp, Anp)
```

```
1.6 s ± 85.6 ms per loop (mean ± std. dev. of 10 runs, 1 loop each)
```

Il apparaît que le temps est encore meilleur que celui du code Numpy. Pour donner une idée, le même code écrit en C++ met 1.46s sur ma machine en utilisant les vector de la STL.

Un temps encore meilleur

S'il est question de comparer à Numpy, on peut optimiser légèrement l'algorithme. En effet quand on fait notre produit ligne par colonne, les éléments de la colonne ne sont pas contigus en mémoire et cela peut être source de perte de performance. Donc l'astuce classique consiste à

— boucler d'abord sur les colonnes et plutôt que sur les lignes (ci-dessous ligne 15)
— à chaque étape de la boucle en colonne, copier la colonne dans un tableau Numpy `t` (un buffer) qui, lui, est à mémoire contiguë (ci-dessous ligne 14, le tableau n'est créé qu'une seule fois)
— faire le produit « scalaire » de chaque ligne (dans le code ci-dessous, c'est ligne 18) et de `t` (ci-dessous, lignes 19-20)

Voici le code

```
1  from time import perf_counter
2  from random import randrange
3  import numpy as np
4  from numba import njit,int64
5  import numpy as np
6
7
8
9  @njit
10 def product_nb(A):
11     n, _=A.shape
12
13     P=np.zeros((n,n), dtype=np.int64)
14     col=np.empty(n, dtype=np.int64)
15     for j in range(n):
16         for k in range(n):
17             col[k]=A[k,j]
18         for i in range(n):
19             for k in range(n):
20                 P[i,j]+=A[i,k]*col[k]
```

```
    return P

def product_np(A):
    return A@A

def build_matrix(n):
    return [[randrange(2) for _ in range(n)] for _ in range(n)]

n=1000
A=build_matrix(n)


Anp=np.matrix(A)

# ============================================


begin_perf = perf_counter()

#  ---- Début de code à exécuter ----
Anp=np.matrix(A)
prod_np=product_np(Anp)
#  ---- Fin de code à exécuter ----

delta_np = perf_counter() - begin_perf

print(f"Numpy : {delta_np:.2f}s")


# ============================================

# warmup
B=np.matrix([[0, 1],
        [1, 0]])
product_nb(B)


begin_perf = perf_counter()

#  ---- Début de code à exécuter ----
Anp=np.matrix(A)
prod_nb=product_nb(Anp)
#  ---- Fin de code à exécuter ----

delta_nb = perf_counter() - begin_perf

print(f"Numba : {delta_nb:.2f}s")
```

qui affiche :

```
Numpy : 1.53s
Numba : 0.66s
```

Cette fois Numba est plus de deux fois plus rapide que Numpy. Cette astuce donne aussi des résultats en Python et encore plus en C++ si on utilise des vector de la STL.

Le produit matriciel et Numpy

Dans ce document, je vais beaucoup utiliser le produit matriciel pour effectuer des comparaisons de performances. L'algorithme est simple et est exigeant en calcul et en mémoire. Le temps de Numpy va servir de référence. Mais pour comparer des choses comparables, j'utiliserai des matrices à *coefficients entiers* et non pas flottants. En effet, l'implémentation de Numpy pour les tableaux en flottants est extrêmement rapide car elle s'appuie sur des bibliothèques spécialisées comme BLAS utilisant l'algorithme GEMM qui exploite les spécificités de l'architecture du processeur, en particulier les différents caches ; pour plus d'information sur cette implémentation, consulter le document (pdf) The Science of Programming Matrix Computations. En revanche, pour des nombres entiers, elle utilise l'algorithme ordinaire qui sera justement celui que j'utiliserai en Python, Numba ou C++ et donc plus approprié pour des comparaisons.

Pour être plus précis concernant l'implémentation du produit matriciel en Numpy, il est pris en charge par la fonction PyArrayMatrixProduct2 du fichier multiarraymodule.c. En regardant plus en détail l'implémentation :

```
#if defined(HAVE_CBLAS)
    if (PyArray_NDIM(ap1) <= 2 && PyArray_NDIM(ap2) <= 2 &&
            (NPY_DOUBLE == typenum || NPY_CDOUBLE == typenum ||
             NPY_FLOAT == typenum || NPY_CFLOAT == typenum)) {
        return cblas_matrixproduct(typenum, ap1, ap2, out);
```

on voit que si Numpy a été compilé avec la bibliothèque Blas (plutôt que *compilé*, en fait, c'est liée avec le linker) et si les matrices sont de type flottant (et pas entier) alors la bibliothèque Cblas est appelée pour réaliser le produit. En outre, le calcul peut être effectué sur plusieurs threads.

Pour savoir si Numpy a été lié a Blas, lire ce message.

Noter que d'une architecture à une autre on peut obtenir des temps très différents, même pour des produits matriciels en flottants.

Fonctions récursives

Il est parfaitement possible d'appeler des fonction récursives définies en mode nopython. Avec les deux bénéfices suivants :

— le plafonnement de taille de la pile des appels, de 1000 par défaut, disparaît ou devient très élévé

— l'exécution est très rapide, ce qui n'est pas le cas en Python pur.

Voici un premier exemple qui montre la disparition du plafonnement de la pile des appels : on calcule récursivement la somme des 200000 premiers entiers ;

```python
from time import perf_counter
from numba import njit

@njit
def s(n):
    if n ==0:
        return 0
    return s(n-1)+n

#warmup
s(1)

n=200000

begin_perf = perf_counter()

print(s(n)==n*(n+1)//2)

delta = perf_counter() - begin_perf

print(f"Temps d'exécution : {delta:.2f}s")
```

```
True
Temps d'exécution : 0.00s
```

Je me suis limité à 200000 pour éviter l'overflow, Numba calculant avec des entiers 64 bits.

Dans l'exemple qui suit, on calcule par une récursivité inefficace un élément du tableau de Pascal. En Python pur, un tel code nécessiterait plusieurs dizaines de secondes. Ici, le code est très rapide :

```python
from time import perf_counter
from numba import njit

@njit
def pascal(n, p):
    if 0 < p < n:
        return pascal(n-1, p)+pascal(n-1,p-1)
    return 1

#warmup
pascal(2,1)

n, p=30,14

begin_perf = perf_counter()

print(pascal(n,p))

delta = perf_counter() - begin_perf
```

```
print(f"Temps d'exécution : {delta:.2f}s")
```

```
145422675
Temps d'exécution : 0.92s
```

Fonction sans décorateur

Ce contenu n'est qu'en rapport très indirect avec Numba.

Parfois on souhaite comparer les performances d'une fonction et de la même fonction décorée par le décorateur `@njit`. Pour éviter de devoir écrire le code deux fois, on écrit la fonction, disons `f` sans décorateur et on obtient la fonction `g` compilée par Numba par `g = njit(f)`, après tout, c'est exactement la définition d'un décorateur. Exemple :

```
from time import perf_counter
from numba import njit

def f(n):
    s=0
    for i in range(n):
        s+=i//100
    return s

n=10**8
# ---------- Python -------------------

begin_perf = perf_counter()

s1=f(n)

delta = perf_counter() - begin_perf

print(f"f en Python : {delta:.2f}s")


# ---------- Numba --------------------

g=njit(f)
# warmup
g(1)

begin_perf = perf_counter()

s2=g(n)

delta = perf_counter() - begin_perf

print(f"f en Numba : {delta:.2f}s")

```

```
36 # ---------- Check -------------------
37
38 print(s1==s2)
```

```
39 f en Python : 6.86s
40 f en Numba : 0.03s
41 True
```

— Ligne 4 : on a écrit qu'une seule fois le code de la fonction dont les performances sont à comparer

— Lignes 24 et 30 : la fonction équivalente de f mais compilée par Numba.

La technique exposée ci-dessus est celle que l'on trouve dans le code source de Numba, par exemple :

```
pyfunc = array_dot_chain
cfunc = jit(nopython=True)(pyfunc)
```

Il existerait des alternatives plus élégantes, comme peut-être :

— disable decorators

— strip decorators

Situation de vectorisation

Soit le problème décrit ci-après, que l'on cherche à coder en Numpy de manière aussi efficace que possible. On part d'une tableau `t` d'entiers positifs, par exemple,

```
t = np.array([2, 4, 0, 3, 0, 1, 4])
```

et on veut créer un tableau `u` de flottants, de même longueur que `t` et tel que son élément à l'indice `i` soit obtenu de la manière suivante :

— si t[i] vaut 0 alors `u[i] = 1.0`

— sinon, et si on pose $k = t[i] > 0$ alors

  — on génère `k` flottants aléatoires entre 0 et 1,

  — on en fait le produit `p`

  — on pose `u[i] = p`.

En soi, le problème n'a rien de difficile, la question est surtout de pouvoir construire `u` rapidement. Chaque élément de `t` doit être transformé par la fonction `f` suivante :

```
def f(k):
    if k:
        return np.prod(np.random.random(k))
    return 1.0
```

Ensuite, on peut utiliser la méthode `vectorize` de Numpy mais qui en fait ne vectorise rien comme expliqué dans les notes de la documentation, elle est équivalente à une boucle `for` en Python pur.

D'où le code suivant :

```
import numpy as np

def f(k):
    if k:
        return np.prod(np.random.random(k))
    return 1.0

t = np.array([2, 4, 0, 3, 0, 1, 4])
print(t)

g=np.vectorize(f)
u=g(t)

print(*("%.2f" %z for z in u))
```

qui affiche

```
[2 4 0 3 0 1 4]
0.46 0.36 1.00 0.34 1.00 0.08 0.15
```

La fonction g a transformé le tableau `t` en le tableau u comme souhaité.

Le problème est que ce code est très peu performant. Faisons un essai avec un tableau `t` de 10 millions d'entiers (générés par une distribution de Poisson car présenté ainsi dans le problème d'origine mais ce point est sans importance) :

```
from time import perf_counter
import numpy as np

def f(k):
    if k:
        return np.prod(np.random.random(k))
    return 1.0

n=10**7
t=np.random.poisson(0.8,n)
k=10
# on affiche les k premiers éléments de t
print(t[:k])

start=perf_counter()

u=np.vectorize(f)(t)

end=perf_counter()
delta=end - start

# on affiche les k premiers éléments de u
print(*("%.2f" %z for z in u[:k]))

print(f"Durée : {delta :.2f}s")
```

qui affiche

```
[0 3 0 1 0 1 1 0 0 1]
1.00 0.24 1.00 0.67 1.00 0.36 0.13 1.00 1.00 0.66
Durée : 26.76s
```

Naturellement, on ne souhaite pas que l'opération soit aussi longue, vu sa simplicité c'est potentiellement réalisable en C en quelques dixièmes de seconde.

Si on maîtrise bien Numpy et qu'on a une certaine pratique de la vectorisation à l'aide des fonctions builtins de Numpy, ce qui suppose de connaître le chapitre Universal functions du manuel de référence, on pourra obtenir le code suivant :

```
from time import perf_counter
import numpy as np

def solve(t):
    t_plus=t>0
    t_cumul_plus=np.zeros(len(t[t_plus]))
    t_cumul_plus[1:]=np.cumsum(t[t_plus])[:-1]
    t_cumul_plus=t_cumul_plus.astype(int)
    temp=np.multiply.reduceat(1+np.random.random(np.sum(t)),t_cumul_plus)
    u=np.full(n,1.0)
    u[t_plus]=temp
    return u


n=10**7
t=np.random.poisson(0.8,n)
k=10
print(t[:k])

start=perf_counter()

u=solve(t)

end=perf_counter()
delta=end - start

print(*("%.2f" %z for z in u[:k]))
print(f"Durée : {delta :.2f}s")
```

qui affiche

```
[1 0 0 0 1 1 1 1 1 1]
1.66 1.00 1.00 1.00 1.54 1.93 1.19 1.04 1.65 1.55
Durée : 0.57s
```

On voit que le temps a été divisé plus de 45. Le code ne contient aucune boucle `for` explicite ni aucune liste Python, il n'utilise que des fonctions built-in de Numpy.

Vectorisation avec Numba

Numba offre la possibilité de vectoriser des tableaux Numpy en décorant par le décorateur `vectorize` la fonction qui réalise le traitement.

Reprenons le problème traité au paragraphe précédent : on part d'une tableau `t` d'entiers positifs, par exemple,

```
t = np.array([2, 4, 0, 3, 0, 1, 4])
```

et on veut créer un tableau `u` de flottants, de même longueur que `t` et tel que l'élément à l'indice `i` soit obtenu de la manière suivante :

- si t[i] vaut 0 alors `u[i] = 1.0`
- sinon, et si on pose $k = t[i] > 0$ alors on génère `k` flottants aléatoires entre 0 et 1, on en fait le produit `p` et on pose `u[i] = p`.

La fonction `f` qui agit sur chaque élément de `t` pour transformer `t` en `u` est la suivante :

```
def f(k):
    if k:
        return np.prod(np.random.random(k))
    return 1.0
```

C'est maintenant que Numba peut intervenir. Pour que `f` devienne une fonction de vectorisation efficace, il suffit de lui placer un décorateur :

```
1 import numpy as np
2 from numba import vectorize, int64, float64
3
4 @vectorize([float64(int64)])
5 def f(k):
6     if k:
7         return np.prod(np.random.random(k))
8     return 1.0
```

Le décorateur doit être importé de Numba (ligne 2) et sa présence va entraîner la compilation de la fonction en mode nopython. La syntaxe `[float64(int64)]` de la ligne 4 indique que `f` agit sur des entiers de type `int64` d'un tableau uni-dimensionnel et qui renvoie un nombre flottant de type float64 ; ces types ont été importés de Numba à ma ligne 2.

Pour vectoriser le tableau `t` en `u`, il suffit de faire un appel `u = f(t)`. D'où le code :

```
from time import perf_counter
import numpy as np
from numba import vectorize, int64, float64

@vectorize([float64(int64)])
def f(k):
    if k:
        return np.prod(np.random.random(k))
    return 1.0

lam=0.8
```

```
k=10

# warmup
n=2
f(np.random.poisson(lam, n))

n=10**7
t=np.random.poisson(lam,n)
print(t[:k])

start=perf_counter()

u=f(t)

end=perf_counter()
delta=end - start

print(*("%.2f" %z for z in u[:k]))
print(f"Durée : {delta :.2f}s")
```

et qui affiche

```
[0 1 0 1 2 1 1 1 1 0]
1.00 0.79 1.00 0.63 0.01 0.85 0.40 0.57 0.73 1.00
Durée : 0.55s
```

La performance est très correcte et totalement comparable au code vectorisé de Numpy tout en étant beaucoup plus simple car trouver le code de Numpy nécessite un certain savoir-faire et est ici assez détourné de l'approche directe.

On peut même l'améliorer en utilisant la fonction `np.rand` qui est équivalente à `np.random.random` :

```
from time import perf_counter
import numpy as np
from numba import vectorize, int64, float64

@vectorize([float64(int64)])
def g(k):
    p=1.
    rand=np.random.rand
    for i in range(k):
        p*=rand()
    return p

lam=0.8
k=10

# warmup
n=2
g(np.random.poisson(lam, n))
```

```
n=10**7
t=np.random.poisson(lam,n)
print(t[:k])

start=perf_counter()

u=g(t)

end=perf_counter()
delta=end - start

print(*("%.2f" %z for z in u[:k]))
print(f"Durée : {delta :.2f}s")
```

et qui affiche

```
[1 1 0 0 1 1 1 0 1 0]
0.64 0.72 1.00 1.00 0.78 0.23 0.39 1.00 0.61 1.00
Durée : 0.19s
```

Par rapport au code utilisant la vectorisation Numpy définie par la méthode `f.vectorize`, on voit que le temps d'exécution a été divisé par 100 et sans rajouter aucune difficulté de conception.

Et naturellement c'est beaucoup plus simple que de créer une extension C de Numpy en utilisant PyUFuncType.

Il est possible de donner une instruction de parallélisation :

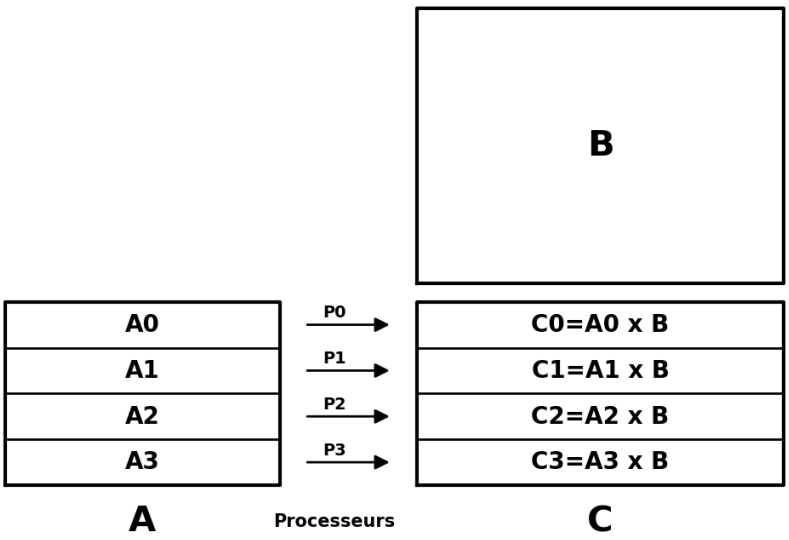
```
@vectorize([float64(int64)], target='parallel')
def f(k):
    if k:
        return np.prod(np.random.random(k))
    return 1.0
```

mais à l'exécution, ce code s'est avéré plus lent.

Parallélisme sur CPU

Numba permet la parallélisation automatique de l'exécution de certaines fonctions sur le CPU. Mais comme cette possibilité semble délicate à paramétrer, je vais me limiter à l'utilisation du parallélisme dans des boucles `for` avec `prange` et je ne vais envisager que des cas simples de parallélisme de données, ce qu'on qualifie parfois d'embarrassingly parallell.

Le cas typique est celui du produit matriciel. Supposons que l'on veuille effectuer le produit $A \times B =: C$ de deux matrices de tailles respectives (n, p) et (p, q). Supposons que l'on dispose par exemple d'un processeur ayant 4 cœurs, qu'on assimilera à des processeurs nommés P0, P1, P2 et P3 :



Il est clair qu'on peut effectuer le calcul de `C` en parallèle :

— on découpe la matrice `A` en 4 sous-matrices ayant `p` colonnes et approximativement `n//4` lignes ;
— chaque processeur `Pi` s'occupe de faire le produit matriciel `Ai` $\times$ `B` $=:$ `Ci` ;
— la matrice `C` est formée de l'empilement des blocs `Ci`.

Rappelons un code possible du produit matriciel de deux matrices à coefficients de type `int64` :

```
1  def product(A, B):
2      n, p=A.shape
3      p, q=B.shape
4
5      C=np.zeros((n,q), dtype=np.int64)
6      for i in range(n):
7          for j in range(q):
8              for k in range(p):
9                  C[i,j]+=A[i,k]*B[k,j]
10     return C
```

Sur un multi-cœur, et d'après la méthode expliquée si dessus, le parallélisme se fera ligne 6 sur la boucle portant sur les lignes d'indices `i` de la matrice `A`. En Numba, le parallélisme se traduira par la fonction décorée ci-dessous :

```
1  import numpy as np
2  from numba import njit, prange
3
4  @njit(parallel=True)
5  def product(A, B):
6      n, p=A.shape
7      p, q=B.shape
8
9      C=np.zeros((n,q), dtype=np.int64)
```

```
10      for i in prange(n):
11          for j in range(q):
12              for k in range(p):
13                  C[i,j]+=A[i,k]*B[k,j]
14      return C
```

Il y a deux modifications :

— ligne 4 : le décorateur `njit` qui assure la compilation en mode nopython prend un argument `parallel=True` ce qui signifie que la fonction va être exécutée (au moins en partie) en parallèle ;

— ligne 10 : on indique que la boucle sur `i` doit être exécutée en parallèle, d'où le `prange(n)` au lieu de `range(n)`.

Dans le code ci-dessous, on compare les temps d'exécution par Numba, sur processeur Intel i5 ayant 4 cœurs, d'un produit matriciel en série et d'un produit matriciel en parallèle :

```
from time import perf_counter
import numpy as np
from numba import njit, prange
from timeit import timeit

@njit(parallel=True)
def product(A, B):
    n, p=A.shape
    p, q=B.shape

    C=np.zeros((n,q), dtype=np.int64)
    for i in prange(n):
        for j in range(q):
            for k in range(p):
                C[i,j]+=A[i,k]*B[k,j]
    return C

@njit
def product_serial(A, B):
    n, p=A.shape
    p, q=B.shape

    C=np.zeros((n,q), dtype=np.int64)
    for i in range(n):
        for j in range(q):
            for k in range(p):
                C[i,j]+=A[i,k]*B[k,j]
    return C

n=1000
A = np.random.randint(low=-1, high=2, size=(n, n), dtype=np.int64)
B = np.random.randint(low=-1, high=2, size=(n, n), dtype=np.int64)
```

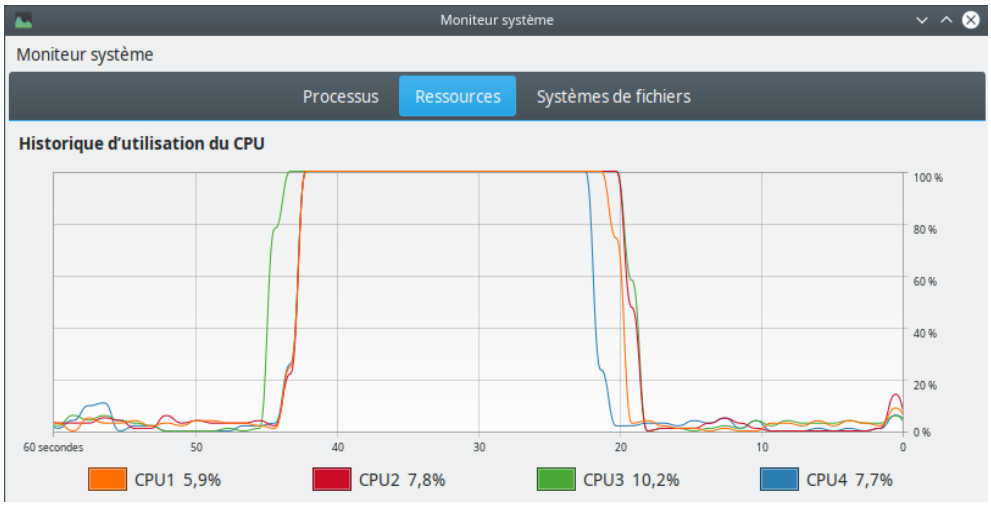
```
# -------------- Numba parallel -------------------------
# warmup
I2=np.matrix([[1, 0], [0, 1]], dtype=np.int64)
product(I2, I2)

begin_perf = perf_counter()

C=product(A, B)

print("Numba parallel : %.2fs\n" %(perf_counter()-begin_perf))

# -------------- Numba serial -------------------------
# warmup
I2=np.matrix([[1, 0], [0, 1]], dtype=np.int64)
product_serial(I2, I2)

number=12
print("Numba serial : %.2fs\n" %(timeit("product_serial(A, B)",
      globals=globals(), number=number)/number))

D=product_serial(A, B)

# ---------------------------------------

# check
print(np.array_equal(C, A @ B) and np.array_equal(C, D))
```

qui affiche

```
Numba parallel : 0.34s

Numba serial : 1.33s

True
```

Comme les temps d'exécution du produit non parallèle étaient assez fluctuants, j'ai utilisé le module `timeit` pour avoir une moyenne. Par ailleurs, on observe que la version parallèle à une durée d'exécution environ 4 fois plus courte que la version normale.

Le moniteur-système sous Linux permet de visualiser l'activité des cœurs. Voici ce qu'on observe pour une exécution en parallèle pour `n=3200` :


Moniteur système
Moniteur système
Processus
Ressources
Systèmes de fichiers
Historique d'utilisation du CPU
100 %
80 %
60 %
40 %
20 %
0 %
60 secondes
50
40
30
20
10
0
CPU1 5,9%
CPU2 7,8%
CPU3 10,2%
CPU4 7,7%

Chapitre III

Structures de données Numba

Les listes spécifiques de Numba

Numba travaille aisément et efficacement avec des tableaux Numpy. Mais ces derniers, sont, par construction, statiques, leur taille est allouée une bonne fois pour toutes et le tableau n'est pas extensible. Pour les rendre plus dynamiques, Numpy dispose de deux méthodes mais elles sont peu efficaces :

— la méthode `append` mais elle récrit un nouveau tableau,

— la méthode `resize` mais elle est écrite en pur Python.

En outre les tableaux Numpy ont le désavantage qu'ils sont formés de sous-tableaux ayant tous la même taille, autrement dit, pas de ragged array qui soit pris en charge sous Numpy.

Depuis sa version 0.45 (juillet 2019), Numba supporte des listes mono-types, extensibles et ayant la même interface que celles de Python. En particulier, les listes emboîtées sont prises en charge ainsi que la méthode `append` ce qui rend ces listes extensibles. Bien sûr, ces listes sont utilisables en mode nopython. Ces listes sont définies dans une classe `List`.

Encore en version `0.54 dev` (mai 2021), ces listes sont considérées comme expérimentales. Pour l'instant, ces listes sont loin d'avoir l'efficacité des tableaux Numpy ou des *vectors* du C++. Les performances sont variables, mais dans de nombreuses situations, elles sont plus proches de celles de Python que de celles d'un langage compilé. Toutefois, les performances se sont améliorées depuis la version `0.50` de Numba.

D'après mes expérimentations,

— les listes Numba sont performantes en tant que listes extensibles : la méthode `append` est efficace

— lorsque la liste doit se comporter **aussi** comme un vrai tableau, avec accès direct à un item en lecture ou écriture (l'équivalent d'un `getitem` ou `setitem`), les performances sont très dégradées, et comparables à celles de Python pur.

Concernant l'absence de *ragged array* sous Numpy, on notera que le module Awkward Array semble les prendre en charge. Voir en fin de cette page.

Les listes Numba sont implémentées dans trois fichiers :

— containers

— List object in Python

— List object in C

Il y est clairement indiqué que les listes Numba sont, à divers titres, largement inspirées de l'implémentation des listes de CPython.

Listes Numba : exemple basique

Les listes de Numba font partie de la classe `List`, accessible depuis le module `typedlist`. On notera le **L** en capitale de `List`. L'interface est quasiment la même que pour les listes Python. Voici un exemple de quelques utilisations :

```
1  >>> from numba.typed import List
2  >>> L = List([2024, 2038])
3  >>> L
4  ListType[int64]([2024, 2038])
5  >>> type(L)
6  <class 'numba.typed.typedlist.List'>
7  >>> L.append(3000)
8  >>> L
9  ListType[int64]([2024, 2038, 3000])
10 >>> M = List([42, 99])
11 >>> L.extend(M)
12 >>> L[-2]
13 42
14 >>> L[1]=2025
15 >>> L
16 ListType[int64]([2024, 2025, 3000, 42, 99])
17 >>> L.append(3.14)
18 Traceback (most recent call last):
19 ...
20 numba.core.errors.TypingError: Failed in nopython mode pipeline (step: nopython fron
```

— Ligne 4 : On aura noté que, par inférence de type, `L` est considérée comme une liste d'entiers sur 64 bits.

— Ligne 6 : on n'obtient qu'un type générique

— Ligne 17 : il faut respecter le type des éléments de la liste.

Listes de listes

C'est semblable aux listes 2D :

```
1 >>> from numba.typed import List
2 >>> A=List([42, 99])
3 >>> B=List([2, 3, 5])
4 >>> L=List([A, B])
5 >>> L
6 ListType[ListType[int64]]([[42, 99], [2, 3, 5]])
7 >>> L.append(A)
8 >>> L
9 ListType[ListType[int64]]([[42, 99], [2, 3, 5], [42, 99]])
```

```
10 >>> L[0][0]=100
11 >>> L
12 ListType[ListType[int64]]([[100, 99], [2, 3, 5], [100, 99]])
13 >>>
```

— Ligne 4 : création d'une liste de listes

Bien sûr, et c'est tout l'intérêt, les listes Numba fonctionnent en mode nopython :

```
from numba.typed import List
from numba import njit

@njit
def f():
    L=List([List([42]*5)])
    L.append(List([2030]*4))
    print(L)

f()
```

qui affiche

```
[[42, 42, 42, 42, 42], [2030, 2030, 2030, 2030]]
```

Créer un type de liste

On peut avoir besoin d'accéder au type d'une liste. La fonction standard `type` ne donne qu'un type générique :

```
from numba.typed import List

L=List([42, 25])
print(type(L))
print(repr(L))
```

qui affiche

```
<class 'numba.typed.typedlist.List'>
ListType[int64]([42, 25])
```

Pour générer un type de liste, on utilise `numba.types`. Voici un exemple d'utilisation :

```
1 from numba.typed import List
2 from numba.types import ListType
3 from numba import int64
4
5 my_type1=ListType(int64)
6 print(my_type1)
7
8 my_type2=ListType(my_type1)
9 print(my_type2)
```

```
ListType[int64]
ListType[ListType[int64]]
```

— Lignes 5 et 10 : on créé le type `List` d'entiers de type `int64`

— Lignes 8 et 11 : on créé le type `List` de List d'entiers de type `int64`

Savoir définir un type de liste est utile quand on a besoin d'une liste vide d'un certain type.

Par ailleurs, partant cette fois d'une liste donnée, je n'ai pas trouvé dans la doc comment accéder à l'objet type de cette liste.

Listes vides

Comme les listes de Numba sont typées en mode nopython, le code suivant renvoie une erreur :

```
1 from numba.typed import List
2 from numba import njit
3
4 @njit
5 def f(L):
6     return len(L)
7
8 L=List([])
9 f(L)
```

```
10 Traceback (most recent call last):
11 [... omis ...]
12 TypeError: invalid operation on untyped list
```

— Ligne 8 : La liste vide non typée est bien définie en mode object

— Ligne 9 : La liste vide non typée n'est pas utilisable en mode nopython

Il existe donc une méthode de la classe `List` qui permet de créer une liste vide et de type donné. Il s'agit de la méthode `empty_list`. Exemple d'utilisation :

```
1 from numba.typed import List
2 from numba import njit, int64
3
4 @njit
5 def f(L):
6     return len(L)
7
8 L=List.empty_list(int64)
9 print(f(L))
```

```
10 0
```

— Ligne 8 : on crée une liste Numba d'entiers 64 bits (importé ligne 2) mais vide.

— Lignes 4 et 9 : elle fonctionne en mode nopython

— Ligne 10 : la liste est bien vide.

Les listes vides sont utiles car on peut avoir à en insérer une dans une liste déjà existante. Voici un exemple d'utilisation :

```
1  from numba.typed import List
2  from numba import njit, int64, types
3
4  A=List([42, 25])
5  B=List.empty_list(int64)
6  L1=List([A, B])
7
8  print("L1 =", L1)
9
10 C=List([55, 66, 77])
11 D=List([33])
12 L2=List([C, D])
13
14 print("L2 =", L2)
15
16 L=List([L1, L2])
17
18 print("L = [L1, L2] =", L)
19
20 empty2d=List.empty_list(types.ListType(int64))
21 L.append(empty2d)
22
23 print("L =", L)
```

```
24 L1 = [[42, 25], []]
25 L2 = [[55, 66, 77], [33]]
26 L = [L1, L2] = [[[42, 25], []], [[55, 66, 77], [33]]]
27 L = [[[42, 25], []], [[55, 66, 77], [33]], []]
```

— Ligne 6 : on crée une liste Numba `L1` de listes Numba d'entiers `int64` dont une, `B`, est vide
— Ligne 12 : on crée de même une liste Numba `L2` de listes Numba d'entiers `int64`
— Ligne 16 : on rassemble `L1` et `L2` dans une liste 3d appelé `L`
— Ligne 21 : on rajoute à `L` la liste vide du bon type.

Liste de tableaux Numpy

On peut créer une liste de tableaux Numpy, pas forcément de même taille, par exemple (ligne 13 ci-dessous) :

```
import numpy as np
from numba.typed import List
from numba import njit

@njit
def f(L):
    s=0
    for i in range(len(L)):
        s+=L[i][0]
    return s
```

```
LL=[[42,81], [42,81, 51, 29], [4]]
L=List(map(np.array, LL))

print(f(L))
```

qui affiche

```
88
```

Performance des listes Numba : la méthode `append`

Le programme ci-dessous détermine la liste des `N` premiers nombres premiers (l'algorithme implémenté est mathématiquement peu efficace mais ce n'est pas le propos ici). On va comparer les performances d'une fonction utilisant une liste Python et d'une fonction utilisant une liste Numba :

```
from time import  perf_counter
import numpy as np
from numba import njit, int64
from numba.typed import List

def primes(N):
    P=[]
    n = 2
    while len(P) < N:
        for p in P:
            if n % p == 0:
                break
        else:
            P.append(n)
        n += 1

    return P

@njit
def primes_nb(N):
    P=List.empty_list(int64)
    n = 2
    while len(P) < N:
        for p in P:
            if n % p == 0:
                break
        else:
            P.append(n)
        n += 1

    return P

N=40000
```

```
print("Testing with Numba List ...")

# Numba compilation warmup
primes_nb(2)

begin_perf=perf_counter()

PP=primes_nb(N)

nb_perf=perf_counter()-begin_perf
print("Numba List: %.2fs\n" %nb_perf)


print("Testing with Python list ...")
begin_perf=perf_counter()

P=(primes(N))

py_perf=perf_counter()-begin_perf
print("Python list: %.2fs" %py_perf)


ratio=py_perf/nb_perf
print("%.2f" %ratio)

print(P==[int(z) for z in PP])
```

qui affiche

```
Testing with Numba List ...
Numba List: 4.83s

Testing with Python list ...
Python list: 33.94s
7.03
True
```

L'amélioration semble seulement substantielle. En fait, elle est remarquable, car exactement le même programme écrit avec des vectors du C++ est moins rapide, sur ma machine, il met 6.3s.

Performance des listes Numba : le tri par sélection

Le programme ci-dessous implémente un tri par sélection et l'applique à une liste de 20000 entiers en comparant les performances des listes Numba par rapport aux listes Python :

```
from time import perf_counter
from random import randrange
from numba import njit, __version__
from numba.typed import List
```

```
def selection(L):
    n=len(L)
    for i in range(n-1):
        jMin=i
        mini=L[i]
        for j in range(i+1, n):
            if L[j] < mini:
                jMin=j
                mini=L[j]
        if mini!=L[i]:
            L[i], L[jMin]= L[jMin], L[i]

print("Numba version :", __version__, end='\n\n')

N=20000
# ------------ Data Python -----------------------

begin_perf = perf_counter()

L=[randrange(10**6) for _ in range(N)]

delta = perf_counter() - begin_perf

print(f"Generation : {delta:.2f}s")


# ------------Numba copy -----------------------

begin_perf = perf_counter()

LL=List(L)

delta = perf_counter() - begin_perf

print(f"Numba Copy : {delta:.2f}s")



# ------------ Python -----------------------
print("Sorting ...")

begin_perf = perf_counter()

selection(L)

delta_py = perf_counter() - begin_perf

print(f"Python list: {delta_py:.2f}s")
```

```
# ------------ Numba + List ----------------------
selection_nb_List=njit(selection)

# warmup
selection_nb_List(List([42]))

print("Sorting ...")

begin_perf = perf_counter()

selection_nb_List(LL)

delta_nb_List = perf_counter() - begin_perf

print(f"Numba List: {delta_nb_List:.2f}s")

# ----------------------------------

print(f"ratio : {delta_py/delta_nb_List:.1f}")

print(L==list(LL))
```

```
Numba version : 0.53.1

Generation : 0.01s
Numba Copy : 0.31s
Sorting ...
Python list: 8.08s
Sorting ...
Numba List: 2.91s
ratio : 2.8
True
```

On voit que l'amélioration apportée par les listes Numba est modeste, d'un facteur $2,8$ mais il s'est amélioré depuis la version 0.50 de Numba où le ratio était de $1,3$. On pourra noter que le coût de la transformation de la liste Python en liste Numba n'a pas été pris en compte.

En comparaison, on avait vu que Numba traitait la même liste mais récrite en tableau Numpy avec une performance améliorée d'un facteur 36.

Performance des listes Numba : listes d'adjacence

Ce qui suit ne nécessite aucune connaissance en théorie des graphes.

On donne un entier n et une liste de m arcs d'un graphe orienté de sommets $0, 1, \ldots, n-1$. On veut créer la liste d'adjacence de ce graphe, c'est-à-dire la liste des listes des voisins de chaque sommets.

Par exemple, si $n = 4$ et $m = 9$, et si la liste des 9 arcs est :

```
(1, 2)
(1, 0)
(3, 1)
(1, 4)
(2, 1)
(2, 3)
(4, 3)
(0, 3)
(4, 1)
```

la liste d'adjacence est vue comme :

```
0: 3
1: 2 0 4
2: 1 3
3: 1
4: 3 1
```

ou encore `L=[[3], [2, 0, 4], [1, 3], [1], [3, 1]]`.

Pour effectuer les tests qui suivent, on génère une liste aléatoire d'arcs dans un fichier texte. Pour le graphe ci-dessus, le fichier serait :

```
5
9
1 2
1 0
3 1
1 4
2 1
2 3
4 3
0 3
4 1
```

Syntaxe du fichier :

— Première ligne : la valeur de `n`
— Deuxième ligne : la valeur de `m`
— Les `m` lignes suivantes : un arc, formé de deux entiers, l'origine et l'extrémité de l'arc.

Pour le test, un fichier pour les valeurs $n = 10000$ et $m = 6000000$ sera utilisé.

L'objectif est donc de faire la transformation suivante :

— objet initial : une liste de listes de deux entiers
— objet final : une liste de `n` listes de longueurs variables.

Voici le code et le résultat de son exécution :

```
from time import perf_counter
import numpy as np
from numba.typed import List
from numba import njit, int64, types, __version__
```

```
def edges2adj(edges, n):
    adj=[[] for _ in range(n)]
    for u,v in edges:
        adj[u].append(v)
    return adj

LIST_int64=types.ListType(int64)

@njit
def edges2adj_nb(edges, n):
    adj=List.empty_list(LIST_int64)
    for i in range(n):
        adj.append(List.empty_list(int64))

    for u,v in edges:
        adj[u].append(v)
    return adj

def capture():
    n=int(input())
    m=int(input())
    return n, [tuple(map(int, input().split())) for _ in range(m)]

print("Numba version : %s" %__version__)

print("Building dataset...")

begin_perf = perf_counter()

n, edges=capture()

delta = perf_counter() - begin_perf

print(f"Dataset: {delta:.2f}s\n")

# --------- Python ----------------


print("Python building adj ...")

begin_perf = perf_counter()

adj=edges2adj(edges, n)

delta_py = perf_counter() - begin_perf

print(f"Python : {delta_py:.2f}s\n")
```

```
# --------- Numba ----------------
# warmup
edges2adj_nb(List({(0, 1), (1, 0)}), 2)

print("Numba building List ...")

begin_perf = perf_counter()

edges_nb=List(edges)

delta = perf_counter() - begin_perf

print(f"List : {delta:.2f}s\n")


print("Numba building adj ...")

begin_perf = perf_counter()

adj_nb=edges2adj_nb(edges_nb, n)

delta_nb = perf_counter() - begin_perf

print(f"Numba : {delta_nb:.2f}s")

print(f"ratio : {delta_py/delta_nb:.1f}")
```

```
Numba version : 0.53.1
Building dataset...
Dataset: 10.34s

Python building adj ...
Python : 0.95s

Numba building List ...
List : 8.45s

Numba building adj ...
Numba : 0.42s
ratio : 2.2
```

Pour donner une idée, en C++, sur la même machine, l'opération de conversion dure 0.13s.
Le code précédent est lancé avec la commande suivante :

```
$ python3 adj_from_file.py < test_10000_600000.txt
```

où `test_10000_600000.txt` est le nom du fichier de test.
Le fichier des arêtes est généré par le code Python suivant :

```
from random import randrange

def maketest(n, m):
    edges=set([])
    while len(edges)<m:
        u=randrange(n)
        v=randrange(n)
        if u!=v:
            edges.add((u, v))
    return edges

n=10000
m=6000000

edges=maketest(n, m)
print(n)
print(m)
for u,v in edges:
    print(u, v)
```

puis par la ligne de commande :

```
python3 maketest.py > test_10000_600000.txt
```

Cet exemple montre que les listes de Numba n'apportent qu'une modeste amélioration des performances par rapport à leur équivalent Python (ratio de 2,2) mais en amélioration depuis la version 0.50 de Numba (ratio de 1,5). Sans compter le coût très important de la conversion.

Dans le cas présent d'une construction d'une liste d'adjacence, Numba pourrait obtenir d'excellentes performances en plaçant le **contenu** de chaque liste d'adjacence dans un tableau Numpy `T` d'entiers et que l'on couplerait à une 2e tableau d'offsets de `n` entiers qui indiquerait le début de chaque liste d'adjacence dans la liste `T`. Mais avec l'inconvénient majeur que les listes d'adjacence seraient statiques.

Performances des dictionnaires Python

CPython utilise en interne et pour l'interpréteur lui-même une table de hachage. Quand vous utilisez un dictionnaire dans un code Python, c'est en fait cette table de hachage qui est appelée.

Pour pouvoir comparer les dictionnaires Python aux dictionnaires de Numba, se pose la question des performances de cette table de hachage.

L'analyse des performances est toujours très délicate, de nombreux facteurs peuvent entrer en jeu comme le choix de la fonction de hachage, la gestion de la mémoire et les fonctionnalités offertes. Une même table peut présenter des performances variables selon le type d'épreuve qu'elle va effectuer (insertion et le type d'insertion, suppression de clés, lecture, écriture, parcours de la table, nature et taille des clés). En plus de cela, de nombreux benchmarks présentent des épreuves très simplifiées.

Les benchmarks de dictionnaires Python utilisés depuis l'API C de Python sont assez rares. En voici deux assez anciens :

— incise.org de 2010, qui n'est plus accessible en ligne et web-archivé,

— preshing de 2011.

Les résultats ne sont pas facilement interprétables. Mais grosso modo, il semble ressortir que, du point de vue de la vitesse d'exécution, les résultats des dictionnaires Python sont acceptables voire corrects mais sans être excellents.

En juillet 2018, J. P. Hansson a proposé des benchmarks assez variés. Sur les graphiques, il semble apparaître que :

— la table de hachage de Python et de la STL du C++ ont des performances comparables mais celles de Python sont peut-être un peu en retrait (vitesse et consommation mémoire) ;

— les performances de la table de Python sont nettement dans la 2e moitié du classement.

Par ailleurs, de nombreux benchmarks, par exemple

— Attractive Chaos en septembre 2018

— Martin Ankerl en avril 2019

semblent montrer que la table de hachage de la STL du C++ a de très mauvaises performances, et cela s'explique assez bien par son implémentation par chaînage (imposée par le standard ISO).

De tout cela, il semble ressortir que la table de hachage de CPython, bien qu'implémentant la technique efficace de l'adressage ouvert à sondage linéaire (*linear probing*) et bien qu'optimisée, a des performances plutôt moyennes par rapport à d'autres tables de hachage généralistes, implémentées en C ou C++ et largement diffusées.

Noter que les deux fichiers sources de l'implémentation de Numba pour son dictionnaire dictobject.h et dictobject.c reprennent intégralement et adaptent l'implémentation de la table de hachage de CPython.

Je comparerai ici les performances des dictionnaires Numba à son équivalent dans la STL du C++, à savoir `unordered_map`.

Les dictionnaires de Numba

Depuis sa version 0.44, Numba met à disposition un dictionnaire en mode nopython. L'usage est très proche de celui des dictionnaires Python. Ci-dessous, je présente des exemples simples et je fais des micro-benchmarks pour comparer avec les dictionnaires de Python.

Construction d'un dictionnaire par insertions

On génère une liste de 3 millions d'entiers aléatoires et on les insère dans un dictionnaire :

```
from time import perf_counter
from random import shuffle
from numba.typed import Dict, List
from numba import njit, __version__


def f(L):
    d={}
    for k in L:
        d[k]=k
    return d
```

```
@njit
def g(L):
    d=Dict()
    for k in L:
        d[k]=k
    return d


print("Numba version : %s" %__version__)

n=3*10**6

begin_perf = perf_counter()

L=list(range(n))
shuffle(L)

delta_py = perf_counter() - begin_perf

print(f"shuffle : {delta_py:.2f}s")

# ------------- Python -------------

begin_perf = perf_counter()

f(L)

delta_py = perf_counter() - begin_perf

print(f"Python : {delta_py:.2f}s")


# ------------- Numba -------------

g(List([5]))

L=List(L)

begin_perf = perf_counter()

g(L)

delta_nb= perf_counter() - begin_perf

print(f"Numba : {delta_nb:.2f}s")

print(f"ratio : {delta_py/delta_nb:.1f}")
```

qui affiche

```
Numba version : 0.53.1
shuffle : 1.97s
Python : 0.72s
Numba : 0.47s
ratio : 1.5
```

Ici, Numba fait mieux d'un facteur de l'ordre 1,5. Le même code écrit en C++ avec la table de hachage de la STL met 0.63s (voire plus si on compte la destruction de l'objet).

Parcours des termes d'un dictionnaire

On crée un dictionnaire d'un million d'entrées et on mesure le temps nécessaire à faire la somme des valeurs associées aux clés.

```
from time import perf_counter
from random import shuffle
import numpy as np
from numba import njit
from numba.typed import Dict, List

def f(d):
    n=len(d)
    s=0
    for k in d:
        s+=d[k]
    return s

@njit
def g(d):
    n=len(d)
    s=0
    for k in d:
        s+=d[k]
    return s

n=10**6


print("Generating data ...")
begin_perf = perf_counter()

L=list(range(n+1))
shuffle(L)

D=Dict()
for k in L:
    D[k]=k


d=dict(D)
```

```
37
38 delta_py = perf_counter() - begin_perf
39
40 print(f"Data: {delta_py:.2f}s")
41
42 # ------------- Python -------------
43
44 begin_perf = perf_counter()
45
46 s1=f(L)
47
48 delta_py = perf_counter() - begin_perf
49
50 print(f"Python : {delta_py:.2f}s")
51
52
53 # ------------- Numba -------------
54
55 DD=Dict()
56 DD[5]=5
57 g(DD)
58
59 begin_perf = perf_counter()
60
61 s2=g(D)
62
63 delta_nb = perf_counter() - begin_perf
64
65 print(f"Numba : {delta_nb:.2f}s")
66
67 # check
68 print(s1 == s2)
69
70 from numba import __version__
71 print("Numba version :", __version__)
72 print(f"ratio : {delta_py/delta_nb:.1f}")
```

```
73 Generating data ...
74 Data: 4.88s
75 Python : 0.23s
76 Numba : 0.03s
77 True
78 Numba version : 0.53.1
79 ratio : 7.0
```

Dans ce code, on n'évalue que le parcours de dictionnaire, pas la création puisque les lignes 31-36 ne sont pas chronométrées. Ici, Numba se montre vraiment très efficace, améliorant le temps d'exécution d'un facteur de 7. Le même code écrit en C++ avec `std::unordered_map` met `0.14s`.

Dictionnaire de dictionnaires

Il est possible de définir des dictionnaires de dictionnaires de type spécifique à Numba. Ci-dessous, voici un code qui marche en mode objet mais pas en mode nopython.

```
from numba import njit, int64, types
from numba.typed import Dict

@njit
def f():
    Dict.empty(int64, int64)
    print("f is OK")

def g():
    Dict.empty(int64, types.DictType(int64, int64))
    print("g is OK")

@njit
def h():
    Dict.empty(int64, types.DictType(int64, int64))
    print("h is OK")

f()
g()
h()
```

Il renvoie un long message d'erreur dont voici un extrait :

```
$ python dict_dict.py
f is OK
g is OK
Traceback (most recent call last):

... omis ...


There are 2 candidate implementations:

... omis ...

def h():
    Dict.empty(int64, types.DictType(int64, int64))
    ^
```

On dirait que le type des valeurs du dictionnaire doit être connu à la **compilation**. On peut y pallier avec les codes suivants mais on y perd en souplesse.

Produit de matrices avec des dictionnaires de dictionnaires

On va effectuer et évaluer les performances d'un produit matriciel. Mais au lieu qu'une matrice soit représentée comme un tableau Numpy, elle sera représentée par un dictionnaire de diction-

naires. Par exemple, si `M` est une matrice carrée d'ordre 10 alors les lignes de la matrice seront les dictionnaires `M[0]`, ..., `M[9]` et, par exemple, l'élément de `M` aux indices `(ligne, colonne) = (3, 5)` vaudra `M[3][5]`.

Plus précisément, on va se contenter de calculer le carré d'une matrice. Voici le code correspondant :

```
from time import perf_counter
from random import randrange
from numba import njit, int64, types
import numpy as np
from numba.typed import Dict, List


def product_direct(A, B):
    n, p=len(A), len(A[0])
    q=len(B[0])

    P=[[0]*q for _ in range(n)]
    for i in range(n):
        for j in range(q):
            for k in range(p):
                P[i][j]+=A[i][k]*B[k][j]
    return P

def square_py(d):
    n=len(d)
    P={i:{j:0 for j in range(n)} for i in range(n)}
    for i in range(n):
        for j in range(n):
            for k in range(n):
                P[i][j]+=d[i][k]*d[k][j]
    return P


INT64_TO_INT64=types.DictType(int64, int64)

@njit
def square_nb(D):
    n=len(D)
    P=Dict.empty(int64,INT64_TO_INT64)
    for i in range(n):
        P[i]=Dict.empty(int64, int64)
        for j in range(n):
            P[i][j]=0
    for i in range(n):
        for j in range(n):
            for k in range(n):
                P[i][j]+=D[i][k]*D[k][j]
    return P
```

```
44
45
46
47 def build_matrices(n):
48     M=[[randrange(2) for _ in range(n)] for _ in range(n)]
49     d={}
50     D=Dict.empty(int64, types.DictType(int64, int64))
51     for i in range(n):
52         d[i]={}
53         D[i]=Dict.empty(int64,int64)
54         for j in range(n):
55             d[i][j]=D[i][j]=M[i][j]
56     return d, D, M
57
58
59 n=400
60 d, D, A=build_matrices(n)
61
62
63 begin_perf = perf_counter()
64
65 PP=product_direct(A, A)
66
67 delta = perf_counter() - begin_perf
68
69 print(f"Python list : {delta :.2f}s")
70
71 # ----------------------------------------
72
73 begin_perf = perf_counter()
74
75
76 P=square_py(d)
77
78 delta_py = perf_counter() - begin_perf
79
80 print(f"Python dict : {delta_py :.2f}s")
81
82 # ----------------------------------------
83 # warmup
84 DD=Dict.empty(int64, types.DictType(int64, int64))
85 dd=Dict.empty(int64,int64)
86 dd[0]=0
87 DD[0]=dd
88 square_nb(DD)
89
90
91 begin_perf = perf_counter()
92
```

```
93
94 PPP=square_nb(D)
95
96 delta_nb = perf_counter() - begin_perf
97
98 print(f"Numba Dict : {delta_nb :.2f}s")
99
100 # ----------------------------------------
101
102
103 print([P[i][j] for i in range(n) for j in range(n)]==
104       [PP[i][j] for i in range(n) for j in range(n)]==
105       [PPP[i][j] for i in range(n) for j in range(n)])
106
107 from numba import __version__
108 print("Numba version : %s" %__version__)
109
110 print(f"ratio : {delta_py/delta_nb:.1f}")
```

et qui affiche

```
1 Python list : 8.00s
2 Python dict : 15.08s
3 Numba Dict : 9.23s
4 True
5 Numba version : 0.53.1
6 ratio : 1.6
```

Commentons rapidement le code :

— Lignes 8 et 65 : même si c'est légèrement hors-sujet, je compare aussi avec un produit classique avec des listes de listes

— Ligne 19 : le carré matriciel est implémenté avec des dictionnaires Python

— Ligne 32 : le carré matriciel est implémenté avec des dictionnaires Numba.

— Lignes 29 et 34 : pour éviter d'avoir le problème de typage rencontré ci-dessus, on fait en sorte que le type des valeurs du dictionnaire soit connu au moment de la compilation de la fonction

Pour donner une idée, le même code écrit en C++ avec la structure `std::unordered_map` donne un temps de `3,38s` alors que l'implémentation de la STL de `unordered_map` est considérée comme très inefficace dans sa catégorie (mais peut-être moins si les instances sont de petite taille comme c'est le cas ici).

Donc, les performances de Numba sont en demi-teinte, n'apportant qu'une modeste amélioration par rapport au code Python pur et étant encore un peu moins de 3 fois plus lent que le code produit par un langage compilé (mais en amélioration depuis la version 0.50 pour laquelle le ratio était de `3,8`).

Construction de dictionnaire de dictionnaires

Evaluons les performances de la construction de dictionnaires de dictionnaires Numba.

```python
from time import perf_counter
from random import randrange
from numba import njit, int64, types
import numpy as np
from numba.typed import Dict


def build_py(M):
    n=len(M)
    return {i:{j:M[i][j] for j in range(n)} for i in range(n)}


INT64_TO_INT64=types.DictType(int64, int64)

@njit
def build_nb(M):
    n=len(M)
    D=Dict.empty(int64,INT64_TO_INT64)
    for i in range(n):
        D[i]=Dict.empty(int64, int64)
        for j in range(n):
            D[i][j]=M[i][j]
    return D


n=3000

begin_perf = perf_counter()

M=[[randrange(2) for _ in range(n)] for _ in range(n)]

delta = perf_counter() - begin_perf

print(f"Data Python : {delta :.2f}s")

begin_perf = perf_counter()

MM=np.array(M)



delta = perf_counter() - begin_perf

print(f"Data Numba: {delta :.2f}s")


# ----------------------------------------
```

```
begin_perf = perf_counter()

d=build_py(M)


delta_py = perf_counter() - begin_perf

print(f"Python dict : {delta_py :.2f}s")

# ----------------------------------------
# warmup
build_nb(np.array([[randrange(2) for _ in range(1)] for _ in range(1)]))


begin_perf = perf_counter()

D=build_nb(MM)

delta_nb = perf_counter() - begin_perf

print(f"Numba Dict : {delta_nb :.2f}s")

# ----------------------------------------

print(dict(D)==d)

from numba import __version__
print("Numba version : %s" %__version__)

print(f"ratio : {delta_py/delta_nb:.1f}")
```

qui affiche

```
Data Python : 5.91s
Data Numba: 0.50s
Python dict : 0.77s
Numba Dict : 0.56s
True
Numba version : 0.53.1
ratio : 1.4
```

Les performances sont meilleures que celles des dictionnaires Python et sont proches de celles du même code écrit en C++ avec `std:unordered_map` qui nécessite `0,58 s`. Le résultat s'est amélioré depuis la version `0,50` pour laquelle les dictionnaires Python étaient plus rapides que les dictionnaires Numba.

Prise en charge de `heapq`

Depuis Numba 0.43, une prise en charge de la structure de données `heapq` de Python est assurée en mode nopython (rappel : le module `heapq` implémente une file de priorité).

L'usage est similaire à l'usage en Python pur. Les éléments de la file doivent être tous du même type et il faut que Numba puisse inférer ce type.

Voici un exemple qui va comparer les performances de heapq en Numba et en Python pur. Le test consiste à remplir une file de priorité depuis une liste aléatoire (Numba ou Python) d'un million d'entiers par alternance des opérations suivantes :

— 20 insertions

— 5 supressions

Une fois le contenu de la liste complètement placé dans la file, on vide un par un les éléments de la file.

```
from time import perf_counter
from heapq import heappush, heappop
from random import shuffle
from numba import njit, int64
from numba.typed import List


def f(L):
    n=len(L)
    H=[]
    i=0
    while i<n:
        for j in range(10):
            heappush(H, L[i])
            heappush(H, L[i])
            i+=1
        for j in range(5):
            heappop(H)
    while H:
        heappop(H)

@njit
def g(L):
    n=len(L)
    H=[0]
    H.pop()
    i=0
    while i<n:
        for j in range(10):
            heappush(H, L[i])
            heappush(H, L[i])
            i+=1
        for j in range(5):
            heappop(H)
    while H:
        heappop(H)


```

```
L=list(range(10**6))
shuffle(L)

# -------------- Python --------------------

begin_perf = perf_counter()


f(L)

delta_py = perf_counter() - begin_perf

print(f"Python : {delta_py:.2f}s")

# -------------- Numba --------------------
L=List(L)


#warmup
g(List(range(20)))


begin_perf = perf_counter()


g(L)

delta_nb = perf_counter() - begin_perf

print(f"Numba : {delta_nb:.2f}s")

from numba import __version__
print("Numba version : %s" %__version__)

print(f"ratio : {delta_py/delta_nb:.1f}")
```

qui affiche

```
Python : 1.72s
Numba : 0.43s
Numba version : 0.53.1
ratio : 4.0
```

L'amélioration est nette et est en progrès par rapport à la version 0,50 où le ratio était de 3. En C++ avec la STL, un code équivalent s'exécute 0.25s.

Quelques commentaires de code :

— les codes des fonctions `f` et g sont très proches

— `f` reçoit une liste Python

— g reçoit une liste Numba

— Lignes 25-26 : la file est `H` car c'est sur elle que se font les opérations `heappush` (ligne 30 par exemple). Pour que Numba puisse inférer le type des éléments de la file, j'utilise une astuce : je place un élément, l'entier `0` qui est vu par Numba comme du type `int64` et je le retire aussitôt car la file doit être vide au départ.

Le module Awkward Array

Numpy ne prend pas en charge le type *ragged array* ni les tableaux extensibles. Le module Awkward Array semble prendre en charge le type *ragged array* et en outre, ce module est compatible avec Numba 0.50.

Après échange avec l'auteur, Jim Pivarski, il apparaît que seule la structure de données `ArrayBuilder` pourrait permettre d'implémenter des vecteurs extensibles (mais non rétractables). Par ailleurs, les strutures de données du module dont essentiellement immuables.

Quoi qu'il en soit, les tableaux Awkward array peuvent être utiles pour des traitements de données complexes. Voici un exemple exécuté sous Google Colab. Le code est extrait d'un fichier Jupyter présent dans le code source de Awkward Array.

On installe le module awkward :

```
!pip install awkward
```

```
Collecting awkward

...Affichage tronqué....

Successfully installed awkward-1.3.0
```

On met à jour Numba :

```
!pip install --upgrade numba
```

```
Collecting numba

...Affichage tronqué....

Successfully installed llvmlite-0.36.0 numba-0.53.1
```

On teste :

```
import numba as nb

@nb.jit(nopython=True)
def run(array):
    out = np.empty(len(array), np.float64)
    for i in range(len(array)):
        out[i] = array[i]["x"]
        for y in array[i]["y"]:
            out[i] += y
    return out
```

```
akarray = ak.Array([{"x": 100, "y": [1.1, 2.2]}, {"x": 200, "y": []},
                    {"x": 300, "y": [3.3]}])

# Works for the layout nodes, but not the high-level ak.Array wrapper yet.
print(run(akarray))
```

```
[103.3 200.  303.3]
```

Chapitre IV

Installation du GPU pour Numba

GPU sous Numba : que dois-je utiliser ?

Ce paragraphe donne juste des instructions rapides pour pouvoir utiliser un GPU avec Numba. Des informations sur la programmation GPU et sur Cuda suivront.

Numba permet d'utiliser un GPU (processeur graphique) pour effectuer des calculs massivement parallèles. Pour cela, Numba utilise l'environnement Cuda de Nvidia.

Solution recommandée

Le plus simple et qui fonctionne toujours est d'utiliser la plateforme en ligne Google Colaboratory qui met à disposition dans des feuilles Jupyter Notebook le package Numba et un GPU, sans aucune installation à réaliser et gratuitement.

Il existe peut-être d'autres solutions en ligne (Cloud Azure, AWS) mais probablement payantes ou avec des restrictions (étudiants).

Matériel sur votre machine

Si vous n'utilisez pas un service en ligne, il faut que

— votre machine (PC ou laptop) dispose d'une carte graphique dédiée (et pas seulement une carte graphique intégrée)

— cette carte graphique soit équipée d'un GPU Nvidia.

De nombreuses machines, même utilisées par des gamers, ne sont pas forcément pourvues de GPU Nvidia. A fortiori, des machines dédiées à un usage basique ont souvent une carte graphique intégrée Intel ou AMD.

Concernant Apple, suite à un conflit avec Nvidia, le support de Cuda s'arrête à la version 10.2. Comme indiqué dans la documentation officielle de Cuda :

CUDA 11.0 does not support macOS for developing and running CUDA applications.

De nombreux MacBook Pro ont une carte graphique Intel sans support de Cuda.

Si vous disposez d'un GPU Nvidia, vous devrez installer deux applications :

— les pilotes Nvidia les plus récents

— les bibliothèques dynamiques Cuda.

L'installation des pilotes sous Linux n'est pas automatique et peut être compliquée si vous devez mettre à jour le pilote. Concernant la mise à disposition de Cuda, le plus simple est d'utiliser la suite Anaconda.

Pour macOS, je n'ai pas eu accès à une machine ayant une carte graphique compatible Cuda. Mais si votre carte l'est, je vous conseille d'essayer d'installer Cuda sous Anaconda suivant la même procédure que sous Linux ou Windows.

Dans ce qui suit, sauf recours à un GPU sur Google Colab, les codes sont exécutés sur une carte graphique GeForce GTX 970.

Utiliser Numba avec un GPU sous Google Colab

Sous Google Colab, Numba peut directement exécuter du code sur les GPU, sans aucune installation préalable. Pour illustrer, on va utiliser le code suivant et qu'il n'est pas nécessaire de comprendre pour notre propos :

tester_gpu_simple.py

```
1  import numpy as np
2  from numba import cuda, int64
3
4  @cuda.jit('(int64[:], int64[:], int64[:])')
5  def cuda_sum(a, b, c):
6      i = cuda.grid(1)
7      c[i] = a[i] + b[i]
8
9  griddim = 5, 1
10 blockdim = 2, 1, 1
11 N = griddim[0] * blockdim[0]
12 cuda_sum_configured = cuda_sum.configure(griddim, blockdim)
13
14 a = np.array(np.random.randint(4, size=N), dtype=np.int64)
15 b = np.array(np.random.randint(4, size=N), dtype=np.int64)
16 c = np.empty_like(a)
17 print("a =", a)
18 print("b =", b)
19
20 cuda_sum_configured(a, b, c)
21
22 print('-'*(2*N+7))
23 print("c =", c)
```

Ce code est une adaptation d'un exemple fourni dans un repo Github de Numba.

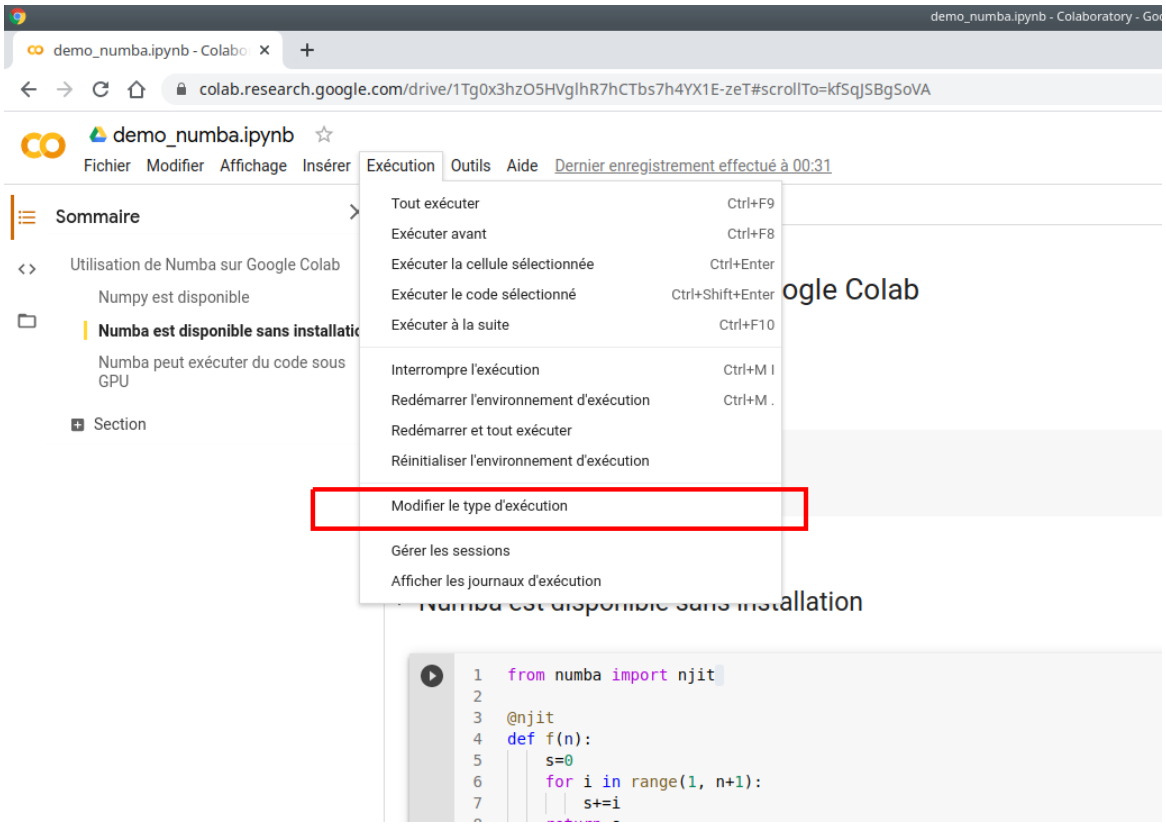
Ce code utilise le GPU (cf. lignes 4, 9-12 et 20) pour effectuer la somme `c` de deux tableaux Numpy `a` et `b` d'entiers aléatoires.

Si vous exécutez sans précaution ce code, il va renvoyer une erreur :

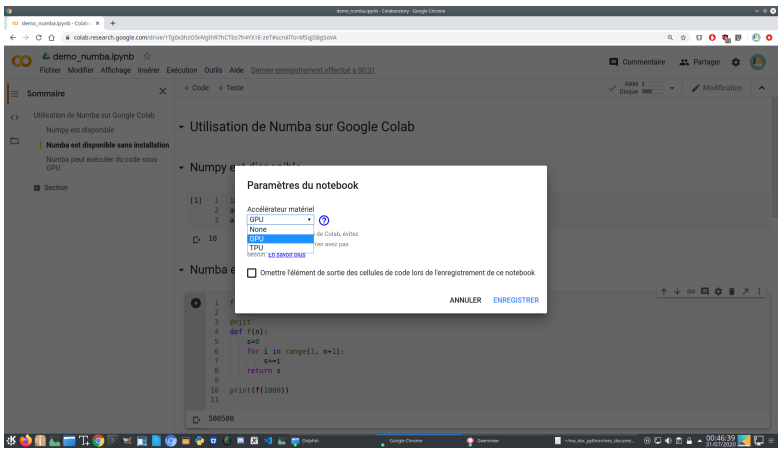
```
CudaSupportError: Error at driver init:
[100] Call to cuInit results in CUDA_ERROR_NO_DEVICE:
```

En effet, par défaut, sous Colab, les GPU ne sont pas activés. Pour les rendre disponibles, effectuer

l'opération suivante :



puis



plus précisément, cliquer sur :

```
Menu Exécution > Modifier le type d'exécution > Accélérateur matériel
                                              > GPU à la place de None
```

Une fois ce changement effectué, le code s'exécutera et affichera par exemple :

```
a = [0 2 1 1 2 1 2 1 2 2]
b = [3 1 1 3 1 2 1 0 2 3]
--------------------------
c = [3 3 2 4 3 3 3 1 4 5]
```

et on voit bien que le tableau `c` est la somme positionnelle des tableaux `a` et `b`.

Inversement, si vous exécutez du code Python avec ou sans Numba et qui n'utilise pas de GPU alors qu'il est activé vous recevrez peut-être un message du type : *Pour tirer le meilleur parti de Colab, évitez d'utiliser un GPU si vous n'en avez pas besoin.*

GPU disponibles

D'après la documentation de Google Colab, les processeurs graphiques disponibles dans la version gratuite sont parmi les modèles Nvidia suivants :

— K80

— T4

— P4

— P100

Pour savoir quel GPU vous utilisez, exécutez le code suivant :

```
from numba import cuda
print(cuda.detect())
```

qui dans mon cas affiche

```
Found 1 CUDA devices
id 0             b'Tesla T4'                              [SUPPORTED]
                      compute capability: 7.5
                           pci device id: 4
                              pci bus id: 0
Summary:
        1/1 devices are supported
True
```

mais, plus souvent, c'est un GPU K80 qui est proposé (et qui est moins rapide).

GPU, Cuda et Numba

Les cartes graphiques utilisées par les jeux vidéos sont munis de processeurs spécifiques qu'on appelle des GPU. A la différence des CPU, un GPU possède plusieurs milliers de cœurs et son architecture est parallèle puisqu'il doit gérer des matrices de pixels. Depuis le début des années 2000, les GPU sont utilisés pour faire du calcul scientifique à la manière des CPU. Cette utilisation détournée des GPU s'appelle du GPGPU. En 2019, 40% de la puissance de calcul du top 500 des super-calculateurs provient de GPU. Le GPGPU est utilisé dans de nombreuses applications.

Cuda est une plateforme développée par le fabriquant de processeurs de puces graphiques Nvidia permettant la programmation GPGPU sans avoir de connaissance en programmation de cartes graphiques. Cuda n'est supporté que par les cartes ayant un GPU Nvidia (par exemple, les cartes AMD Radeon ne sont pas compatibles). Le langage de programmation utilisé sur Cuda est le C++.

Des domaines comme l'apprentissage automatique, et les sciences des données ont des gros besoins en calculs. Depuis les années 2015, des bibliothèques Python très populaires (TensorFlow, PyTorch, Chainer) utilisent Cuda via du code C++. Mais il existe aussi des bibliothèques Python permettant d'utiliser directement Cuda, essentiellement Numba, Cupy et PyCuda. Il existe aussi Pyculib pour des besoins spécialisés mais il semblerait que Cupy soit amené à le remplacer.

Quand on code en Cuda, on écrit des *kernels*. Un kernel est juste une fonction, souvent assez simple dans son objectif de calcul, qui va être exécutée en parallèle sur le GPU sur des centaines ou des milliers de fils d'exécution (je dirai parfois *threads*). Numba permet d'écrire assez simplement des kernels Cuda. En particulier, on écrit ses kernels exclusivement en Python et non pas en C++ comme avec PyCuda.

Pour programmer en Cuda, vous devez exécuter votre code sur une machine disposant d'une carte graphique Nvidia (ou compatible). Il est également possible d'utiliser, gratuitement et avec des limitations, des GPU à distance (Google Colaboratory par exemple).

Dans cette documentation, seul un usage limité et simple de Cuda est présenté. Pour bien connaître Cuda, il vaut mieux utiliser Cuda Toolkit en C++.

Les cartes graphiques des ordinateurs

Si vous avez une carte graphique Nvidia et si elle n'est pas trop ancienne, il est possible que vous l'utilisiez pour faire du GPGPU en passant par Numba. Toutefois, cela nécessite l'installation de divers outils.

La puissance de calcul d'un carte graphique Nvidia est déterminée par un indice (dit *compute capability*) et sa sophistication est déterminée par son architecture (qui porte le nom d'un scientifique célèbre, par exemple Képler). Pour une carte de PC de gamer de bonne facture des années 2015 (architecture Maxwell), l'indice vaut un peu plus de 5. En 2020, une carte graphique de gamer de moyenne gamme et d'architecture Turing a un indice de l'ordre de 7 (compter quelques centaines d'euros). Les cartes graphiques de data center disponibles gratuitement sur Google sont souvent des tesla K80, d'une architecture Képler, d'indice 3.7. Vous pouvez trouver en 2020 des cartes graphiques pour PC d'architecture Képler et d'indice 3 de calcul, comme la Nvidia GT710 à moins de 50 euros. Il existe aussi la possibilité, pour ordinateur portable ayant un port Thunderbolt, de le connecter à un boitier contenant une carte graphique externe (compter des centaines d'euros !).

Pour être utilisable sous Numba, l'indice de calcul doit être supérieure ou égal à 2. Vous trouverez les puissances de calcul des cartes graphiques Nvidia ici : Your GPU Compute Capability et cherchez votre modèle en cliquant dans les catégories.

En général, les spécifications des PC ou d'un ordinateur portable donnent la référence de la carte graphique, ce qui permet de connaître ses caractéristiques. Chercher ensuite sur Internet si elle est compatible Nvidia. Le nom de la marque de la carte n'est généralement pas Nvidia puisque ce dernier est fabriquant de GPU (comme Intel ou ARM sont des fabriquants de CPU) et pas un assembleur de cartes. Les marques sont par exemple MSI, Asus, GigaByte.

Il est aussi possible d'acheter une carte graphique et de l'installer matériellement dans son PC. Voici un document pour installer une carte graphique à intention de GPGPU : Changing your hardware to achieve faster Deep Learning on your PC.

Une machine peut très bien être configurée avec **deux** cartes graphiques réparties ainsi (voir par exemple un montage ICI) :

— une carte graphique intégrée branchée à l'écran ou aux écrans d'affichage

— une carte graphique Nvidia débranchée de toute sortie graphique et dédiée uniquement à du calcul GPU.

Cette configuration permet (peut-être) d'avoir plus de mémoire disponible sur le GPU. Elle ne semble toutefois pas être plus rapide, au moins sur les quelques exemples que j'ai essayés.

Sous Linux, vous pouvez très bien disposer d'une carte graphique Nvidia, en particulier sur ordinateur portable, mais que sa présence ne soit pas visible dans les paramètres ; en effet, par défaut, les distributions Linux installent la carte graphique intégrée (Intel le plus souvent). Enfin, si vous installez Ubuntu 20.04, il détecte automatiquement et installe une carte graphique Nvidia placée sur un port PCIe.

Installer Cuda pour utilisation dans Numba : principe général

Avertissement

Il ne suffit pas que votre machine personnelle dispose d'une carte graphique Nvidia pour que la carte soit utilisable par Numba. Vous pouvez très bien importer le module de Numba appelé `cuda` :

```
from numba import cuda
```

sans que votre GPU soit opérationnel pour Numba.

Principe général d'installation

Je suppose que votre carte graphique a un GPU Nvidia et que vous avez installé Numba. Pour que vous puissiez utiliser votre propre GPU dans Numba, les deux conditions suivantes sont requises :

— étape 1 : installer un pilote Nvidia suffisamment récent de votre carte carte graphique

— étape 2 : installer les bibliothèques Cuda compatibles avec les pilotes de votre carte.

Par ailleurs, la réalisation de ces étapes va dépendre de votre système d'exploitation (et éventuellement de sa version) et, pour la 2de étape, de si vous utilisez ou pas Anaconda.

— Concernant le pilote, sauf pour Ubuntu 20 qui installe automatiquement, c'est une opération que *vous* devez faire vous-même, elle n'est pas automatique. Vous devez vous rendre sur le site de Nvidia pour télécharger les pilotes les plus récents ou éventuellement, désinstaller les pilotes déjà installés et les remplacer par de plus récents sinon, vous risquez de vous trouver en incompatibilité avec l'étape 2.

— Concernant la 2de étape, il s'agit d'installer des bibliothèques partagées (dll sous Windows) que Numba va appeler quand vous utiliserez Cuda. Cela peut se faire, sous Windows, comme sous Linux, assez facilement en utilisant Anaconda. Sous Linux, on peut aussi installer ces bibliothèques en utilisant le kit de développement Cuda téléchargé depuis le site de Nvidia.

Programme de test

Lorsque la première étape est satisfaite, le programme suivant s'exécute sans erreur :

`numba_detect_cuda.py`

```
from numba import cuda
print(cuda.detect())
```

Il devrait afficher quelque chose comme :

```
Found 1 CUDA devices
id 0      b'GeForce GTX 970'                              [SUPPORTED]
                      compute capability: 5.2
                           pci device id: 0
                              pci bus id: 1
Summary:
        1/1 devices are supported
True
```

Cela ne suffit pas à utiliser pleinement votre GPU sous Numba, il faut encore installer Cuda.

Pour tester le plein fonctionnement de Numba sur GPU, vous pourrez utiliser ce fichier de test suivant :

tester_gpu_simple.py

```
import numpy as np
from numba import cuda, int64

@cuda.jit('(int64[:], int64[:], int64[:])')
def cuda_sum(a, b, c):
    i = cuda.grid(1)
    c[i] = a[i] + b[i]

griddim = 5, 1
blockdim = 2, 1, 1
N = griddim[0] * blockdim[0]
cuda_sum_configured = cuda_sum.configure(griddim, blockdim)

a = np.array(np.random.randint(4, size=N), dtype=np.int64)
b = np.array(np.random.randint(4, size=N), dtype=np.int64)
c = np.empty_like(a)
print("a =", a)
print("b =", b)

cuda_sum_configured(a, b, c)

print('-'*(2*N+7))
print("c =", c)
```

qui doit afficher quelque chose du genre :

```
a = [1 2 1 0 2 1 2 2 0 1]
b = [0 3 0 3 3 1 0 2 0 3]
---------------------------
c = [1 5 1 3 5 2 2 4 0 4]
```

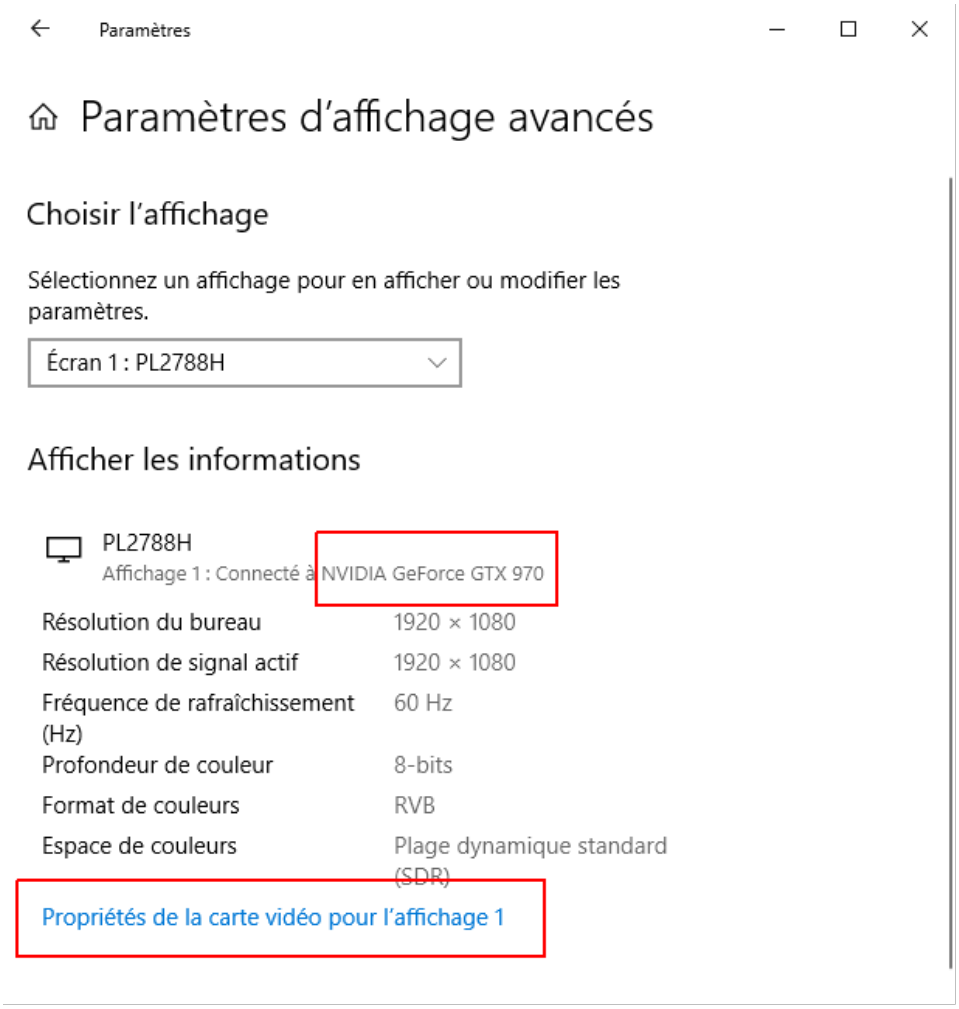
Installation du pilote Nvidia sous Windows

Ce qui suit a été testé sous Windows 10.

Déterminer sa carte graphique

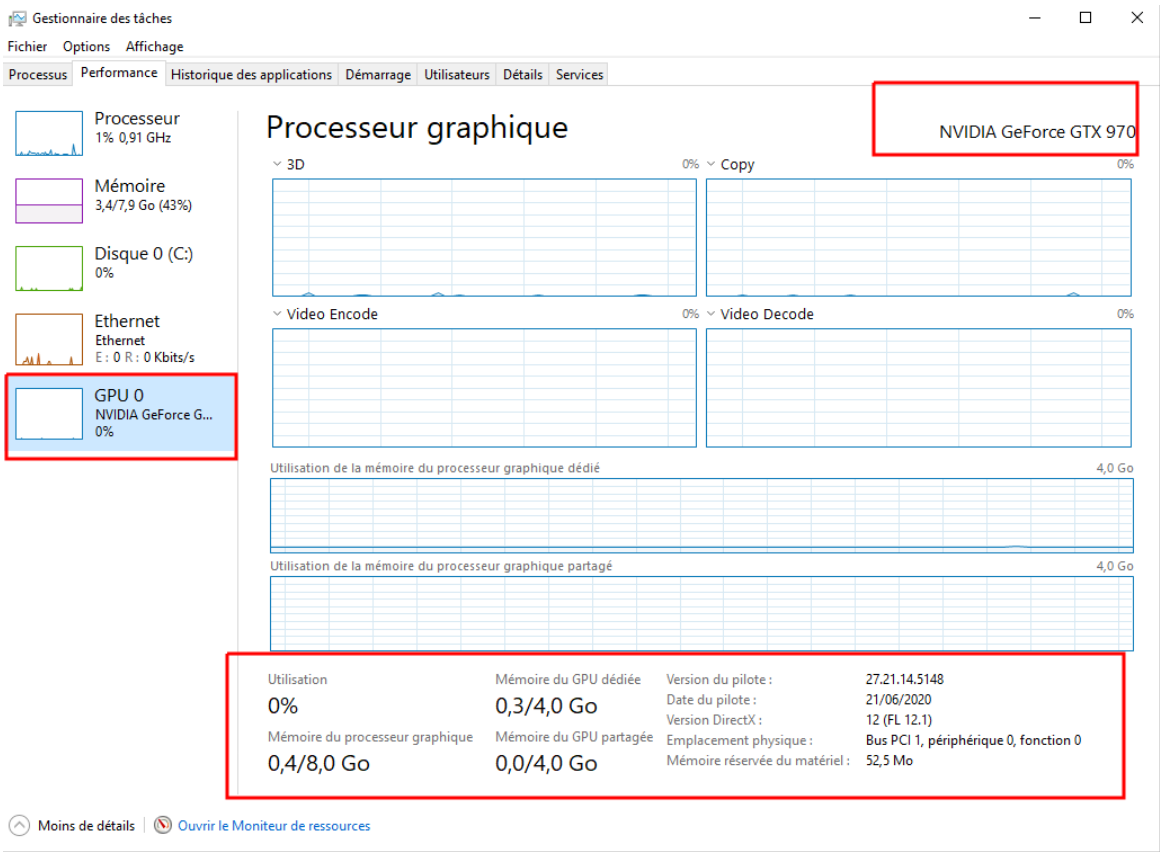
Pour connaître les caractéristiques de votre carte graphique (en supposant qu'elle soit connectée à un écran, après tout, ce n'est pas obligé) :

```
Menu Démarrer > Paramètres Windows (la roue dentée) > Système
              > Affichage > Paramètres d'affichage avancés
```



Sous le nom de l'écran vous devriez lire la référence de la carte graphique. En cliquant sur le lien *Propriétés de la carte vidéo pour l'affichage* vous devriez avoir plus d'information, en particulier sur le pilote mais rien de très utile pour programmer avec Cuda.

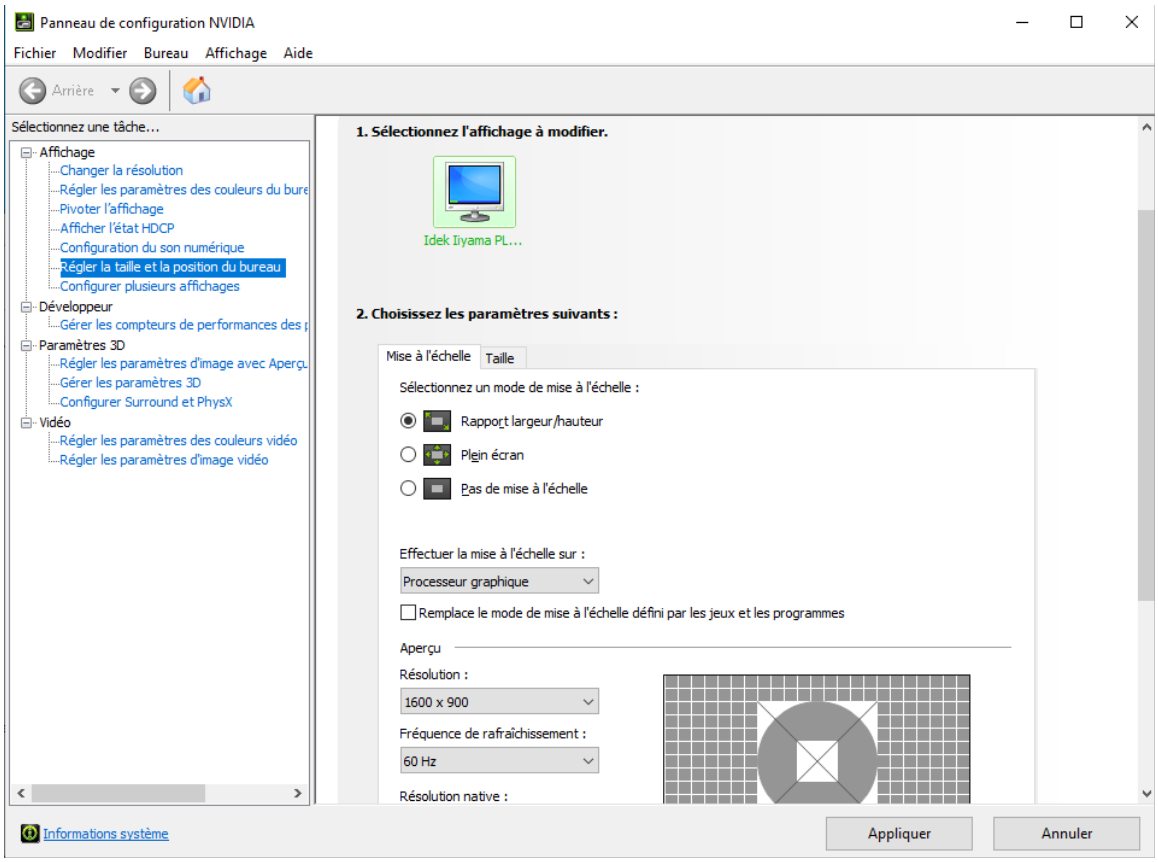
Vous pouvez aussi, via le raccourci `Alt + Ctrl + Suppr` puis **Gestionnaire des tâches**, obtenir des informations sur l'état courant de votre carte graphique (référence, mémoire utilisée, activité) :



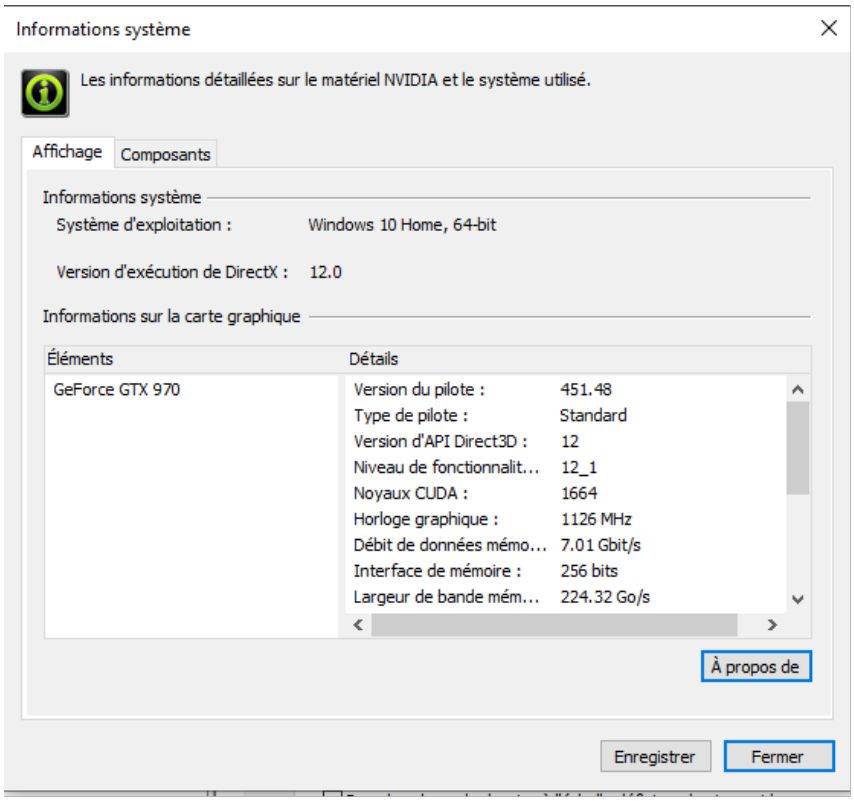
Votre obtiendrez aussi des informations à l'aide du gestionnaire de périphériques :

```
Menu démarrer > Clic Droit : gestionnaire de périphériques
              > Cartes graphiques
```

Si vous avez déjà installé vos pilotes Nvidia, cliquez droit sur le bureau et ouvrez le *Panneau de configuration de Nvidia* :



puis en bas à gauche cliquer sur *Informations système* :

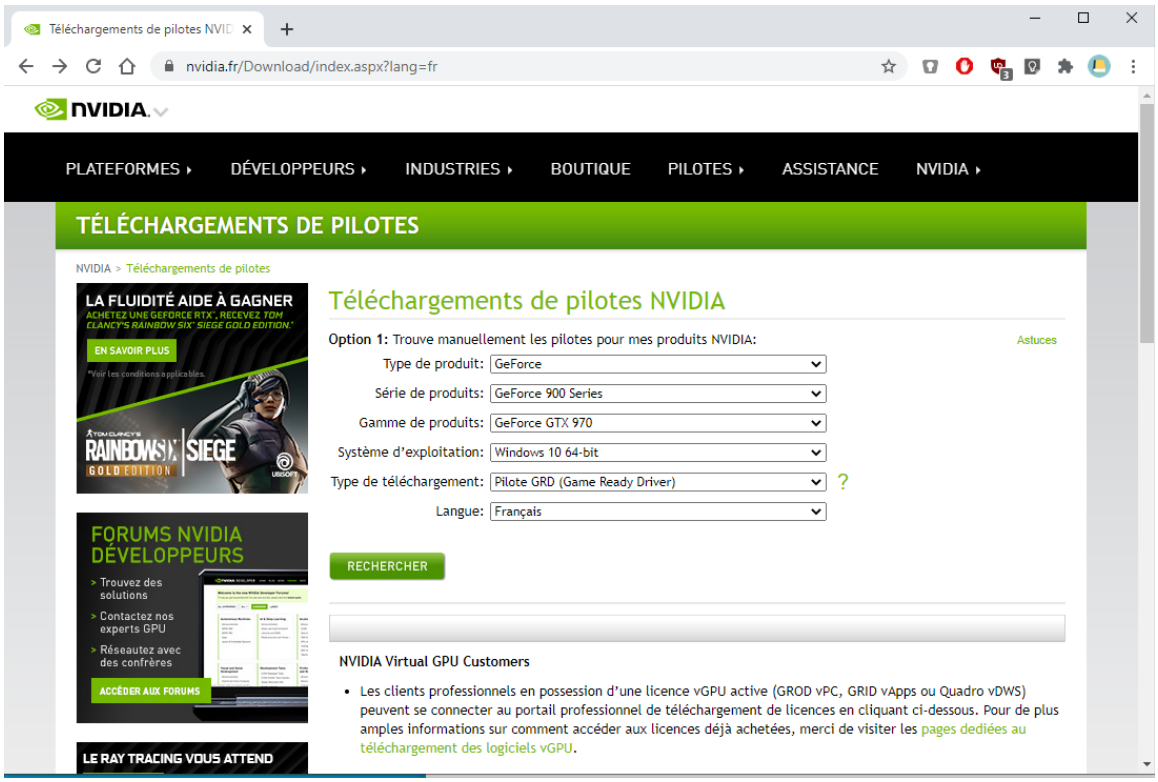


ce qui vous donnera le nom de la carte graphique et la version du pilote ayant un sens pour Cuda (version 451 dans mon cas).

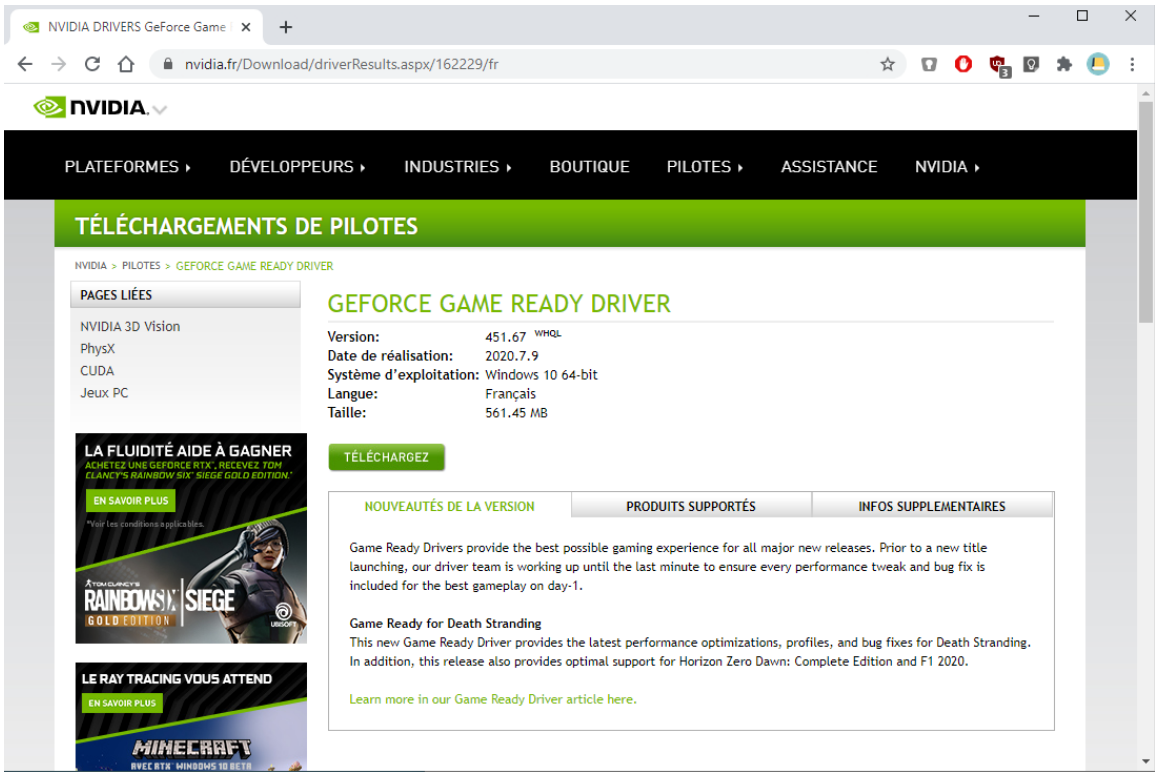
Installation ou mise à jour des pilotes

Si vous voulez mettre à jour vos pilotes (ce que j'ai préféré faire), de mon expérience, il n'est pas nécessaire de désinstaller les précédents car l'installeur mettra à jour.

Se rendre sur le site de Nvidia et dans l'application en ligne :



entrez les références de votre carte graphique et validez. Puis téléchargez le pilote proposé (de taille volumineuse, dans mon cas presque 600 Mo) :



Une fois téléchargé, exécutez le fichier. N'acceptez que les options utiles pour l'installation du pilote sinon l'outil va vous installer des applications sans intérêt pour programmer :

Choisir l'option d'**installation expresse**. L'installation dure au total une dizaine de minutes et pendant celle-ci, parfois, l'écran deviendra noir. A la fin de mon installation, le menu Démarrer n'était plus actif.

Vérifiez comme indiqué plus haut que vos pilotes ont bien été mis-à-jour en consultant comme expliqué ci-dessus le *Panneau de configuration de Nvidia*. Il ne me semble pas nécessaire de redémarrer le système.

Installation sous Windows et Anaconda 3 de Cuda pour Numba

Pour disposer de Numba et pouvoir utiliser le GPU via Numba (ce qui suppose que vous ayez un pilote Nvidia d'installé), je vais utiliser la suite Anaconda 3. Je suppose que vous l'avez déjà installé.

Détection de votre GPU

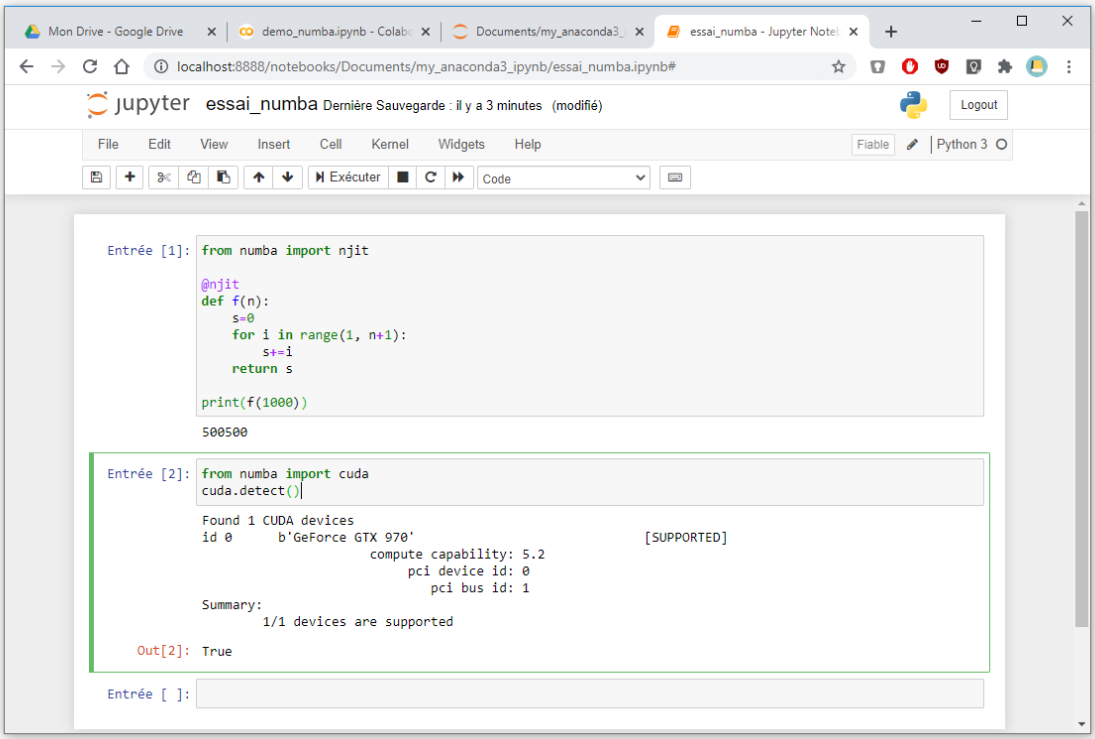
Numba est en mesure de détecter votre GPU. Dans une feuille Jupyter saisir le code suivant :

```
1 from numba import cuda
2 cuda.detect()
```

Il utilise Numba (cf. ligne 1), il doit s'exécuter sans erreur et afficher un message du genre :

```
Found 1 CUDA devices
id 0      b'GeForce GTX 970'                              [SUPPORTED]
                      compute capability: 5.2
                           pci device id: 0
                              pci bus id: 1
Summary:
        1/1 devices are supported
```

La feuille apparaît sous cette forme :



Mais Numba ne peut pas aller plus loin, et lancer des programmes qui s'exécuteraient sur votre GPU. Si vous le faisiez, vous auriez un message tel que

```
NvvmSupportError: libNVVM cannot be found. Do
`conda install cudatoolkit`: Could not find module 'nvvm.dll'
(or one of its dependencies). Try using the full
path with constructor syntax.
```

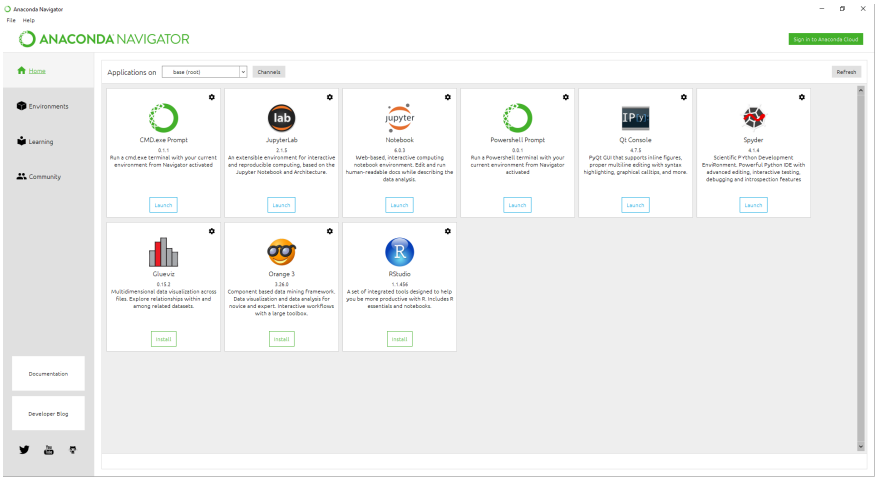
Comme le message le dit, il manque une bibliothèque dynamique (fichier dll) car vous n'avez pas installé Cuda via le paquet cudatoolkit (c'est indiqué dans le message).

Installation de cudatoolkit

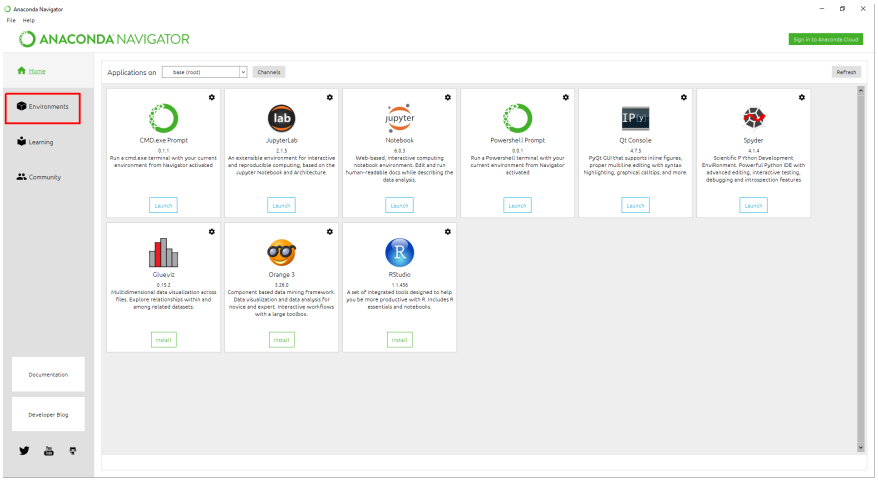
Nous allons utiliser l'installeur d'Anaconda en mode graphique. Le lancer à partir de

```
Menu Démarrer > Anaconda3 > Anaconda-Navigator
```

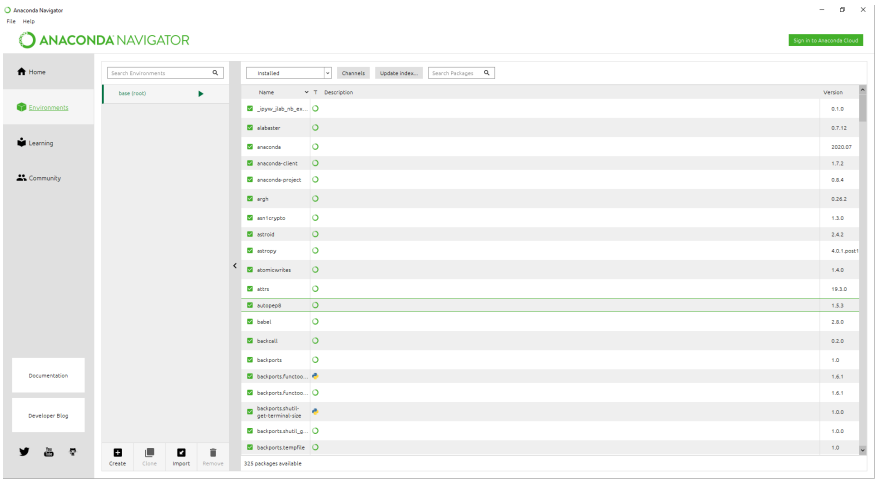
et l'interface se présente ainsi :



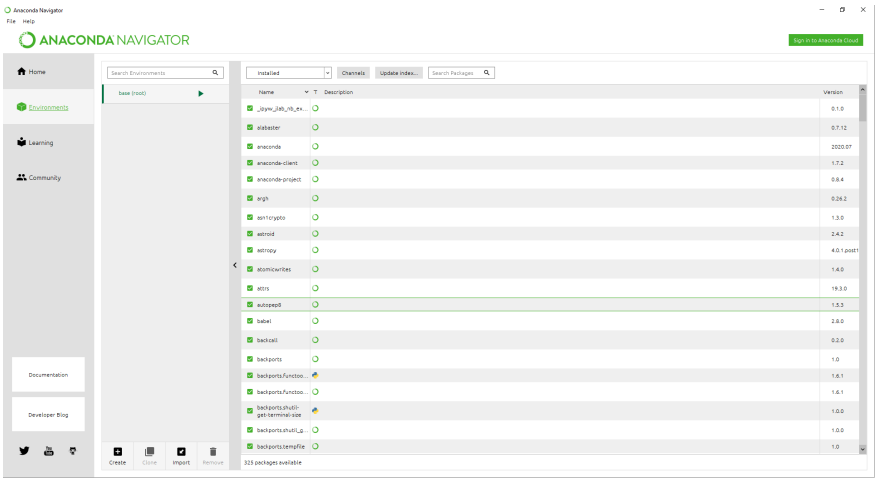
Cliquez sur le menu **Environments** :



ce qui ouvre cette vue

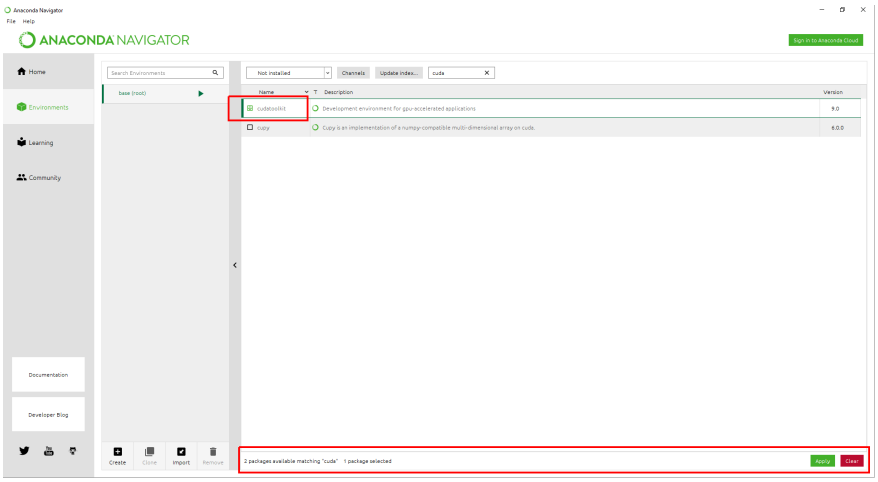


cliquez sur le bouton **Update index** pour faire apparaître les nouveaux packages disponibles :

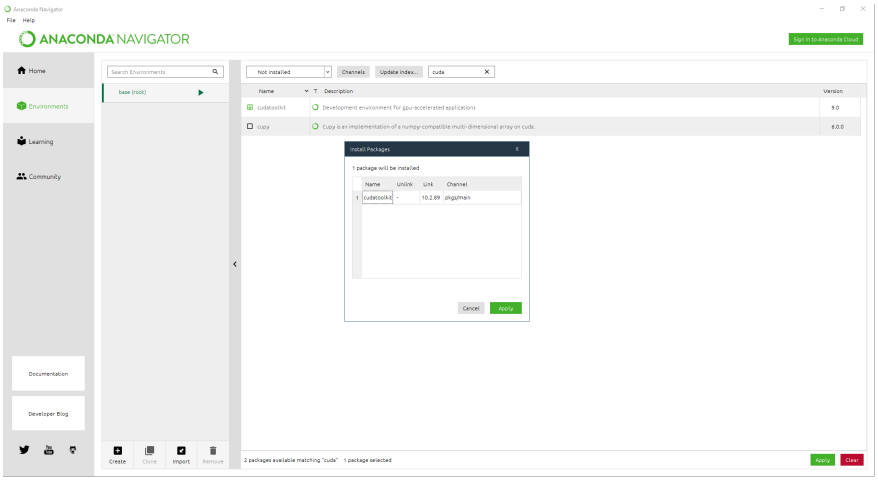


on voit qu'il y en a 325.

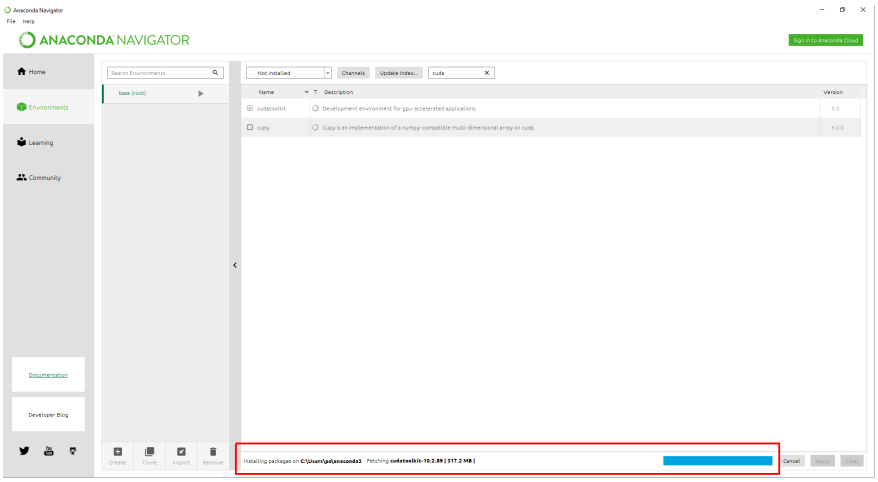
Cliquez sur le menu déroulant pour faire apparaître la ligne **Not installed**, dans la barre de recherche écrire **cuda** et validez. Cela devrait sélectionner au moins deux packages dont un des deux est nommé **cudatoolkit** et que l'on va installer. Cliquer sur la case devant **cudatoolkit** ce qui fait apparaître deux boutons en bas à droite de la fenêtre et valider le bouton vert **Apply** :



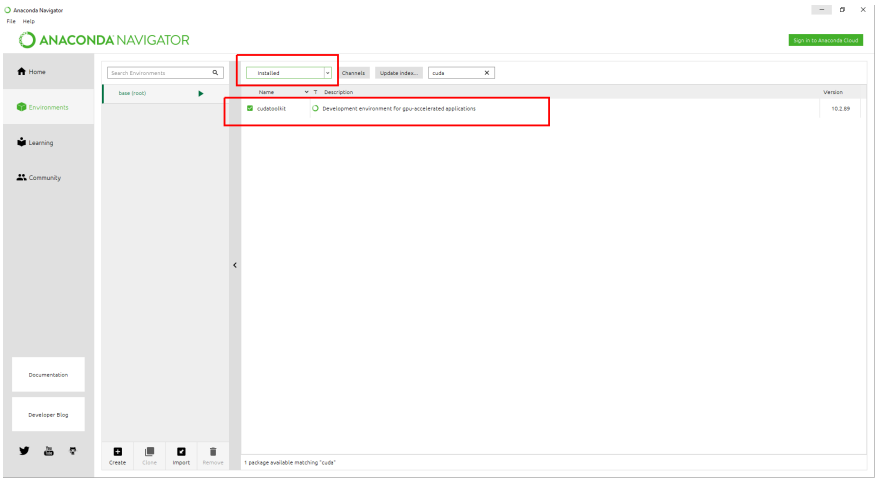
Attendre que le système regarde s'il peut installer le package. Un panneau va finir par s'ouvrir indiquant les packages à installer, ici un seul :



en particulier on voit qu'il va installer Cuda en version 10.2. Valider en cliquant sur le bouton vert **Apply** du panneau ce qui va lancer un téléchargement puis une installation (compter plusieurs minutes) :



Une fois l'installation terminé (ce n'est pas clairement annoncé), assurez-vous que **cudatoolkit** est bien installé ; pour cela, positionnez le menu déroulant sur *Installed*, en ayant gardé dans la barre de recherche le mot **cuda** et vous verrez que la ligne cudatoolkit apparaît :



Tester un programme Numba sous GPU

Ouvrir la feuille Jupyter Notebook déjà utilisée, ou une autre, pensez à **réinitialiser** l'environnement de la feuille :

```
Menu Kernel > Restart and clear output
```

et placer le code Numba suivant dans une cellule :

tester_gpu_simple.py

```
1 import numpy as np
2 from numba import cuda, int64
3
4 @cuda.jit('(int64[:], int64[:], int64[:])')
5 def cuda_sum(a, b, c):
6     i = cuda.grid(1)
```

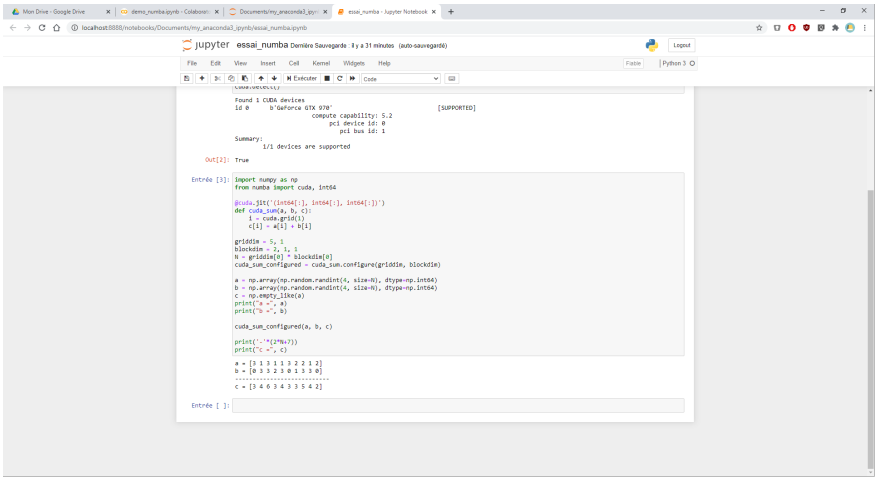
```
7       c[i] = a[i] + b[i]
8
9   griddim = 5, 1
10  blockdim = 2, 1, 1
11  N = griddim[0] * blockdim[0]
12  cuda_sum_configured = cuda_sum.configure(griddim, blockdim)
13
14  a = np.array(np.random.randint(4, size=N), dtype=np.int64)
15  b = np.array(np.random.randint(4, size=N), dtype=np.int64)
16  c = np.empty_like(a)
17  print("a =", a)
18  print("b =", b)
19
20  cuda_sum_configured(a, b, c)
21
22  print('-'*(2*N+7))
23  print("c =", c)
```

Ce code effectue des calculs sur le GPU (cf. ligne 4). Si on lance l'exécution, aucun message d'erreur ne doit apparaître et on lira une sortie du genre :

```
a = [1 2 1 0 2 1 2 2 0 1]
b = [0 3 0 3 3 1 0 2 0 3]
---------------------------
c = [1 5 1 3 5 2 2 4 0 4]
```

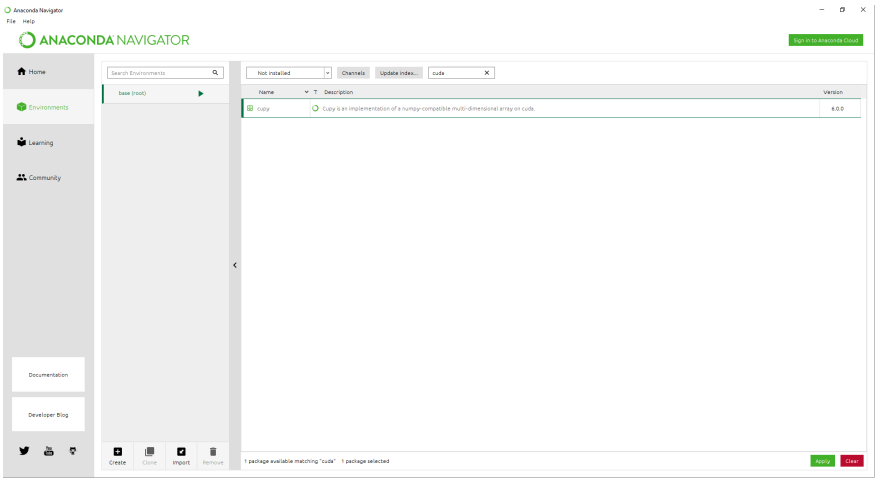
Votre installation de Numba est alors pleinement opérationnelle !

Voici une vue du résultat :

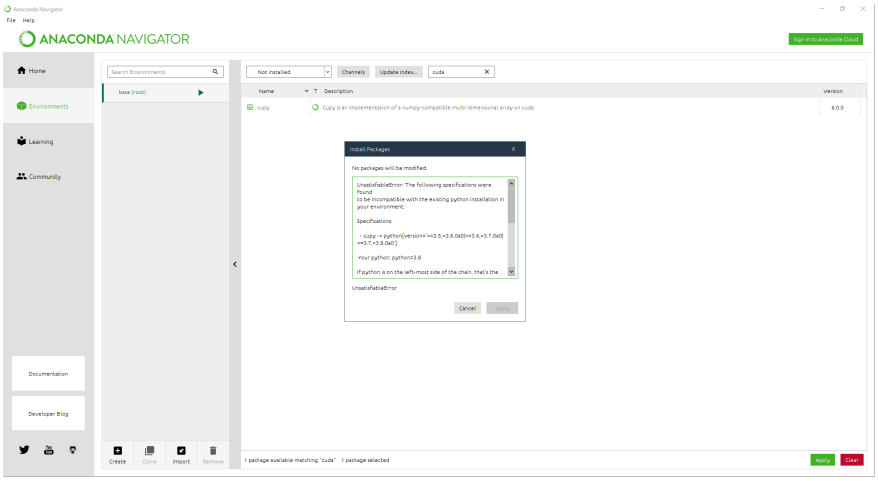


Anaconda n'est pas infaillible

Les installations sous Anaconda ne se passent pas toujours aussi facilement, en particulier lorsque les packages utilisent le GPU. Sur notre lancée, essayons d'installer Cupy qui est un module Python permettant d'utiliser les bibliothèques Cuda. On fait comme précédemment mais pour **cupy** au lieu de **cudatoolkit** , qui est bien listé dans les packages



et à l'étape de préparation de l'installation, après une attente assez longue, on obtient un message d'erreur :



et dont voici le contenu :

```
UnsatisfiableError: The following specifications were found
to be incompatible with the existing python installation in your
environment:

Specifications:

  - cupy -> python[version='>=3.5,<3.6.0a0|>=3.6,<3.7.0a0|>=3.7,
  <3.8.0a0']

Your python: python=3.8

If python is on the left-most side of the chain, that's the version
you've asked for. When python appears to the right, that indicates
that the thing on the left is somehow not available for the python
```

```
version you are constrained to. Note that conda will not change your
python version to a different minor version unless you explicitly specify
that.

The following specifications were found to be incompatible with your CUDA
driver:

  - feature:/win-64::__cuda==11.0=0

Your installed CUDA driver is: 11.0
```

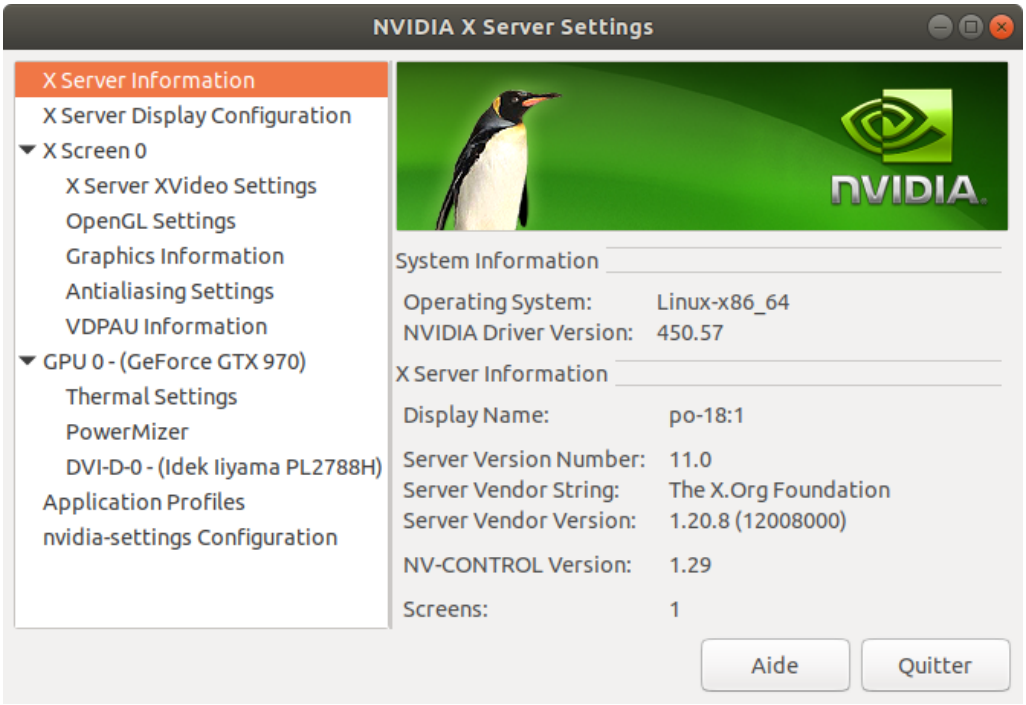
J'ai ensuite essayé d'installer cupy en utilisant une autre source (conda-forge) sans davantage de succès.

Installation du pilote Nvidia sous Ubuntu 20

A partir d'Ubunutu 20.04, les pilotes Nvidia adaptés à votre matériel sont automatiquement disponibles car installés lors de l'installation de l'OS. Pour le confimer, lancer la commande

```
$ nvidia-settings
```

ce qui montre un applet graphique qui donne les caractéristiques du GPU



Linux : détection de la carte graphique

Je ne parlerai ci-dessous que des distributions Ubuntu (séries 18 à 20, les LTS ayant un support jusqu'en 2023 ou 2025).

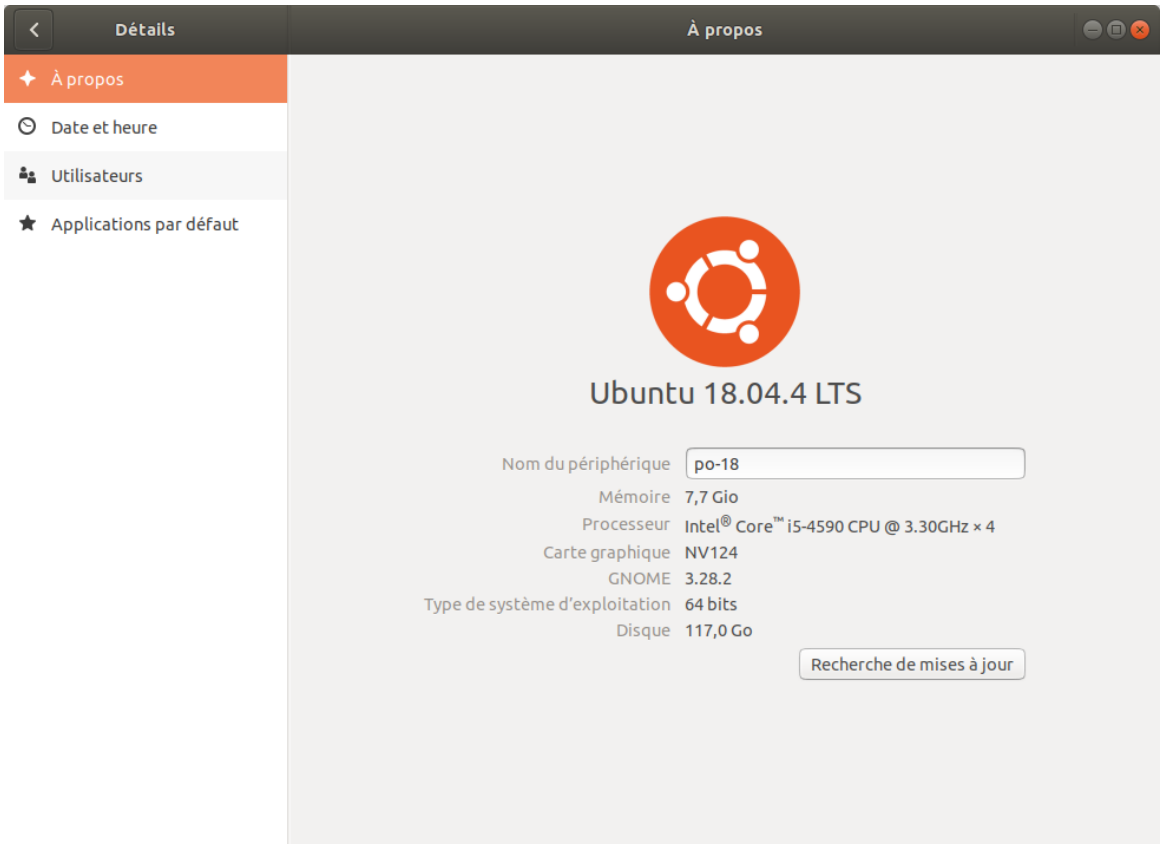
Précision : tout ce qui est expliqué dans cette section est sans rapport avec Anaconda qui **ne** se charge **pas** d'installer un driver matériel.

Détection du matériel

Pour connaître les périphériques graphiques potentiellement disponibles, on peut exécuter la commande suivante :

```
$ sudo lshw -C video
[sudo] Mot de passe de po :
  *-display
       description: VGA compatible controller
       produit: GM204 [GeForce GTX 970]
       fabriquant: NVIDIA Corporation
       identifiant matériel: 0
       information bus: pci@0000:01:00.0
       version: a1
       bits: 64 bits
       horloge: 33MHz
       fonctionnalités: pm msi pciexpress vga_controller bus_master cap_list rom
       configuration: driver=nouveau latency=0
       ressources: irq:34 mémoire:f6000000-f6ffffff mémoire:e0000000-efffffff
                   mémoire:f0000000-f1ffffff portE/S:e000(taille=128)
                   mémoire:c0000-dffff
```

La commande ci-dessus a été exécutée sur une Ubuntu 18.04 fraîchement installée. On voit qu'il y a une carte graphique Nvidia et que le pilote s'appelle *nouveau*. Ce pilote n'est pas du tout capable de permettre de réaliser du GPGPU avec Cuda. Quand on regarde dans les paramètres, rubrique **À propos**, on lit que la carte graphique s'appelle NV124



qui correspond à un pilote dit nouveau.

Pour que vous puissiez utiliser votre GPU avec Numba, le pilote **doit** être un pilote Nvidia avec une ligne comme celle-ci :

```
$ sudo lshw -C video
...
       configuration : driver=nvidia latency=0
...
```

Une autre façon de connaître ses cartes graphiques est par la commande `hwinfo` (à installer au préalable et exécuter en sudo) :

```
$ hwinfo --gfxcard --short
graphics card:
                       nVidia GM204 [GeForce GTX 970]

Primary display adapter: #14
```

On peut aussi essayer ceci (je crois que ça affiche les drivers propriétaires installés ou installables) :

```
$ sudo ubuntu-drivers devices
== /sys/devices/pci0000:00/0000:00:01.0/0000:01:00.0 ==
modalias : pci:v000010DEd000013C2sv000010DEsd00001116bc03sc00i00
vendor   : NVIDIA Corporation
model    : GM204 [GeForce GTX 970]
```

```
driver   : nvidia-driver-440 - third-party free
driver   : nvidia-driver-435 - distro non-free
driver   : nvidia-driver-418-server - distro non-free
driver   : nvidia-driver-450 - third-party free recommended
driver   : nvidia-driver-390 - distro non-free
driver   : nvidia-driver-440-server - distro non-free
driver   : xserver-xorg-video-nouveau - distro free builtin
```

La commande peut très bien ne rien afficher (situation déjà rencontrée) même si vous avez une carte graphique compatible Nvidia, par exemple si votre affichage se fait (par défaut) via une carte graphique intégrée Intel.

Autre possibilité :

```
$lspci | grep -i nvidia
01:00.0 VGA compatible controller: NVIDIA Corporation GM204
                                   [GeForce GTX 970] (rev a1)
01:00.1 Audio device: NVIDIA Corporation GM204 High Definition
                                   Audio Controller (rev a1)
```

Linux : installation automatique des pilotes Nvidia

Il se trouve qu'Ubuntu (en version 18 et sans doute 19) permet de lancer l'installation de pilotes propriétaires (il paraît que ce n'est pas le cas pour les versions antérieures). Pour cela aller dans la logithèque (dite *Logiciels Ubuntu*), taper *pilotes* dans la barre de recherche et choisir *Software Updates*, cliquer sur le bouton **lancer**

cliquer sur l'onglet *Pilotes additionnels* :

(attendre que le panneau soit chargé). Une case à cocher vous indique le pilote éventuellement utilisé.

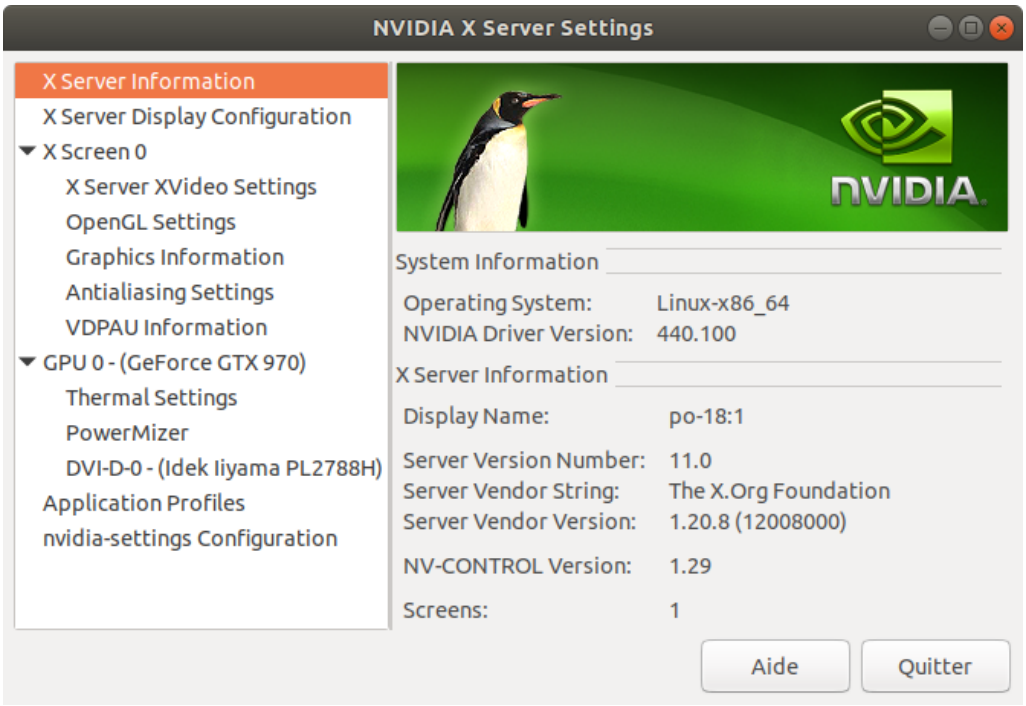
Choisir le pilote recommandé et l'installer (en cliquant sur le bouton) ce qui prend plusieurs minutes :

Ensuite, il faut absolument **redémarrer** l'ordinateur (opération normalement très rare sous Linux) pour que les changements soient pris en compte.

Au redémarrage, pour vérifier, lancer la commande

```
$ nvidia-settings
```

qui doit montrer :



Une alternative (je pense équivalente) à la méthode ci-dessus d'installation du pilote est décrite ICI, regarder la première méthode.

Faire fonctionner Cuda pour Numba sous Linux avec Anaconda

Je suppose que vous avez installé

— Anaconda 3 sous Linux, comme expliqué dans un paragraphe précédent.
— le pilote Nvidia de votre carte graphique.

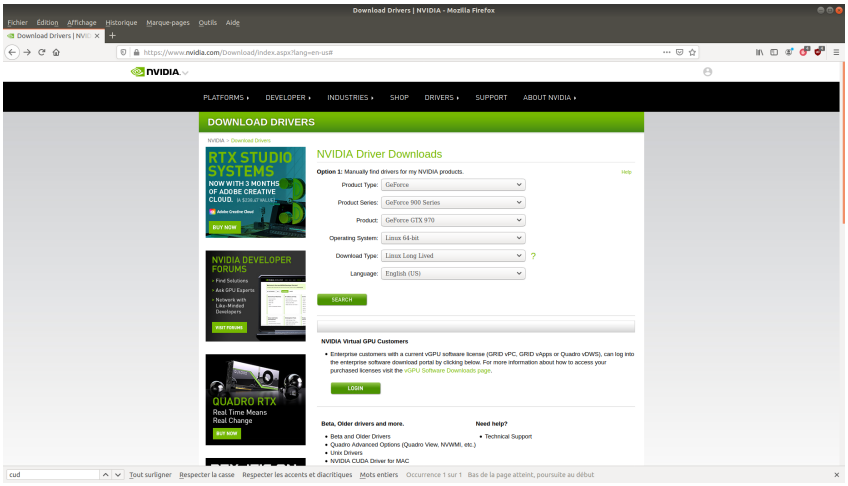
Reste à faire fonctionner le GPU sous Numba. Pour cela, il suffit juste de demander à Anaconda d'installer le package **cudatoolkit**. La procédure est exactement la même que sous Windows. Il suffit juste

— d'ouvrir Anaconda Navigator,
— de cliquer sur le menu **Environments**,
— d'entrer **cuda** dans la barre de recherche
— de positionner le menu déroulant sur **Not installed**
— de sélectionner **cudatoolkit** en cochant la case
— de valider en cliquant sur le bouton **Apply**
— de valider à nouveau quand Anaconda à trouvé le package à installer.

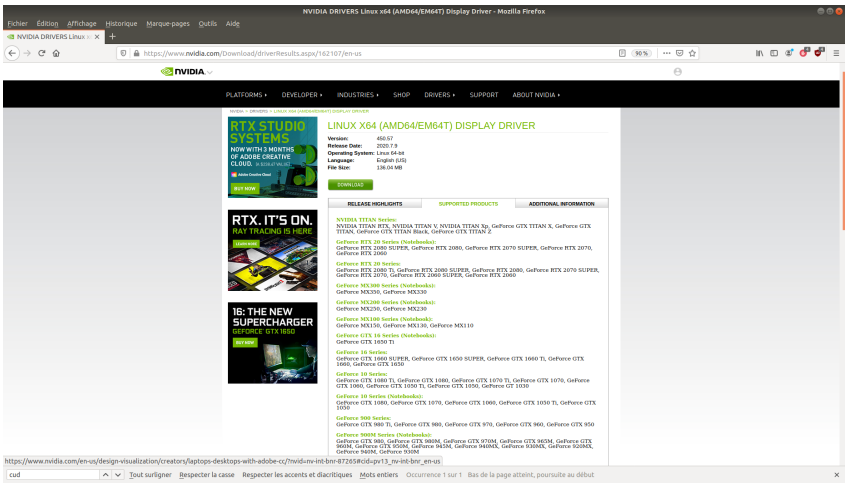
Comme sous Windows, faire un test avec le fichier `tester_gpu_simple.py` pour s'assurer que le GPU exécute sans erreurs les calculs sous Numba + Cuda.

Linux : installation manuelle des pilotes Nvidia

On va installer en ligne de commande les pilotes de la carte Nvidia GeForce 970. Le site de Nvidia indique, en utilisant l'application en ligne :



que le pilote 450 prend en charge la 970 :



Donc, j'essaye le pilote 450. Le code à lancer est le suivant :

```
sudo apt-get purge nvidia*
sudo add-apt-repository ppa:graphics-drivers
sudo apt-get update
sudo apt-get install nvidia-driver-450
```

La première ligne sert à désinstaller des pilotes Nvidia éventuellement déjà installés. La dernière ligne installe le pilote en version 450. L'installation est relativement longue (compter un bon quart d'heure).

Il faut redémarrer la machine sinon le nouveau pilote n'est pas reconnu. J'en veux pour preuve que si on lance

numba_detect_cuda.py

```
from numba import cuda
cuda.detect()
```

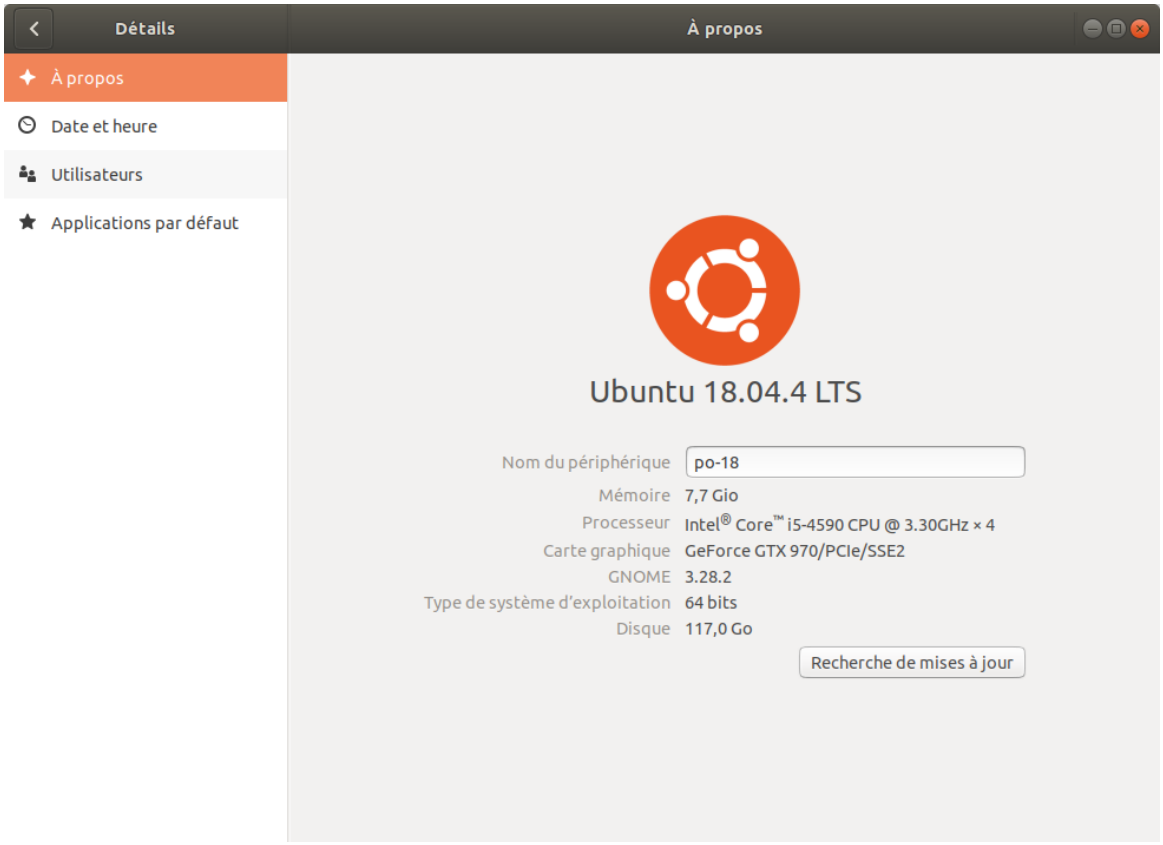
on obtient :

```
$ python3 numba_detect_cuda.py
Traceback (most recent call last):

... omis ...

[100] Call to cuInit results in CUDA_ERROR_NO_DEVICE:
```

Et de même si on ne reboote pas, on obtient

```
$ nvidia-settings

ERROR: NVIDIA driver is not loaded


ERROR: Unable to load info from any available system
```

Une fois redémarré, le panneau **À propos** des paramètres montre le changement de pilote :



La commande suivante montre aussi que le pilote Nvidia est installé :

```
$ sudo lshw -C video
[sudo] Mot de passe de po :
  *-display
       description: VGA compatible controller
       produit: GM204 [GeForce GTX 970]
       fabriquant: NVIDIA Corporation
       identifiant matériel: 0
       information bus: pci@0000:01:00.0
       version: a1
       bits: 64 bits
       horloge: 33MHz
       fonctionnalités: pm msi pciexpress vga_controller bus_master cap_list rom
       configuration: driver=nvidia latency=0
       ressources: irq:34 mémoire:f6000000-f6ffffff mémoire:e0000000-efffffff
       mémoire:f0000000-f1ffffff portE/S:e000(taille=128) mémoire:c0000-dffff
```

Avant installation, la commande

```
lsmod | grep nvidia
```

n'affichait rien. Cette fois, elle affiche :

```
$ lsmod | grep nvidia
nvidia_uvm            970752  0
nvidia_drm             49152  7
nvidia_modeset       1183744  13 nvidia_drm
nvidia              19681280  591 nvidia_uvm,nvidia_modeset
drm_kms_helper        184320  1 nvidia_drm
drm                   491520  10 drm_kms_helper,nvidia_drm
```

Le comportement de Numba a changé, il détecte le GPU :

numba_detect_cuda.py

```
from numba import cuda
cuda.detect()
```

qui lancé donne

```
$ python3 numba_detect_cuda.py
Found 1 CUDA devices
id 0      b'GeForce GTX 970'                              [SUPPORTED]
                      compute capability: 5.2
                           pci device id: 0
                              pci bus id: 1
Summary:
        1/1 devices are supported
```

Toutefois, il est incapable de l'utiliser pour faire des calculs :

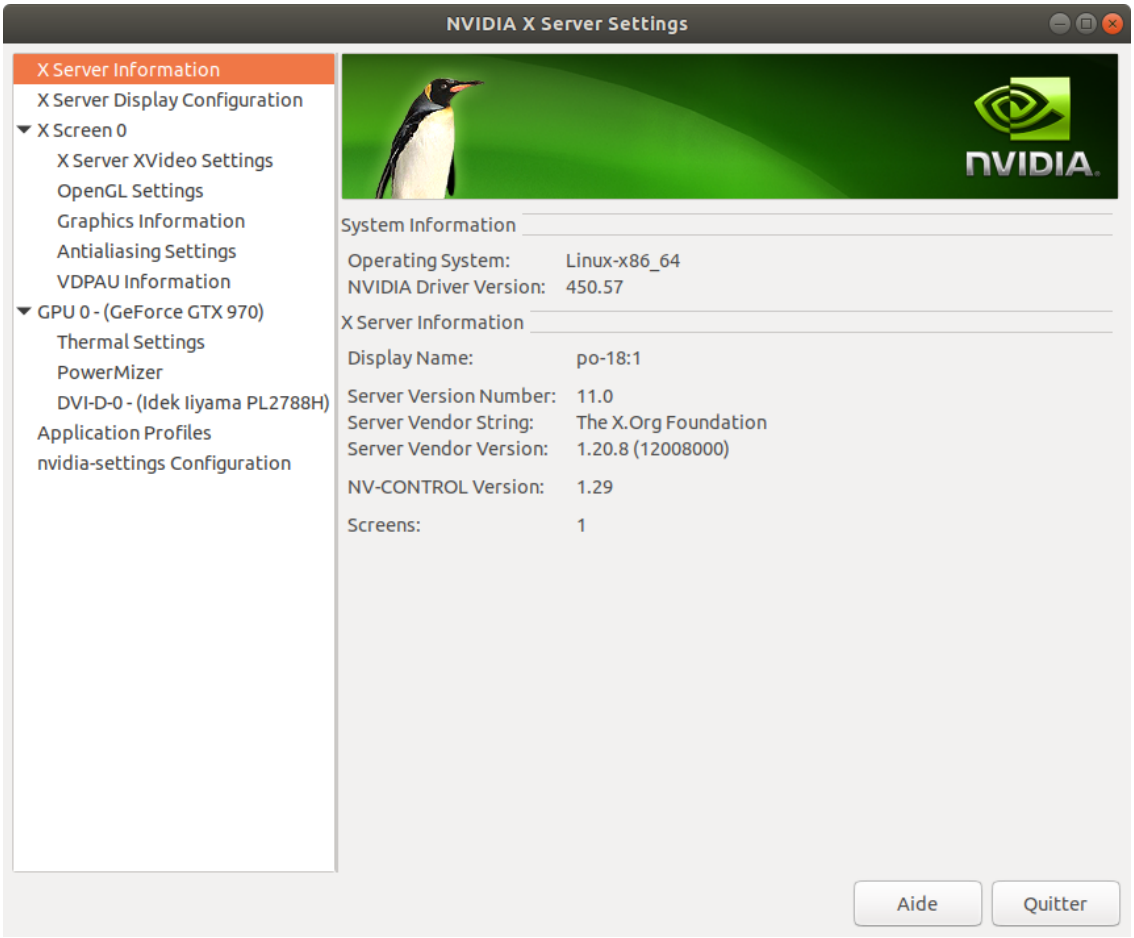
```
$ python3 tester_cuda_matmul.py

... omis ...
```

```
numba.cuda.cudadrv.error.NvvmSupportError: libNVVM cannot be found.
Do `conda install cudatoolkit`: libnvvm.so: cannot open shared object
file: No such file or directory
```

L'applet graphique

```
$ nvidia-settings
```

fonctionne et montre la version du pilote (450) :



De même, la commande ci-dessous montre le comportement en direct du GPU :

```
$ nvidia-smi
Wed Jul 29 19:32:53 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.57       Driver Version: 450.57       CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GeForce GTX 970     Off  | 00000000:01:00.0  On |                  N/A |
| 34%   44C    P8    21W / 173W |    277MiB /  4041MiB |      0%      Default |
```

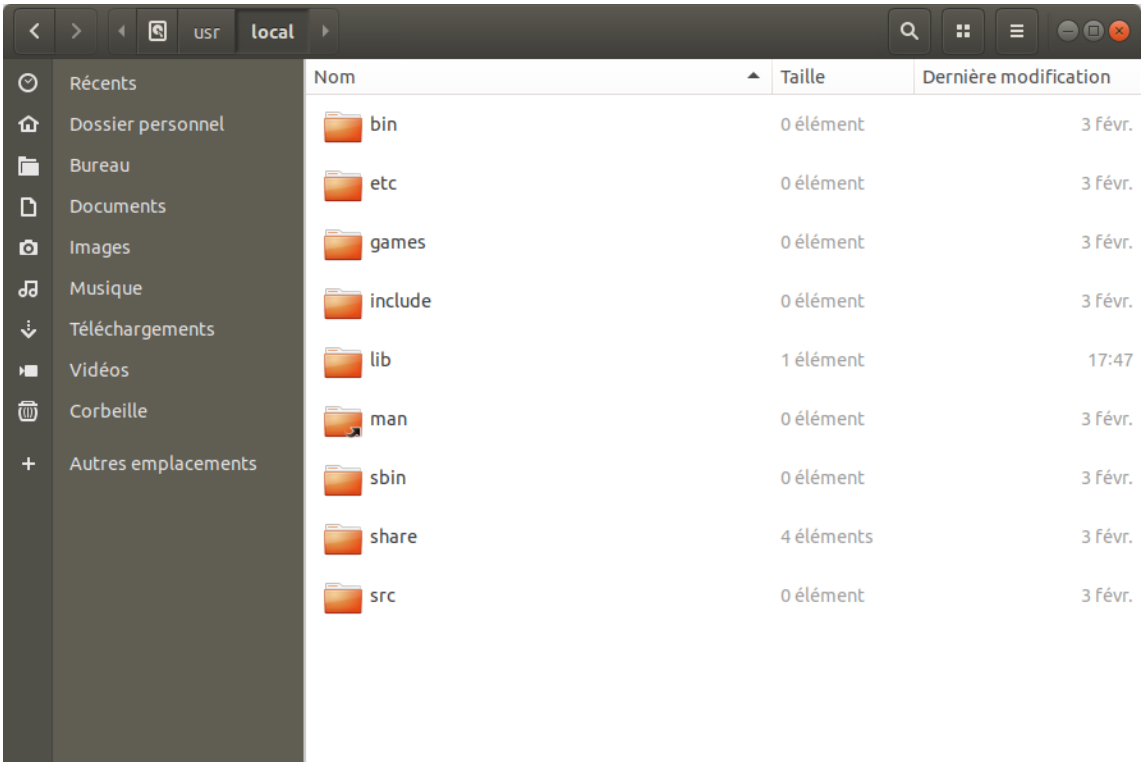
```
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1217      G   /usr/lib/xorg/Xorg                 15MiB |
|    0   N/A  N/A      1275      G   /usr/bin/gnome-shell               49MiB |
|    0   N/A  N/A      1451      G   /usr/lib/xorg/Xorg                117MiB |
|    0   N/A  N/A      1600      G   /usr/bin/gnome-shell               85MiB |
|    0   N/A  N/A      1918      G   gnome-control-center                2MiB |
+-----------------------------------------------------------------------------+
```

Installation de Cuda Toolkit sous Linux

Cette vidéo applique sous Ubuntu 16.04 le même procédé d'installation que celui qui est proposé ci-dessous (mais elle n'explique pas clairement comment choisir la version de Cuda adaptée au pilote)

Je suppose que votre driver Nvidia est installé. Je rappelle que l'installation des bibliothèques Cuda est indispensable à l'utilisation du GPU sous Numba. L'objectif ici est d'utiliser Numba et le GPU **sans passer par Anaconda**. Je me suis basé sur ce compte-rendu.

Par défaut, Cuda s'installera dans le répertoire `/usr/local` que voici avant installation :



Trouver la version et l'installer

Cherchons la version du Cuda toolkit à installer. Pour cela, lancer la ligne de commande suivante :

```
$ nvidia-smi
Wed Jul 29 00:30:29 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.100      Driver Version: 440.100      CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 970     Off  | 00000000:01:00.0  On |                  N/A |
| 37%   47C    P8    22W / 173W |    226MiB /  4041MiB |      2%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1226      G   /usr/lib/xorg/Xorg                            16MiB |
|    0      1282      G   /usr/bin/gnome-shell                          49MiB |
|    0      1455      G   /usr/lib/xorg/Xorg                            85MiB |
|    0      1589      G   /usr/bin/gnome-shell                          68MiB |
+-----------------------------------------------------------------------------+
```

[smi veut dire *System Management Interface*]

En haut du tableau, on lit que le pilote matériel 440 est capable de prendre en charge au plus la version de Cuda 10.2. C'est celle-là que j'installerai ci-dessous. Se rendre sur le site de Nvidia pour télécharger les binaires de Cuda toolkit en version 10.2. Pour trouver la page correspondant à ma version, j'ai consulté cette page CUDA Toolkit Archive du site de Nvidia et j'ai choisi la version 10.2.

Depuis la page de téléchargement, indiquer dans l'application en ligne

— son OS (Linux),

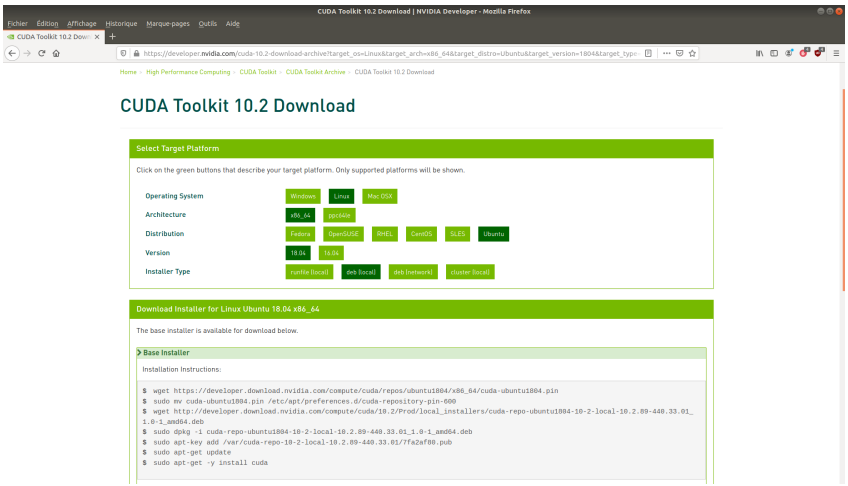
— l'architecture (x86_64),

— la distribution (Ubuntu),

— la version (18.4),

— le type de fichier, à savoir *deb(local)*.

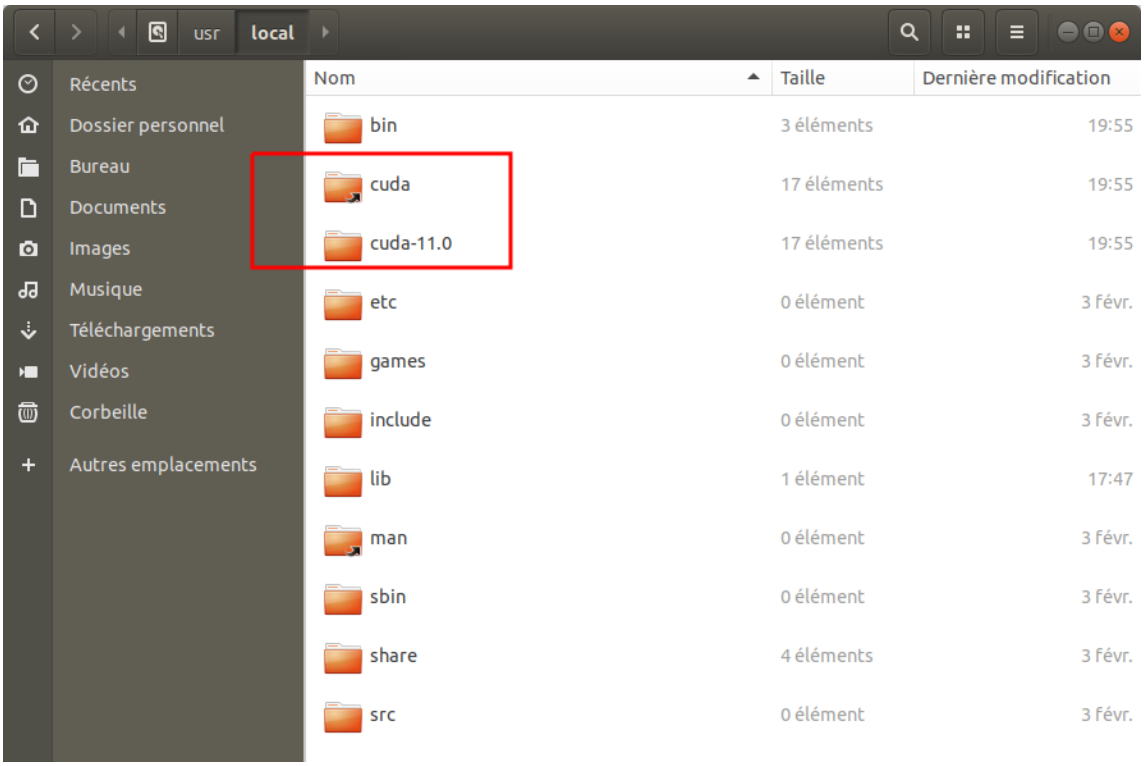
Une fois toutes les options choisies, un panneau s'ouvre :



qui contiennent des lignes de commande, et qui placées dans un fichier bash donnerait :

```
wget "https://developer.download.nvidia.com/compute/"
"cuda/repos/ubuntu1804/x86_64/cuda-ubuntu1804.pin"

sudo mv cuda-ubuntu1804.pin \
/etc/apt/preferences.d/cuda-repository-pin-600

wget "http://developer.download.nvidia.com/compute/cuda/10.2/\
Prod/local_installers/cuda-repo-ubuntu1804-\
10-2-local-10.2.89-440.33.01_1.0-1_amd64.deb"

sudo dpkg -i cuda-repo-ubuntu1804-10-2-local-\
10.2.89-440.33.01_1.0-1_amd64.deb

sudo apt-key add /var/cuda-repo-10-2-local-\
10.2.89-440.33.01/7fa2af80.pub

sudo apt-get update

sudo apt-get -y install cuda
```

et indique les commandes à entrer. Les exécuter ligne par ligne dans un terminal (oudans un fichier bash). Une fois cela terminé, Cuda est installé, vous pouvez d'ailleurs vous en assurer en allant voir le dossier `/usr/local` :



qui contient désormais un répertoire pour Cuda (l'image, c'était pour la version 11 de Cuda).

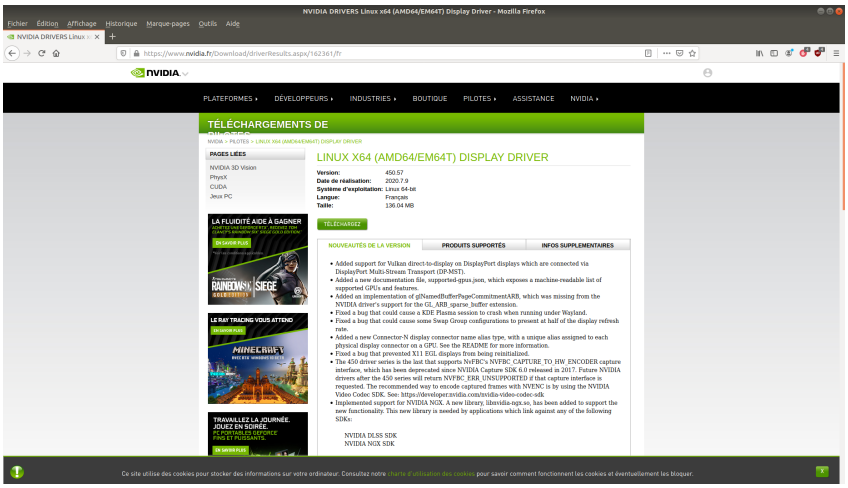
Au passage, cela vous donne accès aux outils de développement C++ de Cuda, par exemple son compilateur :

```
$ /usr/local/cuda/bin/nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Thu_Jun_11_22:26:38_PDT_2020
Cuda compilation tools, release 11.0, V11.0.194
Build cuda_11.0_bu.TC445_37.28540450_0
```

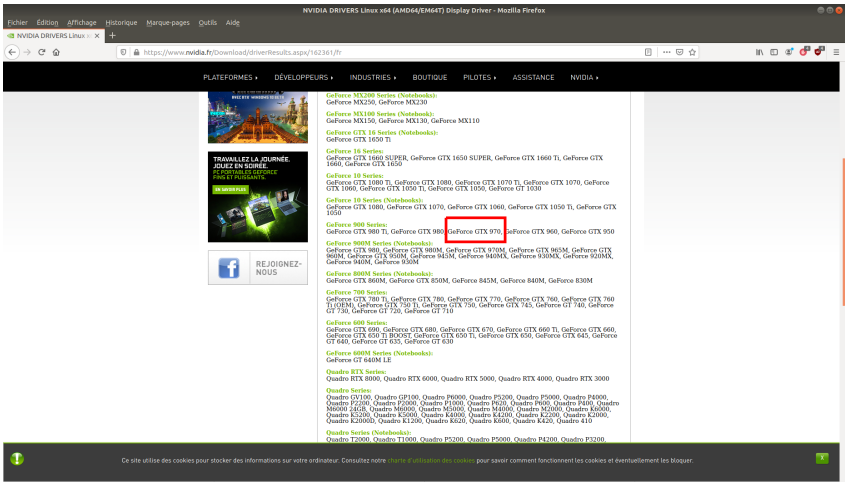
Linux : installation des dernières versions du pilote et de Cuda

Soit toujours à configurer Cuda pour une carte GeForce 970. Cette fois, on voudrait installer les versions compatibles **les plus récentes** possibles.

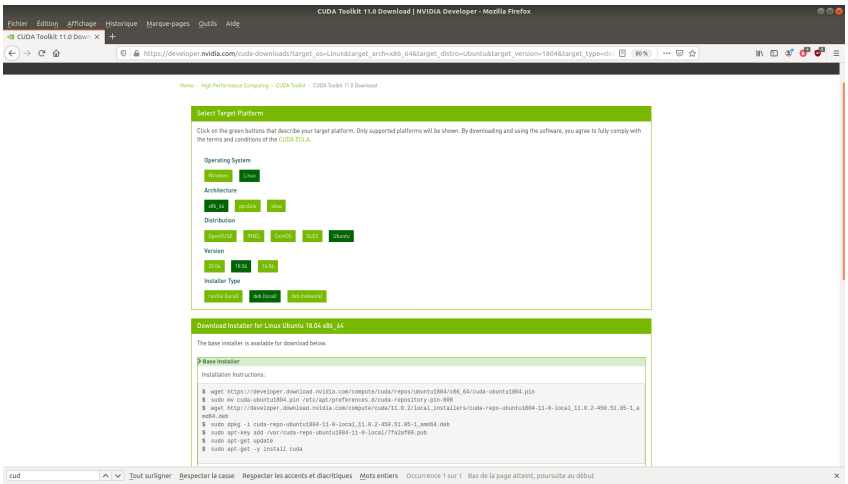
Les cartes supportées par version de pilotes Linux sont détaillées sur une page du site de Nvidia : Pilotes Unix. Plus le numéro de pilote est élevé plus le pilote est à jour et les pilotes sont rétro-compatibles avec d'anciennes versions de Cuda. Plus précisément, cliquez sur une version de pilote puis regardez s'il est adapté à votre matériel (onglet *Produits supportés*). En août 2020, le dernière pilote est le 450, En cliquant sur le lien 450, on arrive à une page de téléchargement :



et en regardant l'onglet **Produits supportés**, on voit que la GeForce 970 est prise en charge :



Le fichier à télécharger est un fichier `run` mais je préfère télécharger un fichier `deb`. Il suffit donc de se rendre sur la page de téléchargement de Cuda Toolkit et en utilisant l'application en ligne :



puis de lancer les commandes indiquées. Ecrit dans un fichier bash, cela donnerait :

```
wget "https://developer.download.nvidia.com/compute/"
"cuda/repos/ubuntu1804/x86_64/cuda-ubuntu1804.pin"

sudo mv cuda-ubuntu1804.pin \
/etc/apt/preferences.d/cuda-repository-pin-600

wget "http://developer.download.nvidia.com/compute/cuda/"
"11.0.2/local_installers/cuda-repo-ubuntu1804-"
"11-0-local_11.0.2-450.51.05-1_amd64.deb"

sudo dpkg -i \
cuda-repo-ubuntu1804-11-0-local_11.0.2-450.51.05-1_amd64.deb

sudo apt-key add \
/var/cuda-repo-ubuntu1804-11-0-local/7fa2af80.pub

sudo apt-get -y install cuda
```

Une fois Cuda installé, tester immédiatement un fichier Numba utilisant le GPU, par exemple le suivant :

tester_gpu_simple.py

```
import numpy as np
from numba import cuda, int64

@cuda.jit('(int64[:], int64[:], int64[:])')
def cuda_sum(a, b, c):
    i = cuda.grid(1)
    c[i] = a[i] + b[i]

griddim = 5, 1
blockdim = 2, 1, 1
N = griddim[0] * blockdim[0]
cuda_sum_configured = cuda_sum.configure(griddim, blockdim)

a = np.array(np.random.randint(4, size=N), dtype=np.int64)
b = np.array(np.random.randint(4, size=N), dtype=np.int64)
c = np.empty_like(a)
print("a =", a)
print("b =", b)

cuda_sum_configured(a, b, c)

print('-'*(2*N+7))
print("c =", c)
```

qui affiche

```
a = [2 3 2 2 3 1 1 0 2 0]
b = [0 1 1 1 3 2 0 2 2 1]
--------------------------
c = [2 4 3 3 6 3 1 2 4 1]
```

Cela signifie que désormais, Numba peut s'exécuter sur le GPU de votre machine et avec les dernières versions.

Linux : autres méthodes pour installer les pilotes Nvidia

D'autres méthodes que celles présentées ci-dessus sont expliquées ici : How to install Nvidia driver in Ubuntu 18.04? en particulier si vous avez besoin de désinstaller un pilote Nvidia pas assez à jour. La méthode utilisant `autoinstall` est particulièrement simple et fonctionne.

Vous verrez parfois dans des forums ou des documentations des commandes d'installation du type :

```
sudo apt-get install nvidia-driver-xxxxx
```

sans que le xxxx soit précisé ni expliqué. D'après ce que j'ai compris, il s'agit de la version Nvidia du pilote (en juillet 2020, cela doit être 450). Cette valeur dépend des spécifications de votre carte et de la version d'Ubuntu. Le tableau ICI permet de savoir comment choisir. Par exemple, pour une Ubuntu 19.10, vous écrirez

```
sudo apt-get install nvidia-driver-390
```

Il est possible d'installer ses pilotes directement depuis le site de Nvidia avec un fichier run ou un fichier `deb`.

Il est également possible et paraît-il plus simple d'installer l'environnement Cuda via Docker. Voir un tutoriel ICI

Problèmes de compatibilité entre Cuda et le pilote matériel

Cuda est aussi un environnement de programmation et d'exécution, il est donc amené à évoluer même si votre carte graphique ne change pas. Votre version du Cuda toolkit doit être adaptée à votre pilote matériel. Chaque version du Cuda toolkit suppose une certaine version minimale de votre pilote graphique. Si votre pilote graphique Nvidia a été installé assez longtemps, il se peut qu'il ne fonctionne pas avec votre version de Cuda Toolkit. Toutefois, les programmes utilisant Cuda sont rétro-compatibles. Ainsi, la commande

```
$ nvidia-smi

+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.57       Driver Version: 450.57       CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
```

montre que le pilote matériel installé (n° 450) peut supporter au maximum les possibilités de Cuda 11.0. Pour utiliser Cuda 11.5, il faudra mettre à jour le pilote, cf. explication ICI.

Pour plus de détails, voir

— la documentation officielle ICI, la sous-section **Cuda driver**

— ce document plus précis CUDA Compatibility and Upgrades
— cette version vulgarisée.

Chapitre V

Programmation en Numba sur le GPU

Modèle de programmation Cuda

La vidéo High-Performance Computing with Python: Numba and GPUs de juillet 2019 fournit une introduction pédagogique à la programmation GPU qu'il pourra être bénéfique de regarder avant d'entrer dans les détails du modèle de programmation CUDA.

Le jargon a imposé les termes suivants que j'utiliserai parfois :

— *device* ie *périphérique* au lieu de GPU

— *host* ie *hôte* pour le CPU.

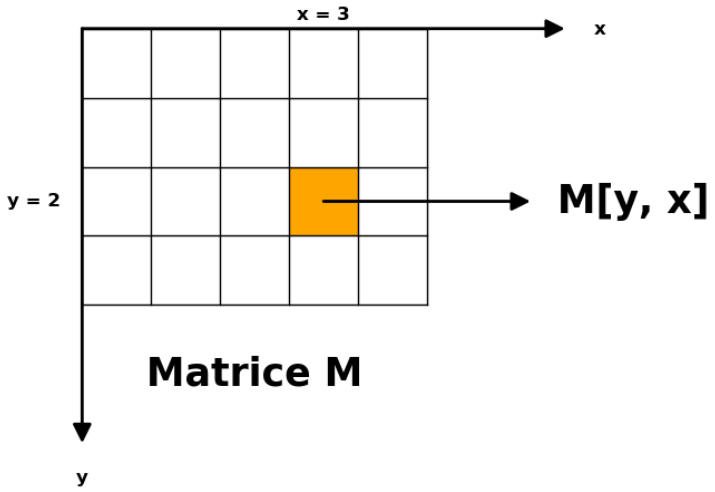
Cela est dû au fait qu'il y a des échanges entre le CPU et le GPU.

Un thread sur GPU est un flot d'instructions exécutées par un cœur GPU, un cœur pouvant exécuter, par exemple, 10 threads. Les threads vont exécuter en parallèle un même kernel, un kernel étant une simple fonction que l'utilisateur écrit à partir des besoins de son programme.

Description

Dans le modèle de programmation Cuda, les threads sont réprésentés par une grille constituée de blocs de threads. La grille et les blocs sont de même dimension `d` où `d` est parmi 1, 2 ou 3. Le principe de tout ce qui va être expliqué ci-dessous est le même pour chacune des dimensions `d`. Puisque les deux exemples qui seront présentés dans ce document calculeront sur le GPU des coefficients d'une matrice, il sera plus commode de choisir $\texttt{d} = 2$ mais $\texttt{d} = 1$ ou $\texttt{d} = 3$ pourrait s'adapter à nos exemples.

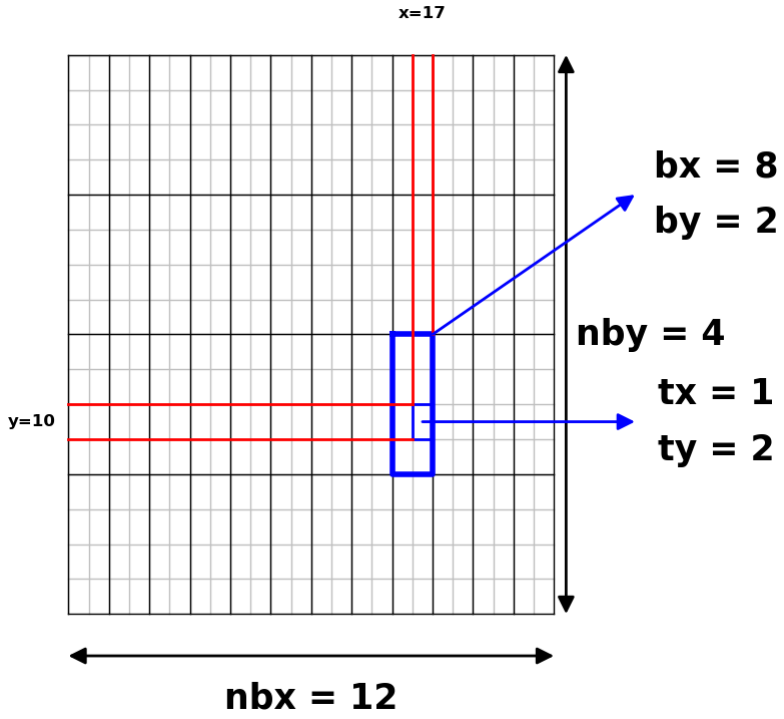
Dans le cas de la dimension 2, attention que l'ordre habituel (`ligne`, `colonne`) dans lequel on se représente un tableau 2D se traduit par `(ordonnée, abscisse)` :



— une coordonnées en largeur est souvent notée en utilisant la lettre x ; une largeur correspond à un écart de colonnes

— une coordonnées en hauteur est souvent notée en utilisant la lettre y ; une hauteur correspond à un écart de lignes.

Entrons dans le détail :



— La grille est constituée de blocs à deux dimensions, nbx blocs en largeur, nby blocs en hauteur

— Chaque bloc de la grille est identifié par les deux indices

  — l'indice bx de colonne dans la grille avec $0 \leq \texttt{bx} < \texttt{nbx}$,

  - l'indice by de ligne dans la grille avec $0 \leq \texttt{by} < \texttt{nby}$.
- Chaque bloc est constitué de threads.
- Chaque bloc a deux dimensions ntx et nty et tous les blocs de la grille ont mêmes dimensions
- Chaque thread d'un bloc est référencé avec deux indices tx et ty où $0 \leq \texttt{tx} < \texttt{ntx}$ et où $0 \leq \texttt{ty} < \texttt{nty}$
- Chaque bloc contient donc le même nombre de threads à savoir $\texttt{ntx} \times \texttt{nty}$.
- Le nombre de threads par bloc est matériellement limité, par exemple à 1024 ; on peut donc envisager des blocs de 32 x 32 threads.
- Si on voit la grille comme une grille de threads (et non de blocs) chaque thread a deux indices absolus :
  - l'indice de colonne : $\texttt{x} = \texttt{bx} \times \texttt{ntx} + \texttt{tx}$,
  - l'indice de ligne : $\texttt{y} = \texttt{by} \times \texttt{nty} + \texttt{ty}$.

Dans le dessin ci-dessus, on a une grille de $\texttt{nbx} \times \texttt{nby} = 12 \times 4$ blocs. Chaque bloc se décompose en $\texttt{ntx} \times \texttt{nty} = 2 \times 4$ threads. On a isolé, en bleu, dans le bloc `(bx, by) = (8, 2)` le thread `(tx, ty) = (1, 2)`. La position absolue de ce thread est `(x, y) = (17, 10)`.

Utlisation du modèle pour le calcul matriciel

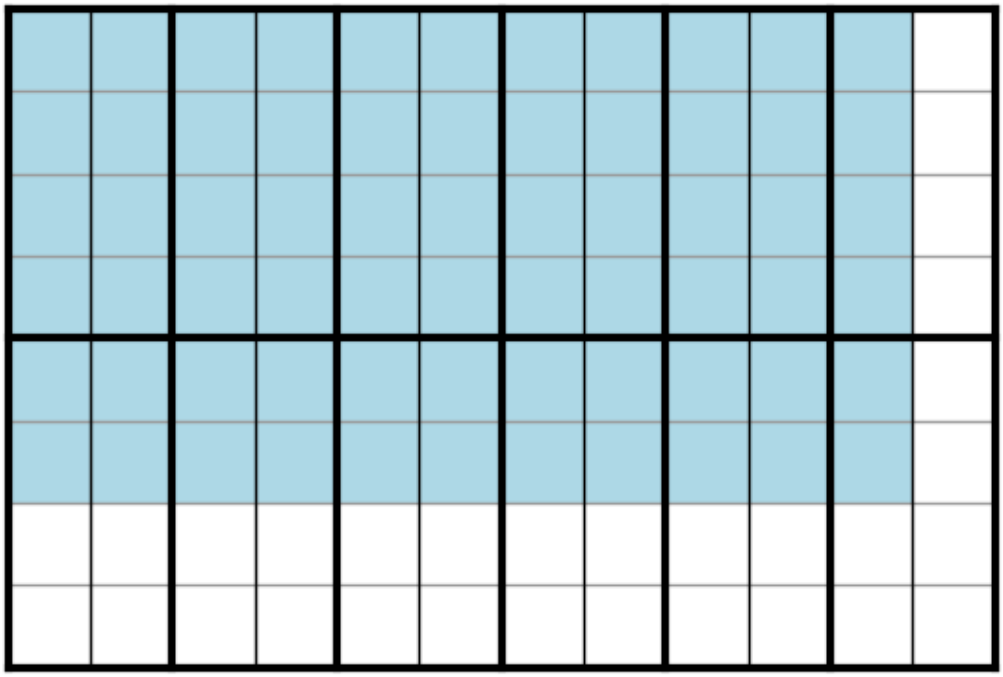
Imaginons qu'on veuille effectuer le produit matriciel $\texttt{A} \times \texttt{B} =: \texttt{C}$ où

- A est une matrice de taille lignes x colonnes $(\texttt{n}, \texttt{p})$,
- B est une matrice de taille lignes x colonnes $(\texttt{p}, \texttt{q})$,
- C est donc une matrice de taille lignes x colonnes $(\texttt{n}, \texttt{q})$.

Pour fixer les idées, cf. figure ci-dessous où la matrice C est en bleu, on va supposer que

$$\texttt{n} = 6, \texttt{p} = 8, \texttt{q} = 11$$

en sorte que C est de largeur 11 et de hauteur 6. Dans le modèle, on **recouvre** la matrice C à calculer par une grille de blocs de threads. Pour fixer les idées, on va choisir chaque bloc de taille `dimt= (2, 4)` donc chaque bloc contient $2 \times 4 = 8$ threads :



On posera $\texttt{ntx} = 2$ et $\texttt{nty} = 4$. Calculons le nombres de blocs en largeur et en hauteur pour recouvrir la matrice `C` :

— chaque bloc à une largeur de $\texttt{ntx} = 2$ et $\texttt{q} = 11$ donc il faut en largeur $\texttt{nbx} = 6$ blocs (et il y aura $6 \times 2 - 11 = 1$ thread inutile en largeur) ;

— chaque bloc à une hauteur de $\texttt{nty} = 4$ et $\texttt{n} = 6$ donc il faut en hauteur $\texttt{nby} = 2$ blocs (et il y aura $4 \times 2 - 6 = 2$ threads inutiles en hauteur).

Comme on le voit sur la figure, la grille est formée de 2×6 blocs.

La grille ainsi constituée a une largeur de 12 threads et une hauteur de 8 threads et donc peut exécuter 96 threads, ce qui excède le nombre total des 66 coefficients de `C` à calculer.

Initialement, les matrices `A` et `B` ainsi qu'une matrice `C` nulle et à la bonne taille sont définies sur l'hôte. Ces matrices sont envoyées sur le device. Pour calculer la matrice `C`, il y aura au moins autant de threads sur le GPU que de coefficients de `C` à calculer. Dans notre exemple, il y aura 96 threads indexés $(\texttt{x}, \texttt{y})$ avec $0 \leq \texttt{x} < 12$ et $0 \leq \texttt{y} < 8$.

Quand on lance un kernel, cf. les exemples de code plus bas, outre les données utiles au calcul, ici `A`, `B` et `C`, on lui donne le couple $\texttt{dimb} = (\texttt{nbx}, \texttt{nby})$ et le couple $\texttt{dimt} = (\texttt{ntx}, \texttt{nty})$. Par ailleurs, quand ce kernel s'exécute, il sait dans quel bloc $(\texttt{bx}, \texttt{by})$ de la grille il s'exécute et quel thread $(\texttt{tx}, \texttt{ty})$ du bloc il exécute. Cela lui permet de savoir dans quel thread $(\texttt{x}, \texttt{y})$ de la grille il s'exécute. Et notre kernel va être écrit en sorte que l'élément en colonne `x` et ligne `y` soit représenté par le thread indexé $(\texttt{x}, \texttt{y})$.

En pratique, il est fréquent que les blocs de threads soient « carrés », ie que $\texttt{ntx} = \texttt{nty}$.

Références

Le modèle de programmation Cuda est décrit en détail dans un document officiel : Cuda programming model.

Plus généralement, concernant la programmation Cuda, on pourra consulter l'ouvrage progressif et détaillé CUDA C Programming paru en 2014.

Illustration par le produit matriciel

Le modèle de programmation Cuda étant décrit, nous disposons de tous les éléments pour comprendre comment on met en œuvre un produit matriciel basique sur GPU. Les matrices sont représentées par des tableaux Numpy. Voici un tel programme :

```
import numpy as np
from numba import cuda

@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def cu_square_matrix_mul(A, B, C):

    n, p=A.shape
    _, q=B.shape

    x, y = cuda.grid(2)

    if x >= q or y >= n:
        return

    for i in range(p):
        C[y, x] += A[y, i] * B[i, x]

def make_data(n, p, q, bound):
    A = np.random.randint(low=-bound, high=bound+1,
                          size=(n, p), dtype=np.int64)
    B = np.random.randint(low=-bound, high=bound+1,
                          size=(p, q), dtype=np.int64)
    C = np.zeros((n, q), dtype=np.int64)
    return A, B, C

def dims(n, q, nt):
    ntx, nty = nt

    # nb colonnes de blocs
    nbx=q//ntx+(q%ntx)

    # nb lignes de blocs
    nby=n//nty+(n%nty)

    return (nbx, nby)

def cuda_mult(A, B, C, nb, nt):

    dA = cuda.to_device(A)
```

```
    dB = cuda.to_device(B)
    dC = cuda.to_device(C)

    cu_square_matrix_mul[nb, nt](dA, dB, dC)
    cuda.synchronize()

    dC.to_host()

def main():
    bound=10
    n=6
    p=8
    q=11

    A, B, C=make_data(n, p, q, bound)

    dimtx=2
    dimty=4

    dimt=(dimtx, dimty)
    dimb = dims(n, q, dimt)

    cuda_mult(A, B, C, dimb, dimt)
    print((C))
    print()
    print(A@B)

main()
```

```
[[  64  -84  132  -34  -12   -8   64 -261   97   -3  -82]
 [-188  -71  -37   87 -200  169 -152   23  142  -83  148]
 [   2  -35  142  -26   15  -89  -41 -109  -66  -83  -60]
 [ 202   13   44  -18  126  -72  157 -249   82  102 -111]
 [ -44  -15  -35   59   86   -3   33   -3  -23   31   54]
 [ 252   63  -73   49  -67   76   65 -167    0  -71  147]]

[[  64  -84  132  -34  -12   -8   64 -261   97   -3  -82]
 [-188  -71  -37   87 -200  169 -152   23  142  -83  148]
 [   2  -35  142  -26   15  -89  -41 -109  -66  -83  -60]
 [ 202   13   44  -18  126  -72  157 -249   82  102 -111]
 [ -44  -15  -35   59   86   -3   33   -3  -23   31   54]
 [ 252   63  -73   49  -67   76   65 -167    0  -71  147]]
```

La sortie montre le calcul du produit matriciel d'abord effectué par Cuda et d'autre part, effectué par Numpy. On observe que les résultats coïncident bien. On reprend l'exemple du produit `C` d'une matrice A de taille $(6, 8)$ et d'une matrice B de taille $(8, 11)$.

Le kernel

Il est déclaré ligne 5. Il s'exécute sur le device. Il reçoit les matrices `A`, `B` et `C` qui se trouvent sur le device. La fonction est décorée ligne 4 par le décorateur `cuda.jit`. C'est ce décorateur qui indique à Numba que la fonction est un kernel. On passe à ce décorateur, sous forme de chaîne de caractères, ici

```
'(int64[:,:], int64[:,:], int64[:,:])'
```

un triplet de types de données transmises au kernel. Ici `int64[:,:]` désigne un tableau 2d d'entiers sur 64 bits. On trouvera quelques indications sur des signatures courantes ici : signature specifications.

Les kernels s'exécutent en parallèle. Chaque kernel calcule à la ligne 16 le coefficient de `C` à la ligne `y` et à la colonne `x` de la grille. Ce calcul résulte tout simplement du produit ligne par colonne, effectué dans le kernel par la boucle for de la ligne 15.

Le code ci-dessus ne le montre pas, mais le kernel quand il s'exécute, connait le bloc et le thread d'exécution ; par ailleurs, suite à l'appel ligne 43, le kernel dispose des dimensions de chaque bloc (le `dimt`) et des dimensions de la grille en blocs (le `dimb`). Donc, à l'aide d'une formule assez simple, le kernel est en mesure de calculer la position absolue $(\mathtt{x}, \mathtt{y})$ du thread dans la grille. Pour nous épargner le calcul, Numba met à disposition ligne 10 une fonction `cuda.grid` qui calcule cette position. Il faut donner à cette fonction la dimension des blocs, dans notre exemple, ce sont les 2 dimensions (le $\mathtt{d} = 2$ expliqué dans la description du modèle Cuda) mais on avait vu en introduction du modèle de programmation Cuda qu'il serait possible de choisir des dimensions 1 ou 3.

On a vu que la grille de threads déborde de la matrice `C`. Donc, le kernel doit d'assurer (cf. ligne 12) que le calcul effectué en $(\mathtt{x}, \mathtt{y})$ correspond à une position légitime de la matrice `C`.

Un kernel ne renvoie rien, il agit par effet de bord.

Dimensions de la grille

Le nombre de threads qu'un bloc peut exécuter est plafonné. Par exemple, pour la carte graphique que j'ai utilisée, il est de 1024. Dans ce cas et si les blocs sont de dimension 2, il est courant de choisir un bloc de taille 32×32. Dans notre petit exemple, on a choisi une taille $(2, 4)$, cf. lignes 56-57. Quand on connait les dimensions de chaque blocs, il est facile de calculer le nombre de blocs suivant chaque dimension pour recouvrir la matrice. C'est ce que fait la fonction `dims(n, q, nt)`. L'idée du calcul par les formules lignes 30 et 33 a été donné lors de la description de l'exemple où on avait trouvé une grille de dimension de $(2, 6)$. Plus loin dans ce document, un aparté explique comment trouver cette formule et lui donner la forme habituellement rencontrée.

Calcul du produit matriciel

Les étapes sont les suivantes :

— lignes 19-20 et 21-22 : sur le host, on génère aléatoirement des matrices Numpy `A` et `B` à coefficients entre -10 et 10 et de type `int64` ainsi qu'une matrice Numpy `C` initialisée à 0

— lignes 39-41 : on déplace les 3 matrices sur le device ; Numba se charge pour nous de l'allocation mémoire et du transfert.

— ligne 43 : on lance le kernel en donnant tous les éléments utiles : les 3 matrices, bien sûr, et aussi, entre crochets, les dimensions de chaque bloc et de la grille.

— ligne 44 : la synchronisation va avoir pour effet que l'hôte va interrompre son fil d'exécution jusqu'à ce que **tous** les threads exécutant le kernel aient terminé.

— ligne 46 : copie la matrice depuis le device vers le host.

— lignes 63 et 65 : on peut alors accéder au résultat et par exemple comparer.

Variante d'écriture du kernel

Le kernel écrit dans le programme ci-dessus :

```
@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def cu_square_matrix_mul(A, B, C):

    n, p=A.shape
    _, q=B.shape

    x, y = cuda.grid(2)

    if x >= q or y >= n:
        return

    for i in range(p):
        C[y, x] += A[y, i] * B[i, x]
```

est parfois écrit de manière moins directe. En effet, Numba met à notre disposition la fonction `cuda.grid` mais il est possible de calculer explicitement `x` et `y`. D'où le code suivant :

```
@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def cu_square_matrix_mul(A, B, C):

    n, p=A.shape
    _, q=B.shape

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh

    if x >= q or y >= n:
        return

    for i in range(p):
        C[y, x] += A[y, i] * B[i, x]
```

En effet, chaque thread connait la position de son bloc dans la grille. On y a accède de la manière suivante :

```
bx = cuda.blockIdx.x
by = cuda.blockIdx.y
```

Il en est de même pour le thread dans son bloc :

```
tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
```

Le thread connaît aussi la dimension `nb` de chaque bloc car il lui a été fournie entre crochets au moment du lancement du kernel :

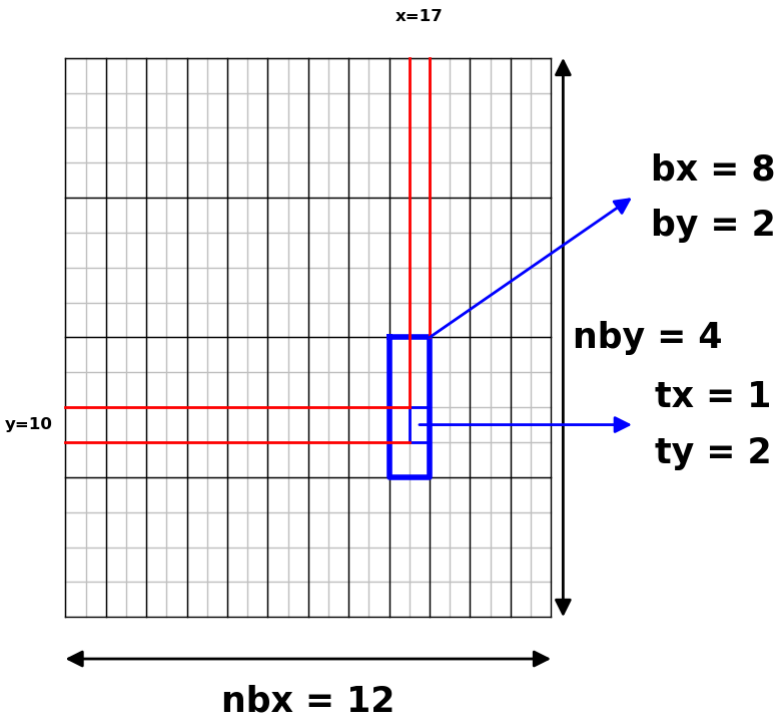
```
cu_square_matrix_mul[nb, nt](dA, dB, dC)
```

Depuis le kernel, on y accède de la manière suivante :

```
bw = cuda.blockDim.x
bh = cuda.blockDim.y
```

autrement dit, on a `nb = (bw, bh)`.

À partir de là, et vue la structure de la grille



on en déduit que

```
x = tx + bx * bw
y = ty + by * bh
```

Produit matriciel : performances

Il est temps de regarder combien notre calcul de produit de matrices sur GPU améliore ce que Python nous propose par défaut, à savoir la multiplication de Numpy. Pour cela on va calculer le produit de deux matrices carrées de taille 2500 :

```
from time import time
import numpy as np
from numba import cuda, int64

@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def kernel_mat_mult(A, B, C):

    n, p=A.shape
    q=B.shape[1]

    x, y = cuda.grid(2)

    if x >= q or y >= n:
        return

    for i in range(p):
        C[y, x] += A[y, i] * B[i, x]

def make_data(n, p, q, bound):
    A = np.random.randint(low=-bound, high=bound+1,
                          size=(n, p), dtype=np.int64)
    B = np.random.randint(low=-bound, high=bound+1,
                          size=(p, q), dtype=np.int64)
    C = np.zeros((n, q), dtype=np.int64)
    return A, B, C

def dims(n, q, nt):
    ntx, nty = nt
    # nb colonnes de blocs
    nbx=q//ntx+(q%ntx)

    # nb lignes de blocs
    nby=n//nty+(n%nty)
    return (nbx, nby)

def cuda_mult(A, B, C, nb, nt):
    dA = cuda.to_device(A)
    dB = cuda.to_device(B)
    dC = cuda.to_device(C)
    kernel_mat_mult[nb, nt](dA, dB, dC)
    cuda.synchronize()
    dC.to_host()
```

```
def host_mult(A, B):
    return A @ B

def main():
    bound=10

    n=p=q=2500

    dstart = time()
    A, B, C=make_data(n, p, q, bound)
    dend = time()
    tcuda = dend - dstart
    print('Data: %.2fs' %tcuda)

    dimtx=32
    dimty=32

    dimt=(dimtx, dimty)
    dimb = dims(n, q, dimt)

    dstart = time()
    cuda_mult(A, B, C, dimb, dimt)
    dend = time()
    tcuda = dend - dstart
    print('Cuda: %.2fs' % tcuda)

    hstart = time()
    P=host_mult(A, B)
    hend = time()
    thost = hend - hstart
    print('Numpy:  %.2fs' % thost)

    print("Check:", np.array_equal(P, C))

main()
```

```
Data: 0.13s
Cuda: 0.63s
Numpy:  27.47s
Check: True
```

L'amélioration est d'un facteur de l'ordre de 40. Noter toutefois que le temps de Numpy n'est pas un bon temps de référence, étant très lent pour des matrices à *coefficients entiers*. Un code écrit en C++ en utilisant des vecteurs de la STL met environ 9s, ce qui donne finalement seulement une amélioration d'un facteur 15.

Activité du GPU pendant un calcul

Grâce à l'outil `nvidia-smi`, on peut visualiser l'état instantané du GPU (processeurs et mémoire). Voici par exemple son état pendant un gros calcul matriciel :

```
$ nvidia-smi
Tue Jul 28 15:11:00 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.05    Driver Version: 450.51.05    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GeForce GTX 970     On   | 00000000:01:00.0 Off |                  N/A |
| 78%   76C    P2   118W / 173W |   3688MiB /  4041MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

(affichage partiel) où on voit l'occupation des threads et l'état de la mémoire. Ci-dessous, la suite de l'affichage avec l'occupation par processus, en particulier le processus python3 de la dernière ligne :

```
+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1038      G   /usr/lib/xorg/Xorg                 61MiB |
|    0   N/A  N/A      1548      G   /usr/lib/xorg/Xorg                128MiB |
|    0   N/A  N/A      1749      G   /usr/bin/gnome-shell              123MiB |
|    0   N/A  N/A      3858      G   nvidia-settings                     3MiB |
|    0   N/A  N/A      4031      C   python3                          3353MiB |
+-----------------------------------------------------------------------------+
```

Et voici, l'affichage une fois le calcul matriciel terminé :

```
po@po-20-04:~$ nvidia-smi
Tue Jul 28 15:11:12 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.05    Driver Version: 450.51.05    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GeForce GTX 970     On   | 00000000:01:00.0 Off |                  N/A |
| 78%   71C    P0    81W / 173W |    332MiB /  4041MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

```
+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1038      G   /usr/lib/xorg/Xorg                 61MiB |
|    0   N/A  N/A      1548      G   /usr/lib/xorg/Xorg                128MiB |
|    0   N/A  N/A      1749      G   /usr/bin/gnome-shell              123MiB |
|    0   N/A  N/A      3858      G   nvidia-settings                     3MiB |
+-----------------------------------------------------------------------------+
po@po-20-04:~$
```

Hélas, la vue fournie n'est qu'instantanée, il faut relancer nvidia-smi pour toute nouvelle vue.

Sous Linux, il existe une application graphique permettant de suivre l'occupation du GPU mais elle est compliquée à installer (il faut compiler les sources depuis Github).

Améliorer les performances en utilisant la mémoire partagée

Ci-dessous, nous allons calculer le produit de deux matrices carrées de grande taille, environ 10000. Plus précisément, on va choisir 10112 qui est le premier multiple de 32 au-dessus de 10000. Ne pas placer une matrice trop grande, vous pourriez avoir une erreur pour mémoire insuffisante :

```
CUDA_ERROR_OUT_OF_MEMORY
```

Le code Cuda de produit matriciel déjà écrit utilisait la mémoire globale du GPU. Voici son exécution pour $n = 10112$

```
from time import time
import numpy as np
from numba import cuda, int64

@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def kernel_mat_mult(A, B, C):

    n, p=A.shape
    q=B.shape[1]

    x, y = cuda.grid(2)

    if x >= q or y >= n:
        return

    for i in range(p):
        C[y, x] += A[y, i] * B[i, x]

def make_data(n, p, q, bound):
    A = np.random.randint(low=-bound, high=bound+1,
                          size=(n, p), dtype=np.int64)
```

```
    B = np.random.randint(low=-bound, high=bound+1,
                          size=(p, q), dtype=np.int64)
    C = np.zeros((n, q), dtype=np.int64)
    return A, B, C

def dims(n, q, nt):
    ntx, nty = nt
    # nb colonnes de blocs
    nbx=q//ntx+(q%ntx)

    # nb lignes de blocs
    nby=n//nty+(n%nty)
    return (nbx, nby)

def cuda_mult(A, B, C, nb, nt):
    dA = cuda.to_device(A)
    dB = cuda.to_device(B)
    dC = cuda.to_device(C)
    kernel_mat_mult[nb, nt](dA, dB, dC)
    cuda.synchronize()
    dC.to_host()

def main():
    bound=10

    n=p=q=10112

    dstart = time()
    A, B, C=make_data(n, p, q, bound)
    dend = time()
    tcuda = dend - dstart
    print('Data: %.2fs' %tcuda)

    dimtx=32
    dimty=32

    dimt=(dimtx, dimty)
    dimb = dims(n, q, dimt)

    dstart = time()
    cuda_mult(A, B, C, dimb, dimt)
    dend = time()
    tcuda = dend - dstart
    print('GPU global: %.2fs' % tcuda)

main()
```

```
Data: 2.07s
GPU global: 37.89s
```

Le même programme exécuté sur Google Colab :

```
from numba import cuda
cuda.detect()
```

```
Found 1 CUDA devices
id 0            b'Tesla K80'                                [SUPPORTED]
                      compute capability: 3.7
                           pci device id: 4
                              pci bus id: 0
Summary:
        1/1 devices are supported
True
```

affiche un temps de :

```
Data: 3.35s
GPU global: 46.11s
```

La documentation de Numba précise que la version présentée ci-dessus est peu performante car elle utilise la mémoire dite *globale* du GPU qui est lente. Elle suggère une implémentation plus rapide qui utilise la mémoire partagée (*shared memory*) du GPU. C'est une mémoire, un peu analogue au cache d'un CPU, en quantité limitée mais mais dont le temps d'accès est plus court que pour la mémoire globale. Elle est partagée par tous les threads du même bloc. Pour plus d'information sur la différence entre ces deux types de mémoire, on pourra consulter : why shared memory is faster than global memory?.

Le programme ci-dessous est un code simplifié de celui donné en exemple dans le code-source de Numba. Pour le produit de deux matrices de taille 10112, le temps est presque divisé par trois. Voici le code :

```
1  from time import time
2  import numpy as np
3  from numba import cuda, int64
4
5  BLOCK = 32
6  BOUND = 10
7
8  @cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
9  def kernel_mat_mult_shared(A, B, C):
10
11     sA = cuda.shared.array(shape=(BLOCK, BLOCK), dtype=int64)
12     sB = cuda.shared.array(shape=(BLOCK, BLOCK), dtype=int64)
13
14     n, _ = A.shape
15
16     bpg = n // BLOCK
17
18     x, y = cuda.grid(2)
19
20     tx = cuda.threadIdx.x
21     ty = cuda.threadIdx.y
```

```
22
23      acc = 0
24
25      for i in range(bpg):
26          sA[ty, tx] = A[y, tx + i * BLOCK]
27          sB[ty, tx] = B[ty + i * BLOCK, x]
28
29          cuda.syncthreads()
30
31          for j in range(BLOCK):
32              acc += sA[ty, j] * sB[j, tx]
33
34          cuda.syncthreads()
35
36          C[y, x] = acc
37
38  def make_data(bpg):
39      n = BLOCK * bpg
40
41      A = np.random.randint(low=-BOUND, high=BOUND + 1,
42                            size=(n, n), dtype=np.int64)
43      B = np.random.randint(low=-BOUND, high=BOUND + 1,
44                            size=(n, n), dtype=np.int64)
45      C = np.empty_like(A)
46      return A, B, C
47
48  def mat_mult_shared(A, B, C, bpg):
49      dA = cuda.to_device(A)
50      dB = cuda.to_device(B)
51      dC = cuda.to_device(C)
52
53      kernel_mat_mult_shared[(bpg, bpg), (BLOCK, BLOCK)](dA, dB, dC)
54      cuda.synchronize()
55
56      dC.to_host()
57      return C
58
59  def go(bpg):
60
61      n = bpg * BLOCK
62
63      start = time()
64      A, B, C = make_data(bpg)
65      end = time()
66      print("Dataset : %.2fs" % (end - start))
67
68      dstart = time()
69      mat_mult_shared(A, B, C, bpg)
70      dend = time()
```

```
71
72      print("GPU shared : %.2fs" % (dend - dstart))
73
74      A=np.array(A, dtype=np.float64)
75      B=np.array(B, dtype=np.float64)
76
77      dstart = time()
78      D = A@B
79      dend = time()
80
81      print("Numpy : %.2fs" % (dend - dstart))
82      D=np.array(D, dtype=np.int64)
83
84      print(np.array_equal(C, D))
85
86  bpg = 316
87  n = BLOCK * bpg
88  print("n =", n)
89  go(bpg)
```

qui affiche

```
n = 10112
Dataset : 2.07s
GPU shared : 13.06s
Numpy : 12.11s
True
```

Deux observations concernant le temps d'exécution :

— l'utilisation de la mémoire partagée a divisé le temps d'exécution par presque 3 ;

— Numpy a réussi à calculer le produit ! mais en passant par un calcul en flottant et le temps est encore meilleur que le temps sur GPU avec mémoire partagé (cette performance remarquable avait été évoquée en 2e partie du document).

Sur le même GPU de Google Colab que ci-dessus, le temps d'exécution est :

```
n = 10112
Dataset : 2.72s
GPU shared : 14.54s
Numpy : 57.97s
True
```

Le ratio pour le GPU est à peu près le même, en revanche, le temps d'exécution de Numpy n'est pas du tout dans le même rapport (il est possible qu'il ne soit exécuté que sur un seul thread).

Décrivons le code Numba ci-dessus. La matrice `C` $=$ `A` $\times$ `B` est découpée est blocs de 32×32, dans le code ci-dessus, `BLOCK`$=32$. Il y a `bpg` $= 316$ blocs (`bpg` pour *block per grid*) d'où une taille de matrice `n` $\times$ `n` où `n` $= 32 \times 316 = 10112$. Ce qui accélère le calcul est que des blocs des matrices A et B sont placés en mémoire partagée et qu'ensuite, les produits ligne par colonnes sont effectués en utilisant la mémoire partagée. Plus précisément, chaque kernel crée deux tableaux `sA` et `sB` dans la mémoire partagée (lignes 11-12) chacun de taille `BLOCK \times BLOCK`. Le thread en cours d'exécution calcule le coefficient de C en position `(ty, tx)` de son bloc de

thread (lignes 20-21). Pour cela, il place dans la mémoire partagée :

— la valeur du coefficient de `A` situé dans le bloc en colonne `i` et en position (`ty`, `tx`) dans le tableau `sA` (ligne 26)
— la valeur du coefficient de `B` situé dans le bloc en ligne `i` et en position (`ty`, `tx`) dans le tableau `sB` (ligne 27)

Tous les threads du même bloc de la grille s'éxécutent simultanément donc

— le bloc situé à la même ligne de bloc que le bloc que le kernel exécute et à la colonne de bloc `i` de la matrice `A` est exactement contenu dans le tableau `sA`
— le bloc situé à la même colonne de bloc que le bloc que le kernel exécute et à la ligne de bloc `i` de la matrice `B` est exactement contenu dans le tableau sB.

Avant que la suite ne s'exécute (les produits ligne par colonne), il faut être certain que tous les threads aient recopié les valeurs. C'est pour cela qu'est écrite ligne 29 une instruction de synchronisation de ces threads. Cela correspond à ce qu'on appelle une **barrière** en programmation multithread.

Ensuite, est effectué le calcul ligne par colonne (ligne 32) qui lui se fait en mémoire partagée. Mathématiquement, le produit matriciel est en fait réalisé comme un produit par blocs.

Noter que le produit ligne par colonne ne fait pas intervenir que les coefficients des tableaux en mémoire partagé définis dans le kernel lui mais aussi dans les autres kernels du même bloc. C'est pour ça qu'il y a une instruction de synchronisation ligne 34

Noter également, biene que ce soit un problème d'une autre nature, qu'il n'est pas possible de définir dynamiquement les dimensions d'un tableau en mémoire partagée (lignes 11-12), elles doivent être connues à la compilation. Ainsi un code tel que

```
@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def cuda_square_mat_mult(A, B, C):
    block = 32
    sA = cuda.shared.array(shape=(block, block), dtype=int64)
```

engendrera une erreur à la compilation.

Selon mes propres essais, il ne semble pas nécessaire de faire une étape de précompilation (warmup) comme on fait avec Numba en mode jit.

Avec quelques précautions, le code ci-dessus pourrait être modifié pour s'appliquer à des matrices de taille qui ne soit pas un multiple de 32, par exemple 10000.

Un autre algorithme ?

Pour information, le produit de deux matrices de taille 10000 calculé en un seul processus par un algorithme de Strassen optimisé met plus de 3 min 30 s. Voici le code sous SageMath :

```
# SageMath et lib flint2

n=10000
p=10000
q=10000

start=time()
A=random_matrix(ZZ,n,p, x=-10, y=10)
```

```
B=random_matrix(ZZ,p,q, x=-10, y=10)
delta = time() - start

print(f"Durée : {delta:.2f}s")



start=time()
A*B
delta = time() - start

print(f"Durée : {delta:.2f}s")
```

```
Durée : 18.86s
Durée : 219.95s
```

Installer Cupy et comparer

Cupy est un module Python qui met à disposition des bibliothèque Cuda en Python. On peut facilement installer Cupy sur Google Colab. On commence par chercher la version de Cuda disponible :

```
!nvcc -V
```

qui affiche

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Sun_Jul_28_19:07:16_PDT_2019
Cuda compilation tools, release 10.1, V10.1.243
```

en sorte qu'on peut essayer d'installer la version de Cupy adaptée à Cuda 10.1 :

```
!pip install cupy-cuda101
```

qui produit l'affichage (légèrement remanié) suivant

```
Collecting cupy-cuda101
  Downloading https://files.pythonhosted.org/packages/\
  d6/0d/37b5e06d198778b3c8163a4ee01a64bf0832300c5a738f85c7ae5ada4fb7/\
  cupy_cuda101-7.7.0-cp36-cp36m-manylinux1_x86_64.whl (366.5MB)
     |████████████████████████████████████████████████████████████████|
     366.5MB 42kB/s

Requirement already satisfied: numpy>=1.9.0 in
/usr/local/lib/python3.6/dist-packages (from cupy-cuda101) (1.18.5)

Requirement already satisfied: fastrlock>=0.3 in
/usr/local/lib/python3.6/dist-packages (from cupy-cuda101) (0.5)

Requirement already satisfied: six>=1.9.0 in
```

```
/usr/local/lib/python3.6/dist-packages (from cupy-cuda101) (1.15.0)

Installing collected packages: cupy-cuda101
Successfully installed cupy-cuda101-7.7.0
```

Le GPU disponible est :

```
from numba import cuda
print(cuda.detect())
```

```
Found 1 CUDA devices
id 0    b'Tesla P100-PCIE-16GB'                              [SUPPORTED]
                      compute capability: 6.0
                           pci device id: 4
                              pci bus id: 0
Summary:
        1/1 devices are supported
True
```

Il ne reste plus qu'à tester le temps d'exécution du produit de deux matrices :

```
from time import time
import numpy as np
from numba import cuda, int64
import cupy as cp

BLOCK = 32
BOUND = 10

@cuda.jit('(int64[:,:], int64[:,:], int64[:,:])')
def cuda_square_mat_mult(A, B, C):

    sA = cuda.shared.array(shape=(BLOCK, BLOCK), dtype=int64)
    sB = cuda.shared.array(shape=(BLOCK, BLOCK), dtype=int64)

    n, _ = A.shape

    bpg = n // BLOCK

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y

    acc = 0

    for i in range(bpg):
        if x < n and y < n:
            sA[ty, tx] = A[y, tx + i * BLOCK]
            sB[ty, tx] = B[ty + i * BLOCK, x]
```

```
        cuda.syncthreads()

        for j in range(BLOCK):
            acc += sA[ty, j] * sB[j, tx]

        cuda.syncthreads()

        if x < n and y < n:
            C[y, x] = acc


def make_data(bpg):
    n = BLOCK * bpg

    A = np.random.randint(low=-BOUND, high=BOUND + 1,
                          size=(n, n), dtype=np.int64)
    B = np.random.randint(low=-BOUND, high=BOUND + 1,
                          size=(n, n), dtype=np.int64)
    C = np.zeros(shape=(n, n), dtype=np.int64)
    return A, B, C

def square_mat_mult(A, B, C, bpg):
    dA = cuda.to_device(A)
    dB = cuda.to_device(B)
    dC = cuda.to_device(C)
    start = time()
    cuda_square_mat_mult[(bpg, bpg), (BLOCK, BLOCK)](dA, dB, dC)
    cuda.synchronize()
    end = time()
    # print("Inside Device : %.2fs" % (end - start))
    dC.to_host()
    return C

def cp_matmul(A, B):
  A=cp.asarray(A)
  B=cp.asarray(B)
  C = A @ B
  cp.cuda.Device().synchronize()
  return cp.asnumpy(C)


def test(bpg):
  n = BLOCK * bpg
  A, B, C=make_data(bpg)
  start = time()
  C=square_mat_mult(A, B, C, bpg)
  end = time()
  print("Numba : %.2fs" % (end - start))
```

```
  start = time()
  D=cp_matmul(A, B)
  end = time()

  print("Cupy : %.2fs" % (end - start))
  return C, D

bpg = 316

n = BLOCK * bpg
print("n =", n)

for _ in range(3):
  C, D=test(bpg)
  print(np.array_equal(C, D))
  print("----------------")
```

qui affiche

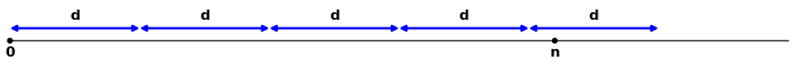
```
n = 10112
Numba : 5.67s
Cupy : 1.07s
True
----------------
Numba : 5.67s
Cupy : 1.07s
True
----------------
Numba : 5.64s
Cupy : 1.07s
True
----------------
```

On est dans un rapport de 1 à 6, c'est plutôt correct si on compare d'une part le code relativement naïf et peu optimisé écrit directement en Python et d'autre part un code de bibliothèque professionnelle optimisé et écrit en C++.

Aparté mathématique

Ci-dessous, je tente d'expliquer d'où vient le petit calcul que l'on rencontre dans les programmes ou la littérature, sorte d'idiome standard Cuda, pour calculer le nombre de blocs pour recouvrir une matrice.

Soient $n > 0$ et $d > 0$ des entiers. On cherche le plus petit entier $N \geq 0$ tel que $Nd \geq n$:



Par exemple, supposons $d = 10$. Alors si $n = 42$ on a $N = 5$ et si $n = 40$ alors $N = 4$

Autrement dit, on cherche à mettre bout à bout le moins de segments de longueur d possibles en partant de 0 pour que le dernier segment recouvre n. Définissons le quotient et le reste : $q = n//d$ et $r = n\%d$. Il est immédiat que :

- si $r = 0$ alors $N = q$
- si $r \neq 0$ alors $N = q + 1$

En Python ou en C++, on pourrait se contenter de la formule $N = q + (r! = 0)$.

Donnons cependant une formule plus mathématique, celle qui est donnée dans les documents Cuda. Vérifions que la formule $N = (n + d - 1)//d$ marche dans les deux cas :

- si $n = dq$ alors $n + d - 1 = dq + (d - 1)$ et comme $0 \leq d - 1 < d$ on a bien $(n + d - 1)//d = q$
- si $n = dq + r$ alors $n + d - 1 = d(q + 1) + (r - 1)$ et comme $0 \leq r - 1 < r < d$ on a bien $(n + d - 1)//d = q + 1$.

On peut aussi exprimer N en fonction de la partie entière supérieure : $N = \lceil \frac{n}{d} \rceil$. En effet, d'une manière générale, $\lceil x \rceil$ est le plus petit entier supérieur ou égal à x .

Alternative de démonstration

Ce qui suit n'a d'intérêt que du point de vue mathématique et peut-être ignoré. Il s'agit juste de montrer comment on pouvait trouver la formule directement.

Si $n \leq d$ alors $N = 1$. On suppose désormais que $n > d$ ce qui entraîne que $N \geq 1$. Alors comme N est le plus petit entier vérifiant la propriété, $N - 1$ ne la vérifie pas et comme $N - 1 \geq 0$, c'est que $(N - 1)d < n$

Montrons que N est forcément le plus grand entier qui vérifie $(N - 1)d < n$. En effet, puisque pour tout entier $M > N - 1$ et donc, puisque M et N sont des entiers, forcément tels que $M \geq N$, on a $Md \geq Nd \geq n$.

Réciproquement, si N est le plus grand entier qui vérifie $(N - 1)d < n$ alors forcément, comme $N + 1$ ne vérifie pas la condition, on a $Nd \geq n$ et N est forcément le plus petit entier qui vérifie cette propriété puisque si $M < N$ alors (inégalité entre entiers) $M \leq N - 1$ donc $Md \leq (N - 1)d < n$.

Donc, finalement, on cherche le plus grand entier N tel que $(N - 1)d < n$ ou encore, puisque c'est une inégalité entre entiers, tel que $(N - 1)d \leq n - 1$ donc $Nd \leq n + d - 1$ et on sait que le plus grand entier N qui vérifie cela est le quotient de la division entière $(n + d - 1)//d$ ce qui redonne le résultat vu plus haut.

Introspection Cuda

Les bibliothèques Cuda fournissent de nombreux outils d'introspection statique et dynamique (profilage) et de débogage. Ci-dessous, le programme fournit quelques données :

```
from numba.cuda.cudadrv import enums
from numba import cuda

from numba.cuda.cudadrv import enums
from numba import cuda

device = cuda.get_current_device()
```

```
print(device.name)

attribs= [name.replace("CU_DEVICE_ATTRIBUTE_", "") for name in
        dir(enums) if name.startswith("CU_DEVICE_ATTRIBUTE_")]
for attr in attribs:
        print(attr, '=', getattr(device, attr))4
```

dont voici seulement un extrait :

```
b'GeForce GTX 970'

...

MAX_BLOCK_DIM_X = 1024
MAX_BLOCK_DIM_Y = 1024
MAX_BLOCK_DIM_Z = 64
MAX_GRID_DIM_X = 2147483647
MAX_GRID_DIM_Y = 65535
MAX_GRID_DIM_Z = 65535
MAX_THREADS_PER_BLOCK = 1024
```

Par exemple, comme signalé plus haut, le nombre maximal de threads par blocs (1024). Ou encore les trois dimensions maximales de la grille, le nombre maximal de threads suivant l'axe des x, etc.

Je n'ai pas réussi à déterminer par ce moyen la mémoire disponible sur le GPU.