Calcul parallèle en Python

Master 1 Informatique

Antoine Vacavant

IUT Le Puy-en-Velay / Institut Pascal

Qu’est ce que Numba ?

- Compilateur de fonction just-in-time
- Avec spécification de type
- Pour les types numériques (float, int, complex)
- Parallélisme CPU ou GPU
- Adapté aux formats array numpy
- N’utilise pas l’API C de Python

Brève description de CUDA

- On développe des kernels qui sont exécutés en GPU

```
void initializeElementsTo(int initialValue, int *a, int N)
{

        if (i < N)
        {
                a[i] = initialValue;
        }
}
```

Brève description de CUDA

- On développe des kernels qui sont exécutés en GPU
- On distribue les données sur des threads

```
__global__
void initializeElementsTo(int initialValue, int *a, int N)
{
	int i = threadIdx.x + blockIdx.x * blockDim.x;
	if (i < N)
	{
		a[i] = initialValue;
	}
}
```

Brève description de CUDA

```
#include <stdio.h>

int main()
{
        int N = 1000;

        int *a;
        size_t size = N * sizeof(int);

        cudaMallocManaged(&a, size);

        size_t threads_per_block = 256;

        size_t number_of_blocks =
                (N + threads_per_block - 1) / threads_per_block;

        int initialValue = 6;

        initializeElementsTo<<<number_of_blocks, threads_per_block>>>
                (initialValue, a, N);

        cudaDeviceSynchronize();

        cudaFree(a);
}
```

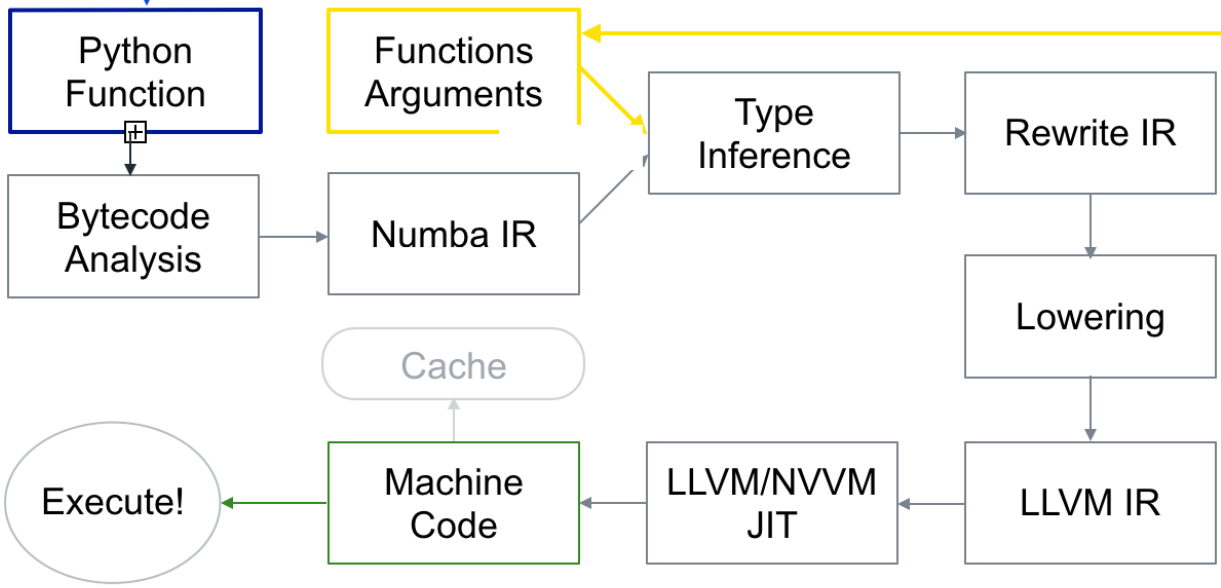
Brève description de CUDA

- Une fois le kernel défini
- Allocation de mémoire sur GPU
- Travail sur GPU
- Synchronisation avec le CPU
- Libération de la mémoire
- Compilation avec NVCC, exemple :

```
!nvcc -arch=sm_70 -o mismatched-config-loop 05-allocate/02-mismatched-config-loop.cu -run
```

Numba in a nutshell

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



Numba in a nutshell

1. La fonction Python est analysée en format bytecode, optimisée et convertie en représentation intermédiaire Numba (IR)
2. L'inférence de type permet de gérer les types « à la numpy » (float -> float64)
3. Le code est ensuite converti en code compatible LLVM
4. Ce dernier est compilé par LLVM et exécuté en machine « just in time » par assembleur

   →https://llvm.org/

Numba in a nutshell

- Bytecode produit à partir du code
- Control flow graph

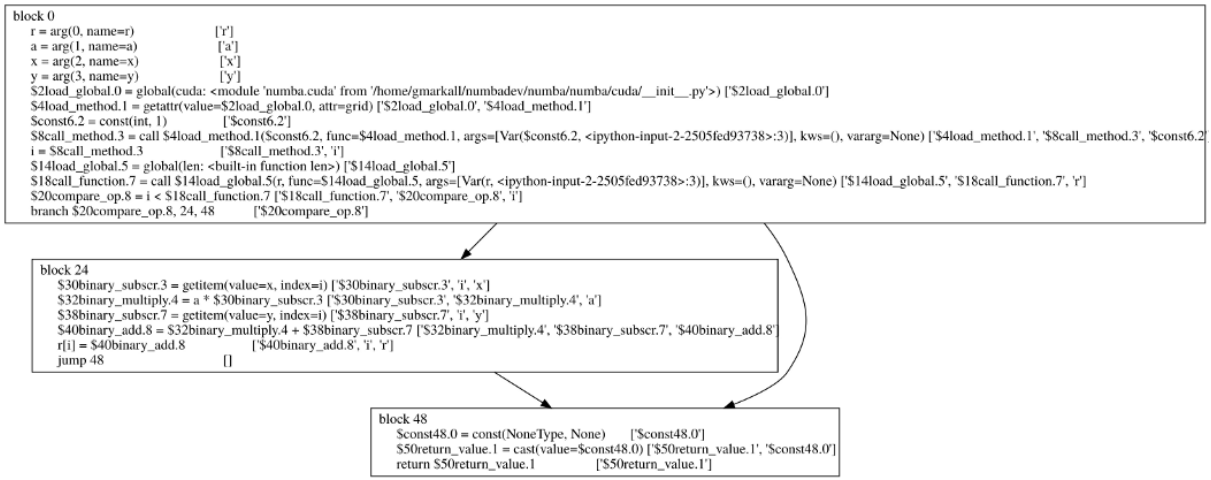
```
3           0 LOAD_GLOBAL              0 (cuda)
            2 LOAD_METHOD              1 (grid)
            4 LOAD_CONST               1 (1)
            6 CALL_METHOD              1
            8 STORE_FAST               4 (i)

5          10 LOAD_FAST                4 (i)
           12 LOAD_GLOBAL              2 (len)
           14 LOAD_FAST                0 (r)
           16 CALL_FUNCTION            1
           18 COMPARE_OP               0 (<)
           20 POP_JUMP_IF_FALSE       46
```

```
CFG adjacency lists:
{0: [24, 48], 24: [48], 48: []}
CFG dominators:
{0: {0}, 24: {24, 0}, 48: {48, 0}}
CFG post-dominators:
{0: {0, 48}, 24: {24, 48}, 48: {48}}
CFG back edges: []
CFG loops:
{}
CFG node-to-loops:
{0: [], 24: [], 48: []}
CFG backbone:
{0, 48}
```

Numba in a nutshell

- IR Numba

```
label 0:
    r = arg(0, name=r)                      ['r']
    a = arg(1, name=a)                      ['a']
    x = arg(2, name=x)                      ['x']
    y = arg(3, name=y)                      ['y']
    $2load_global.0 = global(cuda: <module 'numba.cuda' from '/home/gmarkall/numbadev/numba/nu
    $4load_method.1 = getattr(value=$2load_global.0, attr=grid) ['$2load_global.0', '$4load_me
    $const6.2 = const(int, 1)               ['$const6.2']
    $8call_method.3 = call $4load_method.1($const6.2, func=$4load_method.1, args=[Var($const6.
    i = $8call_method.3                     ['$8call_method.3', 'i']
    $14load_global.5 = global(len: <built-in function len>) ['$14load_global.5']
    $18call_function.7 = call $14load_global.5(r, func=$14load_global.5, args=[Var(r, <ipythor
    $20compare_op.8 = i < $18call_function.7 ['$18call_function.7', '$20compare_op.8', 'i']
    branch $20compare_op.8, 24, 48          ['$20compare_op.8']
label 24:
    $30binary_subscr.3 = getitem(value=x, index=i) ['$30binary_subscr.3', 'i', 'x']
    $32binary_multiply.4 = a * $30binary_subscr.3 ['$30binary_subscr.3', '$32binary_multiply.4
    $38binary_subscr.7 = getitem(value=y, index=i) ['$38binary_subscr.7', 'i', 'y']
    $40binary_add.8 = $32binary_multiply.4 + $38binary_subscr.7 ['$32binary_multiply.4', '$38b
    r[i] = $40binary_add.8                  ['$40binary_add.8', 'i', 'r']
```



Numba in a nutshell

- LLVM code

```
entry:
  %.57 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %.58 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %.59 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %.60 = mul i32 %.59, %.58
  %.61 = add i32 %.60, %.57
  %.93 = sext i32 %.61 to i64
  %.94 = icmp slt i64 %.93, %arg.r.5.0
  br i1 %.94, label %B24, label %B48

B24:                                              ; preds = %entry
  %.119 = icmp slt i32 %.61, 0
  %.120 = select i1 %.119, i64 %arg.x.5.0, i64 0
  %.121 = add i64 %.120, %.93
  %.134 = getelementptr i32, i32* %arg.x.4, i64 %.121
  %.135 = load i32, i32* %.134, align 4
  %.143 = sitofp i32 %.135 to double
  %.144 = fmul double %.143, %arg.a
  %.168 = select i1 %.119, i64 %arg.y.5.0, i64 0
  %.169 = add i64 %.168, %.93
  %.182 = getelementptr i32, i32* %arg.y.4, i64 %.169
  %.183 = load i32, i32* %.182, align 4
  %.191 = sitofp i32 %.183 to double
  %.192 = fadd double %.144, %.191
  %.217 = select i1 %.119, i64 %arg.r.5.0, i64 0
  %.218 = add i64 %.217, %.93
  %.231 = getelementptr i32, i32* %arg.r.4, i64 %.218
  %.232 = fptosi double %.192 to i32
  store i32 %.232, i32* %.231, align 4
  br label %B48
```

Approche jit

- Première approche pour compiler en CPU seul
- Ajoute un décorateur `@jit` avant la fonction

```
from numba import jit
import math

@jit
def hypot(x, y):
      x = abs(x);
      y = abs(y);
      t = min(x, y);
      x = max(x, y);
      t = t / x;
      return x * math.sqrt(1+t*t)
```

Approche ufuncs GPU

- Une ufunc (universal function) est une fonction numpy générale
- Peut être appliquée sur des données de n'importe quelle dimension
- Exemple de ufunc avec **`add()`**

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

np.add(a, b)
```

Approche ufuncs GPU

- Pour créer une ufunc GPU, on peut utiliser le décorateur **`@vectorize`**
- Les opérations + seront exécutées sur chaque élément

```
from numba import vectorize

@vectorize
def add_ten(num):
    return num + 10
```

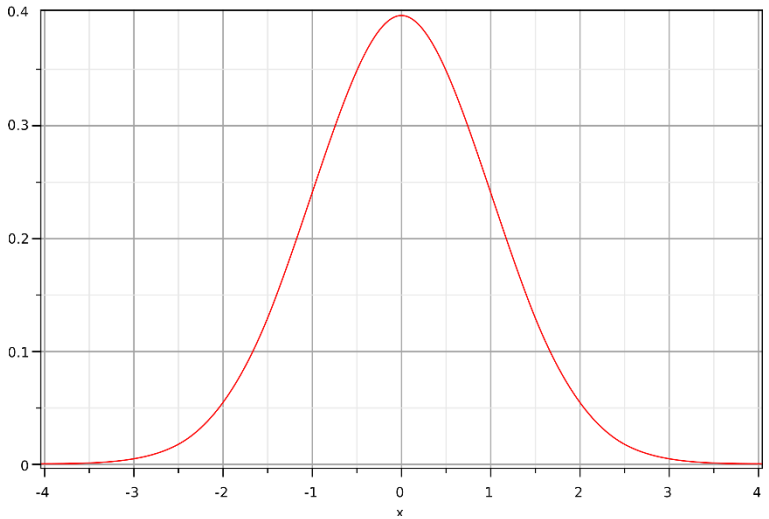
Approche ufuncs GPU

- On peut spécifier les types d'entrées / sorties
- Sur le modèle
  `[type_out(type_in1, type_in2, …)],`
  `target = 'cuda'`

```
@vectorize(['int64(int64, int64)'],
           target='cuda')
def add_ufunc(x, y):
    return x + y
```

Exercice

- Comment calculer et optimiser le calcul d'une distribution gaussienne ?
- Trois paramètres
  - X (dimension quelconque)
  - Moyenne (scalaire)
  - Ecart-type (scalaire)



$$f(x) = \frac{1}{\sigma\sqrt{2\,\pi}}\,\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

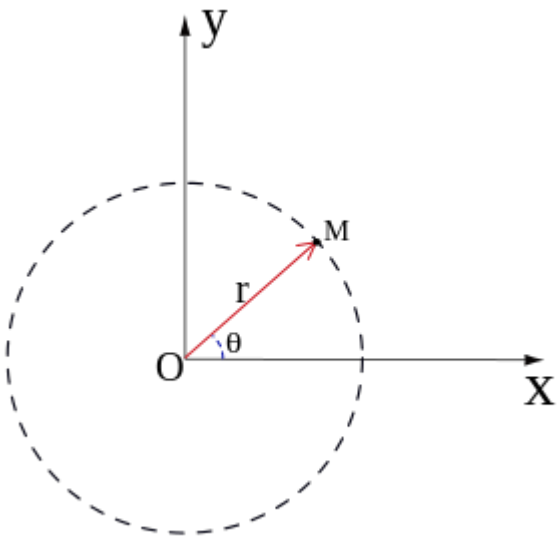
Solution

```
import math

SQRT_2PI = np.float32((2*math.pi)**0.5)

@vectorize(['float32(float32, float32, float32)'],
           target='cuda')
def gaussian_pdf(x, mean, sigma):
    return math.exp(-0.5 * ((x - mean) / sigma)**2) / (sigma
    * SQRT_2PI)


import numpy as np
x = np.random.uniform(-3, 3, size=1000000).astype(np.float32)
mean = np.float32(0.0)
sigma = np.float32(1.0)

%timeit gaussian_pdf(x, mean, sigma)
```

Approche CUDA jit

- Ufuncs : la même opération sur les données
- Lorsque l'on a besoin d'opérations plus complexes : `@cuda.jit`
- Exemple : Calcul de coordonnées polaires

$$x = r\cos\theta, \quad y = r\sin\theta.$$



Approche CUDA jit

```
@cuda.jit(device=True)
def polar_to_cartesian(rho, theta):
        x = rho * math.cos(theta)
        y = rho * math.sin(theta)
        return x, y

@vectorize(['float32(float32, float32, float32, float32)'], target='cuda')
def polar_distance(rho1, theta1, rho2, theta2):
        x1, y1 = polar_to_cartesian(rho1, theta1)
        x2, y2 = polar_to_cartesian(rho2, theta2)

        return ((x1 - x2)**2 + (y1 - y2)**2)**0.5


n = 1000000
rho1 = np.random.uniform(0.5, 1.5, size=n).astype(np.float32)
theta1 = np.random.uniform(-np.pi, np.pi, size=n).astype(np.float32)
rho2 = np.random.uniform(0.5, 1.5, size=n).astype(np.float32)
theta2 = np.random.uniform(-np.pi, np.pi, size=n).astype(np.float32)


polar_distance(rho1, theta1, rho2, theta2)
```

Generalized ufuncs

- Peuvent être utilisées pour manipuler les entrées / sorties de tailles différentes
- Les sorties sont dans la signature (CUDA like)
- Exemple : Calculer la norme L2 d'un vecteur

```
from numba import guvectorize
import math

@guvectorize(['(float32[:], float32[:])'],
'(i)->()', target='cuda')
def l2_norm(vec, out):
      acc = 0.0
      for value in vec:
      acc += value**2
      out[0] = math.sqrt(acc)
```

Remarques complémentaires

- Python Numba GPU autorise
  - Structures if/elif/else
  - Boucles while et for
  - Opérateurs mathématiques basiques
  - Fonctions des modules math et cmath
  - Tuples
- Pourquoi un programme GPU plus lent que CPU ?
  - Les données sont trop réduites
  - Les calculs sont trop simples
  - On copie les données vers/depuis le GPU
  - Les types de données sont trop couteux