

CUDA EN PYTHON (CUPY)

Francisco J. Hernández López

fcoj23@cimat.mx





<https://cupy.dev/>

- Librería de código abierto para cómputo paralelo usando GPU con Python
- Compatible con NumPy/Scipy
- Utiliza las librerías:
 - CUDA Toolkit
 - cuBLAS
 - cuRAND
 - cuSOLVER
 - cusparse
 - cuFFT
 - cuDNN
 - NCCL

INSTALACIÓN

<https://docs.cupy.dev/en/stable/install.html>

▪ Requerimientos:

- [NVIDIA CUDA GPU](#) with the Compute Capability 3.0 or larger.
- [CUDA Toolkit](#): v10.2 / v11.0 / v11.1 / v11.2 / v11.3 / v11.4 / v11.5 / v11.6 / v11.7 / v11.8 / v12.0
 - If you have multiple versions of CUDA Toolkit installed, CuPy will automatically choose one of the CUDA installations. See [Working with Custom CUDA Installation](#) for details.
 - This requirement is optional if you install CuPy from `conda-forge`. However, you still need to have a compatible driver installed for your GPU. See [Installing CuPy from Conda-Forge](#) for details.
- [Python](#): v3.7.0+ / v3.8.0+ / v3.9.0+ / v3.10.0+ / v3.11.0+

INSTALANDO CUPY EN CONDA

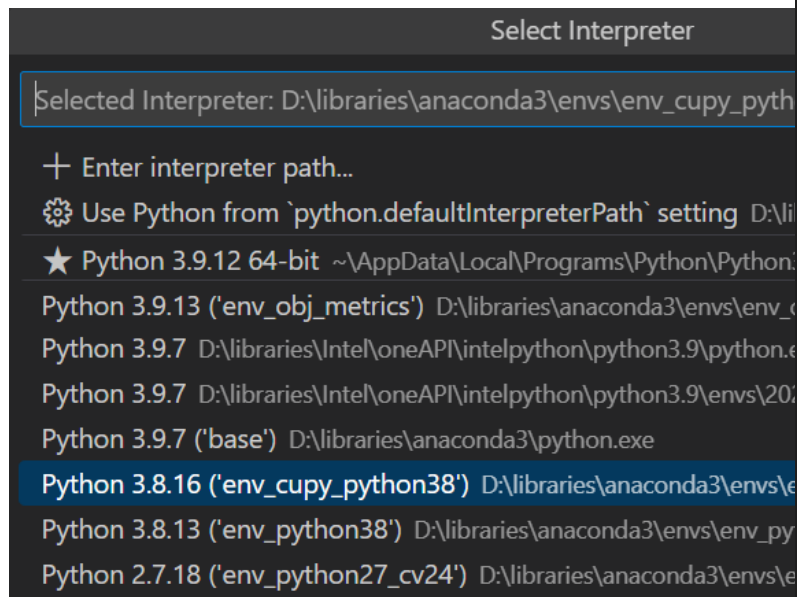
- Creamos un ambiente en conda:
 - `conda create -n env_cupy_python38 python=3.8`
- Entramos al ambiente:
 - `conda activate env_cupy_python38`
- Instalamos Numpy:
 - `conda install -c anaconda numpy`
- Instalamos Scipy:
 - `conda install -c anaconda scipy`
- Instalamos cupy con cudatoolkit=11.2
 - `conda install -c conda-forge cupy cudatoolkit=11.2`

De esta forma queda instalado `cupy-11.5.0` y `cudatoolkit-11.2.2`

EJEMPLO-1

- Abrir VStudio Code
 - Abrir Carpeta
 - Crear archivo .py

Antes de ejecutar el programa hay que seleccionar el ambiente que creamos



Example-1.py X

Example-1.py > ...

```
1 import numpy as np
2 import cupy as cp
3
4 #Declaramos un arreglo en la CPU
5 x_cpu=np.array([1,2,3])
6
7 #Declaramos un arreglo en la GPU
8 x_gpu=cp.array([1,2,3])
9
10 #Calculamos la norma L2 en la CPU
11 norm_L2_x_cpu=np.linalg.norm(x_cpu)
12
13 #Calculamos la norma L2 en la GPU
14 norm_L2_x_GPU=cp.linalg.norm(x_gpu)
15
16 #Mostrar resultados
17 print(norm_L2_x_cpu)
18 print(norm_L2_x_GPU)
```

https://docs.cupy.dev/en/stable/user_guide/basic.html

CuPy. Francisco J. Hernández-López

Enero-Junio 2024

5

ELEGIR GPU PARA EL PROCESAMIENTO

- CuPy ejecuta el código en la GPU que se encuentre activa, la cual es generalmente la GPU con índice cero
- Si contamos con dos GPUs, entonces es posible enviar procesamiento a la otra GPU de la siguiente manera:

```
>>> with cp.cuda.Device(1):  
...     x_on_gpu1 = cp.array([1, 2, 3, 4, 5])  
>>> x_on_gpu0 = cp.array([1, 2, 3, 4, 5])
```

```
20 with cp.cuda.Device(0):
21     y_gpu = cp.array([1, 2, 3, 4, 5])
22     print(y_gpu.device)
23
24 with cp.cuda.Device(1):
25     z_gpu = cp.array([1, 2, 3, 4, 5])
26     print(z_gpu.device)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Python + - [] [] ... ^ X

3.7416573867739413

3.7416573867739413

<CUDA Device 0>

Traceback (most recent call last):

File "d:/fcoj23/CIMAT_MERIDA/Docencia/cimat/ComputoParalelo/ejemplos/cupy/basicExamples/Example-1.py",
line 24, in <module>

with cp.cuda.Device(1):

File "cupy\cuda\device.pyx", line 184, in cupy.cuda.device.Device.__enter__

File "cupy_backends\cuda\api\runtime.pyx", line 374, in cupy_backends.cuda.api.runtime.setDevice

File "cupy_backends\cuda\api\runtime.pyx", line 143, in cupy_backends.cuda.api.runtime.check_status

cupy_backends.cuda.api.runtime.CUDARuntimeError: cudaErrorInvalidDevice: invalid device ordinal

Nota: Cuando solo tenemos disponible una GPU, no es necesario indicar el `.Device()`, por default realizará las operaciones en el device con ID=0

TRANSFERENCIA DE DATOS

- Mover arreglos de la CPU a la GPU y viceversa

```
1 import numpy as np
2 import cupy as cp
3
4 x_cpu = np.array([1, 2, 3])
5
6 #Copiar datos desde la CPU a la GPU
7 x_gpu = cp.asarray(x_cpu)
8
9 #Copiar datos desde la GPU a la CPU
10 y_cpu = cp.asnumpy(x_gpu)
11 #y_cpu = x_gpu.get() #También se puede
12 | | | | | #usar el atributo get()
13 #Mostrar resultados
14 print(x_cpu)
15 print(x_gpu)
16 print(y_cpu)
```


TRANSFERENCIA DE DATOS

```
18 #cp.asarray y cp.asnumpy aceptan cualquier entrada (CPU/GPU)
19 #cp.asarray devuelve un arreglo CuPy en el device
20 #cp.asnumpy devuelve un arreglo Numpy en el host
21 z_cpu=x_cpu+y_cpu
22 #z_aux=x_gpu+y_cpu #No es posible
23 #Podemos hacer lo siguiente
24 z2_cpu=cp.asnumpy(x_gpu)+y_cpu
25 #o esto
26 z3_gpu=x_gpu+cp.asarray(y_cpu)
27
28 print(z_cpu)
29 print(z2_cpu)
30 print(z3_gpu)
```

FUNCIONES CUDA-KERNEL DEFINIDAS POR EL USUARIO


- Hay tres formas de definir CUDA-Kernels en CuPy:
 - CUDA-Kernel **elemental** (ElementwiseKernel)
 - CUDA-Kernel de **reducción** (ReductionKernel)
 - CUDA-Kernel **crudo** (RawKernel)

CUDA-KERNEL ELEMENTAL

- Se define con la clase *ElementwiseKernel*
- La instancia de esta clase define un CUDA-Kernel, el cual puede ser invocado por el método `__call__`
- Consiste en cuatro partes:
 - Lista de argumentos de **entrada**
 - Lista de argumentos de **salida**
 - Código del **cuerpo** del ciclo
 - **Nombre** del CUDA-Kernel

EJEMPLO DE CUDA-KERNEL ELEMENTAL

```
1 import numpy as np
2 import cupy as cp
3
4 #Creando una función CUDA-Kernel para evaluar las diferencias al cuadrado
5 squared_diff = cp.ElementwiseKernel('float32 x, float32 y',
6                                     'float32 z',
7                                     'z = (x - y) * (x - y)',
8                                     'squared_diff')
9
10 x = cp.arange(10, dtype=np.float32).reshape(2, 5)
11 y = cp.arange(5, dtype=np.float32)
12
13 #z=squared_diff(x, y)
14 #También se puede hacer esto
15 z = cp.empty((2, 5), dtype=np.float32)
16 squared_diff(x, y, z)
17
18 print(x)
19 print(y)
20 print(z)
```



```
[[0. 1. 2. 3. 4.]
 [5. 6. 7. 8. 9.]]
[0. 1. 2. 3. 4.]
[[ 0.  0.  0.  0.  0.]
 [25. 25. 25. 25. 25.]]
```

Nota: las variables n , i y nombres comenzando con $_$ son reservados para uso interno

CUDA-KERNEL DE REDUCCIÓN

- Se define con la clase *ReductionKernel*
- Consiste en cuatro partes:
 - Expresión de **mapeo**: Utilizado para el pre-procesamiento de cada elemento a reducir
 - Expresión de la **reducción**: Operador para reducir los múltiples valores mapeados. Las variables especiales a y b se usan como operandos
 - **Mapeo posterior** a la reducción: Se utiliza para transformar los valores reducidos resultantes. La variable especial a se utiliza como entrada. La salida debe escribirse en el parámetro de salida.
 - **Valor de indentidad**: Es el valor inicial de la reducción

EJEMPLO: NORMA L2 DE UN ARREGLO A LO LARGO DE UN EJE

```
1 import numpy as np
2 import cupy as cp
3
4 l2norm_kernel = cp.ReductionKernel('T x', # Entrada
5                                     'T y', # Salida
6                                     'x * x', # 1- Mapeo
7                                     'a + b', # 2- Reducción
8                                     'y = sqrt(a)', # 3- Mapeo posterior
9                                     '0', # 4- Valor inicial
10                                    'l2norm' # Nombre del Kernel
11                                    )
12
13 x = cp.arange(10, dtype=np.float32).reshape(2, 5)
14 y=l2norm_kernel(x, axis=0)
15
16 print(x)
17 print(y)
```

CUDA-KERNEL CRUDO

- Se define con la clase ***RawKernel***
- Con esta clase se puede definir una función Kernel de CUDA desde su código crudo en C
- De esta forma se tiene el control del tamaño de la malla, del bloque, la memoria compartida y el stream

EJEMPLO: SUMA DE VECTORES USANDO CUDA-KERNEL CRUDO

```
1 import numpy as np
2 import cupy as cp
3
4 VectorAdd_kernel_cp = cp.RawKernel(r'''
5     extern "C" __global__
6     void VectorAdd_kernel(const float* x1_d, const float* x2_d, float* y_d) {
7         int tid = blockDim.x * blockIdx.x + threadIdx.x;
8         y_d[tid] = x1_d[tid] + x2_d[tid];
9     }
10    ''', 'VectorAdd_kernel')
11 x1_d = cp.arange(25, dtype=cp.float32).reshape(5, 5)
12 x2_d = cp.arange(25, dtype=cp.float32).reshape(5, 5)
13 y_d = cp.zeros((5, 5), dtype=cp.float32)
14 VectorAdd_kernel_cp((5,), (5,), (x1_d, x2_d, y_d)) # grid, block and arguments
15 |         |         |         |         |         |         |         |         |         |         |
16 # Notar que tanto el grid como el block son 1D
17 print(y_d)
```

CORRESPONDENCIA DE TIPOS DE DATOS

CuPy/NumPy type	Corresponding kernel types	itemsized (bytes)
bool	bool	1
int8	char, signed char	1
int16	short, signed short	2
int32	int, signed int	4
int64	long long, signed long long	8
uint8	unsigned char	1
uint16	unsigned short	2
uint32	unsigned int	4
uint64	unsigned long long	8
float16	half	2
float32	float	4
float64	double	8
complex64	float2, cuFloatComplex, complex<float>	8
complex128	double2, cuDoubleComplex, complex<double>	16

EJEMPLO: SUMA Y MULTIPLICACIÓN USANDO UN MODULO

```
1 import numpy as np
2 import cupy as cp
3
4 loaded_from_source = r'''
5 extern "C"{
6     __global__ void test_sum(const float* x1_d, const float* x2_d, float* y_d, unsigned int N){
7         unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
8         if (tid < N){
9             y_d[tid] = x1_d[tid] + x2_d[tid];
10        }
11    }
12    __global__ void test_mult(const float* x1_d, const float* x2_d, float* y_d, unsigned int N){
13        unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
14        if (tid < N){
15            y_d[tid] = x1_d[tid] * x2_d[tid];
16        }
17    }
18 }'''
```

EJEMPLO: SUMA Y MULTIPLICACIÓN USANDO UN MODULO

```
19 module = cp.RawModule(code=loaded_from_source)
20 ker_sum = module.get_function('test_sum')
21 ker_mult = module.get_function('test_mult')
22 N = 10
23 x1_d = cp.arange(N**2, dtype=cp.float32).reshape(N, N)
24 x2_d = cp.ones((N, N), dtype=cp.float32)
25 y_d = cp.zeros((N, N), dtype=cp.float32)
26 z_d = cp.zeros((N, N), dtype=cp.float32)
27 ker_sum((N,), (N,), (x1_d, x2_d, y_d, N**2))
28 ker_mult((N,), (N,), (x1_d, x2_d, z_d, N**2))
29 print(y_d)
30 print(z_d)
```

GRACIAS POR SU ATENCIÓN

Francisco J. Hernandez-Lopez

fcoj23@cimat.mx

WebPage:

www.cimat.mx/~fcoj23

