COMPUTE UNIFIED DEVICE ARCHITECTURE

...Y PYTHON

PYTHON: ENTRE DOS MUNDOS

- Python es un lenguaje interpretado, EXTREMADAMENTE lento, pero muy flexible gracias a infinidad de librerías.
- Entre otras, NUMBA permite el uso y gestión de CPUs y GPUs, mediante “decoradores” y compiladores:

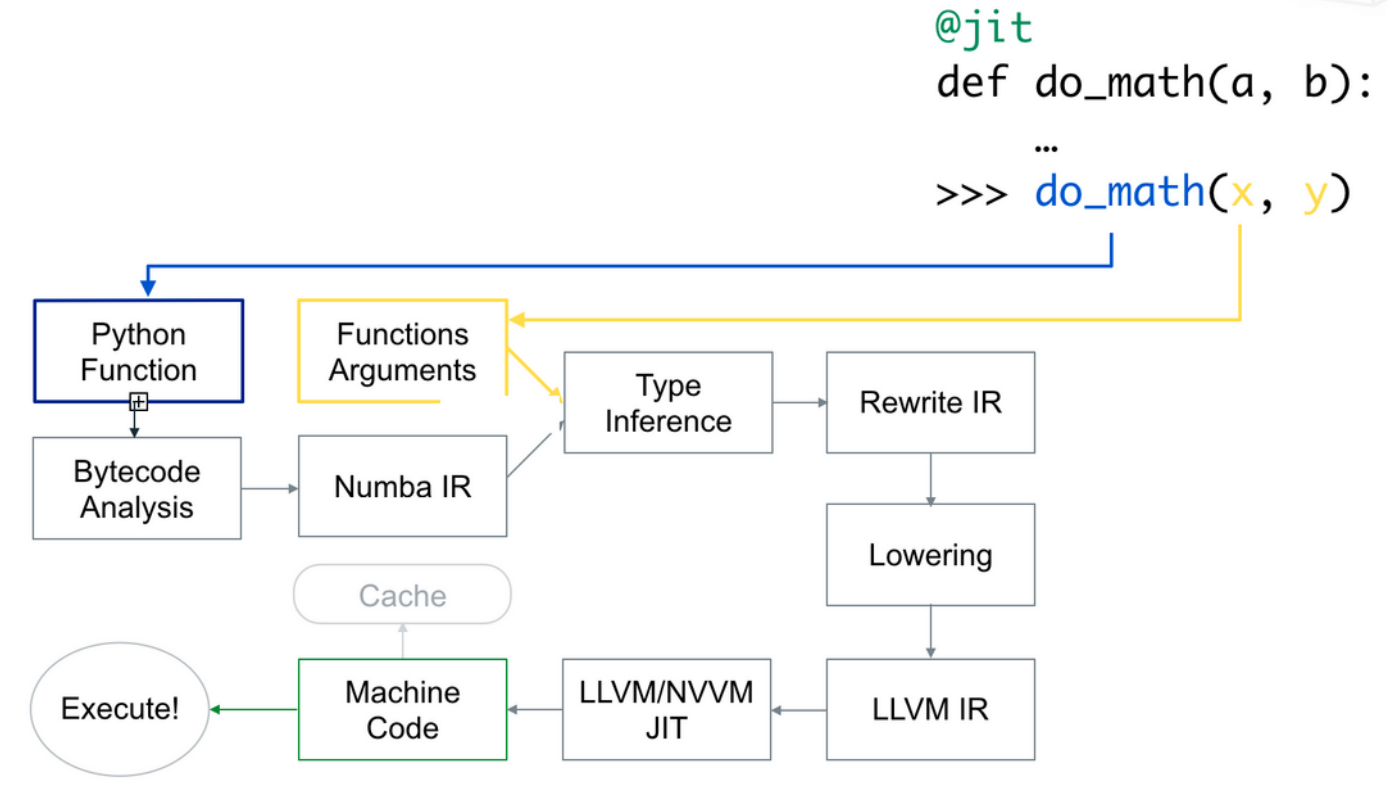
```
from numba import jit
import math
@jit
def hypot(x, y):
    x = abs(x);
    y = abs(y);
    t = min(x, y);
    x = max(x, y);
    t = t / x;
    return x * math.sqrt(1+t*t)
```

El *@jit* le dice a python que debe compilar para CPU esa función→ Ganaremos velocidad.

Ojo: no cualquier código python puede ser compilado por NUMBA (diccionarios... mirar la documentación).

PYTHON: ENTRE DOS MUNDOS

- Esquema de compilación para CPU:



- (Imagen del DLI, Nvidia, 2022)

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- La primera ejecución sobre GPU se puede realizar mediante vectorización:

```
@vectorize(['int64(int64, int64)'], target='cuda') # La firma del tipo de dato y el target son necesarios para las vectorizaciones
def add_ufunc(x, y):
    return x + y
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
add_ufunc(a, b)
```

La función "universal" *add_ufunc* suma dos elementos, pero su operación puede realizarse en paralelo, elemento a elemento.

Su implementación sobre C usaría un bucle *forall*.

Otros tipos de datos pueden ser *float32*, *complex128*... y muchos más, que deberemos declarar al generar el array:

```
x = np.random.uniform(-3, 3, size=1000000).astype(np.float32)
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- ¿Y si la función no tiene estructura *vectorizable*? Tenemos *@cuda*:

```
from numba import cuda
@cuda.jit(device=True [, inline=True])          # Función CUDA, no kernel CUDA. No vectorizable.
def polar_to_cartesian(rho, theta):             # No llamable desde función python estándar.
    x = rho * math.cos(theta)                   # Puede forzarse “inline”
    y = rho * math.sin(theta)
    return x, y                                 #Puede devolver datos

@vectorize(['float32(float32, float32, float32, float32)'], target='cuda')
def polar_distance(rho1, theta1, rho2, theta2):
    x1, y1 = polar_to_cartesian(rho1, theta1)   # llamada a función CUDA desde ufuncs vectorizables
    x2, y2 = polar_to_cartesian(rho2, theta2
    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5
```

Limitaciones de código en GPU:

- if, elif, else
- while, for
- Operadores matemáticos básicos
- Algunas funciones de los módulos math y cmath
- Tuplas
- Y algunas cosas más

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Movimientos entre memorias:
  - NUMBA tiene herramientas para:
    - Mover datos entre memorias
    - Crear datos directamente en alguna memoria concreta
    - Manejar memoria pinned...

```
fn = 100000
noise = (np.random.normal(size=n) * 3).astype(np.float32)    # Datos en memoria Host
t = np.arange(n, dtype=np.float32)
period = n / 23

d_noise = cuda.to_device(noise)                              # Datos movidos a GPU
d_t = cuda.to_device(t)                                      # Datos movidos a GPU
d_pulses = cuda.device_array(shape=(n,), dtype=np.float32)   # Datos directamente generados en memoria de GPU, más adelante podremos copiar
                                                             # de/al host con d_pulses_host=d_pulses.copy_to_host()

make_pulses(d_t, period, 100.0, out=d_pulses)                # Función vectorizada con target=cuda
waveform = add_ufunc(d_pulses, d_noise)                      # Función vectorizada con target=cuda
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Variables intrínsecas. NUMBA ofrece mecanismos para acceder a las variables sintrínsecas de forma análoga a como lo hace CUDA:

```
from numba import cuda

# Hace falta un array de salida “out” como parámetro. Los kernels CUDA (con @cuda.jit) no pueden devolver nada, como los kernels de C.
@cuda.jit('void(int32[:], int32[:], int32[:])')
def add_kernel(x, y, out):
    # Los valores de hilos y bloques no se saben hasta el momento de ejecución del kernel.

    # Esto obtiene el id de un solo hilo de todos los generados
    idx = cuda.grid(1)     # 1 = malla unidimensional, devuelve un valor escalar.
                           # Esta función de NUMBA es equivalente a cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    # Cada hilo procesará un solo valor del array general
    out[idx] = x[idx] + y[idx]
```

- O también:

```
tx = cuda.threadIdx.x   ty = cuda.threadIdx.y   bx = cuda.blockIdx.x   by = cuda.blockIdx.y   bw = cuda.blockDim.x    bh = cuda.blockDim.y
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Variables intrínsecas. También encontraremos opciones para 2 y 3 dimensiones (y algunas variables más):

```
from numba import cuda

@cuda.jit
def get_2D_indices(A):
    x, y = cuda.grid(2)                          # El 2 señala que queremos dos valores en las dos dimensiones disponibles
                                                 # Esa línea es equivalente a las dos siguientes:
                                                 # x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
                                                 # y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
    # Ya podemos operar en 2D
    A[x][y] = x + y / 10

Blocks = (2, 2)                                  # Bloques definidos en 2D: malla de 2x2 bloques. Para una dimensión bastaría con Blocks = 10, p.e.
threads_per_block = (2, 2)                       # Hilos por bloque definidos también en 2D: bloques de 2x2 hilos
get_2D_indices[blocks, threads_per_block](d_A)# Llamada estándar a un kernel en GPU desde python
result = d_A.copy_to_host()
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- Configuración de ejecución. NUMBA tiene una sintaxis específica para determinar bloques e hilos por bloque de forma análoga a como lo hace CUDA:

```
add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
cuda.synchronize()
print(d_out.copy_to_host())
```

```
# Configuración de ejecución (en rojo) entre corchetes
# Barreras de sincronización necesarias, ya sabemos por qué
# Copia final al Host para poder imprimir.
```

  - Por supuesto, hay que elegir valores adecuados para la configuración de ejecución, en las dimensiones adecuadas.
- Ej: uso de strides para estructuras muy grandes:

```
@cuda.jit
def add_kernel(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]
```

```
# Obtenemos el tamaño total de la malla, equivalente a cuda.blockDim.x * cuda.gridDim.x
# Cada hilo trabajará sobre su índice y los demás elementos separados stride de dicho índice.
```

PYTHON: CAMINITO DE LA SINTAXIS CUDA

- NUMBA implementa, también operaciones atómicas:

```
@cuda.jit
def thread_counter_safe(global_counter):
    cuda.atomic.add(global_counter, 0, 1)          # Inrementar 1 en offset 0 del array global global_counter
```

- Para memoria compartida:

```
@cuda.jit
def swap_with_shared(vector, swapped):
    temp = cuda.shared.array(4, dtype=types.int32)          # Reservar memoria shared de tipo int32 con 4 posiciones.
    idx = cuda.grid(1)
    temp[idx] = vector[idx]                                 # Copiar a la memoria shared los valores que necesitamos. Cada hilo copia un dato.
    cuda.syncthreads()                                      # Ya sabemos para qué es esto... Recordad: ¡a nivel de bloque!
    swapped[idx] = temp[3 - cuda.threadIdx.x]               # Cada hilo lee un dato que escribió otro hilo en la memoria shared.
```

- Y para manejar streams, memoria local (de cada hilo), la de constantes...
- Cualquier kernel de NUMBA se verá afectado por la coalescencia (y conflictos entre bancos...), exactamente igual que con los kernels de CUDA.