High-Performance Computing para Modelamiento Matemático con (Py)CUDA

Sesión 2: *Introducción a (Py)CUDA*

Diego Maldonado

Universidad Católica del Maule

16 de diciembre de 2025

Índice

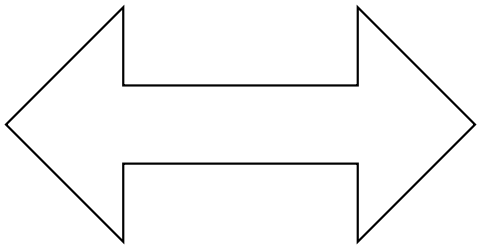
" CUDA® es una plataforma de cálculo paralelo y un modelo de programación desarrollado por NVIDIA para el cálculo general en unidades de procesamiento gráfico (GPU). Con CUDA, los desarrolladores pueden acelerar drásticamente las aplicaciones de cálculo aprovechando la potencia de las GPU."

Extraído de la web de Nvidia https://developer.nvidia.com/cuda-zone

El impacto de CUDA

- Podemos definir la computación sobre tarjetas gráficas como el uso de una tarjeta gráfica (GPU) para realizar cálculos de propósito general.
- El modelo de computación sobre tarjetas gráficas consiste en usar conjuntamente una CPU y una GPU de manera que formen un modelo de computación heterogéneo.
- La parte secuencial de un programa se ejecutará sobre la CPU (Host) y la parte paralelizable del cálculo se ejecutará sobre la GPU (device).

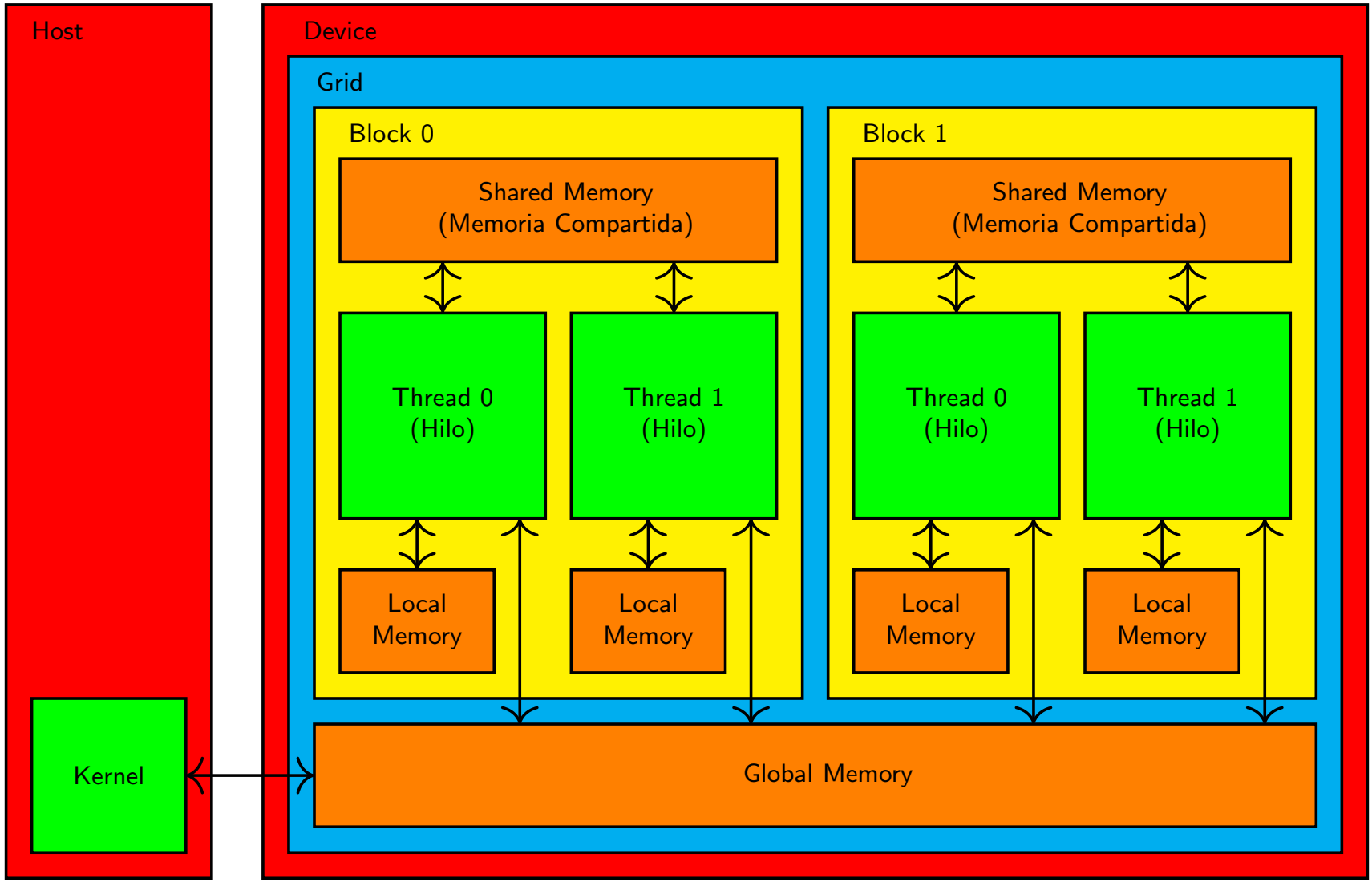
CPU (Host)



GPU (Device)

- Inicialmente, las GPU estaban diseñadas principalmente para el procesamiento de gráficos.
- Para usarlas en cómputo general, los problemas debían reformularse para encajar en el *pipeline* gráfico (por ejemplo, mediante shaders y texturas).
- Al reconocer el potencial de esta capacidad de cómputo, NVIDIA lanzó CUDA (Compute Unified Device Architecture) hacia 2007, facilitando el acceso al paralelismo masivo de las GPU para aplicaciones científicas y de ingeniería.

Diagrama de la arquitectura CUDA (Simplificado)



Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Local Memory: Cada thread tiene asignada una memoria de acceso exclusivo llamada local memory (memoria local).

Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Local Memory: Cada thread tiene asignada una memoria de acceso exclusivo llamada local memory (memoria local).

Block (Bloque): Un Block (Bloque) es un arreglo de hilos de a lo más 1024 hilos.

Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Local Memory: Cada thread tiene asignada una memoria de acceso exclusivo llamada local memory (memoria local).

Block (Bloque): Un Block (Bloque) es un arreglo de hilos de a lo más 1024 hilos.

Share Memory: Cada bloque tiene asignada una memoria que es accesible por todos los hilos de ese bloque llamada Share Memory (memoria compartida). Es una memoria de rápida lectura y escritura y, es mayor capacidad que la memoria local. Los hilos pueden compartir información a través de esta memoria.

Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Local Memory: Cada thread tiene asignada una memoria de acceso exclusivo llamada local memory (memoria local).

Block (Bloque): Un Block (Bloque) es un arreglo de hilos de a lo más 1024 hilos.

Share Memory: Cada bloque tiene asignada una memoria que es accesible por todos los hilos de ese bloque llamada Share Memory (memoria compartida). Es una memoria de rápida lectura y escritura y, es mayor capacidad que la memoria local. Los hilos pueden compartir información a través de esta memoria.

Grid: Un Grid (grilla) es un arreglo de bloques.

Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Local Memory: Cada thread tiene asignada una memoria de acceso exclusivo llamada local memory (memoria local).

Block (Bloque): Un Block (Bloque) es un arreglo de hilos de a lo más 1024 hilos.

Share Memory: Cada bloque tiene asignada una memoria que es accesible por todos los hilos de ese bloque llamada Share Memory (memoria compartida). Es una memoria de rápida lectura y escritura y, es mayor capacidad que la memoria local. Los hilos pueden compartir información a través de esta memoria.

Grid: Un Grid (grilla) es un arreglo de bloques.

Global Memory: La memoria global es una memoria a la cual tienen acceso todos los bloques de la misma grilla. En general es una memoria de lenta lectura y escritura y de mayor capacidad que la memoria compartida.

Las componentes más importantes de la arquitectura CUDA son las siguientes:

Thread (Hilo): Unidad más elemental de procesamiento. Cada thread puede ejecutar una tarea independientemente del resto. Una GPU de sobremesa puede contener desde cientos a miles hilos.

Local Memory: Cada thread tiene asignada una memoria de acceso exclusivo llamada local memory (memoria local).

Block (Bloque): Un Block (Bloque) es un arreglo de hilos de a lo más 1024 hilos.

Share Memory: Cada bloque tiene asignada una memoria que es accesible por todos los hilos de ese bloque llamada Share Memory (memoria compartida). Es una memoria de rápida lectura y escritura y, es mayor capacidad que la memoria local. Los hilos pueden compartir información a través de esta memoria.

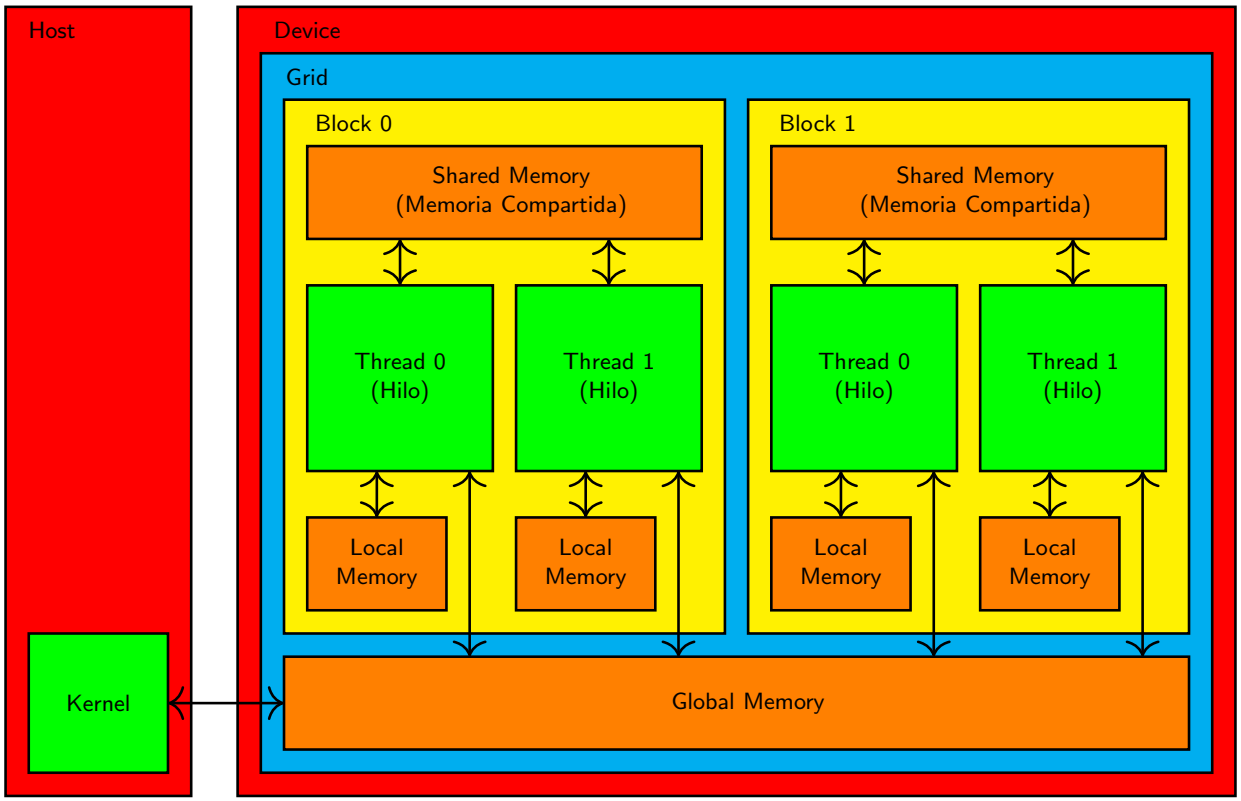
Grid: Un Grid (grilla) es un arreglo de bloques.

Global Memory: La memoria global es una memoria a la cual tienen acceso todos los bloques de la misma grilla. En general es una memoria de lenta lectura y escritura y de mayor capacidad que la memoria compartida.

Kernel: Es una secuencia de instrucciones (programa o algoritmo) que vienen desde el Host para que ejecute la el Device. Las instrucciones son las mismas para todos los hilos. Solo puede intercambiar datos con el Device a través de la memoria global. Además indica la cantidad de bloques e hilos por bloque que se van a utilizar.

Ejercicio

- ¿Cuántos hilos se puede ejecutaren paralelo el dispositivo?
- ¿Cuántos hilos se puede comunicar rápido entre sí en el dispositivo ?
- ¿Cuántos hilos se puede comunicar entre sí en el dispositivo?



CUDA basa su sintaxis en C++. Por lo que los Kernel tienen dicha sintaxis.

Ejemplo

Un ejemplo de un Kernel que suma, elemento a elemento, dos vectores *a* y *b* de tamaño *n* y almacena la respuesta en el vector *c* sería el siguiente:

```
__global__ void vecAdd(double *a,double *b, double *c, int n)
{
  // __global__ : Indica que es un Kernel.
  // void       : Indica que la función no entrega salida.

  int id = threadIdx.x;
  // threadIdx.x: Identificador o índice del hilo.

  // Guard de seguridad: ignora hilos fuera del rango [0, n)
  if (id < n)
      // Cada hilo suma un elemento: c[i] = a[i] + b[i]
      c[id] = a[id] + b[id];
}
```

Nota: //: Comentario de una línea, equivalente a # de Python.

```
// 1. Declaración de variables (definir tipos es OBLIGATORIO)
bool  condicion = true;    // Variable booleana para el if/else
int   limite    = 5;       // Límite para los bucles
int   contador  = 0;       // Contador para el while

// 2. Condicional
if (condicion) {
    printf("La condición es verdadera\n");
    // printf no hace salto de linea como print Python, es necesario agregar \n
} else {
    printf("La condición es falsa\n");
}

// 3. Bucle for:  i empieza en 0, se repite mientras i < limite, y luego ++i
for (int i = 0; i < limite; ++i) {
    printf("For: iteración %d de %d\n", i, limite);
    // primer %d indica que se imprimirá el entero "i"
    // segundo %d indica que se imprimirá el entero "limite"
}

// 4. Bucle while
//    - se repite mientras contador < limite
while (contador < limite) {
    printf("While: contador = %d\n", contador);
    contador++;   // Incremento de contador equivalente a contador = contador +1
}
```

PyCUDA es un ambiente de programación CUDA en Python.
Las ventajas de trabajar con PyCUDA son:

- Python es un lenguaje de más alto nivel que C++.
- Posibilidad de combinar bibliotecas de Python con CUDA (numpy, matplotlib, etc).
- Mantiene la potencia de CUDA.
- Python realiza la asignación de memoria por nosotros.
- Es compatible con entornos en la nube como https://colab.google/

```
1  import pycuda.autoinit
2  import pycuda.driver as drv
3  import numpy as np
4
5  from pycuda.compiler import SourceModule
6  mod = SourceModule("""
7  __global__ void vecAdd(double *a,double *b, double
       *c, int n)
8  {
9    int id = threadIdx.x;
10   if (id < n)
11       c[id] = a[id] + b[id];
12 }
13 """)
14
15 sumar = mod.get_function("vecAdd")
16 n=10
17 a = np.random.randn(n)
18 b = np.random.randn(n)
19
20 dest = np.zeros_like(a)
21 sumar(drv.In(a), drv.In(b), drv.Out(dest), np.int64
       (n),
22         block=(n,1,1), grid=(1,1))
23
24 print(dest-(a+b))
```

Nuestro primer Kernel en (Py)CUDA.

El siguiente link lo llevará al cuaderno con el ejemplo anterior:
https://colab.research.google.com/drive/174jdncAY1xhJkxZLwitRQadRRexhYoDo?usp=sharing

- `drv.In(a)`: indica que el arreglo a se copiará *del host (CPU) al device (GPU)* antes de ejecutar el kernel. Internamente reserva memoria en GPU y pasa al kernel un puntero a esos datos.
- `drv.Out(dest)`: señala que `dest` es un búfer de salida; después de la ejecución del kernel, PyCUDA copia automáticamente los datos desde la GPU de vuelta al arreglo `dest` en el host.
- `block=(400,1,1)`: define la configuración de hilos por bloque. Aquí cada bloque contiene 400 hilos en la dimensión x (y 1 en y y z). Cada hilo obtiene un índice `threadIdx.x` entre 0 y 399.
- `grid=(1,1)`: define la cantidad de bloques en la rejilla de ejecución. Con `grid=(1,1)` y `block=(400,1,1)` se lanzan exactamente 1 bloque de 400 hilos, es decir, 400 hilos en total.
- `SourceModule(...)`: compila en tiempo de ejecución el código CUDA proporcionado como cadena. Devuelve un objeto que gestiona el módulo compilado y permite extraer funciones GPU.
- `sumar = mod.get_function("sumar")`: obtiene un manejador de función (callable) asociado al kernel CUDA llamado `"sumar"` dentro del módulo compilado. Con este objeto se invoca el kernel desde Python, pasando punteros y la configuración de `grid` y `block`.

Nuestro primer código Python+Cuda será el siguiente
hola-mundo-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

L1: inicialización PyCUda.

Nuestro primer código Python+Cuda será el siguiente
hola-mundo-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

L1: inicialización PyCUda.

L2: Biblioteca para cargar Kernels.

Nuestro primer código Python+Cuda será el siguiente
hola-mundo-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

L1: inicialización PyCUda.

L2: Biblioteca para cargar Kernels.

L4: Numpy: Biblioteca de manipulación numérica.

Nuestro primer código Python+Cuda será el siguiente
hola-mundo-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

- L1: inicialización PyCUda.
- L2: Biblioteca para cargar Kernels.
- L4: Numpy: Biblioteca de manipulación numérica.
  - El kernel corresponde a las lineas 9-12.

Nuestro primer código Python+Cuda será el siguiente
`hola-mundo-1.py`

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

- L1: inicialización PyCUda.
- L2: Biblioteca para cargar Kernels.
- L4: Numpy: Biblioteca de manipulación numérica.
  - El kernel corresponde a las lineas 9-12.
- L13: crea en Python la función `HolamMundo` que corre en la GPU usando la función `HolaMundoGPU` del Kernel.

Nuestro primer código Python+Cuda será el siguiente
`hola-mundo-1.py`

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

- L1: inicialización PyCUda.
- L2: Biblioteca para cargar Kernels.
- L4: Numpy: Biblioteca de manipulación numérica.
  - El kernel corresponde a las lineas 9-12.
- L13: crea en Python la función `HolamMundo` que corre en la GPU usando la función `HolaMundoGPU` del Kernel.
- L15: corre el Kernel `HolaMundoGPU` cargando los datos y la configuración de la GPU mediante la función de Python `HolamMundo`.

Nuestro primer código Python+Cuda será el siguiente
`hola-mundo-1.py`

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n)
8  {
9    printf("Hola Mundo!\\n");
10 }
11 """)
12
13 HolaMundo = mod.get_function("HolaMundoGPU
       ")
14
15 HolaMundo(numpy.int32(1),block=(1,1,1))
```

Salida:

```
1  Hola Mundo!
```

Nota

- L1: inicialización PyCUda.
- L2: Biblioteca para cargar Kernels.
- L4: Numpy: Biblioteca de manipulación numérica.
  - El kernel corresponde a las lineas 9-12.
- L13: crea en Python la función `HolamMundo` que corre en la GPU usando la función `HolaMundoGPU` del Kernel.
- L15: corre el Kernel `HolaMundoGPU` cargando los datos y la configuración de la GPU mediante la función de Python `HolamMundo`.
- L15: `block=(1,1,1)` representa las dimensiones del bloque de hilos que se correrán con las instrucciones del

hola-mundo-2.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n)
9  {
10   int i = threadIdx.x;
11   //if (i == n){
12     printf("Hola Mundo!, soy el hilo %d\\n",i);
13   //}
14 }
15 """)
16
17 HolaMundo = mod.get_function("HolaMundoGPU")
18
19 dimx = 3
20
21 HolaMundo(numpy.int32(6),block=(dimx,1,1))
22
23 get_attribute = pycuda.autoinit.device.
       get_attribute
24 print('MAX_BLOCKS_PER_MULTIPROCESSOR:',
       get_attribute(drv.device_attribute.
       MAX_BLOCKS_PER_MULTIPROCESSOR))
```

Salida:

```
1  maximo numero de hilos en dim x: 1024
2  Hola Mundo!, soy el hilo 6
```

Nota

L10: `threadIdx.x` permite recuperar el índice local del hilo que está ejecutando un Kernel.

hola-mundo-2.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n)
9  {
10   int i = threadIdx.x;
11   //if (i == n){
12     printf("Hola Mundo!, soy el hilo %d\\n",i);
13   //}
14 }
15 """)
16
17 HolaMundo = mod.get_function("HolaMundoGPU")
18
19 dimx = 3
20
21 HolaMundo(numpy.int32(6),block=(dimx,1,1))
22
23 get_attribute = pycuda.autoinit.device.
       get_attribute
24 print('MAX_BLOCKS_PER_MULTIPROCESSOR:',
       get_attribute(drv.device_attribute.
       MAX_BLOCKS_PER_MULTIPROCESSOR))
```

Salida:

```
1  maximo numero de hilos en dim x: 1024
2  Hola Mundo!, soy el hilo 6
```

Nota

L10: `threadIdx.x` permite recuperar el índice local del hilo que está ejecutando un Kernel.

L22: `get_attribute` permite obtener parámetros de la GPU. En https://documen.tician.de/pycuda/driver.html#pycuda.driver.device_attribute está la lista de atributos a solicitar.

hola-mundo-2.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n)
9  {
10   int i = threadIdx.x;
11   //if (i == n){
12     printf("Hola Mundo!, soy el hilo %d\\n",i);
13   //}
14 }
15 """)
16
17 HolaMundo = mod.get_function("HolaMundoGPU")
18
19 dimx = 3
20
21 HolaMundo(numpy.int32(6),block=(dimx,1,1))
22
23 get_attribute = pycuda.autoinit.device.
       get_attribute
24 print('MAX_BLOCKS_PER_MULTIPROCESSOR:',
       get_attribute(drv.device_attribute.
       MAX_BLOCKS_PER_MULTIPROCESSOR))
```

Salida:

```
1  maximo numero de hilos en dim x: 1024
2  Hola Mundo!, soy el hilo 6
```

Nota

L10: `threadIdx.x` permite recuperar el índice local del hilo que está ejecutando un Kernel.

L22: `get_attribute` permite obtener parámetros de la GPU. En https://documen.tician.de/pycuda/driver.html#pycuda.driver.device_attribute está la lista de atributos a solicitar.

L25: `MAX BLOCK DIM X` permite recuperar El máximo número de hilos en la dimensión x de un bloque.

hola-mundo-2.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n)
9  {
10   int i = threadIdx.x;
11   //if (i == n){
12     printf("Hola Mundo!, soy el hilo %d\\n",i);
13   //}
14 }
15 """)
16
17 HolaMundo = mod.get_function("HolaMundoGPU")
18
19 dimx = 3
20
21 HolaMundo(numpy.int32(6),block=(dimx,1,1))
22
23 get_attribute = pycuda.autoinit.device.
       get_attribute
24 print('MAX_BLOCKS_PER_MULTIPROCESSOR:',
       get_attribute(drv.device_attribute.
       MAX_BLOCKS_PER_MULTIPROCESSOR))
```

Salida:

```
1  maximo numero de hilos en dim x: 1024
2  Hola Mundo!, soy el hilo 6
```

Nota

L10: `threadIdx.x` permite recuperar el índice local del hilo que está ejecutando un Kernel.

L22: `get_attribute` permite obtener parámetros de la GPU. En https://documen.tician.de/pycuda/driver.html#pycuda.driver.device_attribute está la lista de atributos a solicitar.

L25: `MAX BLOCK DIM X` permite recuperar El máximo número de hilos en la dimensión x de un bloque.

Ejercicio

- Comente las lineas 11 y 13.

hola-mundo-3.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n){
9    int i = threadIdx.x; int j = threadIdx.y;
10     printf("Hola Mundo!, soy el hilo (%d,%d)\\n",i,j);
11 }
12 """)
13
14 HolaMundo = mod.get_function("HolaMundoGPU")
15
16 dimx = 16; dimy = 16
17
18 HolaMundo(numpy.int32(6),block=(dimx,dimy,1))
19
20 get_attribute = pycuda.autoinit.device.get_attribute
21 maxX = get_attribute(drv.device_attribute.MAX_BLOCK_DIM_X)
22 maxY = get_attribute(drv.device_attribute.MAX_BLOCK_DIM_Y)
23 print('Maximo numero de hilos en dim x,y:',maxX,maxY)
```

Salida:

```
1  Maximo numero de hilos en dim x,y: 1024 1024
2  Hola Mundo!, soy el hilo (0,0)
3  Hola Mundo!, soy el hilo (1,0)
4  ...
```

Nota

L10: `threadIdx.y` permite recuperar el índice local y del hilo que está ejecutando un Kernel.

hola-mundo-3.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n){
9    int i = threadIdx.x; int j = threadIdx.y;
10     printf("Hola Mundo!, soy el hilo (%d,%d)\\n",i,j);
11 }
12 """)
13
14 HolaMundo = mod.get_function("HolaMundoGPU")
15
16 dimx = 16; dimy = 16
17
18 HolaMundo(numpy.int32(6),block=(dimx,dimy,1))
19
20 get_attribute = pycuda.autoinit.device.get_attribute
21 maxX = get_attribute(drv.device_attribute.MAX_BLOCK_DIM_X)
22 maxY = get_attribute(drv.device_attribute.MAX_BLOCK_DIM_Y)
23 print('Maximo numero de hilos en dim x,y:',maxX,maxY)
```

Salida:

```
1  Maximo numero de hilos en dim x,y: 1024 1024
2  Hola Mundo!, soy el hilo (0,0)
3  Hola Mundo!, soy el hilo (1,0)
4  ...
```

Nota

L10: `threadIdx.y` permite recuperar el índice local y del hilo que está ejecutando un Kernel.

L22: `block=(dimx,dimx,1)` bloque que es un arreglo bi-dimensional de hilos.

hola-mundo-3.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4
5  import numpy
6
7  mod = SourceModule("""
8  __global__ void HolaMundoGPU(int n){
9    int i = threadIdx.x; int j = threadIdx.y;
10     printf("Hola Mundo!, soy el hilo (%d,%d)\\n",i,j);
11 }
12 """)
13
14 HolaMundo = mod.get_function("HolaMundoGPU")
15
16 dimx = 16; dimy = 16
17
18 HolaMundo(numpy.int32(6),block=(dimx,dimy,1))
19
20 get_attribute = pycuda.autoinit.device.get_attribute
21 maxX = get_attribute(drv.device_attribute.MAX_BLOCK_DIM_X)
22 maxY = get_attribute(drv.device_attribute.MAX_BLOCK_DIM_Y)
23 print('Maximo numero de hilos en dim x,y:',maxX,maxY)
```

Salida:

```
1  Maximo numero de hilos en dim x,y: 1024 1024
2  Hola Mundo!, soy el hilo (0,0)
3  Hola Mundo!, soy el hilo (1,0)
4  ...
```

Nota

L10: `threadIdx.y` permite recuperar el índice local y del hilo que está ejecutando un Kernel.

L22: `block=(dimx,dimx,1)` bloque que es un arreglo bi-dimensional de hilos.

Ejercicio

- Cree un arreglo tridimensional de hilos.
- Pruebe bloques de 8x8, 16x16, 32x32,64x64, 1024x1, 512x2 y 512x3.
- Muestre el valor de

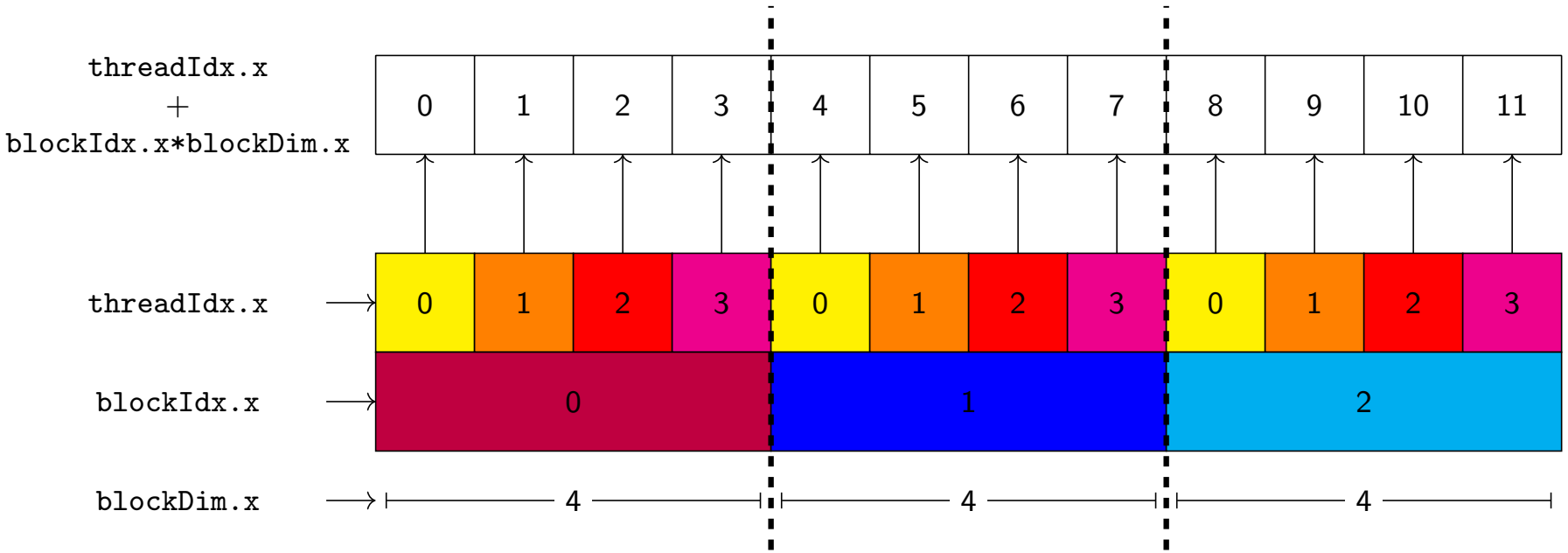
`get_attribute(drv.device_attribute.MAX_THREADS_PER_BLOCK)`

`get_attribute(drv.device_attribute.MULTIPROCESSOR_COUNT)`

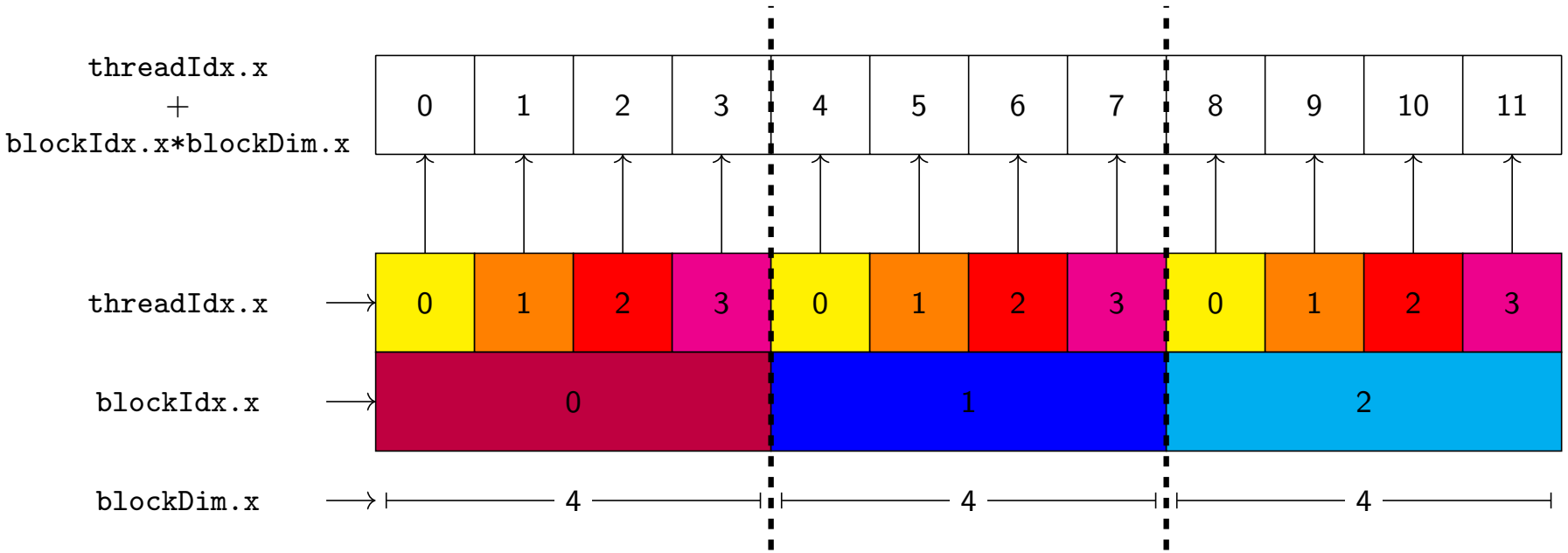
`get_attribute(drv.device_attribute.MAX_BLOCKS_PER_MULTIPROCESSOR)`[a]

[a] Versión $\geq$ 10.2

Podemos ver el proceso de asignación de un índice para cada hilo de la siguiente forma en el caso de una grilla con arreglo 1D de bloques, cada uno con un arreglo 1D de hilos:



Podemos ver el proceso de asignación de un índice para cada hilo de la siguiente forma en el caso de una grilla con arreglo 1D de bloques, cada uno con un arreglo 1D de hilos:



Ejercicio

A cada uno se le asignará un grupo (Bloque) y dentro del bloque tendrá un ID. Cada uno tendrá que calcular su ID global único y anotar en la pizarra la suma del dígito correspondiente de los vectores a y b.
$a = [8, 7, 6, 5, 4, 3, 2, 1]$, $b = [1, 1, 2, 3, 5, 8, 13, 21]$

grilla-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n){
8    int i = threadIdx.x+blockIdx.x*blockDim.x;
9      printf("Hola Mundo!, soy el hilo %d del bloque %d, pero
           mi ID es %d\\n",threadIdx.x,blockIdx.x,i);
10 }
11 """)
12 HolaMundo = mod.get_function("HolaMundoGPU")
13
14 gdim = (3,1,1)
15 bdim = (8,1,1)
16 HolaMundo(numpy.int32(0),block=bdim,grid=gdim)
17
18 get_attribute = pycuda.autoinit.device.get_attribute
19 hilosXmp = get_attribute(drv.device_attribute.
       MAX_THREADS_PER_MULTIPROCESSOR)
20 numdemp  = get_attribute(drv.device_attribute.
       MULTIPROCESSOR_COUNT)
21 THREADS_PER_BLOCK  = get_attribute(drv.device_attribute.
       MAX_THREADS_PER_BLOCK)
22 print('Hilos por Multiprocesado:',hilosXmp)
23 print('Numero de  multiprocesadores:',numdemp)
24 print('Numero de  hilos por bloques:',THREADS_PER_BLOCK)
```

Salida:

```
1  Hilos por Multiprocesado: 2048
2  Numero de  multiprocesadores: 2
3  Hola Mundo!, soy el hilo 352 del bloque 5, pero mi ID es
       2912
4  ...
```

Nota

L10: `blockIdx.x` permite recuperar el índice (en el eje x) del bloque en grilla donde está el hilo actual.

grilla-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n){
8    int i = threadIdx.x+blockIdx.x*blockDim.x;
9      printf("Hola Mundo!, soy el hilo %d del bloque %d, pero
           mi ID es %d\\n",threadIdx.x,blockIdx.x,i);
10 }
11 """)
12 HolaMundo = mod.get_function("HolaMundoGPU")
13
14 gdim = (3,1,1)
15 bdim = (8,1,1)
16 HolaMundo(numpy.int32(0),block=bdim,grid=gdim)
17
18 get_attribute = pycuda.autoinit.device.get_attribute
19 hilosXmp = get_attribute(drv.device_attribute.
       MAX_THREADS_PER_MULTIPROCESSOR)
20 numdemp  = get_attribute(drv.device_attribute.
       MULTIPROCESSOR_COUNT)
21 THREADS_PER_BLOCK  = get_attribute(drv.device_attribute.
       MAX_THREADS_PER_BLOCK)
22 print('Hilos por Multiprocesado:',hilosXmp)
23 print('Numero de  multiprocesadores:',numdemp)
24 print('Numero de  hilos por bloques:',THREADS_PER_BLOCK)
```

Salida:

```
1  Hilos por Multiprocesado: 2048
2  Numero de  multiprocesadores: 2
3  Hola Mundo!, soy el hilo 352 del bloque 5, pero mi ID es
       2912
4  ...
```

Nota

L10: `blockIdx.x` permite recuperar el índice (en el eje x) del bloque en grilla donde está el hilo actual.

L10: `blockDim.x` dimensión del bloque (en el eje x).

grilla-1.py

```
1  import pycuda.autoinit
2  from pycuda.compiler import SourceModule
3  import pycuda.driver as drv
4  import numpy
5
6  mod = SourceModule("""
7  __global__ void HolaMundoGPU(int n){
8    int i = threadIdx.x+blockIdx.x*blockDim.x;
9      printf("Hola Mundo!, soy el hilo %d del bloque %d, pero
           mi ID es %d\\n",threadIdx.x,blockIdx.x,i);
10 }
11 """)
12 HolaMundo = mod.get_function("HolaMundoGPU")
13
14 gdim = (3,1,1)
15 bdim = (8,1,1)
16 HolaMundo(numpy.int32(0),block=bdim,grid=gdim)
17
18 get_attribute = pycuda.autoinit.device.get_attribute
19 hilosXmp = get_attribute(drv.device_attribute.
       MAX_THREADS_PER_MULTIPROCESSOR)
20 numdemp  = get_attribute(drv.device_attribute.
       MULTIPROCESSOR_COUNT)
21 THREADS_PER_BLOCK  = get_attribute(drv.device_attribute.
       MAX_THREADS_PER_BLOCK)
22 print('Hilos por Multiprocesado:',hilosXmp)
23 print('Numero de  multiprocesadores:',numdemp)
24 print('Numero de  hilos por bloques:',THREADS_PER_BLOCK)
```

Salida:

```
1  Hilos por Multiprocesado: 2048
2  Numero de  multiprocesadores: 2
3  Hola Mundo!, soy el hilo 352 del bloque 5, pero mi ID es
       2912
4  ...
```

Nota

L10: `blockIdx.x` permite recuperar el índice (en el eje x) del bloque en grilla donde está el hilo actual.

L10: `blockDim.x` dimensión del bloque (en el eje x).

Ejercicio

- Calcule el máximo número de hilos que pueden correr en paralelo en su GPU
- Corra todos los hilos de su GPU usando 8,16 y 32 bloques.

1. Realice un verificador que entregue un mensaje de error si el usuario está haciendo una asignación incorrecta de hilos, bloques y grilla, de acuerdo a las características de la GPU utilizada.
   Ejemplo: En una GPU con

   - `MAX_THREADS_PER_BLOCK` =1024
   - `MAX_BLOCK_DIM_X` =1024
   - `MAX_BLOCK_DIM_Y` =1024
   - `MAX_BLOCK_DIM_Z` =64

   Entrada

```
1 [1024,2,2]
2 [2,2,128]
3 [8,8,8]
```

   Salida

```
1 Excede el total de hilos (max
      1024)
2 Excede la dimension Z (max 64)
3 Se puede.
```

2. Cree el fichero `grilla2.py` el cual debe trabajar con una grilla bi dimensional de bloques bi dimensionales y entregar la siguiente salida por hilo:

```
1 Hola Mundo!, soy el hilo (0,0) del bloque (0,0), pero mi ID es (0,0) y mi
      IDG es 0.
```

   donde "hilo (0,0)" es el id del hilo dentro del bloque, "ID es (0,0)" es un ID único para cada hilo por componente y "IDG es 0" es un identificador único para cada hilo.