

VERSIÓN PRELIMINAR

# Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

5 de septiembre de 2024

VERSIÓN PRELIMINAR

**Título:** Python en Ámbitos Científicos  
**Autores:** Facundo Batista & Manuel Carlevaro  
**ISBN-13 (versión electrónica):** ???-?-???-???-?  
© Facundo Batista & Manuel Carlevaro  
**Primera Edición (versión preliminar)**  
Escrito con X<sub>3</sub>LaTeX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](#)  
Lugar: Olivos y La Plata, Buenos Aires, Argentina  
Año: 2024  
Web: <http://pyciencia.taniquetil.com.ar/>

10 9 8 7 6 5 4 3 2 1

## Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científicos, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos" que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquivo la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

*Olivos y La Plata, Buenos Aires, Argentina,*

---

Facundo Batista & Manuel Carlevaro

# Índice general

Prefacio	2
Índice general	3
I Python	4
II Herramientas fundamentales	5
III Temas específicos	6
1. Procesamiento en GPU	7
1.1. Arquitectura de una GPU . . . . .	9
1.2. Kernels y threads en CUDA . . . . .	10
1.3. Organización de <i>threads</i> . . . . .	12
1.4. Producto de matrices en GPU con PyCUDA . . . . .	17
1.5. Producto de matrices con PyOpenCL . . . . .	22
1.6. Lecturas recomendadas . . . . .	25
IV Apéndices	26
A. Zen de Python	27
Bibliografía	28

# Parte I Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

## Parte II

### Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

## Parte III


### Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte [II](#).

# 1 | Procesamiento en GPU

En el ámbito del cálculo científico y tecnológico, la demanda de potencia de cálculo es siempre creciente, dada la necesidad de resolver problemas cada vez más complejos, más grandes, más rápido.

Afortunadamente, la industria ha respondido a tales requerimientos con adelantos tecnológicos formidables, tal como lo muestra la ley empírica de Moore<sup>1</sup>. Hasta el fin del siglo pasado, las computadoras incrementaban su potencia de cálculo principalmente por medio del aumento de la frecuencia del reloj (también mejorando el tamaño de las memorias caché y el conjunto de instrucciones de los procesadores), lo que permitía realizar más operaciones aritméticas o lógicas por segundo. Sin embargo, esto requería disipar cantidades mayores de calor, lo que impone un límite físico a la velocidad de los relojes. Por este motivo, desde mediados de los 2000, se comenzaron a implementar estrategias de paralelismo incorporando más núcleos de cálculo en las computadoras. Esto se evidencia claramente en la Figura 1.1.



Módulo	Versión
NumPy	1.26.4
PyOpenCL	2024.2.7
PyCUDA	2024.1.2

[Código disponible](#)

Por otra parte, las placas gráficas fueron desarrolladas inicialmente como *hardware* especializado en acelerar el procesamiento gráfico de aplicaciones científicas o de ingeniería, aunque ya en la última década del siglo pasado se produjeron adelantos tecnológicos significativos traccionados principalmente por la industria de los juegos de video y la demanda creciente de gráficos 3D. En 1999, la empresa NVIDIA<sup>2</sup> presentó la GeForce 256 como la primera GPU (*Graphics Processing Unit*) más que una aceleradora de gráficos, y desde entonces los avances en la tecnología de estas placas se han sostenido a un ritmo vertiginoso, liderados por compañías como NVIDIA, AMD<sup>3</sup> e Intel<sup>4</sup>.

Debido a la naturaleza inherentemente paralela del cálculo de grandes cantidades de píxeles, investigadores y desarrolladores extendieron el uso de las GPUs más allá de las aplicaciones gráficas al ámbito del procesamiento general, en los primeros años del siglo XXI. En particular, algunos problemas asociados a operaciones sobre vectores y matrices fueron adaptados fácilmente a su ejecución en GPU [2, 3, 4], logrando realizar dichas operaciones en menos tiempo que sobre una CPU. Al desarrollo en el *hardware* lo acompañó la mejora en los lenguajes de programación, lo que facilitó significativamente la representación de estructuras y algoritmos sobre las placas

<sup>1</sup> Ver su entrada en [Wikipedia](#).  
<sup>2</sup> <https://www.nvidia.com/es-la/>.  
<sup>3</sup> <https://www.amd.com/es.html>.  
<sup>4</sup> <https://www.intel.la/content/www/xl/es/homepage.html>.



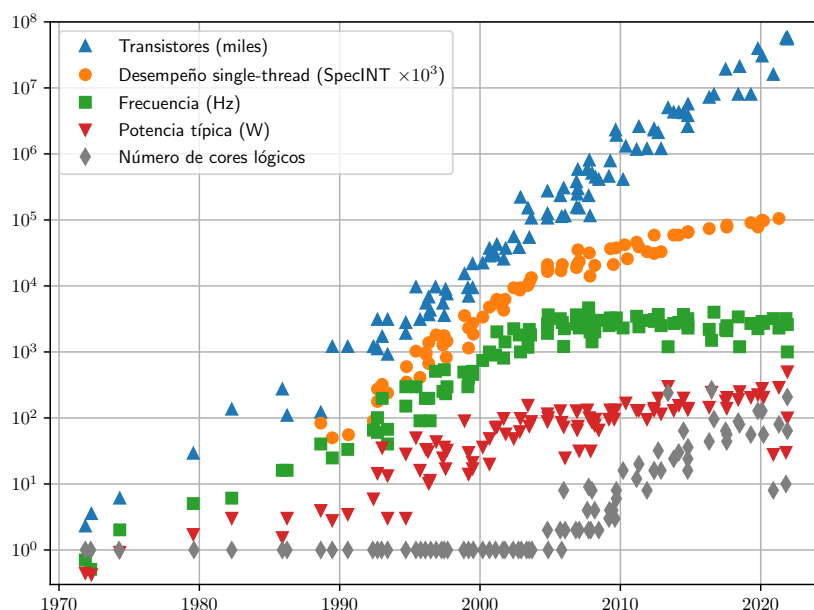


FIGURA 1.1: Evolución de 50 años de microprocesadores. Fuente: [Karl Rupp](#).

gráficas por medio de abstracciones orientadas hacia el cálculo paralelo masivo, más que hacia los detalles de cálculo gráfico (los primeros desarrollos se hicieron expresando los cómputos como una forma de «pintar» un píxel). El lenguaje de programación sobre GPUs más difundido actualmente es CUDA [5], introducido por NVIDIA en 2006, como una extensión del lenguaje C/C++. Una alternativa que constituye un estándar abierto y multiplataforma es OpenCL<sup>5</sup> del Kronos Group, desarrollado inicialmente por Apple Inc. en 2009.

Una vez consolidado el desarrollo del *hardware* y los lenguajes para utilizarlos, la comunidad científica y tecnológica adoptó rápidamente la enorme capacidad de cálculo de bajo costo disponible, y el número de aplicaciones ejecutadas sobre GPUs comenzó a crecer sostenidamente. Un ejemplo notable de los primeros esfuerzos por aprovechar el paralelismo masivo provisto por las placas gráficas fue el trabajo de Raina, Madhavan y Ng en 2009 sobre la posibilidad de entrenar redes neuronales profundas con GPUs[6].

Existen varias herramientas en el ecosistema de Python que permiten acceder al procesamiento en una placa gráfica, abordando de diversas maneras las formas de delegar en una GPU parte del procesamiento que realizamos en un código. Por ejemplo, podemos mencionar Numba<sup>6</sup>, CuPy<sup>7</sup>, PyCUDA<sup>8</sup> o PyOpenCL<sup>9</sup>. En este capítulo haremos una breve introducción a las dos últimas, con algo más de detalle sobre PyCUDA.

<sup>5</sup> <https://www.khronos.org/opencl/>.

<sup>6</sup> <https://numba.readthedocs.io/en/stable/cuda/index.html>.

<sup>7</sup> <https://cupy.dev/>.

<sup>8</sup> <https://documen.tician.de/pycuda/>.

<sup>9</sup> <https://documen.tician.de/pyopencl/>.

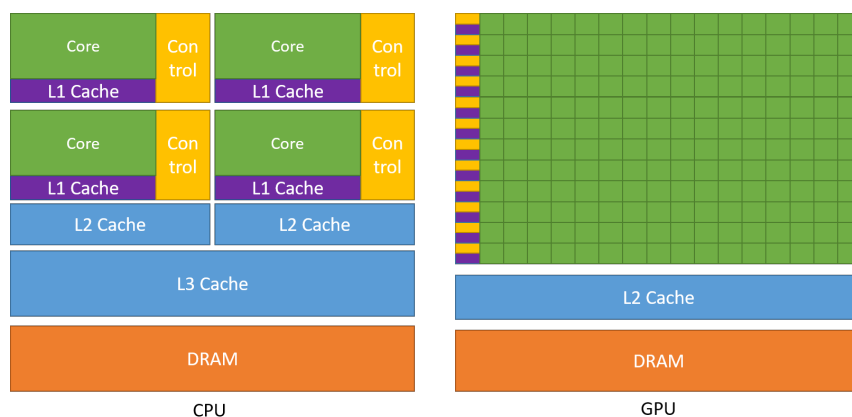


FIGURA 1.2: Comparación de arquitecturas de CPU y GPU: en verde se representan las unidades de cómputo o núcleos (*cores*), en amarillo las unidades de control y el resto (violeta, celeste y anaranjado) diferentes jerarquías de memoria. Fuente: [CUDA C++ Programming Guide](#).

## 1.1. Arquitectura de una GPU

En una CPU actual podemos encontrar varios (decenas) núcleos de cómputo (o *cores*), que son procesadores capaces de realizar tareas secuenciales complejas en tiempos muy cortos gracias a su conjunto de instrucciones. Por ejemplo, con el propósito de optimizar el tiempo de ejecución, una CPU puede ejecutar instrucciones en un orden diferente al establecido por el programa (*out-of-order executions*), o puede predecir las instrucciones más probables a ejecutar en el futuro próximo cuando se encuentra una bifurcación (*multiple branch prediction*). De este modo puede anticiparse preparando los operandos correspondientes y ejecutando con anticipación tales instrucciones (*speculative execution*).

Por su parte, las GPUs proveen una plataforma especializada en cálculo paralelo de operaciones con números representados en punto flotante. Los núcleos de las GPUs tienen un conjunto de instrucciones más simple, por lo que cada uno puede realizar tareas menos complejas y en forma más lenta que una CPU. Sin embargo, una GPU actual dispone de miles de núcleos<sup>10</sup> trabajando simultáneamente con muy poco *overhead*, lo que le permite procesar datos hasta cientos de veces más rápido que una CPU con varios núcleos. Aprovechando el paralelismo masivo que proveen las GPUs, el modelo de programación que utilizan es el SIMT (*Single Instruction, Multiple Threads*) que significa que todos los núcleos ejecutan exactamente la misma operación sobre datos diferentes (SIMD, *Single Instruction, Multiple Data*), combinado con el uso de múltiples hilos de ejecución.

En el caso de las GPUs de NVIDIA, la arquitectura de una GPU dada depende de la familia de dispositivos a la que pertenece. Cada GPU se caracteriza por su capacidad de cálculo o *compute capability*, que se representa por números de revisión mayor y menor ( $X, Y$ ). Los dispositivos que comparten el mismo número de revisión mayor  $X$  pertenecen a la misma arquitectura (Kepler: 3, Maxwell: 5, Pascal: 6, Volta y Turing: 7, Ampere: 8 y Hopper: 9), mientras que el número de revisión menor  $Y$  corresponde a mejoras incrementales dentro de la misma arquitectura. Las especificaciones técnicas de cada capacidad de cálculo se pueden ver en las tablas que ofrece

<sup>10</sup> El modelo GeForce RTX 4090 de NVIDIA dispone de 16 384 CUDA *cores*.

NVIDIA ([aquí](#) y [aquí](#)).

Todas estas arquitecturas se basan en un conjunto de *streaming multiprocessors* (SM), cada uno de los cuales controla la ejecución de un número de hilos o *threads*. Para ello, los SM disponen (según su capacidad de cálculo) de un número de CUDA *cores* para operaciones aritméticas con números enteros o de diferentes grados de precisión de punto flotante, unidades de procesamiento para el cálculo de funciones trascendentales con precisión simple, unidades de cálculo tensorial y procesadores para la administración de los hilos, además de una cantidad limitada memoria *on-chip* que es accesible por los *cores* que administra, y un gran número de registros.

Estos SM son capaces de crear, administrar y ejecutar grupos de 32 hilos que se ejecutan simultáneamente denominados *warps*. Cada *warp* ejecuta una instrucción común a los hilos que administra en forma simultánea. Cuando un SM recibe un bloque de hilos para ejecutar, los divide en *warps* que contienen hilos con identificadores (ID) consecutivos.

## 1.2. Kernels y threads en CUDA

La plataforma de programación provista por NVIDIA denominada CUDA (*Compute Unified Device Architecture*) permite escribir programas para ejecutar en sus GPUs, y consiste en una extensión del lenguaje C/C++ que ofrece un capa de *software* que permite el acceso directo al conjunto de instrucciones y núcleos de cálculo paralelo para la ejecución de funciones denominadas «*kernels*». Estos kernels se ejecutan en la GPU, en forma paralela, utilizando un número  $N$  establecido de hilos o *threads*, sobre arreglos (*arrays*) de números que reciben como argumento. Cada thread que ejecuta el kernel tiene un identificador único que es accesible por el kernel a través de variables propias del lenguaje: `threadIdx`. Este identificador puede ser utilizado para calcular y acceder a elementos de los arrays.

El modelo de programación en CUDA para ejecutar código en la GPU consiste, básicamente, en proceder en la ejecución de las siguientes tareas:

- Definir el código que compone el kernel, proceder a su compilación y generar una función que será invocada desde nuestro programa en la CPU (o *host*).
- Copiar los datos que constituyen los argumentos del kernel desde la memoria del *host* a la memoria de la GPU (o *device*), o transferencia *host* a *device*.
- Lanzar el kernel en la GPU, con una determinada configuración de la división de tareas.
- Copiar los datos resultantes desde la memoria del *device* a la del *host*, o transferencia *device* to *host*.

Veamos un ejemplo simple que consiste en la suma de dos arrays, utilizando PyCUDA [7]. Primero importamos los módulos necesarios para acceder a la GPU y los inicializamos:

```
import pycuda.driver as driver
import pycuda.autoinit
```

Los primeros dos módulos permiten establecer la comunicación con la GPU y realizar los pasos necesarios para enviarle kernels para su ejecución. El submódulo `driver` contiene funciones para la administración de la memoria, propiedades del dispositivo, etc., mientras que el

submódulo `autoinit` se utiliza para la inicialización de la GPU, creación de contexto y limpieza de memoria. Este submódulo no es obligatorio, ya que estas funciones se pueden hacer en forma explícita.

En la celda siguiente vamos a generar el código que constituye el *kernel*. Para esto debemos importar módulo `pycuda.compiler`, que es el que crea un módulo ejecutable en la GPU a partir del código fuente que le suministramos en forma de cadena, haciendo uso del compilador de NVIDIA «nvcc». A continuación definimos el *kernel* definiendo una variable `mod` que contiene el módulo con la definición del kernel. Éste se pasa como argumento al constructor de la clase compuesto por una cadena, que define la función a ejecutar sobre los arrays escrita con sintaxis de C/C++:

CELL 02

```
from pycuda.compiler import SourceModule
mod = SourceModule("""
    __global__ void producto_arrays(float *a, float *b, float *c)
    {
        const int i = threadIdx.x;
        c[i] = a[i] * b[i];
    }
""")
producto_arrays = mod.get_function("producto_arrays")
```

La definición del *kernel* comienza con la palabra `__global__`, que se utiliza para declarar que la función que sigue a continuación es un kernel que se ejecuta en la GPU, invocable desde el *host* (para arquitecturas con capacidad de cómputo 5.0 o superior, la función también se puede invocar desde el *device*). Una función `__global__` debe devolver un tipo `void`, que es la palabra que precede al nombre de la función (`producto_arrays`). Luego, los tres arrays que constituyen los argumentos se pasan como punteros a datos de tipo `float`.

En el cuerpo de la función, delimitado por llaves, la variable entera `i` contiene el valor de `threadIdx.x`, que es el identificador del hilo que ejecuta el kernel. De este modo, cada hilo accede al elemento *i*-ésimo de cada array `a` y `b`, y almacena la suma de ambos en el elemento *i*-ésimo de `c`, paralelizando de este modo la suma sobre los arrays.

La variable `producto_arrays` recibe un *handler* a la función que contiene el módulo `mod`. En la celda siguiente definimos los tres arrays que utilizaremos: `a` y `b`, de tipo `np.float32`, y que contienen valores aleatorios mientras que `c`, del mismo tipo de `a`, se inicializa con ceros.

CELL 03

```
import numpy as np

n_elementos = 400
a = np.random.randn(n_elementos).astype(np.float32)
b = np.random.randn(n_elementos).astype(np.float32)
c = np.zeros_like(a)
```

Ahora lanzamos la ejecución de hilos en la GPU utilizando el *handler* `producto_arrays`, al que le pasamos como argumentos los dos arrays que vamos a sumar, y el array en el que almacenamos el resultado. La transferencia de estos arrays desde y hacia el *device* lo hacemos con los métodos `In`, que transfiere el array al *device* antes de ejecutar el kernel, y `Out`, que indica que el array debe transferirse desde el *device* después de que el kernel finaliza su tarea. Otro parámetro que debemos pasar es `block`, que indica el modo en que se distribuirán las tareas en la GPU y que

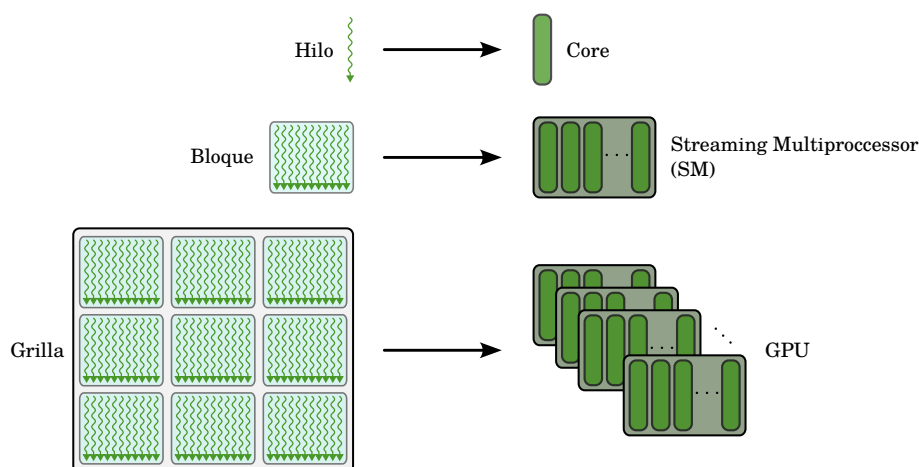


FIGURA 1.3: Jerarquía de hilos en CUDA.

describiremos en la sección siguiente. Finalmente, comparamos los valores almacenados en `c` con el cálculo que realiza NumPy para verificar que sean iguales.

CELL 04	
<pre>producto_arrays(driver.In(a), driver.In(b), driver.Out(c), block=(400,1,1)) np.allclose(c, a * b, rtol=1e-05, atol=1.0e-03)</pre>	
True	

### 1.3. Organización de *threads*

La unidad de ejecución de kernels es el hilo o *thread*, y lo usual es utilizar un gran número de *threads* que debemos gestionar para aprovechar la cantidad de *cores* que nos provee la GPU. Un grupo de hilos se denomina «bloque», y a su vez un conjunto de bloques define una «grilla», tal como se esquematiza en la Figura 1.3. Cada bloque se ejecuta en un *streaming multiprocessor* (salvo excepciones, un bloque no se puede migrar a otro SM) en forma concurrente, según los requerimientos de recursos que necesita. De esta forma, dado que los hilos de un mismo bloque comparten los recursos que provee el SM, dichos hilos pueden comunicarse entre sí a través de la memoria compartida del SM, por medio de barreras de sincronización u operaciones atómicas.

El número de hilos que pueden agruparse en un bloque está limitado, y por esto es posible agrupar bloques en una grilla para hacer posible la ejecución de una gran cantidad de hilos. La división de tareas en un procesamiento en paralelo debe planificarse teniendo en cuenta la cantidad de hilos que puede administrar un bloque, lo que en última instancia depende de la GPU (o múltiples GPUs) que tengamos disponible. En el código siguiente vemos cómo podemos interrogar a la GPU para obtener información sobre los recursos que ofrece:

CELL 01

```
import pycuda.driver as cuda
import pycuda.autoinit

print(f'{cuda.Device.count()} dispositivo(s) encontrado.')
```

```
device = cuda.Device(0)
print(f'Modelo de GPU: {device.name()}')
```

```
print(' Compute Capability: %d.%d' % device.compute_capability())
print(f' Memoria total: {device.total_memory()//((1024)**2)} MB')
```

```
atts = [(str(att), value) for att, value in device.get_attributes().items()]
atts.sort()

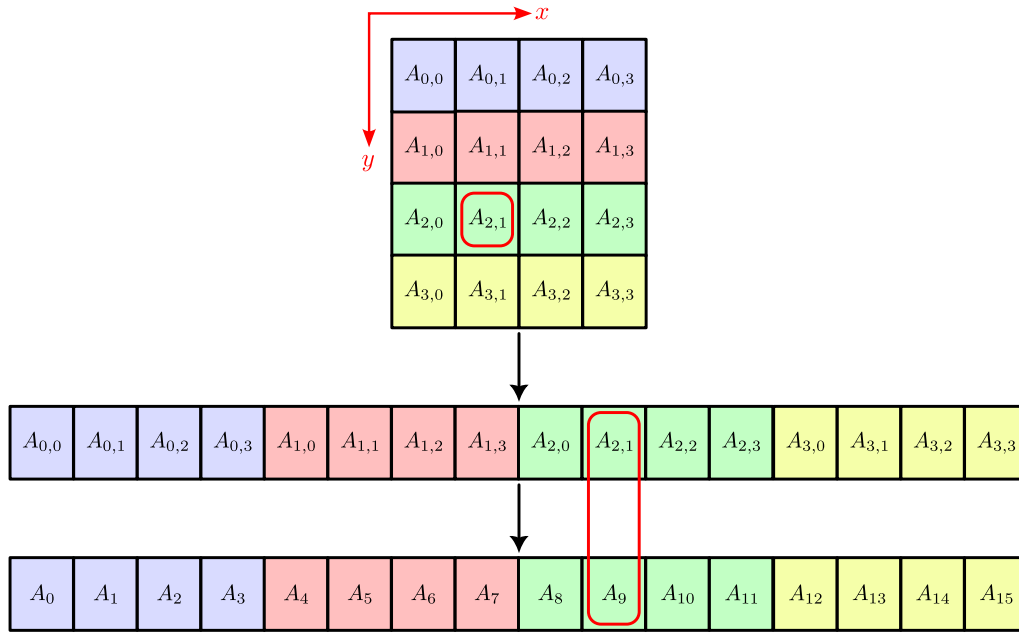
for att, value in atts:
    print(' %s: %s' % (att, value))
```

La salida de esta celda es extensa, por lo que resumimos aquí algunas líneas:

```
1 dispositivo(s) encontrado.
Modelo de GPU: NVIDIA GeForce GTX 1060 3GB
  Compute Capability: 6.1
  Memoria total: 3013 MB
...
MAX_BLOCKS_PER_MULTIPROCESSOR: 32
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
MAX_THREADS_PER_MULTIPROCESSOR: 2048
...
MULTIPROCESSOR_COUNT: 9
WARP_SIZE: 32
```

Aquí podemos ver que en la computadora donde se ejecutó este notebook existe solo una placa cuyo modelo es «NVIDIA GeForce GTX 1060 3GB», que soporta una capacidad de cálculo 6.1 y dispone de una memoria total de 3.013 MB. Hacia el final de la salida de la celda, confirmamos que el tamaño de warp es 32 (cantidad que puede cambiar en futuras versiones del *hardware*), y que la GPU dispone de 9 SM. Además, el número máximo de bloques que puede ejecutar cada SM es 32. Si corremos el mismo código en Google Colab, obtenemos la siguiente salida:

```
1 dispositivo(s) encontrado.
Modelo de GPU: Tesla T4
  Compute Capability: 7.5
  Memoria total: 15102 MB
...
MAX_BLOCKS_PER_MULTIPROCESSOR: 16
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
```

FIGURA 1.4: Identificación del índice de un hilo a partir de su `threadIdx` según el orden por filas.

```

MAX_THREADS_PER_MULTIPROCESSOR: 1024
...
MULTIPROCESSOR_COUNT: 40
WARP_SIZE: 32

```

es decir que la plataforma nos asigna una GPU con mayor capacidad de cálculo (7.5), la mitad de bloques por SM (16) pero más de 4 veces la cantidad de streaming multiprocessors (40 SM), y cinco veces la memoria de la placa anterior.

Por otra parte, podemos notar que los máximos valores para las dimensiones de bloques y grilla se dividen en `_X`, `_Y` y `_Z`, lo que sugiere que estas categorías pueden adquirir representaciones tridimensionales. De hecho, la Figura 1.3 muestra un arreglo bidimensional de bloques en una grilla, y un arreglo unidimensional de hilos en un bloque.

Efectivamente, la variable `threadIdx` es un vector de tres componentes, de modo que los hilos se pueden identificar utilizando un índice unidimensional, bidimensional o tridimensional, lo que genera un bloque de threads de una, dos o tres dimensiones respectivamente, permitiendo de este modo una representación natural de dominios de cómputo como vectores, matrices o volumen. Podemos acceder a cada componente del vector con `threadIdx.x`, `threadIdx.y` y `threadIdx.z`. La definición de un bloque, entonces, queda determinada por una tupla de tres enteros que indican la cantidad de hilos en cada dimensión. Por ejemplo, un bloque bidimensional de 5 hilos en  $x$  y 4 hilos en  $y$  se indica con `block=(5, 4, 1)`.

En consecuencia, la lógica para determinar el índice de un hilo a partir de su `threadIdx` en un bloque, se puede realizar como se esquematiza en la Figura 1.4. En ese caso, pasamos de una representación bidimensional a una unidimensional «concatenando» consecutivamente cada fila (orden por fila o *raw-major order*). Específicamente, para un bloque de dimensiones `blockDim.x` en la dirección  $x$  y `blockDim.y` en la dirección  $y$ , el índice de cada hilo resulta  $\text{threadIdx.x} + \text{blockDim.x} \times \text{threadIdx.y}$  ( $1 + 4 \times 2 = 9$  en el elemento resaltado de la figura).

Esta forma de identificar hilos en un bloque puede generalizarse a una y tres dimensiones:

- 1D:  $\text{threadID} = \text{threadIdx.x}$
- 2D:  $\text{threadID} = \text{threadIdx.x} + \text{blockDim.x} \times \text{threadIdx.y}$
- 3D:  $\text{threadID} = \text{threadIdx.x} + \text{blockDim.x} \times \text{threadIdx.y} + \text{blockDim.x} \times \text{blockDim.y} \times \text{threadIdx.z}$

Tal como comentamos anteriormente, existe un límite en la cantidad de hilos que se pueden agrupar en un bloque, dadas las limitaciones de memoria y cantidad de núcleos de cálculo que dispone un SM. Actualmente ese límite es de 1.024 hilos por bloque. Sin embargo, es posible ejecutar un kernel sobre muchos bloques que tengan la misma forma (o dimensiones en cada «eje»), de modo que el número total de hilos en los que se puede desplegar un cálculo es el número de hilos por bloque multiplicado por el número de bloques.

Al igual que los hilos en un bloque, los bloques pueden organizarse en un arreglo unidimensional, bidimensional o tridimensional, formando de este modo una grilla. La cantidad de bloques en la grilla se suele determinar por la cantidad de datos a procesar, que por lo general excede la cantidad de procesadores en la GPU. De la misma forma que para los bloques, la definición de la grilla requiere de una tupla de tres enteros. Durante la ejecución de un kernel, cada bloque de la grilla se puede identificar mediante las variables `blockIdx.x`, `blockIdx.y` y `blockIdx.z`, mientras que las dimensiones de la grilla se pueden acceder mediante `blockDim.x`, `blockDim.y` y `blockDim.z`. Finalmente, se puede asignar un índice a cada hilo a partir de su ubicación dentro de un bloque y la localización del bloque en la grilla. Veamos un ejemplo en el siguiente código:

CELL 01

```
import pycuda.driver as driver
import pycuda.autoinit
```

CELL 02

```
from pycuda.compiler import SourceModule

mod = SourceModule(r"""
    #include <stdio.h>

    __global__ void t_ident()
    {
        int idx = threadIdx.x + blockDim.x * threadIdx.y + blockDim.x * blockDim.y * threadIdx.z;
        int block_id = blockIdx.x + gridDim.x * (blockIdx.y + blockIdx.z * gridDim.y);
        int thread_id = idx + block_id * (blockDim.x * blockDim.y);
        printf("%9d %11d %11d %11d %10d %10d %10d \n",
            thread_id, threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y, blockIdx.z);
    }
""")

func = mod.get_function("t_ident")
```

En la celda 1 importamos los módulos necesarios e inicializamos la comunicación con la GPU. En la segunda celda definimos el kernel que se ejecutará en el *device*. Esta función no recibe argumentos, pues solo realizará operaciones sobre variables internas provistas por la GPU. La



primera línea de la función determina un índice que identifica cada hilo dentro de un bloque (`idx`), y la línea siguiente identifica el bloque que se está ejecutando (`block_id`), por último, combinando los índices anteriores, podemos asignar un identificador único al hilo en ejecución (`thread_id`), y la tarea de la función será mostrar los valores de todas estas variables. En la celda siguiente, luego de mostrar en la pantalla un encabezamiento, invocamos al *handler* de nuestro módulo pasándole como argumentos dos parámetros, `block` y `grid` que proveen el contexto en el que se ejecutará el kernel en el *device*. Podemos ahora lanzar un total de 16 hilos organizados en una grilla con un solo bloque, compuesto por un arreglo bidimensional de 4 hilos por dimensión, tal como se muestra en la celda 3:

CELL 03						
<pre>print('thread_id threadIdx.x threadIdx.y threadIdx.z blockIdx.x blockIdx.y blockIdx.z') func(block=(4, 4, 1), grid=(1, 1, 1))</pre>						
thread_id	threadIdx.x	threadIdx.y	threadIdx.z	blockIdx.x	blockIdx.y	blockIdx.z
0	0	0	0	0	0	0
1	1	0	0	0	0	0
2	2	0	0	0	0	0
3	3	0	0	0	0	0
4	0	1	0	0	0	0
5	1	1	0	0	0	0
6	2	1	0	0	0	0
7	3	1	0	0	0	0
8	0	2	0	0	0	0
9	1	2	0	0	0	0
10	2	2	0	0	0	0
11	3	2	0	0	0	0
12	0	3	0	0	0	0
13	1	3	0	0	0	0
14	2	3	0	0	0	0
15	3	3	0	0	0	0

Claro que según resulte conveniente para el problema que queremos descomponer en múltiples procesos, podemos organizar la misma cantidad de hilos en forma diferente. Por ejemplo, en la siguiente celda vemos cómo lanzamos el mismo kernel pero ahora en una grilla bidimensional de dos bloques por dimensión, conteniendo un arreglo unidimensional de cuatro hilos cada bloque:

CELL 04

```
print('thread_id threadIdx.x threadIdx.y threadIdx.z blockIdx.x blockIdx.y blockIdx.z')
func(block=(4, 1, 1), grid=(2, 2, 1))
```

thread_id	threadIdx.x	threadIdx.y	threadIdx.z	blockIdx.x	blockIdx.y	blockIdx.z
4	0	0	0	1	0	0
5	1	0	0	1	0	0
6	2	0	0	1	0	0
7	3	0	0	1	0	0
12	0	0	0	1	1	0
13	1	0	0	1	1	0
14	2	0	0	1	1	0
15	3	0	0	1	1	0
0	0	0	0	0	0	0
1	1	0	0	0	0	0
2	2	0	0	0	0	0
3	3	0	0	0	0	0
8	0	0	0	0	1	0
9	1	0	0	0	1	0
10	2	0	0	0	1	0
11	3	0	0	0	1	0

Observamos en la última salida, a diferencia de la celda anterior, que los hilos no aparecen en orden, lo que es muy común que ocurra al lanzar procesos en forma concurrente, que es lo que sucede a nivel de bloque. No sabemos de antemano en qué orden se ejecutarán los cuatro bloques que lanzamos.

## 1.4. Producto de matrices en GPU con PyCUDA

El producto de matrices, operación ubicua en métodos numéricos, constituye un cálculo muy conveniente para realizar en forma paralela, por lo que implementaremos este cálculo para dar un ejemplo práctico de uso de la GPU.

Nuestro objetivo es calcular el producto:

$$C = A \cdot B, \quad A \in \mathbb{R}^{m \times l}, \quad B \in \mathbb{R}^{l \times n} \text{ y } C \in \mathbb{R}^{m \times n}$$

Cada elemento  $c_{ij}$  de  $C$  resulta de realizar la operación:

$$c_{ij} = \sum_{k=1}^l a_{ik} b_{kj}$$

Vemos de aquí que el cálculo de  $c_{ij}$  es independiente del resto de los valores de los elementos de  $C$ , por lo que es posible paralelizar de una manera muy simple el cálculo asignando a cada hilo el cómputo de  $c_{ij}$ . Además, considerando la estructura «bidimensional» de las matrices involucradas, la representación natural para los hilos es agruparlos en forma bidimensional en los bloques.

Como mencionamos anteriormente, cada bloque puede contener un número limitado de hilos, que según hemos visto en los ejemplos previos es 1.024. Una posibilidad consiste en organizar nuestros bloques de forma que contengan un arreglo cuadrado de 32 hilos por fila y 32 hilos por

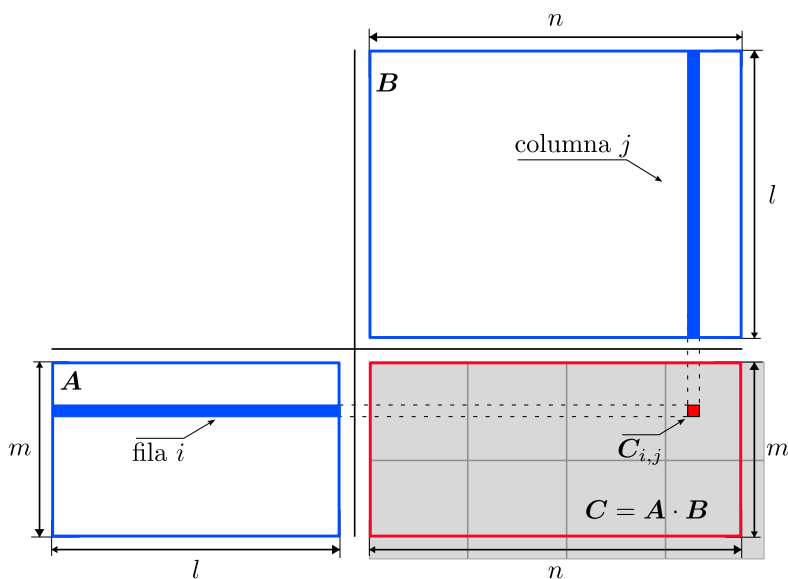


FIGURA 1.5: Esquema de procesamiento de hilos en el producto de matrices. En gris, sobre la matriz  $C$ , se representa una grilla de bloques que cubren (en exceso) las dimensiones de  $C$ .

columna, lo que nos lleva a considerar la necesidad de utilizar varios bloques de hilos si nuestras matrices superan estas dimensiones. La Figura 1.5 esquematiza esta situación, en la que la matriz  $C$  es cubierta con un arreglo de bloques cuadrados.

En nuestro ejemplo vamos a multiplicar matrices con  $m = 4000$ ,  $l = 6000$  y  $n = 2000$ , lo que da un total de 8 millones de hilos para realizar los 48 mil millones de productos y sumas que requiere calcular  $C$ . Vamos a necesitar varios bloques.

Iniciamos nuestro código importando los módulos necesarios para acceder e inicializar el *device*:

```
import pycuda.driver as driver
import pycuda.autoinit
```

CELL 01

Ahora definimos el kernel. Como argumentos, recibirá los arrays que representan las matrices involucradas, y también las dimensiones de las mismas:

CELL 02

```
kernel = """
__global__ void matrix_prod(float *a, float *b, float *c, int n, int l, int m)
{
    // Identificación del hilo 2D en un bloque
    int i = blockIdx.x * blockDim.x + threadIdx.y; // Fila i
    int j = blockIdx.y * blockDim.y + threadIdx.x; // Columna j

    // Verificamos estar dentro de los límites (n x m)
    if(i < n && j < m) {
        float valor = 0;
        for (int k = 0; k < l; k++) {
            valor += a[i * l + k] * b[k * m + j];
        }
        // Almacenamos el resultado
        c[i * m + j] = valor;
    }
}
"""
```

Asociamos la fila  $i$  con la variable `threadIdx.y`, y la columna  $j$  con la variable `threadIdx.x`, dentro del bloque correspondiente. Luego, realizamos el cálculo del elemento  $c_{ij}$  verificando previamente que los índices se encuentran dentro de los límites de filas y columnas de las matrices. Esto es necesario debido a que se crearán más hilos que los que ocuparemos para el cálculo de  $c_{ij}$ , dado que la cantidad de bloques en cada dimensión debe exceder a las filas y columnas que tienen  $A$  y  $B$ , como veremos en la celda siguiente.

Primero incluimos el módulo `time`, que nos permitirá cuantificar el tiempo de ejecución de algunas líneas de código, para comparar la eficiencia del *host* y del *device*.

Luego ya con Numpy definimos las dimensiones de las matrices  $A$  y  $B$ , y luego generamos estas matrices con números aleatorios. Como estas matrices son creadas en el *host*, nombramos los arrays con `a_cpu` y `b_cpu` respectivamente. Es necesario mencionar aquí que dichos arrays bidimensionales son representados en memoria como arrays unidimensionales por fila, y hacemos uso de esto en el kernel al recorrer los arrays con los índices  $i$  y  $j$ . Con el propósito de validar los resultados que obtendremos con la GPU, realizamos el cálculo en la CPU utilizando el método `dot` de NumPy. Registramos el tiempo que tarda en ejecutarse esta última operación.

CELL 03

```

import numpy as np
import time

# Definimos el tamaño de las matrices A(n x l) y B (l x m)
m = np.int32(4000)
l = np.int32(6000)
n = np.int32(2000)

# creamos dos matrices aleatorias
np.random.seed(13)
a_cpu = np.random.randn(m, l).astype(np.float32)
b_cpu = np.random.randn(l, n).astype(np.float32)

# calculamos el resultado en la CPU para comparar con el de la GPU

start_time_CPU = time.time()
c_cpu = np.dot(a_cpu, b_cpu)
end_time_CPU = time.time()
elapsed_time_CPU = end_time_CPU - start_time_CPU

print(f"El tiempo de ejecución fue: {elapsed_time_CPU} segundos")

```

El tiempo de ejecución fue: 0.24214625358581543 segundos

Vemos que el cálculo del producto matricial llevó un poco más de dos décimas de segundo.

Organizaremos nuestros hilos en bloques cuadrados, conteniendo un número de hilos por dimensión que resulta de la raíz cuadrada del máximo número de hilos por bloque que nos informa la GPU. La cantidad de bloques que definen nuestra grilla resulta, en cada dimensión, del entero superior más próximo al cociente entre la cantidad de elementos de matriz y el número de hilos. Al realizar esa cuenta, vemos que necesitamos 125 bloques en la dirección  $x$  y 63 en la  $y$ . Recordemos que el número total de elementos de  $C$  es de 8 millones. Si dividimos este número por la cantidad de hilos por bloque (1.024) nos indica que necesitamos 7812,5 bloques, pero dado que usamos bloques cuadrados sobre matrices que no lo son, el cálculo determina que necesitaremos un total de 7875 bloques para cubrir las filas y columnas de  $C$ , generando de este modo más hilos que los necesarios, y por este motivo debemos verificar que los índices  $i$  y  $j$  en el cálculo que realizamos en el kernel no superen los límites de filas y columnas de  $A$  y  $B$ , respectivamente.

CELL 04

```

from math import sqrt, ceil

device = driver.Device(0)
max_threads_per_block = device.get_attribute(driver.device_attribute.MAX_THREADS_PER_BLOCK)
t_xy = int(ceil(sqrt(max_threads_per_block)))
NBlocks_x = ceil(m / t_xy)
NBlocks_y = ceil(n / t_xy)
print(f"Bloques en x: {NBlocks_x}, bloques en y: {NBlocks_y}, con t_xy: {t_xy}")

```

Bloques en x: 125, bloques en y: 63, con t\_xy: 32

Lo que nos queda por hacer es compilar el kernel, obtener el *handler* a la función, transferir los arrays a la memoria del *device* (mediante el método `gpuarray.to_gpu()`), crear en el *device* el

espacio necesario para almacenar los valores de la matriz  $C$  (`c_gpu`), y ejecutar el kernel con los datos de las matrices y la organización de hilos en bloques, tal como mostramos en las celdas siguientes.

CELL 05

```
from pycuda import compiler

# Compilamos el kernel y obtenemos el handler a la función
mod = compiler.SourceModule(kernel)
func = mod.get_function("matrix_prod")
```

CELL 06

```
from pycuda import gpuarray, tools

# transferimos de la memoria host (CPU) a la memoria del device (GPU)
start_time_CPU2GPU = time.time()
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
end_time_CPU2GPU = time.time()
elapsed_time_CPU2GPU = end_time_CPU2GPU - start_time_CPU2GPU
# creamos un array vacío en la GPU para el resultado (C = A . B)
c_gpu = gpuarray.empty((m, n), np.float32)

print(f"El tiempo de transferencia de datos fue: {elapsed_time_CPU2GPU} segundos")

# Ejecutamos el kernel en la GPU
start_time_GPU = time.time()
func(a_gpu, b_gpu, c_gpu, m, l, n, block=(t_xy, t_xy, 1), grid=(NBlocks_x, NBlocks_y, 1))
end_time_GPU = time.time()
elapsed_time_GPU = end_time_GPU - start_time_GPU

print(f"El tiempo de ejecución fue: {elapsed_time_GPU} segundos")
```

El tiempo de transferencia de datos fue: 0.03139352798461914 segundos  
El tiempo de ejecución fue: 0.0003440380096435547 segundos

En estas celdas registramos el tiempo que lleva realizar dos operaciones: la transferencia de los arrays al *device* y la ejecución del kernel en la GPU. Podemos ver que la transferencia es dos órdenes de magnitud más lenta que la ejecución del kernel (al menos en el *hardware* particular en el que fue ejecutado el código), y que el cálculo del producto matricial, realizado en la GPU, es unas 700 veces más rápido que el cálculo en la CPU con NumPy. Observando lo «costoso» que resulta la transferencia de datos hacia y desde el *host*, es de suma importancia un buen diseño del programa de modo de minimizar estas transferencias.

Finalmente, comparamos el resultado obtenido en el *host* con el obtenido con la GPU, realizando previamente la transferencia *device* a *host* con el método `get()`:

CELL 07

```
np.allclose(c_cpu, c_gpu.get(), rtol=1e-05, atol=1e-03)
```

True

Podemos observar que los cálculos en CPU y GPU son aproximados con una tolerancia absoluta de  $1 \times 10^{-3}$ . Esto se debe a que utilizamos una representación de punto flotante con precisión simple (32 bits), y que en las sumas acumuladas en bucles se suman también los errores de redondeo.

Este ejemplo sencillo muestra la forma en que podemos utilizar la GPU para realizar cálculos numéricos. Sin embargo, es un código que no explota toda la capacidad del *hardware* para hacer más eficiente la tarea. Para ello es necesario considerar aspectos tales como el ancho de banda y latencia de las memorias global, compartida y registros de la GPU, que condicionan el particionado del problema y la organización de los hilos, y otros detalles que exceden el alcance de este libro. Sin embargo, queremos enfatizar la necesidad de evaluar la conveniencia de utilizar la GPU para nuestros cálculos en función del *hardware* disponible y el «tamaño» del problema a resolver. En este ejemplo concreto, vemos que resulta muy conveniente realizar el producto en la GPU, pero el tiempo que requiere la transferencia de datos casi pone en igualdad de condiciones al cálculo en CPU, lo que indica que una buena planificación de las tareas requiere minimizar la transferencia de los datos y aprovechar la potencia de cálculo paralelo masivo provista por la GPU.

## 1.5. Producto de matrices con PyOpenCL

OpenCL (*Open Computing Language*) es un estándar abierto que permite la programación sobre casi cualquier dispositivo disponible en nuestros días: CPUs, GPUs, FPGAs (*Field Programmable Gate Arrays*), aprovechando las ventajas que cada plataforma pueda ofrecer. A diferencia de CUDA, que solo corre programas en los dispositivos de NVIDIA, OpenCL puede correr en cualquier arquitectura y CPU de múltiples *cores* siempre que el fabricante provea un *driver* OpenCL.

Existen muchas similitudes entre OpenCL y CUDA, debido a las semejanzas del abordaje en paralelo y las jerarquías de memoria. OpenCL asume que la ejecución de un programa ocurre en una plataforma conformada por un *host* (típicamente un CPU) y un número de *devices* conectados. Cada *device* está compuesto por un número de unidades de cómputo (CU, *compute units*) que a su vez contienen unidades de procesamiento (PEs, *processing elements*). En una analogía con el modelo CUDA, los PEs corresponden a los CUDA *cores*, y los CUs a los *streaming processors*.

Un programa OpenCL contiene típicamente dos componentes:

- Un programa que controla la ejecución del código en el *device* y provee funcionalidades de entrada y salida de datos.
- Uno o más programas que corren en el *device* o *kernels*.

Cuando un *host* lanza un kernel, éste se ejecuta en el *device* en forma de «elemento de trabajo» (equivalente a un *thread* de CUDA). Cada elemento de trabajo corre en un PE, y puede haber múltiples elementos de trabajo que se agrupan en «grupos de trabajo» (equivalentes a los bloques de CUDA). Cada grupo de trabajo es asignado a un CU.

PyOpenCL ofrece un abordaje «pitónico» a OpenCL, del mismo modo que PyCUDA lo hace con CUDA. Mostraremos a continuación el ejemplo del producto de matrices pero ahora con OpenCL. Para comenzar, importamos el módulo `pyopencl` y establecemos el contacto con la GPU. Primero detectamos la plataforma de ejecución (`platform`), y su correspondiente *device* (que asignamos a la variable `device`). Generamos el contexto de ejecución (`context`) y finalmente una cola

de comandos (queue) que es por donde se envían los comandos al *device* (ejecución del kernel, sincronización y operaciones de transferencia de memoria).

CELL 01

```
import pyopencl as cl

# Configuramos el contexto y cola de comandos
platform = cl.get_platforms()[0]
device = platform.get_devices()[0]
context = cl.Context([device])
queue = cl.CommandQueue(context)
```

En la celda 2 interrogamos al *device* para poder dimensionar el tamaño del grupo de trabajo:

CELL 02

```
# Obtenemos el tamaño máximo de grupo de trabajo y el tamaño máximo de los
# elementos de grupo de trabajo en cada dimensión
max_work_group_size = device.get_info(cl.device_info.MAX_WORK_GROUP_SIZE)
max_work_item_sizes = device.get_info(cl.device_info.MAX_WORK_ITEM_SIZES)
print(
    f"Máximo tamaño de grupo: {max_work_group_size}, "
    f"Máximo tamaño de elementos por dimensión: {max_work_item_sizes}")

# Definimos el tamaño de los bloques
block_size = int(np.sqrt(max_work_group_size))
block_size_x = min(block_size, max_work_item_sizes[0])
block_size_y = min(block_size, max_work_item_sizes[1])
print(f"Elementos por bloque: {block_size_x} x {block_size_y}")
```

---

Máximo tamaño de grupo: 1024, máximo tamaño de elementos por dimensión: [1024, 1024, 64]  
Elementos por bloque: 32 x 32

Vemos que cada bloque puede administrar 1.024 elementos de trabajo que se pueden organizar en un arreglo tridimensional con un máximo de elementos por cada dimensión de 1.024, 1.024 y 64, respectivamente. Decidimos entonces trabajar con grupos bidimensionales cuadrados.

Tal como hicimos el en ejemplo con PyCUDA, dimensionamos y definimos las matrices, y las transferimos al *device*:



CELL 03

```
import numpy as np

# Definimos el tamaño de las matrices A(n x l) y B (l x m)
m = 4000
l = 6000
n = 2000
np.random.seed(13)
a_cpu = np.random.rand(m, l).astype(np.float32)
b_cpu = np.random.rand(l, n).astype(np.float32)
c_cpu = np.empty((m, n), dtype=np.float32)

# Creamos buffers en el dispositivo
mf = cl.mem_flags
a_buf = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_cpu)
b_buf = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_cpu)
c_buf = cl.Buffer(context, mf.WRITE_ONLY, c_cpu.nbytes)
```

En la celda 4 definimos el kernel. A diferencia del ejemplo con pyCUDA, el índice de fila y columna se obtienen a partir de las variables intrínsecas `get_global_id(0)` y `get_global_id(1)`, que ya contienen los identificadores globales de cada hilo, sin tener que hacer el cálculo a partir de su ubicación interna en cada grupo de trabajo. Finalmente, asignamos a la variable `program` al programa compilado listo para su ejecución.

CELL 04

```
# Definimos el kernel OpenCL
program_source = """
__kernel void matmul(const int M, const int N, const int K,
                    __global const float *A, __global const float *B,
                    __global float *C) {

    int row = get_global_id(0);
    int col = get_global_id(1);

    float sum = 0.0f;
    for (int k = 0; k < K; k++) {
        sum += A[row * K + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}
"""

program = cl.Program(context, program_source).build()
```

En la celda 5 determinamos el tamaño en cada dimensión de la grilla que agrupa los grupos de trabajo, a partir de la división entera entre la cantidad de hilos necesarios en cada dimensión (que representan las filas y columnas de las matrices) y de la cantidad de hilos que asignamos a cada grupo de trabajo (que recordemos, usaremos un arreglo cuadrado de hilos). Estas dimensiones de la grilla las asignamos a la variable `global_size`, mientras que en `local_size` definimos el tamaño de cada grupo de trabajo. Una vez organizada la descomposición del problema en la agrupación de tareas, lanzamos el kernel con los argumentos correspondientes (dimensiones de las matrices y los arrays que las contienen), y finalmente transferimos el resultado desde la memoria del *device* (`c_buf`) a la memoria del *host* (`c_cpu`).

CELL 05

```
# Ajustamos el tamaño global (grilla)
global_size_x = (m + block_size_x - 1) // block_size_x * block_size_x
global_size_y = (n + block_size_y - 1) // block_size_y * block_size_y
global_size = (global_size_x, global_size_y)
local_size = (block_size_x, block_size_y)

# Ejecutamos el kernel
matmul_kernel = program.matmul
matmul_kernel.set_args(np.int32(m), np.int32(n), np.int32(l), a_buf, b_buf, c_buf)
cl.enqueue_nd_range_kernel(queue, matmul_kernel, global_size, local_size)

# Transferimos el resultado al host
cl.enqueue_copy(queue, c_cpu, c_buf)

# Esperamos a que terminen las operaciones
queue.finish()
```

Para finalizar, realizamos el producto de matrices usando el método `dot()` de NumPy, y comparamos los resultados obtenidos. Podemos ver que con una buena tolerancia absoluta, ambos resultados coinciden.

CELL 06

```
# Calculamos el resultado en el host para comparar con el del device
c_npy = np.dot(a_cpu, b_cpu)

# Comparamos los resultados obtenidos en la CPU y la GPU
np.allclose(c_npy, c_cpu, rtol=1e-01, atol=1e-8)

True
```

Al igual que el caso de PyCUDA, realizamos una paralelización muy simple del producto de matrices, sin aprovechar detalles propios del *hardware* que permiten una optimización de los recursos disponibles, aumentando de este modo en forma significativa la eficiencia del cómputo.

## 1.6. Lecturas recomendadas

Una buena introducción a la programación con CUDA: Wen-Mei W. Hwu, David B. Kirk e Iz-zat El Hajj. *Programming massively parallel processors*. 4.<sup>a</sup> ed. Cambridge, United States: Morgan Kaufmann, sep. de 2023.

La referencia para la programación en CUDA, que permite comprender el modelo de programación es NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation. Jun. de 2024. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

Un libro que permite la entrada a la programación con CUDA y PyCUDA, con aplicación a la visión por computadora: Bhaumik Vaidya. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA*. Birmingham, England: Packt Publishing, nov. de 2018.

**Parte IV**  
**Apéndices**

## A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [10].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

## Bibliografía

- [1] URL: <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.
- [2] C.J. Thompson, Sahngyun Hahn y M. Oskin. «Using modern graphics architectures for general-purpose computing: a framework and analysis». En: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings*. 2002, págs. 306-317. DOI: [10.1109/MICRO.2002.1176259](https://doi.org/10.1109/MICRO.2002.1176259).
- [3] Jeff Bolz, Ian Farmer, Eitan Grinspun y Peter Schröder. «Sparse matrix solvers on the GPU: conjugate gradients and multigrid». En: *ACM Trans. Graph.* 22.3 (jul. de 2003), págs. 917-924. DOI: [10.1145/882262.882364](https://doi.org/10.1145/882262.882364). URL: <https://doi.org/10.1145/882262.882364>.
- [4] Jens Krüger y Rüdiger Westermann. «Linear algebra operators for GPU implementation of numerical algorithms». En: *ACM Trans. Graph.* 22.3 (jul. de 2003), págs. 908-916. DOI: [10.1145/882262.882363](https://doi.org/10.1145/882262.882363). URL: <https://doi.org/10.1145/882262.882363>.
- [5] NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation. Jun. de 2024. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [6] Rajat Raina, Anand Madhavan y Andrew Y Ng. «Large-scale deep unsupervised learning using graphics processors». En: *Proceedings of the 26th annual international conference on machine learning*. 2009, págs. 873-880.
- [7] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov y Ahmed Fasih. «PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation». En: *Parallel Computing* 38.3 (2012), págs. 157-174. DOI: [10.1016/j.parco.2011.09.001](https://doi.org/10.1016/j.parco.2011.09.001).
- [8] Wen-Mei W. Hwu, David B. Kirk e Izzat El Hajj. *Programming massively parallel processors*. 4.<sup>a</sup> ed. Cambridge, United States: Morgan Kaufmann, sep. de 2023.
- [9] Bhaumik Vaidya. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA*. Birmingham, England: Packt Publishing, nov. de 2018.
- [10] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.