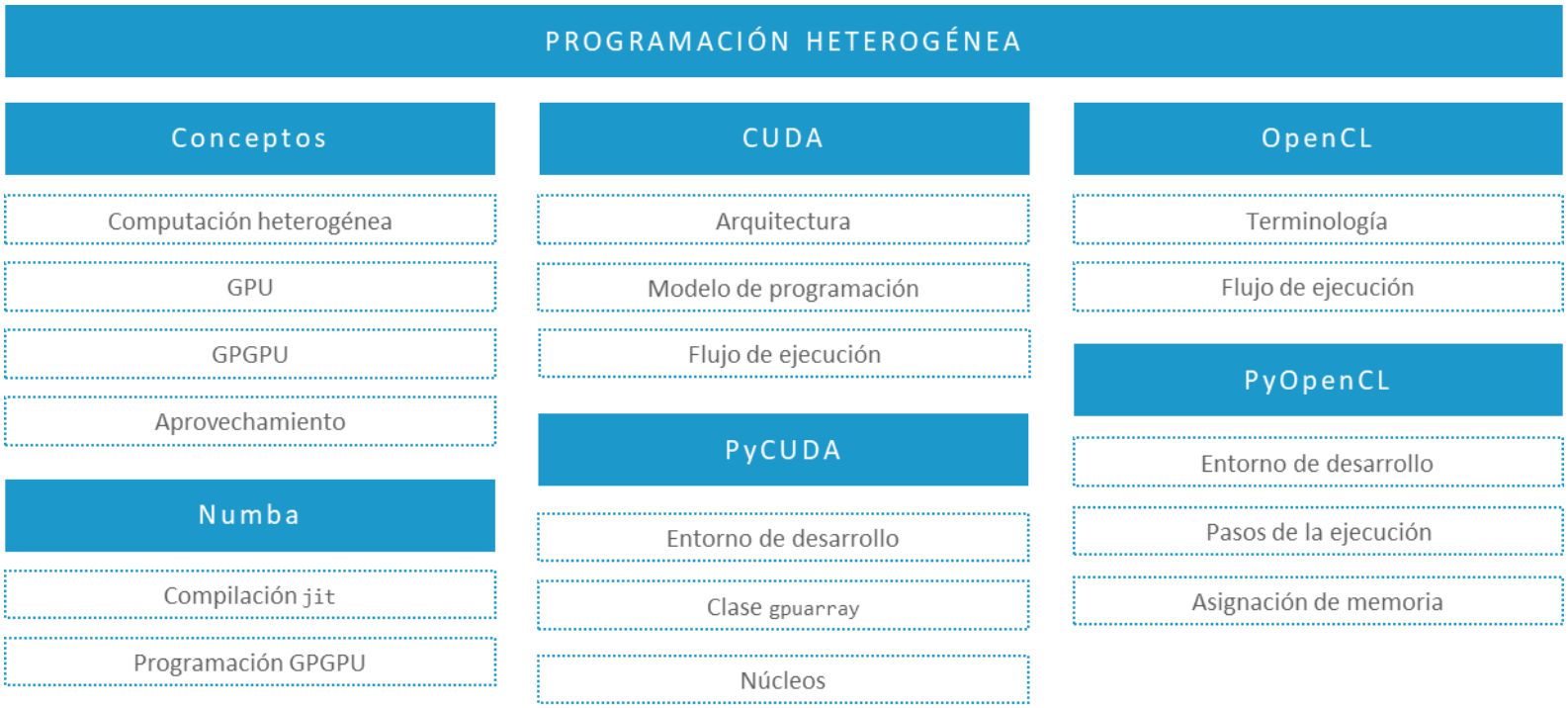
Programación Científica y HPC

Programación heterogénea

Índice

Esquema



Ideas clave

9.1. Introducción y objetivos

La computación heterogénea es el **uso de *hardware* especializado para conseguir una mayor aceleración de los programas,** mientras se tratan aspectos específicos de algunos de sus bloques de código. Estos dispositivos se denominan comúnmente **aceleradores** y, entre ellos, cabe destacar aceleradores criptográficos procesadores de señales digitales o DSP (*Digital Signal Processor*) y las unidades de procesamiento gráfico o GPU (*Graphics Processing Unit*). Estas últimas, actualmente, han adquirido una gran relevancia.

La **principal motivación,** tanto para el desarrollo de las GPU, como para su uso como dispositivos de cálculo general, es **la velocidad.** Hay que tener en cuenta que en el procesamiento de gráficos se ejecutan operaciones de forma constante, lo que implica un procesamiento intensivo. Por ello, poder usar un dispositivo especializado donde estas operaciones pudieran realizarse en *hardware* reduciría, en gran medida, el tiempo que se necesita para ejecutarlas. Actualmente, este planteamiento se extiende a otros tipos de operaciones de cálculo no relacionados con el procesamiento gráfico, pero con similares exigencias de computación, lo que ha dado lugar a la aparición de las **unidades de procesamiento gráfico de propósito general,** que se denomina de forma abreviada GPGPU (*General-Purpose Computing on Graphics Processing Units*).

Por tanto, la computación GPU de propósito general o la computación GPGPU consiste en el uso de una GPU para la realización de cálculos científicos y de ingeniería de propósito general.

Adicionalmente, en este tema se tratarán también **las herramientas de programación heterogénea,** que **permiten adaptar el código para su ejecución**

sobre GPU y las interfaces que proporcionan algunos lenguajes para el uso de lenguajes de GPGPU, que son de bajo nivel.

Entre las bibliotecas más utilizadas se cuenta con OpenCL (*Open Compute Language*), lenguaje basado en C agnóstico al *hardware*, por lo que es compatible con muchas plataformas heterogéneas y CUDA (*Compute Unified Device Architecture*), que proporciona una plataforma de computación paralela que permite el uso de un lenguaje basado en C, específicamente pensado para la programación de las GPU de NVIDIA.

El éxito de las GPGPU en los últimos años se debe a la **facilidad de adaptarse a los modelos de programación paralela** asociados con OpenCL y CUDA. En estos modelos de programación las aplicaciones se diseñan para implementar los módulos o núcleos de cálculo intensivo y asignarlos a la GPU. El resto de la aplicación se ejecuta en la CPU.

En este tema se verá este modo de programación, además de las características que proporciona Python para la adaptación a estos modelos, como es el compilar Numba y las herramientas PyCUDA y PyOPENCL.

9.2. La GPU como dispositivo de cómputo general

Actualmente, las unidades de procesamiento gráfico se utilizan mucho. **Estos dispositivos se diseñaron,** originalmente, **para la aceleración del procesamiento necesario para la visualización de gráficos.** En la siguiente imagen se muestra, de forma resumida, la diferencia entre una CPU y una GPU.

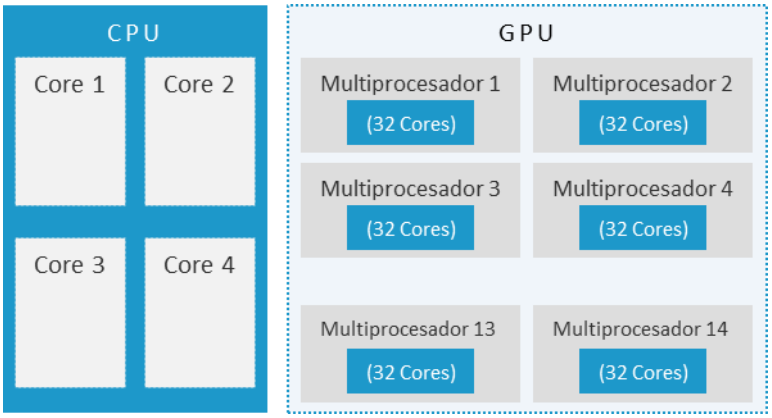


Figura 1. Arquitecturas CPU y GPU. Fuente: adaptado de https://internetpasoapaso.com/gpu-unidad-procesamiento-grafico/

Los datos gráficos que son «rasterizados», es decir, son **convertidos en un conjunto de pixeles que permite que sean enviados y visualizados a un medio de salida digital,** por su naturaleza, **cuentan con las características necesarias para poder ser procesados en paralelo,** consiguientemente, se obtiene una mayor velocidad de procesamiento. Por tanto, nuevamente, el aspecto subyacente bajo la mejora del rendimiento es la posibilidad de procesamiento paralelo. Pero este planteamiento busca una forma de computación no basada en *software*, sino sobre *hardware*. Se trata, por tanto, de **encontrar el hardware necesario que soporte estos cálculos específicos** para que puedan ser ejecutados con el mayor rendimiento.

Debido a esto, las GPU han evolucionado hasta convertirse en procesadores altamente paralelos con operaciones específicas sobre gráficos que se realizan en el propio *hardware*, en lugar de en *software*. El resultado final del uso de estos procesadores es que **el rendimiento del procesamiento de gráficos mejora** considerablemente y, al mismo tiempo, **disminuye la carga computacional de la** unidad central de procesamiento o **CPU** (*Central Processing Unit*), dejándola libre para la ejecución de otras tareas.

En la actualidad, el rendimiento y la capacidad de realizar operaciones complejas de las GPU ha aumentado tanto que no solo realiza tareas relacionadas con el procesamiento gráfico, sino que **soporta otros tipos de cálculo de computación**

intensiva. Eso ha dado lugar a la aparición de las unidades de procesamiento gráfico de propósito general, que se denomina de forma abreviada GPGPU.

Por tanto, la computación GPU de propósito general o la computación GPGPU consiste en el uso de una GPU para la realización de cálculos científicos y de ingeniería de propósito general.

El modelo de computación GPU consiste en usar una CPU y una GPU juntas en un modelo de procesamiento que se conoce como **procesamiento heterogéneo.** De esta forma, la parte secuencial de la aplicación se ejecuta sobre la CPU y la parte de cálculo intensivo es acelerada por la GPU. De esta forma, el rendimiento se incrementa considerablemente gracias a las altas prestaciones con las que cuenta la GPU.

Este es un mecanismo prácticamente transparente para el usuario, que simplemente percibe que la aplicación se ejecuta más rápido.

Aprovechamiento del uso de GPU

Este aprovechamiento consiste en:

- Procesamiento de conjuntos de datos grandes.
- Un paralelismo de grano fino basado en SIMD (*Single Instruction, Multiple Data*), mediante el que todos los núcleos ejecutan el mismo código.
- Baja latencia en las operaciones de coma flotante.

La GPU ha evolucionado a lo largo de los años hasta tener *teraflops* (TFLOPS), es decir, 10^{12} *flops* (*Floating Point Operations Per Second*) de rendimiento. Es importante indicar lo que significa esta medida de rendimiento. Un *flop* es el número de operaciones en coma flotante por segundo, que **sirve para medir el número de operaciones matemáticas que se lleva a cabo en una unidad de procesamiento,** como es en este caso una GPU.

Como se sabe, se denotan como operaciones en coma flotante porque es la manera en la que se denominan a los números reales, que se utilizan en ellas, que pueden ser muy grandes o pequeños, pero que cuentan un número de decimales grande. De hecho, existe un estándar IEEE-754 para la representación de estos números con simple o doble precisión.

Para profundizar sobre la representación de los números reales de coma flotante consulte la página de Portal Electrozona y el trabajo de la Universidad Nacional de Quilmes sobre el estándar IEEE-754.

No obstante, se debe tener en cuenta que **la capacidad de cómputo de una GPU depende de los recursos *hardware* que tenga a disposición,** que son, entre otros, el número de núcleos para computación, el número de hilos que se pueden gestionar de forma simultánea y la memoria local o compartida del dispositivo.

9.3. CUDA y PyCUDA

CUDA es una **arquitectura de *hardware* propietaria y un entorno de desarrollo de NVIDIA.** El interés por esta tecnología en el entorno de la programación paralela se ha avalado por la comunidad de desarrolladores activos. Este modelo se basa en la **ejecución paralela mediante hilos de una misma función sobre distintos datos.** Lo cierto es que las GPU están mejor dotadas para el procesamiento paralelo, porque mientras la CPU pueden manejar unos cuantos hilos, no es comparable con los cientos de núcleos con los que cuenta la GPU, que permite la ejecución simultánea de miles de hilos.

Arquitectura CUDA

Una GPU contiene un *array* de multiprocesadores, cada uno de los cuales contiene procesadores de flujo. Dentro de un multiprocesador, los procesadores de flujo tienen acceso a memoria local compartida, que es pequeña y de baja latencia, lo que la hace ser muy rápida. Por otro lado, la GPU se comunica con la CPU mediante memoria de acceso directo.

La arquitectura CUDA se muestra en la siguiente imagen:

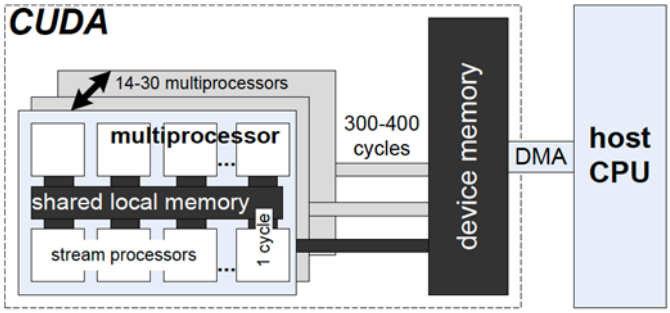


Figura 2. Arquitectura NVIDIA CUDA. Fuente: Chatterjee *et al*., 2011.

A continuación, se presenta el modo de programación CUDA que, básicamente, puede ser descrito mediante sus elementos, que son hilos, bloques de hilos, la forma de organización de los hilos y los núcleos o funciones que se ejecutan.

Por tanto, un proceso en CUDA se describe usando la terminología con la que se denominan los elementos que forman parte de este proceso:

- **`Kernel`:** código que ejecutan los hilos, que son las unidades atómicas de cálculo.
- ***Grid:*** organización de los bloques de hilos a modo de malla. Los bloques de la misma dimensión y tamaño, que ejecutan el mismo núcleo, pueden ser aguapados en una rejilla (*grid*) de bloques.
- ***Block:*** un bloque de hilos que es ejecutado sobre un mismo procesador, de forma que los hilos de un mismo bloque se pueden sincronizar y usar una memoria local compartida.

Modelo de programación

El modelo de programación CUDA mencionado se muestra en la Figura 3:

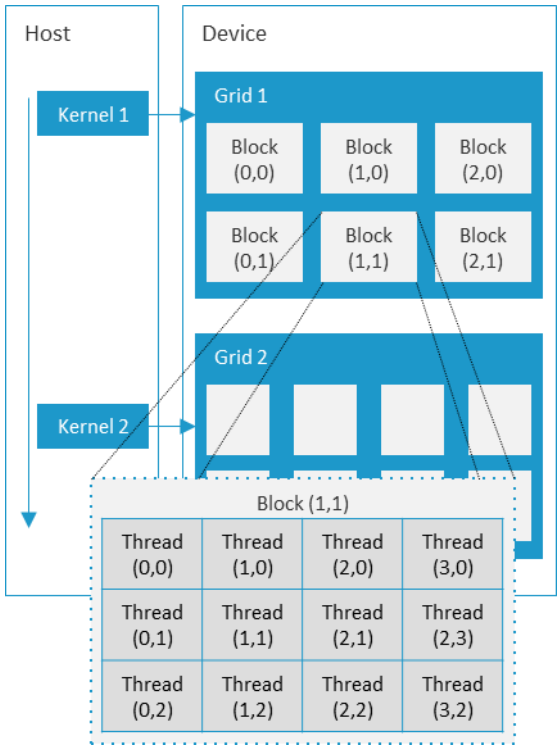


Figura 3. Modelo de programación CUDA. Fuente: adaptado de Gulati *et al.*, 2009.

Flujo de ejecución

Para entender el flujo de ejecución que se lleva a cabo para el modelo de programación CUDA se debe tener en cuenta que existirá un proceso padre, que será el que organice y lance la ejecución de las múltiples copias del *kernel* que se quiere usar. De forma que el flujo se puede describir mediante los siguientes pasos:

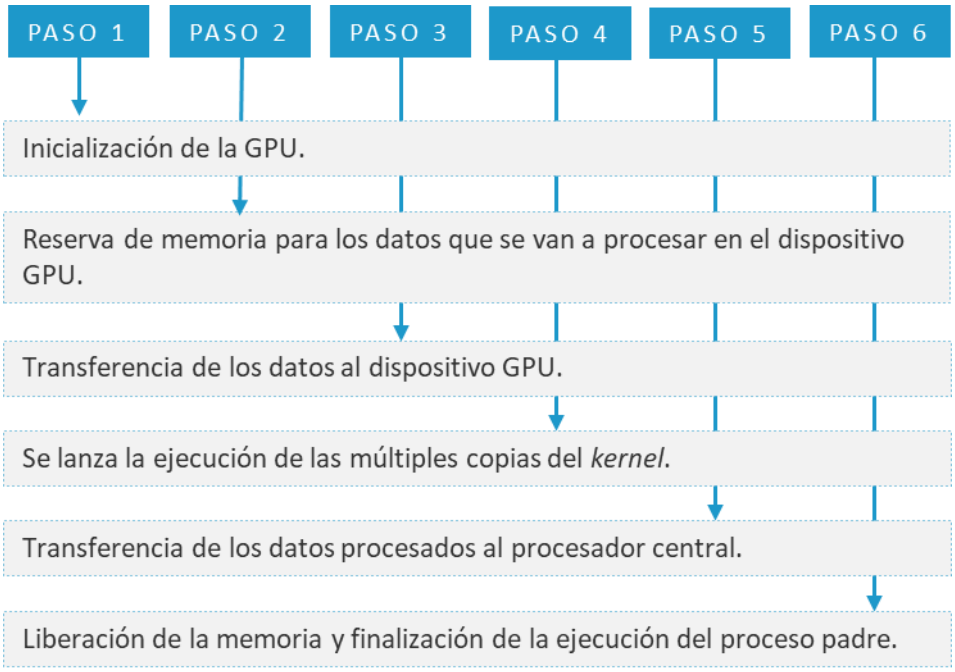


Figura 4. Pasos del flujo de ejecución en CUDA. Fuente: elaboración propia.

Se debe tener en cuenta que se puede implementar un ciclo en el que **los pasos de procesamiento en la GPU se pueden repetir varias veces.** Las instrucciones de código implicadas son:

```
init()
cudaMalloc()
cudaMencpy()
```

Figura 5. Instrucciones de inicialización, reserva de memoria y transferencia de datos a la GPU. Fuente: elaboración propia.

A partir de aquí es necesario tener en cuenta los detalles de implementación de los *kernel* que ejecutará cada hilo, la manera en la que se pueden sincronizar y los datos que pueden compartir.

El conjunto de funciones y estructuras que se pueden usar con CUDA para implementar los *kernels* y gestionar las distintas cuestiones es amplio y se actualiza periódicamente.

Para conocer en profundidad los distintos módulos y estructuras de CUDA pueden consultar la documentación en la página de NVIDIA y el manual de referencias de su documentación y su manual de referencia.

CUDA cuenta con extensiones para varios lenguajes de programación, entre ellos, Python. Las extensiones de CUDA para Python más conocidas son PyCUDA y Numba.

PyCUDA

PyCUDA es una **biblioteca que proporciona acceso a la API de CUDA desde Python,** desarrollada por Andreas Klöckner. Es un proyecto de código abierto bajo licencia MIT muy ligero.

Sus principales características incluyen la recogida de basura automática, ligada al tiempo de vida de un objeto, un mecanismo de abstracción como los módulos y buffers, el acceso completo al controlador y el manejo de errores incorporado.

El objetivo principal de PyCUDA es **permitir el uso de CUDA mediante una abstracción mínima de Python.** También soporta, entre otros, la metaprogramación, es decir, el desarrollo de programas que manipulan otros programas.

Para empezar a trabajar con CUDA, más concretamente con PyCUDA, primero es necesario preparar el entorno de desarrollo compatible con la arquitectura CUDA.

En A fondo se proporciona el enlace para a la documentación oficial de PyCUDA, con los requisitos necesarios para el uso de una GPU de NVIDIA, la instalación del kit de desarrollo de CUDA y otras cuestiones.

A continuación, se mostrarán los pasos que se deben seguir para lanzar una ejecución sobre GPU. Debido a los requisitos de *hardware*, todos los ejemplos se desarrollarán con Google Colab, que pone a disposición una GPU de NVIDIA para poder trabajar con ella. Un ejemplo de cómo prepara el entorno de ejecución se muestra a continuación.

Ejemplo 1. Configuración del entorno de ejecución en Colab

Para comenzar en Colab, para un notebook en el que se quiera implementar el uso de CONDA, primero es necesario cambiar el entorno de ejecución para poder acceder a la GPU que se pone a disposición.

Para el uso de las GPU, se necesita cambiar el entorno de ejecución en Colab, para que las GPU estén visibles y sean utilizables. Este cambio de configuración se realiza en el menú Entorno de Ejecución>Cambiar tipo de entorno de ejecución y elegir GPU.

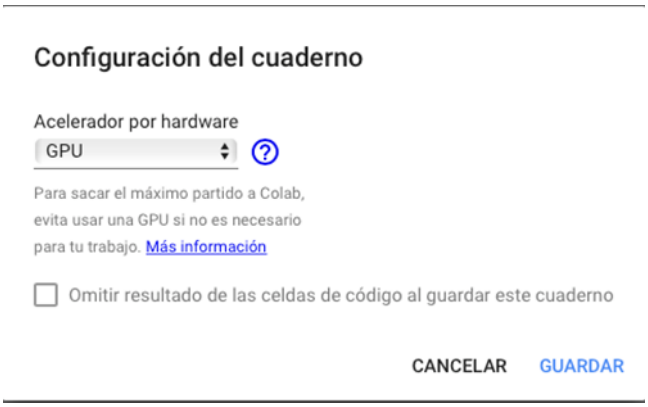


Figura 6. Captura de pantalla para configuración del entorno de ejecución. Fuente: elaboración propia.

Entorno de desarrollo para CUDA con PyCUDA

Se procede a inicializar el entorno de desarrollo, para poder acceder a la API de computación paralela CUDA, por lo cual **es necesario tener instalado el módulo pycuda.**

Nótese que esta instalación, que se muestra en la primera línea del código de la Figura 7, es necesaria cada vez que se crea un notebook nuevo en Colab, pero no cuando se trabaja en otros entornos de ejecución propios o preparados, a tal efecto donde una vez que se ha instalado `pycuda` ya se tiene disponible.

La configuración del entorno de desarrollo se llevará a cabo mediante la importación de los módulos necesarios para el uso del controlador, el compilador y proceder a su inicialización. Las siguientes son las instrucciones necesarias:

```
!pip install pycuda # install cuda
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
```

Figura 7. Código de instalación de `pycuda` y configuración del entorno de desarrollo. Fuente: elaboración propia.

Se verá a continuación algunas cuestiones sobre como paralelizar algunos programas, aunque se debe tener en cuenta que hay muchas formas de aplicación.

Clase `gpuarray`

Después de la inicialización del entorno se deben preparar los datos para su transferencia a la GPU en el momento de la ejecución del núcleo. Para esto hay que tener en cuenta tres cuestiones importantes:

- **La precisión de los datos.** Como ya se ha mencionado, las GPU realizan computación en coma flotante, pero no todas soportan doble precisión, por lo que es necesario asegurar que los datos están en simple precisión.

- **La asignación de memoria para los datos en el dispositivo.** Para ello se necesita realizar una asignación explícita. Hay distintas formas de poder hacer esto.

- **La transferencia de los datos al dispositivo.** Se pasan como parámetro cuando se invoca el núcleo.

Al igual que los *arrays* de Numpy, y con características similares, PyCUDA cuenta con la clase `gpuarray`, que proporciona el soporte necesario para la transferencia de datos a la GPU simplificando enormemente el proceso que se necesita realizar en CUDA.

Con el uso del método `to_gpu()` aplicado a un *array* de Numpy se consigue:

- La conversión de un array de Numpy en un array de PyCUDA.
- Cubrir las cuestiones de asignación y liberación de la memoria en la GPU.
- Transferir los datos del *host* a la GPU.

Núcleos (*kernels*) con PyCUDA

A continuación, se tratará la forma de definir un *kernel*, es decir, una función paralela que se puede lanzar desde el proceso padre (el hilo que se ejecuta en la CPU) a la GPU.

Definición del núcleo. Mediante la función `SourceModule`, del módulo `pycuda.compiler`, se compilará el código de una función implementada en C, obteniéndose código compilado en CUDA y que necesita ser referenciado mediante el uso del método `get_function()` de PyCUDA.

- La **función del kernel escrita en C** debe ir precedida de `__global__` y debe ser una función `void`, porque la salida a través de un parámetro puntero en general es de tipo `float`.

- **Se identifican los hilos** mediante `threadIdx.x`, que se asigna a una variable `i` que, en el caso de las matrices, servirá para referirse a cada uno de los elementos que serán procesados por cada hilo.

- **Invocación del núcleo** usando el nombre de la función y pasando como parámetros los datos preparados, los bloques y, opcionalmente, la rejilla o *grid*.

En el vídeo **Programación con PyCUDA en Colab** podrán ver CPU contra. GPU. La GPU como dispositivo de cómputo. Aprovechamiento y uso de la GPU. PyCUDA, ejemplo de uso y flujo de ejecución.

9.4. OpenCL y PyOpenCL

OpenCL, al contrario que CUDA, **es un estándar abierto que se puede usar sobre distintos** GPU y otros microprocesadores desarrollados en sus nuevas versiones con C++. Presenta una **solución más completa y versátil,** pero no cuenta con la sencillez de CUDA. OpenCL es desarrollado por el grupo Khronos, aunque fue impulsado inicialmente por Apple, a la que se unieron otras grandes empresas.

En las pruebas de rendimiento y en las comparativas con CUDA se han obtenido buenos resultados, aunque sigue existiendo una ligera ventaja en CUDA, debido a que este es específicamente diseñado para un tipo de GPU. **La necesidad de contar con una interfaz común para distintos dispositivos le resta un poco de eficiencia,** pero los avances en los últimos años son prometedores. Todavía le falta un poco de madurez y contar con más fuentes de información, bibliotecas y herramientas de desarrollo.

Sin embargo, incidiendo en una de sus ventajas, **por ser libre y abierto, cuenta con la capacidad de poder ser usado en entornos multiplataforma,** es decir, aplicable sobre cualquier plataforma y cualquier sistema operativo (SO) por lo que las aplicaciones que se desarrollen con OpenCL no dependerán del hardware o SO.

Una característica relevante es que OpenCL **se puede usar tanto para CPU, como para GPU,** aunque se necesita que el *hardware* o *software* sean relativamente modernos para poder soportarlo.

El primer paso que se debe realizar es la configuración de OpenCL, que será distinta según el *hardware* con el que se cuente, ya sea la CPU, la GPU e incluso puede ser configurado para NVIDIA.

Para profundizar sobre las configuraciones de OpenCL y los requisitos de hardware y software consulten el artículo de MQL5.

Terminología

Para familiarizarse con OpenCL es importante presentar la terminología que se usa, al igual que se trató en CUDA. En este caso se cuenta con:

Devices

Procesos computacionales, que se denomina *devices* (dispositivos) con una o más unidades de cálculo.

Kernel

Cada una de las funciones que se ejecutan en paralelo sobre los dispositivos se conocen como *kernel* o núcleo, igual que en CUDA. Estas funciones están escritas en el lenguaje OpenCL y serán compiladas para su ejecución sobre los dispositivos.

Work-ítem

La invocación y ejecución de cada *kernel* es un *work-ítem*, y cuenta con un identificador que es llevado a cabo por una aplicación escrita en C o C++, de forma que los *kernel* se irán poniendo en la cola de espera del dispositivo correspondiente. El dominio computacional es de uno o hasta tres dimensiones.

Los *work-ítem* se pueden agrupar en un *workgroup*, lo que les permitirá compartir memoria y sincronizarse.

Memoria

La memoria está organizada de forma que cada *work-ítem* cuenta con una memoria exclusiva, necesaria para almacenar la información que se requiere para la ejecución del *kernel*. Además, existe una memoria compartida dentro del *workgroup* y la memoria global es compartida entre los mismos.

Ejemplo 2. Un ejemplo sencillo de OpenCL

Se quiere paralelizar la ejecución sobre dos *array* de datos que sume los elemento de los dos *arrays* generando un *array* nuevo. Un *kernel* para esta función sería:

```
kernel void suma(globalconst float *a,globalconst float *b, globalfloat *resultado){
        int id = get_global_id(0);
                    /* se obtiene el identificador del hilo
        resultado[id] = a[id] + b[id];
}
```

Figura 8. *Kernel* en OpenCL que suma los elementos de dos *arrays*. Fuente: elaboración propia.

Flujo de ejecución

El flujo de ejecución habitual consta de los siguientes pasos:

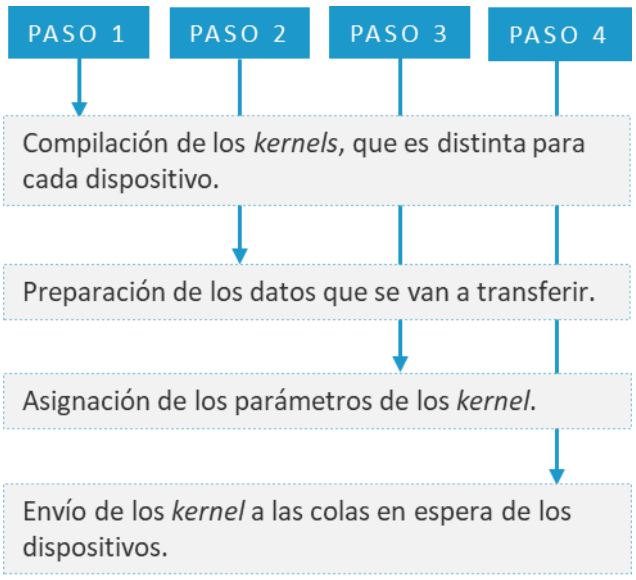


Figura 9. Pasos del flujo de ejecución. Fuente: elaboración propia.

Para profundizar sobre todas las cuestiones relacionadas con OpenCL se cuenta con su registro, que contiene especificaciones formateadas de la API OpenCL, el lenguaje de programación OpenCL C, el entorno OpenCL SPIR-V y las extensiones OpenCL.

Para acceder y consultar Khronos OpenCL Registry pueden visitar su página oficial.

Al igual que con CUDA se cuenta con una extensión para Python, PyOpenCL que se estudiará a continuación.

PyOpenCL

PyOpenCL es una biblioteca que proporciona acceso a la API de OpenCL desde Python, desarrollada por Andreas Klöckner. Es un **proyecto de código abierto bajo licencia MIT.**

Cuenta con características similares a las de PyCUDA con respecto a la recogida de basura automática ligada al tiempo de vida de un objeto, un mecanismo de abstracción sobre las estructuras de datos y los errores, con una sobrecarga mínima.

El objetivo principal de PyOpenCL **es permitir el uso de OpenL mediante una abstracción ligera de Python,** aunque también soporta, entre otros, la metaprogramación y las plantillas.

El flujo de un programa PyOpenCL **es casi exactamente el mismo que el de un programa C o C++ para OpenCL.** El programa «padre» prepara el *kernel* del código que se va a ejecutar sobre el dispositivo y los datos para la ejecución, lo lanza y luego espera el resultado.

Para empezar a trabajar con PyOpenCL es necesario preparar el entorno de desarrollo compatible con la arquitectura OpenCL.

En A fondo se proporciona el enlace a la documentación oficial de PyOPenCL con los requisitos necesarios para la instalación y uso.

Para probar PyOpenCL con las GPU de Colab se debe proceder de forma similar a como se vio para PyCUDA, es decir, se debe cambiar el entorno de ejecución y seleccionar la GPU.

Entorno de desarrollo para OpenCL

Antes de pasar a ver como empezar a implementar una aplicación en PyOpenCL es importante señalar que la aplicación contará con los elementos descritos para OpenCL, como los dispositivos (*device*), que pueden ser de una gran variedad de tipos, y núcleos (*kernels*), que es el código que se va a ejecutar sobre un dispositivo y que es una función C que podrá ser compilada por cualquier dispositivo con los drivers de PyOpen CL.

Se deben tener en cuenta las agrupaciones de *kernels*, que se denomina programa y que realizará la tarea de asignar los *kernels* a los dispositivos, **y el contexto,** que es el grupo de dispositivos. Es muy importante la **definición de la cola de ordenes o comandos,** que ordena la ejecución de los núcleos en el dispositivo correspondiente.

Como requisito se debe tener instalado el módulo `pyopencl`.

```
!pip install pyopencl # install cuda
```

Figura 10. Instalación de `pyopenc`. Fuente. Elaboración propia.

Para la configuración del entorno se importan los módulos que se necesiten y se define la plataforma, el dispositivo, el contexto y la cola.

```
import pyopencl as cl
plataforma = cl.get_platforms()[0]
dispositivo = plataforma.get_devices()[0]
contexto = cl.Context([dispositivo])
cola = cl.CommandQueue(contexto)
```

Figura 11. importación del módulo `pyopencl` y definiciones necesarias para la ejecución. Fuente: elaboración propia.

Obsérvese que se está seleccionando la primera de las plataformas y el primer dispositivo.

Dada la versatilidad de OpenCL es interesante conocer los dispositivos disponibles y sus características para su selección y explotación de su rendimiento.

```
import pyopencl as cl
def informacion_dispositivos() :
    print('Plataformas y dispositivo OpenCL ')
    for plataforma in cl.get_platforms():

        print('Nombre de la plataforma: ' + plataforma.name)
        print('Proveedor: ' + plataforma.vendor)
        print('Version: ' + plataforma.version)
        print('Perfil: ' + plataforma.profile)

        for dispositivo in plataforma.get_devices():

            print(' Nombre del dispositivo ' \
                  + dispositivo.name)
            print(' Tipo del dispositivo: ' \
                  + cl.device_type.to_string(dispositivo.type))
            print(' Velocidad máxima del reloj: {0} Mhz'\
                  .format(dispositivo.max_clock_frequency))
            print(' Unidades de computo: {0}'\
                  .format(dispositivo.max_compute_units))
            print(' Memoria local: {0:.0f} KB'\
                  .format(dispositivo.local_mem_size/1024.0))
            print(' Memoria constante: {0:.0f} KB'\
                  .format(dispositivo.max_constant_buffer_size/1024.0))
            print(' Memoria global: {0:.0f} GB'\
                  .format(dispositivo.global_mem_size/1073741824.0))
            print(' Tamaño máximo del buffer/imagenes: {0:.0f} MB'\
                  .format(dispositivo.max_mem_alloc_size/1048576.0))
            print(' Tamaño máximo del grupo de trabajo: {0:.0f}'\
                  .format(dispositivo.max_work_group_size))
    print('\n')

informacion_dispositivos()
```

Figura 12. Código para obtener información de la plataforma y el dispositivo. Fuente: elaboración propia.

Y se obtiene la siguiente información:

```
Plataformas y dispositivo OpenCL
Nombre de la plataforma: NVIDIA CUDA
Proveedor: NVIDIA Corporation
Version: OpenCL 1.2 CUDA 11.2.109
Perfil: FULL_PROFILE
 Nombre del dispositivo Tesla T4
 Tipo del dispositivo: ALL | GPU
 Velocidad máxima del reloj: 1590 Mhz
 Unidades de computo: 40
 Memoria local: 48 KB
 Memoria constante: 64 KB
 Memoria global: 15 GB
 Tamaño máximo del buffer/imagenes: 3777 MB
 Tamaño máximo del grupo de trabajo: 1024
```

Figura 13. Captura de pantalla con la información del código de la Figura 12. Fuente: elaboración propia.

Al igual que PyCUDA, PyOpenCL cuenta con múltiples funcionalidades sobre las que podrán profundizar consultando el recurso de documentación que se proporciona en la sección A fondo.

Pasos que se deben seguir para la ejecución de un kernel

Los pasos básicos son los siguientes:

- Se debe definir un *kernel* en el que se debe capturar la identificación del hilo que lo va a ejecutar. Es una función escrita en C con la siguiente sintaxis en la cabecera: `__kernel void nombre_del_kernel (argumentos){código}`.

- Se genera el contexto y se asigna al *kernel* mediante la declaración de un programa: `programa = cl.Program(contexto, codigo del kernel).build()`.

- Se declaran las variables en el *host* y se asigna memoria en GPU para las variables que vaya a utilizar el *kernel*. Por convención, se aconseja que el identificador de las variables del *host* empiece por h, y por g o por d el identificador de las del dispositivo.

- Se lanza el *kernel* y se encola en la cola de trabajos para el dispositivo. Cuando termina la ejecución se copian los datos de resultados en la correspondiente variable del *host* al desencolar el trabajo: `programa.nombre_kernel(cola, dato.shape, None, argumentos_del_kernel)`.

Al igual que se ha comentado en PyCUDA, se debe tener en cuenta el uso de la simple precisión en los datos de coma flotante.

Asignación de memoria

Una de las cuestiones que se deben considerar es la preparación de los datos para ser transferidos al dispositivo correspondiente. Se debe realizar:

- **Asignación de memoria para los datos que se van a procesar.** Se cuenta con la siguiente instrucción para datos de entrada: `vector_gpu = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=vector_entrada)`.

- **Asignación de memoria para los datos de salida.** En la siguiente instrucción se asigna memoria para el resultado del procesamiento: `resultado_gpu = cl.Buffer(context, mf.WRITE_ONLY, vector.nbytes)`.

9.5. Programación GPGPU con Numba

Numba es un **compilador `jit`** (*Just In Time*) de Python **de alto rendimiento**, haciendo uso de la infraestructura LLVM (*Low Level Virtual Machine*), que traduce el *bytecode*, es decir, el código intermedio correspondiente a las funciones implementadas en Python. **El código máquina optimizado se obtiene en tiempo de ejecución.**

Está diseñado para usar con *arrays* y funciones de Numpy, además genera código optimizado y con mejor rendimiento usando, tanto la CPU, como la GPU.

Sin embargo, se debe tener en cuenta que **existen dos modos de funcionamiento** de Numba. Con uno de ellos no se consigue mejoras de rendimiento significativas. El modo **con el que se consigue mejorar el rendimiento es el que se conoce como modo *nopython*.**

Al usar Numba lo que se consigue es eliminar una de las dependencias de PyCuda y PyOpenCL, que es la necesidad de implementar los núcleos en código C o C++, con lo que consigue un rendimiento similar al de estos lenguajes. Sin embargo, hoy por hoy, sigue siendo **PyCUDA la forma más eficiente de programar en CUDA.**

Lo importante es conocer qué se puede usar de Python y qué no, para evitar problemas de programación, con el referente de que Numba no mejora el rendimiento de cualquier código Python, sino que **está enfocado en el cálculo de operaciones matemáticas con *arrays* de datos numéricos** de tipo `int`, `float` o `complex`.

Para poder empezar a usar Numba se debe instalar mediante la siguiente orden: `conda install numba` o bien con `pip install numba`.

Para profundizar en Numba y sus características consulten su página.

Se presenta a continuación un ejemplo que sirve para tratar los distintos aspectos que se deben considera y para obtener su rendimiento.

Ejemplo 3. Implementación de una función que devuelve los valores mínimos locales dentro de una malla o tabla de números

Estos mínimos serán aquellos valores que son menores que los ocho que se encuentran a su alrededor. Primero, se presenta la forma de hacerlo sin usar Numba y se extrae el rendimiento.

La junción `buscar_min` devuelve dos *arrays* con las posiciones de los mínimos locales.

Para poder entender el ejemplo se presenta primero una ejecución sobre una malla de 6x6 y se la muestra, los resultados obtenidos y los valores de las celdas que tienen el mínimo.

```
import numpy as np
import numba
np.random.seed(0)

#se crea una malla de 36 números
data = np.random.randn(6, 6)
print("Los datos son:\n")
print(data)

def busca_min(malla):
    minimos_columna= []
    minimos_fila = []
    """recorre las celdas de la mañana.El mínimo local se encontrará en la celda
    cuyo valor cumpla todas las condiciones"""
    for i in range(1, malla.shape[1]-1):
        for j in range(1, malla.shape[0]-1):
            if (malla[j, i] < malla[j-1, i-1] and
                malla[j, i] < malla[j-1, i] and
                malla[j, i] < malla[j-1, i+1] and
                malla[j, i] < malla[j, i-1] and
                malla[j, i] < malla[j, i+1] and
                malla[j, i] < malla[j+1, i-1] and
                malla[j, i] < malla[j+1, i] and
                malla[j, i] < malla[j+1, i+1]):
                minimos_columna.append(i)
                minimos_fila.append(j)
    return np.array(minimos_fila), np.array(minimos_columna)



print("El resultado es:\n")
print(busca_min(data))
print(data[1,1])
print(data[3,2])
```

Figura 14. Código de una función que devuelve los valores mínimos locales dentro de una malla o tabla de números. Fuente: elaboración propia.

Los resultados en los que se ve que las celdas que contienen un valor mínimo local son la [1,1] y la [3,2].

```
Los datos son:

[[ 1.76405235  0.40015721  0.97873798  2.2408932   1.86755799 -0.97727788]
 [ 0.95008842 -0.15135721 -0.10321885  0.4105985   0.14404357  1.45427351]
 [ 0.76103773  0.12167502  0.44386323  0.33367433  1.49407907 -0.20515826]
 [ 0.3130677  -0.85409574 -2.55298982  0.6536186   0.8644362  -0.74216502]
 [ 2.26975462 -1.45436567  0.04575852 -0.18718385  1.53277921  1.46935877]
 [ 0.15494743  0.37816252 -0.88778575 -1.98079647 -0.34791215  0.15634897]]
El resultado es:

(array([1, 3]), array([1, 2]))
-0.1513572082976979
-2.5529898158340787
```

Figura 15. Resultado de la ejecución del código anterior. Fuente: elaboración propia.

Se ve a ahora para una malla de cuatro millones de números y se obtiene su tiempo de ejecución.

```python
def busca_min(malla):
    minimos_columna= []
    minimos_fila = []
    """recorre las celdas de la mañana.El mínimo local se encontrará en la celda
    cuyo valor cumpla todas las condiciones"""
    for i in range(1, malla.shape[1]-1):
        for j in range(1, malla.shape[0]-1):
            if (malla[j, i] < malla[j-1, i-1] and
                malla[j, i] < malla[j-1, i] and
                malla[j, i] < malla[j-1, i+1] and
                malla[j, i] < malla[j, i-1] and
                malla[j, i] < malla[j, i+1] and
                malla[j, i] < malla[j+1, i-1] and
                malla[j, i] < malla[j+1, i] and
                malla[j, i] < malla[j+1, i+1]):
                minimos_columna.append(i)
                minimos_fila.append(j)
    return np.array(minimos_fila), np.array(minimos_columna)


%timeit busca_min(data)
```

```
1 loop, best of 5: 5.11 s per loop
```

Figura 16. Código de una función que devuelve los valores mínimos locales dentro de una malla con 4 millones de números. Fuente: elaboración propia.

Uso de Numba. `numba.jit (nopython=True)`

Se va a usar Numba en el modo no Python porque soporta más códigos.

```python
busca_min_jit = numba.jit(nopython=True)(busca_min)
%timeit busca_min_jit(data)
```

```
The slowest run took 6.76 times longer than the fastest. This could mean that an intermediate result is being cached.
1 loop, best of 5: 74.4 ms per loop
```

Figura 17. Código de uso de `numba.jit`. Fuente: elaboración propia.

Se puede ver que ha mejorado considerablemente el rendimiento.

Este ejemplo está tomado a partir del proporcionado en *Cómo acelerar tu código Python con numba – Pybonacci*, una web que proporciona tutoriales y soluciones de Python para aplicaciones científicas.

Decorador `@guvectorize`

El decorador `guvectorize()` **permite escribir funciones que trabajarán con un número arbitrario de elementos de *arrays* de entrada y devolver *arrays* de**

diferentes dimensiones. No devuelven el *array*, sino que lo recibe como parámetro y es rellenado por la propia función.

Para profundizar sobre `guvectorize` consulten el manual proporcionado por Numba y los ejemplos de uso en la página de Hot Examples.

Anexo. Metaprogramación en Python

Un ejemplo habitual de metaprogramación en Python son los **decoradores que permiten cambiar el comportamiento de una función.** Por ejemplo:

```
@decorador
def funcion (argumentos):
            …
equivale a
funcion=decorador(funcion)
```

Figura 18. Ejemplo de decoradores. Fuente: elaboración propia.

El decorador recibe una función como parámetro y lo importante es la definición de este.

Para profundizar sobre metaprogramación, decoradores y su implementación en Pyhton pueden consultar los tutoriales de la página de IBM.

9.6. Referencias bibliográficas

Chatterjee, D., Deorio, A. y Bertacco, V. (2011). Gate-Level Simulation with GPU Computing. *ACM Trans. Design Autom. Electr. Syst., 16*(3), pp. 1-26. https://doi.org/10.1145/1970353.1970363

Documentician (s. f.). *Welcome to PyOpenCL's documentation!* [Código]. Documentician. pyopencl 2021.2.6 documentation (tician.de)

Gulati, K., Croix, J. F., Khatri, S. P. y Shastry, R. (2009). *Fast circuit simulation on graphics processing units*. Asia and South Pacific Design Automation Conference [Conferencia], pp. 403-408.

Numba (s. f.). *Examples* [Código]. Numba. Examples — Numba 0.50.1 documentation (pydata.org)

Zambrano, N. (s. f.). *Arquitectura CPU y GPU* [Imagen]. Internet paso a paso. 【GPU】¿Qué es la Unidad de Procesamiento Gráfico del PC? ▷ 2021 (internetpasoapaso.com)

9.7. Cuaderno de ejercicios

Ejercicio 1. Un ejemplo de uso de gpuarray y de procesamiento en la GPU

```
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
import time as t
#se define un array de numpy con números aleatorios de tamaño #1024.
#Importante, el tipo debe ser de simple precisión
x = np.random.randn(100000000).astype(np.float32)
#se convierte a gpuarray, con esto se asigna memoria para la GPU
x_gpu = gpuarray.to_gpu(x)
#se realizan las operaciones, gpuarray proporciona la característica
#de que cada multiplicación sea ejecutada por un hilo
inicio=t.time()
x_triple=3*x_gpu
final=t.time()-inicio
print("Tiempo ejecución en GPU %f"%final)
#se transfieren los datos al host
x_host = x_cuadrado.get()
inicio=t.time()
x=3*x
final=t.time()-inicio
print("Tiempo ejecución secuencial %f"%final)

Tiempo ejecución en GPU 0.002142
Tiempo ejecución secuencial 0.077480
```

Figura 19. Uso de gpuarray. Comparación de tiempo de ejecución paralela y secuencial. Fuente: elaboración propia.

Es importante especificar los tipos de datos de coma flotante para evitar sobrecargas innecesarias, se puede realizar aplicando al *array* el método `astype` o con la opción `dtype` en el constructor de la clase `numpy.array`.

Ejercicio 2. Usar PyCUDA para paralelizar el código de una función que eleva al cuadrado todos los elementos de un *array* con un millón de elementos

```
import numpy as np
import time as t
import pycuda.autoinit
from pycuda.compiler import SourceModule
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray
#se define un array de numpy con números un millón de números aleatorios.  Se pasa simple precisión
x = np.random.randn(1000000).astype(np.float32)

#se construye un nuevo  array con los cuadrados de los elementos de x
#esto es equivalente a un bucle de un millón de iteraciones
inicio = t.time()
x_cuadrado=x*x
total=t.time()-inicio
print("El tiempo de la ejecución secuencial es %f"%total)

# se define el kernel que se va ejecutar
kernel = SourceModule("""  __global__ void cuadrado( float *x) {
const unsigned int i = threadIdx.x + threadIdx.x*blockDim.x;
x[i] = x[i]*x[i];}
""" )
#se da nombre  al kernel
cuadrado = kernel.get_function("cuadrado")
#se preparan los datos para la transferencia
x_gpu = gpuarray.to_gpu(x)

inicio = t.time()
#se ejecuta el kernel
cuadrado(x_gpu, block=(1024, 1, 1), grid=(1,1,1))
salida_gpu = gpuarray.empty_like(x_gpu)
total=t.time()-inicio
print("El tiempo  de la ejecución en paralelo en la GPU es %f"%total)
#con get() se obtienen los datos nuevos
```

```
El tiempo de la ejecución secuencial es 0.001048
El tiempo  de la ejecución en paralelo en la GPU es 0.000370
```

Figura 20. Código pata la ejecución paralela de una función que eleva al cuadrado todos los elementos de un *array* con un millón de elementos. Fuente: elaboración propia.

Ejercicio 3. Cálculo paralelo sobre GPU del cuadrado de un cantidad de 50 millones de datos usando PyOpenCL

Se realiza la comparación del tiempo de ejecución paralela con la secuencial. En la definición del *kernel* se obtiene el identificador del hilo mediante la función `get_global_id(0)`, lo que permite que cada operación sea realizada por hilos distintos

```
import numpy as np
import pyopencl as cl
from time import time
#definicion del kernel
codigo_kernel = """
__kernel void square ( __global float* input, __global float* output) {
unsigned int i = get_global_id(0);
output[i] = input[i] * input[i] ; }
"""
contexto = cl.create_some_context ()
cola = cl.CommandQueue (contexto)
#se crea el programa
programa = cl.Program(contexto, codigo_kernel).build()
#array de 50 millones de número en coma flotante como dato de entrada. Es importante definir la precision
h_entrada = np.arange(50000000).astype(np.float32)
#variable de salida, en este caso del mismo tipo que la de entrada
h_salida = np.empty(50000000).astype(np.float32)
#opciones de la memoria asignada
mf = cl.mem_flags
#datos de entrada para  el dispositivo
d_entrada = cl.Buffer(contexto, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_entrada)
#datos de salida del dispositivo
d_salida = cl.Buffer(contexto, mf.WRITE_ONLY, h_salida.nbytes)
inicio=time()
#invocación  al kernel con la cola, la forma de los datos y un parametro por defecto+ parametros del kernel
programa.square(cola,h_entrada.shape, None, d_entrada, d_salida)
cola.finish()
total=time()-inicio
print ("La ejecución del kernel ha durado", total, "segundos")
# se obtiene la salida del kernel
cl.enqueue_copy(cola, h_salida, d_salida)
print ("El resultado es ",h_salida)
#ejecución secuencial
i= 0
inicio = time()
while i < len(h_salida):
  h_salida[i] = h_entrada[i] * h_entrada[i]
  i+=1
total = time()-inicio
print ("la salida secuencial es",h_salida)
print ("La ejecución secuencial ha  durado", total, "segundos")
```

Figura 21. Código para el cálculo paralelo sobre GPU del cuadrado de un cantidad de cincuenta millones de datos usando PyOpenCL. Fuente: elaboración propia.

La salida obtenida, en la que se ve el mejor rendimiento de la ejecución en paralelo, es:

```
La ejecución del kernel ha durado 0.04581856727600098 segundos
El resultado es  [0.0000000e+00 1.0000000e+00 4.0000000e+00 ... 2.4999995e+15 2.5000001e+15
 2.5000001e+15]
la salida secuencial es [0.0000000e+00 1.0000000e+00 4.0000000e+00 ... 2.4999995e+15 2.5000001e+15
 2.5000001e+15]
La ejecución secuencial ha  durado 31.698238372802734 segundos
```

Figura 22. Resultado de la ejecución del código anterior. Fuente: elaboración propia.

Se ve en los tiempos como el tiempo de ejecución es mucho menor en la ejecución paralela.

Ejercicio 4. Suma de dos vectores de 50 millones de dato

Esta implementación se ha realizado a partir del código encontrado en la página de Documentician.

```
import numpy as np
import pyopencl as cl
import time
#vectores de 500 millones de  datos
h_vector1 = np.random.rand(500000000).astype(np.float32)
h_vector2 = np.random.rand(500000000).astype(np.float32)
print(h_vector1)
print(h_vector2)
#se crea el contexto y la cola
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
#opciones de la asignación de memoria
mf = cl.mem_flags
#asignacion de memoria para los vectores en el dispositivo
d_vector1 = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_vector1)
d_vector2= cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_vector2)
#se crea el programa co  la definición ddel kernel
programa = cl.Program(ctx, """
__kernel void suma(
    __global const float *a_g, __global const float *b_g, __global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()
#memoria para el resultado en el dispositivo
d_res = cl.Buffer(ctx, mf.WRITE_ONLY, h_vector1.nbytes)
inicio=time.time()
#invocación del nucleo
programa.suma(queue, h_vector1.shape, None, d_vector1, d_vector2, d_res)
total=time.time()-inicio
print("Tiempo de ejecución paralela",total)
#vector en el que se recogen los resultados en el host
h_res = np.empty_like(h_vector1)
cl.enqueue_copy(queue, h_res, d_res)
print(h_res)
# ejecución en secuencial usando operadore de Numpy
inicio=time.time()
resultado=h_vector1+h_vector2
total=time.time()-inicio
print("Tiempo ejecución secuencial",total)
print(resultado)
```

Figura 23. Código para la suma de dos vectores de cincuenta millones de datos. Fuente: https://documen.tician.de/pyopencl/

Los resultados que se obtienen son:

```
[0.38576913 0.40784228 0.9602411  ... 0.500335   0.38496515 0.7768583 ]
[0.5113993  0.7056333  0.17923756 ... 0.6484     0.35099706 0.8618845 ]
Tiempo de ejecución paralela 0.856809139251709
[0.89716846 1.1134756  1.1394787  ... 1.148735   0.7359622  1.6387428 ]
Tiempo ejecución secuencial 0.5060977935791016
[0.89716846 1.1134756  1.1394787  ... 1.148735   0.7359622  1.6387428 ]
```

Figura 24. Resultado de la ejecución del código anterior. Fuente: elaboración propia.

Cabe destacar que el tiempo secuencial es mejor que el paralelo. Esto demuestra lo optimizadas que están las operaciones Numpy.
Desarrollen la teoría organizada por apartados. Tened en cuenta que el texto que redacten servirá a los alumnos para estudiar la materia correspondiente, así que, en la medida de lo posible, utiliza **frases sencillas** y fácilmente entendibles, evita los rodeos lingüísticos, las expresiones redundantes, etc.

Piensen que su material estará disponible para el estudiante en **formato web**: traten de no escribir párrafos excesivamente largos e intenten que haya cierta armonía entre ideas y párrafos.

Para facilitar el estudio, es interesante que incluyan elementos visuales, siempre que se pueda, como figuras o tablas. Se numerarán de forma independiente, por un lado, las figuras y, por otro lado, las tablas.

Ejercicio 5. Creación de fractales con Numba

El código que se muestra creará fractales usando el conjunto de Mandelbrot y se extrae de los ejemplos de la documentación oficial de Numba en los que se consiguen una mejora importante con el uso de este.

Primero se debe conocer lo que es un fractal, que es un objeto geométrico en el que se repite el mismo patrón a diferentes escalas y con diferente orientación.

El conjunto de Mandelbort, uno de los fractales más conocidos, se define en el plano complejo mediante una sucesión recursiva:

$$\begin{cases} z_0 = 0 \in \mathbb{C} \\ z_{n+1} = z_n^2 + c \end{cases}$$

Donde $c \in C$ pertenece al conjunto, si la sucesión queda acotada y, además, se conoce que todo punto que se encuentre a una distancia mayor de dos del origen no pertenece al conjunto.

El cálculo del conjunto para un complejo dado c con su parte real x e imaginaria y se implementa según esta función, para un máximo de iteraciones dadas y que retorna un color para los puntos que están en el conjunto y otro para los que no.

```
def mandel (x,y,max_iters):
    i=0
    c=complex(x,y)
    z=0.0j
    for i in range(max_iters):
        z=z*z+c
        if (z.real*z.real+z.imag*z.imag)>=4:
            return 1
    return 255
```

Figura 25. Código para la construcción de un conjunto de Mandelbort. Fuente: https://numba.pydata.org/numba-doc/latest/user/examples.html

Luego, se crea el fractal para un rango de valores para las partes reales e imaginarias, y un ancho y alto para la escala. Se rellena una matriz de enteros que representa los valores de color que se obtienen al invocar a la función `mandel`.

```
def create_fractal(min_x,max_x,min_y,max_y,image,iters):
    height=image.shape[0]
    width=image.shape[1]

    pixel_size_x=(max_x-min_x)/width
    pixel_size_y=(max_y-min_y)/height

    for x in range(width):
        real=min_x+x*pixel_size_x
        for  y in range(height):
            imag=min_y+y*pixel_size_y
            color=mandel(real,imag, iters)
            image[y,x]=color

    return image
```

Figura 26. Código para la creación de un fractal. Fuente: https://numba.pydata.org/numba-doc/latest/user/examples.html

Para representar gráficamente el fractal se usa `imshow`, que es una función para visualización de matrices que representa la imagen y la recibe como parámetro.

```
image=np.zeros((500*2,750*2),dtype=np.uint8)
inicio=time()
img=create_fractal(-2.0,1.0,-1.0,1.0,image,20)
print(time()-inicio)
imshow(img)
```

Figura 27. Código para la visualización de un fractal. Fuente: https://numba.pydata.org/numba-doc/latest/user/examples.html

La ejecución de este código da como resultado la siguiente imagen, tamién se muestra el tiempo de ejecución.

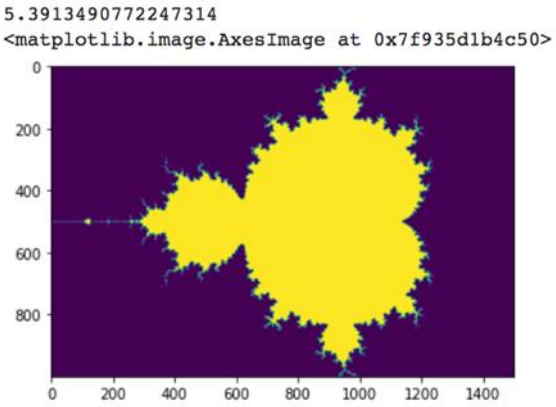


Figura 28. Resultado de la ejecución del código anterior. Fuente: elaboración propia.

Ahora se verá el rendimiento cuando se ejecuta con Numba, mediante el uso del decorador `jit`, en cada una de las funciones.

```python
from matplotlib.pylab import imshow, ion
import numpy as np
from time import time
from numba import jit

@jit
def mandel (x,y,max_iters):
    i=0
    c=complex(x,y)
    z=0.0j
    for i in range(max_iters):
        z=z*z+c
        if (z.real*z.real+z.imag*z.imag)>=4:
            return 1
    return 255

@jit
def create_fractal(min_x,max_x,min_y,max_y,image,iters):
    height=image.shape[0]
    width=image.shape[1]

    pixel_size_x=(max_x-min_x)/width
    pixel_size_y=(max_y-min_y)/height

    for x in range(width):
        real=min_x+x*pixel_size_x
        for  y in range(height):
            imag=min_y+y*pixel_size_y
            color=mandel(real,imag, iters)
            image[y,x]=color

    return image

image=np.zeros((500*2,750*2),dtype=np.uint8)
inicio=time()
img=create_fractal(-2.0,1.0,-1.0,1.0,image,20)
print(time()-inicio)
imshow(img)
```

Figura 29. Código para crear un fractal usando `jit`. Fuente: elaboración propia.

El resultado es:

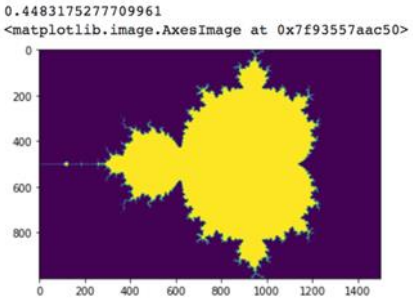


Figura 30. Resultado del código anterior. Fuente: elaboración propia.

Como se ve, mejora mucho su tiempo de ejecución.

A fondo

PyCUDA's documentation

Documentación oficial PyCUDA (https://documen.tician.de/pycuda/).

Página oficial de PyCUDA, desarrollada por Andreas Kloeckner, con instrucciones de instalación y tutoriales.

TPUs de Google

Grande, A. (2020*). TPUs de Google: el imparable avance del hardware especializado para ML*. Paradigma Digital. TPUs de Google: el imparable avance del hardware especializado para ML - Paradigma (paradigmadigital.com)

Se presentan unos dispositivos *hardware* para la aceleración del procesamiento especializado para las necesidades de procesamiento, que requiere el aprendizaje automático.

PyOpenCL's documentation

Documentación oficial de PyOpenCL (https://documen.tician.de/pyopencl/).

Página oficial de PyOIpenCL elaborada y desarrollada por Andreas Kloeckner, con instrucciones de instalación y tutoriales.

Test

1. La computación heterogénea se basa en:

A. El uso de distintos lenguajes de programación.

B. El uso de distintos paradigmas de programación.

C. El uso de *hardware* especializado para conseguir una mayor aceleración de la ejecución.

D. Ninguna de las anteriores es verdadera.

2. Las GPU:

A. Proporciona una forma de computación concurrente basada en *software*.

B. Proporciona una forma de computación concurrente basada en *hardware*.

C. Proporciona una forma de computación paralela basada en *software*.

D. Proporciona una forma de computación paralela basada en *hardware*.

3. CUDA es:

A. Una arquitectura propietaria para GPU de NVIDIA.

B. Una arquitectura libre para GPU de NVIDIA.

C. Una arquitectura propietaria para cualquier GPU.

D. Una arquitectura libre para cualquier GPU.

4. OpenCL es:

A. Un estándar propietario para programación paralela para GPU de NVIDIA.

B. Un estándar abierto para programación paralela para GPU de NVIDIA.

C. Un estándar propietario para programación paralela para GPU.

D. Un estándar abierto para programación paralela para GPU.

5. Un *kernel* de PyCUDA:

A. Es un núcleo de la GPU.

B. Es una función para ejecución paralela lanzada desde el proceso padre.

C. Es el proceso principal en programación paralela.

D. Ninguna de las anteriores es verdadera.

6. Un `gpuarray` es:

A. Un *array* de PyOpenCl para gestionar datos en la GPU.

B. Un *array* de PyOpenCl para gestionar datos en el *host*.

C. Un *array* de PyCUDA para gestionar datos en la GPU.

D. Ninguna de las anteriores es verdadera.

7. En OpenCL:

A. Cada *work-ítem* tiene memoria propia y memoria compartida con los *work-ítem* de su mismo *workgroup*.

B. Cada *work-ítem* solo tiene memoria propia.

C. Cada *work-ítem* no tiene memoria propia, pero sí memoria compartida con los *work-ítem* de su mismo *workgroup*.

D. Ninguna de las anteriores es verdadera.

8. Un *kernel* de PyOpenCL:

A. Es un núcleo de la GPU.

B. Es una función para ejecución paralela lanzada desde el proceso padre.

C. Es el proceso principal en programación paralela

D. Ninguna de las anteriores es verdadera.

9. Numba es:

A. Un estándar propietario para programación paralela para GPU de NVIDIA.

B. Un estándar abierto para programación paralela para distintos dispositivos.

C. Es un compilador `jit` que genera código optimizado.

D. Ninguna de las anteriores es verdadera.

10. El compilador Numba:

A. Genera código optimizado en tiempo de compilación.

B. Genera código optimizado en tiempo de ejecución.

C. No genera código.

D. Ninguna de las anteriores es verdadera.