Programación Científica y HPCI

Máster Universitario en Ingeniería Matemática y Computación

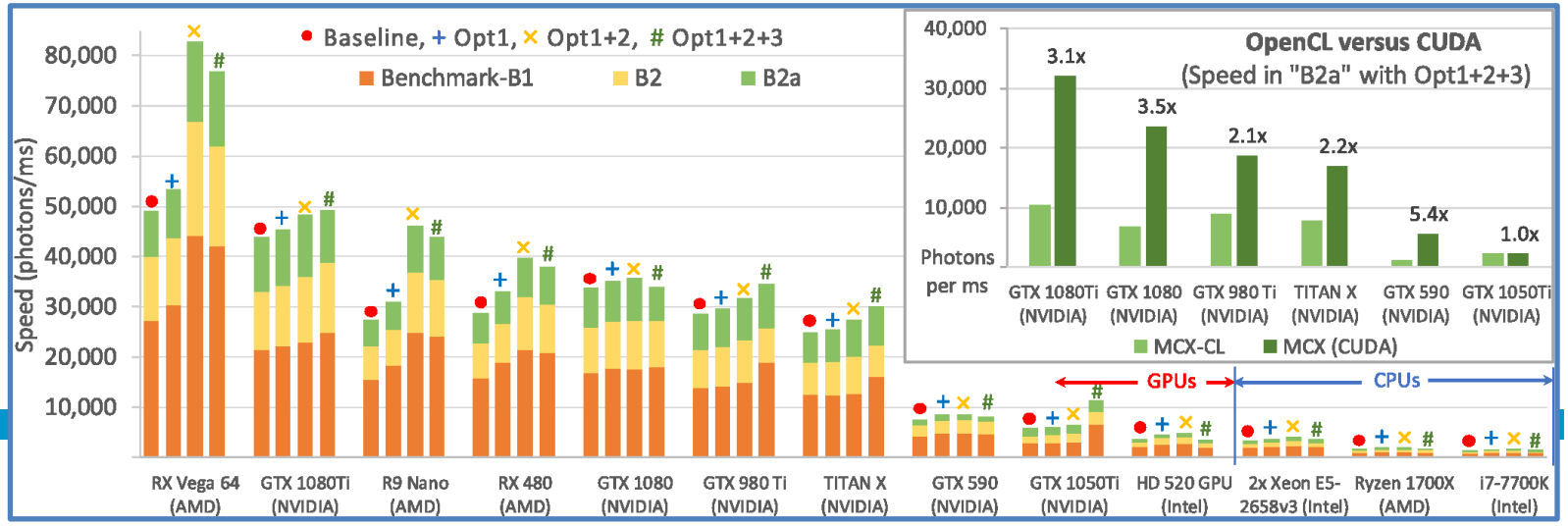
Tema 9-II / Programación Heterogénea- HPC

OpenCL y PyOpenCL

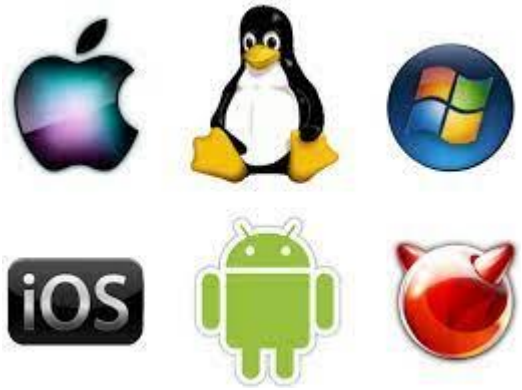
- OpenCL y su comparación con CUDA.
- OpenCL, estándar abierto compatible con distintos GPU y microprocesadores.
- Desarrollado por el grupo Khronos, con la contribución inicial de Apple y otras grandes empresas.

Rendimiento y comparativa con CUDA

- Resultados positivos en pruebas de rendimiento y comparativas con CUDA.
- Ligera ventaja de CUDA debido a su diseño específico para un tipo de GPU.
- Desafíos en la eficiencia debido a la necesidad de una interfaz común para distintos dispositivos.
- Avances prometedores en los últimos años.



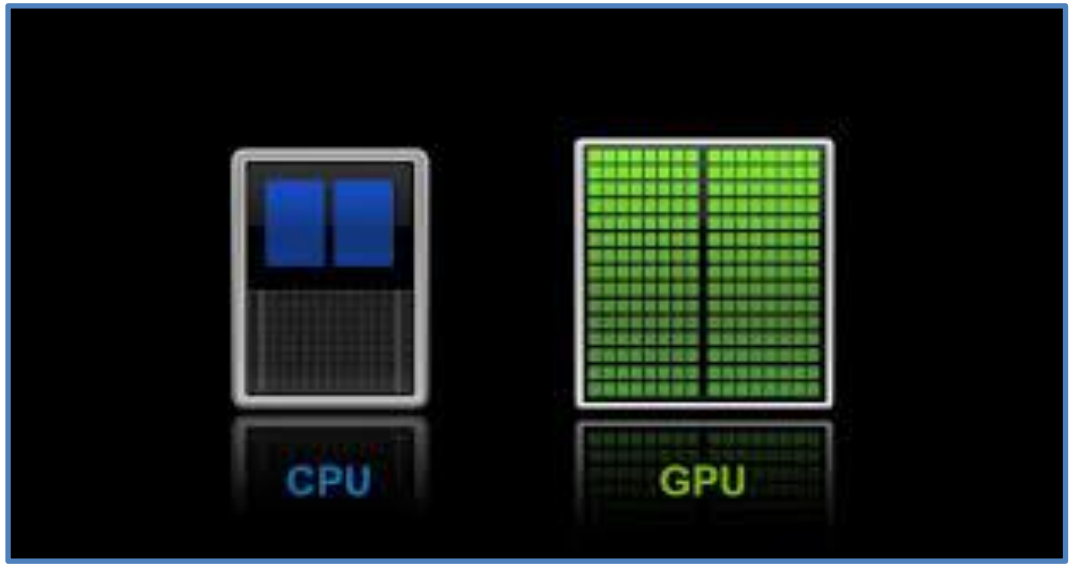
Multiplataforma y sin dependencia de hardware o SO



- Ventaja de OpenCL al ser libre y abierto, permitiendo su uso en entornos multiplataforma.
- Aplicable en cualquier plataforma y sistema operativo.
- Las aplicaciones desarrolladas con OpenCL no dependen del hardware o SO.

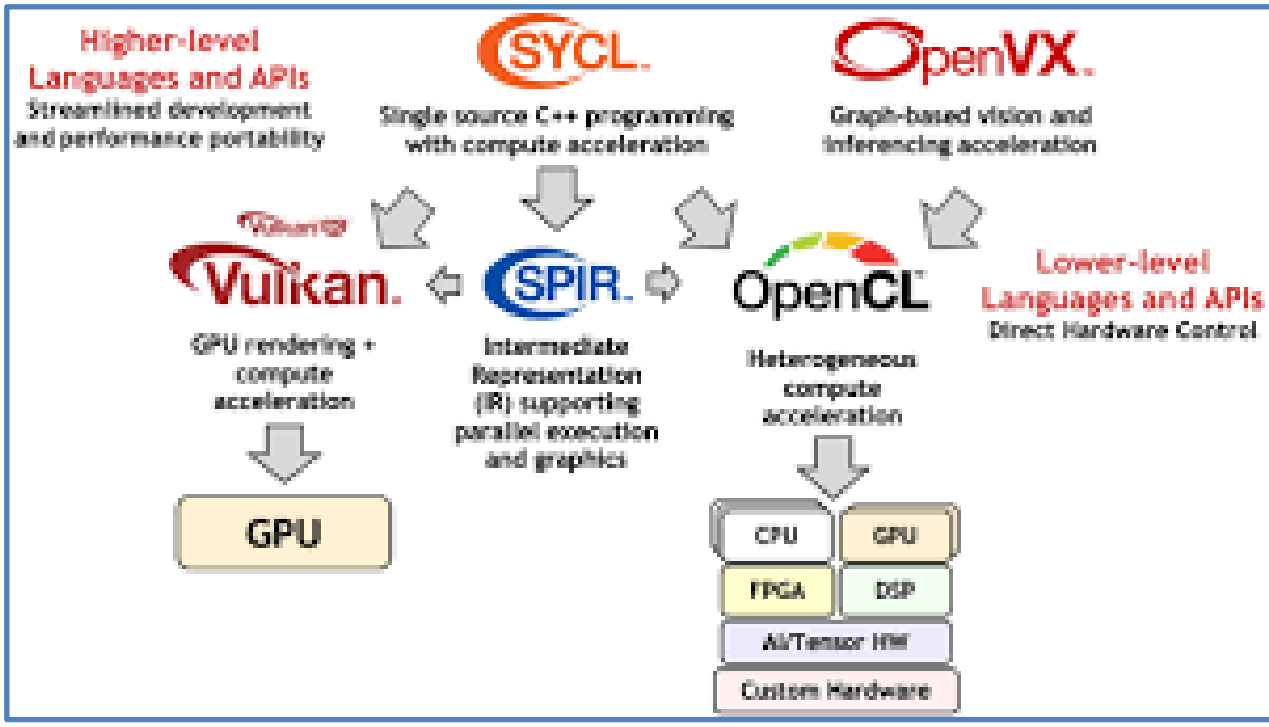
Uso en CPU y GPU

- OpenCL se puede utilizar tanto en CPU como en GPU.
- Requisitos de hardware o software relativamente modernos para soportar OpenCL.



Configuración de OpenCL

- El primer paso es la configuración de OpenCL según el hardware disponible.
- Configuración para CPU, GPU y posible configuración para NVIDIA.



Terminología de OpenCL

- Devices: Procesos computacionales con una o más unidades de cálculo.
- Kernel: Funciones ejecutadas en paralelo sobre los dispositivos.
- Work-ítem: Invocación y ejecución de cada kernel.
- Memoria: Organización de la memoria en work-ítem, workgroup y memoria global.

Circuito de memoria ROM

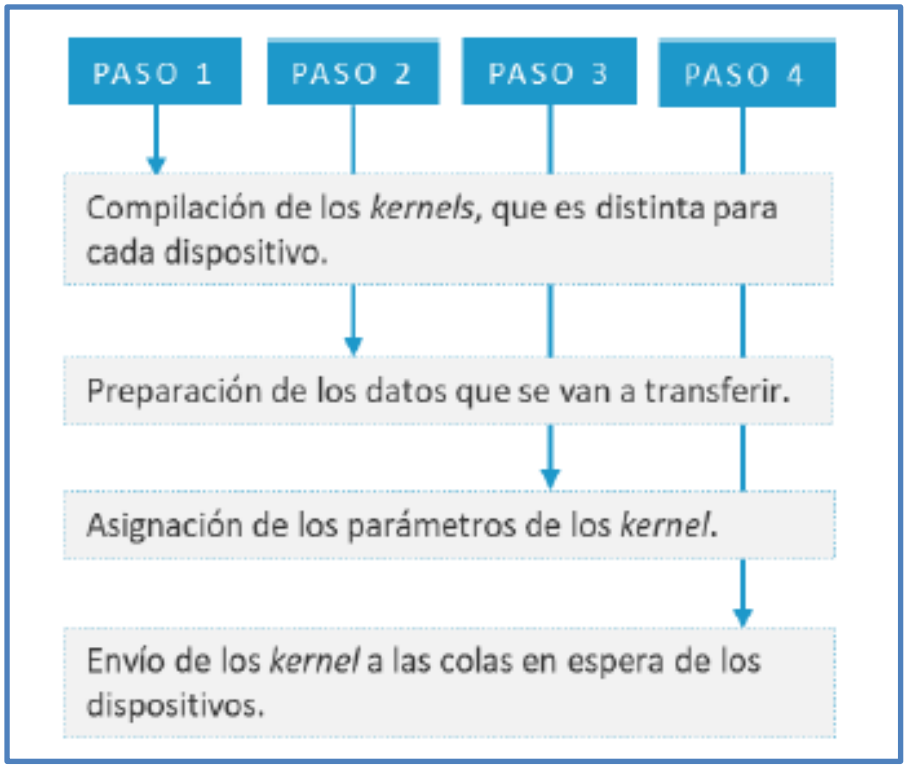
Memoria RAM
(Expandible e intercambiable)

Ejemplo de OpenCL

```
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];
}
```

- Ejemplo sencillo de un kernel en OpenCL que suma los elementos de dos arrays.

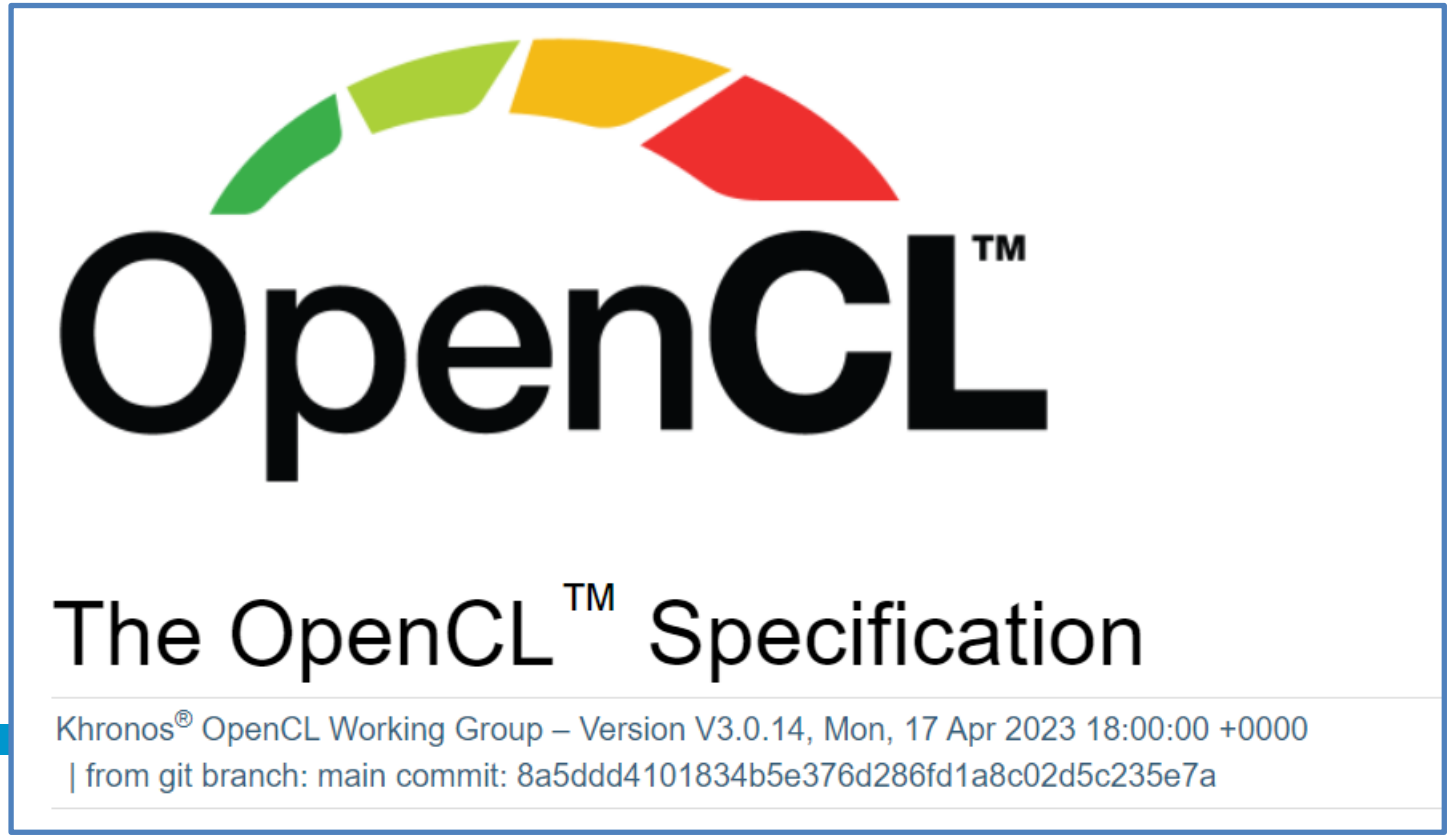
Flujo de ejecución de OpenCL



- Pasos: Compilación de los kernels, preparación de los datos, asignación de parámetros y envío de los kernels a las colas de espera.

Registro y documentación de OpenCL

- Referencia al registro de OpenCL que contiene especificaciones formateadas de la API, lenguaje de programación, entorno y extensiones.
- Importancia del registro para obtener información detallada sobre OpenCL.

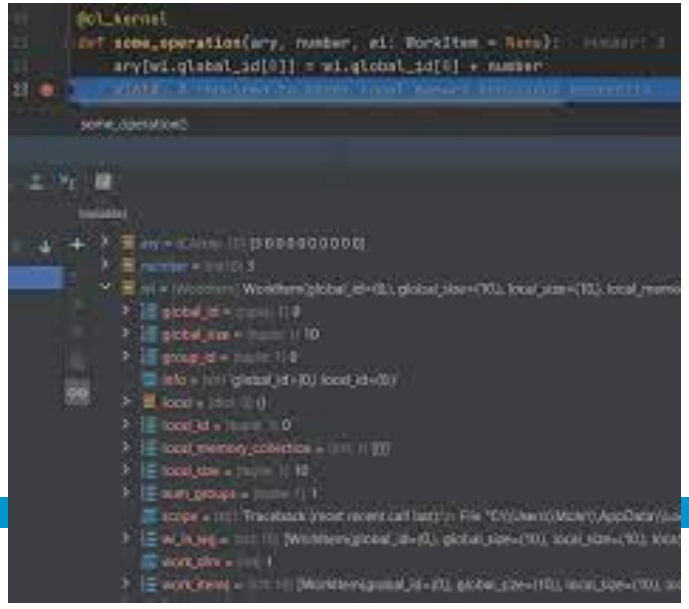


PyOpenCL - Introducción y características

- PyOpenCL como biblioteca para acceder a la API de OpenCL desde Python.
- Desarrollado por Andreas Klöckner como un proyecto de código abierto bajo licencia MIT.
- Características similares a PyCUDA en términos de recogida de basura automática y abstracción de estructuras de datos y errores.

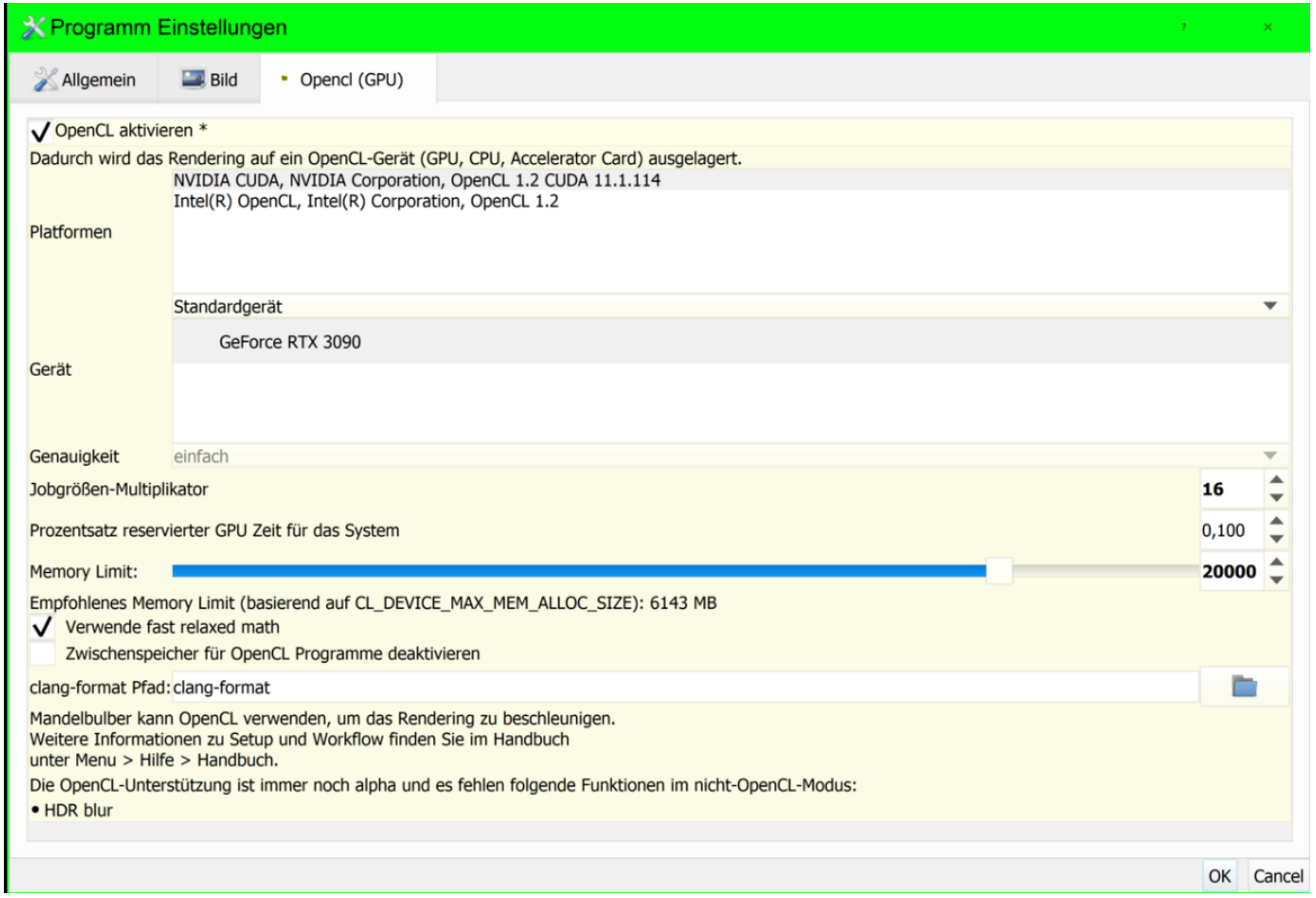
Uso de PyOpenCL

- Objetivo principal de PyOpenCL: permitir el uso de OpenCL mediante una abstracción ligera de Python.
- Soporte para metaprogramación y plantillas.
- Flujo de un programa PyOpenCL similar al de un programa C o C++ para OpenCL.



Configuración del entorno de desarrollo para OpenCL

- Importante preparar el entorno de desarrollo compatible con la arquitectura OpenCL.
- Tener en cuenta los elementos clave como dispositivos, núcleos, programas, contexto y cola de órdenes o comandos.
- Evaluar requisito de instalación del módulo PyOpenCL.



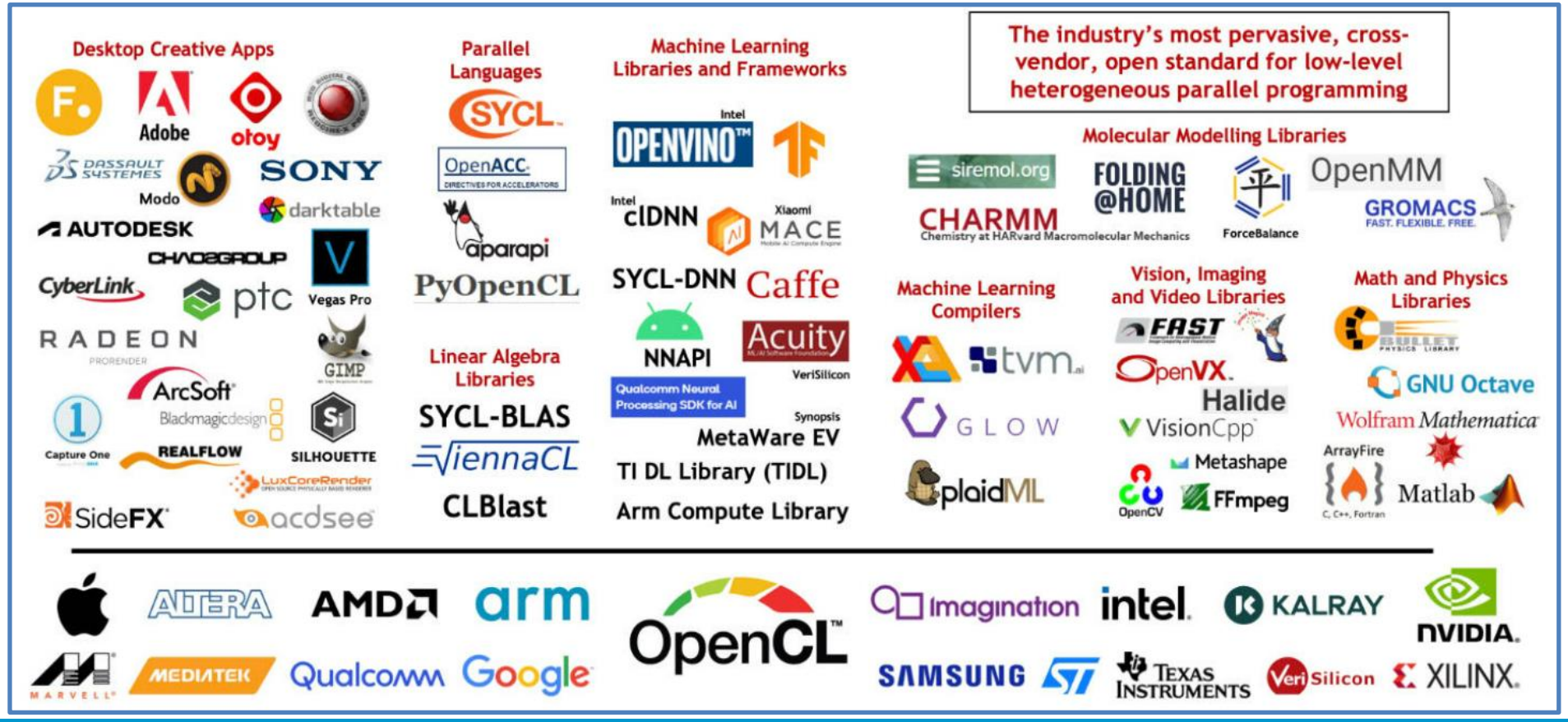
Configuración del entorno de desarrollo para OpenCL

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

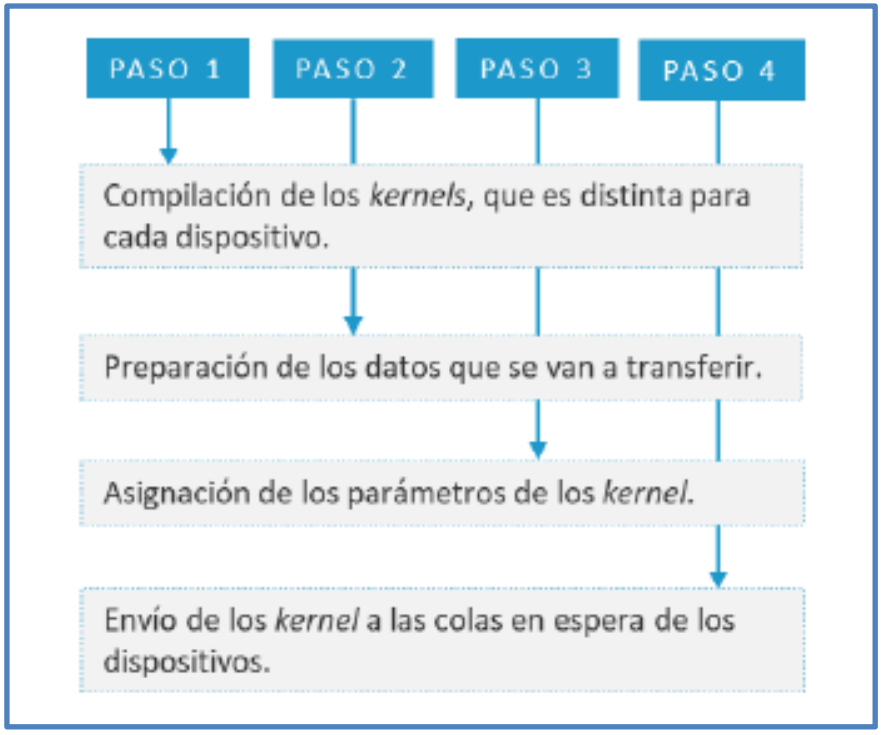
- Código de configuración del entorno: importación de módulos, definición de plataforma, dispositivo, contexto y cola.

Exploración de dispositivos en OpenCL

- Mención especial de la versatilidad de OpenCL y la importancia de conocer los dispositivos disponibles y sus características.
- Evaluar la selección y explotación de dispositivos para obtener un mejor rendimiento.



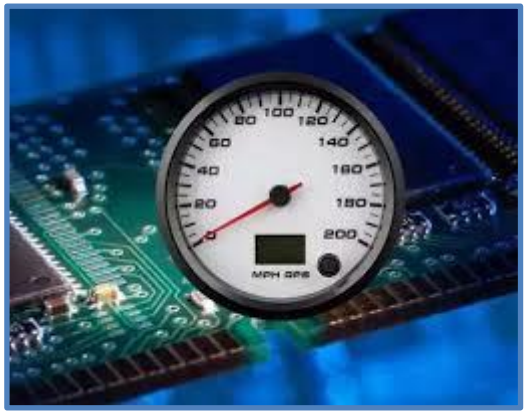
Pasos para la ejecución de un kernel en OpenCL



- Definición del kernel con la identificación del hilo.
- Generación del contexto y asignación del kernel al programa.
- Declaración de variables en el host y asignación de memoria en GPU.
- Lanzamiento del kernel y copia de los datos de resultados al host.

Asignación de memoria en OpenCL

- Importancia de la asignación de memoria para los datos a procesar.
- Ejemplo de asignación de memoria para datos de entrada y salida.
  - `resultado_gpu = cl.Buffer(context, mf.WRITE_ONLY, vector.nbytes)`
- Uso de la instrucción cl.Buffer() para asignar memoria en el dispositivo.



Programación GPGPU con Numba

- Numba como un compilador jit de Python de alto rendimiento.
- Uso de la infraestructura LLVM para generar código optimizado en tiempo de ejecución.
- Dos modos de funcionamiento de Numba: modo nopython y otro modo sin mejoras de rendimiento.
- Comparativa con PyCUDA y PyOpenCL en términos de rendimiento.
  - Hoy por hoy, sigue siendo **PyCUDA la forma más eficiente de programar en CUDA**

Uso de Numba con el modo nopython

- Importancia del modo nopython de Numba para obtener mejoras de rendimiento.
- Enfoque en el cálculo de operaciones matemáticas con arrays de datos numéricos.
- Instrucción de instalación de Numba con conda o pip.
  - **`conda install numba` o bien con `pip install numba`**

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

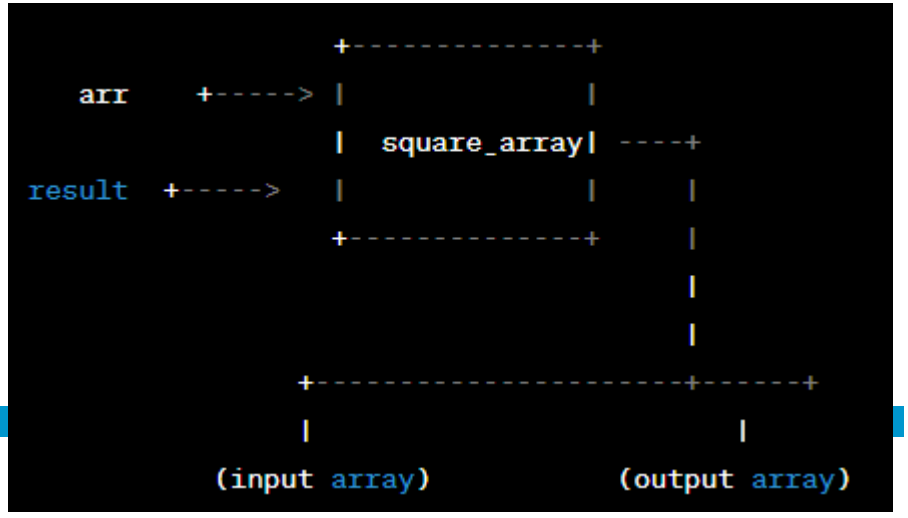
- Decorador numba.jit y el parámetro nopython=True.
- Mejora de rendimiento obtenida con el modo nopython.
- Uso de Numba para acelerar el código.

```python
import numba

@numba.jit
def add_numbers(a, b):
    return a + b

result = add_numbers(3, 5)
print(result)
```

Uso de Numba - decorador @guvectorize

- Decorador @guvectorize para funciones que trabajan con arrays de entrada y devuelven arrays de diferentes dimensiones.
- Funciones que reciben el array como parámetro y lo rellenan internamente.

```python
import numpy as np
from numba import guvectorize

@guvectorize(['void(int64[:], int64[:])'], '(n)->(n)', target='parallel')
def square_array(arr, result):
    for i in range(arr.shape[0]):
        result[i] = arr[i] * arr[i]
```



www.unir.net