ARQUITECTURA DE SISTEMAS DISTRIBUIDOS
3º GRADO EN ING. INFORMÁTICA. TEC. INFORMÁTICAS
PRÁCTICA X. Introducción a OpenCL.

1. MOTIVACIÓN Y OBJETIVOS.

Durante las prácticas anteriores hemos ido gradualmente abriendo el ámbito en donde explotamos el paralelismo: desde el paralelismo a nivel de instrucciones (superescalares, algoritmo de Tomasulo), hasta el paralelismo a nivel de procesos (MPI) pasando por el paralelismo a nivel de hilos (OpenMP). En la última práctica abriremos aún más el ámbito con el paralelismo a nivel de sistemas. Al principio el paralelismo se ha focalizado en un único sistema, con lo que era difícil llamarlo sistema distribuido, pero con MPI hemos pasado realmente a una arquitectura con sistemas distribuidos.

Sin embargo, y desde hace unos años existe un nuevo paradigma de computación: aquella que, estando aún confinada en un único sistema, hace uso de ejecución distribuida en donde el procesador del sistema hace las veces de director de orquesta, controlando y distribuyendo tareas a decenas o incluso cientos de nodos de ejecución que residen en la misma placa base que el procesador. Estos nodos de ejecución están presentes, entre otros sitios, en las tarjetas gráficas basadas en GPGPU (General Processing Graphics Processing Unit). Los procesadores gráficos tradicionales (GPUs) han adquirido capacidad de poder ejecutar programas independientes, y aunque siguen especializados en tareas propias de manejo de imágenes, sus unidades de cómputo son lo suficientemente flexibles como para realizar cualquier cálculo de propósito general: son las GPGPUs.

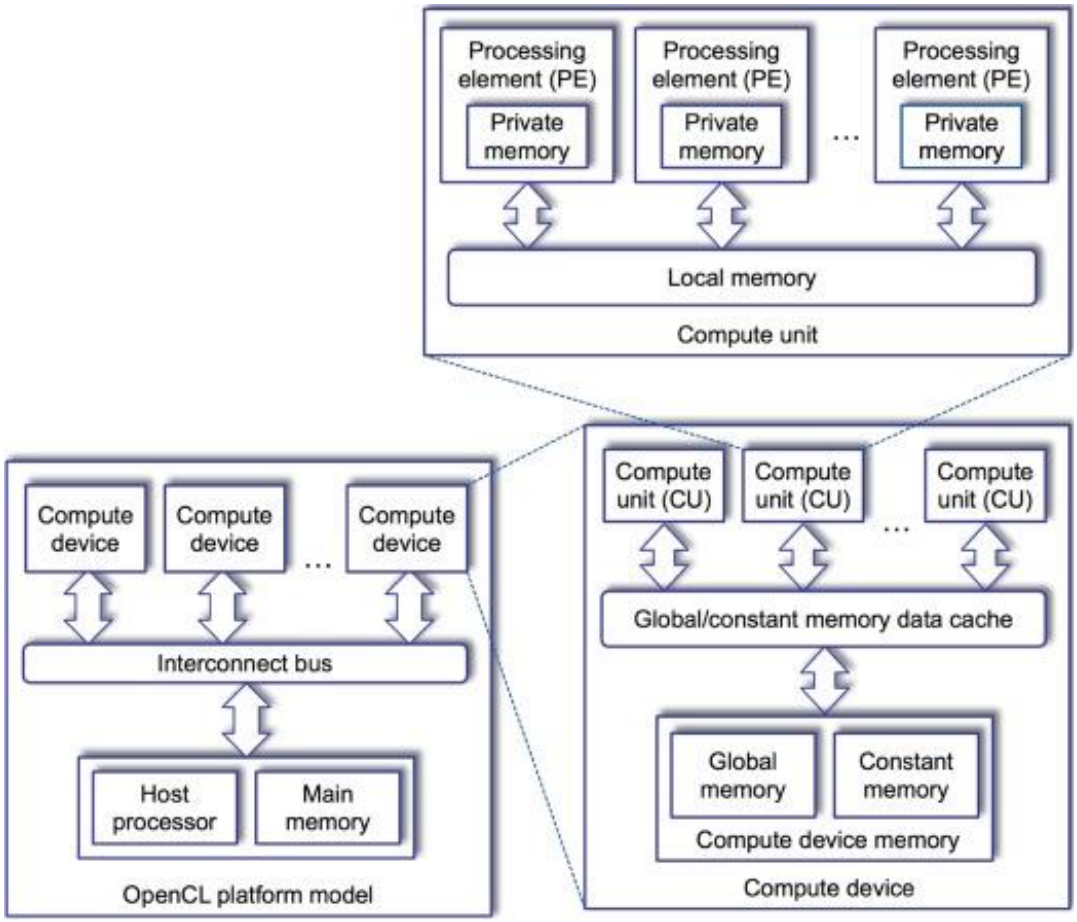
De esta forma, queremos con esta práctica introducir al alumnado a los conceptos básicos de la computación distribuida con GPGPUs. De las dos implementaciones disponibles, CUDA y OpenCL, hemos escogido esta última por no ceñirse unicamente a GPGPUs de un único fabricante (NVidia) y por abrir el abanico de la computación distribuida a dispositivos que no son necesariamente tarjetas gráficas. Por otro lado, OpenCL requiere de bastante código en la parte del procesador para arrancar todo el sistema y un ejemplo, aunque sea pequeño, tendrá bastantes más líneas de código fuente que su equivalente en CUDA.

Así, con esta práctica se pretenden los siguientes objetivos:

- Ver la relación entre host, plataformas y dispositivos, y cómo obtener información de las capacidades OpenCL disponibles en nuestro host.
- Escribir kernels y ejecutarlos en varios dispositivos completamente diferentes unos de otros.
- Comprobar la mejora que se puede llegar a alcanzar comparando una implementación de una operación gráfica (filtro de media, como el realizado en las prácticas 1 y 2) ejecutanda tanto en OpenCV como hicimos en aquella ocasión, como en un kérnel de OpenCL usando código propio.

2. (brevísima y necesariamente incompleta) INTRODUCCION A OPENCL

OpenCL es el acrónimo de "Open Computing Languaje": OpenCL es un API estándar e independiente de plataforma diseñado para soportar el desarrollo de aplicaciones de cómputo paralelo fácilmente portables en entornos de computación heterogéneos. Un sistema compatible con OpenCL consta de un host (un sistema basado en CPU) y uno o más dispositivos compatibles OpenCL conectados a él mediante algún tipo de bus de interconexión (por ejemplo, PCIe), tal y como se muestra en la siguiente figura:



Cada dispositivo de computación puede ser una tarjeta gráfica (integrada en el sistema o externa), la propia CPU del host (que puede hacer tanto el papel de procesador host como de dispositivo OpenCL), o tarjetas aceleradoras basadas en ASIC o FPGA. Un dispositivo de computación (compute device) contiene una memoria propia a la que se conectan, normalmente mediante una caché, varias unidades de computación (compute unit). Cada unidad de computación tiene su propia memoria local compartida entre varios elementos de procesamiento (processing element), cada uno de los cuales tiene su propia memoria privada.

La jerarquía mostrada se completa con un elemento más: las plataformas. Desde el punto de vista de una aplicación OpenCL, un host contiene una o más plataformas, cada una de ellas conteniendo uno o más dispositivos de computación, cada uno de ellos... etc.

Como ejemplo, se muestra la salida del primer ejercicio de esta práctica, en un sistema con dos plataformas y dos dispositivos (uno de ellos aparece "repetido" en ambas plataformas):

```
Plataforma: Intel(R) OpenCL
  Dispositivo: Intel(R) Pentium(R) CPU G4400    2 unidades de computacion.
    Workgroup size: 8192   Workgroup item sizes: 8192/8192/8192

Plataforma: AMD Accelerated Parallel Processing
  Dispositivo: Tahiti                          32 unidades de computacion.
    Workgroup size: 256   Workgroup item sizes: 128/128/128
  Dispositivo: Intel(R) Pentium(R) CPU G4400    2 unidades de computacion.
    Workgroup size: 1024   Workgroup item sizes: 1024/1024/1024
```

El modelo de ejecución paralela de OpenCL funciona a grandes rasgos de la siguiente forma: el host envía un trabajo (work) a un dispositivo OpenCL. Este trabajo es algún tipo de proceso (el kérnel) que se debe realizar a todos los elementos de un conjunto de datos (elementos de un vector, píxeles en una imagen, etc). OpenCL particiona el trabajo global en grupos de trabajos locales de un tamaño máximo igual a un valor establecido por el usuario y envía cada grupo a los elementos de computación del dispositivo. Cada elemento de computación recibe un work-item.

El particionamiento del grupo global de trabajo puede hacerse en una sola dimensión, en dos o en tres dimensiones. Así por ejemplo, tomando el dispositvo de nombre "*Tahiti*" (una tarjeta gráfica ATI Radeon) listado justo encima, para procesar un vector lineal de 16384 elementos, puede particionarlo en grupos de 256 elementos, o items de trabajo, como máximo.

Por otra parte, si lo que se pretende es un trabajo de computación en una imagen 2D de hasta 1024×1024 píxeles (tamaño del grupo de trabajo global), tanto éste como el grupo de trabajo local será de dos dimensiones y en el código usaremos vectores de dos elementos (ver ejercicio 3). El valor del tamaño del grupo local en cada dimensión no puede ser mayor de 128 (seguimos usando el dispositivo *Tahiti* como ejemplo). Además, el número total de work-items del grupo local (el producto de los valores de los tamaños de los grupos locales en cada dimensión) no puede ser mayor de 256, así que se pueden usar varios tipos de particionamiento: 2×128 , 16×16 , 4×64 , etc. No todos los particionamientos proporcionarán el mismo rendimiento en el dispositivo. Depende del propio dispositivo y del problema a tratar.

El número de work-items en cada dimensión en el grupo local debe ser un divisor entero del tamaño del grupo global de trabajo en esa misma dimensión. Así, los particionamientos anteriores son posibles porque además de las restricciones anteriores, el valor de cada uno de ellos es divisor entero del valor de su misma dimensión en el grupo global (1024). También serían válidos particionamientos de 8×8, 1×32 o 2×64, ya que cumplen con las restricciones del párrafo anterior, y tanto el 2, el 8, el 32 y el 64 son divisores enteros de 1024. Sin embargo particionamientos como 3×16, 10×10, o 1×200 no son válidos porque aunque los dos primeros cumplen con los requisitos del párrafo anterior, los valores de alguna de sus dimensiones no son divisores enteros de 1024 (por ejemplo, 3 y 10). El tercero además no sería válido porque una de sus dimensiones (200) es mayor que el valor máximo admitido para esa dimensión (128).

El esquema de una aplicación OpenCL puede describirse en pseudocódigo de la siguiente forma:

1. Enumeración

```
Enumerar las plataformas disponibles en el host
Para cada plataforma, hacer:
  Enumerar los dispositivos disponibles en esa plataforma
  Para cada dispositivo, hacer:
    Si es el dispositivo que buscamos, terminar enumeración e ir al paso 2
  Fin para dispositivos
Fin para plataformas
```

2. Ejecución

```
Abrir una vía de comunicación desde el host al dispositivo
Compilar kérnel para ese dispositivo
Establecer los datos a enviar al kérnel
Ejecutar kérnel con un determinado tamaño de grupo
Recoger los resultados
Liberar todos los recursos usados
Cerrar via de comunicación
```

En los ejercicios propuestos veremos primero cómo realizar la enumeración de plataformas y dispositivos, y cómo interrogar al dispositivo sobre sus propias capacidades, necesarias para más tarde establecer los tamaños de grupo adecuados. Luego veremos cómo realizar los pasos necesarios para ejecutar un kérnel simple, y por último veremos otro ejemplo de kérnel, pero con dos dimensiones, y mediremos prestaciones.

Esta mini introducción a OpenCL se complementa con la lectura del capítulo 14 de *Programming Massively Parallel Processors. A hands-on approach, 2nd edition (David B. Kirk,Wen-mei W. Hwu, ed. Elsevier, 2013)* disponible en PDF como parte del material de esta práctica. Se recomienda su lectura tras haber leído esta misma introducción, el texto del boletín explicando los ejercicios 1 y 2 y el código fuente que se corresponde con estos dos ejercicios.

La documentación oficial de OpenCL corre a cargo del grupo Khronos, que es quien mantiene el estándar. En esta práctica trabajaremos con la versión 1.2 . https://www.khronos.org/opencl/

3. REALIZACIÓN DE LA PRÁCTICA.

EJERCICIO 1. Abre la solución **asd_px_opencl**. Dentro de ella elige el proyecto **ej1_enumeracion** para que sea el proyecto inicial y abre dentro de él, el fichero **enumeracion.c** .

OpenCL usa dos funciones para realizar la enumeración de plataformas y dispositivos. `clGetPlatformIDs` es la función usada para enumerar plataformas. Podeis ver en el código que se usa dos veces. La primera vez se usa para determinar cuántas plataformas hay, devolviendo este valor en la variable `platform_count`. La segunda vez la llamamos con el parámetro `platform_count` y la dirección de un vector, `platform_ids` donde se guardarán los identificadores de todas las plataformas que haya. Supondremos, por simplificar el código, que no habrá más de 16 plataformas.

La otra función es `clGetDeviceIDs`, que funciona de forma muy similar a la primera. Tiene dos parámetros extra: uno que indica a qué plataforma queremos interrogar sobre los dispositivos que contiene, y el otro para indicar qué tipo de dispositivos nos interesa obtener en la enumeración. En el código usamos `CL_DEVICE_TYPE_ALL` para referirnos a todos los dispositivos disponibles, pero también podemos usar el valor `CL_DEVICE_TYPE_GPU` para obtener sólo dispositivos basados en GPGPUs, `CL_DEVICE_TYPE_CPU` para obtener sólo dispositivos basados en CPUs, o `CL_DEVICE_TYPE_ACCELERATOR` para obtener dispositivos aceleradores basados en ASIC, DSP o FPGA.

Relacionadas con estas dos funciones, hay otras dos para obtener información relativa a una plataforma o un dispositivo. La función `clGetPlatformInfo` se usa para interrogar a una plataforma sobre sus características (como por ejemplo su nombre), y `clGetDeviceInfo` se usa con dispositivos para el mismo fin.

El trabajo en este primer ejercicio es completar el doble bucle de enumeración, para mostrar por pantalla todas las plataformas disponibles indicando para cada una su nombre, y dentro de cada plataforma, todos sus dispositivos, cada uno con su nombre (`CL_DEVICE_NAME`), el número de unidades de computación disponibles (`CL_DEVICE_MAX_COMPUTE_UNITS`), el tamaño máximo de un grupo local de trabajo (`CL_DEVICE_MAX_WORK_GROUP_SIZE`), y los tamaños máximos disponibles en cada una de las tres dimensiones (`CL_DEVICE_MAX_WORK_ITEM_SIZES`).

La información del manual de Khronos para `clGetDeviceInfo` está aquí: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clGetDeviceInfo.html . Para el valor del argumento `param_value_size` puede usarse el operador de C `sizeof` seguido del nombre de la variable.

Anota en la hoja de resultados los datos recogidos para este ejercicio 1. Nos harán falta más tarde en el ejercicio 3.

EJERCICIO 2. Ahora vamos a usar el proyecto **ej2_cifrado**, en la misma Solución. Actívalo y márcalo como proyecto de inicio. Abre dentro de él, el fichero **cifrado.c** .

En este ejercicio nos centraremos en la creación de un kérnel para desencriptar un texto contenido en una cadena de caracteres de C. Para cada carácter de la cadena, la descriptación consiste en restar 3 al código ASCII del carácter encriptado. El kérnel tomará tres parámetros: un puntero a la cadena fuente (`src`), un puntero a la cadena destino (`dst`) y el número de caracteres de la cadena (`lcadena`). Dentro del kérnel, `src[i]` o `dst[i]` se referiran al valor del i-ésimo carácter de la cadena fuente o destino. Debemos asegurarnos que `i` es un valor válido comprendido entre 0 y `lcadena-1`

OpenCL usa el lenguaje CL para codificar kérnels. Un kérnel es una función con una sintaxis muy parecida a la de una función en lenguaje C. El host que pretende usar el kérnel lo compila como parte de la ejecución de una aplicación compatible OpenCL. Es decir, que cuando compilamos el ejercicio con Visual Studio sólo compilamos el código C del host. El código del kérnel no es compilado hasta que durante la ejecución del programa en el host, éste llama a una determinada función que hace que OpenCL tome el código fuente del kérnel y lo compile.

Esto último tiene sentido en tanto y cuanto que no es hasta que el programa compatible OpenCL se ejecuta y realiza la enumeración, que se sabe qué dispositivos están disponibles. A diferencia de CUDA, OpenCL trabaja con dispositivos de distintos fabricantes, y por tanto incompatibles en código binario entre sí.

Una función kérnel sencilla tiene esta pinta:

```
kernel void nombre_funcion_kernel (global parametro_tipo vector_en_memoria_global[],
                                   parametro_en_memoria_privada, ...)
{
  size_t i = get_global_id(0);   // averiguar el número de work-item del grupo global de trabajo para la dimensión 0
  if (i hace referencia a un elemento válido)
  {
    procesar_elemento i
  }
}
```

Una función kérnel es una función `void`. Sólo puede devolver resultados a través de punteros. Los parámetros que por su tamaño no puedan estar en la memoria privada de los elementos de computación, o bien vayan a usarse para devolver datos desde todos los elementos de computación hacia el host, se definirán como `global` (en la práctica esto significa que cualquier vector, sobre todo si es grande, estará definido de esta forma, y en nuestro ejercicio en particular, los dos vectores `src` y `dst`). Los parámetros que no lleven el modificador `global` se asumirán privados de cada elemento

de computación. Normalmente serán valores escalares pasados como valor desde el host. En nuestro ejercicio, esto se aplica al parámetro `lcadena`.

El esquema del kérnel requerido por el ejercicio es:

```
kernel void cifrado (global char src[], global char dst[], int lcadena)
{
  size_t i = get_global_id(0);   // averiguar qué elemento de src debe procesar este work-item,
                                 // OJO porque i puede ser mayor que la longitud de la cadena

  // a partir de aquí, completar el código para este kérnel

}
```

Nótese el nombre de la función y el orden de los parámetros en la misma. El nombre de la función será usado más adelante, al generar el objeto kérnel con la función `clCreateKernel` . Podemos tener un fichero de código fuente CL con varias funciones kérnel, y usar sólamente una de ellas. Una función kérnel puede llamar a otras funciones internas (sin el modificador `kernel` en su cabecera) como haría normalmente en C. En cuanto al orden de los parámetros, afecta a la función `clSetKernelArg` que envía los parámetros del host al kérnel, y en donde tenemos que especificar el número de orden del parámetro que estamos enviando (de 0 en adelante, con 0 siendo el parámetro de más a la izquierda en la definición de la función kérnel)

El que la compilación del código fuente del kérnel se demore hasta la propia ejecución del programa en el host significa que tiene que haber alguna forma de enviar un texto con el código fuente de dicho kérnel a una función de OpenCL. Una de las formas de hacer esto es almacenando todo ese texto como una cadena de caracteres de C, que o bien es leida de un fichero externo (ver ejercicio 3) o bien está incluida como parte del código fuente del host, en una definición de cadena. Esto es lo que usaremos para este ejercicio.

El trabajo en este segundo ejercicio consiste en implementar una función kérnel en lenguaje CL, en donde cada work-item trabajará con un carácter de la cadena `src`, restándole 3 y guardando el nuevo valor en el elemento correspondiente de la cadena `dst`, ejecutar esta función kérnel y comprobar su correcto funcionamiento con todos los dispositivos disponibles en el host que estemos usando.

Hay que asegurarse de que el número de work-item asignado (el valor de *i*) esté dentro de los limites de la longitud de la cadena. Si no lo está, el work-item no debe hacer nada. Tal y como están definidos los grupos de trabajo para el ejercicio, se va a enviar un grupo global de 1024 work-items, en grupos locales de 64 work-items. Esto significa que nuestro programa podrá procesar cadenas de hasta 1024 caracteres. OJO, porque la cadena que ponemos como ejemplo tiene menos caracteres.

Una vez pensada la función, escríbela modificando la inicialización de la variable `program_code` en el ejercicio 2 para que contenga todo el código fuente de la misma. Esta es la declaración original del puntero a cadena `program_code` con el código a completar. Hay varias lineas en blanco, de sobra para implementar la función.

```
const char *program_code = ""
"kernel void cifrado (global char src[], global char dst[], int lcadena)\n"
"{                                                                       \n"
"  size_t i = get_global_id(0);                                          \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"                                                                        \n"
"}                                                                       \n"
;
```

Cuando compruebes que el programa funciona y desencripta correctamente el mensaje, copia en la hoja de resultados el código fuente del kérnel que has escrito.

EJERCICIO 3. Activa el proyecto **ej3_opencv_vs_opencl**, márcalo como proyecto de inicio y dentro de él, abre el fichero **opencv_vs_opencl.c** .

Este ejercicio se basa en los que hicimos en las prácticas 1 y 2. En esta ocasión, usamos OpenCV para aplicar un efecto blurring a una imagen cargada desde fichero. Realizamos la misma operación pero usando un kérnel bajo OpenCL. En ambos casos repetimos la operación muchas veces midiendo el tiempo que se tarda y nos quedamos con el valor más bajo, mostrándolo en pantalla.

El cambio más importante es que ahora trabajamos en dos dimensiones, con lo que el tamaño del grupo global de trabajo no es un valor, sino un vector con dos valores: número de work-items en una dimensión, y número de work-items en otra dimensión. Para este ejercicio, la primera dimensión se corresponderá a la coordenada Y, y la segunda dimensión a la X. Así, que `global_work_size[0]` valga 1024 significa que nuestro programa podrá procesar imágenes

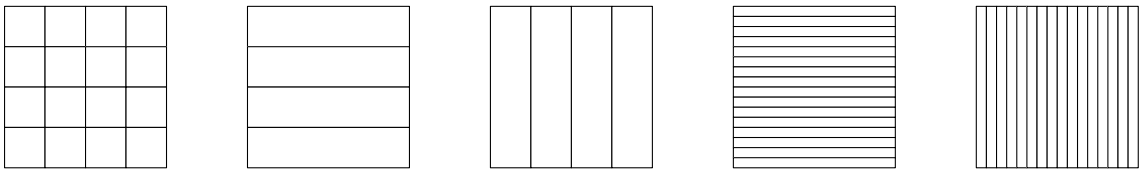
de hasta 1024 pixeles de alto. Por otra parte, `global_work_size[1]` delimita el número de píxeles de ancho que como máximo procesaremos (también 1024). Las imágenes de ejemplo que suministramos tienen diferentes tamaños, siendo precisamente la más grande de 1024×1024 píxeles.

Asimismo, el tamaño del grupo local de trabajo también es ahora un vector de dos dimensiones. Por defecto, se ha dejado a 1×1 píxeles, lo que significa que se envían a los elementos de computación 1×1 = 1 work-item cada vez. Por supuesto, esto no está aprovechando los múltiples elementos de computación que hay en el dispositivo. Pronto cambiaremos esto.

El programa comienza cargando una imagen (a elegir, descomentando la línea de código deseada). Esta imagen se envía a OpenCV para hacerle un blurring, como el de la práctica 1. La operación se repite una serie de veces, midiendo lo que tarda cada vez, y tomando el valor mínimo. A continuación se realiza la enumeración de plataformas y dispositivos OpenCL, y para cada uno de ellos, se envía la imagen para que se haga la misma operación de blurring. Como en el ejercicio 2, se crea la vía de comunicación entre host y dispositivo, se carga el código fuente del kérnel (en esta ocasión, desde un fichero separado, con extensión .cl), se compila, se crea el kérnel, se establecen los parámetros, y acto seguido se repite una serie de veces la medida del tiempo que se tarda en ejecutar el kérnel y recoger los resultados, tomando de nuevo el valor mínimo.

El trabajo con este ejercicio consiste en:

- Completar la función kérnel, abriendo el fichero *kernel_blur.cl* desde Visual Studio y escribiendo el resto del código siguiendo las indicaciones (aunque nada impide que escribamos la versión desenrollada que usamos en la práctica 2).
- Consultando las notas obtenidas durante la ejecución del ejercicio 1, llegar a un valor óptimo para los elementos del vector de dos elementos `local_work_size`, que define cuántos work-items se envían tanto en el eje Y como en el eje X. También se puede usar la función `clGetDeviceInfo` con `CL_DEVICE_MAX_WORK_GROUP_SIZE` para obtener de forma automática un valor adecuado del cual derivar los valores para `local_work_size`. ¿Qué es mejor? ¿Particionar la imagen en cuadros, en bandas horizontales o verticales, o por filas o columnas?



Cuadros:	Bandas horizontales o verticales:	Filas o columnas:
`local_work_size[0] ≈ local_work_size[1]`	`local_work_size[0] << local_work_size[1], ó local_work_size[0] >> local_work_size[1]`	`Una de los dos elementos de local_work_size = 1, y el otro = maximo_posible`

- Compilar, ejcutar y anotar en la hoja de resultados los valores pedidos para los dispositivos OpenCL del sistema, obteniendo la aceleración respecto de la versión OpenCV (tres decimales es suficiente).

Práctica OpenCL: HOJA DE RESULTADOS

Alumno/a: _____ Grupo: _____

EJERCICIO 1. Enumeración de dispositivos OpenCL

Nombre del dispositivo	Nº unidades de computación	Tamaño máximo de grupo de trabajo local	Número máximo de work-items en cada dimensión

EJERCICIO 2. Cifrado de un texto (proceso OpenCL en un vector 1D)

```
kernel void cifrado (global char src[], global char dst[], int lcadena)
{
  size_t i = get_global_id(0);









}
```

EJERCICIO 3. OpenCL versus OpenCV (proceso OpenCL en un vector 2D)

Imagen escogida: `spike` ☐ `carta_ajuste` ☐ `monsters_inc` ☐ `textura1024x1024` ☐

Nombre del dispositivo	Número de work-items en Y / X	Tiempo en ciclos/pixel	Aceleración respecto de OpenCV
OpenCV	**N/A**		**1**

Contestar: ¿con qué tipo de particionamiento para el número de work-items se consigue el mejor rendimiento de OpenCL respecto de OpenCV en imágenes grandes? Justificar.