Introducción a OpenCL

Leonardo Scandolo
leonardo@fceia.unr.edu.ar

Arquitectura del Computador*
Departamento de Ciencias de la Computación
FCEIA-UNR

*Actualizado 28 de noviembre de 2022 (D. Feroldi, feroldi@fceia.unr.edu.ar)

1. Una historia de GPUs (modernos)

Un GPU (Graphics Processing Unit) moderno es un *coprocesador* que se encuentra en la mayoría de las computadoras de uso personal. Los primeros ejemplos de este tipo de coprocesadores se remontan a computadoras de los 70's y 80's, y podían tener diferentes usos (dibujar sprites, hostear la memoria de video, etc.) Sin embargo, a partir de los 90's, los GPU's empezaron a confluir en la misma función: acelerar el proceso de un tipo de algoritmos gráficos llamados algoritmos de *rasterización*. Los algoritmos de rasterización toman como entrada formas geométricas (triángulos por lo general), y los modifican a través de multiplicaciones con matrices que representan proyecciones y roto-traslaciones para finalmente mostrarlos en pantalla de manera que parezcan ser parte de una escena en 3 dimensiones.

El proceso de cada forma geométrica se daba por separado, siguiendo una arquitectura de *pipeline*. Las etapas de dicho pipeline eran originalmente fijas y algunas eran parametrizables, de manera de poder elegir por ejemplo colores, matrices de proyecciones, tamaños, etc. Dado que cada forma geométrica era procesada independientemente de las demás, se generó una arquitectura paralela con muchos procesadores dedicados a las mismas tareas, y que podían procesar muchas formas al mismo tiempo. La Figura 1 muestra un pipeline sencillo.

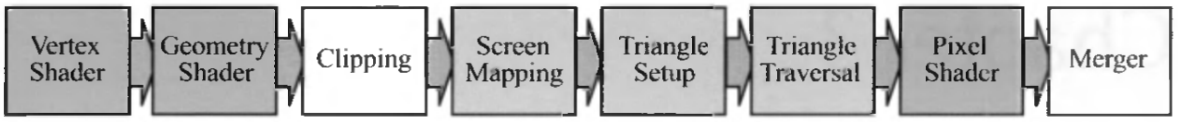


Figura 1: Pipeline simple para un GPU

Con el tiempo, se le fueron agregando más parametros a las etapas fijas, pero fue evidente que este tipo de arquitectura era muy limitada. Por lo tanto, algunas etapas (más notoriamente la etapa de *vertex shading* y *pixel o fragment shading*) se convirtieron en etapas programables. Esto quiere decir que se definió un lenguaje (lenguaje de shaders) con el cual se puede definir (dentro de ciertas restricciones) el comportamiento de las etapas programables. De esta manera, fue necesario que los procesadores de los GPUs fueran de propósito cada vez más general. A mediados de la década del 2000, los procesadores gráficos empezaron a ser construidos con cientos de procesadores de propósito general.

2. Introducción

Cuando los procesadores gráficos empezaron a poder ser programables, algunos investigadores notaron que algunos algoritmos de origen no gráfico podían implementarse en los shaders de un GPU, y la imágen final generada contendría la solución al problema que se intentaba resolver. Dado que los GPUs poseían cientos de procesadores (a una frecuencia mucho menor que una CPU), los algoritmos implementados de esta manera eran muchas veces más rápidos que la implementación en CPU. Las compañías productoras de GPU (notablemente NVIDIA y ATI) se dieron cuenta del mercado para usar GPUs como procesadores generales para resolver problemas no gráficos. A partir de esto,

NVIDIA crea CUDA, que es un lenguaje que permite interactuar con la memoria de sus GPUs, y escribir programas que serán ejecutados dentro del mismo. Para estandarizar este tipo de cálculos, el mismo ente que lleva adelante el estándar libre OpenGL, crea OpenCL (Open Computing Language), para proveer un framework estándar que puedan implementar todos los fabricantes de GPUs y CPUs, y que permite definir algoritmos paralelos para correr en los procesadores que provean soporte al estándar.

3. Arquitectura de un dispositivo OpenCL

3.1. Dispositivos OpenCL

Para estandarizar los diferentes tipos de procesadores que pueden implementar el estándar OpenCL, se definen *dispositivos OpenCL*, que es una máquina abstracta cuyas características son parametrizadas dependiendo del dispositivo real subyacente. Estos dispositivos son programables a través de una librería estándar definida por OpenCL. Esta librería está definida en forma de headers y funcionalidades, y cada fabricante provee su propia implementación. De esta manera es posible leer y escribir la memoria de un dispositivo, y enviar cálculos para que se ejecuten en el mismo. Dichos cálculos están definidos en un lenguaje especial llamado *OpenCL C*, que es muy parecido al lenguaje C.

Cada *dispositivo* expone una cantidad de *unidades de cómputo*, o *compute units*. Estas *unidades de cómputo* son grupos de uno o más procesadores. Los procesadores que forman parte de una *unidad de cómputo* son llamados *elementos de procesamiento* o *processing elements*. Todos los *elementos de procesamiento* dentro de una misma *unidad de cómputo* comparten recursos, como memoria y cache. Al momento de ejecutar instrucciones se comportan como una línea larga SIMD[1]. Esto significa que todos los *elementos de procesamiento* de una misma *unidad de cómputo* ejecutan las mismas instrucciones, aunque posiblemente leyendo y escribiendo a lugares diferentes de la memoria. En el caso de expresiones condicionales (`IF-THEN-ELSE`) simplemente se desactivan los *elementos de procesamiento* correspondientes durante la ejecución de la rama condicional que no deben ejecutar. La Figura 2 muestra un esquema de la arquitectura descripta en el párrafo anterior.

Tanto los cómputos que se quieren realizar en un dispositivo OpenCL, como las lecturas o escrituras a memoria del dispositivo se deben encolar en una *cola de comandos* del dispositivo. Normalmente no hay un orden estricto en el orden de ejecución de los comandos de una cola de comandos. Sin embargo, hay varias formas de asegurar un orden. Una forma es llamando a una función llamada *clFinish* que espera hasta que todos los comandos de la cola hayan sido terminados. Otra forma es encolar una barrera en la cola, lo cual asegura que todos los comandos encolados desde ese punto serán ejecutados sólo luego que los comandos existentes en la cola sean ejecutados. OpenCL también define *eventos* que kernels pueden lanzar para que se ejecuten otros kernels que esperan esos eventos.

[1] *Single Instruction Multiple Data*

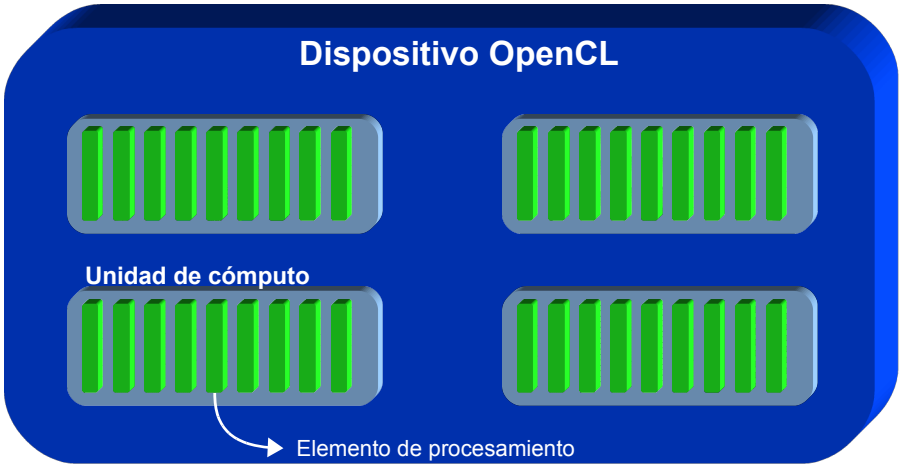


Figura 2: Diagrama de la arquitectura básica expuesta por OpenCL

3.2. Jerarquía de memoria

OpenCL también asume una jerarquía de memoria que está ligada a la forma en que se organizan los procesadores de un dispositivo. Cada *elemento de procesamiento* tiene acceso a su propia memoria privada, que consiste generalmente de registros. A su vez, los *elementos de procesamiento* de una misma *unidad de cómputo* tienen acceso a una memoria compartida que todos los elementos pueden leer y escribir, pero que no es accesible para los elementos de otras *unidades de cómputo*. Finalmente existe una memoria global del *dispositivo* que es accesible a todos los *elementos de procesamiento*.

Como es de esperar cada jerarquía de memoria tiene tamaños y tiempos de acceso diferentes, por lo cual, en la mayoría de los casos, el tiempo de acceso a registros de un *elemento de procesamiento* es menor al de la memoria compartida de una *unidad de cómputo*, que es a la vez menor al de la memoria global de un *dispositivo*. El caso contrario se da con el tamaño de cada jerarquía de memoria.

Cada *dispositivo* posee la capacidad de transferir datos desde y hacia la memoria principal de la computadora donde se encuentra. Los tiempos de transferencia en este caso son por lo general altos. La Figura 3 ejemplifica la jerarquía de memoria que expone OpenCL.

3.3. Kernels de OpenCL

Los algoritmos ejecutados en OpenCL son llamados *kernels*. Éstos son escritos en un lenguaje de programación propio de OpenCL, muy parecido al lenguaje C. Dichos algoritmos serán ejecutados por los *elementos de procesamiento* de un *dispositivo*. Al momento de ejecutar un *kernel* en un *dispositivo* es posible definir cuantas instancias de ese cómputo se ejecutarán. De esta manera un mismo cómputo puede ser ejecutado una gran cantidad de veces con una sola llamada a la biblioteca OpenCL mediante la capacidad de cómputos paralelos de los dispositivos que implementan el estándar OpenCL.

Cada instancia a ejecutar de un *kernel* es llamada un *work item*. Cuando se define la cantidad de *work items* a ejecutar, también es necesario definir cómo serán agrupadas

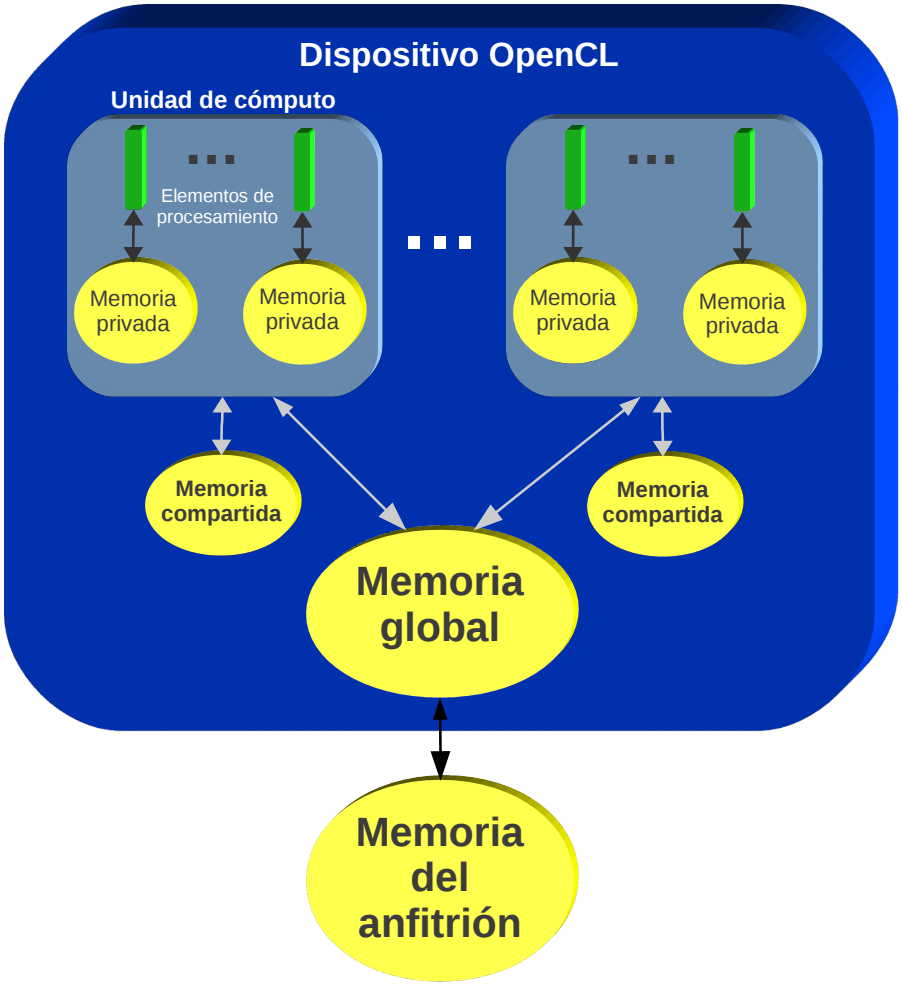


Figura 3: Diagrama de la jerarquía de memoria expuesta por OpenCL

esas instancias. Los *work item* se agrupan porque cada instancia será ejecutada por un *elemento de procesamiento*, y cada grupo será ejecutado por una *unidad de cómputo*. Cada grupo de *work items* es llamado un *work group*. Los *work items* de un mismo *work group* tienen acceso a memoria compartida dado a que se ejecutan en la misma *unidad de cómputo*. Los *work items* de diferentes *work groups* sólo pueden comunicarse a través de la memoria global.

Para poder ejecutar código sobre diferentes datos, cada *work item* tiene acceso a un identificador global que lo diferencia de todos los otros *work items* y a un identificador local que lo identifica dentro de su *work group*.

La Figura 4 ejemplifica la manera en que un *kernel* puede modificar datos distintos mientras ejecuta las mismas instrucciones. El *kernel* contiene sólo una instrucción que establece el valor de una posición de un arreglo. Tanto la posición como el valor son función del identificador de cada *work item* que se ejecute.

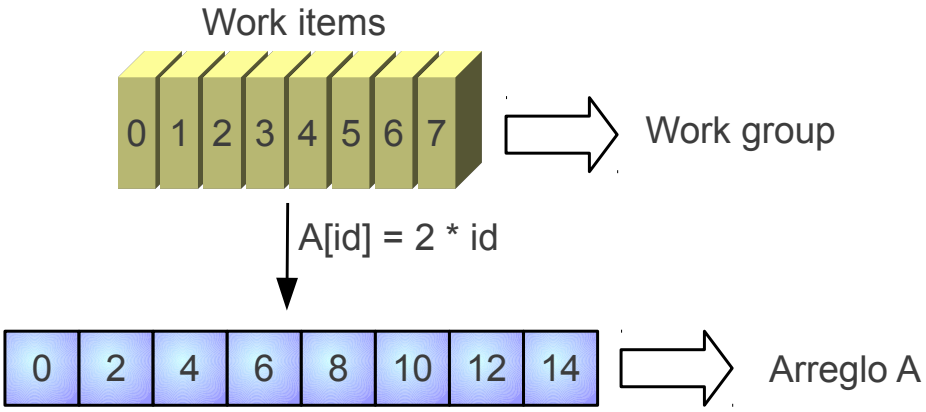
La cantidad de *work items* en un mismo *work group* está limitada por la cantidad de *elementos de procesamientos* por *unidad de cómputo* en el dispositivo OpenCL. Sin embargo, la cantidad total de *work items* no está acotada, dado que en caso de haber más *work groups* que *unidades de cómputo*, simplemente se ejecutarán primero tantos como

```
__kernel void fill_array(global int* A) {

    size_t id = get_global_id(0);

    A[id] = 2 * id;
}
```

(a) Código de un *kernel* de OpenCL que modifica los valores de un arreglo global.



(b) Esquema del código ejecutado por el *kernel*.

Figura 4: Ejemplo de un cómputo para modificar un arreglo usando un *kernel* de OpenCL

unidades haya disponible, y luego que se completen sus cálculos, se continuará con los siguientes, hasta culminar la ejecución de todos los *work groups*. Se desprende de esta lógica de ejecución que existe una expectativa de concurrencia para la ejecución de los *work items* de un mismo *work group*, pero no para *work items* de diferentes *work groups*.

Una característica importante de la definición de instancias de un *kernel* a ejecutar es que pueden definirse grupos multidimensionales, es decir que el identificador de cada *work item* y cada *work group* puede tener generalmente hasta 3 dimensiones. Esto facilita la división de trabajo en una gran cantidad de algoritmos paralelos.

3.4. Limitaciones de OpenCL

Las propiedades del modelo de cómputo de OpenCL afectan el diseño de la solución que se propone. Por lo tanto a continuación se discutirán algunas limitaciones y factores que afectan la eficiencia de los programas que utilizan OpenCL.

Un factor importante que afecta la eficiencia de OpenCL es que la ejecución de un *kernel* es ineficiente cuando diferentes *work items* de un mismo *work group* ejecuten ramas distintas de una expresión condicional. Esto es debido al modelo de cómputo expuesto basado en SIMD. Cuando un *kernel* no sufre este problema, se dice que los datos que está procesando son *coherentes*. Por lo tanto es siempre mejor agrupar los datos a procesar de manera que sean lo más coherentes posibles. A pesar de que no es una limitación que no permita hacer un cálculo, el modelo de OpenCL fue creado con aplicaciones de alto desempeño como objetivo, por lo cual si no es posible obtener un buen rendimiento es

preferible hacer cómputos en unidades de procesos tradicionales, como CPUs.

El lenguaje de definición de *kernels* en OpenCL es un lenguaje muy parecido al lenguaje C. Sin embargo presenta dos limitaciones muy importantes: la falta de recursión y la falta de memoria dinámica.

La falta de recursión implica que cualquier algoritmo usado en OpenCL debe ser completamente iterativo. Esta es sólo una limitación sobre la facilidad de escribir código, dado que cualquier cómputo recursivo (que termine) puede escribirse de forma iterativa.

La segunda limitación es más problemática, dado que implica que debemos establecer la cantidad de memoria que un *kernel* va a utilizar antes de comenzar su ejecución.

Otra limitación importante del modelo es la separación entre memoria del dispositivo que implementa OpenCL y la memoria convencional del sistema. La memoria del dispositivo puede ser accedida rápidamente durante la ejecución de un *kernel* en OpenCL. Sin embargo escribir o leer la memoria del dispositivo desde fuera del mismo puede ser lento, y por lo tanto se debe minimizar en lo posible estas operaciones.

Un último inconveniente resultante del uso de arquitecturas masivamente paralelas es que dado que están diseñadas para hacer cálculos utilizando muchos procesadores, la subutilización de los mismos decrementa la velocidad en la cual se ejecuta un algoritmo. En particular, dado que cada *work group* es asignado a una *unidad de cómputo* diferente, es siempre más eficiente crear *work groups* que utilicen todos los recursos de la *unidad de cómputo* donde se ejecutarán.

Resumiendo, las limitaciones más importantes del modelo OpenCL son:

- Necesidad de tener coherencia de datos.
- Falta de recursión.
- Falta de memoria dinámica.
- Lentitud de traspaso de datos entre sistema y dispositivo OpenCL.
- Necesidad de utilizar eficientemente los *elementos de procesamiento* disponibles.

4. Ejemplo de un programa en OpenCL

4.1. Inicialización de un dispositivo OpenCL

```
#include <stdlib.h>
#include <stdio.h>
#include <CL/opencl.h>

int main(int argc, char** argv)
{
        cl_int errNum;

//////////////////////////////
// Crear un contexto de OpenCL usando la primer plataforma disponible
```

```
//////////////////////////////
        cl_context context = 0;
        cl_platform_id firstPlatformId;
        cl_uint numPlatforms;
        errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
        if (errNum != CL_SUCCESS || numPlatforms <= 0)
        {
                puts("No se encontraron plataformas OpenCL.\n");
                return 1;
        }
        cl_context_properties contextProperties[] =
                {CL_CONTEXT_PLATFORM,
                 (cl_context_properties)firstPlatformId,
                 0
                };
        context = clCreateContextFromType(contextProperties,
                                          CL_DEVICE_TYPE_GPU,
                                          NULL, NULL, &errNum);
        if (errNum != CL_SUCCESS)
        {
                puts("No se pudo crear un contexto en GPU\n");
                return 1;
        }

//////////////////////////////
// Crear una cola de comandos en el primer dispositivo disponible,
// en el contexto creado
//////////////////////////////
        cl_command_queue commandQueue = 0;
        cl_device_id *devices;
        cl_device_id device = 0;
        size_t deviceBufferSize;
        errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL,
                                  &deviceBufferSize);
        if (errNum != CL_SUCCESS)
        {
                puts("No se pudo obtener deviceBufferSize\n");
                return 1;
        }
        devices = (cl_device_id*)malloc(deviceBufferSize);
        errNum = clGetContextInfo(context, CL_CONTEXT_DEVICES,
                                  deviceBufferSize, devices, NULL);
        if (errNum != CL_SUCCESS)
        {
                puts( "No se pudieron obtener los identificadores de los dispositivos
                return 1;
```

```
        }
        device = devices[0];
        commandQueue = clCreateCommandQueue(context,
                                            device, 0, NULL);
        if (commandQueue == NULL)
        {
                puts( "No se pudo crear la cola de comandos\n");
                return 1;
        }
        free(devices);

...
```

4.2. Un kernel de OpenCL sencillo

```
kernel matrix_multiplication(const int Mdim,
                             const int Ndim,
                             const int Pdim,
                             global float* A,
                             global float* B,
                             global float* C)
{
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp;

    if( (i < Ndim) && (j <Mdim)) {
        tmp = 0.0;

        for(k=0;k<Pdim;k++)
            tmp += A[i*Ndim+k] * B[k*Pdim+j];
    C[i*Ndim+j] = tmp;
  }
}


};
```

4.3. Crear un kernel OpenCL y encolarlo

```
// Asumiendo que tenemos
```

```
// #define ARRAY_SIZE 32
// y que "extern char* program_source" es la declaración del string
// que contiene el kernel

...
//////////////////////////////
// Crear un programa OpenCL desde el código fuente en program_source
//////////////////////////////
        cl_program program = 0;
        program = clCreateProgramWithSource(context, 1,
                                            (const char**)&program_source,
                                            NULL,&errNum);
        if (errNum != CL_SUCCESS)
        {
                puts("No se pudo crear el programa.\n");
                return 1;
        }
        errNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
        if (errNum != CL_SUCCESS)
        {
                puts("No se pudo construir el programa.\n");
                return 1;
        }


//////////////////////////////
// Crear el kernel de OpenCL
//////////////////////////////
        cl_kernel kernel = 0;
        kernel = clCreateKernel(program, "hello_kernel", NULL);
        if (kernel == NULL)
        {
                puts("No se pudo crear el kernel\n");
                return 1;
        }
//////////////////////////////
// Crear los objetos de memoria que van a ser usados como
// argumentos del kernel. Primero se crean los arreglos de memoria que
// van a ser usados para guardar los argumentos del kernel.
//////////////////////////////
        cl_mem memObjects[3] = { 0, 0, 0 };
        float result[ARRAY_SIZE];
        float a[ARRAY_SIZE];
        float b[ARRAY_SIZE];
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
```

```
                a[i] = i;
                b[i] = i * 2;
        }
        memObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
                                       CL_MEM_COPY_HOST_PTR,
                                       sizeof(float) * ARRAY_SIZE, a,
                                       NULL);
        memObjects[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
                                       CL_MEM_COPY_HOST_PTR,
                                       sizeof(float) * ARRAY_SIZE, b,
                                       NULL);
        memObjects[2] = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                       sizeof(float) * ARRAY_SIZE,
                                       NULL, NULL);
////////////////////////////
// Establecer los argumentos del kernel (result, a y b)
////////////////////////////
        errNum = clSetKernelArg(kernel, 0,
                                sizeof(cl_mem), &memObjects[0]);
        errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem),
                                 &memObjects[1]);
        errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem),
                                 &memObjects[2]);
        if (errNum != CL_SUCCESS)
        {
                puts("No se pudieron establecer los argumentos del kernel.\n");
                return 1;
        }
////////////////////////////
// Encolar el kernel para su ejecución
////////////////////////////
        size_t globalWorkSize[1] = { ARRAY_SIZE };
        size_t localWorkSize[1] = { 1 };
        errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                                        globalWorkSize, localWorkSize,
                                        0, NULL, NULL);
        if (errNum != CL_SUCCESS)
        {
                puts("No se pudo encolar el kernel para ejecutar.\n");
                return 1;
        }
////////////////////////////
// Copiar el resultado a un buffer en el anfitrión
////////////////////////////
        errNum = clEnqueueReadBuffer(commandQueue, memObjects[2],
                                     CL_TRUE, 0, ARRAY_SIZE * sizeof(float),
```

```
                                        result, 0, NULL, NULL);

        if (errNum != CL_SUCCESS)
        {
                puts("Error al leer el buffer de resultado.\n");
                return 1;
        }
////////////////////////////
// Imprimir el resultado
////////////////////////////
        for (int i = 0; i < ARRAY_SIZE; i++)
        {
                printf("%f ",result[i]);
                /* cout << result[i] << " "; */
        }
        puts("\nPrograma ejecutado exitosamente.\n");

...
```

5. Bibliografía

- A. Munshi et al, *OpenCL Programming Guide*, 2012.
- *OpenCL 1.1 specification*, Khronos Group, disponible en: *www.khronos.org/registry/cl/specs/opencl-1.1.pdf*.
- *OpenCL C specification*, Khronos Group, disponible en: *www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf*.