OpenCL Overview, Implementation, and Performance Comparison

Juan A. Fraire
Universidad Nacional de Córdoba
Córdoba, Argentina
juanfraire@gmail.com

Pablo Ferreyra
Universidad Nacional de Córdoba
Córdoba, Argentina
ferreyra@famaf.unc.edu.ar

Carlos Marques
Universidad Nacional de Córdoba
Córdoba, Argentina
marques@famaf.unc.edu.ar

***Resumen*—High performance parallel computing was something exclusive for expensive specialized hardware some years ago. But now we can find powerful parallel processors in many home graphics card whose interface has been recently opened by many manufacturers for general purpose computing. OpenCL, created by the world most important processors manufacturers, went a little further, aiming for a platform and manufacturer independent parallel language. However, understanding this new processing paradigm is challenging and critical for future computation demanding applications. The first approach of this document is to provide a deep technical background of OpenCL architecture. Second, we propose an implementation of a matrix product calculation OpenCL kernel directly implemented in C++ without wrappers so as to describe in detail the OpenCL programming flow. Thirdly, different platforms and algebraic scenarios are created for this program concluding that the improvement of calculation performance can reach up to 3 orders of magnitude over the same algorithm in plain C++.**

I. Introducción

Tiempo atrás el CPU (Central Processing Unit) era capaz de lidiar con las demandas de procesamiento por si misma. Sin embargo, actualmente, diferentes elementos con capacidad de procesamiento de naturalezas muy diferentes coexisten e interactúan en cualquier sistema de cómputo moderno. Además, los procesadores modernos han migrado hacia el paralelismo como un cambio importante hacia el incremento de performance. Inclusive las CPUs hoy proponen exclusivamente arquitecturas de múltiples núcleos. GPUs (Graphic Processing Units) han evolucionado similarmente de una estructura de renderización fija (fixed pipeline) hacia múltiples cores programables de alta performance, dejando atrás su función única de procesamiento gráfico. Por ejemplo, la GPU de una ATI Radeon 5870 cuenta con 1600 elementos de procesamiento que pueden trabajar de forma paralela para prácticamente cualquier tipo de cálculo. La programación de estas poderosas unidades permitió abrir las puertas a su utilización para cálculos de propósitos generales (General Purpose Computing on GPU). Actualmente CUDA y CTM/CAM son dos ejemplos de plataformas dependientes del fabricante disponibles para GPGPU proporcionadas por Nvidia y ATI respectivamente.

Este escenario cae bajo la categoría de sistemas heterogéneos: la coexistencia de múltiples elementos de procesamiento en una misma plataforma, de características mas o menos poderosas y de naturaleza y arquitecturas potencialmente diferentes. Esta diferencia entre dispositivos deriva en modelos de programación igualmente desiguales. Los modelos de programación para CPU suponen espacios de memoria compartidos y -salvo ciertas excepciones- no incluyen operaciones de vectores. Por otro lado, GPGPU se caracteriza con un esquemas de jerarquías de memorias mas complejos -explícitos- y gran capacidad de cálculo vectorial, pero son dependientes del fabricante del dispositivo. Este escenario hace difícil, si no imposible, que el desarrollador aproveche completamente la potencia de procesamiento disponible en los sistemas modernos de computación. Esta, es la razón principal que impulsó a Intel, AMD, Apple, IBM, Nvidia, entre otros, a unirse como parte del Khronos Group, un consorcio independiente con el objetivo de crear OpenCL, el Open Computing Language.

OpenCL es un estandard abierto a la industria para la programación de colecciones heterogéneas de CPUs, GPUs, y otros dispositivos discretos de cálculo disponibles en una plataforma dada. Esto abarca desde dispositivos hand-held como celulares o palms (que ya disponen de procesadores de múltiples cores), PCs hogareñas, hasta servidores mainframe. OpenCL permite al desarrollador asignar tareas especificas de un programa al elemento de procesamiento mas apropiado de la plataforma sin necesidad de cambiar el lenguaje utilizado en el código. Por otro lado OpenCL fue diseñado para explotar al máximo el paralelismo dentro de un mismo dispositivo -como el GPU- así como dentro de una plataforma, permitiendo la ejecución de tareas en múltiples núcleos de múltiples dispositivos. Por ejemplo, la porción de código de un programa que es fácilmente paralelizable, se puede enviar a la GPU, mientras se le asigna el código de naturaleza mas serial y especifica al CPU. OpenCL es más que un lenguaje, es un marco de

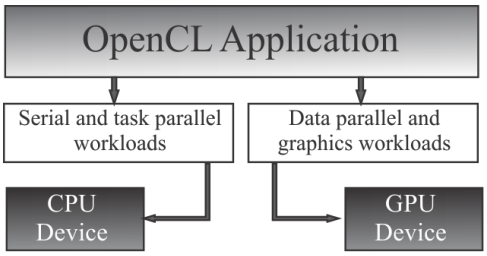

Figura 1. Interacción de dispositivos OpenCL

trabajo para programación en paralelo independiente de la plataforma y del fabricante. El objetivo principal de OpenCL es lograr escribir código eficiente y portable simultáneamente, rompiendo el eterno compromiso de estas dos características.

De esta manera, cada dispositivo compatible con OpenCL cuenta con su propia implementación especifica, que, en tiempo de ejecución, interpreta el lenguaje OpenCL del programa que le es requerido ejecutar. Así, es responsabilidad de cada fabricante crear drivers de su propio producto que le saquen la máxima capacidad de procesamiento para una instrucción OpenCL dada. Estos drivers están siendo desarrollados y perfeccionados al momento de la escritura de este trabajo para IBM Cell Processors (el procesador de PlayStation3), ATI GPUs, Nvidia GPUs, ATI CPUs, Intel CPUs, y un importante número de DSPs. Inclusive es conocido que Altera y Xilinx están desarrollando mappers de código OpenCL a lenguajes de descripción de hardware para su implementación en FPGAs (Field Programmable Gate Arrays).

El presente trabajo se estructura en tres grupos de secciones. Las secciones II a IV proporcionan el contenido teórico principal describiendo la arquitectura haciendo uso de diferentes modelos y enfoques. En la sección VII se propone como caso de estudio un kernel de producto de matrices con el fin de comparar su performance de ejecución en una CPU y una GPU, logrando demostrar mejoras en performance de 2 a 3 órdenes de magnitud. Luego, en la sección VIII se analizan y discuten los resultados obtenidos.

II. Modelo de Plataforma

El estandard propuesto por el Khronos Group analiza la arquitectura OpenCL por medio de modelos. Uno de ellos es el modelo de la plataforma, otros son el modelo de ejecución, de memoria, y de programación.

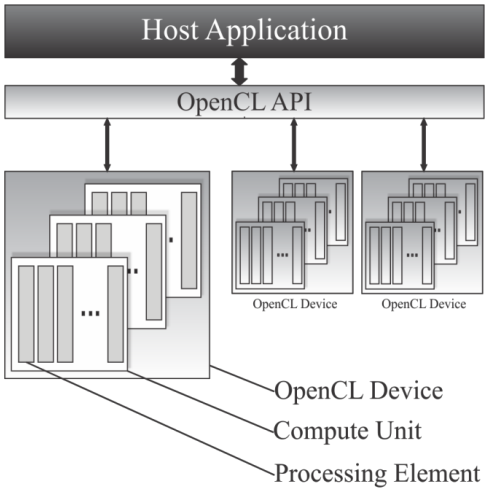

Figura 2. Modelo de Plataforma OpenCL

El modelo de plataforma define un unidad de host conectada a uno o mas dispositivos compatibles con OpenCL. El rol de host puede ser adjudicado a una CPU corriendo un sistema operativo con soporte OpenCL. El host básicamente ejecuta una aplicación que se encarga de gestionar comandos e instrucciones para enviarlos a la implementación de OpenCL API de un dispositivo OpenCL dado. A su vez, esta API se encarga de manejar su dispositivo en función de las órdenes recibidas. Cualquier GPU, CPU, o DSP puede ser utilizado como dispositivos OpenCL. Como se define en la especificación OpenCL, un dispositivo OpenCL consiste en una colección de uno o mas unidades de cómputo, que son, a su vez, subdivididas en elementos de procesamiento. Éstos últimos pueden ejecutar SIMP (single instrucción múltiple data) -típicamente una GPU- o SPMD (single program múltiple data) -típicamente una CPU-.

III. Modelo de Ejecución

El modelo de ejecución consta de dos componentes: Kernels (similares a una función en C que se ejecuta en uno o mas dispositivos OpenCL), y un programa host (ejecuta el programa central que gestiona los demás dispositivos).

El programa host puede ser escrito en cualquier lenguaje que soporte el sistema. C y C++ son ampliamente utilizados para acceder APIs actualmente disponibles en Linux, MacOS y Windows. Numerosos ”wrappers”(C#, .NET, Python, entre otros) han salido a luz en el último tiempo proporcionando una interfaces intermedia -muchas veces mas simple- para acceder a las funcionalidades OpenCL. Estos últimos proporcionan un medio mas intuitivo y amigable, y, en algunos casos, la posibilidad de una interface gráfica para el programa. Sin embargo, dado el enfoque simplista de los ”wrappers”suelen esconder capacidades de la plataforma. Ejemplo de estos son: Cloo, OpenCLTemplate, OpenCL.NET, OpenTK, PyOpenCL, entre otros.

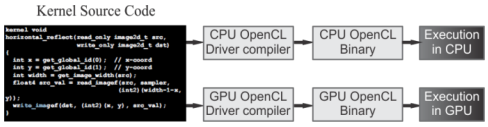

Figura 3. Compilación en tiempo de ejecución

Kernels, por otro lado, son porciones de código similares una función en C escrita en lenguaje especificado por OpenCL. Los Kernels son usualmente declarados en el programa host como una literal constante o leídos desde un archivo (la extensión .cl está siendo utilizada para estos). Luego, el programa host le pide a la API OpenCL de un dispositivo dado que compile dicho kernel en tiempo de ejecución haciendo uso de las llamadas definidas en la especificación OpenCL. La implementación OpenCL en tiempo de ejecución (proporcionada por cada fabricante) compila y construye el código máquina para su dispositivo. La compilación es dependiente del dispositivo, pero como esta es realizada en tiempo de ejecución, el código se vuelve completamente portable. La figura 3 ilustra este procedimiento. De esta manera, la performance se mantiene óptima dado que cada fabricante puede crear compiladores y linkeadores cuyos binarios resultantes utilicen la mayor capacidad de procesamiento de su propio producto. En caso de necesidad, el binario de este proceso puede ser guardado

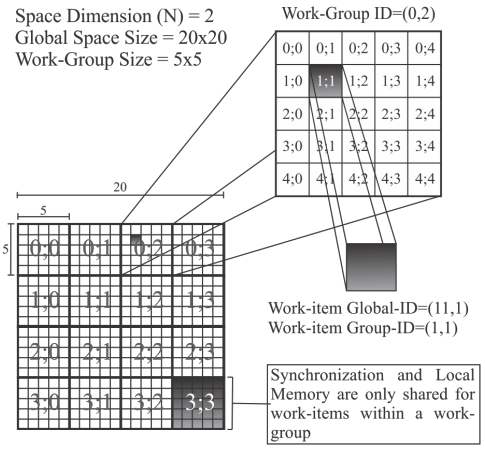

Figura 4. N=2 y 20x20 Kernels

por el programa host para en el futuro mejorar la performance evitando la compilación. Además de gestionar la compilación y ejecución de los Kernels, el programa host maneja lecturas y escritura de los dispositivos. Básicamente, es el proceso de intercambiar datos desde la RAM del host desde y hacia la memoria del dispositivo (en caso de una GPU sería la RAM de Video). En estas porciones de memoria residen las variables que el kernel a ejecutar tomará como entrada para su tarea en la ejecución. El modelo de memoria se describe en la sección IV. Las siguientes sub-secciones describen los kernels y programas host en mayor detalle:

III-A. Propiedades del Kernel

Los Kernels son puestos en cola de ejecución por la aplicación de host. OpenCL explota el paralelismo computacional de datos dentro de un dispositivo dado definiendo el problema en un espacio indexado de N dimensiones, donde N=1, 2 o 3. Este espacio es definido en el momento en que el Kernel es puesto en cola de ejecución. La decisión del número de dimensiones es crucial para el desarrollo exitoso del algoritmo a ejecutar. Por ejemplo, si lo que ejecuta el programa es un algoritmo de procesamiento de imagen, lo mas probable es que la dimensión mas apropiada sea N=2, mientras que para el análisis de un volumen N sea tentativamente 3. Este espacio virtual contiene todos los elementos físicos -dentro de un dispositivo dado- que ejecutaran de forma paralela el kernel en cola. Cada elemento de procesamiento recibe el nombre de "work-item", y puede ser visto como un punto en este espacio N-dimensional. Cada punto tiene, de esta manera, un único identificador global (Global ID) que puede ser expresado como coordenadas de estas dimensiones, es decir: (x), (x,y), o (x,y,z) para N=1, 2, o 3 respectivamente.

Cada work-item ejecuta el mismo código, pero el recorrido especifico del programa en cada uno de ello, así como las variables sobre la que trabaje pueden variar. Por ejemplo, el código en ejecución puede hacerse auto-consciente de su ID global -por medio de funciones definidas en la especificación como get_global_id()-. Dado que este ID es único por work-item en el espacio de dimensión N, puede ser utilizado para apuntar a una área especifica de la memoria. Por ejemplo, es muy común ver su utilización como valor de des-referencia de un arreglo de la forma .ªrray_name[get_global_id(0)]"que será sólo accedido por un work-item. La función get_global_id() es ampliamente utilizada en los kernels OpenCL como índice de des-referencia, permitiendo tener un gran número de "trabajadores"(work-items) operando en paralelo en diferentes zonas de un arreglo. Realizar tareas de este tipo en forma serial probablemente involucraría un ciclo "whileçon tantas iteraciones como la longitud del arreglo. Esto evidencia el potencial de mejora que se puede obtener por medio de GPGPU donde, por ejemplo, 1600 elementos de procesamiento (ATI Radeon 5870) pueden ejecutar en un ciclo lo que un "while"haría en 1600. El incremento de la velocidad es inmenso, pero solo verdadero en casos específicos donde el algoritmo es fácilmente paralelizable. Fuera de esta especificidad, las CPUs tradicionales muestran mejor comportamiento. Por ejemplo: la búsqueda del mayor valor de un arreglo seguramente se ejecutará mejor en una CPU dado que el arreglo debe ser recorrido de manera lineal. Se puede apreciar entonces el potencial de OpenCL de asignar diferentes partes de un programa al dispositivo mas capaz de ejecutarlas en función de su naturaleza serial o paralela.

OpenCL permite agrupar dichos work-items dentro de grupos o work-groups. Estos work-groups se definen como subespacios del espacio N-dimecional, y se le es asignado un ID de grupo de la misma dimensión que N. De esta manera, cada work-item, además de tener su ID global, también tiene un ID de grupo compartido con los demás elementos de su grupo. OpenCL asegura que todos los work-items dentro de un work-group sean ejecutados concurrentemente en una unidad de cómputo dado con el fin de compartir memoria y sincronización. Un ejemplo de ilustrativo de un espacio bidimensional de tamaño 20x20 con grupos de 5x5 se muestra en la figura 4. Si no se define en las instrucciones de plataforma OpenCL, la gestión de grupos se realiza automáticamente por la API. Los work-groups juegan un papel importante en la sincronización ya que esta, en OpenCL, queda solo definida dentro de un grupo. El sincronismo es fundamental en el modelo de programación concurrente propuesto por OpenCL ya que este no exige que los kernels sean ejecutados exactamente al mismo tiempo. Esto quiere decir que puede haber kernels que se ejecuten antes que otros. Entonces se puede dar el caso donde, por ejemplo, primero se tenga que escribir un arreglo en .ªrray_name[get_global_id(0)]z luego se tenga que leer .ªrray_name[get_global_id(0)-1]". Todos los work-items realizaran la misma tarea, y si todos la ejecutasen en el mismo tiempo el algoritmo funcionaría perfectamente. Sin embargo, como se dijo, esto no es así, y como no se garantiza ejecución simultanea, un kernel puede escribir y leer una variable desactualizada del arreglo. Para evitar esta situación, OpenCL provee mecanismos de sincronización llamados "barriersz "fences". Estos sólo toman validez dentro de un work-group y son tratados en la sección V.

Finalmente, OpenCL permite la creación de "Kernels

Nativos", que son kernels OpenCL pero con extensiones diseñadas para dispositivos específicos de uno o mas fabricantes. Esta filosofía se hereda de OpenGL, la librería gráfica, capaz de implementar funciones propias de cada tipo de placa de video. Este esquema permite aprovechar ventajas de procesamiento y funcionalidades específicas. Es claro entonces que una aplicación host puede, luego de entender que dispositivos están disponibles en una plataforma, decidir si utilizar un kernel especializado o no.

III-B. Propiedades del Host

El programa host es el encargado de configurar y administrar la ejecución de kernels OpenCL por medio del uso de contextos. Los contextos se definen como una colección de uno o mas dispositivos OpenCL cuya creación, configuración, y subsecuente liberación son gestionadas por el programa host por medio de las API OpenCL. Una vez que un dispositivo OpenCL se crea y configura apropiadamente -ya sea una CPU, una GPU, u otro- el host puede puede poner en cola escrituras y lecturas de memorias, objetos de programas (kernels), binarios, y comandos de sincronización para un dispositivo especifico. En caso de la utilización de más de un dispositivo, diferentes colas de comandos deben ser utilizadas para cada uno. Estas colas pueden ser configuradas para ser ejecutadas en orden o sin orden. Es importante tener en cuenta que el ordenamiento de las colas impacta solo en los comandos de host y no los kernels individuales, cuya ejecución ordenada no puede ser garantizada. Queda claro a esta altura que en OpenCL existen herramientas de sincronismo a nivel de dispositivo (barriers y fences para sincronizar kernels), y a nivel plataforma (ordenamiento de colas de comandos). Se profundiza mas en mecanismos de sincronización en la sección V.

IV. Modelo de Memoria

Evidentemente, se deduce de los modelos anteriores que es inexistente acceso simultáneo a la memoria por parte del host y los kernels. Dada esta independencia, la gestión de la memoria debe ser explícita para transferir memoria del host a uno o mas dispositivos. OpenCL provee diferentes llamadas a la API para este fin ya sea para leer o escribir datos.

OpenCL define, como se muestra en la figura 5, diferentes zonas de memoria en función de su grado de acceso:

- Memoria Host: Es la región de memoria disponible para el programa Host.
- Memoria Global: Es la región de memoria a la que todos los work-items tienen acceso de lectura y escritura tanto del lado host como del dispositivo. Esta memoria debe ser reservada en tiempo de ejecución. En caso de dispositivos GPU este es el típico caso de la memoria del frame buffer.
- Memoria Constante: Es la región de memoria global a la que los dispositivos solo tienen acceso de lectura. Sin embargo, el programa host tiene permiso de lectura y escritura.

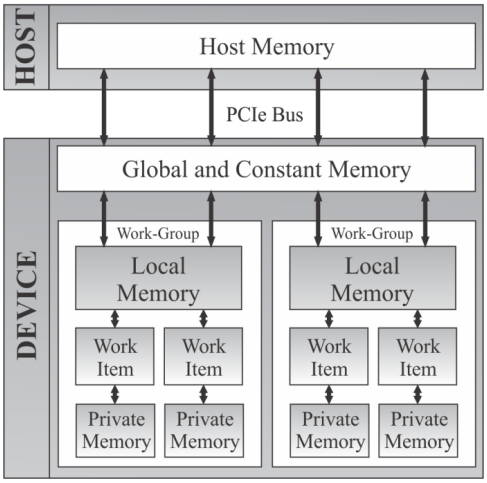

Figura 5. N=2 y 20x20 Kernels

- Memoria Local: Es la región de memoria compartida por todos los work-items dentro de un work-group. En el caso de GPU este suele ser el almacenamiento local de cada unidad de cómputo.
- Memoria Privada: Es la región de memoria accedida exclusivamente de manera local por un work-item dado.

Es importante notar que la aplicación host solo accede a la memoria global del dispositivo. Acceder a la memoria Local no esta permitido en la especificación OpenCL. En caso de que se requieran transferencia de memoria directas hacia la memoria local, el driver OpenCL usará la memoria global como estadío intermedio, lo mismo resulta factible en el camino inverso.

Respecto a la consistencia de memoria, OpenCL supone un modelo relajado",o en otras palabras, el estado de la memoria no se garantiza consistente para todos los work-items en todo momento. Se puede lograr consistencia explicita entre memoria Global y Local entre los work-items de un work-group dado por medio del mecanismo de "barreras".

Cuadro I
Taza de Datos PCIe

Single Lane Capacity		**16 Lane Capacity**	
V1.x	250 MB/s	V1.x	4 GB/s
V2.x	500 MB/s	V2.x	8 GB/s
V3.x	1 GB/s	V3.x	16 GB/s

En general, la transferencia de datos entre la memoria host y el dispositivo OpenCL se da por medio de una interface como por ejemplo PCIe. Diferentes taza de datos para este mecanismo están disponibles en el mercado y se resumen en el cuadro I. PCIe modernos sobre 16 pistas pueden lograr hasta 8Gbps teóricos bidireccionales. A pesar de que aparente ser una taza satisfactoría para cualquier cálculo promedio, esta se vuelve rápidamente el çuello de botella"de un programa OpenCL. Esto queda claro al obvervar el ancho de banda de 150Gbps de memoria RAM con la que cuenta una ATI Radeon 5870 por ejemplo. Esto quiere decir que un work-

item ejecutandose en esta GPU accederán la memoria del dispositivo al menos 20 veces mas rápido que a la del host. Esto resulta crítico a la hora de diseñar la secuencia de transferencia de memoria de un programa OpenCL. Es siempre visto como muy buena práctica minimizar la transferencia de datos host-dispositivo y viceversa. Se recomienda transferir al dispisitivo tanta información como sea posible durante la inicialización -y dejarla allí- antes de iniciar la ejecución del los Kernels. Esto no supondría mayores inconvenientes, al menos en las GPUs dada la gran capacidad de memoria RAM con la cuentan.

V. Modelo de Programación

El modelo de programación define mecanismos de paralelismo de datos y de tareas así como híbridos de los mismos. Sin embargo, el modelo de paralelismo de datos es el principal fundamento de OpenCL.

- Modelo de Paralelismo de Datos: Este modelo define la computación como una secuencia de instrucciones constantes aplicada a múltiples elementos datos. Los índices de dimensión N son la base para mapear a diferentes porciones del objeto de memoria. Cabe destacar que una región de memoria puede ser accedida por múltiples kernels.
- Modelo de Paralelismo de Tareas: En este modelo los kernels son ejecutados independientemente de si número de índice. Aquí se ejecutan diferentes kernels en diferentes work-items.

Cuando utilizamos el modelo de paralelismo de datos, el manejo de consistencia de memoria debe declararse de manera explícita por medio de comando de sincronismo. En OpenCL existen dos dominios de sincronización: work-items en un work-group, y cola de comandos. El primer permite controlar ejecución de los kernels dentro de un dispositivo OpenCL mientras que el segundo permite administrar el paralelismo entre diferentes dispositivos.

V-A. Sincronismo a nivel dispositivo

Las barreras (”barriers”) permite colocar puntos de control en la ejecución de kernels dentro de un work-group. Las barreras son comandos embebidos en OpenCL que son llamados desde los Kernels y pueden ser vistos como puntos en los cuales todos los kernels controlan, y eventualmente esperan, que los demás lleguen al mismo estado. Un work-item dado no continuará su ejecución a menos que todos los demás dentro de su grupo lleguen al mismo punto de control. Si recordamos que no se garantiza ejecución simultanea de los kernels, las barreras resultan una herramienta fundamental para controlar la ejecución y ordenar el acceso a la memoria.

Se debe prestar cuidado especial cuando se utilizan barreras dentro de instrucciones condicionales, dado que si por cualquier circunstancia, un solo kernel no ejecuta la barrera, termina bloqueando la ejecución de todo el programa. La regla general deberá ser: o todos los work-items entran en la sentencia condicional o todos la rechazan.

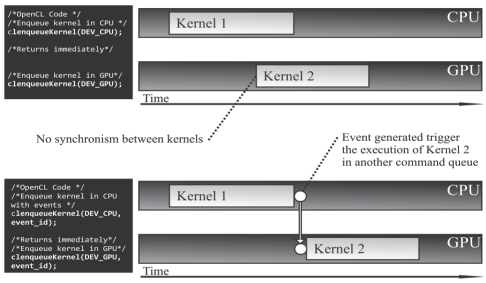

Figura 6. Sincronismo por eventos

V-B. Sincronismo a nivel host

Las colas de ejecución de comandos son utilizadas para leer y escribir memoria, compilar y ejecutar kernels, entre otros. Cada uno de estos comandos pueden estar indicados con una bandera de bloqueo o no, y pueden ser ejecutados en orden o no. La bandera de bloqueo indica si la función puesta en la cola retorna inmediatamente o espera que efectivamente se ejecute la acción en el dispositivo. El uso inteligente de esta bandera permite al programa host administrar de la mejor forma posible diferentes ejecuciones en diferentes dispositivos.

El uso de comandos con bandera de bloqueo es probablemente la manera mas sencilla de mantener el programa host y el dispositivo en sincronismo. Sin embargo el host queda esperando la ejecución del comando perdiendo precioso tiempo que podría ser utilizado para -por ejemplo- administrar actividades de input output de usuario.

El concepto de eventos implementado en OpenCL permite a la aplicación no bloquearse y continuar con otras tareas, para luego volver y controlar el estado del evento. En caso de que deban poner en cola de ejecución kernels destinados a diferentes dispositivos resulta fundamental que el programa host controle la ejecución ordenada por medio de eventos para lograr la sincronía de los dispositivos. Comandos como clWaitForEvents o clEnqueueWaitForEvents permite al host esperar la ocurrencia de un evento determinado, o poner esta espera en la cola de ejecución de un dispositivo dado. La utilización de eventos se ilustra en la Figura 6.

VI. Marco de Trabajo

Además del análisis de OpenCL por medio de los modelos, se complementa con el concepto de marco de trabajo o ”framework”que apunta a describir las diferentes implementaciones de las funciones en un sistema.

VI-A. Compilador OpenCL

La especificación del lenguaje describe la sintaxis y la interface de programación de Kernels a ser compilados en tiempo de ejecución. Este lenguaje esta basado en el estandard ISO C99 con algunas extensiones y restricciones mínimas. Respecto al tipo de datos y funciones, OpenCL soporta escalares y vectores, conversiones implícitas (casting), funciones matemáticas y de control de work-item.

VI-B. Plataforma OpenCL

La API de Plataforma es la que permite, por medio del sistema operativo, acceder al set completo de dispositivos OpenCL compatibles. La plataforma responde a consultas sobre estados, dispositivos disponibles y sus capacidades asociadas. La creación y administración de contextos se realiza también en este marco.

VI-C. API de tiempo de ejecución

La API de tiempo de ejecución gestiona objetos asociados a un dispositivo en particular como colas de comandos, objetos de memoria, kernels, entre otros.

VII. Implementación: Multiplicación de Matrices

Con la finalidad de aplicar los modelos desarrollados, y de medir la mejora en velocidad que se puede obtener en un sistema de computación hogareño, en este trabajo se propone un algoritmo simple de convolución de matrices en OpenCL. Es claro que este tipo de algoritmos son altamente paralelizables dado que cada iteración de cálculo para cada elemento C_{mn} de la matriz resultado es independiente de los demás. Fácilmente se puede destinar un kernel sencillo para cada elemento para realizar la tarea en paralelo.

$$A_{mxn} \times B_{nxm} = C_{mxn} \quad donde \quad c_{i,j} = \sum_{k=1}^{n} a_{i,k} + b_{j,k} \quad (1)$$

$$C_{2x2} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \quad (2)$$

$$C_{2x2} = \begin{pmatrix} a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} & a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{2,2} \\ a_{2,1} \cdot b_{1,1} + a_{2,2} \cdot b_{2,1} & a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} \end{pmatrix} \quad (3)$$

Dado que la implementación a evaluar es sobre matrices de dos dimensiones, resulta conveniente enmarcar el algoritmo de solución del problema en un espacio de ejecución justamente de N=2 dimensiones. Cada kernel de dicho espacio entonces tendrá un identificador global expresado de la forma (x,y), al que mapearemos a los índices mxn de nuestra matriz resultado ilustrada en la Figura 6. El enfoque descripto nos permite expresar el algoritmo de manera apropiada para ser resuelto por un programa basado en OpenCL.

El código ejemplificado en la Figura 7 aplica el esquema descripto de obtener el identificador global del kernel en ejecución (único para cada uno de los mxn kernels). Estos identificadores globales (x,y) se corresponden con el índice mxn, y son usados como dereferenciadores de los arreglo que almacenan las matrices a multiplicar A y B. Un bucle "for"nos permite desplazarnos por toda las columnas y filas de las matrices A y B basándonos en la ubicación derivada con las funciones OpenCL get_global_id(). Una vez acumulados los valores del elemento m,n, se almacena en el arreglo resultado C.

Se decidió evaluar el programa en una plataforma con dos dispositivos: una CPU Intel i7 y una GPU ATI Radeon 5870 Mobility. La plataforma utilizada es "*OpenCL 1.1 AMD-APP-SDK-v2.5 (793.1) FULL_PROFILE*"la que acusa que las características de los dispositivos descriptos son los mostrados en el cuadro II. El programa host desarrollado inicializa el contexto OpenCL y pregunta al usuario si desea realizar la multiplicación usando el algoritmo en C++ plano (sólo en CPU) o en OpenCL pudiendo elegir enviar los kernels al CPU o al GPU. También se planteó como variable el tamaño de las matrices a convolucionar. Es de esperar que a mayor tamaño se explote mejor el carácter de paralelo del algoritmo. Se realizó un promedio de 100 simulaciones para cada caso de los expuestos en las Figuras 8 y 9. Para la medición del tiempo de ejecución se utilizó la librería time.h. Los puntos de medición en el caso del lenguaje C++ plano se colocan al principio y final de la función multiplicación. Pero en OpenCL se mide el envío de los datos a la memoria del dispositivo, la ejecución de los kernels, y la lectura de los resultados hacia el programa host. Este esquema nos permite representar la experiencia final del usuario en la que influye los tiempos de transferencia, en este caso, por puerto PCIe.

Cuadro II
Hardware para prueba

CPU		**GPU**	
Intel(R) i7 CPU Q720 @1.60GHz		**ATI HD 5870 Mobility**	
ComputeUnits	8	ComputeUnits	10
ClockRate	1596MHz	ClockRate	700MHz
GlobalMem	2048MB	GlobalMem	1024MB
LocalMem	32Kb	LocalMem	32Kb
2D ImageSize	8192	2D ImageSize	8192
3D ImageSize	2048	3D ImageSize	2048
WorkItemsSize	1024	WorkItemsSize	256
WorkGroupSize	1024	WorkGroupSize	256

Es interesante notar que el tiempo de ejecución del algoritmo puede mejorar de 12.7 a 0.086 segundos para el escenario de matrices de 1024. Esto es coherente con lo expresado en las primeras secciones sobre la capacidad de paralelismo de las GPUs modernas. Esta mejora en velocidad se hace despreciable para tamaño de matrices pequeños, donde el gasto extra de transferir datos a la memoria de los dispositivos se convierte en prohibitivo. Es necesario destacar que no son

```
__kernel void
   matrixProd(__global float*C,
              __global float* A,
              __global float* B,
              int colA, int colB)
  {
     int x=get_global_id(0);
     int y=get_global_id(1);
     float valor=0;

     for(int k=0; k<colA; ++k)
     {
        float eA = A[y*colA + k];
        float eB = B[k*colB + x];
        valor += eA * eB;
     }
     C[y*colA + x] = valor;
  }
```

Figura 7. Kernel Producto de Matrices

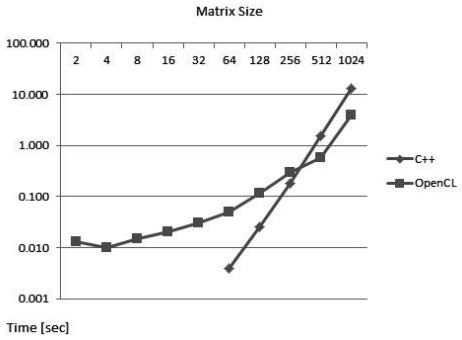

Figura 8. Intel i7: C++ vs OpenCL

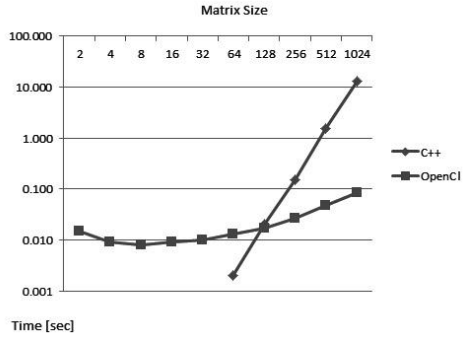

Figura 9. Intel i7: C++ vs ATI Radeon: OpenCL

hardware aislados, si no que son capacidades de procesamiento disponibles en conjunto en cualquier sistema de computación hogareño, pero se encuentran desperdiciados y ociosos en la mayoría de los casos. OpenCL, permite explotar al máximo las capacidades de procesamiento disponibles.

VIII. Conclusión

La especificación OpenCL del Khronos Group provee una perspectiva técnica y abstracta sobre el camino a la implementación del lenguaje. Por otro lado los ejemplos proporcionados por Nvidia o ATI son funcionales pero pecan de explicaciones claras del funcionamiento. El presente documento tiene la intención de cubrir el espacio de conocimiento generado por estas dos últimas. Un ejemplo de implementación muy simple nos permitió aplicar los modelos teóricos y hacer unas primeras mediciones del impacto que podría generar la mejora de velocidad de cálculo que se puede obtener al utilizar inteligentemente los recursos de una plataforma que se muestra mal explotada la mayor parte del tiempo. Se demostró un caso en el que al utilizar el dispositivo mas capaz de un sistema dado se puede mejorar en 3 ordenes de magnitud la eficiencia del algoritmo propuesto. Como futuro trabajo se destaca la implementación de algoritmos mas complejos (se está trabajando en la problemática de sistemas de partículas N-Body). La utilización de mecanismos de medición de tiempo mas precisos también es una mejora del presente trabajo en el futuro. No es difícil concluir que OpenCL resulta una tecnología prometedora que probablemente este presente en la mayoría de las plataformas de procesamiento del futuro cercano. La mejora en velocidad se puede aplicar a numerosas aplicaciones como audio, reconocimiento de voz, video, imagen, bases de datos, física de tiempo real, entre otros. Es claro el camino que abre OpenCL: habilitar la ejecución de aplicaciones antes destinadas a supercomputadoras en sistemas de computo disponibles al público en general.

Referencias

[1] OpenCL Specification v1.1 – Khronos Group (Revision 36, Sep 30, 2010) http://www.khronos.org.
[2] Introduction to OpenCL programming ATI/AMD (Publication #137-41768-10 Rev:A Issue date May 2010) https://ssl-developer.amd.com.
[3] OpenCL Programming Guide – ATI Stream Computing (May 2010 Rev 1.0f) https://amd.com
[4] CMSoft Documentation (OpenCL template creators) http://www.cmsoft.com.br/
[5] A Comprehensive Performance Comparison of CUDA and OpenCL.- Jianbin Fang (IEEE Parallel Processing (ICPP), 2011)
[6] Invitation to OpenCL - Alvarez, P.L. Networking and Computing (ICNC), 2011 - IEEE