Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7019-7041 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

LINEAR ALGEBRA ON GENERAL PURPOSE GRAPHICS PROCESSING UNITS USING OPENCL

Santiago D. Costarelli[a], **Rodrigo R. Paz**[a,b], **Lisandro Dalcin**[a,b] **and Mario A. Storti**[a,b]

[a]*Universidad Nacional del Litoral, Facultad de Ingeniería y Ciencias Hídricas, Santa Fe, Argentina.*

[b]*Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC), Instituto de Desarrollo Tecnológico para la Industria Química (INTEC), Consejo Nacional de Investigaciones Científicas y Tecnológicas (CONICET), Universidad Nacional del Litoral (UNL), Güemes 3450, (S3000GLN) Santa Fe, Argentina, http://www.cimec.org*

Keywords: OpenCL, CUDA, GPGPU, Tesla, Fermi.

Abstract. Motivados por el vasto crecimiento sobre las tecnologías en placas procesadoras de video y las versiones recientes de dispositivos para cálculo de propósito general GPGPU's (General Purpose Graphic Processing Unit), es que se presenta en este trabajo algunos desarrollos de cálculo paralelo en Algebra Lineal utilizando OpenCL, un estándar de reciente concepción para la programación en GPGPU's libre de regalías, que utiliza un subconjunto de operaciones definidas del estándar C99 de la ISO.

En las primeras secciones se presenta la arquitectura CUDA (teniendo en cuenta que OpenCL se basa en ella) analizando los elementos que la componen, brindando lineamientos básicos en cuanto al uso.

A continuación se presentan diferentes métodos para el cálculo de operaciones básicas de Algebra Lineal. Con respecto a las implementaciones de las operaciones propuestas, se analizan diferentes versiones de la misma operación valiendonos de las bondades que nos brinda la arquitectura CUDA, entre ellas: operaciones en memoria global, memoria local (o shared), combinaciones de las anteriores y cálculo multiGPU; teniendo en cuenta aquí las propiedades de *memory coalescing*, divergencia de workItems dentro del warp, por nombrar algunas. Además, se estudia la influencia de las cifras decimales exigidas sobre el costo computacional y la utilización de parámetros de optimización ya sea mediante directivas al compilador o funciones built-in provistas por OpenCL.

Finalmente, se estudian las performances de los algoritmos anteriormente nombrados utilizando para los casos de estudio tecnologías NVIDIA Tesla C1060 y procurando establecer las deficiencias de la misma en relación a la arquitectura NVIDIA Fermi.

1 INTRODUCCIÓN

Gracias al continuo avance de la tecnología, es natural hoy en día hablar de computadoras con multiprocesadores. Para aquellos habituados al cálculo computacional, esto no es materia nueva pues desde hace ya algunos años se utilizan clusters de computadoras para la resolución de grandes problemas en ciencias e ingeniería.

Conforme la tecnología de los procesadores fue madurando, un límite físico se fue imponiendo sobre estos, el tamaño del transistor, por lo cual se comenzaron a desarrollar multiprocesadores compuestos de procesadores de menores frecuencias sobre el mismo dispositivo y de esta forma, mediante la utilización de balanceo de procesos sobre los mismos, se logró incrementar el rendimiento con respecto a los predecesores procesadores mononúcleo.

Paralelamente, las compañías desarrolladoras de placas de video (GPU) comenzaron una lucha desenfrenada por el dominio sobre el poder del cálculo gráfico, principalmente para juegos y diseño 3D. Con el avance del tiempo, el nivel de cálculo soportado por las GPU's fue potencialmente creciendo hasta lograr eclipsar al del CPU, esto es lógicamente fácil de comprender, pues la placa de video esta ideada para un cálculo intensivo mediante la incorporación de múltiples unidades aritmético lógicas sin muchas particularidades sobre controles de flujos y memorias caché, mientras que el CPU esta seriamente limitado por lo anterior, pues posee un control de flujo mucho más exigente, además de una memoria caché de tamaño importante, utilizada para lectura de intrucciones de manera más ágil y de contar con pocas UAL (unidades aritmético lógicas). Por estos motivos el tamaño del chip en las GPU es mucho menor y la densidad de transistores es notoriamente mayor, lo que le confiere a la GPU una supremacía en poder de cómputo.

Además cabe destacar que cuando un sistema operativo tiene la posibilidad de ejecutar múltiples procesos simultáneos, la realidad que se encuentra por detrás de ello es la técnica de *Time Sharing*, que consiste en otorgar paquetes de tiempo al proceso durante el cual este se podrá ejecutar, el problema esta en que cada proceso tiene asociado a sí mismo un contexto, que es básicamente la información que necesita para su ejecución, y la tarea de conmutarlos cada vez que un nuevo proceso toma el control de ejecución requiere de muchos ciclos de reloj. En el caso de las GPU's esto no ocurre pues los registros son alocados por unidades especiales (denominados thread en la arquitectura CUDA, workItems en la nomenclatura OpenCL) y por lo tanto no hay intercambio de registros o estados entre threads del GPU, es decir, presenta contextos livianos en comparación al CPU.

Así es como nace la idea de la utilización de GPU's para la resolución de problemas de propósito general con posibilidad de paralelismo, el término correcto sería GPGPU.

La arquitectura a estudiar se conoce como CUDA, fue desarrollada por NVIDIA y tiene como principal aporte la posibilidad de desarrollar aplicaciones que puedan correr sobre cientos o miles de threads simultáneamente, esto es, permite que un código compilado pueda ejecutarse sobre distintos dispositivos con independencia de la cantidad de procesadores que disponga. Así, gracias al escaso peso de su información de contexto, los threads pueden crearse, intercambiarse y organizarse con un alto grado de eficiencia por parte del hardware, lo cual va a contracorriente con lo que se conoce en CPU's. A lo anterior se lo encuentra generalmente en la bibliografía como escalabilidad, dado que la arquitectura ajusta la solución al hardware que se posee de forma directa.

En este artículo indagaremos sobre las nociones básicas de OpenCL, que es un estándar para GPGPU libre de regalías, con independencia del proveedor del dispositivo de cálculo (plataforma heterogénea), una API y un lenguaje particular descendiente del estándar

C99. Trataremos de definir la estructura básica de toda aplicación OpenCL, además de la implementación de algoritmos básicos del Algebra Lineal con el objetivo de luego integrar los mismos con otras estructuras y así poder conformar, por ejemplo, solvers lineales capaces de utilizar el poder que nos ofrece OpenCL en aplicaciones de CFD.

2 ARQUITECTURA

La arquitectura en sí tiene como elemento primordial un *computeDevice*, que es un dispositivo de cálculo con posibilidad OpenCL y que está alojado en un periférico al que denominaremos *host*. A su vez, estos estan compuestos de *computeUnits*, que entre los elementos que conforman a estos últimos encontramos los *processingElements*, unidades de funciones especiales (SFU's), unidades de doble precisión y memoria caché. El esquema comentado anteriormente se observa en la Fig. 1.

Cabe destacar que la disposición y cantidad de los ítems declarados anteriormente es totalmente dependiente del hardware que se posea, para el caso de la Tesla C1060 disponemos de 30 computeUnits cada una con 8 processingElements con posibilidad sobre cálculos a simple precisión o enteros, 1 unidad de doble precisión y 2 unidades de funciones especiales. Cada computeElement puede realizar 3 operaciones de punto flotante por cada ciclo de reloj lo que brinda un rendimiento teórico pico de 1296 Mhz x 240 cores x 3 operaciones de punto flotante por clock (1 multiplicación/división, 1 multiplicación y 1 de otro tipo de función), un total de 933 Gflops (Howard (2009)).

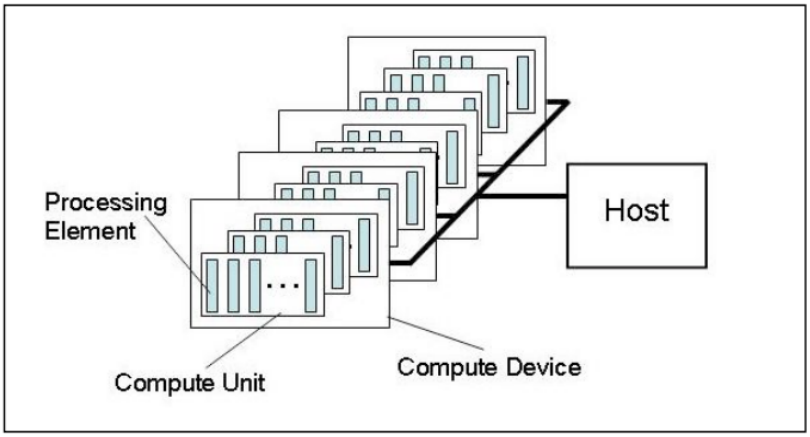

Figure 1: Modelo de la plataforma de ejecución donde se observan los componentes de la arquitectura CUDA en terminología de OpenCL. Group (2009).

Las funciones que serán lanzadas sobre estos dispositivos se denominan kernels y el modo en que se ejecutan sobre los mismos son detallados a continuación.

Básicamente un kernel tiene como unidad fundamental un workItem, que se podría definir como el elemento básico de la arquitectura constituyentes de los denominados workGroups, definiendo estos últimos lo que se conoce como espacio de índices, cuyo conjunto define la partición del problema.

Una vez efectuada la división del problema en workGroups, se prosigue a derivar los mismos sobre los computeUnits con el objetivo de formar una cola sobre cada uno como se observa en la Fig. 2, donde block se refiere a los workGroups y SM a los computeUnits (la terminología anterior es la respectiva a CUDA).

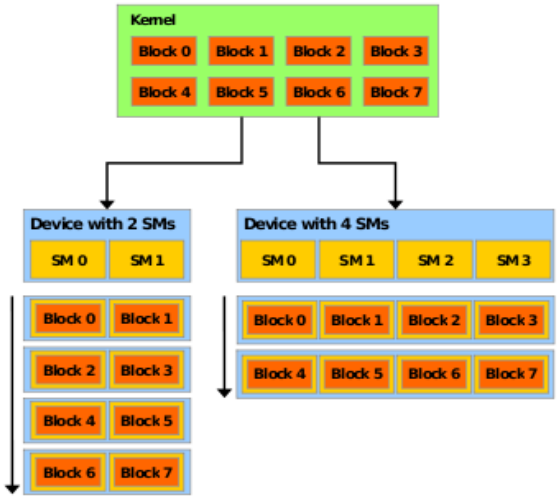

Figure 2: Distribución de workGroups sobre los computeDevices, en el ejemplo se tienen dos computeDevices que presentan 2 y 4 computeUnits respectivamente, sobre los cuales se encolan 8 workGroups.NVIDIA (2009b).

Para poder tratar con cientos de workItems por computeDevice la arquitectura recurre a un scheduler denominado *SIMT (single-instruction,multiple-thread)* que básicamente desgloza estos workGroups en *warps* que estan conformados según el computeCapability del dispositivo, para el caso de la Tesla C1060 con computeCapability 1.3, por 32 workItems con ID's crecientes, es decir, sucesivos, y los deriva al computeUnit siempre y cuando se dispongan de aquellos recursos necesarios para la ejecución de cierto workGroup. En caso de no cumplir la condición de recursos solicitados, la arquitectura automáticamente reduce la cantidad de workGroups por computeUnit hasta que los recursos utilizados se encuentren dentro de los límites posibles. Esto se realiza teniendo en cuenta que los workItems de un warp en particular ejecutan la misma instrucción (operación) y de esta forma se amortiza la búsqueda y ejecución de la instrucción con varios workItems. Lo anterior teniendo en cuenta que todos los workItems del warp siguen el mismo *camino de ejecución*, tema que será tratado más adelante.

Como se ha dicho anteriormente, los workItems dentro del mismo warp tienen ID's crecientes, es decir, para workGroups multidimensionales se hace una proyección unidimensional, entonces por ejemplo uno podría identificar los ID's del warp k-ésimo mediante $k * 32$ para el primer ID's del workItem mientras que $(k + 1) * 32 - 1$ para el último, esto siempre y cuando el workGroup sea múltiplo del warpsize, de otra forma, se completa el faltante con workItems extras. Por ejemplo, si se tuviera un bloque de 56 workItems, se tendrían dos warps, el primero con 32 útiles, mientras que el segundo sólo tendría 24 útiles, y 8 agregados.

Para compensar el tiempo de cálculo o posibles demoras en la ejecución inmediatamente se trata de preparar otro warp, el cual tiene asociado otro conjunto de entradas, con el objetivo de que ante cualquier evento que produzca un retraso, la unidad SIMT pueda asignar un nuevo warp a la unidad y así mantenerla ocupada. Por ejemplo si se tiene que la latencia de una operación en cierta memoria es de C ciclos, luego se necesitarán $C/(k*4)$ warps para mantener al computeUnit ocupado en su totalidad dado que cada computeUnit ejecuta k instrucciones con 4 ciclos de demora del tipo supuesto por warp, denominado generalmente como zero-overhead, de esta forma el acceso a memoria se amortiza con una cierta cantidad de warps que ejecutan ciertas intrucciones asociadas a una determinada cantidad de ciclos de reloj.

La latencia entonces es ocultada intercambiando workItems, con la intensión de mantener a los computeUnits ocupados la mayoría del tiempo, por ende un ejercicio interesante es

averiguar cuantos workItems por computeUnit son necesarios para ocultar la latencia aritmética de determinado dispositivo.

Además, cabe nombrar que debido a que un computeUnit esta integrado por 8 processingElements, y a su vez el warp tiene 32 workItems, tenemos un total de 4 de los anteriores compartiendo el dispositivo de cálculo, por ende aquí también se esta, de alguna manera, compensando el tiempo que pudiera no ser utilizado.

Otro aspecto a tener en cuenta dentro de GPGPU, es el *computeCapability* de nuestro dispositivo, de acuerdo al mismo, el SIMT podrá mantener 24 o 32 warps y un máximo de 512 o 1028 workItems activos por computeUnit (en dispositivos con computeCapability 1.3 la relación introducida anteriormente se obtiene realizando 32 warps * 32 workItems = 1024 workItems).

Además, como regla base de la arquitectura, no se permiten workGroups con más de 512 workItems. A su véz, la capacidad de workGroups por computeDevice tampoco es ilimitada, existe una relación estrecha entre la cantidad de memoria local y los registros consumidos por el kernel, por ende es recomendable no utilizar una cantidad exagerada de registros o memoria local asociados al kernel.

Una consideración importante acerca de los registros merece ser introducida aquí. De acuerdo a las especificaciones la GPGPU Tesla C1060 posee unos 16384 registros por computeUnit, como es computeCapability 1.3 puede tener 1024 workItems activos lo que da un total de $16384/1024 = 16$ [registros/computeElement], en caso de exceder ese límite la arquitectura se contrapone incorporando menor cantidad de workItems por computeUnit, lo que como se analizó anteriormente, pierde capacidad para contrarestar el tiempo de retardo de operaciones costosas. Según Kirk and mei Hwu (2010) la reducción de workItems esta en relación con el tamaño del workGroup, p.e. si se disponen de 1024 workItems residiendo en un computeUnit en workGroups de 256 workItems y se utilizan más de 16 registros por workItem, luego cada computeUnit sólo podrá tener activos 768 workItems. Además otro aspecto que limita los workItems dentro del computeUnit es la memoria local, que como se sabe se dispone solamente de 16 Kbytes por computeUnit, de las especificaciones de la Tesla C1060 se tiene que puede tener 8 workGroups activos por computeUnit, se determina entonces que cada computeElement dispone de 2 Kbytes de memoria local para su utilización. De lo anterior se deriva que en caso de utilizar 8 workGroups cada uno utilizando 2 Kbytes o menos, se obtiene el máximo numero de workGroups a ejecutar,de otra forma si por ejemplo utilizaran 4 Kbytes sólo se podría disponer de 4 workGroups en ejecución.

Con respecto a la jerarquía de memoria en OpenCL tenemos 4 tipos: local, private, constant y la global. La memoria local tiene una relación estrecha con los workGroups, y es una memoria on-chip que lógicamente permite acceso con latencias mucho menores que sobre la global para workItems del mismo workGroup. Si bien la velocidad es una ventaja, el punto en contra son los 16 Kbytes de disponibilidad por computeUnit. Funciona al estilo caché para reducir accesos a la memoria global, además no tiene el problema de coalescence de la global. Los accesos a la misma se realizan en dos grupos de 16 workItems cada uno (halfWarp).

La memoria global en general es una vasta área de memoria de acceso lento donde por ejemplo se cargan los datos del host, la constant (que tiene una performance de 32 bits por warp por clock por computeUnit) es una memoria cacheada con un total de 64 Kbytes por computeUnit y tiene la peculiaridad de que los workItems de un halfWarp pueden acceder tan rápido a ella como si fuera el tiempo de acceso a un registro, sólo si todos ellos acceden a la misma dirección de memoria, de otra forma ocurren accesos serializados y esto reduce la eficiencia de esta memoria. Finalmente, en la memoria private se guardan registros y es de

acceso casi inmediato.

Lo anterior se puede observar en la Fig. 3.

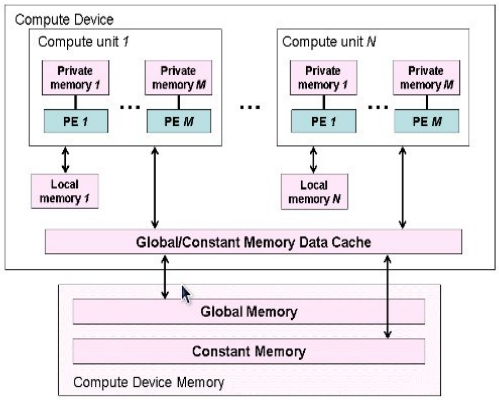

Figure 3: Jerarquía de memoria en OpenCL. Group (2009).

Un concepto que hay que analizar es el de halfWarp, básicamente tiene origen en la forma en que está organizada la memoria local. Internamente se encuentra dividida en 16 vecindades denominadas *bancos* que se conforman con palabras de 32 bits, más especificamente, palabras sucesivas de 32 bits se derivan a bancos sucesivos, con lo cual una operación de halfWarp que toma 2 ciclos de reloj y lee los 32 bits por workItem (NVIDIA (2009b)), lo puede hacer de manera óptima si la lectura se hace sobre distintos bancos (la arquitectura parece estar optimizada para elementos alineados a 4 bytes).

Debido a la disposición de los bancos y de los warp, es necesario un ajuste sobre alguno para simplificar el acceso a memoria y tratar de realizar la máxima cantidad de accesos en forma paralela. Con este objetivo el halfWarp original es subdividido en dos elementos, un warp superior, y otro inferior y los accesos sobre los bancos se hace en dos pasadas, con lo que se tiene asegurado que el acceso al mismo banco por dos workItems de distintos halfWarps no produzcan una serialización en el acceso. En caso de ocurrir la serialización primero se atenderán a los que no presentan dicho inconveniente, y luego a los que si.

De lo anterior nace el concepto de que el NDRange (espacio de índices del problema) tenga en sus dimensiones valores múltiplos de 32, para estar ligado a la dimensión del warp y así maximizar el rendimiento de la operación (es decir, aprovechando todos los workItems dentro del warp).

La razón de generar colas de warps por computeUnit, teniendo en cuenta que sólo se ejecuta uno en un preciso instante tiempo, se encuentra en que cuando el warp esta asociado a operaciones de alto grado de retardo, como lecturas de la memoria global, la arquitectura ubica a este en colas de warp en espera y se asigna un nuevo warp al computeUnit. Sea el caso de existir sólo un warp disponible para la ejecución, inmediatamente se asigna sin más complicaciones. Si existieran múltiples warps activos esperando la ejecución listos para ejecutarse se aplica un criterio de selección. Finalmente en caso de no existir otro warp listo para su ejecución, se espera a que alguno de la lista de espera continúe con su ejecución. El hecho de ejecutar warps disponibles para ejecución no agrega retardo al tiempo total de ejecución, lo que se denomina *zero-overhead thread scheduling*.

Lo que sí es importante en la lectura de la memoria global es el *memory coalescing*, que básicamente consiste en que los halfwarp tomen los datos de la memoria global en posiciones

dentro del bloque de memoria a leer, es decir, que no existan saltos o intervalos entre los datos a leer que esten fuera de la misma, pues de esta forma se desperdicia rendimiento. De lo anterior se tiene que si bien en dispositivos con computeCapability 1.2 o superior un acceso coalesced se tiene incluso cuando workItems del mismo halfWarp accesan a la misma posición de memoria, esto no ocurre con dispositivos con computeCapability anteriores a 1.2, que necesitan además del acceso sobre la misma porción de memoria, que los datos se encuentren contiguos.

Para nuestro caso (Tesla C1060) cuando los workItems dentro de un warp ejecutan una instrucción de carga, el hardware detecta cuando estos accesos se realizan sobre el bloque, de esta forma se realizan lecturas con una eficiencia próxima a la máxima otorgada por la memoria global, luego se produce un acceso coalesced.

Es importante notar que la memoria local no requiere accesos coalesced, puesto que tiene un latencia mucho menor a la global.

Recordar entonces que *sea cual sea el computeCapability de nuestro dispositivo, como la memoria se lee en bloques (y es ahí de donde se obtienen los altos anchos de banda en la arquitectura CUDA), mientras más workItems accedan a posiciones de memoria del mismo bloque, mejor es el rendimiento de lectura del warp*.

Con respecto a los controles de flujo en la naturaleza SIMT de la arquitectura CUDA, podemos identificar dos recomendaciones posibles para garantizar un buen esquema para el rendimiento de la aplicación.

El primero es el *unroll*, podemos decir que su función básica es la de indicarle al compilador que intente de optimizar alguna secuencia del tipo iterativo mediante el uso de ramificación de predicado. El *unroll* de los ciclos for se utiliza con el objetivo de reducir las instrucciones redundantes en una operatoria sobre la cual se tienen los límites de la sumatoria, es decir, se reemplaza un lazo que internamente presenta una comparación y un índice incremental, por una operación *inline* que prescinde de estos últimos. El segundo se da cuando al menos un workItem de un warp en particular no sigue el mismo camino de ejecución que los demás, se serializan los diferentes caminos de ejecución, así la cantidad de operaciones por warp aumenta lo que deriva en una pérdida de rendimiento. A esto se le denomina *warp divergente*. Aunque dos warps distintos pueden acceder a distintos caminos de ejecución sin afectar la performance.

Entonces queda claro que se deben evitar bifurcaciones o branchs en predicados, pues afectan a la performance del kernel. Si el branch fuera pequeño, en general no es problema, por ejemplo para una PDE (partial differential equation) con condiciones de frontera sencillas se pueden procesar tanto nodos interiores como exteriores de la misma manera; de manera contraria, si el branch fuera muy grande, internamente el compilador realiza algunas operaciones que tienen como resultado una caida abrupta de rendimiento, en el caso de la PDE se recomienda tratar nodos interiores y exteriores por separado. Otro ejemplo podría ser una lista grande de datos a procesar, si existieran dos tipos de datos uno simple y el otro complejo de procesar, en general se realiza una primera pasada que divida los datos en dos sublistas que serán procesadas por separado.

Entonces, dicho de otra manera, si se tienen condiciones if anidadas se necesitará varias pasadas de warps con el objetivo de cumplimentar con todos los bloques condicionales, esto es, de ir por donde los workItem necesitan ir, condición que lógicamente es serial y por ende, lenta. Otra condición problemática es cuando se presentan ciclos donde dependiendo del workItem que se trata tiene cierto límite, por ejemplo, si disponemos de un kernel con un lazo de 10 y 20 iteraciones según se trate de una cierta clase de workItems o de otra, luego se necesitarán dos pasadas de warps, la primera que tome los workItems que necesiten realizar el lazo con 10 iteraciones y a continuación una segunda para aquellos workItems que requieran efectuar el

lazo con 20 pasadas.

Como bien se ha nombrado anteriormente, la arquitectura posee unidades especiales que entre las tareas que realiza encontramos la resoluciones de funciones matemáticas implementadas por hardware, las cuales en esencia estan optimizadas y se aconseja su utilización. Funciones al estilo de las *native*, o las *mad* que son adecuadas cuando el grado de precisión necesario no es de vital importancia en la aplicación, mientras que aquellas como *dot*, *mix*, *max*, entre otros, son precisas pero con rendimientos menores que las anteriores.

3 INTRODUCCIÓN AL LENGUAJE

Toda aplicación en OpenCL tiene básicamente el mismo esquema o esqueleto, esto es, una primera instancia, o capa inicial, en donde se definen los elementos primordiales sobre los cuales se adecuaran luego las funcionalidades de capas superiores se podría decir, conocido en la jerga de OpenCL como los *contextos*, los *dispositivos* asociados a los anteriores y una *cola de comandos* sobre las cuales descansarán las operaciones de R/W entre el host y el device.

Al inicio se deberá consultar a OpenCL qué tipo de *plataforma* se tienen disponibles, entendiendo a esta como el host más una colección de dispositivos manejados por OpenCL, que permiten a una aplicación compartir recursos y ejecutar kernels en los dispositivos asociados a ella. A continuación se deberá crear un contexto, que es un ambiente en donde los kernels se ejecutan y el dominio sobre el cual está definido la sincronización y el manejo de memoria. En este punto se recalca que las operaciones de sincronización pueden efectuarse o bien a nivel host, o bien sobre workItems de un mismo workGroup.

El paso siguiente es asociar una *cola de comandos* a cada dispositivo, de esta forma podremos distribuir el cálculo asignando parte del problema a cada dispositivo y resolviéndolo por separado. Por defecto ni CUDA ni OpenCL distribuyen el cálculo a todos los computeDevices.

Hasta aquí se ha desarrollado lo que antes se ha denominado como capa base, pues es el manejo de la arquitectura a su más bajo nivel. A continuación se introduce la capa intermedia.

Luego de la capa inicial viene una capa intermedia en donde se define un *programa*, al cual se le asocian los *kernels* y luegos sobre estos últimos se compilan sólo aquellas funcionalidades de los mismos que se requieran, esto es, en un archivo con extensión *.cl* se pueden ubicar varios kernels, por ende en el proceso de compilación uno debe indicar explícitamente que compilar. Continuando con la capa intermedia, se compilan los kernels seleccionados. Vale la pena nombrar que, dado dos dispositivos de distinto tipo, por ejemplo GPU y CPU, el programa que se esta corriendo en un momento determinado a alto nivel es el mismo, pero luego al momento de compilar se crean dos binarios totalmente distintos, cada uno con las particularidades del hardware donde corre, se deriva entonces que el mismo puede correrse sobre ambas plataformas.

En otras palabras, como primer paso en la capa intermedia se creará un programa, que consiste en un conjunto de kernels y poseen entre su información los fuentes o el binario del programa, el número de kernels que al momento actual se tiene disponibles para ser corridos en el device, la lista de dispositivos para el cual está construido, entre otros. A continuación se necesitará compilar y enlazar el programa ejecutable, el cual se ha leido con anterioridad para los dispositivos que le correpondan al programa y al contexto pertinente.

Es importante recalcar que, no como en CUDA con su compilador *nvcc*, en OpenCL no esta totalmente regularizado el compilador y queda a disposición del proveedor del device, por ende en el proceso de compilación el hecho de debuggear o ver el resultado de compilación de un kernel parece imposible. Para palear esta desventaja, se puede utilizar una función provista por OpenCL que retorna el estado de compilación resultante del compilador por defecto de

OpenCL.

Finalmente se introduce la capa superior, que en sí define los argumentos de los kernels, el tamaño por workGroup que estan conformados por un tamaño en relación a un eje coordenado euclideano en donde, si se toma como referencia el monitor de la pantalla, el orígen se encuentra en la esquina superior izquierda, con el eje x en dirección horizontal hacia la derecha y el eje y hacia abajo, perpendicular al eje x.

Luego se procede a un *NDRange* en el cual se efectúa una solicitud al host para que compute el kernel de interés. En este se define un espacio de índices y sobre cada punto del mismo se define la ejecución de una instancia del kernel.

Una de las formas en que se organizan los workItems en el espacio de índices es mediante dos tipos especiales de coordenadas, las locales y las de grupo. Las primeras estan referidas a un espacio de índices dentro del workGroup, mientras que las segundas se corresponden a una división del espacio de índices generales propuesto para la resolución del problema a tratar dado las dimensiones generales del problema y las del workGroup. Esta situación se presenta en la Fig. 4.

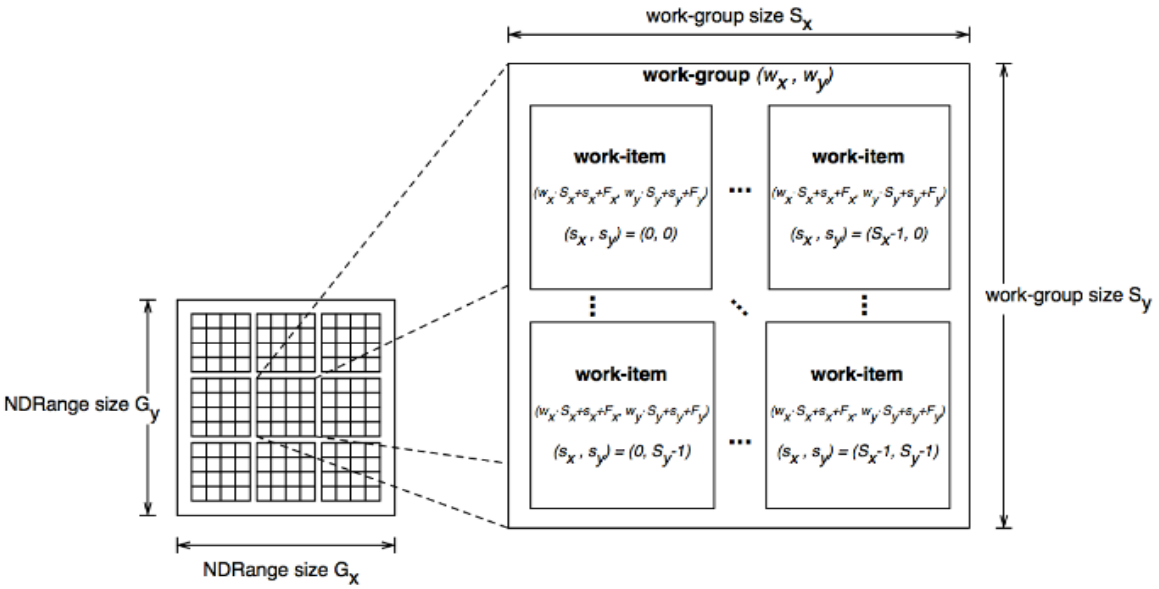

Figure 4: Modelo de Ejecución. Se observa como se divide la memoria del device (objeto memoria), en workGroups que son derivados a los computeUnits, cada uno de éstos con sus respectivos workItems constituyentes.Group (2009).

Existen otras formas de acceder a los índices con otras formas de coordenadas, las globales, que no presentan una división explícita alguna del espacio de índices del problema, aunque también presenta la subdivisión en workGroups, son sólo distintas formas de expresar lo mismo, alguna más conveniente que la otra de acuerdo a la aplicación.

Cabe aclarar que el proceso de ejecucion del kernel no es inmediato al momento de encolarlo sobre un determinado contexto, es más, en sí ninguna operación que se encole tiene ejecución instantánea, sino que depende de los tiempos de carga sobre la cola, y de las operaciones que se hayan encolado previamente sobre la misma. Existen campos como *Blocking and Non-Blocking Enqueue API calls* en donde se indica si se desea que sólo se retorne el control de la aplicación luego que se haya finalizado la tarea pertinente, o bien el control se retorne inmediatamene después que la operación se encoló sin importar el estado actual de la tarea. De esta forma por

ejemplo se puede procesar algo sobre el host mientras el device se encuentra ocupado realizando alguna operación, finalizada la ejecución sobre el device se vuelve a operar con él.

Por último se procesa una operación de lectura sobre el buffer del dispositivo para retornar el resultado esperado para finalmente liberar los recursos solicitados.

4 CASO DE ESTUDIO

Dada la extensión del trabajo propuesto, sólo se analizará el caso del producto Matriz-Matriz, puesto que resulta un ejemplo sencillo que incorpora muchos de los contenidos teóricos vistos anteriormente.

El hardware que se dispone presenta:

1. GPU/GPGPU: NVIDIA Geforce 210 (1 unidad) + NVIDIA Tesla C1060 (2 unidades).
2. CPU: Intel Core i7 950 @ 3.07 Ghz.

Se utilizó el SO Fedora 12, con la versión de kernel 2.6.32.16-141, los drivers para las GPU y GPGPU's provistos por NVIDIA versión 256.35 y como compilador gcc 4.4.4.

4.1 Producto Matriz-Matriz

Como se ha indicado anteriormente, una primera aproximación a la operación de interés tendrá en cuenta para las pruebas sólo el caso de matrices con disposición de sus datos en memoria global, que como se ha comentado anteriormente, tiene una latencia de entre 400-600 ciclos, o de unas 100-150 veces la de la memoria local (NVIDIA (2009b)).

Básicamente lo que realiza el código es tomar las coordenadas de cada valor a computar en la matriz resultado y realizar un producto vectorial entre el vector fila en la matriz A y el vector columna de B que se corresponden a su posición en el espacio de índices.

El código del kernel se muestra en el Listing 1. Las convenciones sobre las dimensiones de las matrices con las que se trabajaran en los distintos kernels que se presentarán son las mismas y se expresan a continuación: la matriz A posee M filas y N columnas, mientras que la matriz B posee N filas y P columnas, con lo cual se obtiene una matriz resultante C de M filas y P columnas.

Listing 1: Multiplicación de Matrices con memoria Global.

```
#define bSize 16

__kernel void
matMult(__global scalarType* C,
                                __global scalarType* A,
                                __global scalarType* B,
                                int M,
                                int N,
                                int P,
                                __local scalarType* Asub,
                                __local scalarType* Bsub)
{
        int gidx = get_global_id(0);
        int gidy = get_global_id(1);
```

```
        scalarType Csub = .0;
        for(int i=0;i<N;i++) Csub += A[gidy*N+i] * B[i*P+gidx];
        C[gidy*P+gidx] = Csub;
}
```

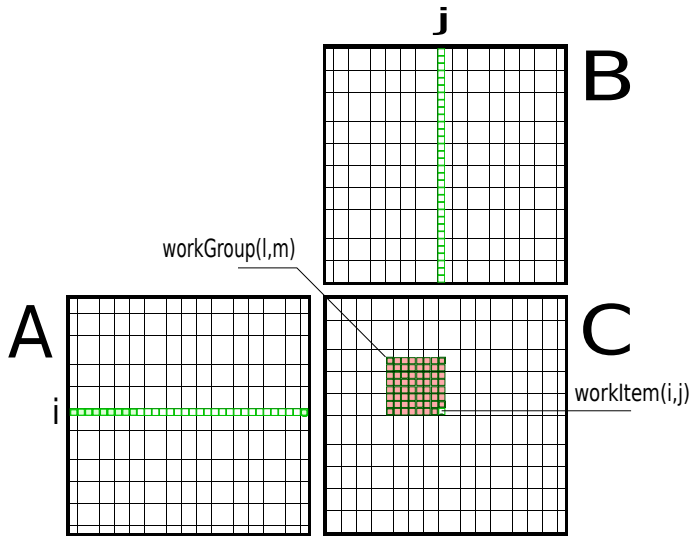

Figure 5: Cálculo M-M en memoria global. El bloque rosado representa a un workGroup, el elemento interior en verde un workItem en particular que necesita de los valores en los vectores en verde para obtener su resultado.

La operación en sí, se observa en la Fig. 5, en donde el área coloreada con rosado involucra aquellos workItems que serán ejecutados sobre el mismo computeUnit.

Además se observa como se efectúa el cálculo sobre cada workItem de ese workGroup en particular, y cabe nombrar que para el ejemplo se ha tomado un work group de 8x8, es decir, simétrico, aunque esto podría no ser así.

El problema con la implementación anterior es que la lectura de los datos sobre la memoria global presenta un rendimiento inferior a la lectura efectuada sobre la memoria local, por ende es natural la idea de utilizar esta última para realizar la operación. Esta claro que en matrices grandes no es posible simplemente derivar toda la estructura en la memoria local debido a la escasez de disponibilidad que tiene.

Para solucionar el problema anterior se busca que sobre cada workGroup se efectuen transacciones entre la memoria global y local para que luego, los workItems que residan sobre el mismo, pudiesen tomar sus datos de la local y sobre ésta realizar las operaciones pertinentes con el objetivo de aprovechar la performance extra que brinda. Lo anterior es válido sólo si el tiempo de cálculo puede hacer frente al tiempo de mapeo global-local, es decir, el tiempo que demora el traspaso de datos de memoria global a local.

Una solución posible se conoce en la bibliografía como el *tiles method*, que describe una forma de computar la matriz con el objetivo de aprovechar muchas de las ventajas de la memoria local y de los accesos coalesced del warp.

Lo anterior se observa en el Listing 2, en donde se observa como bloques del mismo tamaño que los workGroups de las matrices A y B, son ubicados en memoria local para luego, realizar productos vector-vector según corresponda y finalmente guardar un resultado parcial sobre cada workItem.

En el paso siguiente, se toman otros dos bloques sucesivos no solapados (uno en dirección

horizontal para la matriz A y uno vertical para la B), se efectúa la misma operación de producto vectorial y se actualiza el contenido del workGroup mediante una suma. El proceso se repite hasta que en esa sección de filas y columnas no quepa otro elemento a ser cargado en la local.

Listing 2: Multiplicación de Matrices con memoria Local.

```
#define bSize 16

__kernel void
matMult(__global scalarType* C,
                                __global scalarType* A,
                                __global scalarType* B,
                                int M,
                                int N,
                                int P,
                                __local scalarType* Asub,
                                __local scalarType* Bsub)
{
        int gidx = get_group_id(0);
        int gidy = get_group_id(1);
        int lidx = get_local_id(0);
        int lidy = get_local_id(1);
        int aBegin = gidy*N*bSize;
        int aStep = bSize;
        int aEnd = aBegin+N-1;
        int bBegin = gidx*bSize;
        int bStep = P*bSize;
        int baseL = lidy*bSize+lidx;
        int baseGA = lidy*N+lidx;
        int baseGB = lidy*P+lidx;
        scalarType Csub = .0;
        for (int i = aBegin,j = bBegin; i < aEnd; i+=aStep,j+=
           bStep) {
             Asub[baseL] = A[i+baseGA];
             Bsub[baseL] = B[j+baseGB];
             barrier(CLK_LOCAL_MEM_FENCE);
    #pragma unroll bSize
             for (int j=0;j<bSize;j++) Csub += Asub[lidy*
                bSize+j]*Bsub[j*bSize+lidx];
             barrier(CLK_LOCAL_MEM_FENCE);
        }
        C[gidy*bSize*P+gidx*bSize+lidy*P+lidx] = Csub;
}
```

Como se observa en el Listing 2, existe cierta restricción sobre la memoria local en la ejecución causada por la función *barrier*, que indica una sincronización entre los workItems de un workGroup. De lo anterior nace la pregunta de cómo la arquitectura CUDA asegura que estos workItems se ejecutan en tiempos similares para no tener retardo excesivo. La respuesta viene de la mano de los recursos y las asignaciones que la misma realiza, esta se asegura de

proveer a todos los workItems del mismo workGroup recursos con el objetivo de cumplir las restricciones temporales y evitar tiempos de espera excesivos.

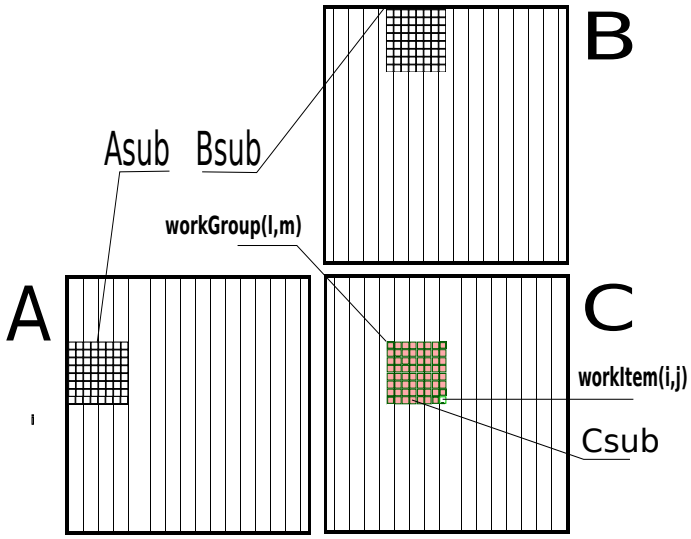

Figure 6: Cálculo producto Matriz-Matriz en memoria local utilizando el Tile's Method.

Lo anterior se presenta gráficamente en la Fig. 6, en donde se observan con remarcado negro los bloques que se van cargando en la memoria local, en color rosado el workGroup seleccionado y por ultimo un workItem en particular.

De los contenidos teóricos previos se tienen que para tener el máximo número de workGroups por computeUnit, la memoria local por cada computeElement no puede tener un tamaño mayor a 2 Kbytes, lo que en el cálculo matriz-matriz con el criterio de los bloques cuadrados nos impone un limite de 1 Kbytes para cada bloque, lo que resulta en bloques con dimensiones máximas de $16x16x4$ bytes $= 1024$ bytes para matrices con tipos de datos con alineamientos de 4 bytes, por ejemplo int, float, entre otros. Con esta elección, se tiene la mayor cantidad de workGroups activos posibles y además la memoria local se encuentra totalmente ocupada, como resultado se espera que el ancho de banda resultante de una implementación con estas características sea mayor que aquella obtenida utilizando bloques de otros tamaños, teniendo en cuenta que la cantidad de registros utilizados no supera la máxima de 16 impuestos por el hardware que se posee.

Para el caso de elementos con 8 bytes, como por ejemplo double, se tiene entonces $16x16x8 = 2048$ bytes por cada bloque, luego los bloques de las matrices A y B ocupan 4 Kbytes, por ende sólo se pueden disponer al momento de 4 workGroups activos por computeUnit,es decir, una eficiencia del 50% con respecto al caso anterior, manteniendo la cantidad de workItems en su capacidad máxima. Recordar que también se deben tener en cuenta la cantidad de registros utilizados por el kernel, tema que será tratado más adelante cuando se realicen las pruebas.

Ahora bien, como se ha introducido anteriormente existen operaciones en OpenCL que nos permiten acortar el rango de operación a nivel de bits, con lo cual el rendimiento obtenido teóricamente mejoría si se tiene en cuenta todos los cálculos que se deben realizar, más en este caso de *Algebra Lineal* con matrices o vectores de grandes dimensiones. En particular, si estas operaciones se aplicaran sobre los lazos iterativos (for) internos, la cantidad de cálculos resultaría enorme, y es aquí donde el rendimiento podría resultar beneficiado. Esto en OpenCL se realiza agregandole a las opciones de compilación la bandera *cl-mad-enable*

y luego utilizar las operacion built-in provistas por OpenCL. Es necesario corroborar que los datos se encuentren dentro de los límites impuestos por la precisión de 24 bits, en general el comportamiento en caso de no cumplir tal condición es propia de la implementación OpenCL que se disponga.

Tambien en general se utiliza *cl-fast-relaxed-math* para permitirle al compilador realizar optimizaciones agresivas según el lo crea conveniente, reduciendo el nivel de precisión con el que realiza las cuentas en operaciones de coma flotante.

Las operaciones optimizadas en general se traducen a una o más operaciones nativas del dispositivo que en su conjunto presentan un rendimiento mayor a la misma operación implementada sin el prefijo *_native*. Con respecto a la precisión en general, esta definida por la implementación de OpenCL que se este utilizando.

Con respecto al cálculo repetitivo sobre los lazos for, por ejemplo, de los índices de vector, en general es buena práctica estudiar cuando resulta práctico evitar cálculos repetitivos a costa de disponibilidad de registros, o bien cuando es conveniente guardar ciertos valores con el objetivo de no repetir cálculos.

Teniendo en cuenta lo anterior modificamos el código como se muestra en el Listing 3.

Listing 3: Multiplicación de Matrices con memoria Local.

```
#define bSize 16

__kernel void
matMult(__global scalarType* C,
                                __global scalarType* A,
                                __global scalarType* B,
                                int M,
                                int N,
                                int P,
                                __local scalarType* Asub,
                                __local scalarType* Bsub)
{
  int gidx = get_group_id(0);
        int gidy = get_group_id(1);
        int lidx = get_local_id(0);
        int lidy = get_local_id(1);
        int aBegin = mul24(mul24(gidy,N),bSize);
        int aStep = bSize;
        int aEnd = aBegin+N-1;
        int bBegin = mul24(gidx,bSize);
        int bStep = mul24(P,bSize);
        int baseL = mad24(lidy,bSize,lidx);
        int baseGA = mad24(lidy,N,lidx);
  int baseGB = mad24(lidy,P,lidx);
        scalarType Csub = .0;
        for (int i = aBegin,j = bBegin; i < aEnd; i+=aStep,j+=
           bStep) {
                Asub[baseL] = A[i+baseGA];
                Bsub[baseL] = B[j+baseGB];
```

```
                barrier(CLK_LOCAL_MEM_FENCE);
                #pragma unroll bSize
                for (int j=0;j<bSize;j++) Csub = mad(Asub[lidy*
                   bSize+j],Bsub[j*bSize+lidx],Csub);
                barrier(CLK_LOCAL_MEM_FENCE);
        }
        C[gidy*bSize*P+gidx*bSize+lidy*P+lidx] = Csub;
}
```

Ahora bien, hasta aquí se han aprovechado algunas de las características principales de la arquitectura pero no todas. La arquitectura dispone de unidades especiales que se encargar de optimizar ciertas tareas en particular, especialmente se podrían nombrar las operaciones con vectores y aquellas sobre imágenes. Entonces, uno podría sin mayores modificaciones al código anterior utilizar estas bondades con lo cual operaciones secuenciales como el producto punto realizado por el *for* fueran realizadas por el dispositivo en forma paralela, por ejemplo si se tuvieran m enteros y se requiriera multiplicarlos por un determinado valor se requerirían m multiplicaciones, ahora si dispusieramos de una estructura de datos al estilo *intm*, donde representa un vector de m componentes de tipo entero, luego sólo se necesitaría una operación, lo cual es una ganancia interesante más aún el tipo de datos con los que se trabaja son float (pues se conoce su carga computacional con respecto a otros tipos mas sencillos como por ejemplo, enteros).

Dada la extensión del texto, las versiones mediante utilización de imágenes o vectores no serán presentadas.

5 RESULTADOS

5.1 Presentación de la formula de cálculo del ancho de banda

El ancho de banda, que se define como la cantidad de datos que pueden ser procesados por determinado dispositivo por unidad de tiempo, es un indicador que permite medir y comparar los rendimientos de las distintas implementaciones que se han propuesto. En general, dos tipos de ancho de banda se utilizan: el teórico, aquel obtenido utilizando características físicas y lógicas del dispositivo, y el efectivo, aquel medido al momento de las pruebas.

Teniendo en cuenta lo anterior, la metodología a seguir en las pruebas será la siguiente: se evaluará la banda efectiva de cada implementación mediante la fórmula que se presenta en NVIDIA (2009b). Teniendo en cuenta que B_e representa al ancho de banda efectivo de la ejecución, B_r y B_w a la cantidad de elementos leidos y escritos por el kernel respectivamente, k la cantidad de bytes para la representación del tipo de los elementos de la matriz y T el tiempo total de ejecución. La unidad de medida será [Gbytes/s], entonces:

$$B_e = (((B_r + B_w) * k)/(1024^3))/T \tag{1}$$

5.2 Consideraciones para los cálculos en double, float y half

Las pruebas se efectuarán variando el tamaño de los workGroups sobre los algoritmos desarrollados previamente. Además para las pruebas se añadio una porción de código que cambia los tipos de datos pasando por half, float y double según corresponda.

Las implementaciones de prueba se escogieron no sólo con el objetivo de comprobar los rendimientos de las diferentes implementaciones propuestas, sino también de funciones *MAD* y

enrrolamiento de loops (*unrool*) provistos por OpenCL.

Finalmente se presentarán primero los resultados obtenidos con la utilización de double y luego de float, siendo el tipo de datos half no soportado por los dispositivos que se tienen para las pruebas.

5.3 Implementación multiGPU

El gabinete de pruebas del que se dispone contiene 2 GPGPU's NVIDIA Tesla C1060 con una capacidad de 30 computeUnits por dispositivo, además de una NVIDIA 210 con 2 computeUnits. Luego, un balance de cargas no puede simplemente reducir la dimensión de la matriz equitativamente puesto que se estaría encomendado demasiada carga a la gráfica con menor capacidad que, por supuesto, presenta pocos computeUnits como para hacer frente a las Tesla. Es por ello que en la implementación propuesta se desarrollo una función que básicamente dado un conjunto de dispositivos identificados en un contexto, la cantidad de computeUnit que dispone, el total de computeUnits de todos los dispositivos, el soporte o no de los tipos de datos especificados en el cálculo y las dimensiones de workGroups con los que se este operando, distribuye la carga en cada placa. Con respecto a la inclusión en el balance de los tipos de datos esto es así puesto que si se requiere computar resultados en half o double, luego debe asegurarse que los dispositivos lo soporten, de no ser así no se utilizan para el cálculo.

Es necesario introducir la cuestión que para la ejecución de la operación $AxB = C$, donde A,B,C denotan matrices de MxN, NxP y MxP respectivamente, se divide a la matriz A en submatrices, cada una de las cuales presenta cierta cantidad de filas de A de acuerdo al coeficiente de capacidad de cómputo obtenido con la función presentada en el parrafo anterior. Luego la matriz B es copiada entera a todas los dispositivos que se dispongan, como resultado, existe un alta carga del bus PCI-E a medida que la dimensión de las matrices aumenta. El resultado se obtiene al ensamblar los resultados parciales de cada dispositivo. Para trabajos futuros sería interesante medir y comparar los tiempos de copia de valores iniciales para realizar los cálculos con respecto a la reducción de operaciones a realizar por cada dispositivo al disminuir la dimensión de la matriz A y poder así tener un valor de medición mas representativo.

Para la medición de tiempos de ejecución (ATI (2010)), como se sabe, cada dispositivo cuenta con una cola de comandos y por ende, en estas colas irán aquellas operaciones de carga/descarga que deban ser computadas por ese dispositivo en particular. Luego, antes de medir tiempos se impone a todos los dispositivos asociados al contexto que liberen su cola de comandos, para posteriormente asociarle a cada uno el espacio de índices a computar. Luego, se toma un objeto evento y se solicita a OpenCL que no retorne a CPU sin haber realizado la tarea, esto para cada dispositivo, claro esta que la ejecución se realiza en paralelo, es decir, al no tener comandos previos en la cola de comandos todos los dispositivos comienzan su ejecución en tiempos similares, las diferencias pueden ser sutiles y suelen darse en la forma en que alocan recursos. Luego, puede tomarse como el tiempo de ejecución de la tarea completa (sin tener en cuenta las cargas de datos iniciales en dispositivos, esto es, sin medir los tiempos de carga de cada porción de A y la matriz completa B) como el mayor valor de todos los tiempos retornados por cada dispositivo, puesto que el objeto evento (uno por cada dispositivo) guarda en su estado el tiempo de comienzo y finalización del procesamiento.

5.3.1 Matriz-Matriz double

Antes de avanzar con los resultados se definen dos variaciones de la implementación sobre la memoria local con el objetivo de comprobar rendimientos cuando se utilizan funciones MAD

(localMAD) o no (local).

Con respecto a las implementaciones en CPU se utilizaron dos: la primera es una versión sin optimización alguna que realiza el producto Matriz-Matriz mediante un triple lazo for, y la segunda emplea las funciones dgemm y sgemm de las BLAS provistas por la API de Intel MKL, empleando para los cálculos los 4 cores de microprocesador que se dispone.

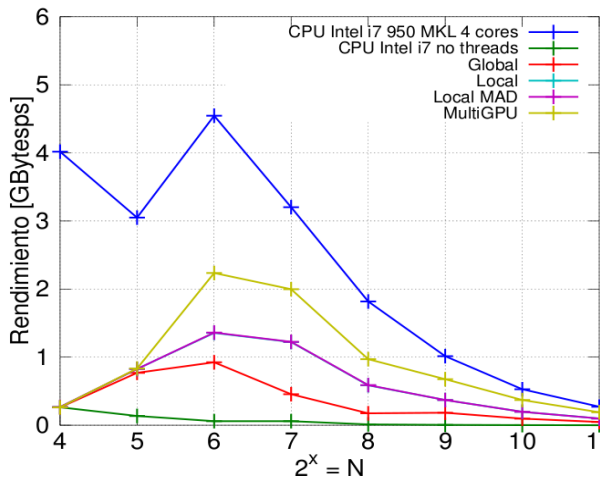

Figure 7: Producto M-M, 4x4 = 16 workItems por workGroup [double].

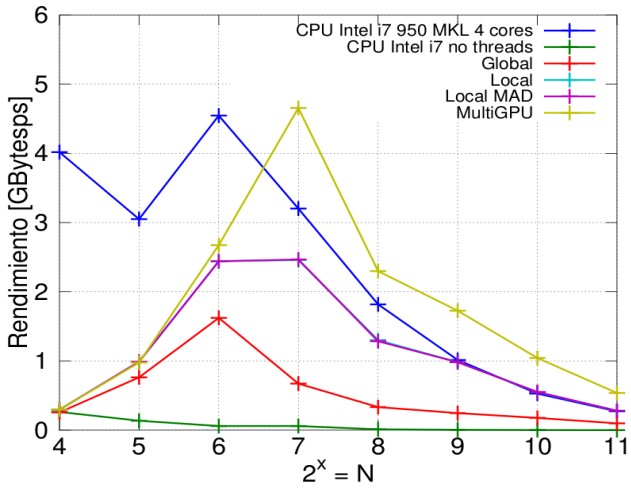

Figure 8: Producto M-M, 8x8 = 64 workItems por workGroup [double].

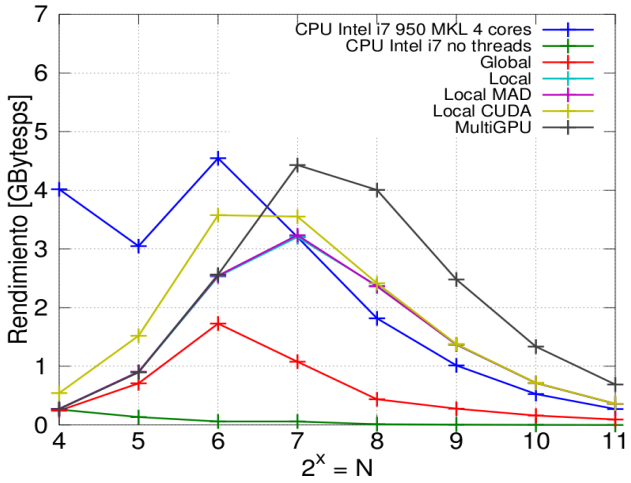

Figure 9: Producto M-M, 16x16 = 256 workItems por workGroup [double].

Al analizar la gráfica 7 se observa como las implementaciones en local tienen ventaja sobre aquella en global, característica que típicamente se presentará en la práctica y que se deriva de las latencias de lecturas sobre las distintas memorias, la lectura de datos dentro del mismo segmento en la memoria global, entre otras.

La Fig. 8 muestra el rendimiento utilizando workGroups de 64 workItems, en este caso la utilización de memoria local se comienza a aprovechar con mayor utilidad (aunque no la máxima) y como resultado se tiene un incremento de rendimiento con respecto al caso anterior. Ahora bien, el tipo de double esta alineado a 64 bits, o 8 bytes, y realizando un simple promedio se tiene que para realizar los cálculos cada processingElement dispone de 2 Kbytes de memoria

local para realizar sus cuentas. En realidad cada processingElement utiliza $64 * 8 = 512$ bytes, por lo tanto no la esta utilizando de manera óptima.

Finalmente en la Fig. 9 se presenta el caso de workGroups con 256 workItems cada uno donde la memoria local se aprovecha se aprovecha de manera más eficiente, en efecto, cada processingElement requiere de 4 Kbytes, con lo cual sólo 4 workGroups se tiene activos por computeUnit, como contrapartida tenemos un uso óptimo de la memoria local lo que se traduce a un incremento notorio.

La próxima potencia de dos para los kernels presentados anteriormente es 32, lo que resultaría en workGroups de 1024 workItems, lo cual como se ha visto, excede el límite máximo impuesto por la arquitectura, y por ende 16 es la máxima potencia de dos con el cual se puede trabajar el método de bloques cuadrados.

Para corroborar el correcto funcionamiento de la implementación se utiliza *computeprof*, una aplicación provista por NVIDIA que permite obtener ciertas características del código implementado. Los resultados obtenidos muestran que no existen accesos sin coalisión ni divergencias de workItems, que se cuentan con 15 registros utilizados por processingElement, entre otros.

Luego dado que se esta en presencia de 15 registros por processingElement, cada workGroup presenta $16x16x15 = 3840$ registros lo que esta claramente por debajo del límite de 16384. Entonces en teoría se pueden tener simultáneamente hasta 4 workGroups activos por computeUnit. Suponiendo que ahora se tienen 17 registros por processingElement, tener 4 workGroups activos violaría la cantidad de registros máxima, entonces se podría tener sólo 3 workGroups activos con un total de 768 activos. Como conclusión, se ha perdido 1/4 de beneficio a nivel de processingElements.

La versión de MKL comienza con un interesante rendimiento para matrices de hasta 128 elementos, pero luego aquellas implementaciones sobre GPGPU toman la delantera.

Con respecto a la implementación multiGPU se observa un rendimiendo similar o inferior a aquella monoGPU para matrices con dimensiones por lado de hasta 128 elementos, luego de este valor comienza a ser superior porque inicialmente el tamaño de la matriz no compensa los tiempos de carga host/device y descarga device/host, como se habló anteriormente, los tiempos de carga deben amortizarse con los tiempos de cómputo en el dispositivo. Y luego mantiene la relación de 2:1 conforme las dimensiones de las matrices aumentan.

La versión de CUDA resulta superior para matrices con dimensión de lado de hasta 128 elementos, luego coincide con la versión de OpenCL.

Otra particularidad a tener en cuenta es que a mayor potencia de dos se tenga como valor del lado del workGroup, más limitaciones sobre el tamaño de las matrices se tiene, puesto que las implementaciones son totalmente dependientes de esto. En general una solución sencilla es un redimensionado al próximo múltiplo del bSize.

Es importante comentar, para futuras pruebas, el hecho de que no se produzcan serializaciones en los cálculos dentro del workGroup dado que todos los workItems del warp siguen el mismo camino de ejecución, esto es, no hay ninguna sentencia condicional que pueda producir que distintos workItems puedan realizar distintas operaciones dado ciertos casos en particular. Cabe recordar el descenso abrupto de rendimiento presente en la arquitectura en presencia de estos casos.

Con respecto a las implementaciones utilizando funciones *mad* y aquellas operando sobre 24 bits no parecen agregar mejora alguna por lo cual su uso no es recomendado puesto que nublan el código y no dan ventaja alguna.

5.3.2 Matriz-Matriz float

La Fig. 12 muestra el rendimiento utilizando workGroups de 256 workItems, el tipo de datos float presenta una precisión de 4 bytes.

Cada processingElement utiliza entonces $256 * 4 = 1024$ bytes por submatriz, lo que deriva en un uso de memoria local de 2 Kbytes. Al tener 8 computeElement por computeUnit se tienen los 16 Kbytes aprovechados a la máxima eficiencia. Aunque como para el caso de doubles, se tienen 4 workGroups activos por computeUnit dada la cantidad de registros utilizados por el kernel.

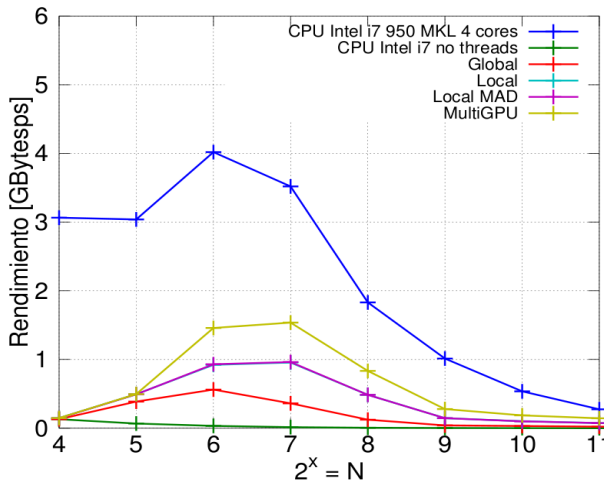

Figure 10: Producto M-M, 4x4 = 16 workItems por workGroup [float].

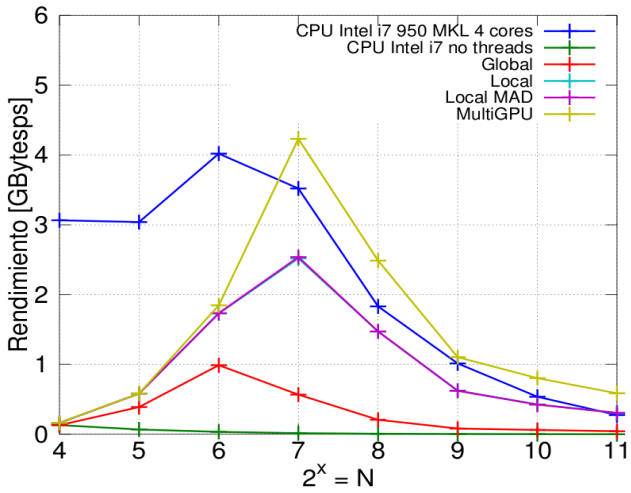

Figure 11: Producto M-M, 8x8 = 64 workItems por workGroup [float].

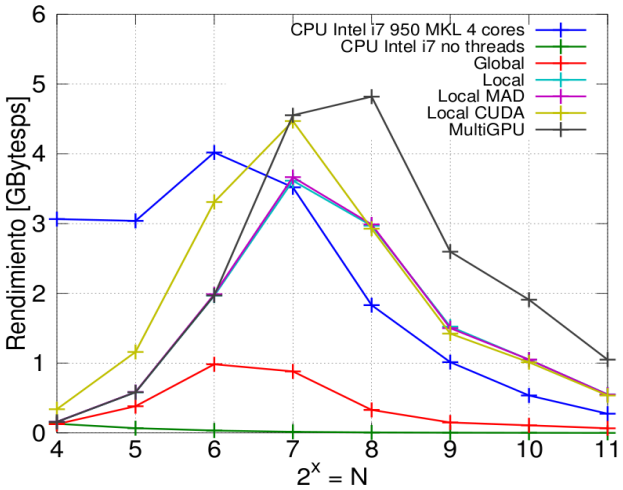

Figure 12: Producto M-M, 16x16 = 256 workItems por workGroup [float].

Además se observan los resultados obtenidos con la misma operación implementada en CUDA. Inicialmente parece existir una ventaja con respecto a la versión de OpenCL, para matrices con 64 elementos por lado existe una diferencia de 1.5 GBps, mientras que con 128 de 1GBps, luego la relación parece estacionarse y la versión de OpenCL resulta ser, por poca diferencia, superior a la de CUDA.

Los resultados de comparaciones entre memoria global y local, implementación multiGPU

y MKL resultan similares al caso de double, aunque la versión de MKL resulta con un mayor ancho de banda para el caso de double que en float, como es lógico, mientras que las implemetaciones sobre GPGPU presentan mayor rendimiento para datos con tipo float, aunque la diferencia no es importante.

6 CONSIDERACIONES

6.1 Predicados

Un control de flujo de la forma IF-ELSE (además de introducir una posible división de caminos de ejecución) no tiene la posibilidad de ejecutar ambos caminos en paralelo como un predicado, entonces si es posible, deberían utilizarse estos últimos sin excepción.

En general en este punto pueden hacerse dos consideraciones básicas:

1. Como se ha visto en general los controles de flujo en las GPU son lentos y por ende puede resultar más conveniente tratar los distintos bloques condicionales en distintos kernels (recordando la pobre capacidad de las GPGPU's para resolver estas condiciones puesto que no tienen suficientes elementos de control de flujo en hardware). Esto se contrapone con lo que en general se hace en la CPU, donde es conveniente utilizar controles de flujo para reducir la carga en instrucciones que no se utilicen.

2. Las CPU poseen una enorme cantidad de registros por lo cual es conveniente guardar datos para evitar cálculos repetitivos, en las GPU esto no es así, y es necesario ahorrar los registros disponibles, puesto que en general posee gran velocidad es conveniente realizar algunos cálculos de nuevo que guardar datos en registros.

6.2 Pinned Memory

Un aspecto interesante a considerar es la demora (poco ancho de banda) que existe en el bus PCI-E para la transmisión de datos entre *host-device*, del órden de 4 Gbytes/s máximo teórico en cada dirección para la generación 1 de PCI-E 16x, mientras que la segunda version dobla el rendimiento, esto es a 8 Gbytes/s en cada dirección, en comparación con el ancho de banda del acceso del *dispositivo* a su memoria del órden del Tbytes/s. Una opción interesante que provee la arquitectura CUDA es la *pinned o page-locked memory*, que básicamente es una memoria que tiene la particularidad de tener el mayor ancho de banda posible para transmisiones entre *host-device* dado a que es una memoria que no puede ser paginada, esto es, las páginas en que son divididas un programa no pueden ser enviadas a disco por el controlador de memoria del sistema operativo para cargar otras, y por ende todas sus páginas se encuentra siempre disponibles. Hay que recalcar que por defecto *no se aloca memoria pinned* si no es explicitamente solicitado mediante el campo CL_MEM_ALLOC_HOST_PTR durante la declaración y definición de objetos memoria en OpenCL, así el driver mismo se encarga de realizar el alocado.

Entre las ventajas de trabajar con este tipo de memoria podemos nombrar (NVIDIA (2009b)) el óptimo ancho de banda entre *host-device*, la posibilidad de algunos dispositivos de poder realizar transmisiones de datos concurrentemente con la ejecución del kernel y, también para algunos dispositivos, la posibilidad de *mapear* la *pinned memory* al espacio de índices del dispositivo, y así evitar toda solicitud de porciones de memoria sobre el dispositivo y de cargar todos los datos a procesar sobre el mismo, sino que las las transmiciones se van realizando a medida que el kernel accesa la memoria mapeada.

6.3 Operaciones paralelizables

Es un aspecto primordial que indica cual beneficioso resultará la paralelización, en general de acuerdo a su resultado se realiza o no el esfuerzo de paralelizar alguna tarea. Consiste en separar el código en dos partes, una secuencial y otra paralela, donde se ubican los tiempos requeridos por las operaciones secuenciales y paralelas respectivamente. La proporción de código paralelizable será la de mayor importancia, puesto que a mayor valor se puede ver que optimizando el mismo se logra un speedup notorio en comparación con el optimizado de código secuencial. Para aclarar dudas se propone la situación en donde el 70% del tiempo de la aplicación es serial y el restante 30% es paralelo, con un speedup de $50x$ sobre la porción paralela reduciremos el tiempo de cálculo de la porción paralela a 0.6% lo que da un speedup sobre el cálculo total de 1.4. Por el contrario si el 95% del tiempo de ejecución fuera paralelizable y se aplicara un speedup de $50x$ a esa porción de código luego se obtendría un speedup de la operación total de 14. Se concluye a partir de lo anterior que a medida que la porción de código paralelizable es mayor, optimizarla resulta de mayor beneficio.

Visto desde otro punto de vista, no todo es conveniente procesarlo en la GPU, es conveniente realizar un balance sobre cuanto cuesta enviar los datos al dispositivo para ser procesados previo a su retorno, a lo menos el factor debe ser O(N) como lo es en un producto matricial (mientras que en una suma vectorial se tiene O(1), lo que indica que es indiferente procesarlo o bien en la CPU o en la GPU, a menos que se utilicen técnicas de optimización). Estos números se obtienen haciendo el cociente entre las operaciones a realizar y el tamaño de los objetos que tienen que ser derivados al dispositivo, por ejemplo, para el producto matricial se tienen cerca de $2N^3 - N^2$ operaciones que dado un N grande se tiene $O(N^3)$, sobre $3N^2$ transferencias u $O(N^2)$, por ende se obtiene el O(N).

6.4 NVIDIA Tesla C1060

La GPGPU Tesla C1060 (NVIDIA (2009c)) presenta un bus PCI-E v2.0 basada en la GPU Tesla T10 ideada con exclusividad para soluciones de computación de alta performance (HPC). Posee una performance de procesamiento teórico máximo de 933 GFlops/s, 4 Gbytes de memoria GDDR3 con un ancho de banda de 102 Gbytes/s. La organización de núcleos interna consta de 30 computeUnits, cada uno con un conjunto de 8 processingElements que en su totalidad conforman un conjunto de 240 procesadores. Cada núcleo trabaja a una frecuencia de 1296 GHz. Con respecto a la memoria RAM, trabaja a 800 Mhz, con una interface de 512 bits y los 4 Gbytes de memoria RAM se componen por 32 chips de 32 Mbytes GDDR3 136-pines. La precisión para números flotantes tanto simples como dobles responden al estándar IEEE 794.

6.5 Introducción a Fermi

Básicamente la arquitectura Fermi (NVIDIA (2010)) presenta 16 computeUnits, cada uno con 32 processingElements, 16 unidades de carga/descarga y 4 unidades especiales (SFUs).

Además incorpora dos unidades SIMT y dos unidades de instruction dispatch.

Cada computeUnit puede tener 1536 threads concurrentes para cubrir las latencias de cargas con la memoria DRAM. A medida que los workGroups van completando su ejecución liberan al computeUnit y el scheduler le asigna un nuevo workGroup. La unidad SIMT crea, maneja, organiza y ejecuta thread concurrentes en grupos de 32 workItems denominados warps, identicamente a la arquitectura TESLA.

La comunicación entre la CPU y la GPU ocurre via el bus PCI-E, en un modo bidireccional.

La arquitectura GPU balancea su poder de cómputo mediante controladores de memoria DRAM paralelos destinados a optimizar el ancho de banda de la misma. Las GPU FERMI utilizan 6 interfaces de alta velocidad GDDR5 DRAM de 64 bits de longitud de palabra. Además poseen direcciones de 40 bits capaces de direccionar hasta 1Tbyte de espacio de direcciones tanto en CPU como en GPU.

Se tiene por computeUnit una memoria on-chip de 64 Kbytes de los cuales 48 Kbytes se pueden corresponder con la shared y 16 Kbytes al caché L1 o a la inversa, y todos los computeUnits comparten otra caché L2 con un total de 768 Kbytes, esta última hace de puente entre las interfaces de las 6 memorias de 64 bits y la interface PCI-E.

Incluye protección de memoria ECC para mejorar la integridad de los datos, puede detectar errores sobre 1 y 2 bits, pero sólo corregir si existe 1 bit de error.

FERMI implementa el conjunto de instrucciones PTX2.0 (parallel thread execution).

La SFU ejecuta instrucciones de punto flotante de 32 bits para aproximaciones rápidas de reciprocos, raices cuadradas reciprocas, funciones trascendentales, entre otras. Las aproximaciones son precisas incluso con 22 bits de mantisa.

En dispositivos TESLA con computeCapability de 1.x se pueden correr kernels distintos pero no simultáneos, mientras que en FERMI, utilizando CUDA 3.1 se pueden correr hasta 16 kernels concurrentes distintos.

Además la arquitectura provee de funciones al estilo de printf() y además el soporte de recursiones que es TESLA no estaban implementadas, proveyendo al programador de hasta 1 Kbytes de memoria por workItem en la cola de recursión.

Como un dato extra se tiene que en las implementaciones actuales, los datos leidos por el warp se pierden una vez efectuada la operación solicitada, en la arquitectura FERMI en cambio existe una especie de caché que permite su reutilización para futuros warps.

7 CONCLUSIONES

Inicialmente puede resultar confuso programar en OpenCL puesto que se trabaja a un nivel más bajo que en CUDA, sin un conocimiento relativamente avanzado de la arquitectura resulta imposible realizar hasta el más simple desarrollo. La ventaja que ofrece es la de proveer de un código totalmente personalizado, cuyos componentes estan especificados, no como en CUDA que se utilizan funciones provistas por la API que poseen una programación de código cerrado.

La idea de una introducción corta pero completa a la arquitectura CUDA tuvo como objetivo formar una base teórica sólida para poder desarrollar implementaciones eficientes. Ejemplos sencillos se utilizaron para mostrar numéricamente algunos de los temas propuestos.

Como se ha analizado en las pruebas, la arquitectura CUDA parece una solución interesante para aquellos que necesiten un poder de cómputo intensivo, aunque la ventaja con respecto al mismo cálculo en una versión optimizada de CPU realmente no resultó como se esperaba (al menos para la operación propuesta).

La limitación de la memoria local y la cantidad es registros es notoria, puesto que aguardan estrecha relación con el rendimiento, punto que fue mejorado en la nueva arquitectura FERMI.

Los resultados obtenidos para doubles reflejan un buen comportamiento de la arquitectura sobre este tipo de datos, en general observando las gráficas de las pruebas si bien se tiene un descenso de rendimiento al utilizar este tipo de datos, producen resultados satisfactorios.

Actualmente se poseen los kernels de otras operaciones de Algebra Lineal como son Matriz-Vector, producto punto, norma vectorial, suma de vectores y combinaciones lineales, pero dada la extensión del trabajo y el tiempo requerido para efectuar las pruebas, no se han podido incluir

en este artículo. Es necesario aclarar que las operaciones actuales sólo involucran matrices llenas, recientemente se esta comenzando a desarrollarlas en su versión sparse.

8 AGRADECIMIENTOS

Este trabajo recibió apoyo financiero desde el Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET, Argentina, grant PIP 5271/05), Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, Argentina, grants PICT 01141/2007, PICT 2008-0270 "Jóvenes Investigadores"), y la Universidad Nacional del Litoral (UNL, Argentina, grants CAI+D 2009 65/334).

REFERENCES

ATI. Ati stream sdk v2.01 performance and optimization. 2010.

Group K. *The OpenCL specification. Version 1.0. Revision 48.* 2009.

Howard P.G. Gpu computing: More exciting times for hpc!. 2009.

Kirk D. and mei Hwu W. *Programming Massively Parallel Processors:A Hands-on Approach.* 2010.

NVIDIA. Nvidia opencl best practices guide. version 1.0. revision julio 2009. 2009a.

NVIDIA. Opencl programming guide for the cuda architecture. version 2.3. revision 27/8/2009. 2009b.

NVIDIA. Tesla c1060 computing processor board. bd-04111-001_v05 enero 2009. 2009c.

NVIDIA. Tuning cuda applications for fermi. 2010.