UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escuela Técnica Superior de Ingeniería Informática

PROYECTO FIN DE CARRERA

Programación de propósito general y paralela usando GPU.

Titulación

Ingeniería Técnica en Informática de Gestion

Presentado por

Jorge Navarro Domínguez

Dirigido y tutorizado por

Luis José Saiz Adalid

Valencia 6 de Septiembre de 2011

Índice de contenido

1 Introducción a las tarjetas gráficas

La historia de las tarjetas gráficas comenzó a finales de 1960, cuando los ordenadores necesitaban una nueva forma de plasmar la información que procesaban, utilizando monitores de vídeo en lugar de impresoras. Las primeras tarjetas gráficas se denominaban “tarjetas de vídeo” ya que sólo eran capaces de visualizar texto a 40x25 o 80x25 (40 u 80 caracteres x 25 líneas en pantalla). La aparición de los primeros chips gráficos, comenzó a dotar de capacidades gráficas a equipos dotados de bus S-100 o Eurocard.

El auge del ordenador personal y las primeras videoconsolas abarataron los costes de producción haciendo que estos chips se integrasen en las placas base de los ordenadores de la época.

Posteriormente, se sucedieron varias controladoras para gráficos (MDA, CGA, HGC, EGA, IBM 8514, MCGA, VGA, SVGA, XGA), siendo VGA (Video Graphics Array) la que más aceptación tuvo. Compañías como ATI y S3 Graphics fueron obligadas a trabajar sobre dicha tarjeta para mejorarla en resolución y número de colores, creando así el estándar SVGA. El estándar SVGA (Super Video Graphics Array) es capaz de alcanzar 2MB de memoria VRAM (Video Random Access Memory) con resoluciones de 1024x768 píxeles a 256 colores.

El gran boom de las tarjetas gráficas llegó en 1995 de la mano de compañías como Matrox, Creative, S3 y ATI fabricando las primeras tarjetas 2D/3D que cumplían con el estándar SVGA, pero incorporando funciones 3D. En 1997, 3dfx lanzó el chip gráfico **Voodoo** que, además de tener una gran potencia de cálculo, incorporaba efectos 3D como el Mip Mapping[1], Z-Buffering[2] o Antialiasing[3], entre otros. Después de **Voodoo**[4] se suceden una serie de lanzamientos de tarjetas gráficas como **Voodoo2** de 3dfx y las míticas **TNT**[5] y **TNT2**[6] de NVIDIA.

Las prestaciones de las tarjetas gráficas crecían de una forma exponencial, de tal forma que el puerto PCI donde se conectaban empezó a quedarse corto en ancho de banda y a generar cuello de botella, por lo que Intel desarrolló el puerto AGP que solucionaba dicho problema.

Desde 1999 hasta 2002, NVIDIA dominó el mercado de las tarjetas gráficas con la serie

1 Colecciones de imágenes de mapas de bits que acompañan a una textura para incrementar la velocidad de renderizado.
2 Parte de la memoria de una GPU que se encarga de gestionar la profundidad de las imágenes 3D.
3 Suavizado de bordes y disumulo de los bordes de los polígonos, con lo que se consigue una apariencia más realista.

GeForce. Durante este período las tarjetas gráficas mejoraron los algoritmos 3D que utilizaban, mejoraron también la velocidad de las memorias de las tarjetas gráficas y su capacidad, pasando de los 32MB de GeForce, hasta los 64MB y 128MB de GeForce 4.

Desde 2006 hasta la actualidad, NVIDIA y ATI (comprada en 2006 por AMD) se han repartido el mercado de las tarjetas gráficas con sus series de chips GeForce y Radeon, respectivamente.

Consolas como Sony PlayStation3 y Microsoft XBOX360 utilizan chips gráficos derivados de los más potentes aceleradores 3D.

Como es sabido, la tecnología avanza a un ritmo vertiginoso y actualmente las tarjetas gráficas están empezando a utilizarse para realizar tareas para las que en un primer momento no fueron diseñadas. Aprovechándose de su alta capacidad de núcleos de cálculo son capaces de realizar tareas en las que se necesitan calcular datos de forma paralela más eficientemente que los procesadores (CPU) más actuales a un coste mucho más económico.

4 Lanzada al mercado en 1998.
5 Lanzada al mercado a mediados de 1998.
6 Lanzada al mercado a comienzos de 1999.

2 Planteamiento del proyecto

2.1 Elección del proyecto

Desde pequeño he sido un gran aficionado a los ordenadores y más concretamente a los videojuegos. Dada esta afición que tengo, con relativa frecuencia consulto en internet páginas relacionadas con tarjetas gráficas y últimamente había leído varias veces la palabra GPGPU.

Por esta razón, cuando revisé las opciones de proyecto disponibles a mi alcance y ví que había un proyecto que utilizaba la misma palabra en el título, decidí investigar un poco al respecto.

En mi opinión, la información que encontré resultó ser fascinante, utilizar la tarjeta gráfica para desbancar hasta el momento a la unidad de computó estándar, la CPU.

También sabía que supondría un reto, ya que la tecnología utilizada era muy reciente y eso suponía que no habría demasiada documentación. Además, casi la totalidad de las fuentes donde me he documentado están escritas en inglés.

2.2 Objetivos generales del proyecto

El objetivo del proyecto es demostrar que se pueden utilizar las capacidades de cómputo de las tarjetas gráficas modernas para solventar algoritmos complejos en los que se precisa calcular muchas operaciones matemáticas de forma paralela, reduciendo considerablemente el tiempo de ejecución de los programas utilizados para tal fin y alcanzando una tasa mayor de operaciones en coma flotante por segundo.

Para demostrar esto se utilizará un algoritmo de multiplicación de matrices. Este algoritmo se escribirá en el lenguaje de programación C, para el programa que se ejecuta utilizando la CPU, y el lenguaje OpenCL utilizado para programar la tarjeta gráfica o GPU.

Posteriormente se registrarán los resultados y se comprobará que usando el programa escrito en OpenCL y ejecutado en la GPU, se obtiene un mayor rendimiento que el programa ejecutado en la CPU.

3 Introducción a GPGPU

GPGPU o **General-Purpose Computing on Graphics Processing Units** es un concepto reciente dentro de la informática que aprovecha la capacidades de cómputo de una GPU. La GPU es un procesador diseñado para realizar cómputos implicados en la generación de gráficos 3D interactivos. La utilización de la GPU para realizar estos cálculos ha sido posible gracias a la incorporación de etapas programables y una mayor precisión aritmética en el renderizado de los gráficos, que permite a los desarrolladores de software usar el flujo de procesamiento de la tarjeta gráfica para calcular datos que no forman parte de un gráfico.

Las características más reseñables de un GPU son su bajo precio en relación a su potencia de cálculo, un gran paralelismo y la optimización para cálculos en coma flotante. Estas características resultan atractivas para su uso en aplicaciones de ámbito científico, de simulación, compresión de vídeo y recodificación de vídeo, entre otras. Han surgido muchas implantaciones de simulaciones de fluidos, bases de datos y algoritmos de clustering, etc.

Las diferencias entre las arquitecturas de la GPU y la CPU, concretamente el acceso a memoria, es donde se localizan las mayores dificultades a la hora de programar los diferentes dispositivos.

Las CPU están diseñadas para el acceso aleatorio a memoria, lo que permite que se puedan utilizar punteros a posiciones arbitrarias en memoria, mientras que en la GPU el acceso a memoria es mucho más restringido. Tomaremos como ejemplo el procesador de vértices y el procesador de píxeles, ambos procesadores de la GPU. Mientras que el procesador de vértices usa un modelo “scatter”, donde el programa lee una posición predeterminada de la memoria y escribe en una o varias posiciones arbitrarias de la memoria, el procesador de píxeles usa un modelo “gather”, donde el programa puede leer varias posiciones arbitrarias, pero escribe solo en una posición predeterminada.

La persona encargada de diseñar el algoritmo a implementar en la GPU, deberá adaptar los accesos a memoria y las estructuras de datos a las características de la GPU. Para almacenar datos en la GPU se suele utilizar un buffer 2D, en lugar de utilizar una “textura”.

Se podría pensar que cualquier programa implementado en la GPU es más eficiente que su equivalente implementado en la CPU, pero esto no es del todo cierto. Las implementaciones no serán igual de eficientes en las 2 arquitecturas. Solamente los algoritmos con un alto grado de paralelismo, con estructuras de datos simples y con una

alta intensidad aritmética, son los que mayor beneficio obtendrán de una implementación en la GPU.

Al principio, el desarrollo de software GPGPU se hizo en lenguaje ensamblador. Posteriormente la Universidad de Stanford desarrolló una herramienta llamada BrookGPU[7], que es una extensión de ANSI C, que proporciona nuevos tipos de datos y operaciones automáticamente convertidas a una implementación que aprovecha la GPU sin intervención explícita por parte del programador. Actualmente existen herramientas creadas por las dos empresas líderes del mercado: ATI/AMD y NVIDIA. CUDA[8] por parte de NVIDIA y AMD Stream[9] por parte de ATI/AMD, siendo ambas SDK + IDE. Existe otra herramienta, que utilizaremos en este proyecto, llamada OpenCL[10]. Esta herramienta fue creada por Apple y está siendo mantenida por el Grupo Khronos[11].

Aunque las ventajas del uso de la GPU en ciertas aplicaciones son evidentes, no le faltan las críticas a esta nueva tecnología, en concreto las referidas a usar un procesador para fines completamente diferentes a los que se pensaba en el momento de diseñarlo. Otro argumento es la falta de precisión de los registros en coma flotante presentes en la GPU para muchas aplicaciones científicas, usualmente se utilizan 16 o 32 bits para representar un número real en la GPU, mientras que se utilizan 32, 64 o más bits en las CPU modernas. Otra crítica, es que debido a la rápida evolución que sufren las tarjetas gráficas, implementaciones de algoritmos que funcionan óptimamente en un modelo de la GPU, pueden dejar de hacerlo o no hacerlo tan óptimamente como se desearía, en un modelo posterior.

7 http://graphics.stanford.edu/projects/brookgpu/
8 http://www.nvidia.es/object/cuda_home_new_es.html
9 http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx
10 http://www.khronos.org/opencl/

4 OpenCL

4.1 Introducción a OpenCL

OpenCL (Open Computing Language) es una herramienta para escribir programas con paralelismo a nivel de datos y de tareas que puede ejecutarse tanto en CPUs, GPUs como en otros procesadores. OpenCL incluye un lenguaje de programación (basado en C99, que elimina cierta funcionalidad y se extiende con operaciones vectoriales), para escribir *kernels*[12], además de un API para definir y posteriormente controlar las plataformas y dispositivos a utilizar.

Usando OpenCL, un programador puede escribir programas de propósito general que se ejecuten en las GPUs sin necesidad de mapear los algoritmos en una API de gráficos 3D como OpenGL[13] o DirectX[14]. Por lo tanto, OpenCL provee una abstracción hardware de bajo nivel además de un entorno de trabajo para soportar "programación" y muchos de los detalles que serán expuestos del hardware subyacente.

Apple creó la especificación original y la propuso al Grupo Khronos para que se convirtiera en un estándar abierto y libre de derechos. El 16 de Junio de 2008 Khronos creó el Compute Working Group con representantes de CPU, GPU, procesadores embebidos y compañías de software. El grupo trabajó durante cinco meses para acabar los detalles técnicos de la especificación OpenCL 1.0 el 18 de Noviembre de 2008. La especificación fue revisada por miembros del grupo Khronos y aprobada para su liberación pública el 8 de Diciembre de 2008.

AMD ha decidido adoptar OpenCL en lugar de su API Close to Metal. El 9 de Diciembre de 2008, NVIDIA anunció su intención de añadir pleno soporte para la especificación OpenCL 1.0 en su GPU Computing Toolkit. El 30 de Octubre de 2009, IBM lanza su primera implementación OpenCL como parte de sus XL compilers.

La especificación OpenCL 1.1 fue ratificada por el grupo Khronos el 14 de Junio de 2010 y añade funcionalidad significativa para mejorar la flexibilidad, capacidad y rendimiento de la programación paralela.

11 http://www.khronos.org/
12 Funciones que se ejecutan sobre dispositivos soportados por OpenCL.
13 Open Graphics Library, API multilenguaje y multiplataforma para escribir aplicaciones que usan gráficos 2D y 3D.
14 API proprietaria de Microsoft, para desarrolar tareas relacionadas con multimedia y especialmente video juegos.

4.2 Instalación de AMD APP SDK v2.4 con OpenCL 1.1

En este apartado explicaremos como instalar AMD APP SDK para Windows y Linux si tu ordenador dispone de una GPU ATI/AMD.

Apple añade soporte completo para OpenCL desde la versión 10.6 de MacOSX “Snow Leopard”, tanto en CPUs de Intel como para GPUs de ATI/AMD. Por contra, ordenadores antiguos de Mac que usen Intel Graphics como GPU, no pueden acelerar OpenCL.

4.2.1 Prerrequisitos

Para poder crear aplicaciones OpenCL sobre una plataforma AMD necesitaremos el AMD SDK APP[15]. Aquí explicaremos como instalar la versión **2.4**, pero la última versión disponible al momento de escribir este documento es la 2.5. Posteriores versiones deberían instalarse de forma similar. De todos modos, recomiendo encarecidamente leer la documentación suministrada por AMD antes de realizar la instalación.

A continuación se muestran los dispositivos soportados por OpenCL:

AMD APU Family with AMD Radeon™ HD Graphics	A-Series E2-Series C-Series E-Series G-Series
AMD Radeon™ HD Graphics	6900 Series (6970,6950) 6800 Series (6870,6850) 6700 Series (6790,6770,6750) 6600 Series (6670) 6500 Series (6570) 6400 Series (6450)
ATI Radeon™ HD Graphics	5900 Series (5970) 5800 Series (5870,5850,5830) 5700 Series (5770,5750) 5600 Series (5670) 5500 Series (5570) 5400 Series (5450)
ATI FirePro™ Graphics	V8800 V7800 V5800 V4800 V3800
ATI Mobility Radeon™ HD	5800 Series (5870, 5850, 5830) 5700 Series (5770, 5750, 5730) 5600 Series (5650) → Tarjeta donde se

15 http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx#one

	realizarán las pruebas. 5400 Series (5470, 5450, 5430)
ATI Mobility FirePro™	M7820 M5800
AMD CPU	X86 CPU con SSE 2.x or later

4.2.2 Windows:

Debes contar con permisos de Administrador para instalar el SDK. También debes tener la última versión de los drivers ATI Catalyst[16].

AMD APP SDK está soportado por los siguientes sistemas operativos Windows:

- Windows XP SP3(32-bits), SP2(64-bits)
- Windows Vista SP1(32/64-bits)
- Windows 7 (32/64-bits)

Los compiladores soportados son:

- Microsoft Visual Studio **(MSVS) 2008 Professional Edition**
- Microsoft Visual Studio **(MSVS) 2010 Professional Edition**
- Intel C Compiler **(ICC)11.x**
- Minimalist GNU for Windows **(MinGW) [GCC 4,4]**

Paso 1. Elige el AMD APP SDK .exe que se corresponde con tu sistema y haz doble clic sobre él.

- ✔ Para 32-bits Windows XP, Vista y Windows 7, elige:
  AMD-APP-SDK-v2.4-Windows-32.exe
- ✔ Para 64-bits Windows XP, Vista y Windows 7, elige:
  AMD-APP-SDK-v2.4-Windows-64.exe

Los archivos extraídos son guardados automáticamente en C:\AMD\SUPPORT\<nombre del archivo descargado >\.

Paso 2. Si el programa de instalación no se inicia automáticamente después de que los archivos hayan sido extraídos, accede al subdirectorio C:\AMD\SUPPORT y haz doble clic en el fichero *Setup.exe*. Una pantalla de bienvenida aparecerá, permitiéndote elegir el lenguaje. La opción por defecto es Inglés. Pulsa *siguiente*.

16 http://support.amd.com/us/gpudownload/Pages/index.aspx

Paso 3. Haz clic sobre el icono de instalación. La siguiente pantalla te permitirá elegir el tipo de instalación que deseas, además de la ruta de instalación por defecto.

La opción *Express* instala:

- AMD APP SDK v2 Developer
- AMD APP SDK v2 Samples
- AMD APP SDK v2 Runtime
- AMD APP Profiler 2.3
- AMD APP KernelAnalyzer 1.9

Eligiendo *Custom* te permitirá seleccionar los componentes que desees instalar. Si seleccionas *Express*, continua en el paso 4. Si seleccionas *Custom*, salta al paso 9.

Paso 4. Si has elegido Express, pulsa *siguiente*, aparecerá el Contrato de Licencia del Usuario Final o CLUF. Hay que leer con detenimiento ese contrato y pulsar *aceptar* en caso de que se esté de acuerdo. Aparecerá una pantalla con un mensaje temporal “Analizando Sistema”. Este programa estará detectando el tipo de gráficos hardware y software actualmente instalados en el sistema. Si una versión anterior de uno de los componentes ya se encuentra instalado, un mensaje de alarma aparecerá, indicando que no se puede continuar con la instalación a no ser que sea eliminado.

Paso 5. Cuando el Asistente de Instalación de Windows aparezca, pulsa *Siguiente*. Aparecerá una pantalla donde podrás aceptar o cambiar la ruta de instalación de los archivos. Pulsa *Siguiente*.

Paso 6. Cuando el Acuerdo de Licencia aparezca, primero pulsa el botón *Siguiente* para “Yo acepto los términos de el Acuerdo de Licencia”. Luego, pulsa *Siguiente*.

Paso 7. En la siguiente ventana, pulsa *Instalar* para empezar la instalación de los archivos SDK. Una barra de progreso se mostrará. Después de que la instalación se complete, una ventana de confirmación aparecerá. Pulsa *Finalizar*. Esto completa la instalación de AMD APP SDK V2.4.

Paso 8. Cuando la instalación está terminada, una ventana de confirmación aparecerá. Pulsa *Finalizar*. Los siguientes pasos son solo si elegiste la opción *Custom* en el paso 3.

Paso 9. Si pulsaste sobre *Custom* en el paso 3, pulsa *Siguiente* y una pantalla con el mensaje temporal “Analizando Sistema” se mostrará. El programa está detectando qué

tipo de gráficos hardware y software están actualmente instalados en el sistema. Después de varios segundos, la pantalla de Instalación Customizada aparecerá. Esto te permite elegir qué componentes quieres instalar. Por defecto, los 3 componentes están marcados.

Paso 10. Selecciona el/los conponente(s) que quieres instalar y pulsa *Siguiente*. El Contrato de Licencia del Usuario Final aparecerá. Pulsa *Siguiente*.

Paso 11. Cuando el Asistente de Instalación de Windows aparezca, pulsa *Siguiente*. Se mostrará una pantalla donde podrás aceptar o cambiar la ruta de instalación de los archivos. Pulsa *Siguiente*.

Paso 12. Una barra de progreso aparecerá, seguida por una ventana que confirma que la instalación se ha completado. Pulsa *Finalizar*.

Establecer manualmente las Variables de Entorno.

Si el contenido de las variables de entorno se corrompe, por cualquier causa, debes asegurar el valor apropiado para las siguientes variables de entorno:

La variable *AMDAPPSDKROOT* debe ser establecida a:

C:\Program Files\AMD PP (para sistemas de 32-bits)

C:\Program Files (x86)\AMD PP (para sistemas de 64-bits)

Si la configuración por defecto no fue usada, modifica el valor de la localización especificada durante la instalación.

La variable *AMDAPPSDKSAMPLEROOT* debe ser establecida a:

C:\Users\<nombre usuario>\Documents\AMD APP\ (para Windows Vista o Windows 7)

C:\Documents and Settings\<nombre usuario>\My Documents\AMD APP (para Windows XP)

Si la configuración por defecto no fue usada, modifica el valor de la localización especificada durante la instalación.

La variable PATH debe incluir:

$(*AMDAPPSDKROOT*) \bin\x86 (para sistemas de 32-bits)

$(*AMDAPPSDKROOT*) \bin\x86_64 (para sistemas de 64-bits)

$(*AMDAPPSDKSAMPLEROOT*) \bin\x86 (para sistemas de 32-bits)

$(*AMDAPPSDKSAMPLEROOT*) \bin\x86_64 (para sistemas de 64-bits)

4.2.3 Linux:

Debes contar con permisos de root para instalar el SDK. También debes tener la última versión de los drivers ATI Catalyst[17].

Paso 1. Desempaquetar el SDK a una localización de tu elección, por ejemplo si queremos descomprimirlo en **/home/<nombre_usuario>/ ,** introduciríamos en la terminal:

```
cd ~
```

Para sistemas de 32-bits, descomprimimos el archivo .tgz:

```
tar -xvzf AMD-APP-SDK-v2.4-lnx32.tgz
```

Para sistemas de 64-bits, descomprimimos el archivo .tgz:

```
tar -xvzf AMD-APP-SDK-v2.4-lnx64.tgz
```

Paso 2. Ahora debemos establecer las variables de entorno. Para no tener que asignarlas cada vez que encendamos el ordenador, podemos añadir la asignación de las variables de entorno a tu archivo .bashrc que podrás encontrar en **/home/<nombre_usuario>/.bashrc**, para ello modifica con tu editor preferido el archivo **~/.bashrc**

y añade las siguientes lineas:

- Para la instalación de 32-bits:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH":/home/nombre_usuario/AMD-APP-SDK-
            v2.4-lnx32/lib/x86/"

export LIBRARY_PATH=$LIBRARY_PATH":/home/nombre_usuario/AMD-APP-SDK-v2.4-
            lnx32/lib/x86/"

export C_INCLUDE_PATH=$C_INCLUDE_PATH":/home/nombre_usuario/AMD-APP-SDK-
            v2.4-lnx32/include/"
```

- Para la instalación de 64 bits:

17 http://support.amd.com/us/gpudownload/Pages/index.aspx

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH":/home/nombre_usuario/AMD-APP-SDK-
            v2.4-lnx64/lib/x86_64/"

export LIBRARY_PATH=$LIBRARY_PATH":/home/nombre_usuario/AMD-APP-SDK-v2.4-
            lnx64/lib/x86_64/"

export C_INCLUDE_PATH=$C_INCLUDE_PATH":/home/nombre_usuario/AMD-APP-SDK-
            v2.4-lnx64/include/"
```

Paso 3. Para finalizar la instalación debes registrar el OpenCL ICD. En caso contrario, los programas de ejemplo y otras aplicaciones fallarán al ejecutarse. Para registrar el ICD introduce en la terminal:

```
            su
```

Introduce la contraseña de root.

```
            cd /

            tar xfz ~/AMD-APP-SDK-v2.4-lnxXX/icd-registration.tgz
```

Donde XX es la versión de 32 o 64 bits que hayas descargado.

Si no se realiza este paso, los programas de ejemplo y otras aplicaciones no se ejecutarán y aparecerá el error CL_PLATFORM_NOT_FOUND_KHR (-1001).

4.3 La arquitectura OpenCL

Para describir las principales ideas detrás de OpenCL, se usará una jerarquía de modelos:

- Modelo de plataforma.
- Modelo de ejecución.
- Modelo de memoria.
- Modelo de programación.

4.3.1 Modelo de la plataforma

El modelo de plataforma definido por OpenCL es la de un anfitrión (a partir de ahora llamado host) conectado a uno o más dispositivos OpenCL (huéspedes). Un dispositivo OpenCL consiste en una o más unidades de cómputo (“cores” o “compute units” en inglés) que a su vez contienen uno o más elementos de procesamiento. Las operaciones en el dispositivo se suceden en los elementos de procesamiento (“processing elements”). Un host es cualquier computador con una CPU corriendo un sistema operativo estándar. Los dispositivos pueden ser una GPU, DSP (Digital Signal Processor) o un CPU multi-núcleo.

Una aplicación OpenCL corre sobre un host de acuerdo a los modelos nativos de la plataforma del host. La aplicación OpenCL envía **órdenes** desde el host para ejecutar operaciones sobre los elementos de procesamiento en el dispositivo. Los Elementos de procesamiento ejecutan instrucciones SIMD (Single Instruction, Multiple Data) o SPMD (Single Program, Multiple Data). Las instrucciones SPMD son ejecutadas sobre dispositivos de propósito general como CPUs, mientras que las instrucciones SIMD requieren un procesador de vectores en una GPU o una unidad de vectores en una CPU.

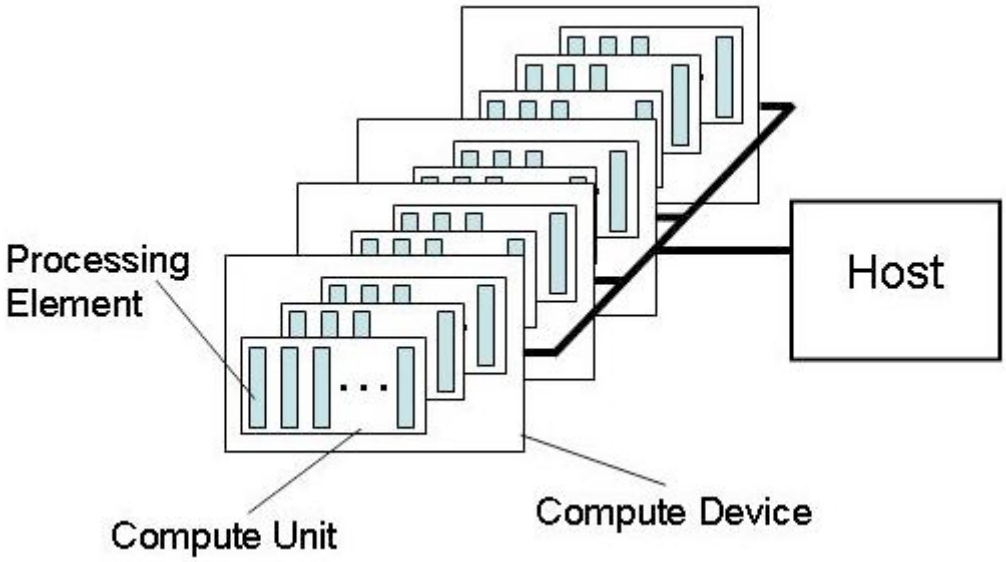


Figura 1. Modelo de plataforma, un host abarca uno o varios dispositivos de cómputo. Cada uno de ellos con una o varias unidades de computo, cada una con uno o varios elementos de procesamiento.

4.3.2 Modelo de la ejecución

El modelo de ejecución de OpenCL comprende dos componentes: los kernels y los programas host. Los Kernels son unidades básicas de código que se ejecutan sobre uno o más dispositivos OpenCL. Los Kernels pueden ser paralelos a nivel de datos y de tareas, de forma similar a una función en lenguaje C. El programa host se ejecuta en el ordenador host, define el **contexto** del dispositivo, y maneja la cola de las instancias de la ejecución del kernel usando **colas de órdenes**. Los Kernels se ponen en cola en orden, pero pueden ser ejecutados en ese orden o no.

Kernels

OpenCL explota la computación paralela sobre los dispositivos al definir el problema en un índice espacial. Cuando un kernel se pone en cola para ejecutarse por el programa host, un índice espacial se define. Cada elemento independiente de ejecución en este índice espacial se llama un objeto de trabajo (work-item). Cada “work-item” ejecuta la misma funcionalidad del kernel, pero con datos diferentes. Cuando una orden del kernel es puesto en la cola de órdenes, un índice espacial debe ser definido para permitir al dispositivo mantener un registro del número total de “work-items” que requiere la ejecución. El índice espacial puede ser N=1, 2 o 3. Un vector de datos será considerado N=1; procesar una imagen será N=2, y un volumen 3D será N=3.

Procesar una imagen de 1024x1024 se resolverá de la siguiente forma: el índice espacial global comprende un espacio bidimensional de 1024 por 1024 por lo tanto, un kernel de ejecución por pixel con un total de 1048576 ejecuciones. Con este índice espacial, a cada “work-item” se le asigna un identificador global único. Por ejemplo, el

"work-item" para el pixel x=30, y=22 tendrá el identificador global (30, 22).

OpenCL también permite agrupar "work-items" para formar "work-groups". El tamaño de cada "work-group" se define por su propio índice espacial local. Todos los "work-items" en el mismo "work-group" son ejecutados juntos en el mismo dispositivo. La razón para ejecutarlos en un mismo dispositivo es para permitir a los "work-items" compartir memoria local y sincronización. Los "work-items" globales son independientes y no pueden ser sincronizados. La sincronización está solamente permitida entre los "work-items" en un mismo "work-group".

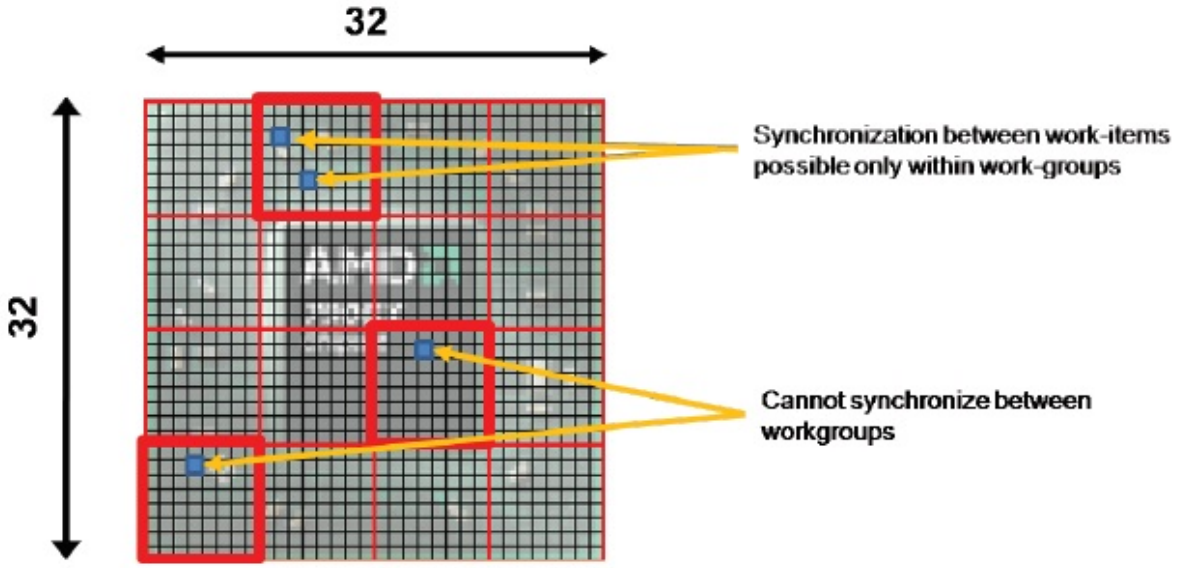


Figura 2. Agrupando Work-items en Work-groups.

El modelo de ejecución soporta dos categorías de kernels: **kernels OpenCL** y **kernels nativos**.

Los **kernels OpenCL** están escritos en lenguaje OpenCL C y compilados con el compilador OpenCL. Todas las implementaciones OpenCL son capaces de ejecutar kernels OpenCL.

Los **kernels nativos** son kernels extendidos que pueden ser funciones especiales definidas en código de aplicación o exportadas desde la librería diseñada a un acelerador particular. El API OpenCL incluye funciones para la consulta de las capacidades de los dispositivos para determinar si los kernels nativos son soportados.

Programa Host

El programa Host es el responsable de establecer y manejar la ejecución de los kernels en el dispositivo OpenCL a través del uso de un **contexto**. A través de la API OpenCL, el host puede crear y manipular el contexto al incluir los siguientes recursos:

1. **Dispositivos**: Un conjunto de dispositivos OpenCL usado por el host para ejecutar kernels.
2. **Objeto Programa**: La fuente del programa o el programa objeto que implementa un kernel o una colección de kernels.
3. **Kernels**: Las funciones OpenCL específicas que son ejecutadas en un dispositivo OpenCL.
4. **Objetos Memoria**: Un conjunto de buffers de memoria o mapas de memoria común al host y a los dispositivos OpenCL.

Después de que el contexto sea creado, las colas de órdenes son creadas para manejar la ejecución de los kernels sobre los dispositivos OpenCL que fueron asociados con el contexto. Las colas de órdenes aceptan tres tipos de órdenes:

- **Órdenes de ejecución del kernel**: ejecutan los comandos del kernel sobre los dispositivos OpenCL.
- **Órdenes de memoria**: transfiere objetos de memoria entre el espacio de memoria del host y el espacio de memoria de los dispositivos OpenCL.
- **Órdenes de sincronización**: Define el orden en el que los comandos serán ejecutados.

Como se ha indicado, las colas de comandos pueden ser ejecutadas de forma **ordenada** y de forma **desordenada**. En la forma **ordenada**, los comandos son ejecutados como fueron introducidos en la cola, es decir, tiene que terminar un comando previo para que uno posterior se pueda ejecutar. Mediante la forma **desordenada**, los comandos no tienen que esperar a que termine un comando previo para poder ejecutarse, por lo que el programador tiene que definir de forma explícita la sincronización de los comandos.

4.4 El modelo de memoria

Como entre el host y los dispositivos OpenCL no está compartido el espacio de direccionamiento de la memoria, el modelo de memoria OpenCL establece cuatro regiones de memoria accesible por los “work-items” cuando están ejecutando el kernel.

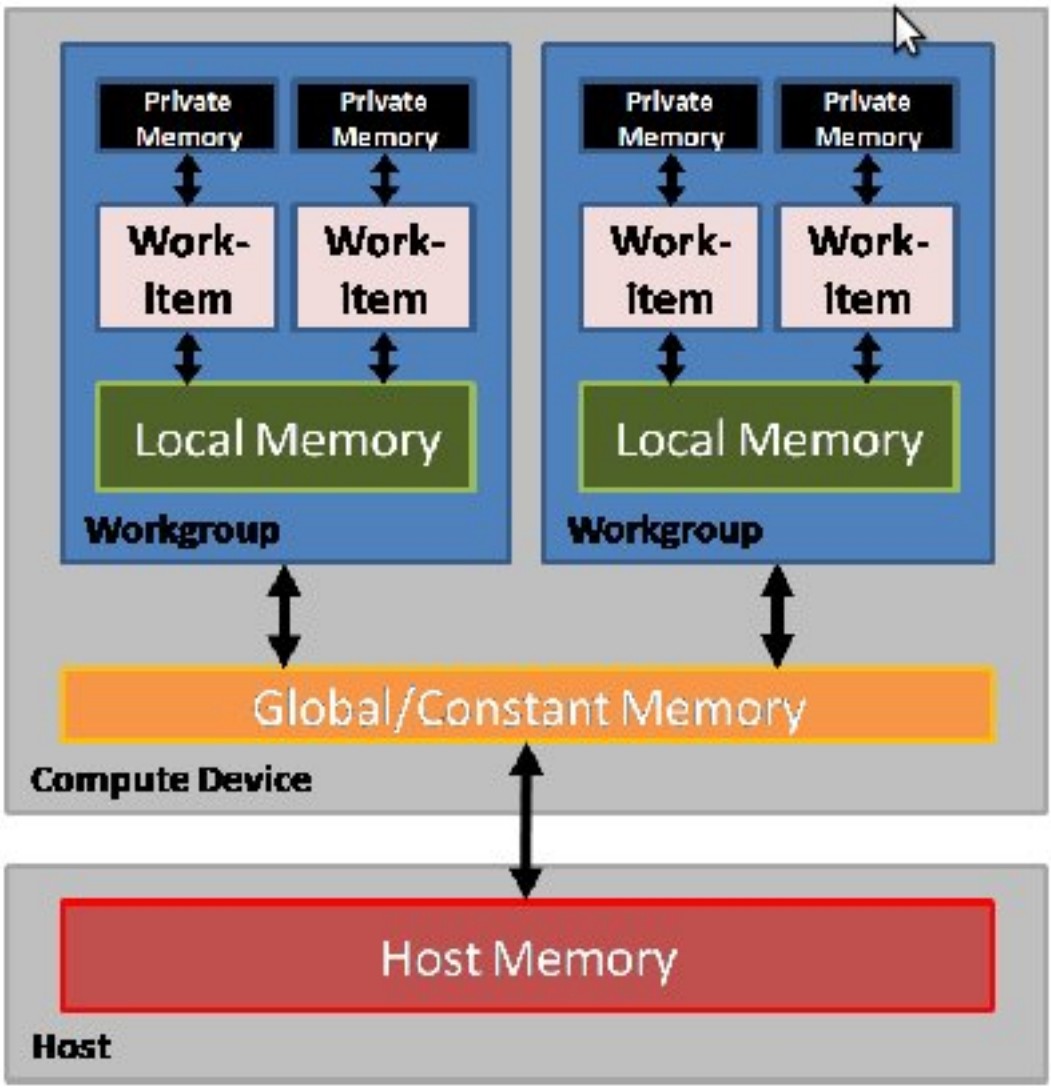


Figura 3. Muestra las regiones de la memoria accesible por el host y el dispositivo OpenCL.

Como se puede ver en la imagen, se distinguen cuatro regiones distintas de memoria:

Memoria Global: Esta región de memoria permite el acceso de lectura y escritura a todos los “work-items” en todos los “work-groups”. Un “work-item” puede leer o escribir cualquier elemento de un objeto alojado en esta memoria. Las lecturas y escrituras de la memoria global podrían ser cacheadas dependiendo de las capacidades del dispositivo. En esta región de memoria solo pueden ser alojados elementos por el host durante el tiempo de ejecución.

Memoria Constante: Es una región de la memoria global que se mantiene constante durante la ejecución del kernel. El host es quien aloja e inicializa objetos en la memoria constante. Los “work-item” solo tienen acceso de lectura.

Memoria Local: Es una región local de memoria accesible por el “work-group”. Esta memoria puede ser usada para alojar variables que son compartidas por todos los “work-item” en un “work-group”.

Memoria Privada: Es una región privada de memoria accesible por un solo “work-item”. Las variables que se definen en la memoria privada de un “work-item” no son visibles por otro “work-item”.

La aplicación que se ejecuta en el host usa la API OpenCL para crear objetos en la memoria global, y para encolar comandos que operan con estos objetos en memoria.

El host y la memoria del dispositivo OpenCL son independientes. Esto es necesario ya que el host está definido fuera de OpenCL. Aunque son independientes, necesitan interaccionar. Esta interacción puede ocurrir de dos formas: copiando datos de forma explicita o al des/mapear regiones de un objeto en memoria.

Copiar datos de forma explícita: El host encola comandos para transferir datos entre la memoria del dispositivo y la del host. Las transferencias de datos entre memorias pueden ser bloqueantes o no-bloqueantes. Para una transacción de memorias no-bloqueantes, la función OpenCL pide información a la memoria sin importarle que la memoria del host sea segura usarla. De manera bloqueante, la función OpenCL espera a que la memoria le indique que ya no está bloqueada y que es seguro copiar/leer información.

El método del Des/Mapeo: Mediante este método, el host puede mapear una región de la memoria del dispositivo OpenCL en su propio espacio de direccionamiento. Este comando de mapeo puede ser, al igual que el anterior método de interacción, bloqueante y no-bloqueante. Una vez que el host ha mapeado la memoria del dispositivo, el host puede leer o escribir en esa región. El host desmapea la región cuando todos los accesos a esta región mapeada se han realizado.

4.5 Modelo de programación

Como se ha dicho, el modelo de ejecución de OpenCL soporta los modelos de programación paralela de datos y de tareas. Aunque el modelo que conduce el diseño de OpenCL es el paralelo de datos.

Modelo de programación paralela de datos: En este modelo se define la computación como una secuencia de instrucciones aplicada a múltiples elementos de la memoria del dispositivo. El índice espacial de OpenCL define los “work-items” y cómo la información es mapeada dentro de los “work-items”. OpenCL implementa una versión relajada del modelo de programación paralela de datos, donde no se requiere un mapeado uno-a-uno estricto.

En el modelo explícito, el programador define el número total de “work-items” que se ejecutan en paralelo y también cómo los “work-items” son divididos a lo largo de los “work-groups”. En el modelo implícito, el programador especifica el número total de “work-items” y OpenCL se encarga de dividirlos en “work-groups” y de manejarlos.

Modelo de programación paralela de tareas: En este modelo una sola instancia del kernel es ejecutado independientemente del índice espacial. Es equivalente a ejecutar un kernel en una unidad de cómputo con un “work-group” que contiene solamente un “work-item”. Bajo este modelo, los usuarios expresan el paralelismo al:

-Usar tipos de vectores de información implementada por el dispositivo.

-Encolar múltiples tareas

-Encolar kernels nativos desarrollados usando un modelo de programación ortogonal a OpenCL.

Sincronización

Existen dos dominios de sincronización en OpenCL:

-”Work-items” en un solo “work-group”.

-Comandos encolados en una cola de comandos en un contexto simple.

La sincronización entre “work-items” en un “work-group” se realiza utilizando una barrera. Todos los “work-items” de un “work-group” deben ejecutar la barrera antes de que alguno de ellos prosiga su ejecución, por lo que la barrera debe ser alcanzada por todos los “work-items” o por ninguno de ellos. No existe ningún mecanismo de sincronización entre “work-groups”.

Los puntos de sincronización entre los comandos en una cola de comandos son:

Barrera de la cola de comandos: La barrera de la cola de comandos se asegura de que todos los comandos previamente encolandos han terminado su ejecución y que las actualizaciones de los objetos en memoria son visibles para los comando encolados posteriormente. Esta barrera solo puede ser usada para sincronizar comandos en una sola cola de comandos.

Esperar a un evento: Todas las funciones de la API OpenCL que encolan comandos, devuelven un evento que identifica al comando y actualiza el objeto en memoria. Un comando que espera a ese evento, garantiza que la actualización del objeto en memoria es visible para otros comandos.

4.6 Anatomía de un programa OpenCL.

Una aplicación OpenCL tipica empieza preguntando al sistema por la disponibilidad de dispositivos OpenCL. Cuando los dispositivos han sido identificados, un contexto permite la creación de colas de comandos, la creación de programas y kernels, el manejo de memoria entre el host y el dispositivo OpenCL y el envio de kernels a ejecución.

El siguiente ejemplo muestra la estructura básica de un programa en OpenCL, donde se utiliza un buffer de entrada, se multiplican los valores, y se guarda el resultado en un buffer de salida. Se observa la relación entre el dispositivo, contexto, programa, kernel, cola de comandos y buffers.

```
#include "CL/cl.h"

const char *KernelSource = "__kernel void hello(__global float *input, __global
float *salida)\n"\
"{\n"\
" size_t id = get_global_id(0);\n"\
"  salida[id] = entrada[id] * entrada[id];\n"\
"}\n"\
"\n";

int main (void)
{
        cl_context contexto;
        cl_context_properties propiedades[3];
        cl_kernel kernel;
        cl_command_queue cola_comandos;
        cl_progam program;
        cl_uint num_plataformas=0;
        cl_platform_id id_plataforma:
        cl_device_id id_dispositivo;
        cl_uint num_dispositivos;
        cl_mem entrada, salida;
        size_t global;

        float datosentrada[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        float resultados[10]={0};

        int i;
        //obtiene una lista de las plataformas disponibles
        if(clGetPlatformIDs(1, &id_plataforma, &num_plataformas)!=CL_SUCCESS)
        {
                printf("Imposible obtener id de plataforma\n");
                return 1;
        }

        //Intentar obtener un dispotivo GPU soportado
        if(clGetDeviceIDs(id_plataforma, CL_DEVICE_TYPE_GPU, 1, &id_dispositivo,
&num_dispositivos)!=CL_SUCCESS)
        {
                printf("Imposible obtener id de dispositivo\n");
```

```
                return 1;
        }

        //lista de propiedades del contexto – la ultima propiedad debe ser 0.
        propiedades[0]=CL_CONTEXT_PLATFORM;
        propiedades[1]=(cl_context_properties) id_plataforma;
        propiedades[2]=0;

        //creamos un contexto con el dispositivo GPU
        contexto =
clCreateContext(propiedades,1,&id_dispositivo,NULL,NULL,NULL);

        //creamos una cola de comandos usando el contexto y el dispositivo
        cola_comandos = clCreateCommandQueue(contexto, id_dispositivo, 0, NULL);

        //creamos un programa con el codigo fuente del kernel
        programa = clCreateProgramWithSource(contexto, 1, (const char
**)&KernelSource, NULL, NULL);

        //compilamos el programa
        if(clBuildProgram(programa, 0, NULL, NULL, NULL, NULL)!=CL_SUCCESS)
        {
                printf("Error al compilar el programa\n");
                return 1;
        }

        //especificamos que kernel ejecutara el programa
        kernel = clCreateKernel(programa, "hello", NULL);

        //creamos los buffers de entrada y salida
        entrada = clCreateBuffer(contexto, CL_MEM_READ_ONLY, sizeof(float) * 10,
NULL, NULL);
        salida = clCreateBuffer(contexto, CL_MEM_READ_ONLY, sizeof(float) * 10,
NULL, NULL);

        //cargamos los datos en los buffers
        clEnqueueWriteBuffer(cola_comandos, entrada, CL_TRUE, 0, sizeof(float) *
10, datosentrada, 0, NULL, NULL);

        //especificamos los argumentos del kernel a ejecutar
        clSetKernelArg(kernel, 0, sizeof(cl_mem), &entrada);
        clSetKernelArg(kernel, 1, sizeof(cl_mem), &salida);
        global = 10;

        //ponemos en cola los comandos del kernel a ejecutar
        clEnqueueNDRangeKernel(cola_comandos, kernel, 1, NULL, &global, NULL, 0,
NULL, NULL);
        clFinish(cola_comandos);

        //copiamos los resultados del buffer salida al vector resultados[]
        clEnqueueReadBuffer(cola_comandos, salida, CL_TRUE, 0, sizeof(float)*10,
resultados, 0, NULL, NULL);

        //imprimimos los resultados
        printf("Salida: ");
        for(i=0;i<10;i++)
        {
```

```
                printf("%f ", resultados[i]);
        }

        //limpieza
        clReleaseMemObject(entrada);
        clReleaseMemObject(salida);
        clReleaseProgram(programa);
        clReleaseKernel(kernel);
        clReleaseCommandQueue(cola_comandos);
        clReleaseContext(contexto);

        return 0;
}
```

4.7 Extensiones de OpenCL

El lenguaje utilizado por OpenCL es un subconjunto del ISO C99 con unas excepciones:

- No tiene recursión, ni punteros a funciones, ni cabeceras estándar.
- No tiene bitfields (campos de bits).
- No tiene las palabras clave "*extern*", "*static*", "*auto*" o "*register*".
- No hay arrays con tipos de datos menores que un "int".

Existen diversos proyectos que buscan acercar la computación GPGPU de OpenCL a otros lenguajes de programación. De entre estos proyectos destacan:

JOCL: Java bindings for OpenCL

Te permite utilizar JAVA para escribir el código OpenCL de la aplicación host.

PyOpenCL:


PyOpenCL

Este proyecto permite utilizar PYTHON para escribir el código OpenCL de la aplicación host. Dada la simplicidad de Python, se puede escribir en una línea,

```
program.foo(queue, (20, 16), (5, 4), a, b, c)
```

lo que en lenguaje tradicional de OpenCL se escribiría en 7:

```
foo_kernel = clCreateKernel(program, "foo", NULL);
             clSetKernelArg(foo_kernel, 0, sizeof(a), &a);
             clSetKernelArg(foo_kernel, 1, sizeof(b), &b);
             clSetKernelArg(foo_kernel, 2, sizeof(c), &c);
             size_t global_size[2] = {20, 16};
             size_t local_size[2] = {5, 4};
             clEnqueueNDRangeKernel(queue, foo_kernel, 2, NULL, global_size,
             local_size, 0, NULL, NULL);
```

5 Solución al proyecto

Para demostrar que utilizando la tecnología OpenCL se puede incrementar el rendimiento de los computadores que realizan la ejecución de un algoritmo, se decidió utilizar un algoritmo de multiplicación de matrices cuadradas. Este algoritmo es fácilmente paralelizable, condición esencial para que sea rentable la implementación en OpenCL. Se han creado 3 programas muy parecidos entre sí, pero la función donde se realiza la multiplicación de las matrices es diferente en cada uno de ellos.

Por una parte tenemos el programa que se ejecuta utilizando totalmente la CPU, donde la multiplicación de matrices se realiza de forma secuencial:

```
void secuencialMatrixMulCPU(int hA, int wA, int wB, float *A, float *B, float
*C)
{
        int i, j, k;
        for (i=0; i<hA; i++)
        {
          for (j=0; j<wB; j++)
          {
            for(k=0; k<wA; k++)
            {
                C[i*hA+j] += A[i*hA+k] * B[k*wA+j];
            }
          }
        }
}
```

Como se observa, son 3 bucles “for” que recorren las matrices A y B para generar la matriz C. Este fragmento de código no es un kernel, ya que este programa no utiliza la tecnología OpenCL, pero nos va a servir como comparación a los programas que sí utilizan OpenCL.

Se pasan como parámetros la altura y anchura de la matriz A, la anchura de la matriz B y los vectores que contienen los datos de las dos matrices a multiplicar. Obviamente, para multiplicar matrices es necesario que la altura de B sea igual a la anchura de A.

El siguiente fragmento es del segundo programa, se ejecuta en la CPU una primera vez y posteriormente se ejecuta sobre la GPU.

```
// Código fuente del Kernel
const char *KernelSource = "\n"                                    \
  "__kernel void MatrixMultSimple(                                 \n" \
  "     __global float *mC,                                        \n" \
  "     __global float *mA,                                        \n" \
  "     __global float *mB,                                        \n" \
  "     const int Mdim, const int Ndim, const int Pdim)            \n" \
  "{                                                               \n" \
  "     int k;                                                     \n" \
  "     int i = get_global_id(0);                                  \n" \
  "     int j = get_global_id(1);                                  \n" \
  "     float sum = 0.0f;                                          \n" \
```

```
  "     // multiplicamos cada elemento de la columna con cada           \n" \
  "     // elemento de las columnas apuntadas por el work-item actual   \n" \
  "     for (k=0; k< Pdim; k++)                                         \n" \
  "             sum += mA[i * Ndim + k] * mB[k * Pdim + j];             \n" \
  "     // copiamos la suma al buffer del dispositivo.                  \n" \
  "     mC[i * Ndim + j] = sum;                                         \n" \
  "}                                                                    \n" \
  "\n";
```

El código muestra, asignado a una variable, el código fuente del kernel que se ejecuta en el dispositivo. Como se ve, el código del kernel se almacena en un vector de caracteres. Se realizó de esta forma ya que solo se va a utilizar un kernel, por lo que era más simple hacerlo así. En el caso de que se vayan a utilizar diversos kernels en un mismo programa, es conveniente usar el fichero "kernels.cl" donde se pondrán todos los kernels a utilizar y luego hay que indicar cual de ellos se quiere utilizar en cada caso.

Este kernel toma como parámetros los 3 vectores con los datos de la matriz A y B y uno que se rellenará con la información de C; Mdim, Pdim y Ndim se corresponden con la altura y anchura de la matriz A y anchura de la matriz B respectivamente. Al principio se obtienen los índices globales, que indican al sistema en que posición de las matrices A y B se encuentra los datos que tiene que multiplicar. Después se realiza la multiplicación de la suma parcial en el único bucle "for" del kernel, para posteriormente copiar en el vector del dispositivo el resultado de la multiplicación.

Hay que decir, que la ejecución sobre la GPU y la CPU se diferencian a través del flag CL_DEVICE_TYPE_CPU 51 y CL_DEVICE_TYPE_GPU 51. En este programa se le pasan los flags anteriores a la función que contiene todo el código *OpenCL* ***void MatMultOpencl****(cl_uint Mdim, cl_uint Pdim, cl_uint Ndim,* ***int*** *tipoDisp)* y dentro de ésta, a la función *clGetDeviceIDs(platform_id,* ***tipoDisp****, 1, &device_id, &num_devs_returned)*. TipoDist puede ser o CL_DEVICE_TYPE_CPU o CL_DEVICE_TYPE_GPU.

Finalmente, el siguiente fragmento es del tercer programa:

```
// Código fuente del Kernel
const char *KernelSource = "\n"                                  \
  "__kernel void matrixMul(                                      \n"\
  "     __global float* mC,                                      \n"\
  "     __global float* mA,                                      \n"\
  "     __global float* mB, int wA, int wB, int block_size)      \n"\
  "{                                                             \n"\
  "  // índice de bloque                                         \n"\
  "  int bx = get_group_id(0);                                   \n"\
  "  int by = get_group_id(1);                                   \n"\
  "  // índice de hilo                                          \n"\
  "  int tx = get_local_id(0);                                   \n"\
  "  int ty = get_local_id(1);                                   \n"\
  "  // índice de la primera sub-matriz de A procesada          \n"\
  "  // por el bloque                                           \n"\
  "  int aBegin = wA * block_size * by;                          \n"\
  "  // índice de la última sub-matriz de A procesada           \n"\
  "  // por el bloque                                           \n"\
  "  int aEnd   = aBegin + wA - 1;                               \n"\
  "  // Tamaño usado para iterar sobre las sub-matrices de A    \n"\
  "  int aStep  = block_size;                                    \n"\
```

```
"  // índice de la primera sub-matriz de A procesada                    \n"\
"  // por el bloque                                                     \n"\
"  int bBegin = block_size * bx;                                        \n"\
"  // Tamaño usado para iterar sobre las sub-matrices de B             \n"\
"  int bStep  = block_size * wB;                                        \n"\
"  float Csub = 0;                                                      \n"\
"  // Bucle sobre todas las sub-matrices de A y de B                   \n"\
"  // requeridas para realizar el cómputo de bloque de sub-matriz     \n"\
"  for (int a = aBegin, b = bBegin;                                     \n"\
"           a <= aEnd;                                                  \n"\
"           a += aStep, b += bStep)                                     \n"\
"  {                                                                    \n"\
"      // Declaración del array A en memoria local                     \n"\
"      // usado para almacenar las sub-matrices de A                   \n"\
"      __local float As[16][16];                                       \n"\
"      // Declaración del array B en memoria local                     \n"\
"      // usado para almacenar las sub-matrices de B                   \n"\
"      __local float Bs[16][16];                                       \n"\
"      // Carga las matrices desde la memoria global                   \n"\
"      // a la memoria local; cada hilo carga                          \n"\
"      // un elemento de cada matriz                                   \n"\
"      As[ty][tx] = mA[a + wA * ty + tx];                              \n"\
"      Bs[ty][tx] = mB[b + wB * ty + tx];                              \n"\
"      // Sincronización para asegurarnos que las matrices             \n"\
"      // han sido cargadas                                            \n"\
"      barrier(CLK_LOCAL_MEM_FENCE);                                   \n"\
"      // Multiplicación de las dos matrices juntas;                   \n"\
"      // cada hilo computa un elemento                                \n"\
"      // del bloque sub-matriz                                        \n"\
"      for (int k = 0; k < block_size; ++k)                            \n"\
"          Csub += As[ty][k] * Bs[k][tx];                              \n"\
"      // Sincronización para estar seguros que la                     \n"\
"      // computación anterior está hecha antes de cargar dos nuevas   \n"\
"      // sub-matrices de A y B en la próxima iteración                \n"\
"      barrier(CLK_LOCAL_MEM_FENCE);                                   \n"\
"  }                                                                    \n"\
"  // Escribir la sub-matriz en la memoria del dispositivo             \n"\
"  // cada hilo escribe un elemento                                    \n"\
"  int c = wB * block_size * by + block_size * bx;                      \n"\
"  mC[c + wB * ty + tx] = Csub;                                         \n"\
"}                                                                      \n"\
"\n";
```

Este kernel está optimizado para usarlo en la GPU. A los parámetros utilizados en la función anterior para multiplicar las matrices y el kernel, se le añade un nuevo parámetro, el tamaño de bloque (block_size), ya que el kernel utiliza “work-groups”, por lo que hay que indicar de qué tamaño es cada bloque. Este kernel crea dos arrays en la memoria local del dispositivo que se rellenan con la información de las matrices de la memoria global. Hay que decir que la memoria local del dispositivo es más rápida que la memoria del host, ya que el kernel no tiene que pedir la información a la memoria del host, después transferirla a la memoria global/constante del dispositivo y de ahí transferir la información a la memoria local del dispositivo. Si se realiza de una sola vez es más rápido que realizando este proceso para cada dato que se necesite.

Otro punto a destacar es el uso de las barreras (barrier) para realizar la sincronización de los work-groups, de lo contrario no tendríamos la certeza de que todos los hilos han terminado su ejecución antes de pedir cargar en memoria local un nuevo bloque de sub-matrices.

6 Resultados

6.1 Especificaciones del ordenador de pruebas

El equipo con el que se han realizado las pruebas es un portátil con la siguiente configuración:

Sistema operativo:

Ubuntu Linux 10.04 LTS con kernel 2.6.32-32-generic

Arquitectura:

Plataforma: Procesador Intel® Core™ i5
Conjunto de chips: Chipset Intel® HM55 Express

CPU:

Nombre del procesador: Procesador Intel® Core™ i5-460M
Velocidad del procesador(GHz): 2,53 Ghz con TurboBoost hasta 2,80
Caché L3 (MB): 3
Número de núcleos: 2

Tarjeta Gráfica:

Nombre de la tarjeta gráfica: Gráficos ATI Mobility Radeon™ HD 5650
Memoria RAM de vídeo (MB): 1 GB

Memoria:

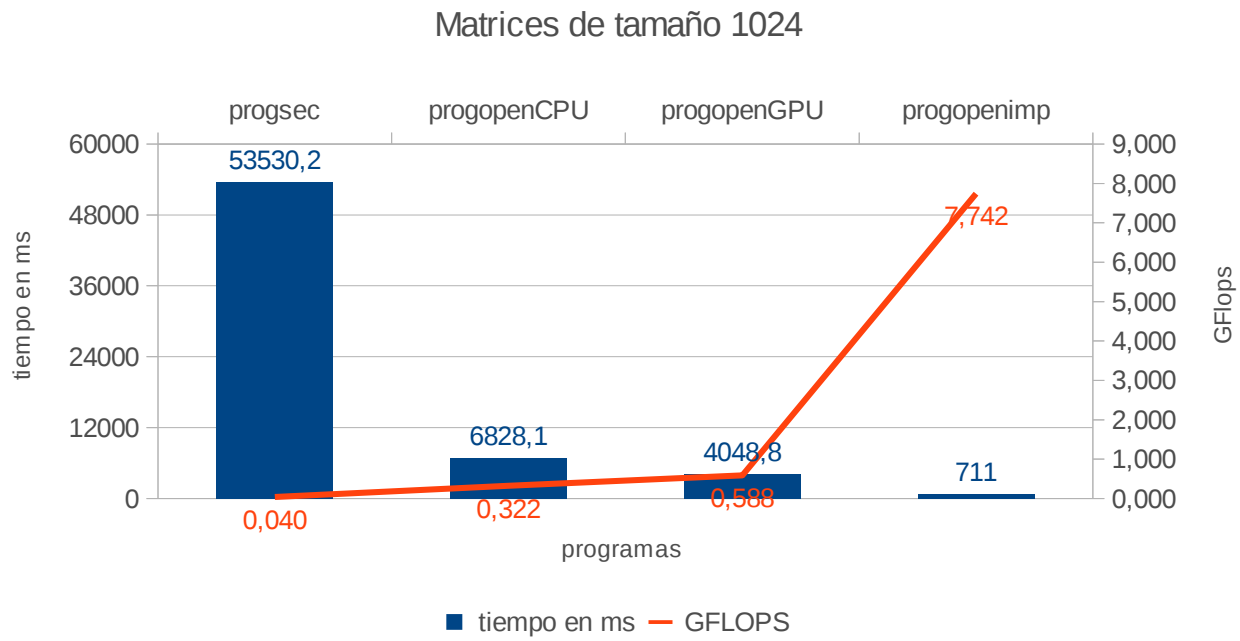
Tamaño de memoria (GB) : 4
Especificaciones de memoria: PC3-8500
Velocidad de memoria (Mhz): 1066
Tipo de memoria: DDR3 SDRAM
Memoria máxima compatible (GB): 8

Unidades:

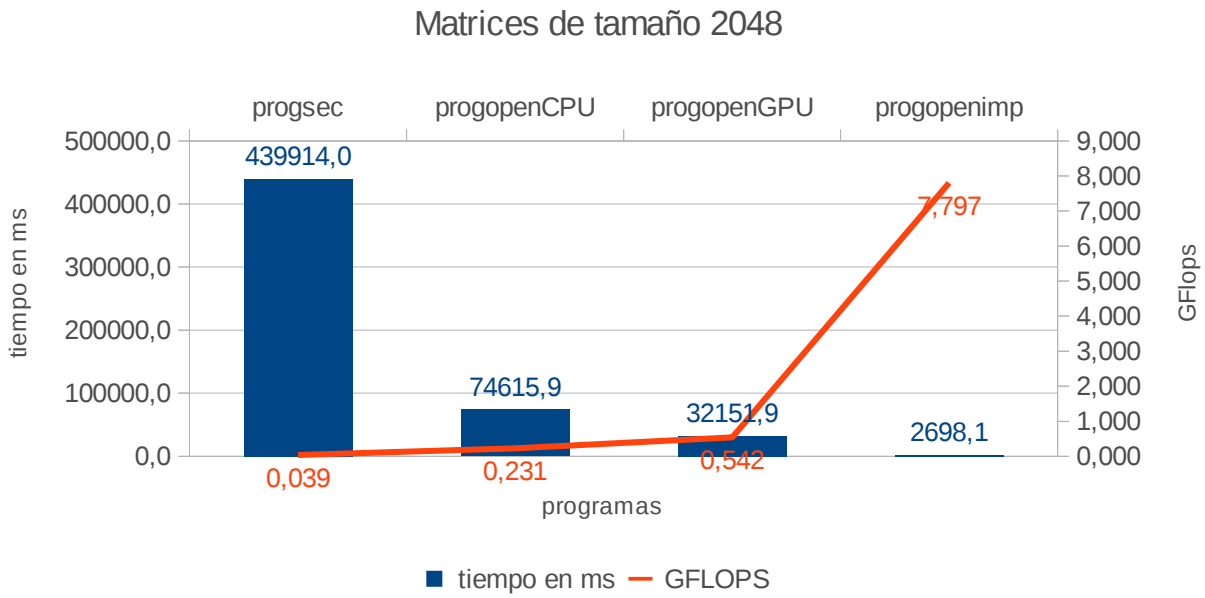
Tipo de unidad de disco duro: Serial ATA
Capacidad de disco duro (GB): 500 GB
Velocidad de disco duro (rpm): 5400
Tipo de unidad óptica: Unidad de DVD SuperMulti

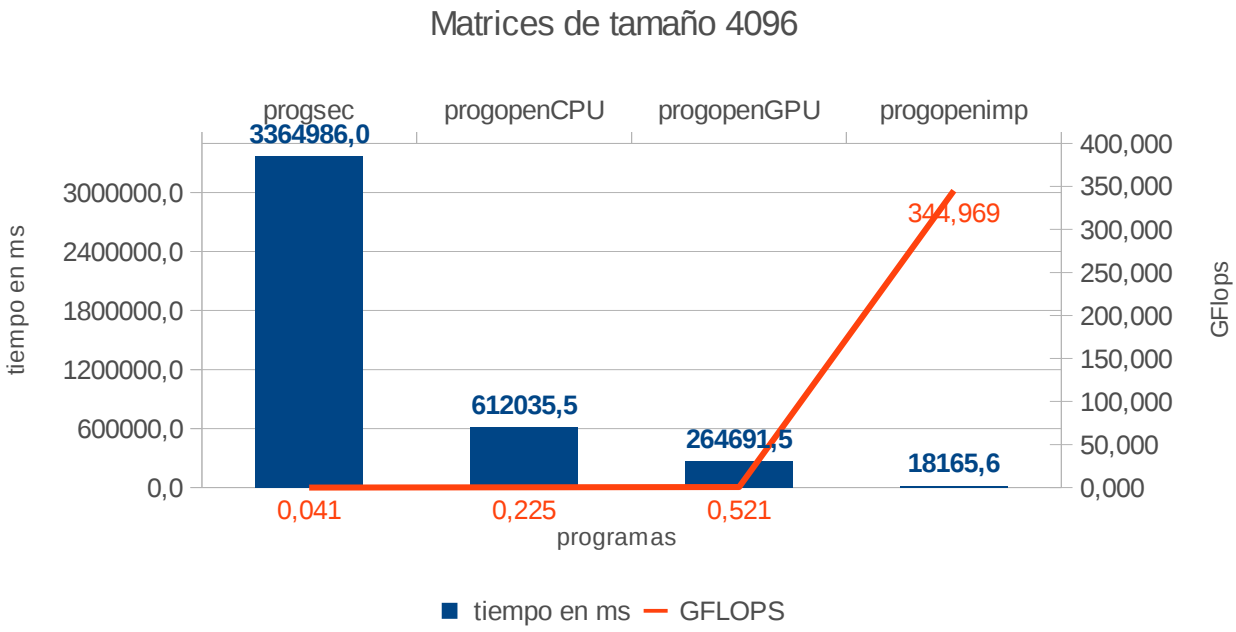
6.2 Gráficas de resultados

Los resultados obtenidos se van a presentar en forma de gráficas. Las pruebas se han realizado utilizando tamaños de matrices cuadradas de 1024x1024, 2048x2048 y 4096x4096 datos.

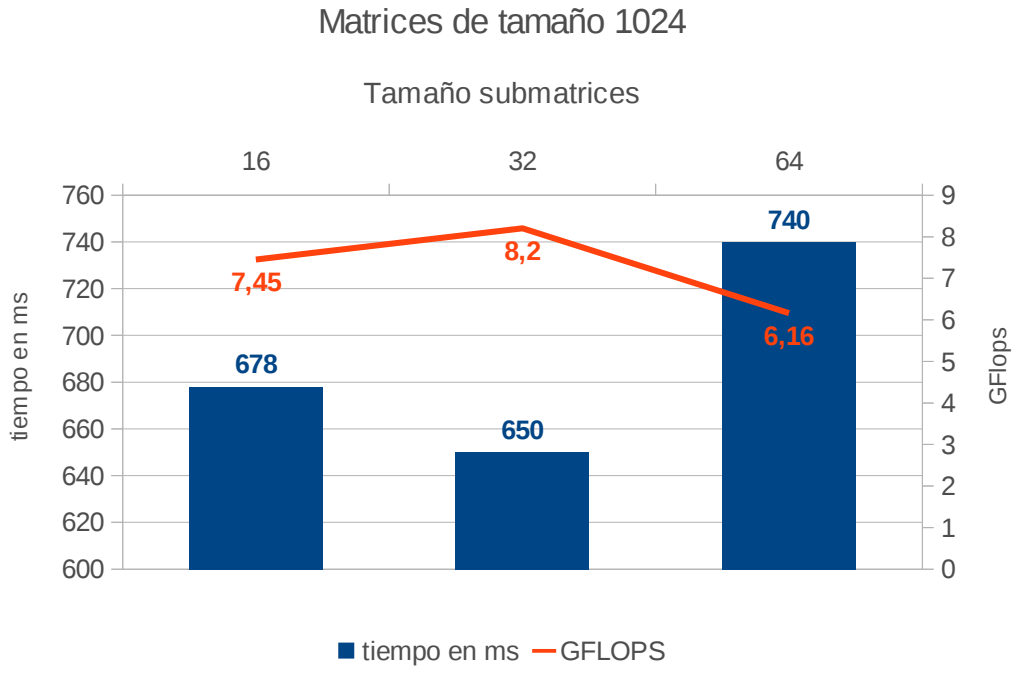


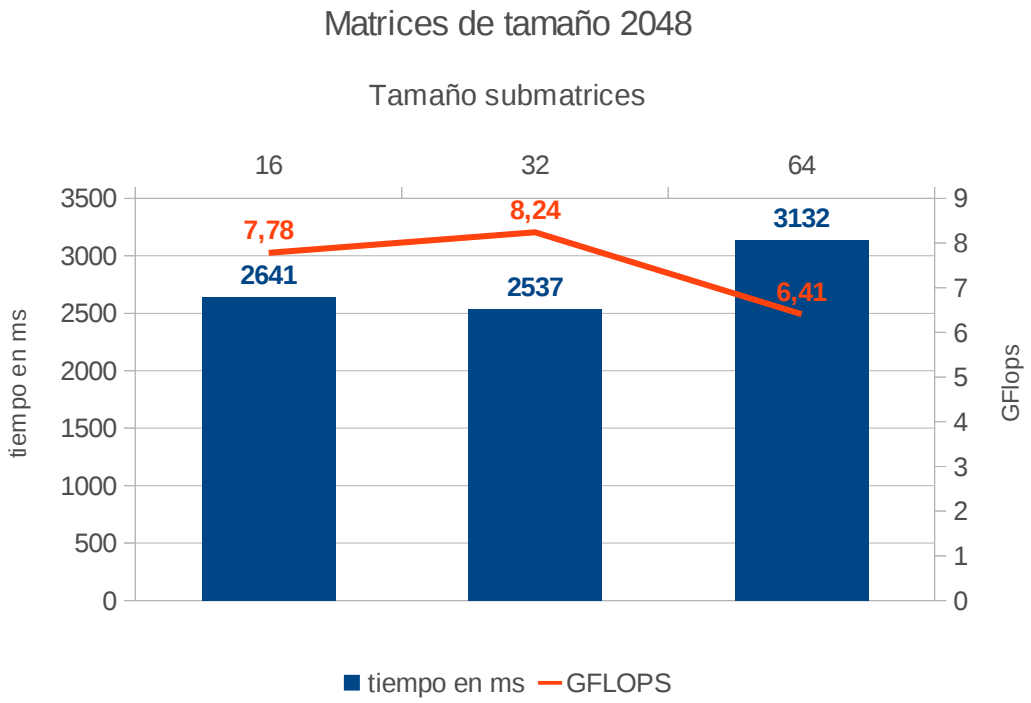
Las columnas azules muestran el tiempo en milisegundos utilizado en la ejecución del programa. La linea roja indica el número de Gflops alcanzado por el programa.

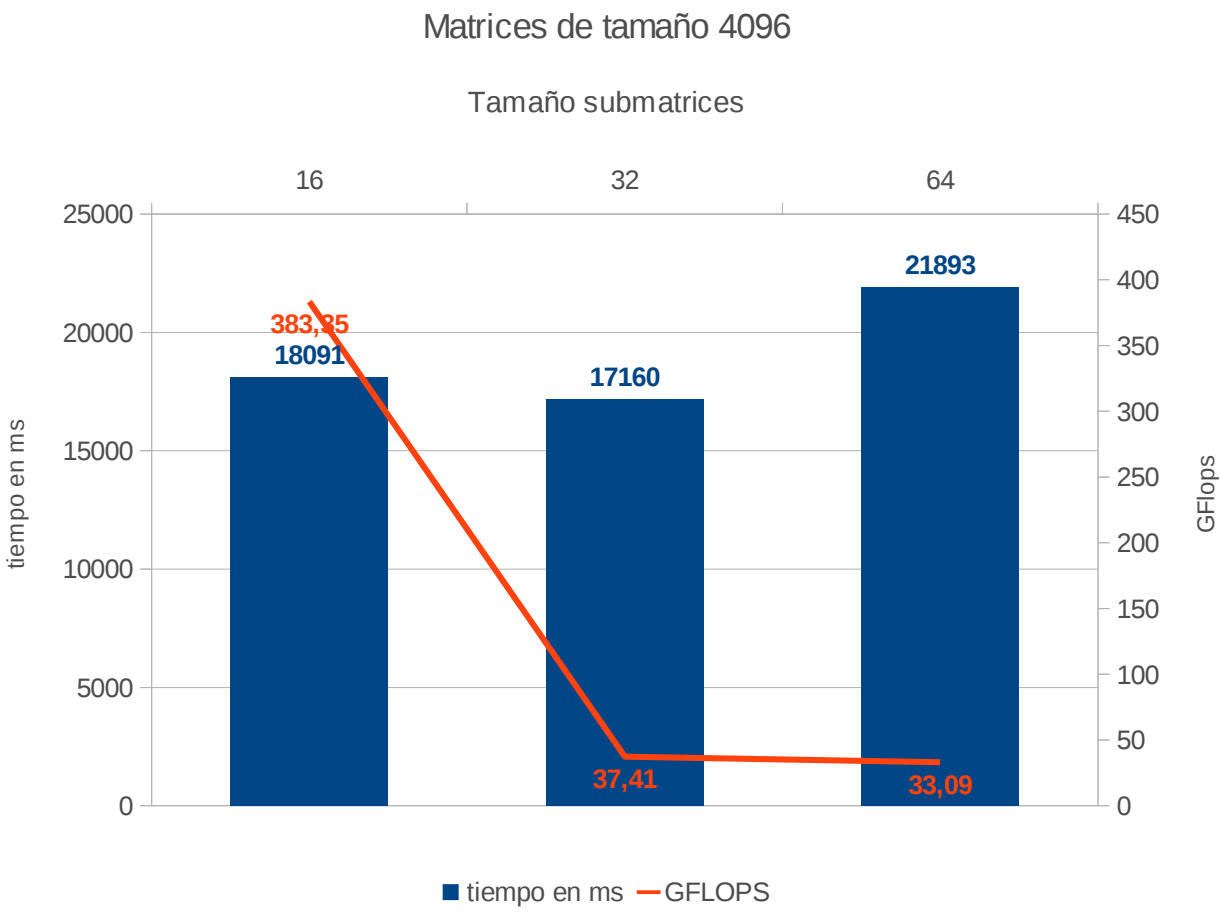




A continuación se muestran los resultados con tamaños de submatrices de 16x16, 32x32 y 64x64 valores.




Matrices de tamaño 2048
Tamaño submatrices
16
32
64
3500
3000
2500
2000
1500
1000
500
0
9
8
7
6
5
4
3
2
1
0
tiempo en ms
GFlops
7,78
8,24
6,41
2641
2537
3132
tiempo en ms
GFLOPS



Para submatrices de tamaño 128 se ha obtenido un error en la aplicación por tamaño insuficiente de recursos locales (Insufficient Local Resources!).

Para matrices de tamaño 8192 la tarjeta gráfica de pruebas no tiene suficiente memoria para almacenar las matrices en la memoria del dispositivo.

Como se observa, la multiplicación de matrices con tamaño de submatrices 32x32 es la que menor tiempo de ejecución obtiene para las pruebas realizadas con la tarjeta gráfica de pruebas. La multiplicación de matrices de 4096 con tamaño de submatrices 16x16 obtiene 383,35 Gflops ya que se utiliza el tiempo de ejecución del kernel para el calculo de este valor. El tiempo de ejecución del kernel sobre la GPU con tamaño de matrices 4096 y submatrices de 16x16 es: 358.519 ms y sobre submatrices de 32x32 es 3674.3 ms.

Se puede observar que el tamaño óptimo de las submatrices para la multiplicación de matrices en la tarjeta de pruebas es de 32x32.

7 Conclusiones, resumen y palabras clave.

7.1 Conclusiones y objetivos conseguidos.

Se ha conseguido hacer funcionar la tecnología OpenCL en el ordenador portátil de pruebas.

Se ha conseguido mejorar el programa progopen para que utilice la memoria interna del dispositivo y por tanto, obtener un mejor rendimiento.

Se ha conseguido obtener un rendimiento espectacular en programas donde su algoritmo puede ser altamente paralelizable. La mejora en el tiempo de ejecución del programa OpenCL es de 196 veces más rápido que el programa secuencial que utiliza la CPU con un tamaño de submatrices de 32x32. La mejora en el número de operaciones en punto flotante por segundo del programa OpenCL es de 8413 veces más rápido que el programa secuencial que utiliza la CPU usando un tamaño de submatrices de 16x16.

Se ha conseguido demostrar que los tamaños de las submatrices influyen en el rendimiento del programa. Se observa que el tamaño optimo de las submatrices para la tarjeta de pruebas es de 32x32, siendo el tamaño 16x16 el segundo más óptimo. El tamaño de submatrices 64x64 no proporciona el rendimiento deseado.

Por lo tanto, se demuestra que la tecnología GPGPU es una herramienta viable a utilizar en un futuro próximo como unidad de cómputo estándar. Se economiza el tiempo de ejecución y se optimiza la utilización de la unidad de cómputo.

7.2 Resumen

En este proyecto se han diseñado e implementado 3 programas que permiten demostrar la gran capacidad de cálculo en paralelo de la tecnología OpenCL debido a que las GPU's actuales disponen de un elevado número de núcleos de cálculo.

Se ha demostrado que se utilizan las GPU's de forma más eficiente que los procesadores (CPU) más actuales a un coste mucho menor, reduciendo drásticamente el tiempo de ejecución de las aplicaciones que utilizan OpenCL e incrementando las operaciones en punto flotante por segundo.

Para la implementación de estos programas se ha utilizado el SDK + IDE OpenCL v2.4 aunque la última versión a la hora de escribir este documento es OpenCL v2.5 que fue liberada el día 08/03/2011.

Para permitir la correcta ejecución de los programas es necesario disponer de una GPU soportada por OpenCL. En este mismo documento se muestra una tabla de las GPU's de AMD/ATI que son soportadas por OpenCL.

7.3 Palabras clave

OpenCL, GPGPU, multiplicación, matrices, ATI, AMD, NVIDIA, CUDA.

8 Bibliografía

Documentación OpenCL:

The OpenCL 1.1 Specification - Document Revision: 44. Khronos OpenCL Working Group.

AMD APP SDK v2.4 – Installation Notes.

Introduction to OpenCL Programming Training Guide Publication #: 137-41768-10 Rev: A Issue Date: May, 2010.

Webs:

http://gpgpu-computing4.blogspot.com/2009/09/matrix-multiplication-1.html

http://www.thebigblob.com/getting-started-with-opencl-and-gpu-computing/

http://developer.amd.com/zones/openclzone/pages/gettingstarted.aspx

http://www.codeproject.com/KB/showcase/Portable-Parallelism.aspx

http://my.safaribooksonline.com/book/programming/9780132488006

Información general:

http://gpgpu.org/

http://es.wikipedia.org/

http://www.khronos.org/opencl/

9 Anexo 1

progsec.c

```
#include <CL/cl.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>


#define widthA 2048
#define heightA 2048
#define widthB 2048
#define heightB 2048

// Generador de números aleatorios.
void randomInit(float* data, int size)
{
        for (int i = 0; i < size; ++i)
                data[i] = rand() / (float)RAND_MAX;
}

void secuencialMatrixMulCPU(int hA, int wA, int wB, float *A, float *B, float *C)
{
        int i, j, k;
        for (i=0; i<hA; i++)
        {
          for (j=0; j<wB; j++)
          {
            for(k=0; k<wA; k++)
            {
                C[i*hA+j] += A[i*hA+k] * B[k*wA+j];
            }
          }
        }
}


int main(int argc, char* argv[])
{

        // Semilla para srand()
        srand(2011);

        unsigned int mem_tam_A;
        unsigned int mem_tam_B;
        unsigned int mem_tam_C;
        float* h_A;
        float* h_B;
        float* h_C;

        struct timeval startTime, endTime;
        long secs, usecs, tiempotrans;
```

```
        double t;

        cl_uint hA, wA, wB, hB;
        hA = heightA;
        hB = heightB;
        wA = widthA;
        wB = widthB;


        // Creamos espacio en el host para las matrices A y B.
        unsigned int tam_A = widthA * heightB;
        mem_tam_A = sizeof(float) * tam_A;
        h_A = (float*) malloc(mem_tam_A);

        unsigned int tam_B = widthB * heightB;
        mem_tam_B = sizeof(float) * tam_B;
        h_B = (float*) malloc(mem_tam_B);

        // Creamos espacio en el host para la matriz resultado C.
        unsigned int tam_C = widthB * heightA;
        mem_tam_C = sizeof(float) * tam_C;
        h_C = (float*) malloc(mem_tam_C);

        // Inicializamos las matrices A y B con valores arbitrarios.
        randomInit(h_A, tam_A);
        randomInit(h_B, tam_B);

        // Multiplicación por el CPU de forma secuencial usando C.
        gettimeofday(&startTime, NULL);
        secuencialMatrixMulCPU(hA, wA, wB, h_A, h_B, h_C);
        gettimeofday(&endTime, NULL);

        /* Print performance numbers */
        printf("Multiplicacion de A[%d][%d] x B[%d][%d] = C[%d]
[%d]\n",hA,wA,hB,wB,hA,hB);
        secs = endTime.tv_sec - startTime.tv_sec;
        printf("Secs %ld\n", secs);
        usecs = endTime.tv_usec - startTime.tv_usec;
        tiempotrans = ((secs)*1000 + usecs/1000.0)+0.5;
        printf("Algoritmo secuencial CPU (ms) %ld\n",tiempotrans);

        float flops = 2 * wA * wB;
        float perf = (flops / secs) * hA * 1e-9;
        printf("GFlops alcanzados: %f\n", perf);

        free(h_A);
        free(h_B);
        free(h_C);

        return 0;
}
```

10 Anexo 2

progopen.c

```
#include <CL/cl.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

unsigned int mem_tam_A;
unsigned int mem_tam_B;
unsigned int mem_tam_C;
float* h_A;
float* h_B;
float* h_C;

// Codigo fuente del Kernel
const char *KernelSource = "\n"                                          \
  "__kernel void MatrixMultSimple(                                       \n" \
  "     __global float *mC,                                              \n" \
  "     __global float *mA,                                              \n" \
  "     __global float *mB,                                              \n" \
  "     const int Mdim, const int Ndim, const int Pdim)                  \n" \
  "{                                                                     \n" \
  "     int k;                                                           \n" \
  "     int i = get_global_id(0);                                        \n" \
  "     int j = get_global_id(1);                                        \n" \
  "     float sum = 0.0f;                                                \n" \
  "     // multiplicamos cada elemento de la row con cada                \n" \
  "     // elemento de las columnas apuntadas por el work-item actual    \n" \
  "     for (k=0; k< Pdim; k++)                                          \n" \
  "             sum += mA[i * Ndim + k] * mB[k * Pdim + j];              \n" \
  "     // copiamos la suma al buffer del dispositivo.                   \n" \
  "     mC[i * Ndim + j] = sum;                                          \n" \
  "}                                                                     \n" \
  "\n";

// Generador de numeros aleatorios.
void randomInit(float* data, int size)
{
        for (int i = 0; i < size; ++i)
                data[i] = rand() / (float)RAND_MAX;
}
// Imprime resultados
void muestraResultadoskernel(cl_ulong startTime, cl_ulong endTime, cl_uint hA,
cl_uint wA, cl_uint hB, cl_uint wB, int tipoDisp)
{
        double sec, flops, perf;
        printf("Multiplicacion de A[%d][%d] x B[%d][%d] = C[%d]
[%d]\n",hA,wA,hB,wB,hA,hB);

        sec = 1e-9 * (endTime - startTime);
        printf("Tiempo de Kernel (ms) : %G\n" ,sec * 1000);
```

```
        flops = 2 * wA * wB;
        perf = (flops / sec) * hA * 1e-9;

        printf("GFlops alcanzados : %G\n", perf);
}

void muestraResultados(timeval startTime, timeval endTime, cl_uint hA, cl_uint
wA, cl_uint hB, cl_uint wB, int tipoDisp)
{
        long secs, usecs, tiempotrans;
        if(tipoDisp == 5)
        {
                secs = endTime.tv_sec - startTime.tv_sec;
                usecs = endTime.tv_usec - startTime.tv_usec;

                tiempotrans = ((secs)*1000 + usecs/1000.0)+0.5;
                printf("CPU --> Tiempo total en ms %ld\n\n",tiempotrans);
                printf("**********************************************************
********\n\n");
        }
        else
        {
                secs = endTime.tv_sec - startTime.tv_sec;
                usecs = endTime.tv_usec - startTime.tv_usec;

                tiempotrans = ((secs)*1000 + usecs/1000.0)+0.5;
                printf("GPU --> Tiempo total en ms %ld\n\n",tiempotrans);

        }

}
// Codigo OpenCL
void MatMultOpencl(cl_uint Mdim, cl_uint Pdim, cl_uint Ndim, int tipoDisp)
{
        // Variables OPENCL
        cl_uint num_devs_returned=0;
        cl_context_properties properties[3];
        cl_device_id device_id;
        cl_int err;
        cl_platform_id platform_id;
        cl_uint num_platforms_returned=0;
        cl_uint start=0;
        cl_uint end=0;
        cl_context context;
        cl_command_queue command_queue;
        cl_program program;
        cl_kernel kernel;
        cl_mem input_buffer1, input_buffer2, output_buffer;
        size_t global[2];
        cl_event events[2];

        // Obtenemos una lista de plataformas disponibles.
        err = clGetPlatformIDs(1, &platform_id, &num_platforms_returned);
        if (err != CL_SUCCESS)
        {
                printf("No es posible encontrar una plataforma vÃ¡lida\n");
```

```
                exit(1);
        }

        // Intentamos obtener un dispositivo soportado por la GPU o la CPU.
        err = clGetDeviceIDs(platform_id, tipoDisp, 1, &device_id,
&num_devs_returned); //Antes: clGetDeviceIDs(platform_id, devType, 1,
&device_id, NULL);
        if (err != CL_SUCCESS)
        {
                printf("Imposible obtener un dispositivo válido\n");
                exit(1);
        }

        // lista de propiedades del contexto - debe terminar con 0.
        properties[0]= CL_CONTEXT_PLATFORM;
        properties[1]= (cl_context_properties) platform_id;
        properties[2]= 0;

        // Creamos un contexto con el dispositivo encontrado
        context = clCreateContext(properties, 1, &device_id, NULL, NULL, &err);
        if(!context)
        {
                printf("Imposible crear un contexto con el dispositivo
encontrado\n");
                exit(1);
        }

        // Creamos una cola de comandos usando el contexto y el dispositivo
        command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &err);
        if(!command_queue)
        {
                printf("Fallo al crear la cola de comandos\n");
                exit(1);
        }

        // Cargamos el condigo fuente del kernel
        program = clCreateProgramWithSource(context, 1, (const char **)
&KernelSource, NULL, &err);
        if(!program)
        {
                printf("Fallo al crear el programa usando el codigo fuente del
kernel\n");
                exit(1);
        }

        // Compilamos el programa
        err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
        if (err != CL_SUCCESS)
        {
                printf("Error número %d",err);
                printf("Error contruyendo el programa\n");

                char buffer[4096];
                size_t length;

                clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
```

```
sizeof(buffer), buffer, &length);
                printf("%s\n",buffer);

                exit(1);
        }

        kernel = clCreateKernel(program, "MatrixMultSimple", &err);
        if(!kernel || err != CL_SUCCESS)
        {
                printf("%d",err);
                printf("Fallo al crear el kernel computacional!\n");

                exit(1);
        }


        // Creamos espacio en el dispositivo para las matrices y asignamos a las
matrices de entrada los datos aletorios anteriormente obtenidos.
        cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR, mem_tam_A, h_A, &err);
        if(err != CL_SUCCESS)
        {
                printf("Fallo al crear espacio para la matriz A en el
dispositivo\n");
                exit(1);
        }
        cl_mem d_B = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR, mem_tam_B, h_B, &err);
        if(err != CL_SUCCESS)
        {
                printf("Fallo al crear espacio para la matriz B en el
dispositivo\n");
                exit(1);
        }
        cl_mem d_C = clCreateBuffer(context, CL_MEM_READ_WRITE, mem_tam_C, NULL,
&err);
        if(err != CL_SUCCESS)
        {
                printf("Fallo al crear espacio para la matriz C en el
dispositivo\n");
                exit(1);
        }

        // Parametros del kernel
        clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_C);
        clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_A);
        clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_B);
        clSetKernelArg(kernel, 3, sizeof(int), &Mdim);
        clSetKernelArg(kernel, 4, sizeof(int), &Ndim);
        clSetKernelArg(kernel, 5, sizeof(int), &Pdim);

        global[0] = (size_t)Ndim;
        global[1] = (size_t)Mdim;

        // Ponemos en cola el comando kernel para ejecucion
        err = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global,
NULL, 0, NULL, &events[0]);
```

```
        if(err != CL_SUCCESS)
        {
                printf("%d",err);
                printf("Fallo al poner el kernel en la cola de ejecución.\n");
                exit(1);
        }

        // Esperamos a que termine de ejecutarse todas las operaciones de la
cola de comandos.

        // Codigo para realizar el profiling y calculo de tiempos del kernel.
        clWaitForEvents(1, &events[0]);

        /* Calculo de la performance */
        cl_ulong startTime;
        cl_ulong endTime;

        // Obtiene tiempo de inicio del kernel
        clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_START,
sizeof(cl_ulong), &startTime, 0);
        // Obtiene tiempo de finalización del kernel
        clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &endTime, 0);

        // Informa por pantalla de los resultados del kernel
        muestraResultadoskernel(startTime, endTime, Ndim, Pdim, Pdim, Mdim,
tipoDisp);


        // Recogemos los resultados del dispositivo y los ponemos en la matriz
host C.
        err = clEnqueueReadBuffer(command_queue, d_C, CL_TRUE, 0, mem_tam_C,
h_C, 0, NULL, &events[1]);
        if (err != CL_SUCCESS)
        {
                printf("Fallo al leer el resultado de la matriz C del
dispositivo\n");
                exit(1);
        }

        // Espera a que acaben todos los eventos.
        clWaitForEvents(1, &events[1]);

        //Libera el evento de lectura.
        clReleaseEvent(events[1]);

        // limpieza.
        clReleaseMemObject(d_A);
        clReleaseMemObject(d_B);
        clReleaseMemObject(d_C);
        clReleaseProgram(program);
        clReleaseKernel(kernel);
        clFlush(command_queue);
        clReleaseCommandQueue(command_queue);
        clReleaseContext(context);

}
```

```
int main(int argc, char* argv[])
{
        // Semilla para srand()
        srand(2011);

        struct timeval startTime, endTime;
        double t;
        int Mdim, Ndim, Pdim;
        const char * argumento = argv[1];

        if(argc == 2)
        {
                Mdim = atoi(argumento);
                Ndim = atoi(argumento);
                Pdim = atoi(argumento);
                printf("El tamaño de las matrices es: %d\n\n", Ndim);
        }
        else
        {
                Mdim = 1024;
                Ndim = 1024;
                Pdim = 1024;
        }
        // Creamos espacio en el host para las matrices A y B.
        unsigned int tam_A = Ndim * Pdim;
        mem_tam_A = sizeof(float) * tam_A;
        h_A = (float*) malloc(mem_tam_A);

        unsigned int tam_B = Pdim * Mdim;
        mem_tam_B = sizeof(float) * tam_B;
        h_B = (float*) malloc(mem_tam_B);

        // Creamos espacio en el host para la matriz resultado C.
        unsigned int tam_C = Ndim * Mdim;
        mem_tam_C = sizeof(float) * tam_C;
        h_C = (float*) malloc(mem_tam_C);

        // Inicializamos las matrices A y B con valores arbitrarios.
        randomInit(h_A, tam_A);
        randomInit(h_B, tam_B);

        /*****************************************/
        /* Multiplicación por el CPU usando OpenCL.*/
        /*****************************************/
        gettimeofday(&startTime, NULL);
        MatMultOpencl(Ndim, Pdim, Mdim, CL_DEVICE_TYPE_CPU);
        gettimeofday(&endTime, NULL);

        muestraResultados(startTime, endTime, Ndim, Pdim, Pdim, Mdim);

        /*****************************************/
        /* Multiplicación por el GPU usando OpenCL.*/
        /*****************************************/
        gettimeofday(&startTime, NULL);
        MatMultOpencl(Ndim, Pdim, Mdim, CL_DEVICE_TYPE_GPU);
        gettimeofday(&endTime, NULL);
```

```
        muestraResultados(startTime, endTime, Ndim, Pdim, Pdim, Mdim);

        free(h_A);
        free(h_B);
        free(h_C);

        return 0;
}
```

11 Anexo 3

progopenimp.c

```
#include <CL/cl.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

#define BLOCK_SIZE 16

unsigned int mem_tam_A;
unsigned int mem_tam_B;
unsigned int mem_tam_C;
float* h_A;
float* h_B;
float* h_C;

// Codigo fuente del Kernel
const char *KernelSource = "\n"                                  \
  "__kernel void matrixMul(                                                  \n"\
  "     __global float* mC,                                                  \n"\
  "     __global float* mA,                                                  \n"\
  "     __global float* mB, int wA, int wB, int block_size)                  \n"\
  "{                                                                         \n"\
  "  // indice de bloque                                                     \n"\
  "  int bx = get_group_id(0);                                               \n"\
  "  int by = get_group_id(1);                                               \n"\
  "  // indice de hilo                                                       \n"\
  "  int tx = get_local_id(0);                                               \n"\
  "  int ty = get_local_id(1);                                               \n"\
  "  // indice de la primera sub-matriz de A procesada                       \n"\
  "  // por el bloque                                                        \n"\
  "  int aBegin = wA * block_size * by;                                      \n"\
  "  // indice de la última sub-matriz de A procesada                        \n"\
  "  // por el bloque                                                        \n"\
  "  int aEnd   = aBegin + wA - 1;                                           \n"\
  "  // Tamaño usado para iterar sobre las sub-matrices de A                 \n"\
  "  int aStep  = block_size;                                                \n"\
  "  // indice de la primera sub-matriz de A procesada                       \n"\
  "  // por el bloque                                                        \n"\
  "  int bBegin = block_size * bx;                                           \n"\
  "  // Tamaño usado para iterar sobre las sub-matrices de B                 \n"\
  "  int bStep  = block_size * wB;                                           \n"\
  "  float Csub = 0;                                                         \n"\
  "  // Bucle sobre todas las sub-matrices de A y de B                       \n"\
  "  // requeridas para realizar el computo de bloque de sub-matriz         \n"\
  "  for (int a = aBegin, b = bBegin;                                        \n"\
  "            a <= aEnd;                                                    \n"\
  "            a += aStep, b += bStep)                                       \n"\
  "  {                                                                       \n"\
  "      // Declaración del array A en memoria local                        \n"\
  "      // usado para almacenar las sub-matrices de A                      \n"\
```

```
  "      __local float As[16][16];                                    \n"\
  "      // Declaración del array B en memoria local                  \n"\
  "      // usado para almacenar las sub-matrices de B                \n"\
  "      __local float Bs[16][16];                                    \n"\
  "      // Carga las matrices desde la memoria global                \n"\
  "      // a la memoria local; cada hilo carga                       \n"\
  "      // un elemento de cada matriz                                \n"\
  "      As[ty][tx] = mA[a + wA * ty + tx];                           \n"\
  "      Bs[ty][tx] = mB[b + wB * ty + tx];                           \n"\
  "      // Sincronización para asegurarnos que las matrices          \n"\
  "      // han sido cargadas                                         \n"\
  "      barrier(CLK_LOCAL_MEM_FENCE);                                \n"\
  "      // Multiplicación de las dos matrices juntas;                \n"\
  "      // cada hilo computa un elemento                             \n"\
  "      // del bloque sub-matriz                                     \n"\
  "      for (int k = 0; k < block_size; ++k)                         \n"\
  "          Csub += As[ty][k] * Bs[k][tx];                           \n"\
  "      // Sincronización para estar seguros que la                  \n"\
  "      // computación anterior está hecha antes de cargar dos nuevas\n"\
  "      // sub-matrices de A y B en la próxima iteración             \n"\
  "      barrier(CLK_LOCAL_MEM_FENCE);                                \n"\
  "  }                                                                \n"\
  "  // Escribir la sub-matriz en la memoria del dispositivo          \n"\
  "  // cada hilo escribe un elemento                                 \n"\
  "  int c = wB * block_size * by + block_size * bx;                  \n"\
  "  mC[c + wB * ty + tx] = Csub;                                     \n"\
  "}                                                                  \n"\
  "\n";

// Generador de numeros aleatorios.
void randomInit(float* data, int size)
{
        for (int i = 0; i < size; ++i)
                data[i] = rand() / (float)RAND_MAX;
}
// Imprime resultados
void muestraResultadoskernel(cl_uint startTime, cl_uint endTime, cl_uint hA,
cl_uint wA, cl_uint hB, cl_uint wB)
{
        double sec, flops, perf;
        printf("Multiplicacion de A[%d][%d] x B[%d][%d] = C[%d]
[%d]\n",hA,wA,hB,wB,hA,hB);
        printf("**************************************************************
\n");
        sec = 1e-9 * (endTime - startTime);
        printf("Tiempo de Kernel (ms) : %G\n" ,sec * 1000);

        flops = 2 * wA * wB;
        perf = (flops / sec) * hA * 1e-9;

        printf("GFlops alcanzados : %G\n", perf);
}

void muestraResultados(timeval startTime, timeval endTime, cl_uint hA, cl_uint
wA, cl_uint hB, cl_uint wB)
{
        long secs, usecs, tiempotrans;
```

```
        secs = endTime.tv_sec - startTime.tv_sec;
        usecs = endTime.tv_usec - startTime.tv_usec;

        tiempotrans = ((secs)*1000 + usecs/1000.0)+0.5;
        printf("GPU --> Tiempo total en ms %ld\n\n",tiempotrans);

}
// Codigo OpenCL
void MatMultOpencl(cl_uint hA, cl_uint wA, cl_uint wB, int tipoDisp)
{
        // Variables OPENCL
        cl_uint num_devs_returned=0;
        cl_context_properties properties[3];
        cl_device_id device_id;
        cl_int err;
        cl_platform_id platform_id;
        cl_uint num_platforms_returned=0;
        cl_uint start=0;
        cl_uint end=0;
        cl_uint block_size=16;
        cl_context context;
        cl_command_queue command_queue;
        cl_program program;
        cl_kernel kernel;
        cl_mem input_buffer1, input_buffer2, output_buffer;
        size_t global[2];
        size_t local[2];
        cl_event events[2];

        // Obtenemos una lista de plataformas disponibles.
        err = clGetPlatformIDs(1, &platform_id, &num_platforms_returned);
        if (err != CL_SUCCESS)
        {
                printf("No es posible encontrar una plataforma válida\n");
                exit(1);
        }

        // Intentamos obtener un dispositivo soportado por la GPU o la CPU.
        err = clGetDeviceIDs(platform_id, tipoDisp, 1, &device_id,
&num_devs_returned);
        if (err != CL_SUCCESS)
        {
                printf("Imposible obtener un dispositivo válido\n");
                exit(1);
        }

        // lista de propiedades del contexto - debe terminar con 0.
        properties[0]= CL_CONTEXT_PLATFORM;
        properties[1]= (cl_context_properties) platform_id;
        properties[2]= 0;

        // Creamos un contexto con el dispositivo encontrado
        context = clCreateContext(properties, 1, &device_id, NULL, NULL, &err);
        if(!context)
        {
                printf("Imposible crear un contexto con el dispositivo
```

```
encontrado\n");
                exit(1);
        }

        // Creamos una cola de comandos usando el contexto y el dispositivo
        command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &err);
        if(!command_queue)
        {
                printf("Fallo al crear la cola de comandos\n");
                exit(1);
        }

        // Cargamos el condigo fuente del kernel
        program = clCreateProgramWithSource(context, 1, (const char **)
&KernelSource, NULL, &err);
        if(!program)
        {
                printf("Fallo al crear el programa usando el codigo fuente del
kernel\n");
                exit(1);
        }

        // Compilamos el programa
        err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
        if (err != CL_SUCCESS)
        {
                printf("Error número %d\n",err);
                printf("Error contruyendo el programa\n");

                char buffer[4096];
                size_t length;

                clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
sizeof(buffer), buffer, &length);
                printf("%s\n",buffer);

                exit(1);
        }

        kernel = clCreateKernel(program, "matrixMul", &err);
        if(!kernel || err != CL_SUCCESS)
        {
                printf("%d",err);
                printf("Fallo al crear el kernel computacional!\n");

                exit(1);
        }


        // Creamos espacio en el dispositivo para las matrices y asignamos a las
matrices de entrada los datos aletorios anteriormente obtenidos.
        cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR, mem_tam_A, h_A, &err);
        if(err != CL_SUCCESS)
        {
                printf("Fallo al crear espacio para la matriz A en el
```

```
dispositivo\n");
                exit(1);
        }
        cl_mem d_B = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR, mem_tam_B, h_B, &err);
        if(err != CL_SUCCESS)
        {
                printf("Fallo al crear espacio para la matriz B en el
dispositivo\n");
                exit(1);
        }
        cl_mem d_C = clCreateBuffer(context, CL_MEM_READ_WRITE, mem_tam_C, NULL,
&err);
        if(err != CL_SUCCESS)
        {
                printf("Fallo al crear espacio para la matriz C en el
dispositivo\n");
                exit(1);
        }

        global[0] = (size_t)hA;
        global[1] = (size_t)wB;

        local[0] = BLOCK_SIZE;
        local[1] = BLOCK_SIZE;

        // Parametros del kernel
        clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_C);
        clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_A);
        clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_B);
        clSetKernelArg(kernel, 3, sizeof(int), &wA);
        clSetKernelArg(kernel, 4, sizeof(int), &wB);
        clSetKernelArg(kernel, 5, sizeof(int), &block_size);

        // Ponemos en cola el comando kernel para ejecucion
        err = clEnqueueNDRangeKernel(command_queue, kernel, 2, local, global,
NULL, 0, NULL, &events[0]);
        if(err != CL_SUCCESS)
        {
                printf("%d",err);
                printf("Fallo al poner el kernel en la cola de ejecuciÃ³n.\n");
                exit(1);
        }

        // Esperamos a que termine de ejecutarse todas las operaciones de la
cola de comandos.
        //clFinish(command_queue);

        // Codigo para realizar el profiling y calculo de tiempos del kernel.
        clWaitForEvents(1, &events[0]);

        /* Calculo de la performance */
        cl_ulong startTime;
        cl_ulong endTime;

        // Obtiene tiempo de inicio del kernel
        clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_START,
```

```
sizeof(cl_ulong), &startTime, 0);

        // Obtiene tiempo de finalizacion del kernel
        clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_END,
sizeof(cl_ulong), &endTime, 0);

        // Informa por pantalla de los resultados del kernel
        muestraResultadoskernel(startTime, endTime, hA, wA, wA, wB);

        // Recogemos los resultados del dispositivo y los ponemos en la matriz
host C.
        err = clEnqueueReadBuffer(command_queue, d_C, CL_TRUE, 0, mem_tam_C,
h_C, 0, NULL, &events[1]);
        if (err != CL_SUCCESS)
        {
                printf("Fallo al leer el resultado de la matriz C del
dispositivo\n");
                exit(1);
        }

        // Espera a que acaben todos los eventos.
        clWaitForEvents(1, &events[1]);

        //Libera el evento de lectura.
        clReleaseEvent(events[1]);

        // limpieza.
        clReleaseMemObject(d_A);
        clReleaseMemObject(d_B);
        clReleaseMemObject(d_C);
        clReleaseProgram(program);
        clReleaseKernel(kernel);
        clFlush(command_queue);
        clReleaseCommandQueue(command_queue);
        clReleaseContext(context);

}

int main(int argc, char* argv[])
{
        // Semilla para srand()
        srand(2011);

        struct timeval startTime, endTime;
        double t;
        const char * argumento = argv[1];

        cl_uint hA, wA, wB, hB;


        if(argc == 2)
        {
                hA = atoi(argumento);
                wA = atoi(argumento);
                hB = atoi(argumento);
                wB = atoi(argumento);
                printf("El tamaño de las matrices es: %d\n\n", hA);
```

```
        }
        else
        {
                hA = 1024;
                hB = 1024;
                wA = 1024;
                wB = 1024;
        }


        // Creamos espacio en el host para las matrices A y B.
        unsigned int tam_A = hA * wA;
        mem_tam_A = sizeof(float) * tam_A;
        h_A = (float*) malloc(mem_tam_A);

        unsigned int tam_B = hB * wB;
        mem_tam_B = sizeof(float) * tam_B;
        h_B = (float*) malloc(mem_tam_B);

        // Creamos espacio en el host para la matriz resultado C.
        unsigned int tam_C = hA * wB;
        mem_tam_C = sizeof(float) * tam_C;
        h_C = (float*) malloc(mem_tam_C);

        // Inicializamos las matrices A y B con valores arbitrarios.
        randomInit(h_A, tam_A);
        randomInit(h_B, tam_B);

        /*****************************************/
        /* Multiplicación por el GPU usando OpenCL.*/

        /*****************************************/
        gettimeofday(&startTime, NULL);
        MatMultOpencl(hA, wA, wB, CL_DEVICE_TYPE_GPU);
        gettimeofday(&endTime, NULL);

        muestraResultados(startTime, endTime, hA, wA, hB, wB);

        free(h_A);
        free(h_B);
        free(h_C);

        return 0;
}
```