2430 : DESARROLLO DE SOFTWARE PARA EL PROCESADO NUMÉRICO EN TARJETAS GRÁFICAS

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
Ero Rodríguez Losada
i dirigit per
Xavier Cartoixà Soler

Bellaterra, 09 de Septiembre de 2010

Universitat Autònoma de Barcelona

etse
Escola Tècnica Superior d'Enginyeria

El sotasignat, ...

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en

Ero Rodríguez Losada

I per tal que consti firma la present.

Signat: .....

Bellaterra, 09 de Setembre de 2010

Índice

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1. Motivación

Dada la creciente capacidad de cálculo que vienen ofreciendo los procesadores de las tarjetas gráficas en los últimos años (de ahora en adelante Graphics Processing Units, GPUs) del mercado doméstico, de hecho, creciendo muy por encima de la Ley de Moore [1] que establece el ratio de crecimiento de rendimiento para las CPUs, en el presente proyecto se pretende ofrecer un solución para el aprovechamiento de esta capacidad con fines no enfocados a la composición y renderizado de gráficos, sino a la especialización de cálculos sobre matrices dispersas.

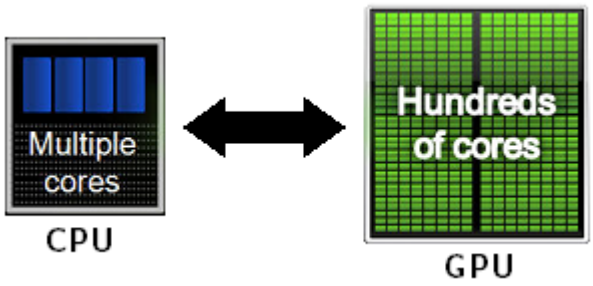


Figura 1.1: Cores CPU vs GPU

En los últimos años ha existido una creciente especial atención de la comunidad científica en esta área, impulsando numerosas investigaciones en el uso de hardware grafico con otros fines, en especial en campos como la dinámica molecular en el plegamiento de proteínas [2], el procesado de señales - FFT [3], el análisis secuencias biológicas mediante Hidden Markov Model [4] y un largo etcétera, sin olvidar el campo que nos concierne, las operaciones sobre grandes matrices/vectores (BLAS, que veremos en el capítulo posterior).

2. Objetivos

Se establece así como objetivo principal de este proyecto la obtención de un conjunto de rutinas que puedan ser usadas en forma de librería especializada en el cómputo del producto de matrices dispersas por vector (spMV). Estas rutinas se encuentran en el corazón de los algorísmos iterativos en álgebra lineal para la resolución de grandes sistemas de ecuaciones con origen en un amplio abanico de campos del ámbito científico y tecnológico. Aunque se buscará la optimización para todo tipo de matrices dispersas, gran parte de las pruebas y el esfuerzo se centrará en el tipo de matrices dispersas hermíticas, lo cual implicará dar soporte para datos de precisión float, double y sus análogos para números complejos. Estas rutinas podrán ser ejecutadas sobre cualquier plataforma de procesamiento que se encuentre disponible en la máquina, usando únicamente la CPU o la GPU.

Con el objetivo de poder ejecutar estas librerías sobre el máximo número de plataformas posibles, todo el código será programado bajo el framework Open Computing Language (OpenCL) creado inicialmente por Apple Inc. y convertido en estándar abierto y libre de licencias por Khronos Group, liberando la primera versión estable en diciembre de 2008.

Todo el código aquí desarrollado será liberado posteriormente bajo licencia GPLv2 [5].

Como objetivos específicos del proyecto, estableceremos:

- Implementación rutinas para la multiplicación de matrices dispersas por vector sobre la GPU.
- Para cada una de las rutinas anteriores daremos soporte para datos de tipo float, double, float complex y double complex.
- Realización estudio de comparativas de las rutinas anteriores sobre la GPU, con sus diferentes formatos de almacenamiento de datos, con sus equivalentes desarrollados en BLAS (Basic Linear Algebra Subprograms) sobre CPU.

Capítulo 2

Sparse Matrix Vector Multiplication

1. Representación de Matrices Sparse

1.1. Introducción

Las matrices dispersas se encuentran en un amplio abanico de problemas en campos científicos como física e ingenierías, muy comúnmente en la resolución de grandes sistemas numéricos en los que participan ecuaciones diferenciales parciales. En muchos de los métodos usados para resolver estos sistemas, el producto sparse Matrix-Vector es la operación básica. Es por ello que actualmente varias investigaciones y proyectos que se desarrollan en este campo están enfocadas en la optimización de ésta operación.

1.1.1. Métodos de resolución de sistemas

Habitualmente nos encontramos con el problema de solucionar sistemas de ecuaciones lineales:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= k_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= k_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= k_n \end{aligned}$$

que escritos en forma matricial podría ser $Ax = k$, donde A sería la matriz con los coeficientes (a_{ij}), x y k los vectores $(x_1 \ldots x_n)$ y $(k_1 \ldots k_n)$ respectivamente. Asumiendo esta notación, existen varios métodos directos e iterativos desarrollados y ampliamente usados en sistemas computacionales para obtener una solución en un tiempo finito. En nuestro caso, prestaremos especial atención a los métodos iterativos pues se pueden aplicar a los sistemas dispersos que son demasiado grandes para ser tratados por métodos directos como la descomposición de Cholesky, y en especial veremos como ejemplo el método del gradiente conjugado para ver la motivación del presente proyecto.

Ejemplo: Conjugate Gradient (CG) El método del gradiente conjugado ha recibido mucha atención y ha sido ampliamente utilizado en años recientes. Aunque los pioneros de este método fueron Hestenes y Stiefel (1952) [6], el interés actual arranca a partir de que Reid (1971) [7] lo planteara como un método iterativo, que es la forma en que se le usa con mayor frecuencia en la actualidad. El método CG es un método iterativo que se emplea para obtener la solución en como máximo n iteraciones siempre que no se encuentren errores de redondeo (donde n es el orden de la matriz) en sistemas donde la matriz A sea simétrica y definida positiva. Dado el origen del sistema, es habitual que en la mayoría de las ocasiones la matriz sea definida positiva, con lo cual este prerrequisito no representa un problema. Del mismo modo existen variaciones del método original permitiendo ser empleado sobre matrices que no sean simétricas El algoritmo básico de este método sería:

- Paso inicial: Seleccionamos un vector de solución estimada x_0 y calculamos el resido r_0 y la dirección p_0.

$$p_0 = r_0 = k - Ax_0 \quad (x_0 \text{ arbitrario}) \tag{1.1}$$

- Pasos generales: Habiendo determinado la estimación x_i, el resido r_i, y la dirección p_i, calculamos x_{i+1}, r_{i+1}, y p_{i+1}.

  Si $(x, y) = (x_1y_1 + x_2y_2 + \ldots + x_ny_n)$ es el producto escalar,

$$|x| = \sqrt{x_1^2 + \ldots + x_n^2} = \sqrt{(x, x)} \tag{1.2}$$

$$a_i = \frac{|r_i|^2}{(p_i, Ap_i)} \tag{1.3}$$

$$x_{i+1} = x_i + a_ip_i \tag{1.4}$$

$$r_{i+1} = r_i + a_iAp_i \tag{1.5}$$

$$b_i = \frac{|r_{i+1}|^2}{|r_i|^2} \tag{1.6}$$

$$p_{i+1} = r_{i+1} + b_ip_i \tag{1.7}$$

  y así sucesivamente hasta dar con un valor r_m suficientemente pequeño como para darlo como aproximación valida a la solución h del sistema. Claramente, optimizando el producto Ap_i que es empleado en cada iteración, obtendremos un tiempo de respuesta significativamente mejor en el cómputo global del algoritmo.

1.2. Matrix Storage

> En el campo del análisis numérico una matriz dispersa es una matriz poblada principalmente por ceros.
>
> Harry M. Markowitz.

Auque no existe una definición única de lo que es una matriz dispersa, podríamos definirla como una matriz de grandes dimensiones cuyos elementos diferentes de cero crecen linealmente en relación con el orden de la matriz, y que habitualmente se encuentran entorno a la diagonal

de dicha matriz. Aunque en nuestro caso usualmente trataremos con matrices hermíticas, y con el objetivo que extender la optimización del producto al mayor tipo de matrices, nos centraremos en aprovechar las propiedades de las matrices dispersas, dejando a un lado para una posible etapa posterior de optimización las propiedades que presentan las matrices hermíticas. Con el objetivo de obtener alguna ventaja del gran número de elementos igual a cero, se requerirán esquemas especiales para su almacenamiento, los cuales se presentaran a continuación. La clave principal en que se basan estos esquemas es en el no almacenamiento de los elementos cero, y que a su vez permita realizar las operaciones típicas sobre matrices. Dado que únicamente necesitaremos realizar cálculos aritméticos con los valores diferentes de cero y teniendo en cuenta el alto coste de almacenamiento para su posterior acceso (más aún en el caso de GPUs, donde tendremos menor capacidad de almacenamiento aunque a una latencia mucho mas baja) buscaremos aprovechar el principio de localidad y es por ello que uno de los puntos clave a tener en cuenta será la forma de almacenamiento de dicha matriz, la cual a su vez impondrá ciertas restricciones en el algoritmo a usar para obtener el producto matricial.

1.2.1. Coordinate Storage (COO)

Uno de los métodos más simples y usados es el esquema Coordinate Storage. Dado N el número total de valores diferentes de cero, la implementación típica de este método usa tres arrays unidimensionales de tamaño N. El primer array contendrá los valores diferentes de zero (AA). Los otros dos, siempre de valores enteros, contendrán los correspondientes índices de fila (JR) y columna (JC) para cada elemento. La matriz:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 0 \\ 0 & 0 & 7 & 0 \\ 0 & 0 & 8 & 9 \end{pmatrix} \tag{1.8}$$

se representará como:

$$\begin{array}{ll} AA & \boxed{1\,2\,3\,4\,5\,6\,7\,8\,9} \\ JR & \boxed{0\,0\,0\,0\,1\,1\,2\,3\,3} \\ JC & \boxed{0\,1\,2\,3\,1\,2\,2\,2\,3} \end{array}$$

Este esquema nos permite almacenar los valores en un orden arbitrario, aunque habitualmente es común almacenarlos listándolos por fila o por columna, en cuyo caso tendremos información redundante de lo cual surge una mejora que introduce el siguiente esquema.

1.2.2. Compressed Row Storage(CRS)

Usado por el popular Matlab, éste es quizás el método más empleado y del cual se basan una gran variedad de métodos propuestos por diferentes equipos de investigación con el objetivo de

optimizarlo para una plataforma en concreto. Si listamos los elementos del vector AA por fila, mantenemos el vector JC y reemplazamos el vector JR por un vector de enteros que contenga la posición de inicio de cada fila en el vector JC, obtenemos el esquema CRS,

$$AA\ \boxed{1\,2\,3\,4\,5\,6\,7\,8\,9}$$
$$JC\ \boxed{0\,1\,2\,3\,1\,2\,2\,2\,3}$$
$$IA\ \boxed{0\,4\,6\,7\,9}$$

Podemos observar que este nuevo vector tendrá $m+1$ elementos. Esto es debido a que su último elemento representará una columna ficticia que será necesaria para calcular el número de elementos de la fila m de la matriz. Un algoritmo básico para la ejecución del producto spMV con este esquema de forma secuencial podría ser:

```
for (int i = 0; i < m ; i++) {
        // obtenemos indice superior
        isuperior = IA[i+1];
        // bucle sobre fila i
        for (int j = IA[i]; j < isuperior; j++) {
                b[i] = b[i] + AA[j] * v[JC[j]];
        }
}
```

Siendo m el número de $filas+1$, b el vector resultante y v el vector del producto MxV.

1.2.3. Compressed Column Storage(CCS)

Una de las variaciones más comunes y obvias que se desprenden del esquema CRS es el llamado esquema Compressed Column Storage (CCS). Consiste en almacenar la matriz AA por columnas, sustituyendo así el vector JC por JR e igualmente generando el vector IA, en este caso indicando el cominzo de cada columna en el vector JR.

1.2.4. Modified Row Storage (MRS)

Esta otra variación explota el hecho de que generalmente los elementos diferentes de cero se encuentran en la diagonal y a su vez su acceso es mas frecuente. Como resultado de estas observaciones, el esquema propone un almacenamiento separado de los elementos de la diagonal y los del resto de la matriz. Un primer vector AA contendrá los elementos diferentes de cero.

- Sus n primeros elementos corresponden a los valores que forman la diagonal, dejando generalmente la posición $n+1$ vacía
- A partir del $n+1$, se almacenan los elementos por fila excluyendo los que forman la diagonal.

Un segundo vector IA de enteros contendrá:

- Para los $n+1$ primeros valores indican los índices de la matriz AA donde comienza cada fila.
- Para los elementos a partir del $n+1$, indica el índice de la posición de columna de la matriz original.

Dada la matriz (1.8), obtenemos:

$$AA \boxed{1\ 5\ 7\ 9\ *\ 2\ 3\ 4\ 6\ 8}$$
$$IA \boxed{5\ 8\ 9\ 10\ 10\ 1\ 2\ 3\ 2\ 3}$$

Nota: para $IA(k) = IA(k+1)$ indica que el elemento $k+1$ es la columna ficticia.

1.2.5. Compressed Diagonal Storage (CDS)

Las matrices con estructura diagonal son matrices cuyos elementos diferentes de cero se encuentran a lo largo de un pequeño número de diagonales. Podemos almacenar estas diagonales en un array rectangular DIAG de tamaño $n \times nd$ (nd es el número de diagonales) y un vector de enteros ID de tamaño nd que contendrá los offsets de la posición de las diagonales respecto la diagonal principal, cumpliendo así,

$$DIAG(i,j) = a_{i,i+ID(j)}$$

Partiendo de nuevo de la matriz ejemplo (1.8), obtenemos los arrays:

$$DIAG = \begin{vmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \\ 8 & 9 & 0 & 0 & 0 \end{vmatrix} ID \boxed{-1\ 0\ 1\ 2\ 3}$$

1.2.6. Jagged Diagonal Storage (JDS)

Una forma simplificada de este método, llamado ITPACK, funcionaria de la siguiente manera:

$$COEF = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \\ 7 & 0 & 0 & 0 \\ 8 & 9 & 0 & 0 \end{vmatrix} JCOEF = \begin{vmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 2 & 3 & 3 & 3 \end{vmatrix}$$

Tal como se puede observar, necesitamos dos arrays de tamaño $n \times Nd$. En el primer array COEF, se eliminan todos los elementos igual a cero en la matriz y los restantes se desplazan hacia la izquierda ocupando los huecos vacíos; al mimso tiempo por la derecha se rellanan las

filas con ceros hasta que todas tengan la misma longitud, quedando una estructura similar al CDS. El segundo array, JCOEF, contiene los índices de la posición de columna que ocupa cada elemento. Notar que para los elementos auxiliares 0 usados para rellenar la matriz COEF se ha usado un valor arbitrario, en este caso el número de fila que ocupan aunque podría ser otro cualquiera.

1.3. Selección del formato de almacenamiento

A la vista de las características anteriores descritas, podríamos realizar una comparativa con tres de los formatos más representativos: COO, CRS y CDS.

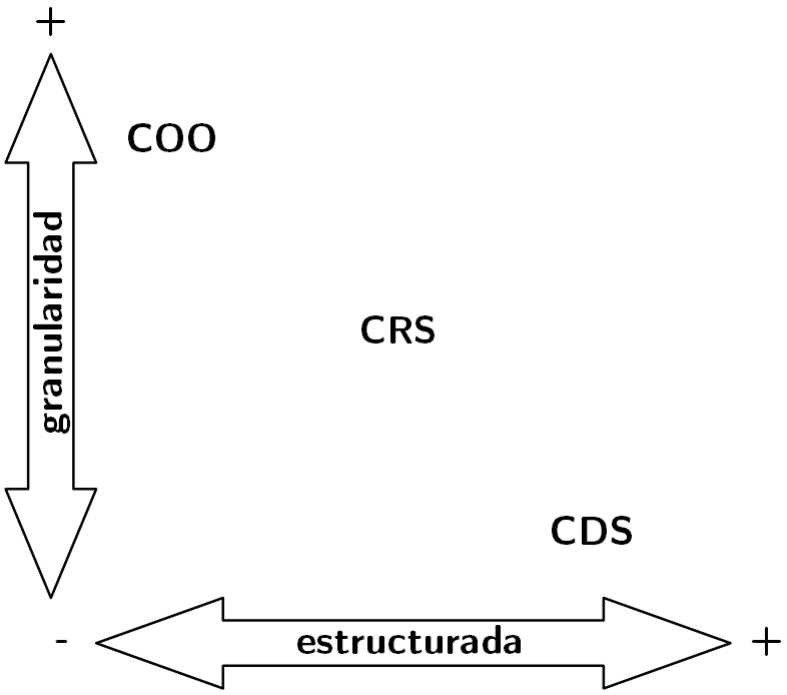


Figura 2.1: Comparativas almacenamiento matrices

Según podemos observar en la figura 2.1, de adoptar el formato COO nos permitiría una gran granularidad de paralelismo (a nivel de elemento), mientras que en el otro extremo encontramos el formato CDS con una granularidad mucho más grande (a nivel de fila). En cuanto a la estructura de la matriz sobre el cual se aplique el formato, COO almacena una matriz completamente desestructurada mientras que CDS exige que la matriz sea completamente estructurada. Como termino medio entre ambos formatos para estas dos características encontramos el CRS, que nos permitirá una granularidad variable (tal como veremos en el capítulo 5 con los dos kernels desarrollados) y un formato medianamente estructurado. Por ello, nuestra elección se decanta por el formato CRS.

2. APIs

2.1. Compute Unified Device Architecture (CUDA)

CUDA [8] es una arquitectura de cálculo paralelo de nVidia pensada para aprovechar la gran potencia de la GPU. su Software Development Kit (SDK) de ofrece una API de programación, incluyendo compilador y un conjunto de herramientas de desarrollo, y una arquitectura, todo bajo licencia propietaria.

2.2. Open Computing Language (OpenCL)

OpenCL [5] es un estándar abierto para la programación paralela con propósito general sobre CPUs, GPUs y otros procesadores. Su entorno de desarrollo consiste en un lenguaje, una API, librerías y un runtime-system para el desarrollo de software que incorporará entre otros el compilador necesario. OpenCL establece una API para coordinar la programación paralela entre procesadores heterogéneos, así como un lenguaje de programación dentro de un entorno especificado. Las características principales de OpenCL son:

- Modelo de paralelismo a nivel de datos y tareas.
- Lenguaje basado en ISO C99 con extensiones para el paralelismo.
- Adopción norma IEEE 754 de datos numéricos.
- Define un perfil de configuración para los diferentes dispositivos.
- Interoperación con OpenGL, OpenGL ES y otras APIs.

En el siguiente capítulo veremos con detalle el modelo de plataforma, ejecución, memoria y programación que presenta OpenCL así como la relación entre la arquitectura a alto nivel presentada por OpenCL y la microaquitectura presentada a bajo nivel por los procesadores RV770 de ATI sobre los que realizaremos las pruebas.

2.3. ATI Stream

ATI Stream technology [9] es un conjunto de tecnologías hardware y software que permite a las GPUs de ATI trabajar simultáneamente con la CPU para acelerar así aplicaciones de propósito general. El ATI Stream Software Development Kit (SDK) es una plataforma de desarrollo creada por AMD para permitir desarrollar rápida y fácilmente aplicaciones aceleradas gracias a ATI Stream technology. Su SDK ésta pensado para el desarrollo a alto nivel usando OpenCL.

3. Librerías

3.1. Basic Linear Algebra Subprograms (BLAS)

BLAS [10] es actualmente la más popular y base para muchas otras librerías que veremos a continuación. Se trata de un paquete software que proporciona las operaciones algebraicas básicas sobre vectores y matrices densas. Existen numerosas optimizaciones realizadas según su arquitectura y que además incorporan multithreading, algunas de ellas son: AMD Core Math Library (ACML), nVidia cuBLAS, Apple Velocity Engine , IBM ESSL , Intel MKL, ...

3.2. Linear Algebra Packaged (LAPACK)

Paquete de rutinas para la resolución de problemas de algebra lineal como pueden ser: valores propios, sistemas de ecuaciones lineales, descomposición de matrices, etc. Escrito en Fortran. Aparece en su primera versión en 1992, postulándose como sucesor de LINPACK, el cual estaba orientado a máquinas con memoria compartida. LAPACK [11] depende de las subrutinas BLAS para explotar eficientemente la arquitectura de las memorias caché actuales. La librería Magma de nVidia implementa sobre CUDA algunas de las funciones de Lapack.

3.3. Parallel Linear Algebra Package (PLAPACK)

PLAPACK es una infraestructura desarrollada por The University of Texas at Austin para facilitar la implementación a alto nivel de abstracción de algoritmos paralelos en el área del algebra lineal.

3.4. Scalable LAPACK (ScaLAPACK)

Scalable LAPACK [12], es una librería software para la computación paralela de álgebra lineal en computadores con memoria distribuida. Esta librería incluye un subconjunto de rutinas LAPACK rediseñadas para computadores paralelos MIMD (Multiple Instructions Multiple Data) con memoria distribuida. Está escrito al estilo Single-Program-Multiple-Data y usa el paso de mensajes explícito para la comunicación entre procesos. ScaLAPACK asume que las matrices están distribuidas de forma cíclica por bloques. Las bases de construcción de ScaLAPACK son la versión distribuida de BLAS, PBLAS y BLACS para las tareas de comunicación.

Capítulo 3

OpenCL

1. Introducción

OpenCL es un framework que nos permite programar para entornos heterogéneos. Por ello, el código desarrollado para éste es capaz de ser ejecutado sobre cualquier procesador como p.e. intel x86, RV770 de ATI/AMD o Cell/B.E de IBM.

2. Modelo de plataforma

Los dispositivos OpenCL se comunican con el host, en el cual estará corriendo la aplicación de control de ejecución. El mismo host puede ser también un dispositivo OpenCL. Los elementos mas importantes que forman el modelo son:

Kernels Llamamos kernel a una función escrita en un lenguaje que permite ser compilado para su ejecución en cualquier dispositivo que soporte OpenCL. Aunque los kernel son puestos en la cola de instrucciones para su ejecución por la aplicación host escrita en C, el kernel puede ser compilado por separado con el objetivo de optimizarlo para las características concretas del dispositivo sobre el que se ejecutará. Podemos escribir el kernel OpenCL en un fichero separado o incluirlo dentro del código de la aplicación host. OpenCL también permite compilar los kernels en tiempo de ejecución al lanzar la aplicación host o usar un binario precompilado.

Devices La especificación de OpenCL se refiere de esta forma a cualquier dispositivo con capacidad computacional de la máquina. Para cada device existe una o mas unidades de cómputo (Compute Unit (CU)) (comúnmente llamado núcleo o core). Esta unidad de cómputo es un hardware capaz de ejecutar e interpretar un conjunto de instrucciones. Un dispositivo como la CPU puede tener una o mas CU en cuyo caso se las suele denominar multicore CPU. Generalmente se establece la correspondencia entre número de CU y el número de posibles instrucciones independientes que un dispositivo puede ejecutar al mismo tiempo. Las CPUs actuales usualmente contienen entre dos y ocho CU, incrementando año tras año.

Hosts El programa que realiza las llamadas a las funciones OpenCL para crear el contexto de ejecución en el cual los kernels se ejecutaran es conocido como aplicación host y el dispositivo en el cual se ejecute dicha aplicación es conocido como el dispositivo host. Antes de que los kernels puedan ejecutarse la aplicación debe completar los pasos:

1. Determinar qué dispositivos hay disponibles.
2. Seleccionar los dispositivos apropiados para la aplicación.
3. Crear las colas de instrucciones para los dispositivos seleccionados.
4. Reservar los objetos en memoria (buffer object o image object) necesarios para la ejecución de los kernels.

3. Modelo de ejecución

La ejecución de un programa OpenCL involucra la ejecución simultanea de múltiples instancias de un kernel en uno o más dispositivos OpenCL controlados por el host. A cada instancia del kernel se la denomina **work-item**. Cada work-item ejecuta el mismo código pero sobre diferentes datos y se ejecuta sobre un solo core del multiprocesador o CU en la GPU. Al lanzar la ejecución de un kernel sobre un dispositivo, hemos de definir en número de work-items que usaremos para completar el proceso sobre los datos. La definición del tamaño se hace definiendo lo que llamamos **index-space**. OpenCL soporta un index-space de 1, 2 o 3 dimensiones.

OpenCL agrupa los work-items en work-groups. Podemos sincronizar los work-items dentro de cada work-group pero no work-items de diferentes work-groups o sincronizar entre work-groups. Cada work-item tienen un único identificador global (global ID), que establece su localización dentro del index-space.

El alcance del index-space global en cada dimensión lo denotamos como $S_{g,x}$ y $S_{g,y}$ y han de ser múltiples del tamaño del work-group que denotamos como $S_{w,x}$ y $S_{w,y}$. Dado el ID del work-group (x_w, y_w), el tamaño de los work-groups $(S_{w,x}, S_{w,y})$ y el ID local de un work-item $(x_{i,l}, y_{i,l})$, el ID global y el número de work-groups lo podemos calcular:

$$\begin{aligned}(x_{i,g}, y_{i,g}) &= (x_w S_{w,x} + x_{i,l}, y_w S_{w,y} + y_{i,l}) \\ (N_{g,x}, N_{g,y}) &= (\frac{S_{g,x}}{S_{w,x}}, \frac{S_{g,y}}{S_{w.y}})\end{aligned}$$

Ejemplo: nuestro kernel modifica los píxel de una imagen de 64x64 píxel. Podemos definir un index-space bidimensional de tamaño 64x64 estableciendo así un work-item para cada píxel de la imagen. OpenCL se encargará de forma transparente de asignar cada work-item sobre uno de los procesadores disponibles.

Ejemplo: El work item del index-space bidimensional del ejemplo anterior con global ID (13,52) nos indica que su posición en el eje X es 13 y 52 para el eje Y.

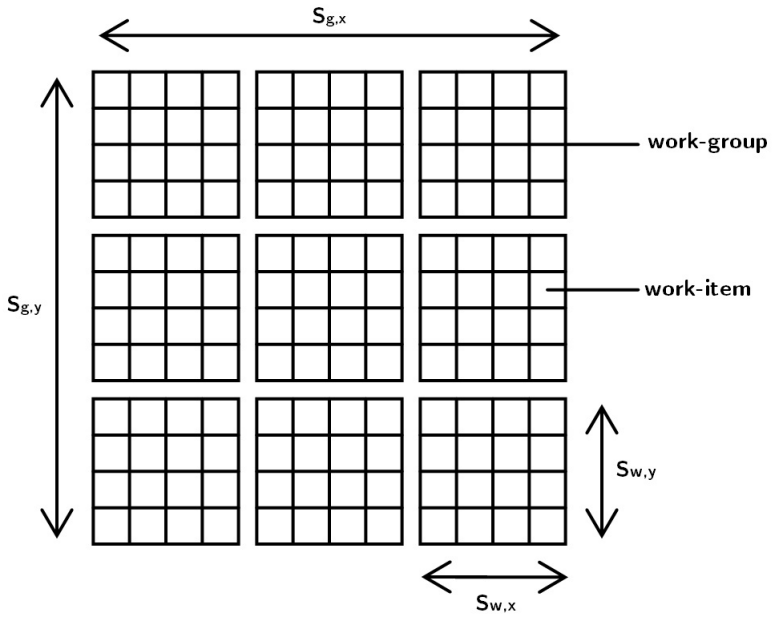


Figura 3.1: Ejemplo index-space 2D

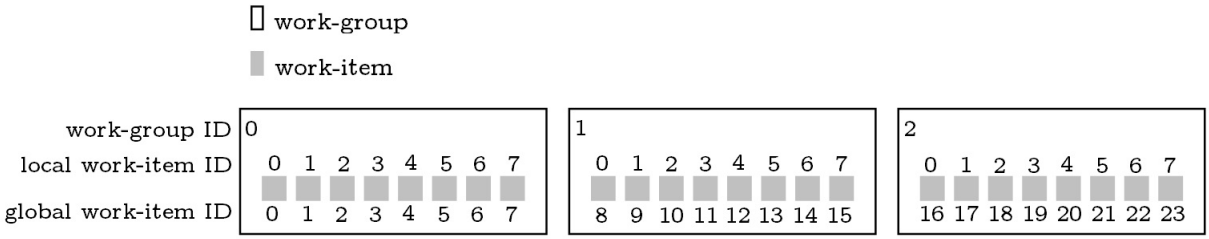


Figura 3.2: ID asociados a cada espacio

A su vez, cada uno de los work-group que agrupan los work-items también tiene un único ID similar al work-item global ID, en este caso estableciendo la posición del work-group en el index-space. El número de work-items en cada dimensión debe ser divisible entre los work-groups de esa dimensión.

> Ejemplo: sobre un ndex-space global de tamaño 64 por 256, cada uno de nuestros work-groups podrían contener 8 x 32 work-items, estableciendo así un array de work-groups de tamaño 8 x 8.
>
> - 8 work-groups en la dimensión X x 8 work-items por cada work-group = 64 work-items en X.
> - 8 work-groups en la dimensión Y x 32 work-items por cada work-group = 256 work-items en Y

En este ejemplo, el work-group con ID (1,6) se encuentraría en la posición 1 del eje X y la

posición 8 en el eje Y.

Con el objetivo de generalizar una aplicación OpenCL para correr sobre la mayor variedad de plataformas posibles generalmente puede no ser recomendable preestablecer un tamaño fijo de work-items y work-groups. Para ello se deberían usar la rutinas de OpenCL *clGetDeviceInfo* con el flag *CL_DEVICE_MAX_WORK_ITEM_SIZES* y *clGetKernelWorkgroupInfo* con el flag *CL_KERNEL_WORK_GROUP_SIZE* para determinar el tamaño máximo del work-group para un dispositivo y un kernel dado. Este número cambiara para cada dispositivo y kernel.

La aplicación que corre sobre el host establece el contexto en el cual el kernel se ejecuta, incluyendo la reserva de memoria de los distintos tipos, la transferencia de datos entre los objetos en memoria y la creación de las colas de comandos usadas para el control de la secuencia de estos. La sincronización para asegurar el orden de ejecución deseado es responsabilidad del programador, haciendo uso de los comandos que se incluyen para tal efecto el la API de OpenCL.

4. Modelo de memoria

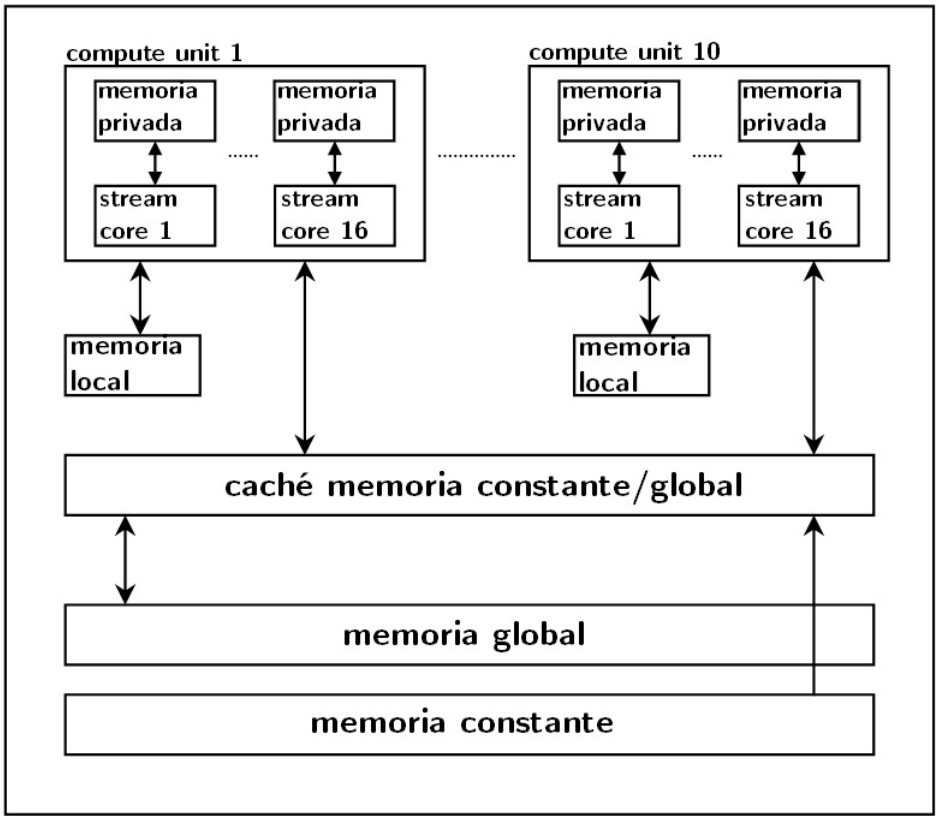


Figura 3.3: Modelo de memoria

OpenCL generaliza los diferentes tipos de memoria disponibles en:

- **Global memory** está disponible para todos los work-items de todos los work-groups. Es la memoria principal del dispositivo en el modelo de plataforma.

- **Constant memory** es una región de la memoria global reservada para solo lectura por los work-items y que se mantiene constante durante la ejecución del kernel. Únicamente puede ser escrita y leída por la aplicación del host.
- **Local memory** es accesible para lectura y escritura por un work-group específico y puede ser usado para mantener datos compartidos por todos los work-items de un work-group.
- **Private memory** es accesible únicamente para cada work-item.

5. Modelo de programación

El modelo de programación de OpenCL está basado en la existencia de un dispositivo host, soportado por una API y un número de dispositivos conectados a través de un bus, los cuales son programados usando el lenguaje OpenCL C. La API del host se divide en dos capas: la capa de plataforma y la de ejecución.

Los dispositivos son capaces de llevar a cabo la ejecución paralela a nivel de tareas o de datos.

Un kernel puede ser ejecutado como una función con un índice sobre un dominio multi-dimensional y cada uno de estos kernels, en tiempo de ejecución, son llamados work-items; el número total de índices es definido como el *global work-size*. A su vez éste puede ser dividido en subdominios llamados work-groups, y los work-items que lo forman pueden comunicarse mediante la memoria global o la memoria local compartida. Los work-items pueden también ser sincronizados mediante barreras o defensas de ejecución.

En la construcción de una aplicación OpenCL, en primer lugar se determina las plataformas presentes que pueden ser escogidas. Seguidamente, como segundo paso se construirá el contexto de ejecución, el cual tendrá asociados un número de dispositivos (como GPU o la CPU). Dentro de ese contexto, OpenCL garantiza una consistencia *relajada* entre los dispositivos, i.e. los objetos de memoria como buffers o imagenes son alocatados por contexto pero solo se garantiza que los cambios realizados sobre ellos por un dispositivo sean visibles por otro dispositibo en los puntos definidos de sincronización. Para llevar a cabo esta sincronización, OpenCL provee *eventos* con los cuales poder asegurar el orden de ejecución deseado.

Un gran número de operaciones se realizan con respecto al contexto dado; por otro lado, otro gran número de operaciones se realizan con respecto a un dispositivo concreto. Por ejemplo, la compilación y la ejecución del kernel se realizan para un dispositivo, y a su vez, para las operaciones como movimiento de datos de memoria local se usa el *command queue* como interfície específica para ese dispositivo.

La mayoría de los programas desarrollados con OpenCL siguen un mismo patrón:

1. Identificación plataforma
2. Selección de dispositivos que formarán el contexto
3. Reservas de memoria
4. Creación de los *command queue*.
5. Transferencia de datos y comandos.

Capítulo 4

Microarquitectura del HW

1. Conceptos previos

Aquí describiremos algunos conceptos fundamentales para la comprensión de capítulos posteriores. Hemos de tener presente que algunos de estos términos descritos son habitualmente designados con otro nombre, en especial, los términos referidos a componentes de la arquitectura donde cada fabricante da un nombre a sus unidades operacionales. Por ello, el lector no ha de extrañarse si estos términos son usados para designar otros elementos de manera diferente a como aquí se exponen en otros documentos bibliográficos. La terminología aquí usada se corresponde con la empleada por ATI/AMD en su arquitectura RV770, según la que se diseñó el chipset de la GPU sobre la que realizaremos los test, la HD4850. También veremos algunos conceptos de OpenCL que debido a su importancia y fuerte dependencia del hardware han de estar presentes en este capítulo.

SIMD (Single instruction, multiple data) La arquitectura SIMD se basa en explotar diferentes flujos de datos dentro de un mismo flujo de instrucciones para realizar operaciones que pueden ser paralelizables de manera natural. En CPU un claro ejemplo es el llamado Streaming SIMD Extensions (SSE), el cual requiere el uso de tipos de datos vectorizados (float4) para activar la generación de código empaquetado SSE y obtener un buen rendimiento del hardware SIMD. Por otro lado, el hardware VLIW en las GPU es mas flexible y permite incluso el uso eficiente de los Processing Elements (PE) (las unidades de procesado a más bajo nivel) sin el uso explícito de datos vectorizados.

VLIW (Very Long Instruction Words) La arquitectura VLIW implementa un nivel de paralelismo a nivel de instrucción. Cada instrucción VLIW indica el estado en el que se encuentra todas las unidades funcionales (i.e. todas las ALUs involucradas) y sus operaciones (en nuestro caso, la GPU que usaremos implementa una arquitectura de VLIW de rango 5, que se corresponde con los 5 PE, de este modo, la planificación de la ejecución se realizará entre el compilador y el programador, evitando los inconvenientes que implicaría la planificación en tiempo de ejecución en una GPU.

Compute Unit (CU) CU es la unidad de procesado a más alto nivel. Se corresponde con el número total de núcleos de procesado, p.e. en un Intel Core 2 tendremos 2 CU.

Stream Core Es la unidad responsable de las operaciones aritméticas como suma, resta, multiplicación, división, operaciones a nivel de bits y otras operaciones trascendentales. Es en estas unidades donde se ejecutan las instrucciones VLIW de un thread particular. Esta compuesto por cinco PE además de otras unidades como el *Branch Unit* (BU) o registros.

Processing Element (PE) Es la unidad fundamental del cálculo de las operaciones. Cada PE lleva a cabo una única instrucción de la instrucción VLIW. Existen dos tipos de PE según las instrucciones que pueden llevar a cabo, básicas como suma, producto, etc. o especiales como sin, cos, etc.

Wavefront La división *física* (depende del hardware, no es posible su modificación) de un work-group en grupos de work-items se denomina wavefront. El tamaño del wavefront se corresponde con el número de work-items que la GPU puede ejecutar de forma paralela sobre una misma Compute Unit (CU).

Con estos conceptos previos, a continuación se describirá de forma breve los aspectos más importantes de la arquitectura RV770 de ATI/AMD.

2. Microarquitectura ATI/AMD RV770

2.1. Historia del modelo

A mediados del año 2008 ATI/AMD lanza al mercado la serie de GPUs Radeon4xxx que incorporan el nuevo chipset RV770 mejorando la arquitectura de la anterior generación R600. Pasaría a ser su primera generación de chipsets para GPU que podrá correr software siguiendo especificaciones OpenCL publicadas en 2009 aunque no dará soporte completo para todas las características.

Esta arquitectura es técnicamente muy similar a la anterior Unified Shader Architecture empleada por primera vez en los chipsets R600. Igualmente el diseño se sigue basando en el concepto de VLIW.

2.2. Esquemas

En la figura 4.2 podemos ver un esquema básico de los elementos mas importantes que forman parte de la arquitectura RV770. En en nivel superior encontramos la GPU. En nuestro caso la HD4850 de ATI/AMD. LA GPU está a su vez compuesta por 10 Compute Units o SIMD cores, cada uno con 16 unidades stream core o ALUs. En total tendremos por lo tanto,

$$10\ \textit{SIMD core} * \frac{16\ \textit{stream Core}}{1\ \textit{SIMD core}} = 160\ \textit{stream core} \tag{2.1}$$

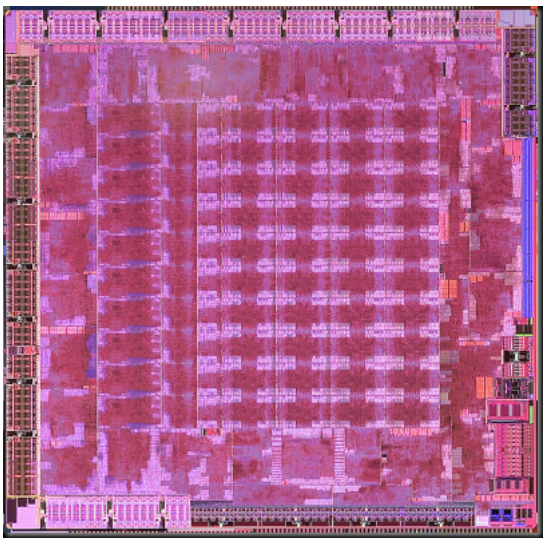


Figura 4.1: Chip RV770 de ATI/AMD

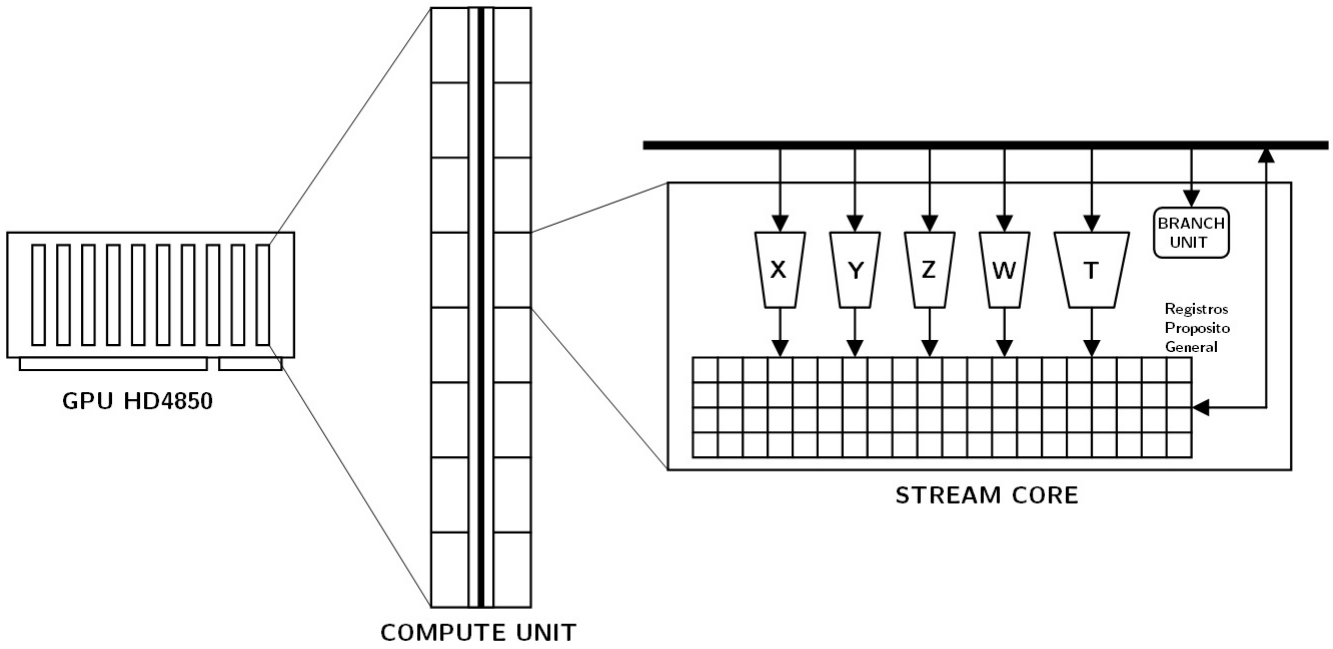


Figura 4.2: Diagrama básico de la aquitectura RV770

los cuales establecerán la "granularidad de ejecución de los thread" (tendremos 160 pipelines de instrucciones simultáneas) donde cada stream core del RV770 ejecuta un paquete de 5-VLIW instrucciones. En cada stream core se encuentran 5 processing elements o ALUs denominadas x, y, z, w y t. De todas estas, las 4 primeras (xyzw son simétricas y capaces de realizar una multiplicación-suma de un número de precisión simple en punto flotante por ciclo. La ALU restante (t), es una unidad denominada Special Function Unit (SFU) capaz de realizar funciones como sin, cos, etc. además de todas las operaciones realizadas por el resto de ALUs. La precisión doble en

punto flotante se consigue uniendo las ALUs xyzw de cada SP formando pares. Como resultado de esta unión obtenemos dos unidades dobles con capacidad de realizar una multiplicación o suma por ciclo.

A su vez, cada CU contiene 16384 registros de propósito general (GP registers) que se encuentran físicamente repartidos entre todos los *stream cores*. Cada uno de estos registros contiene 4x32-bit (4x números en punto flotante de precisión simple o 4x integer de 32-bit), lo que nos da un total de 256 kB para cada CU. Estos registros son compartidos entre todos los wavefronts activos que se ejecuten sobre un compute unit; al contrario de lo que ocurre en las CPU donde cada thread tiene asignado un número fijo de registros, en este caso cada kernel alocatará únicamente los registros que vaya a necesitar. El programador ha de minimizar el uso de estos registros siempre que sea posible ya que a mayor número de registros usados por work-item, menor será el número de wavefronts que estarán activos al mismo tiempo sobre una CU.

En cuanto a memoria global, en nuestro caso el chipset HD4850 montado sobre una placa del fabricante Club3d cuenta con 512MB GDDR3 de dos canales con un bus de 256bits.

3. ATI/AMD HD4850

3.1. Características generales HD4850

Teniendo presente las anteriores características presentadas podemos obtener una medida de la capacidad teórica de la GPU:

Flops/s Precision simple:

$$10\ \text{CU} * \frac{16\ \text{SIMD}}{1\ \text{CU}} * \frac{5\ \text{PE}}{1\ \text{SIMD}} * \frac{2\ \text{flops}}{1Hz} * \frac{625\ \text{MHz}}{1\text{s}} \quad = \quad 1000000\text{Mflop/s} = 1\ \text{TFlop/s}$$

Flops/s Precision doble:

$$\begin{aligned} 10\ \text{CU} * \frac{16\text{SIMD}}{1\text{CU}} * \frac{2\ \text{doble-PE}}{1\ \text{SIMD}} * \frac{1\ \text{flops}}{1\ \text{ciclo}} * 625\ \text{MHz/s} \quad &= \quad 200\ \text{GFlop/s doble precisión} \\ &+ \quad 200\ \text{GFlops/s precison simple unidad t} \end{aligned}$$

Bandwith (1GHz):

$$1\ \text{GHz/s} * 2\ \text{canales} * \frac{1\ \text{canal}}{256\ \text{bits}} = 64\ \text{GB/s}$$

3.2. HD4850 bajo OpenCL

Tal como se introdujo en la breve historia del modelo, el diseño de esta GPU comenzó en el 2005 y finalizó con su lanzamiento en junio del 2008, 4 meses antes de la finalización de los

detalles técnicos de la especificación de OpenCL v1.0, que fue presentada el 18 Noviembre del 2008. Debido a que el diseño de la arquitectura RV770 fue previa a OpenCL, algunos aspectos no pudieron ser soportados por lo cual las GPU basadas en esta arquitectura no son recomendadas si se busca obtener un gran rendimiento. De hecho ATI/AMD hizo público un comunicado anunciando que no se realizará ningún trabajo futuro con el fin de dar soporte completo para esta arquitectura y anteriores.

A continuación podemos ver los aspectos que OpenCL reconoce de la GPU HD4850 con el entorno software de desarrollo ofrecido por ATI/AMD *ati-stream-sdk-v2.01-lnx32* y el driver *ATI Catalyst 10.3*:

```
Device Type:                                    CL_DEVICE_TYPE_GPU
  Device ID:                                    4098
  Max compute units:                            10
  Max work items dimensions:                    3
    Max work items[0]:                          256
    Max work items[1]:                          256
    Max work items[2]:                          256
  Max work group size:                          256
  Preferred vector width char:                  16
  Preferred vector width short:                 8
  Preferred vector width int:                   4
  Preferred vector width long:                  2
  Preferred vector width float:                 4
  Preferred vector width double:                0
  Max clock frequency:                          650Mhz
  Address bits:                                 32
  Max memeory allocation:                       134217728
  Image support:                                No
  Max size of kernel argument:                  1024
  Alignment (bits) of base address:             4096
  Minimum alignment (bytes) for any datatype:   128
  Single precision floating point capability
    Denorms:                                    No
    Quiet NaNs:                                 Yes
    Round to nearest even:                      Yes
    Round to zero:                              No
    Round to +ve and infinity:                  No
    IEEE754-2008 fused multiply-add:            No
  Cache type:                                   None
  Cache line size:                              0
  Cache size:                                   0
  Global memory size:                           134217728
  Constant buffer size:                         65536
```

```
Max number of constant args:          8
Local memory type:                    Global
Local memory size:                    16384
Profiling timer resolution:           1
Device endianess:                     Little
Available:                            Yes
Compiler available:                   Yes
Execution capabilities:
  Execute OpenCL kernels:             Yes
  Execute native function:            No
Queue properties:
  Out-of-Order:                       No
  Profiling :                         Yes
Platform ID:                          0x7f41552114a8
Name:                                 ATI RV770
Vendor:                               Advanced Micro Devices, Inc.
Driver version:                       CAL 1.4.553
Profile:                              FULL_PROFILE
Version:                              OpenCL 1.0 ATI-Stream-v2.0.1
Extensions:                           cl_khr_icd
```

Las características mas importantes que hemos de destacar ya que han de ser consideradas profundamente a la hora de desarrollar el software son:

Max compute units Es el número de CU de la GPU que OpenCL podrá usar como máximo.

Max work items Es el número máximo de work-items que nos permite usar para cada dimensión si en el comando *clEnqueueNDRangeKernel* hacemos uso de las tres dimensiones. El número total de work-items por lo tanto será:

$$\text{Max work items}[0] \times \text{Max work items}[1] \times \text{Max work items}[2] = 256^3$$
$$= 16{,}777{,}216 \text{ work-items}$$

Max work group size Es el número máximo de work-items que puede tener un work-group. Recordemos que en nuestra la arquitectura RV770 este número es preferible que sea múltiple de 64 con el fin de obtener un mejor rendimiento, lo que nos deja los valores : 64, 128 y 256.

Image support Este flag indica si es posible el uso de la caché de texturas de la GPU. En el caso de ATI/AMD, las GPU de la serie HD58xx hacen uso de una caché de texturas de hasta 8192 $\times$ 8192 píxels, cada uno de los cuales representa un valor float4 (se corresponde con los canales RGBA), lo que nos permitiría el uso de un total de:

$$8192 \times 8192 \text{ píxels} \times 4\text{Bytes} = 256\text{MB}$$

que podríamos usar para almacenar alguno de los elementos en memoria caché.

Global memory size Es el tamaño del que se dispone en la memoria principal de la GPU. En este caso la HD4850 del fabricante Club3D tiene una memoria de 512MB, de los cuales OpenCL solo reconoce (y el programador solo podrá hacer uso) 128MB.

Local memory type El tipo de memoria local del que podrá hacer uso el software OpenCL puede ser *Local* o *Global*.

Local La memoria se corresponde con una caché on-chip del procesador. Para las arquitecturas que lo soportan suele ser del orden de 32KB y tiene un tiempo de acceso unas 10x mas rápido que la memoria global.

Global La memoria local es emulada sobre la memoria global de la GPU como en el caso de la RV770.

Cache type Indica el tipo de cache de memoria global. Puede ser tres tipos:

None No existe caché para la memoria global (RV770).

Read only Existe una caché de solo lectura para la memoria global.

Read/Write Existe caché de lectura y escritura para la memoria global.

Extensions En este parámetro se muestran las extensiones soportadas por el dispositivo, como p.e. la extensión *cl_khr_fp64* que ofrece el soporte de datos double en el kernel.

Tal como se puede ver esta GPU sobre OpenCLv1.0 presenta grandes deficiencias en cuanto a soporte por parte de OpenCL se refiere y que lastrarán el rendimiento general:

1. Memoria local emulada sobre la memoria global $\downarrow$
2. La memoria global no se cachea $\downarrow\downarrow$
3. Solo son detectados 128Mb de los 512MB disponibles $\downarrow$
4. Sin acceso a la caché de texturas $\downarrow\downarrow$

Capítulo 5

Código Kernels

1. Introducción

En este capítulo describiremos la implementación del código desarrollado. Como kernels que llevan a cabo las operaciones de multiplicación-suma de los elementos de la matriz y vector sobre la GPU, se han desarrollado dos variantes que diferirán principalmente en su granularidad de paralelismo y con ello el orden de acceso a los datos; las denominaremos como ***kernel 1*** y ***kernel 2***. Nuestro objetivo principal como clave para obtener el mejor rendimiento posible sobre la arquitectura ATI/AMD RV770 se basa en minimizar los accesos a memoria, evitar caminos divergentes de los diferentes work-items de un mismo work-group, evitar dependencias de datos (en especial entre work-groups) y aprovechar tipos de datos vectorial que ofrece OpenCL.

2. Flujo de datos para el spMV

En esta breve sección veremos el flujo de los datos así como los buffers y memoria utilizada a lo largo de toda la ejecución.

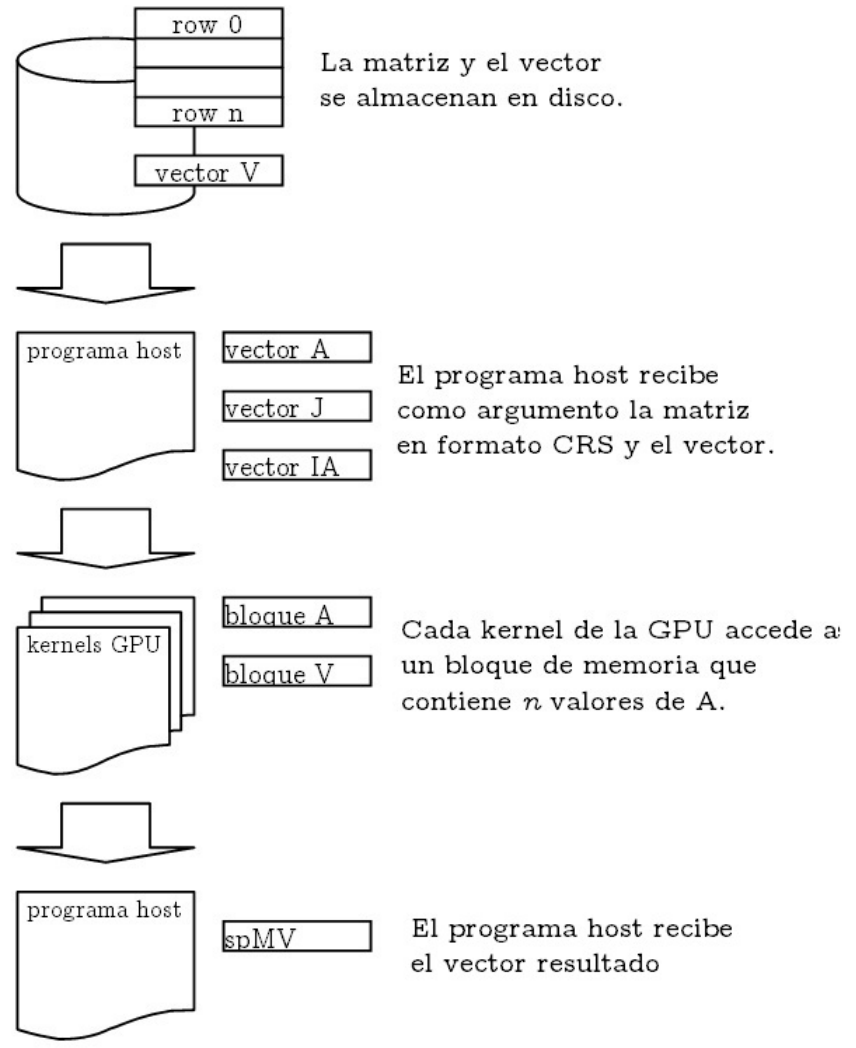


Figura 5.1: Flujo datos en spMV

Tal como se observa en la figura superior, tanto la matriz origen como el vector son almacenados en disco. Dado que el software desarrollado será liberado en forma de librería para ser cargado en tiempo de compilación del programa anfitrión, se han desarrollado funciones específicas para la lectura y almacenamiento en memoria de las matrices. Con el objetivo de generalizar para todo tipo de formatos de almacenamiento, estas funciones no serán integradas en la librería dejando al programador la tarea de preparación de los datos a pasar a la función de nuestra librería y usando las desarrolladas en el presente proyecto bajo criterio de éste.

Los formatos para los que se ha desarrollado una rutina para su lectura o ya existe y es proporcionada por MatrixMarket son los siguientes:

- dpt. de Enginyeria Electrònica:

```
18847 18847  203926
row 0: (0, 10.5446) (8, -0.00891528) ... (18360, -0.00891528)
                                    .
                                    .
                                    .
row 18847: (18831, 0.0256314) (18367, 0.0256314) . . . (18847, -50.4463)
```

- MatrixMarket Coodinate format

```
18847 18847  203926
0 0 10.5446
1 0 0.00891528
5 0 -0.00891528
        .
        .
        .
18844 18846 0.0256314
18844 18847 0.0256314
18847 18847 -50.4463
```

- y Harwell-Boeing

Una vez leídos y almacenados en RAM desde el programa anfitrión, estas matrices serán pasadas a nuestra función como datos de entrada. Al trabajar con CRS, necesitaremos los siguientes argumentos:

Datos entrada		
Nombre	tipo C99	Descripción
krnl	int [1,2]	Indica cual de los dos kernels desarrollados se desea emplear
A	*float/*double	vector de valores no nulos de la matriz de origen
J	integer	indices de columna para los valores del vector A
IA	integer	indices de inicio cada fila sobre los vectores A y J
V	*float/*double	valores del vector a multiplicar con la matriz origen
anchura	integer	anchura de la matriz de origen
altura	integer	altura de la matriz de origen
numeronz	integer	número de elementos no nulos en la matriz

Tabla 5.1: Argumentos función

En caso de tratar con valores complejos, con el fin evitar definir estructuras o tipos nuevos que provoquen un manejo complejo de la memoria en los kernels, se ha optado por tratar cada valor complejo como dos elementos consecutivos del vector de entrada o salida.

Para poder ejecutar el kernel todos los datos a procesar deben de estar en la memoria de la GPU, sin embargo, el kernel no tiene capacidad para acceder a la memoria fuera de la GPU y por ello esta acción ha de ser llevada a cabo desde el lado del host. Para hacer esto, un objeto de memoria ha de ser creado para permitir el acceso del host a la memoria de la GPU y copiar los datos allí. Para crear el objeto de memoria usamos la función específica del API llamada *clCreateBuffer* y que usamos en nuestro código tal como sigue:

```
inBufAA = clCreateBuffer(context, CL_MEM_READ_ONLY,
  sizeof(float)* nz, 0, &ret);
inBufJC = clCreateBuffer(context, CL_MEM_READ_ONLY,
  sizeof(int)* nz , 0, &ret);
inBufIA = clCreateBuffer(context, CL_MEM_READ_ONLY,
  sizeof(int)* (height+1) , 0, &ret);
inBufV  = clCreateBuffer(context, CL_MEM_READ_ONLY,
  sizeof(float)* width, 0, &ret);
outBufY = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  sizeof(float) * width, 0, &ret);
```

Esta memoria alocatada podrá ser accedida desde el host mediante el puntero al objeto de memoria que retorna. Como primer argumento indicamos el contexto del cual es parte el objeto en memoria. Como segundo argumento especificamos el flag indicando como será usada la memoria por parte del kernel; en nuestro caso de lectura para los datos que forman parte de la matriz y el vector, y de escritura para los datos de salida. El tercer argumento especifica el número de bytes a alocatar.

Para copiar los datos desde el lado del host hacia la memoria de la GPU, usamos la función $clEnqueueWriteBuffer()$:

```
ret = clEnqueueWriteBuffer(command_queue, inBufAA, CL_TRUE, 0,
        sizeof(float) * nz, AA, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, inBufJC, CL_TRUE, 0,
        sizeof(int) * nz, JC, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, inBufIA, CL_TRUE, 0,
        sizeof(int) * (height+1), IA, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, inBufV, CL_TRUE, 0,
        sizeof(float) * (width) , V, 0, NULL, NULL);
```

El termino “Enqueue” que observados en la función indica que la instrucción de copia de los datos es añadida en la cola de comandos antes de poder ser procesada. El segundo argumento es el puntero a la memoria de la GPU sobre la que se copiaran los datos desde el host, siendo el quinto argumento el tamaño de los datos en bytes y el sexto el puntero a los datos del host. El tercer argumento especifica si el comando es síncrono o asíncrono. Al pasarle “CL_TRUE” indicamos que sea ejecutada de forma síncrona, lo que hace que el host no ejecute el siguiente comando de la cola hasta que la copia de datos finalice. Si en su lugar hubiésemos pasado “CL_FALSE”, la copia se realiza asíncrona, encolando la instrucción de copia e inmediatamente ejecutando la siguiente instrucción del lado del host.

Una vez copiados los datos en la memoria global de la GPU, cada kernel accederá al bloque de memoria necesario. En caso de que OpenCL disponga de soporte para memoria local en la GPU, estos serán copiados en la memoria local compartida para los kernel que compongan un workgroup.

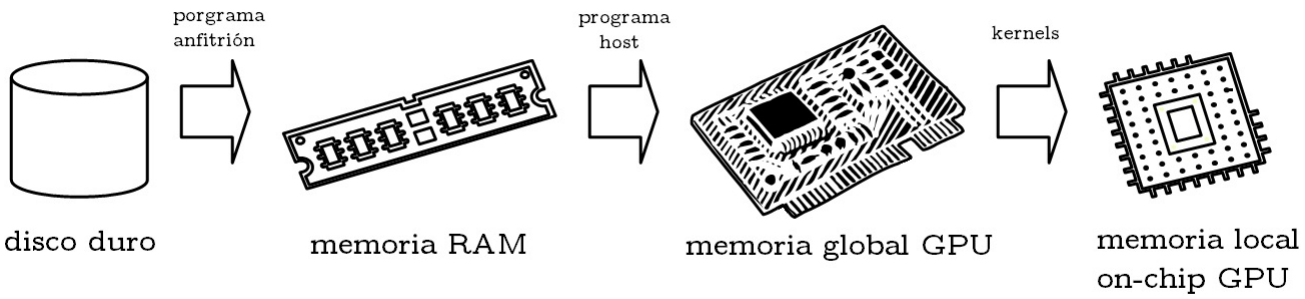


Figura 5.2: Niveles de memoria básicos

Una vez finalizados todos los kernels, el programa host leerá la memoria de la GPU y guardará en RAM los datos resultantes, retornando como salida un vector de tamaño $width$ ($2*width$ en caso de valores complejos).

```
ret = clEnqueueReadBuffer(command_queue, outBufY, CL_TRUE, 0,
  sizeof(float) * width, output, 0, NULL, &events[1]);
```

3. Ejecución de los kernels

Tal como se ha visto en los capítulos previos, una vez el programa host obtiene acceso a los datos y ha creado los buffers necesarios en la memoria de la GPU, entran en acción los kernels ejecutados sobre la GPU, que son los que realmente realizarán el producto matriz-vector.

Para ello, desde el programa host se hace uso de una función de la API de OpenCL llamada *clEnqueueNDRangeKernel()*.

```
size_t global_work_size[] = {width,0,0};
size_t local_work_size[] = {BLOCKSIZE,0,0};

ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
  global_work_size, local_work_size, 0, NULL, &events[0]);
```

En *global_work_size[]* indicamos el número total de work-items (kernel en ejecución) que tendremos, mientras que en *local_work_size[]* indicamos el tamaño de cada work-group. El primero siempre ha de ser múltiple del segundo.

Con la función *clEnqueueNDRangeKernel()* indicamos la intención de ejecutar un mismo kernel a lo largo de los múltiples procesadores de la GPU. El quinto y sexto argumento argumento determinan el tamaño total de work-items. En nuestro caso, tal como veremos en la sección siguiente, dependiendo del kernel que ejecutemos, estos números variarán. Los pasos que sigue la instrucción los podríamos resumir en:

1. Creación work-items en el host.
2. Procesar los datos correspondientes al ID global en el kernel.

Una vez entran en ejecución los kernels en la GPU, el programa host queda en espera de que finalice el evento $clEnqueueNDRangeKernel()$ para poder leer el buffer de memoria de la GPU donde almacenamos el resultado. Internamente, el orden de ejecución, agrupación, etc. de los kernels dependerá de la arquitectura de la GPU; así pues, en este caso la describiremos para la hd4850 con procesador rv770. Características a tener en cuenta:

1. Los work-items se agruparán en work-groups
2. Cada work-group solo se podrá ejecutar sobre una única compute unit (contamos con 10 en total)
3. Los work-groups son divididos en wavefronts (el mejor rendimiento se obtiene cuando el tamaño de los work-groups es múltiple del tamaño del wavefront).
4. Los wavefronts ejecutan N work-items en paralelo, donde N es específico para cada chip (en nuestro caso 64)
5. Una instrucción es ejecutada a lo largo de todos los work-items que pertenecen al wavefront en paralelo.

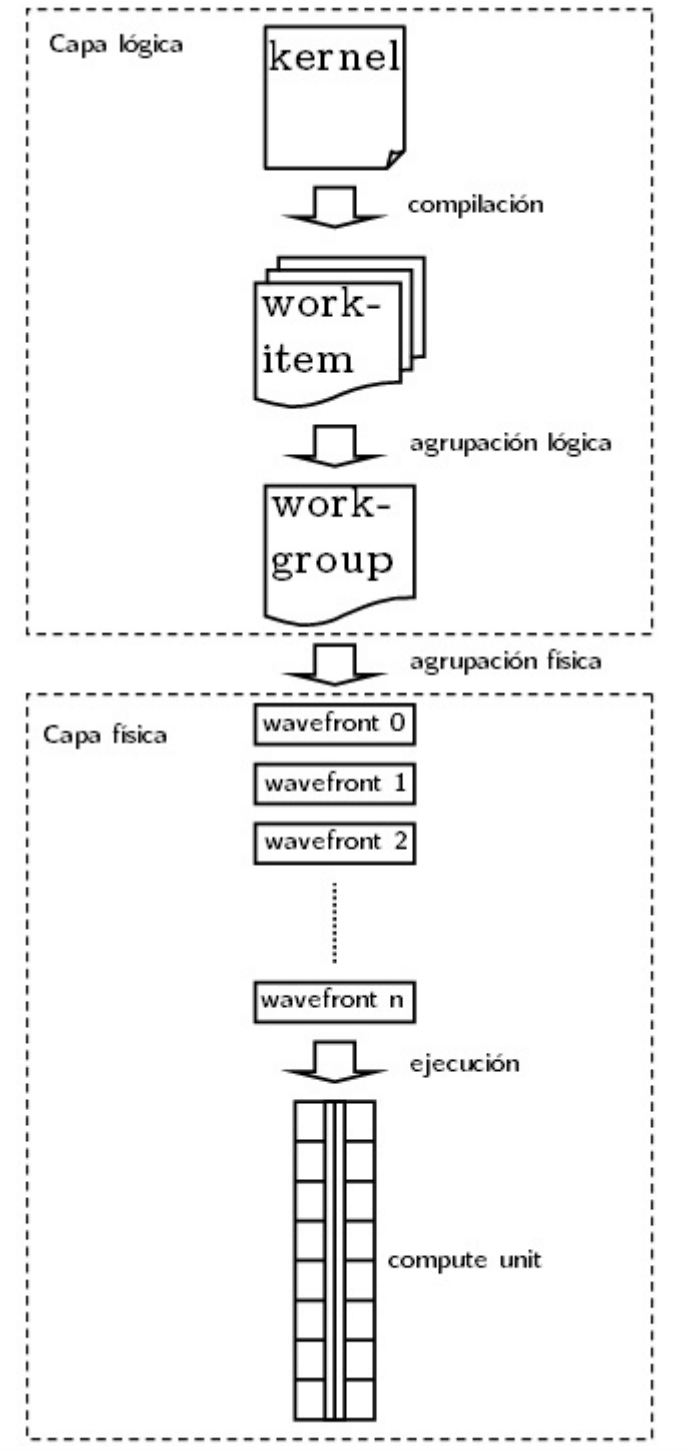


Figura 5.3: Capas del modelo de ejecución

4. Código de los kernels

En esta sección veremos las dos variantes del kernel desarrollado (obviaremos el uso de memoria local) para datos simples y complejos.

4.1. Kernel 1

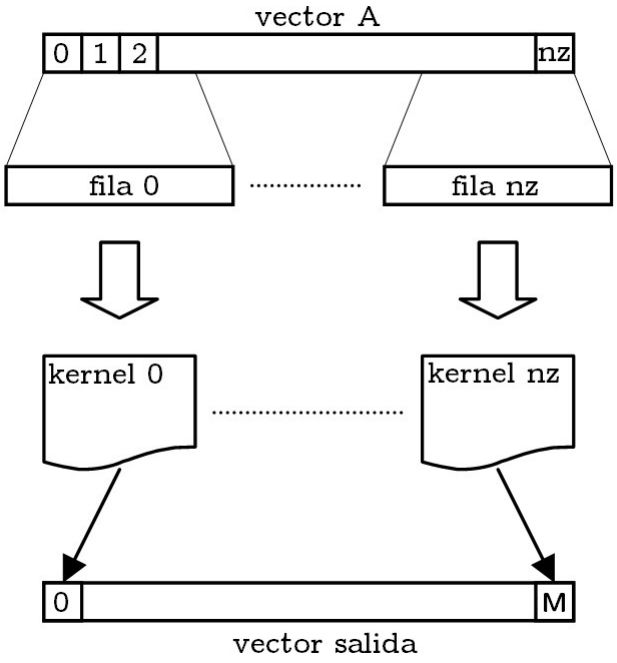


Figura 5.4: Flujo del kernel 1

Dado que el producto escalar resultado de cada fila por el vector se puede calcular independientemente, es facil y sencillo realizar una primera aproximación para paralelizarlo tal como se ha hecho en el kernel 1. Como vemos en la figura anterior, para este primer kernel asignamos un único work-item por fila sea cual sea el tamaño de cada fila; en total tendremos tantos work-items como altura tenga la matrix origen. De está forma conseguimos que todos los work-items sean ejecutados (no quedará ninguno *ocioso*). Lamentablemente, a su vez, este método presenta varios inconvenientes. El mayor de los problemas es la manera en que los work-items de un work-group acceden a los vectores de índices y valores del formato CRS. Mientras que los valores e índices para cada fila son almacenados consecutivamente, cada work-item accede a ellos de forma secuencial.

Ejemplo: matriz 4x4. 4 work-items.

$$AA \boxed{1\,2\,3\,4\,5\,6\,7\,8\,9}$$
$$JC \boxed{0\,1\,2\,3\,1\,2\,2\,2\,3}$$
$$IA \boxed{0\,4\,6\,7\,9}$$

donde:

step 1:	k0	-	-	-	k1	-	k2	k3	-
step 2:	x	k0	-	-	x	k1	x	x	k3
step 3:	x	x	k0	-	x	x	x	x	x
step 4:	x	x	x	k0	x	x	x	x	x

Tabla 5.2: Tabla de secuencia de acceso a los datos del kernel 1

- *ki* representa un dato que está siendo leído por el work-item *i*,
- - representa los valores pendientes de leer,
- y *x* representan los valores ya leídos por algún work-item.

En un primer paso los kernels acceden simultáneamente a los valores de las posiciones 0, 4, 6 y 7. En este caso los bancos de datos se alinean a 4Bytes de manera que cuando el work-item 1 solicite el valor 0, se traerá a memoria los valores de las posiciones 0, 1, 2 y 3 y sólo será usado el primer elemento.

Por otro lado, tal como vemos en en código abajo mostrado, cada work-item tendrá un bucle con el cual recorre todos los elementos de su fila y esto puede lastrar el rendimiento en caso de que el número de elementos diferentes de cero de cada fila de cada work-item que formen parte de un mismo work-group y sean divididos dentro de un mismo wavefront sea diferente. En este caso, hemos dejado que sea el compilador sea quien decida el tamaño de cada work-group ya que no será un elemento diferenciador desde el punto de vista del programador.

```
__kernel void spMVf_kernel(__global read_only float *AA,
                           __global read_only int *JC,
                           __global read_only int *IA,
                           __global read_only float *V,
                            int read_only width,
                           __global write_only float *Y)
{
        uint kid = get_global_id(0);

        int rowstart = IA[kid];
        int rowend = IA[kid+1];

        float acumulador=0;
        float4 auxV;
        float4 auxAA;

        int4 k;

        for(int i=rowstart; i<rowend;i+=4)
```

```
        {
                auxAA.x=AA[i];
                auxAA.y=AA[i+1];
                auxAA.z=AA[i+2];
                auxAA.w=AA[i+3];
                k = ((int4)(i,i+1,i+2,i+3) < (rowend));

                auxV.x = k.x ? V[JC[i]] : 0;
                auxV.y = k.y ? V[JC[i+1]] : 0;
                auxV.z = k.z ? V[JC[i+2]] : 0;
                auxV.w = k.w ? V[JC[i+3]] : 0;

                acumulador+=dot(auxAA,auxV);
        }
        Y[kid]=acumulador;
}
```

Listing 5.1: Código del kernel 1 para datos reales

En esta versión del kernel para datos de precisión simple tipo float, en la primera línea obtenemos el ID global del work-item mediante el comando *get_global_id(0)* que identificará al mismo tiempo la fila sobre la que operará (tenemos un work-item por fila). En las dos posteriores líneas, obtenemos del vector IA los índices que delimitarán los valores de la fila sobre la que operaremos en los vectores AA y JC. A continuación podemos ver el bucle principal, en el cual para optimizar el rendimiento calculamos productos de 4 en 4 mediante el producto escalar que ofrece la función *dot()* sobre dos vectores de tamaño 4 floats y lo guardamos en un acumulador. Una vez completado el bucle, guardamos el valor obtenido en el vector de salida.

Para el caso de realizar el cálculo para valores complejos se ha tenido en cuenta la representación matricial del producto de números complejos:

$$(A * X) \equiv (A_r + A_i) * (X_r + X_i) \equiv (Y_r + Y_i) \tag{4.1}$$

$$\begin{pmatrix} A_r & -A_i \\ A_i & A_r \end{pmatrix} * \begin{pmatrix} X_r \\ X_i \end{pmatrix} = \begin{pmatrix} Y_r \\ Y_i \end{pmatrix} \tag{4.2}$$

De esta forma, después de construir las dos matrices en memoria privada como *auxA* y *auxV* usamos la función matemática *dot()* para obtener el producto escalar que formará cada uno de los componentes del número complejo resultante:

```
__kernel void spMVfc_kernel1(__global read_only float2 *AA,
                             __global read_only int *JC,
```

```
                        __global read_only int *IA,
                        __global read_only float2 *V,
                         int read_only width,
                        __global write_only float2 *Y)
{
        uint kid = get_global_id(0);
        int rowstart = IA[kid];
        int rowend = IA[kid+1];

        float4 auxA;
        int col;
        float acumr=0,acumi=0;

        for(int i=rowstart; i<rowend; i+=1)
        {
                auxA.x=AA[i].x;
                auxA.y=-(AA[i].y);
                auxA.z=AA[i].y;
                auxA.w=AA[i].x;
                col=JC[i];
                acumr+= dot(auxA.lo,V[col]);
                acumi+= dot(auxA.hi,V[col]);
        }

        Y[kid]= (float2)(acumr,acumi);
}
```

Listing 5.2: Código del kernel 1 para datos complejos

En este segundo kernel, el primer elemento diferenciador con el anterior kernel que podemos observar es la definición de los argumentos de entrada. Los vectores que contienes valores float complejos (AA, V y Y) se acceden mediante vectores de tamaño 2 floats aprovechando que los valores de la tupla (num_real, num_img) se guardan consecutivamente. Dentro del bucle creamos la matriz y el vector y realizamos los productos escalares necesarios, obteniendo así los valores reales e imaginario de la tupla resultante que guardaremos en el vector y aL finalizar el bucle.

4.2. Kernel 2

Para solucionar el problema que presentaba el kernel 1, se ha diseñado este segundo kernel. Tal como podemos observar en la figura anterior, en este caso a cada fila de la matriz original se le asignarán 64 work-items. Estos 64 work-items se repartirán toda la carga de trabajo de cada fila. En este caso, los work-items de un work-group acceden a los vectores de índices y valores de

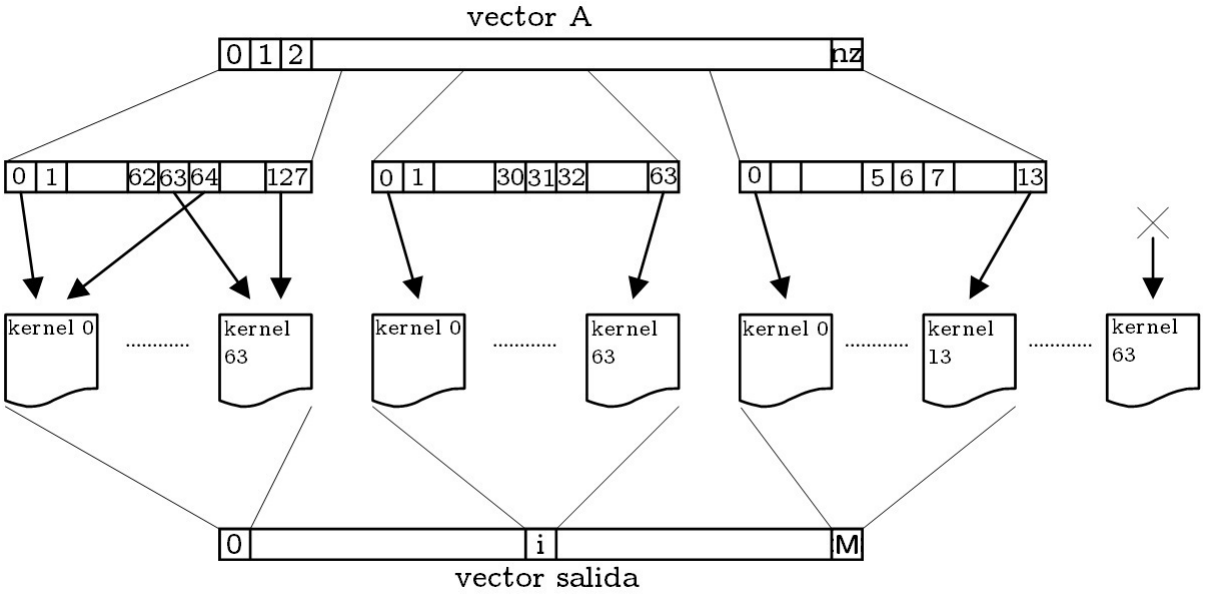


Figura 5.5: Flujo del kernel 2

forma contigua y simultáneamente, mejorando así el acceso que presentaba el kernel anterior. En el caso de que el número de elementos no cero de la fila sea menor o igual a 64, solo se realizará una iteración en el bucle del kernel y por lo tanto solo habrá un acceso a memoria para cada work-item. En el caso de que una fila contenga mas de 64 valores no cero, notar que aunque el bucle sea de varias iteraciones, siempre serán las mismas para todos los work-items, con el único inconveniente de que puedan quedar algunos de ellos en estado ocioso en el caso de que el número de elementos no cero no sea múltiple de 64.

```
__kernel void spMVf_kernel2(__global read_only float *data,
                            __global read_only int *indices,
                            __global read_only int *ptr,
                            __global read_only float *V,
                            int read_only width,
                            __global write_only float *Y)
{
        uint kid = get_global_id(0);
        uint blockid = get_group_id(0);
        uint localkid = get_local_id(0);
        uint row = blockid;

        if(row < width){

                int rowstart = ptr[row];
                int rowend = ptr[row+1];
                float val=0;
                float sum=0;
```

```
                __local float sumvals[64];

                for(int i=rowstart + localkid; i<rowend; i+=64)
                        val+=data[i]*V[indices[i]];

                sumvals[localkid]=val;

                barrier(CLK_LOCAL_MEM_FENCE);
                if(localkid == 0){
                        for(int i=0; i<64; i++)
                                sum+=sumvals[i];
                        Y[row]=sum;
                }

        }
}
```

[caption=Código kernel 2 para datos reales]

En este kernel, en primer lugar, con el fin de identificar la fila sobre la que operaremos y dentro de ésta cual de sus elementos, usamos los comandos *get_group_id(0)* y *get_local_id(0)*. Con el primero obtenemos ID del work-group al que pertenece el work-item y que identificará la fila sobre la que operar (tenemos tantos work-groups como filas), mientras que con el segundo obtendremos los valores dentro de la fila con los que operaremos (estarán en las posiciones get_local_id(0) + 64x). Una vez obtenidos los índices que delimitan la fila, crearemos un vector en memoria local donde se almacenarán los valores acumulados que obtenga cada work-item del grupo una vez finalize el bucle en el cual iremos realizando el producto de todos los elementos que le pertenezcan al wok-item. Antes de almacenar el valor resultante del work-item mediante el uso del comando *barrier(CLK_LOCAL_MEM_FENCE)* realizamos una síncronización que obligará a que todos los work-items hayan escrito en memoria local sus valores acumulados antes de proseguir con el código. Esto se realiza para asegurar la consistencia de los datos leídos por el work-item 1 de cada work-group, que será el encargado de calcular el resultado de la fila a partir de los valores acumulados por cada work-item guardados en el vector de memoria local y guardarlo sobre el vector de salida *Y*.

De forma análoga al anterior kernel, existe su versión para números complejos:

```
__kernel void spMVfc_kernel2(__global read_only float2 *data,
                        __global read_only int *indices,
                        __global read_only int *ptr,
                        __global read_only float2 *V,
                        int read_only width,
                        __global write_only float2 *Y)
{
        uint kid = get_global_id(0);
```

```
        uint blockid = get_group_id(0);
        uint localkid = get_local_id(0);
        uint row = blockid;

        if(row < width){

                uint rowstart = ptr[row];
                uint rowend = ptr[row+1];
                float4 auxA;
                float2 val=(float2)(0.f,0.f);
                float2 sum=(float2)(0.f,0.f);
                int col;
                __local float2 sumvals[64];

                for(int i=rowstart + localkid; i<rowend; i+=64)
                {
                        auxA.x=data[i].x;
                        auxA.y=-(data[i].y);
                        auxA.z=data[i].y;
                        auxA.w=data[i].x;
                        col=indices[i];
                        val.x+= dot(auxA.lo,V[col]);
                        val.y+= dot(auxA.hi,V[col]);
                }
                sumvals[localkid].x=val.x;
                sumvals[localkid].y=val.y;

                barrier(CLK_LOCAL_MEM_FENCE);
                if(localkid == 0){
                        for(int i=0; i<64; i++)
                        {
                                sum.x+=sumvals[i].x;
                                sum.y+=sumvals[i].y;
                        }
                        Y[row]=(float2)(sum.x,sum.y);
                }
        }
}
\n";
```

[caption=Código del kernel 2 para datos complejos]

4.3. Conclusiones

El método de almacenamiento CRS nos permite un número variable de elementos no cero para cada fila. Aunque esto es una ventaja para la representación eficiente del mayor número de tipos de matrices dispersas, esta característica introduce la posibilidad de ejecuciones divergentes en los work-items:

1. En el kernel 1: cuando se aplica a una matriz con un número de elementos no cero muy variable entre filas, muchos work-items que formen un wavefront permanecerán ociosos mientras el work-item con la fila de mayor tamaño siga iterando sobre ésta.

2. En el kernel 2: La eficiente ejecución del kernel exige que las filas de la matriz contengan como mínimo 64 elementos no cero y, en caso de ser mayor, múltiple de 64.

Como resultado, podemos determinar que el rendimiento de los dos kernels desarrollados para el formato de almacenamiento CRS son altamente sensibles al tamaño de las filas de la matriz.

5. Optimización

OpenCL nos proporciona una API estándar que lleva a cabo tareas como operaciones vectorizadas SIMD, paralelismo a nivel de datos. paralelismo a nivel de tareas y transferencias de memoria. Sin embargo, la capa de abstracción es relativamente cercana al hardware, minimizando el sobrecosto de usar OpenCL. Gracias a este nivel de abstracción es posible ajustar el código para obtener el máximo rendimiento del procesador escogido usando OpenCL. Además, permite el uso de la API específica del procesador para el caso concreto de que una función no sea soportada por OpenCL. Esto último iría en contra del propósito de OpenCL ya que el código generado dejaría de ser portable, sin embargo, OpenCL prioriza maximizar el rendimiento del dispositivo por encima incluso de la portabilidad. El rendimiento de un código puede variar ampliamente dependiendo del dispositivo sobre el cual se ejecute el kernel, en especial si ese dispositivo existen unidades SIMD. En resumen, programar sobre OpenCL no implica que ese código se ejecute mas rápido sobre un dispositivo, por ello, el hardware ha de ser escogido con precaución con el objetivo de maximizar el rendimiento.

5.1. Tratamiento de saltos

El tratamiento de saltos en una GPU en comparación con una CPU es una operación mucho más crítica que obliga al desarrollador a enfocar un gran esfuerzo en evitar divergencias de caminos en los kernels. Mientras que en los algoritmos para la CPU es habitual incluir código dentro de condicionales para evitar su ejecución si no es necesario evitando así la sobrecarga de trabajo al procesador, en la GPU es común que sea más costoso en términos de rendimiento evaluar el condicional y sus caminos divergentes a ejecutar ese código.

Tal como vimos en el capítulo de la arquitectura del procesador RV770, dentro de cada Stream Core encontramos además de los 5 Processing Elements, una unidad especial denominada Branch

Execution Unit que es la que se ocupa de evaluar y detectar todos los posibles caminos de los diferentes work-items que forman el wavefront. Únicamente se ejecutarán simultáneamente los work-items que ejecuten un mismo camino, es decir, para un if cualquiera, primero se ejecutaran todos los work-items del wavefront que cumplan la condición y posteriormente todos los que no la cumplan; el número de threads que se ejecutan para cada rama de ejecución se denomina branch granularity.

De esta forma, siempre que exista como mínimo un work-item dentro del wavefront que difiera del resto quedaran stream cores en estado ocioso reduciendo el rendimiento general del kernel. En el caso de ATI, la branch granularity es igual al wavefront, 64. En nuestro código, debido a que hemos aplicado la técnica del loop unrolling, accedemos a los elementos del vector AA de cuatro en cuatro, pudiendo pertenecer 1, 2 o 3 de estos elementos a una fila diferente de la cual estamos realizando el producto escalar. Para evitar esto, una manera inmediata sería evaluar si estos elementos pertenecen a nuestra fila mediante condicionales:

```
for(int i=rowstart; i<rowend;i+=4)
{
        auxAA.x=AA[i];
        auxAA.y=AA[i+1];
        auxAA.z=AA[i+2];
        auxAA.w=AA[i+3];

        if(i+1>rowend) auxV.y=0;
        if(i+2>rowend) auxV.y=0;
        if(i+3>rowend) auxV.y=0;

        acumulador+=dot(auxAA,auxV);
}
```

[caption=Ejemplo de código con ramas divergentes]

de esta forma, anulamos los elementos que no pertenezcan a la fila multiplicándolos por 0. Como vemos, necesitamos tres if para realizar esta comprobación. Si el número de elementos diferentes de cero de una fila no es múltiple de cuatro, siempre obtendremos un wavefront la ejecución del cual deba ser dividida en dos pasos. Claramente, cuanto menor número de elementos no cero en una fila, mayor será el impacto sobre el rendimiento total del kernel. Para evitar este inconveniente, sustituimos los condicionales por una evaluación que será identica para todos los threads pertenecientes al wavefront:

```
        kk = ((int4)(i,i+1,i+2,i+3) < (rowend));

        auxV.x = kk.x ? V[JC[i]] : 0;
        auxV.y = kk.y ? V[JC[i+1]] : 0;
        auxV.z = kk.z ? V[JC[i+2]] : 0;
        auxV.w = kk.w ? V[JC[i+3]] : 0;
```

[caption=Código de código sin ramas divergentes]

tal como vemos, en la primera línea generamos un vector de 4 integer que nos indicaran con 0 si el elemento no pertenece a la fila y con 1 si pertenece. En las cuatro líneas posteriores, cargamos bien el valor del vector V o bien 0 en función del vector anterior. De esta forma, evitamos cualquier camino divergente sustituyendo tres *if* con caminos divergentes que implican ejecución en dos pasos por 5 operaciones que se pueden ejecutar simultáneamente.

5.2. Loop Unrolling

El uso eficiente del hardware GPU requiere que el kernel contenga suficiente paralelismo como para ocupar los cinco processing elements. En el caso de las instrucciones con cadenas de dependencias entre ellas, éstas son planificadas en diferentes instrucciones dejando así varios Processing elements en estado ocioso.

```
for(int i=rowstart; i<rowend;i++)
{
        acum+= AA[i] * V[JC[i]];
}
```

[caption=Ejemplo de código sin Loop Unrolling]

Una técnica clásica para obtener ese paralelismo extra es el loop unrolling Como puede observarse en el código, se ha aplicado la técnica del LoopUnrolling sobre el bucle. Esta técnica consiste en "desenrollar" el bucle parcialmente ejecutando así varias iteraciones en un mismo ciclo del bucle. Es habitual que en muchos casos esta técnica ya sea aplicada automáticamente por el compilador, aunque en nuestro caso, para facilitarle la labor es recomendable hacerlo explícitamente.

```
for(int i=rowstart; i<rowend;i+=4)
{
        acum1+= AA[i] * V[JC[i]];
        acum2+= AA[i+1] * V[JC[i+1]];
        acum3+= AA[i+2] * V[JC[i+2]];
        acum4+= AA[i+3] * V[JC[i+3]];
}
```

[caption=Ejemplo de código usando Loop Unrolling]

Con esta técnica buscamos aprovechar el diseño de los procesadores VLIW-5way (los denominados stream core en stream computing) en los que se basa la arquitectura RV770 de ATI/AMD para aumentar el rendimiento. La arquitectura VLIW-5way implementa una forma de paralelismo a nivel de instrucción En nuestro caso hemos pasado a 4 iteraciones por ciclo de bucle, operando sobre 4 elementos para cada iteración (además mantenemos un número potencia de 2).

Realizando esto conseguimos que el compilador para la GPU empaquete las instrucciones eficazmente en los slots de la instrucción VLIW, que será ejecutada por los Processing Elements denominados X Y Z W de forma que cada uno de ellos realizará una operación de multiplicación entre un elemento de la matriz y el vector por ciclo de reloj. Como inconveniente, esta técnica

implicará para nuestro kernel un aumento del espacio requerido, que se traducirá en un mayor número de registros necesarios, en este caso hasta 15 registros de propósito general serán necesarios (sin loop unrolling únicamente necesitamos 8).

Capítulo 6

Resultados

En este capítulo mostraremos los resultados detallados de los tiempos de las implementaciones propuestas para un número de matrices de diferentes tamaños.

Todas las medidas han sido tomadas en la estación de trabajo descrita a continuación:

Componente	Descripción
CPU	Intel Core 2 6300@1.86GHz
Memoria	1GB
GPU	HD 4850, 512MB
Bus	PCIe x16
S.O.	Fedora 11 x86_64

Tabla 6.1: Descripción de la estación de trabajo

Para la recolección de algunas estas medidas se ha hecho uso de la herramienta *Stream KernelAnalyzer* ofrecida por ATI.

Esta herramienta nos ofrece un cálculo aproximado del tiempo esperado de ejecución del kernel, número de registros usados, etc. Estos valores son tomados de forma estática, sin ejecutar el kernel. Debido a esto, ha de ser el desarrollador, conocedor del kernel en profundidad y de los posibles datos de entrada que reciba, quien "sintonize" de forma adecuada estos valores para poder aproximarse lo más fielmente a la realidad que sea posible. En nuestro caso se han desechado los datos aquí ofrecidos y únicamente se mostrarán a modo de presentación ya que nuestros kernels mantienen una gran dependencia de los datos de entrada como para poder considerar que estas medidas tomadas estáticamente puedan ser dadas por válidas en un cálculo riguroso.

Para el kernel 1 de datos reales, los valores que nos ofrece esta herramienta para la GPU HD4870 (arquitectura RV770) son:

Caracterítica	valor
Name	Radeon HD 4870
GPR	15
Scratch Reg	0
Min	2.80
Max	1714.72
Avg	101.80
ALU	75
Fetch	14
Write	1
Est Cycles	96.05
ALU:Fetch	1.07
BottleNeck	ALU Ops
Thread{}Clock	0.17
Throughput	125 M Threads{}Sec

Tabla 6.2: Kernel 1 - Stream KernelAnalizer

De estos valores, de cara a la optimización del código podemos tomar en cuenta como puntos de referencia:

- GPR (General Purpose Registers):indicará en número total de GPR usados por el kernel,
- ALU: número de instrucciones de operaciones sobre la ALU usadas,
- Fect y Write: instrucciones de acceso y escritura a memoria global respectivamente,

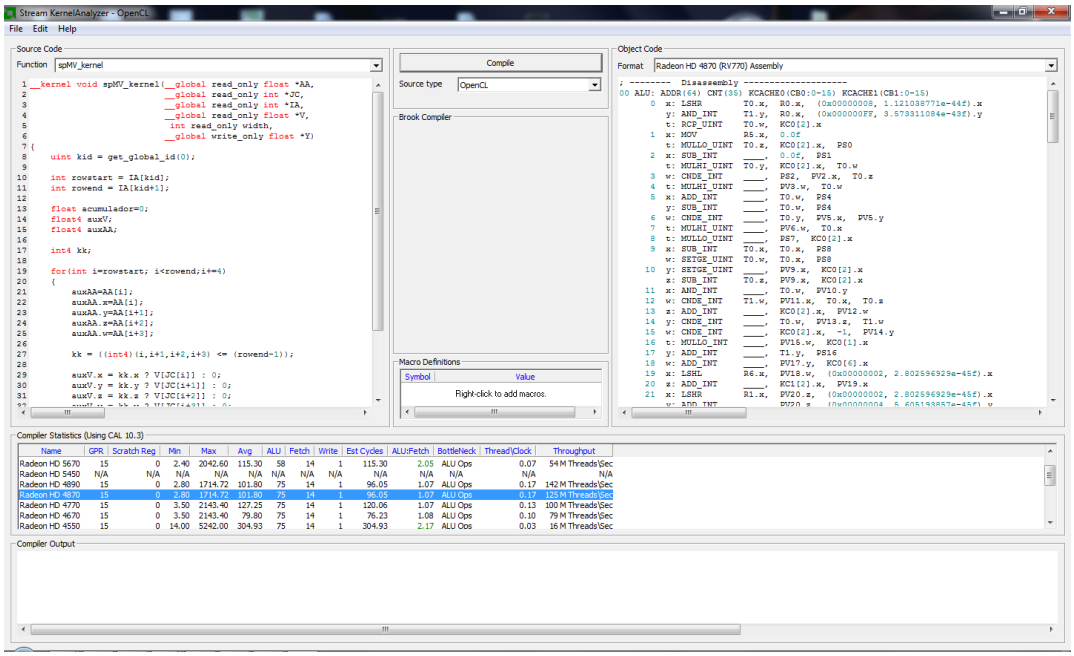


Figura 6.1: Steam KernelAnalizer - OpenCL

- BottleNeck: punto de cuello de botella que impide un mejor rendimiento. Puede ser un buen lugar por donde el programador podría empezar a buscar alternativas de cara la optimización.

Además, se hará uso del propio mecanismo ofrecido por el runtime de OpenCL que activamos mediante el flag *CL_QUEUE_PROFILING_ENABLE* en la creación de la cola y posteriormente obtenemos los datos deseados mediante la instrucción *clGetEventProfilingInfo*:

```
/* Setup OpenCL */
command_queue = clCreateCommandQueue(context, device_id,

                .
                .
                .
                .

/* Tiempo */
cl_ulong startTime, endTime;
cl_ulong kernelExecTimeNs;
int br, bw;
double bandwidth, flops, perf;

ret = clFlush(command_queue);
ret = clFinish(command_queue);

clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_START,

clGetEventProfilingInfo(events[0], CL_PROFILING_COMMAND_END,

kernelExecTimeNs = endTime - startTime;
```

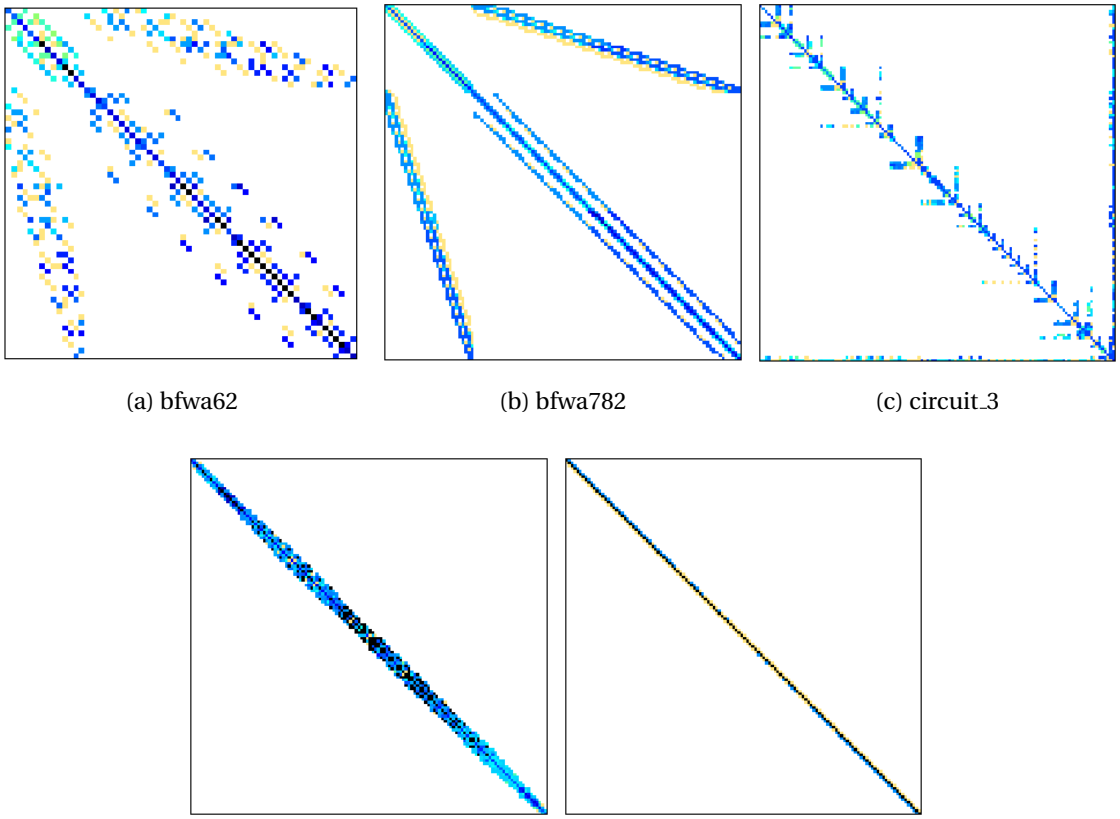
Listing 6.1: Código activación y uso modo profiling

1. Tiempos

1.1. Introducción

En esta sección se muestran las gráficas correspondientes a los tiempos totales de ejecución de los diferentes kernels sobre un conjunto de prueba de matrices descrito en las siguiente tablas.

Para las cálculos con números reales se han usado las siguientes matrices:



(a) bfwa62 (b) bfwa782 (c) circuit_3

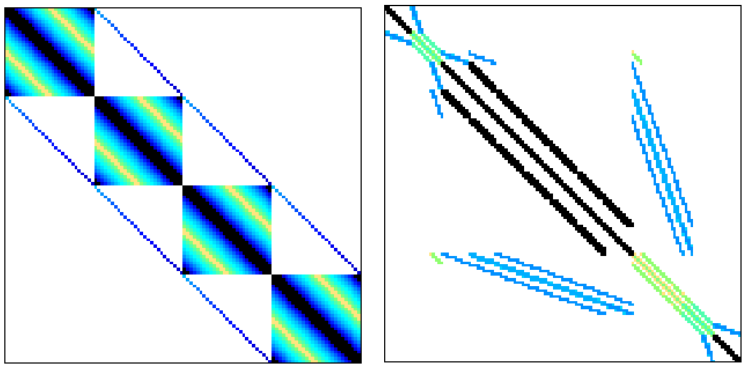
(d) ship_003 (e) tmt_unsym

Figura 6.2: Matrices del conjunto de test para datos float reales

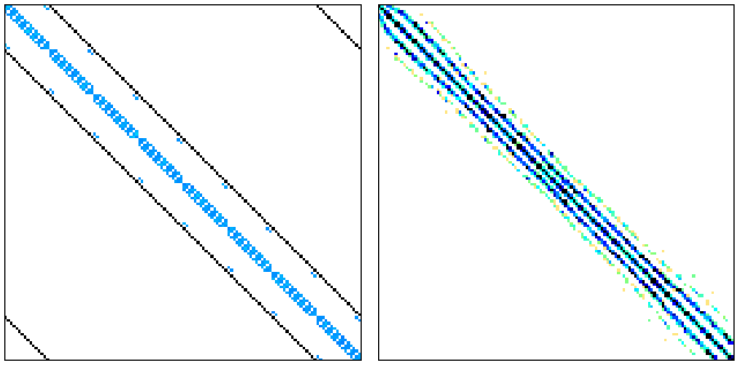
Matriz	filas	columnas	nonzeros	ceros explícitos	origen
bfwa62	62	62	450	0	problema electromagnetismo
bfwa782	782	782	7.514	0	problema electromagnetismo
circuit_3	12.127	12.127	48.137	0	problema simulación de circuito
ship_003	121.728	121.728	3.777.036	4.308.998	problema estructural
tmt_unsym	917.825	917.825	4.584.801	0	problema electromagnetismo

Tabla 6.3: Descripción de las matrices del conjunto de test para datos float reales

y para los cálculos con número complejos las siguientes:



(a) qc324 (b) mplate



(c) conf5_4-8x8-05 (d) vfem

Figura 6.3: Matrices del conjunto de test para datos float complejos

Matriz	filas	columnas	nonzeros	ceros explícitos	origen
qc324	324	324	450	0	problema electromagnetismo
mplate	782	782	7.514	0	problema acústica
conf5_4-8x8-05	49.152	49.152	1.916.928	0	problema de química cuántica
vfem	93.476	93.476	1.434.636	0	problema electromagnetismo

Tabla 6.4: Descripción de las matrices del conjunto de test para datos float complejos

1.2. Gráficas de Tiempo

1.2.1. Datos reales con precisión simple

En la siguientes figuras podemos observar una gráfica del tiempo que tarda el kernel1 y el kernel2 para realizar el producto con precisión float para cada una de las matrices del conjunto de test. El tiempo se expresa en *ns* y se ha representado en el eje de las Y con escala logarítmica.

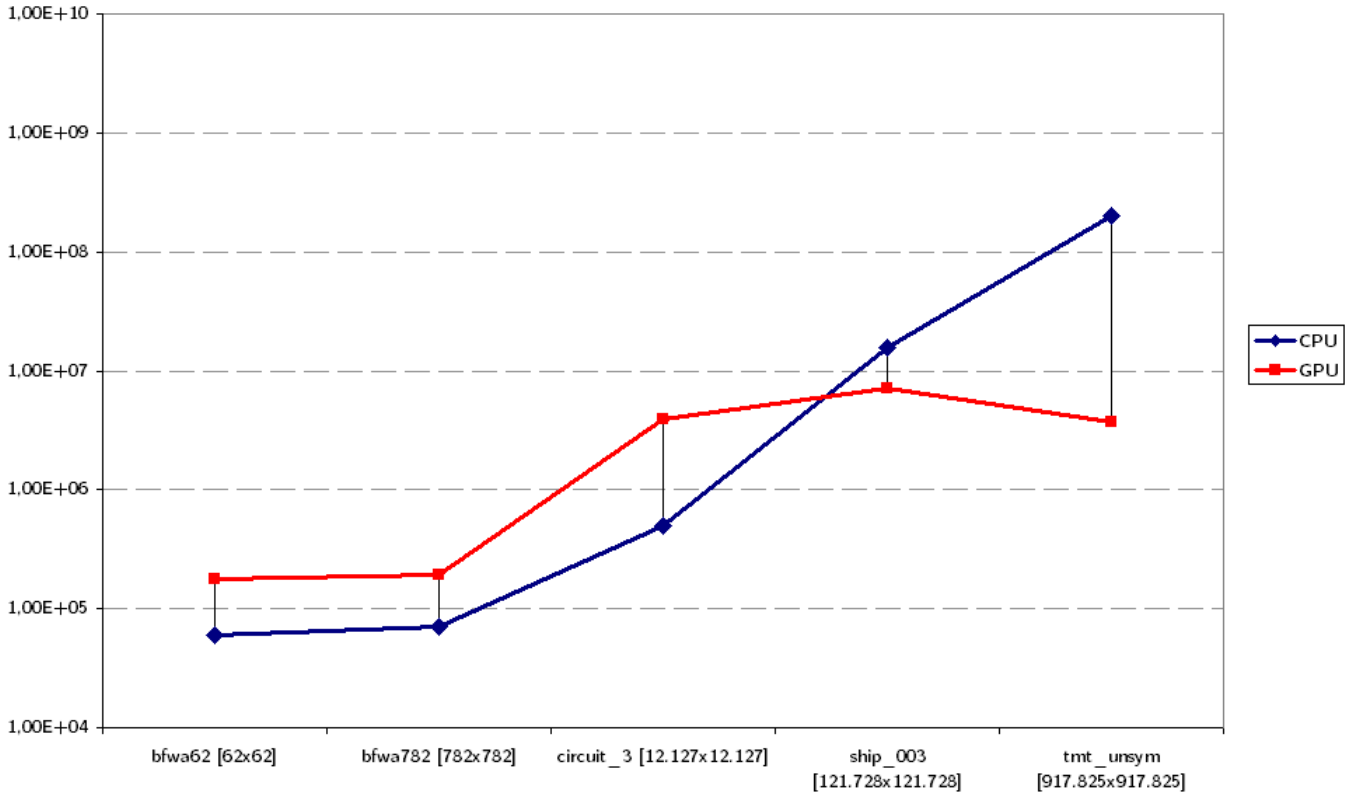


Figura 6.4: Gráfica de tiempos del kernel 1 para datos float reales

Matriz	Tiempos [ns]	
	CPU	GPU
bfwa62	59.826	175.738
bfwa782	70.918	192.366
circuit_3	503.002	3.919.120
ship_003	15.463.827	7.152.572
tmt_unsym	201.667.775	3.679.649

Tabla 6.5: Tiempos del kernel 1 para datos float reales

En este caso, cuando se alcanza una matriz cuya dimensión se aproxima a 10^5, la capacidad de cálculo de la CPU se ve claramente superada por la GPU llegando a doblar su tiempo de ejecución. Una vez alcanzado el rango de matrices entorno a 10^6 vemos como el tiempo de la CPU se dispara distanciándose en un factor entorno a 60.

Podemos observar también que el kernel 2 no solo se mantiene sino que ademas disminuye de tiempo; este comportamiento se puede achacar a la estructura de la matriz *ship_003* que a pesar de tener una dimensión y un número de elementos no cero menor, también contiene elementos cero explícitos que son evaluados como si de cualquier otro valor se tratase. Esto hace que realmente la matriz *ship_003* contenga aproximadamente el doble de productos a realizar que la matriz *tmt_unsym*.

En esta otra gráfica se muestran los valores para el kernel2 en las mismas condiciones anteriores. Claramente la CPU ve agravada su diferencia respecto con la GPU:

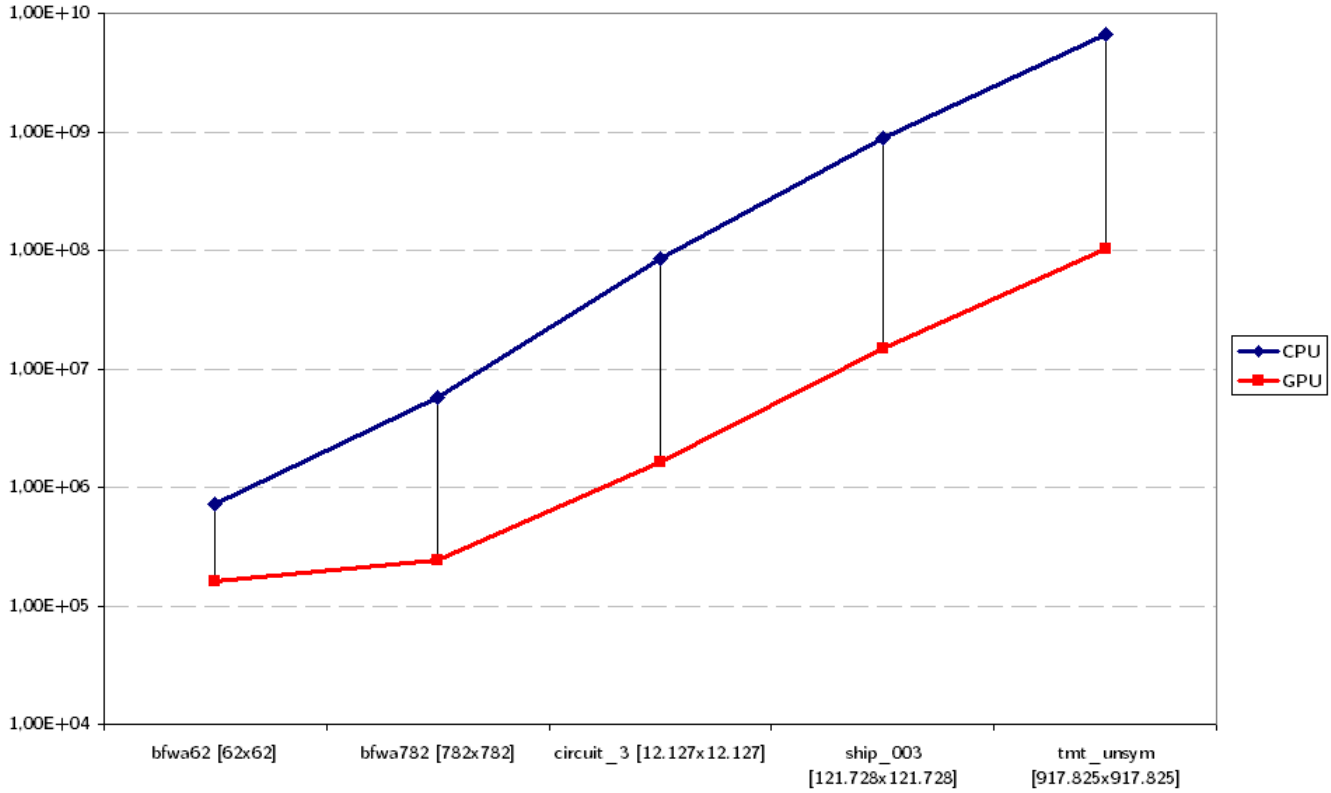


Figura 6.5: Gráfica de tiempos del kernel 2 para datos float reales

Matriz	Tiempos [ns]	
	CPU	GPU
bfwa62	718.282	162.347
bfwa782	5720331	240920
circuit_3	83.911.586	1.626.093
ship_003	875.199.253	14.803.266
tmt_unsym	6.714.947.428	104.086.224

Tabla 6.6: Tiempos del kernel 2 para datos float reales

En este caso, podemos observar como siempre se mantiene una proporcionalidad entre los tiempos de ejecución.

Si comparamos ahora las gráficas para los dos kernels anteriores:

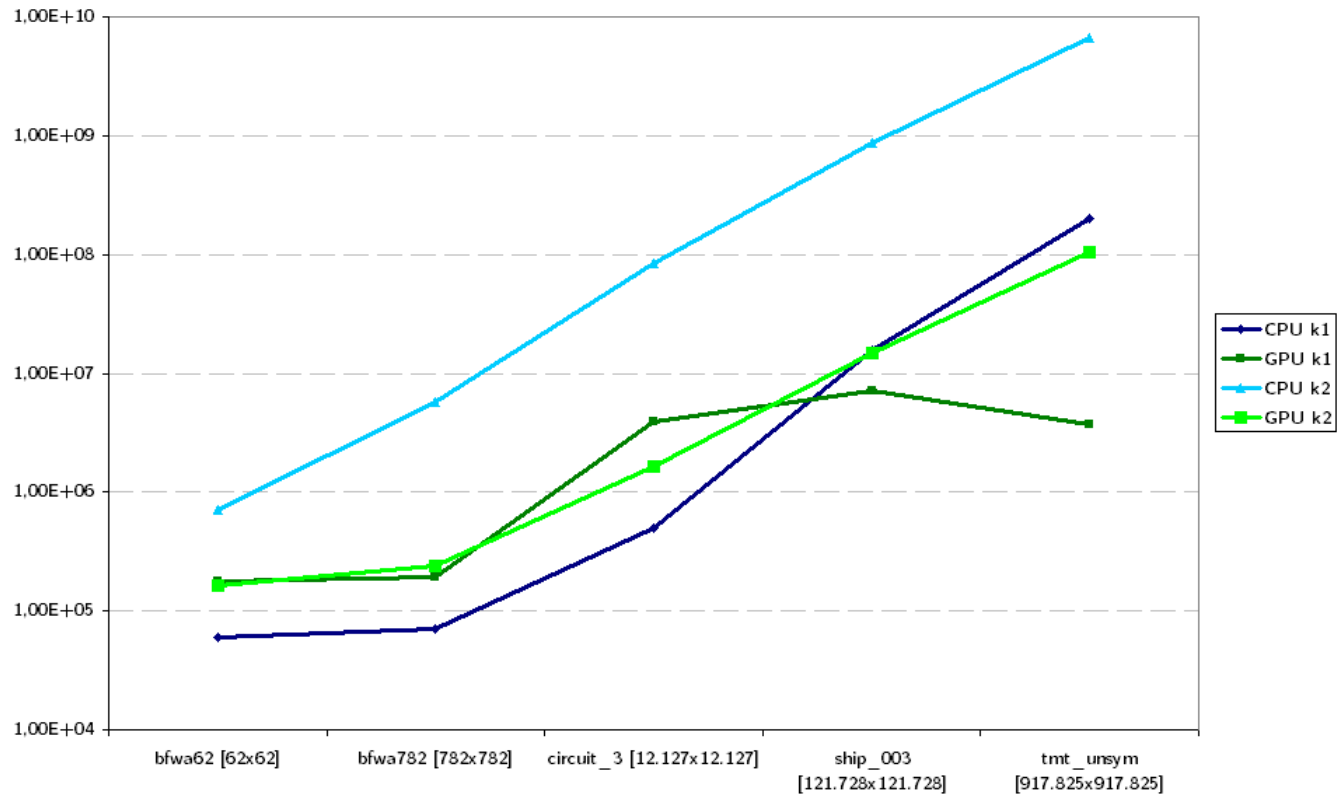


Figura 6.6: Tiempos del kernel 1 vs kernel 2 para datos float reales

podemos observar que en general el kernel 2 presenta unos tiempos superiores a su análogo kernel 1. Esta diferencia entre kernels se puede atribuir a la baja densidad de elementos no cero por fila de las matrices dispersas. Recordemos que en el kernel 2 se ejecutaban 64 work-items por fila sea cual sea su número de elementos.

1.2.2. Datos complejos con precisión simple

Para el cálculo con valores complejos se han obtenido los resultados que mostraremos a continuación. En primer lugar, para el kernel 1 obtenemos los tiempos:

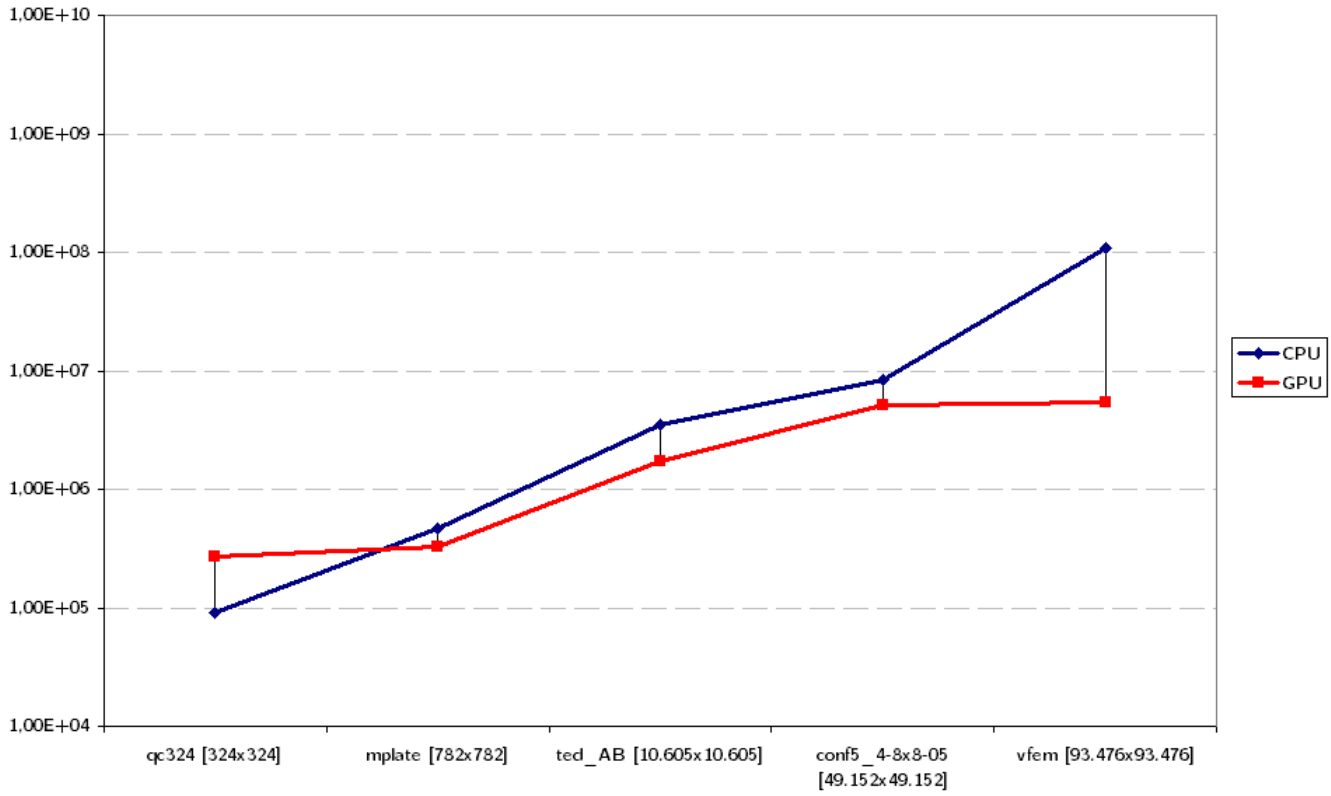


Figura 6.7: Gráfica de tiempos del kernel 1 para datos float complejos

Matriz	Tiempos [ns]	
	CPU	GPU
qc324 [324x324]	90.602	270.604
mplate [782x782]	460.090	326.001
conf5_4-8x8-05 [49.152x49.152]	8.267.551	5.076.049
vfem [93.476x93.476]	109.964.749	5.379.470

Tabla 6.7: Tiempos del kernel 1 para datos float complejos

En este caso se puede observar como para la primera matriz, con un tamaño de 324x324 el tiempo de ejecución de la GPU queda por encima de la CPU. El overhead provocado por la transferencia de datos hacia la GPU crece linealmente con el tamaño de los datos de entrada,

mientras que la complejidad del algoritmo muy por encima de estos valores. Por ello para pequeños problemas el efecto del overhead es más visible ya que la CPU no sufre de forma tan agravante el tiempo de transferencia de datos.

En cuanto a valores complejos sobre el kernel 2:

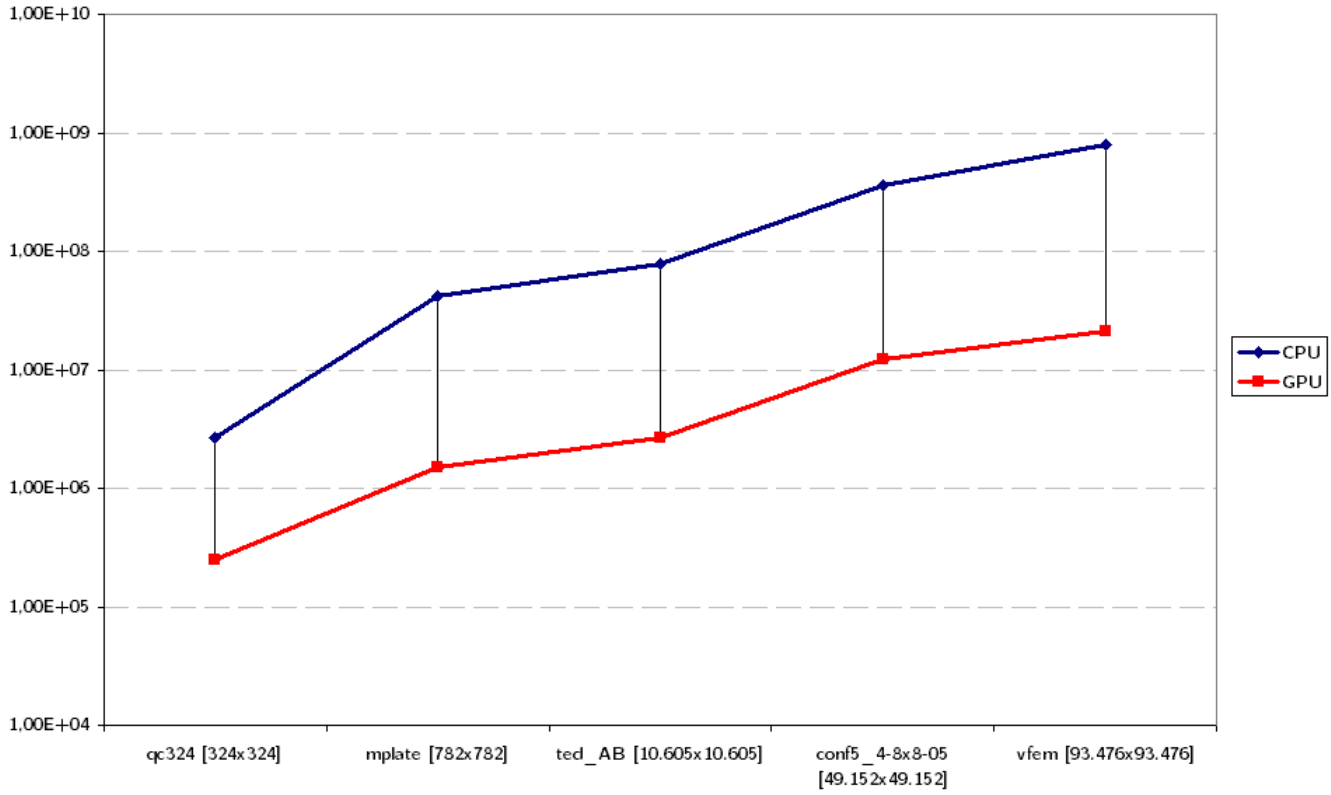


Figura 6.8: Gráfica de tiempos del kernel 2 para datos float complejos

Matriz	Tiempos [ns]	
	CPU	GPU
qc324 [324x324]	2.634.406	252.003
mplate [782x782]	42.210.130	1.514.266
conf5_4-8x8-05 [49.152x49.152]	364.058.985	12.327.753
vfem [93.476x93.476]	796.319.332	20.883.678

Tabla 6.8: Tiempos del kernel 2 para datos float complejos

En este caso, tal como ocurría con el kernel 2 para valores reales, los tiempos para la GPU se mantienen en todo momento por debajo de los tiempos de la CPU.

Comprando los dos kernels para valores complejos:

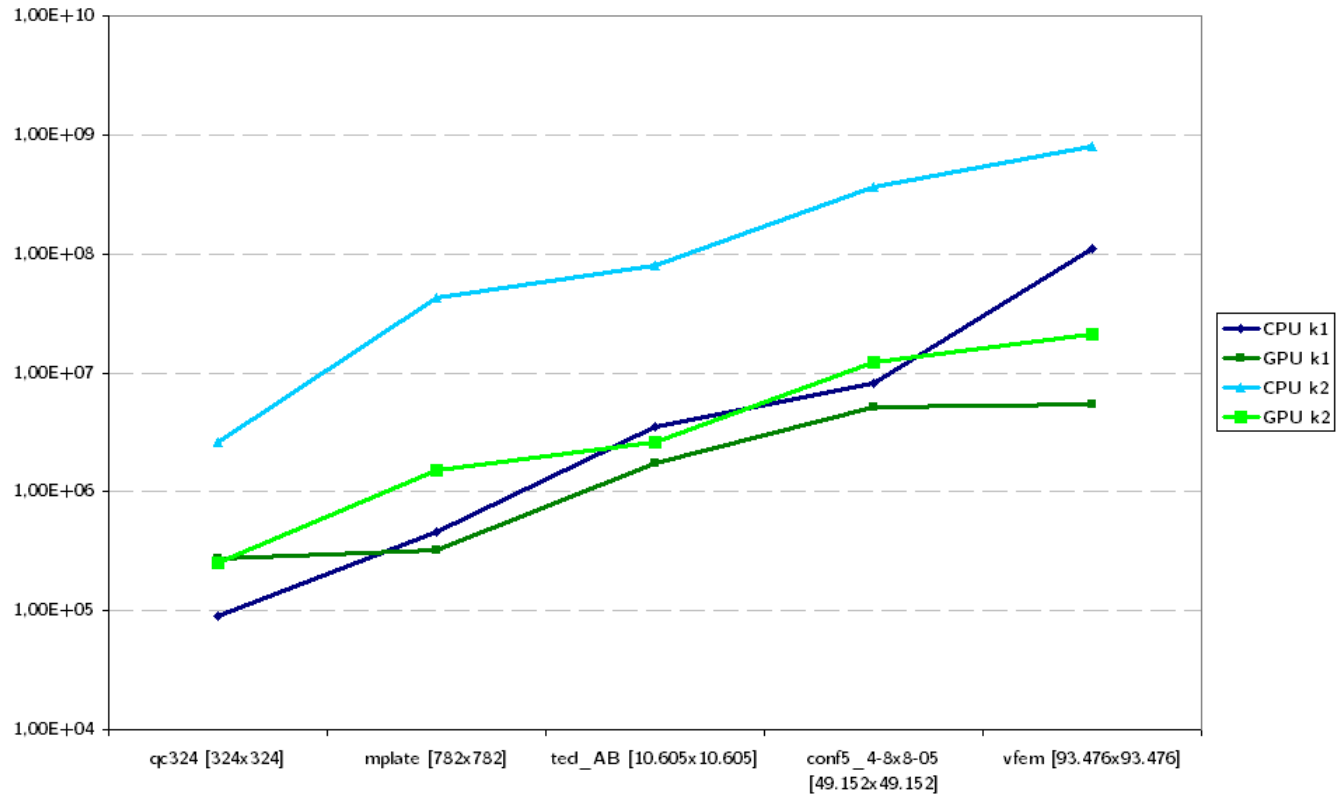


Figura 6.9: Tiempos del kernel 1 vs kernel 2 para datos float complejos

veremos que el kernel 2 es hasta dos veces más lento que el kernel1. El kernel 2 es muy sensible a la estructura de la matriz de entrada y por ello para poder obtener un buen rendimiento es necesario que el número medio de elementos no cero por fila sea muy superior a la media que se encuentra en las matrices dispersas típicas. En general podemos afirmar que el kernel 1 ofrece mejores tiempos que el kernel 2.

1.2.3. Datos double

Para poder hacer uso de la datos tipo *double* es necesario que el dispositivo soporte alguna extensión que ofrezca esta característica. En nuestro caso de ha hecho uso de la extensión que proporciona KhronosGroup llamada *cl_khr_fp64* y que será activada incluyendo la directiva *#pragma OPENCL EXTENSION cl_khr_fp64 : enable* en el inicio del código del kernel que vaya a usar el tipo double o sus derivado vectorizados (double2, double4, etc.)

NOTA:
Las gráficas aquí mostradas para las rutinas con precisión double, tanto reales como complejos, no han sido tomadas en el equipo descrito anteriormente. Debido a algún tipo de bug, OpenCL sobre cualquiera de las versiones publicadas hasta la fecha del SDK proporcionadas por ATI/AMD sobre Fedora 11 (oficialmente sólo son soportadas las distribuciones: openSUSE 11.2, Ubuntu 10.04/9.10 y Red Hat Enterprise Linux 5.5/5.4) produce un error del compilador interno de OpenCL:

Fallo en clBuildProgram
Program build log:
Internal error: Link failed.
Make sure the system setup is correct.

Por lo anterior expuesto, las gráficas aquí mostradas no han de ser usadas como medida comparativa con el resto de rutinas, únicamente se exponen con objeto de estudio del comportamiento de los kernels.

Todas las medidas han sido tomadas sobre una Máquina virtual con las siguientes características:

Componente	Descripción
CPU	Intel Core 2 8600@2.40GHz
Memoria	1024MB
S.O.	ubuntu-10.04 i386

Tabla 6.9: Descripción de la estación de trabajo 2

La gráfica de tiempos para datos con precisión double de los kernels 1 y 2 la podemos observar en la figura 6.10

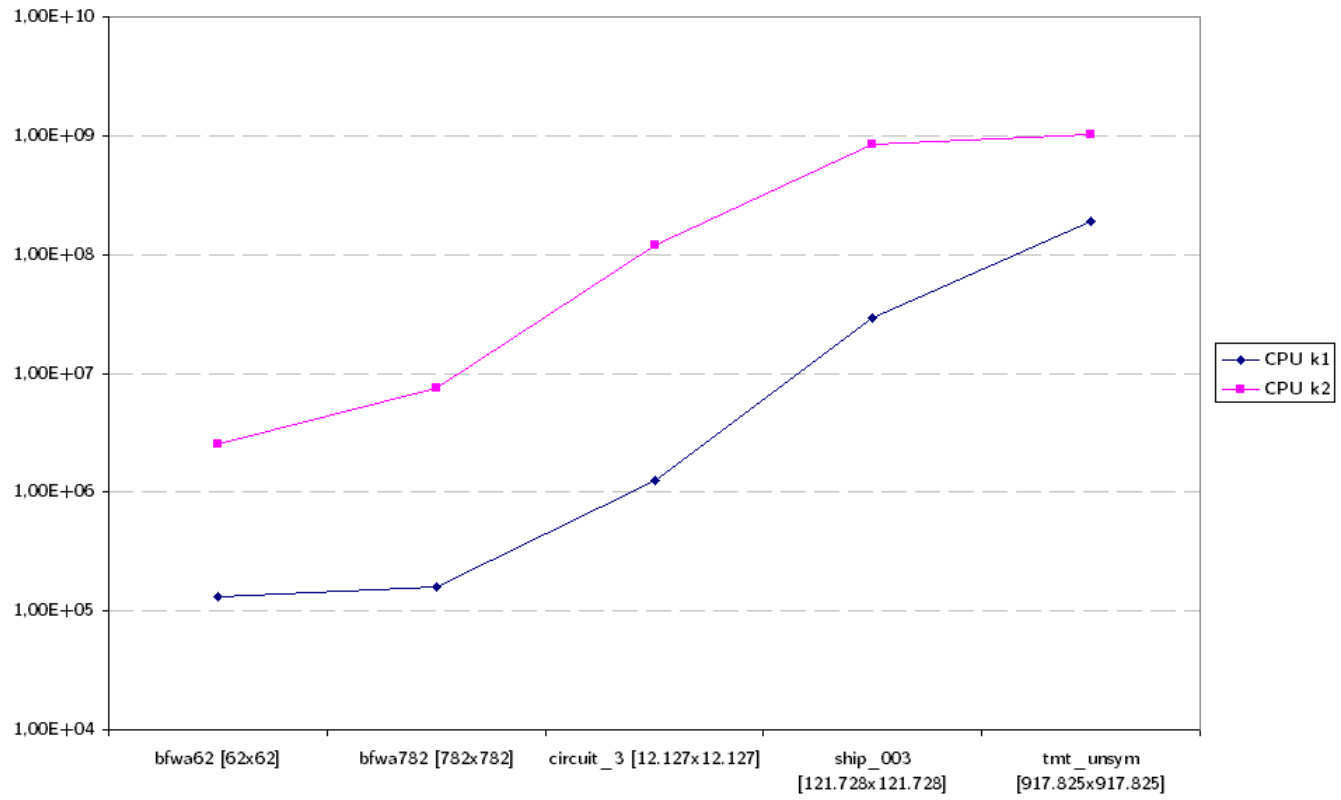


Figura 6.10: Gráfica de tiempos del kernel 1 y 2 para datos double reales

Matriz	Tiempos [ns]	
	kernel 1	kernel 2
bfwa62 [62x62]	131.515	2.537.322
bfwa782 [782x782]	159.438	7.573.992
circuit_3 [12.127x12.127]	1.261.262	120.923.090
ship_003 [121.723x121.723]	29.430.648	857.088.019
tmt_unsym [917.325x917.325]	191.296.841	1.036.371.861

Tabla 6.10: Tiempos del kernel 1 y 2 para datos double reales

En ésta se aprecia claramente como, al igual que en el caso de la precisión simple, el kernel 2 sigue resultando más lento que el kernel 1 aunque con matrices de mayor dimensión la diferencia relativa se ve disminuida. Para datos double complejos:

Para valores complejos se obtiene unos tiempos proporcionalmente mayores que en el caso anterior

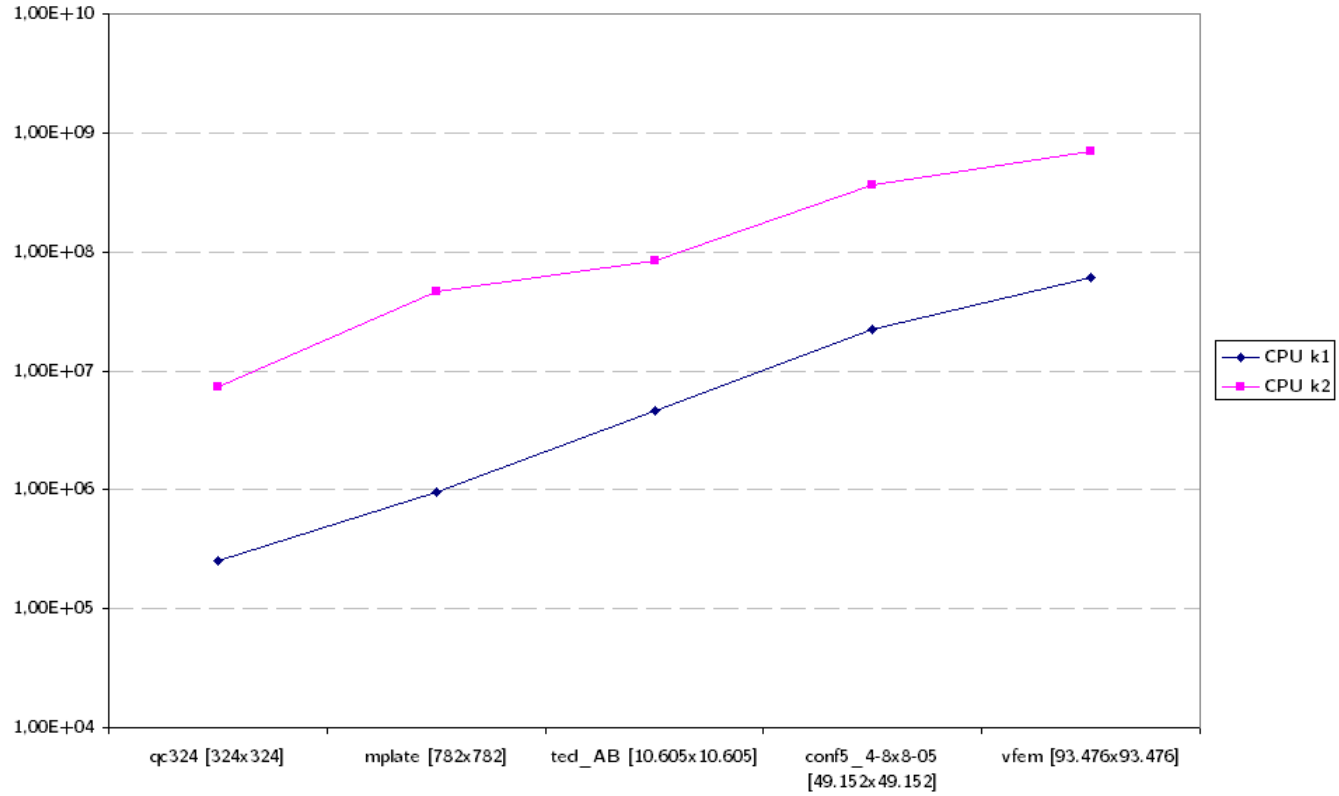


Figura 6.11: Gráfica de tiempos del kernel 1 y2 para datos double complejos

Matriz	Tiempos [ns]	
	kernel 1	kernel 2
qc324 [324x324]	252.420	7.294.209
mplate [782x782]	956.347	46.529.308
ted_AB [10.605x10.605]	4.658.877	83.084.110
conf5_4-8x8-05 [49.152x49.152]	22.088.136	365.675.121
vfem [93.476x93.476]	264.173.857	697.544.134

Tabla 6.11: Tiempos del kernel 1 y 2 para datos double reales

Capítulo 7

Conclusión

La implementación de los algoritmos propuestos especializados en el producto matriz-vector están optimizados para su ejecución sobre dispositivos GPU aunque también puede ser usado sobre la CPU. Todos las librerías y frameworks usadas en el desarrollo son libres e independientes de cualquier plataforma lo que hace que el presente proyecto pueda ser usado en un futuro sin ningún tipo de restricción.

El uso de OpenCL en su actual versión 1.1 presenta algunos desafíos para el programador. Las actuales herramientas ofrecidas para la compilación de los kernels únicamente permiten su compilación online mediante el uso de librerías específicas en tiempo de ejecución (existe un compilador desarrollado por Fixstars que permite la compilación separada del kernel pero se encuentra en una fase temprana y no se encuentra actualizado a las especificaciones de la versión 1.1). Esta falta de aislamiento del kernel del resto del programa junto con la escasa información ofrecida en caso de error dificulta enormemente todo el desarrollo. Además, hemos de tener en cuenta que actualmente no existe ningún debugger para los kernels ejecutados sobre un dispositivo que no sea la CPU y el uso de operativas i/o tipo *printf()* no son posibles debido a que el kernel se ejecuta sobre la GPU, dejándonos únicamente como alternativa el uso de los buffers escritos por el kernel y devueltos al host una vez éste finalice.

De cara a obtener un buen rendimiento, existen una gran cantidad de parámetros que el programador debe de ajustar o tener en cuenta como p.e. work-group size, número registros, tamaño wavefront, anchura bus, etc. La gran mayoría de estos parámetros guardan una estrecha relación con el hardware de la GPU sobre la cual se ejecute lo que obliga al programador a conocer en profundidad la arquitectura. A su vez, un pequeño cambio en uno de estos parámetros puede comportar una gran variación en el rendimiento aunque puede exigir la modificación de gran parte del código del kernel.

Tal como se muestra en las gráficas de la sección anterior, mediante el uso de la GPU para propósitos generales, como en nuestro caso el producto matriz-vector, podemos obtener incrementos muy significativos en el rendimiento comparados con el uso de la CPU llegando a rebajar los tiempos de ejecución muy por debajo del 50

En una futura revisión del software desarrollado, de cara a mejorar el actual rendimiento mediante un acceso a memoria más eficiente aprovechando la localidad espacial de los datos de

las diagonales en matrices dispersas, se propondría el uso de otros formatos de almacenamiento más especializamos como el CDS (también conocido como DIA), lo que conllevaría el desarrollo del código por completo para los nuevos kernels. A su vez, en caso de contar con una GPU con soporte par almacenar datos en los registros de texturas de imágenes se podría desarrollar el uso de esta memoria evitando así el caro acceso a la memoria global de la GPU que se utiliza actualmente y que no cuenta con memoria caché.

Anexo A

Anexos

1. Uso de la librería

Para poder hacer uso de la librería es necesario tener previamente instaladas las librerías de OpenCL. El software desarrollado se puede obtener bajo licencia GNU General Public License v2 en `http://sourceforge.net/projects/spmvocl/` con su Makefile si se desea crear librería a partir el código fuente correspondiente. También se ha dispuesto un conjunto de herramientas complementarias para la conversión de matrices con formato MatrixMarket format (ordenado por columna) a su equivalente ordenado por fila para su importación en CRS. Se ha incluido además el código empleado para realizar el conjunto de test con el fin de ser instructivo para el usuario final a modo de ejemplo de uso.

El paquete completo contiene los ficheros:

Fichero	Descripción
libspmvocl.a	Librería estática ya compilada
spmvocl.c	Código de la librería
Makefile.lib	Makefile para la creación de la librería
sample.c	Código ejemplo con llamadas a las funciones de la librería
spmvocl.h	Archivo de cabecera de la librería

Tabla A.1: Ficheros del paquete

Y como herramientas auxiliares se incluyen:

Fichero	Descripción
ccs2crs.c	Código fuente para la conversión CCS a CRS
ccs2crscomplex.c	Código fuente para la conversión de matrices complejas CCS a CRS

Tabla A.2: Ficheros auxiliares

También se han incluido un pequeño conjunto de matrices para su evaluación extraídas de

`http://www.cise.ufl.edu/research/sparse/matrices/` y transformadas mediante las herramientas *ccs2crs* y *ccs2crscmplx*.

Las rutinas externas que podrán ser llamadas como función de la librería *spmvocl* desarrollada son:

Función	Descripción
int sspmv(...)	Producto matriz-vector para datos de precisión simple reales.
int dspmv(...)	Producto matriz-vector para datos de precisión doble reales.
int cspmv(...)	Producto matriz-vector para datos de precisión simple complejos.
int zspmv(...)	Producto matriz-vector para datos de precisión double complejos.

Tabla A.3: Funciones de la librería

Para poder hacer uso de estas es necesario cargar el archivo de cabecera *spmvocl.h* dentro del código anfitrión mediante *#include "spmvocl.h"*

Los argumentas de entrada se encuentra descritos en la tabla 5.1 de la página 29

2. Código del software

```
1
2   //      spmvocl.c
3   //
4   //      Copyright 2010 erodriguez <ero.rodriguez@gmail.com>
5   //
6   //      This program is free software; you can redistribute it and/or modify
7   //      it under the terms of the GNU General Public License as published by
8   //      the Free Software Foundation; either version 2 of the License, or
9   //      (at your option) any later version.
10  //
11  //      This program is distributed in the hope that it will be useful,
12  //      but WITHOUT ANY WARRANTY; without even the implied warranty of
13  //      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  //      GNU General Public License for more details.
15  //
16  //      You should have received a copy of the GNU General Public License
17  //      along with this program; if not, write to the Free Software
18  //      Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
19  //      MA 02110-1301, USA.
20
21  #include <stdlib.h>
22  #include <stdio.h>
23  #include <string.h>
24  #include <CL/cl.h>
25  #include <ctype.h>
26  #include <time.h>
27
28  #define BLOCKSIZE 64
29
30  int sspmv(int krnl, float *AA, int *JC, int *IA, float *V, float **output, int width, int height, int nz);
31  int dspmv(int krnlin, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz);
32  int cspmv(int krnlin, float *AA, int *JC, int *IA, float *V, float **output, int width, int height, int nz);
33  int zspmv(int krnlin, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz);
34
35  struct opencl
36  {
37          cl_platform_id          platform_id;
38          cl_uint                         ret_num_platforms;
39          cl_device_id            device_id;
40          cl_uint                         ret_num_devices;
41          cl_context                      context;
42          cl_program                      program;
43          cl_kernel                       kernel;
44          cl_command_queue        command_queue;
45          cl_mem                          inBufAA;
46          cl_mem                          inBufJC;
47          cl_mem                          inBufIA;
48          cl_mem                          inBufV;
49          cl_mem                          outBufY;
50          cl_event events[2];
51  };
52
53  const char *kernelsd=
54  #pragma OPENCL EXTENSION cl_amd_fp64 : enable
55
56  __kernel void spMVd_kernel1(__global read_only double *AA,
57                              __global read_only int *JC,
58                              __global read_only int *IA,
59                              __global read_only double *V,
60                               int read_only width,
61                              __global write_only double *Y)
62  {
63          uint kid = get_global_id(0);
64          int rowstart = IA[kid];
65          int rowend = IA[kid+1];
66          double acumulador=0;
67          double4 auxV;
68          double4 auxAA;
69          int4 kk;
70          for(int i=rowstart; i<rowend;i+=4)
71          {
72                  auxAA.x=AA[i];
73                  auxAA.y=AA[i+1];
74                  auxAA.z=AA[i+2];
```

```
75                  auxAA.w=AA[i+3];
76                  kk = ((int4)(i,i+1,i+2,i+3) <= (rowend-1));
77                  auxV.x = kk.x ? V[JC[i]] : 0;
78                  auxV.y = kk.y ? V[JC[i+1]] : 0;
79                  auxV.z = kk.z ? V[JC[i+2]] : 0;
80                  auxV.w = kk.w ? V[JC[i+3]] : 0;
81                  acumulador+=dot(auxAA,auxV);
82          }
83          Y[kid]=acumulador;
84  }
85
86  __kernel void spMVd_kernel2(__global read_only double *data,
87                              __global read_only int *indices,
88                              __global read_only int *ptr,
89                              __global read_only double *V,
90                              int read_only width,
91                              __global write_only double *Y)
92  {
93          uint kid = get_global_id(0);
94          uint blockid = get_group_id(0);
95          uint localkid = get_local_id(0);
96          uint row = blockid;
97          if(row < width){
98                  int rowstart = ptr[row];
99                  int rowend = ptr[row+1];
100                 double val=0;
101                 double sum=0;
102                 __local double sumvals[64];
103                 for(int i=rowstart + localkid; i<rowend; i+=64)
104                         val+=data[i]*V[indices[i]];
105                 sumvals[localkid]=val;
106                 barrier(CLK_LOCAL_MEM_FENCE);
107                 if(localkid == 0){
108                         for(int i=0; i<64; i++)
109                                 sum+=sumvals[i];
110                         Y[row]=sum;
111                 }
112         }
113 }
114
115 __kernel void spMVdc_kernel1(__global read_only double2 *AA,
116                              __global read_only int *JC,
117                              __global read_only int *IA,
118                              __global read_only double2 *V,
119                               int read_only width,
120                              __global write_only double2 *Y)
121 {
122         uint kid = get_global_id(0);
123         int rowstart = IA[kid];
124         int rowend = IA[kid+1];
125         double4 auxA;
126         int col;
127         double acumr=0,acumi=0;
128         for(int i=rowstart; i<rowend; i+=1)
129         {
130                 auxA.x=AA[i].x;
131                 auxA.y=(AA[i].y)*(-1);
132                 auxA.z=AA[i].y;
133                 auxA.w=AA[i].x;
134                 col=JC[i];
135                 acumr+= dot(auxA.lo,V[col]);
136                 acumi+= dot(auxA.hi,V[col]);
137         }
138         Y[kid]= (double2)(acumr,acumi);
139 }
140
141 __kernel void spMVdc_kernel2(__global read_only double2 *data,
142                              __global read_only int *indices,
143                              __global read_only int *ptr,
144                              __global read_only double2 *V,
145                              int read_only width,
146                              __global write_only double2 *Y)
147 {
148         uint kid = get_global_id(0);
149         uint blockid = get_group_id(0);
150         uint localkid = get_local_id(0);
151         uint row = blockid;
```

```
152         if(row < width){
153                 uint rowstart = ptr[row];
154                 uint rowend = ptr[row+1];
155                 double4 auxA;
156                 double2 val=(double2)(0,0);
157                 double2 sum=(double2)(0,0);
158                 int col;
159                 __local double2 sumvals[64];
160                 for(int i=rowstart + localkid; i<rowend; i+=64)
161                 {
162                         auxA.x=data[i].x;
163                         auxA.y=(data[i].y)*(-1);
164                         auxA.z=data[i].y;
165                         auxA.w=data[i].x;
166                         col=indices[i];
167                         val.x+= dot(auxA.lo,V[col]);
168                         val.y+= dot(auxA.hi,V[col]);
169                 }
170                 sumvals[localkid].x=val.x;
171                 sumvals[localkid].y=val.y;
172                 barrier(CLK_LOCAL_MEM_FENCE);
173                 if(localkid == 0){
174                         for(int i=0; i<64; i++)
175                         {
176                                 sum.x+=sumvals[i].x;
177                                 sum.y+=sumvals[i].y;
178                         }
179                         Y[row]=(double2)(sum.x,sum.y);
180                 }
181         }
182 }
183 ;
184
185
186
187 const char *kernelsf =
188 __kernel void spMVf_kernel1(__global read_only float *AA,
189                           __global read_only int *JC,
190                           __global read_only int *IA,
191                           __global read_only float *V,
192                            int read_only width,
193                           __global write_only float *Y)
194 {
195         uint kid = get_global_id(0);
196         int rowstart = IA[kid];
197         int rowend = IA[kid+1];
198         float acumulador=0;
199         float4 auxV;
200         float4 auxAA;
201         int4 kk;
202         for(int i=rowstart; i<rowend;i+=4)
203         {
204                 auxAA.x=AA[i];
205                 auxAA.y=AA[i+1];
206                 auxAA.z=AA[i+2];
207                 auxAA.w=AA[i+3];
208                 kk = ((int4)(i,i+1,i+2,i+3) <= (rowend-1));
209                 auxV.x = kk.x ? V[JC[i]] : 0;
210                 auxV.y = kk.y ? V[JC[i+1]] : 0;
211                 auxV.z = kk.z ? V[JC[i+2]] : 0;
212                 auxV.w = kk.w ? V[JC[i+3]] : 0;
213                 acumulador+=dot(auxAA,auxV);
214         }
215         Y[kid]=acumulador;
216 }
217
218 __kernel void spMVf_kernel2(__global read_only float *data,
219                           __global read_only int *indices,
220                           __global read_only int *ptr,
221                           __global read_only float *V,
222                           int read_only width,
223                           __global write_only float *Y)
224 {
225         uint kid = get_global_id(0);
226         uint blockid = get_group_id(0);
227         uint localkid = get_local_id(0);
228         uint row = blockid;
```

```
229         if(row < width){
230                 int rowstart = ptr[row];
231                 int rowend = ptr[row+1];
232                 float val=0;
233                 float sum=0;
234                 __local float sumvals[64];
235                 for(int i=rowstart + localkid; i<rowend; i+=64)
236                         val+=data[i]*V[indices[i]];
237                 sumvals[localkid]=val;
238                 barrier(CLK_LOCAL_MEM_FENCE);
239                 if(localkid == 0){
240                         for(int i=0; i<64; i++)
241                                 sum+=sumvals[i];
242                         Y[row]=sum;
243                 }
244         }
245 }
246 __kernel void spMVf_kernel2_local(__global read_only float *data,
247                           __global read_only int *indices,
248                           __global read_only int *ptr,
249                           __global read_only float *V,
250                           int read_only width,
251                           __global write_only float *Y)
252 {
253         uint kid = get_global_id(0);
254         uint blockid = get_group_id(0);
255         uint localkid = get_local_id(0);
256         uint row = blockid;
257         if(row < width){
258                 int rowstart = ptr[row];
259                 int rowend = ptr[row+1];
260                 float val=0;
261                 float sum=0;
262                 __local float sumvals[64];
263                 for(int i=rowstart + localkid; i<rowend; i+=64)
264                         val+=data[i]*V[indices[i]];
265                 sumvals[localkid]=val;
266                 barrier(CLK_LOCAL_MEM_FENCE);
267                 if(localkid == 0){
268                         for(int i=0; i<64; i++)
269                                 sum+=sumvals[i];
270                         Y[row]=sum;
271                 }
272         }
273 }
274
275 __kernel void spMVfc_kernel1(__global read_only float2 *AA,
276                           __global read_only int *JC,
277                           __global read_only int *IA,
278                           __global read_only float2 *V,
279                            int read_only width,
280                           __global write_only float2 *Y)
281 {
282         uint kid = get_global_id(0);
283         int rowstart = IA[kid];
284         int rowend = IA[kid+1];
285         float4 auxA;
286         int col;
287         float acumr=0,acumi=0;
288         for(int i=rowstart; i<rowend; i+=1)
289         {
290                 auxA.x=AA[i].x;
291                 auxA.y=(AA[i].y)*(-1);
292                 auxA.z=AA[i].y;
293                 auxA.w=AA[i].x;
294                 col=JC[i];
295                 acumr+= dot(auxA.lo,V[col]);
296                 acumi+= dot(auxA.hi,V[col]);
297         }
298         Y[kid]= (float2)(acumr,acumi);
299 }
300
301 __kernel void spMVfc_kernel2(__global read_only float2 *data,
302                           __global read_only int *indices,
303                           __global read_only int *ptr,
304                           __global read_only float2 *V,
305                           int read_only width,
```

```
306                         __global write_only float2 *Y)
307 {
308         uint kid = get_global_id(0);
309         uint blockid = get_group_id(0);
310         uint localkid = get_local_id(0);
311         uint row = blockid;
312         if(row < width){
313                 uint rowstart = ptr[row];
314                 uint rowend = ptr[row+1];
315                 float4 auxA;
316                 float2 val=(float2)(0.f,0.f);
317                 float2 sum=(float2)(0.f,0.f);
318                 int col;
319                 __local float2 sumvals[64];
320                 for(int i=rowstart + localkid; i<rowend; i+=64)
321                 {
322                         auxA.x=data[i].x;
323                         auxA.y=(data[i].y)*(-1);
324                         auxA.z=data[i].y;
325                         auxA.w=data[i].x;
326                         col=indices[i];
327                         val.x+= dot(auxA.lo,V[col]);
328                         val.y+= dot(auxA.hi,V[col]);
329                 }
330                 sumvals[localkid].x=val.x;
331                 sumvals[localkid].y=val.y;
332                 barrier(CLK_LOCAL_MEM_FENCE);
333                 if(localkid == 0){
334                         for(int i=0; i<64; i++)
335                         {
336                                 sum.x+=sumvals[i].x;
337                                 sum.y+=sumvals[i].y;
338                         }
339                         Y[row]=(float2)(sum.x,sum.y);
340                 }
341         }
342 }
343 ;
344
345
346 /*
347  *
348  * RUTINAS FLOAT
349  *
350  */
351
352 void setup_openCL_f(int krnl, int complex, float *AA, int *JC, int *IA, float *V, int width, int height, int nz, struct opencl*
        config )
353 {
354         int     ret;
355         size_t source_size;
356         cl_device_local_mem_type        local_mem_type;
357         cl_bool                                         img_support;
358
359         /* Get Platform/Device Information */
360         ret = clGetPlatformIDs(1, &(config->platform_id), &(config->ret_num_platforms));
361         ret = clGetDeviceIDs(config->platform_id, CL_DEVICE_TYPE_GPU, 1, &(config->device_id),&(config->ret_num_devices));
362         if(ret==CL_DEVICE_NOT_FOUND)
363                 ret = clGetDeviceIDs(config->platform_id, CL_DEVICE_TYPE_CPU, 1, &(config->device_id),&(config->ret_num_devices));
364
365         /* Create OpenCL Context */
366         config->context = clCreateContext(NULL, 1, &(config->device_id), NULL, NULL, &ret);
367
368         /* Create command queue */
369         config->command_queue = clCreateCommandQueue(config->context,config->device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
370
371         /* Create Buffer Object */
372         config->inBufAA = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(float)* nz * complex, 0, &ret);
373         config->inBufJC = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(int)* nz , 0, &ret);
374         config->inBufIA = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(int)* (height+1) , 0, &ret);
375         config->inBufV  = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(float)* width * complex, 0, &ret);
376         config->outBufY = clCreateBuffer(config->context, CL_MEM_WRITE_ONLY , sizeof(float) * width * complex, NULL, &ret);
377
378         /* Copy input data to memory object */
379         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufAA, CL_TRUE, 0, sizeof(float) * nz * complex, AA, 0, NULL,
              NULL);
380         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufJC, CL_TRUE, 0, sizeof(int) * nz, JC, 0, NULL, NULL);
```

```
381         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufIA, CL_TRUE, 0, sizeof(int) * (height+1), IA, 0, NULL, NULL
                );
382         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufV, CL_TRUE, 0, sizeof(float) * width * complex , V, 0, NULL
                , NULL);
383
384
385         /* Create kernel program from source file */
386         source_size = sizeof(char)*strlen(kernelsf);
387         config->program = clCreateProgramWithSource(config->context, 1, (const char **)&kernelsf,(const size_t *)&source_size, &
                ret);
388         ret = clBuildProgram(config->program, 1, &(config->device_id), NULL, NULL, NULL);
389
390         /* Create data parallel OpenCL kernel */
391         ret = clGetDeviceInfo(config->device_id, CL_DEVICE_LOCAL_MEM_TYPE, sizeof(local_mem_type), &local_mem_type, NULL);
392
393         if(complex==1)
394         {
395                 if(krnl==1)
396                         config->kernel = clCreateKernel(config->program,"spMVf_kernel1", &ret);
397                 else
398                         config->kernel = clCreateKernel(config->program, "spMVf_kernel2", &ret);
399         }
400         else
401         {
402                 if(krnl==1)
403                 {
404                         config->kernel = clCreateKernel(config->program, "spMVfc_kernel1", &ret);
405                 }
406                 else if(krnl==2)
407                 {
408                         config->kernel = clCreateKernel(config->program, "spMVfc_kernel2", &ret);
409                 }
410         }
411
412         /* Set OpenCL arguments */
413         ret = clSetKernelArg(config->kernel, 0, sizeof(cl_mem), (void *)&(config->inBufAA));
414         ret = clSetKernelArg(config->kernel, 1, sizeof(cl_mem), (void *)&(config->inBufJC));
415         ret = clSetKernelArg(config->kernel, 2, sizeof(cl_mem), (void *)&(config->inBufIA));
416         ret = clSetKernelArg(config->kernel, 3, sizeof(cl_mem), (void *)&(config->inBufV));
417         ret = clSetKernelArg(config->kernel, 4, sizeof(int), (void *)&width);
418         ret = clSetKernelArg(config->kernel, 5, sizeof(cl_mem), (void *)&(config->outBufY));
419
420 }
421
422 void run_kernel_f(int krnl,int complex, float **output, int width, struct opencl* config)
423 {
424         int ret;
425
426         /* Define workspace dimension and size */
427         /* Execute OpenCL kernel as data parallel */
428         if(krnl==1)
429         {
430                 size_t global_work_size[] = {width,0,0};
431                 ret = clEnqueueNDRangeKernel(config->command_queue, config->kernel, 1, NULL, global_work_size, NULL, 0, NULL, &(
                        config->events[0]));
432         }
433         else if(krnl==2)
434         {
435
436                 size_t global_work_size[] = {width*BLOCKSIZE,0,0};
437                 size_t local_work_size[] = {BLOCKSIZE,0,0};
438                 ret = clEnqueueNDRangeKernel(config->command_queue, config->kernel, 1, NULL, global_work_size, local_work_size, 0,
                         NULL, &(config->events[0]));
439         }
440
441         /* Wait kernel finalization (it's running in async mode) */
442         ret = clWaitForEvents(1, &(config->events[0]));
443
444         /* Transfer result to host */
445         ret = clEnqueueReadBuffer(config->command_queue, config->outBufY, CL_TRUE, 0, sizeof(float) * width * complex, *output, 0,
                 NULL, &(config->events[1]));
446
447         ret = clWaitForEvents(1, &(config->events[1]));
448 }
449
450
451 /*
```

```
452  *
453  * RUTINAS DOUBLE
454  *
455  */
456
457 void setup_openCL_d(int krnl, int complex, double *AA, int *JC, int *IA, double *V, int width, int height, int nz, struct opencl*
       config )
458 {
459         int      ret;
460         size_t source_size;
461         cl_device_local_mem_type         local_mem_type;
462         cl_bool                                          img_support;
463
464         /* Get Platform/Device Information */
465         ret = clGetPlatformIDs(1, &(config->platform_id), &(config->ret_num_platforms));
466         ret = clGetDeviceIDs(config->platform_id, CL_DEVICE_TYPE_GPU, 1, &(config->device_id),&(config->ret_num_devices));
467         if(ret==CL_DEVICE_NOT_FOUND)
468                 ret = clGetDeviceIDs(config->platform_id, CL_DEVICE_TYPE_CPU, 1, &(config->device_id),&(config->ret_num_devices));
469
470         /* Create OpenCL Context */
471         config->context = clCreateContext(NULL, 1, &(config->device_id), NULL, NULL, &ret);
472
473         /* Create command queue */
474         config->command_queue = clCreateCommandQueue(config->context,config->device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
475
476         /* Create Buffer Object */
477         config->inBufAA = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(double)* nz * complex, 0, &ret);
478         config->inBufJC = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(int)* nz , 0, &ret);
479         config->inBufIA = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(int)* (height+1) , 0, &ret);
480         config->inBufV  = clCreateBuffer(config->context, CL_MEM_READ_ONLY , sizeof(double)* width * complex, 0, &ret);
481         config->outBufY = clCreateBuffer(config->context, CL_MEM_WRITE_ONLY , sizeof(double) * width * complex, NULL, &ret);
482
483         /* Copy input data to memory object */
484         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufAA, CL_TRUE, 0, sizeof(double) * nz * complex, AA, 0, NULL,
                NULL);
485         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufJC, CL_TRUE, 0, sizeof(int) * nz, JC, 0, NULL, NULL);
486         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufIA, CL_TRUE, 0, sizeof(int) * (height+1), IA, 0, NULL, NULL
              );
487         ret = clEnqueueWriteBuffer(config->command_queue, config->inBufV, CL_TRUE, 0, sizeof(double) * width * complex , V, 0,
              NULL, NULL);
488
489         /* Create kernel program from source file */
490         source_size = sizeof(char)*strlen(kernelsf);
491         config->program = clCreateProgramWithSource(config->context, 1, (const char **)&kernelsf,(const size_t *)&source_size, &
              ret);
492         ret = clBuildProgram(config->program, 1, &(config->device_id), NULL, NULL, NULL);
493
494         /* Create data parallel OpenCL kernel */
495         ret = clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_TYPE, sizeof(cl_uint), &localType, 0);
496         ret = clGetDeviceInfo(config->device_id, CL_DEVICE_LOCAL_MEM_TYPE, sizeof(local_mem_type), &local_mem_type, NULL);
497
498         if(complex==1)
499         {
500                 if(krnl==1)
501                         config->kernel = clCreateKernel(config->program,"spMVd_kernel1", &ret);
502                 else
503                         config->kernel = clCreateKernel(config->program, "spMVd_kernel2", &ret);
504         }
505         else
506         {
507                 if(krnl==1)
508                 {
509                         config->kernel = clCreateKernel(config->program, "spMVdc_kernel1", &ret);
510                 }
511                 else if(krnl==2)
512                 {
513                         config->kernel = clCreateKernel(config->program, "spMVdc_kernel2", &ret);
514                 }
515         }
516
517         /* Set OpenCL arguments */
518         ret = clSetKernelArg(config->kernel, 0, sizeof(cl_mem), (void *)&(config->inBufAA));
519         ret = clSetKernelArg(config->kernel, 1, sizeof(cl_mem), (void *)&(config->inBufJC));
520         ret = clSetKernelArg(config->kernel, 2, sizeof(cl_mem), (void *)&(config->inBufIA));
521         ret = clSetKernelArg(config->kernel, 3, sizeof(cl_mem), (void *)&(config->inBufV));
522         ret = clSetKernelArg(config->kernel, 4, sizeof(int), (void *)&width);
523         ret = clSetKernelArg(config->kernel, 5, sizeof(cl_mem), (void *)&(config->outBufY));
```

```
524
525 }
526
527 void run_kernel_d(int krnl,int complex, double **output, int width, struct opencl* config)
528 {
529         int ret;
530
531         /* Define workspace dimension and size */
532         /* Execute OpenCL kernel as data parallel */
533         if(krnl==1)
534         {
535                 size_t global_work_size[] = {width,0,0};
536                 ret = clEnqueueNDRangeKernel(config->command_queue, config->kernel, 1, NULL, global_work_size, NULL, 0, NULL, &(
                        config->events[0]));
537         }
538         else if(krnl==2)
539         {
540
541                 size_t global_work_size[] = {width*BLOCKSIZE,0,0};
542                 size_t local_work_size[] = {BLOCKSIZE,0,0};
543                 ret = clEnqueueNDRangeKernel(config->command_queue, config->kernel, 1, NULL, global_work_size, local_work_size, 0,
                         NULL, &(config->events[0]));
544         }
545
546         /* Wait kernel finalization (it's running in async mode) */
547         ret = clWaitForEvents(1, &(config->events[0]));
548
549         /* Transfer result to host */
550         ret = clEnqueueReadBuffer(config->command_queue, config->outBufY, CL_TRUE, 0, sizeof(double) * width * complex, *output,
                0, NULL, &(config->events[1]));
551
552         ret = clWaitForEvents(1, &(config->events[1]));
553 }
554
555 /*
556  *
557  * RUTINAS AUXULIARES
558  *
559  */
560 void display_prfmc(int krnl, int complex,int width, int height, int nz, struct opencl *config)
561 {
562         int ret;
563         cl_ulong startTime, endTime;
564         cl_ulong kernelExecTimeNs;
565         int br, bw;
566         double bandwidth,flops,perf;
567
568         ret = clFlush(config->command_queue);
569         ret = clFinish(config->command_queue);
570
571         clGetEventProfilingInfo(config->events[0], CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &startTime, NULL);
572         clGetEventProfilingInfo(config->events[0], CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &endTime, NULL);
573         kernelExecTimeNs = endTime - startTime;
574         br=(nz *4*complex)*2+(height+1)*4*complex +(width*4*complex);
575         bw=1*width*4*complex;
576
577         bandwidth=(br+bw)/(double)kernelExecTimeNs;
578         flops = 2 * nz*complex;
579         perf = (flops / kernelExecTimeNs);
580
581         printf("Tiempo ejecucion kernel %: %d ns\n", krnl,kernelExecTimeNs);
582         printf("Bytes leidos y escritos de meoria global por el kernel: % | %\n", br, bw);
583         printf("Bandwidth efectivo: % GB/s\n", bandwidth);
584         printf("Gflops/s: %\n", perf);
585 }
586
587
588 void clear_openCL(struct opencl* config)
589 {
590         /* Finalization */
591         int ret;
592
593         ret = clReleaseKernel(config->kernel);
594         ret = clReleaseProgram(config->program);
595         ret = clReleaseMemObject(config->inBufAA);
596         ret = clReleaseMemObject(config->inBufJC);
597         ret = clReleaseMemObject(config->inBufIA);
```

```
598         ret = clReleaseMemObject(config->inBufV);
599         ret = clReleaseMemObject(config->outBufY);
600         ret = clReleaseCommandQueue(config->command_queue);
601         ret = clReleaseContext(config->context);
602         ret = clReleaseEvent(*(config->events));
603 }
604
605
606 /*
607  *
608  * RUTINAS PRINCIPALES
609  *
610  */
611
612 //
613 // name: sspmv
614 // @param int krnlin, float *AA, int *JC, int *IA, float *V, float **output, int width, int height, int nz
615 // @return int
616 //
617 int sspmv(int krnl, float *AA, int *JC, int *IA, float *V, float **output, int width, int height, int nz)
618 {
619         /* Define openCL varibles */
620         struct opencl config;
621
622         *output=(float *)malloc(width*sizeof(float));
623
624         setup_openCL_f(krnl,1,AA,JC,IA,V,width,height,nz,&config);
625         run_kernel_f(krnl,1,output,width,&config);
626         display_prfmc(krnl,1, width,height,nz,&config);
627         clear_openCL(&config);
628
629         return 1;
630 }
631
632 //
633 // nombre: dspmv
634 // @param int krnlin, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz
635 // @return int
636 int dspmv(int krnl, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz)
637 {
638         struct opencl config;
639
640         *output=(double *)malloc(width*sizeof(double));
641
642         setup_openCL_d(krnl,1,AA,JC,IA,V,width,height,nz,&config);
643         run_kernel_d(krnl,1,output,width,&config);
644         display_prfmc(krnl,1, width,height,nz,&config);
645         clear_openCL(&config);
646
647         return 1;
648 }
649
650 //
651 // nombre: cspmv
652 // @param int krnlin, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz
653 // @return int
654 int cspmv(int krnl, float *AA, int *JC, int *IA, float *V, float **output, int width, int height, int nz)
655 {
656         struct opencl config;
657
658         *output=(float *)malloc(width*sizeof(float));
659
660         setup_openCL_f(krnl,2,AA,JC,IA,V,width,height,nz,&config);
661         run_kernel_f(krnl,2,output,width,&config);
662         display_prfmc(krnl,2, width,height,nz,&config);
663         clear_openCL(&config);
664
665         return 1;
666 }
667
668 //
669 // nombre: zspmv
670 // @param int krnlin, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz
671 // @return int
672 int zspmv(int krnl, double *AA, int *JC, int *IA, double *V, double **output, int width, int height, int nz)
673 {
674         struct opencl config;
```

```
675
676         *output=(double *)malloc(width*sizeof(double));
677
678         setup_openCL_d(krnl,2,AA,JC,IA,V,width,height,nz,&config);
679         run_kernel_d(krnl,2,output,width,&config);
680         display_prfmc(krnl,2, width,height,nz,&config);
681         clear_openCL(&config);
682
683         return 1;
684 }
```

Bibliografía

[1] Gordon Moore. The future of integrated electronics. *Fairchild Semiconductor internal publication*, 1964.

[2] `http://folding.stanford.edu/`.

[3] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X. doi: http://doi.acm.org/10.1145/582034.582089.

[4] Anders Krog. *An introduction to hidden markov models for biological sequences. In Computational Methods in Molecular Biology*. Elseiver, 1998.

[5] `http://www.khronos.org/opencl/`.

[6] M. R. Hestenes and E. Stiefel. Methods of conjuate gradients for solving linear systems. *J. Res. Nat. Bur. Standards*, (49):409–436, 1952.

[7] J. K. Reid. On the method of conjugate gradients for the solution of large sparse linear equations. In J. K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 231–254. Academic Press, New York, NY, USA, 1971.

[8] `http://www.nvidia.com/object/cuda_home_new.html`.

[9] `http://developer.amd.com/gpu/ATIStreamSDK`.

[10] `http://www.netlib.org/blas/`.

[11] `http://www.netlib.org/lapack/`.

[12] `http://www.netlib.org/scalapack/`.

Resumen

Debido al gran número de transistores por mm^2 que hoy en día podemos encontrar en las GPU convencionales, en los últimos años éstas se vienen utilizando para propósitos generales gracias a que ofrecen un mayor rendimiento para computación paralela. Este proyecto implementa el producto sparse matrix-vector sobre OpenCL. En los primeros capítulos hacemos una revisión de la base teórica necesaria para comprender el problema. Después veremos los fundamentos de OpenCL y del hardware sobre el que se ejecutarán las librerías desarrolladas. En el siguiente capítulo seguiremos con una descripción del código de los kernels y de su flujo de datos. Finalmente, el software es evaluado basándose en comparativas con la CPU.

Degut al gran nombre de transistors per mm^2 que avui en dia podem trobar a les GPU convencionals, en els darrers anys es venen utilitzant per a propòsits generals degut a que ofereixen un major rendiment per a la computació paral·lela. Aquest projecte implementa el producte sparse matrix-vector a sobre OpenCL. Als primers capítols fem una revisió de la base teòrica necessària per a comprendre el problema. Després veurem els fonaments d'OpenCL y del hardware a sobre el que s'executarà les llibreries desenvolupades. Al següent capítol seguirem amb una descripció del codi dels dos kernels i el seu flux de dades. Finalment, el software es avaluat en base a comparatives amb la CPU.

Due to the large number of transistors per mm^2 that today we can find in conventional GPU, in recent years it have been used for general purpose since they offer higher performance for data parallel computations. This project implements the product sparse matrix-vector on OpenCL. In the first chapter we review the theoretical basis necessary to understand the problem. Then we will see the foundations of OpenCL and hardware on which to run the developed software. In the next chapter will continue with a description of the kernel code and its data flow. Finally, the software is evaluated based on comparisons with the CPU.