UNIVERSIDAD DE

ALMERÍA

ESCUELA POLITÉCNICA SUPERIOR

Y

FACULTAD DE CIENCIAS EXPERIMENTALES

INGENIERÍA INFORMÁTICA

Acelerando el producto de matrices dispersas con GPUs en entorno MATLAB: MAT-ELLRT

Alumno:

Diego Jesús García Guimarey

Directores:

Dra. Gracia Ester Martín Garzón

Gloria Ortega López

Almería, Octubre de 2013

Índice general

1. Introducción ... 6
2. GPU computing ... 8
2.1 Modelo computacional ... 10
2.1.1 Topología del GRID ... 11
2.1.2 Divergencia ... 11
2.1.3 Ocupación ... 11
2.1.4 Gestión de la memoria ... 12
2.2 Arquitecturas GPU ... 14
2.2.1 Arquitectura Fermi ... 14
2.2.2 Arquitectura Kepler ... 15
2.3 CUDA como interfaz de programación ... 17
2.3.1 Estructura de un programa CUDA ... 18
2.4 Integrando GPU computing (MEX-file) ... 22
3. Algebra dispersa ... 25
3.1 Almacenamiento por coordenadas (COO) ... 25
3.2 Compressed Row Storage (CRS) ... 25
3.3 ELLPACK-R ... 26
3.4 ELLR-T ... 29
4. MAT-ELLR-T ... 32
4.1 Introducción ... 32
4.2 Load Matrix ... 36
4.3 Configuration GPU ... 39
4.4 Matrix ELLRT ... 41
4.5 Kernel SpMV ... 44
4.6 Interfaz de usuario ... 46
4.6.1 Bloque de procesamiento de datos ... 48
4.6.2 Bloque de cálculo ... 51
5. Evaluaciones ... 55
5.1 Introducción ... 55
5.2 Recursos para la evaluación ... 55
5.3 Matrices a evaluar ... 56
5.4 Evaluaciones de SpMV ... 57
5.5 Evaluaciones de CG ... 60

6. Conclusiones y trabajos futuros 64
7. Bibliografía 65
Anexos 68

Índice de figuras

Figura 2. 1 Distribución del procesador CPU 4 cores y GPU 8SM 8
Figura 2. 2 Modelo computacional CPU vs GPU 10
Figura 2. 3 Arquitectura Fermi 14
Figura 2. 4 Arquitectura kepler 15
Figura 2. 5 Diferencia entre Fermi y Kepler 16
Figura 2. 6 GPU dinámica 16
Figura 2. 7 Interfaces de programación para la GPU 17
Figura 2. 8 Llamadas del host al device 18
Figura 2. 9 Arquitectura de un Grid 20
Figura 2. 10 Identificadores de los threads 20
Figura 2. 11 Diferencia entre código C y CUDA 21
Figura 2. 12 Entradas y salidas de datos a través del MEX 24

Figura 3. 1 Ilustración del formato CRS 26
Figura 3. 2   Formato ELLPACK-R y kernel para computar SpMV en GPUs 27
Figura 3. 3 Histograma de un ejemplo de una matriz pequeña que sirve como entrada hipotéticos pequeños warps de ocho hilos que colaboran en la computación de SpMV en la GPU 28
Figura 3. 4 El almacenamiento de memoria ELL-T de la matriz dispersa cumplimiento la coalescencia y las condiciones de alineación para T = 2. 30

Figura 4. 1 Esquema de la integración de MAT-ELLR-T por módulos 35
Figura 4. 2 Descomposición por rutinas de LoadMatrix 37
Figura 4. 3 Descomposición por rutinas de Configuration GPU 40
Figura 4. 4 Descomposición por rutinas de MatrixELLR-T 42
Figura 4. 5 Descomposición por rutinas de SpMV 44
Figura 4. 6 Imagen del flujo de ejecución 33
Figura 4. 7 Imagen de los módulos por bloques 34
Figura 4. 8 Interfaz de PackELLR-T 46
Figura 4. 9 Formato de entrada del fichero .mtx de la matriz 48
Figura 4. 10 Interfaz de ejecución del modulo Simetria_s 48
Figura 4. 11 Código que verifica si existe una matriz cargada 49
Figura 4. 12 Código Maltab para el guiado del cálculo de la matriz en formato ELLRT 50
Figura 4. 13 Llamada a las funciones que integran la rutina ELLR-T 50
Figura 4. 14 Llamadas que permite realizar el paso 1,2 y 3 con una solo llamada 50
Figura 4. 15 Uso de la función SpMV 51
Figura 4. 16 Preprocesado de la matriz para poder realizar el CG. 52
Figura 4. 17 Uso de la función conjgrad (integra la rutina SpMV para el cálculo cg) 52
Figura 4. 18 Función conjgrad con la integración de la rutina SpMV 53
Figura 4. 19 Código MATLAB para el cálculo de SpMV utilizando MAT-ELLRT 54

Figura 5. 1 Gráfica del cálculo de 100 iteraciones con SpMV en tiempo de ejecución en segundos CPU con Intel Xeon (4 cores) E5640 y GPU con Tesla C2050 (488 cores) 58
Figura 5. 2 Gráfica de ganancias de MAT-ELLRT frente al SpMV de MATLAB ejecutadas en una GPU Tesla C2050 (488 cores) y una CPU Intel Xeon (4 cores) E5640 59
Figura 5. 3 Modificación de la matriz para la utilización de CG 60
Figura 5. 4 Gráfica de 100 iteraciones de CG en segundos en CPU con Intel Xeon (4 cores) E5640 y GPU con Tesla C2050 (488 cores) 62

Figura 5. 5 Gráfica de ganancia con 100 iteraciones de CG de MAT-ELLRT frente al SpMV de MATLAB ejecutadas en una GPU Tesla C2050 (488 cores) y una CPU Intel Xeon (4 cores) E5640 63

Índice de Tablas

Tabla 2. 1 Tipos de memoria y características 12
Tabla 2. 2 Ancho de banda de las memorias de la GPU 13

Tabla 5. 1 Tipo y formato de las matrices 56
Tabla 5. 2 Evaluaciones de 100 iteraciones con SpMV sin alterar las matrices 57
Tabla 5. 3 Evaluación de CG con 100 iteraciones 61

1. Introducción

Este proyecto tiene como objetivo general ofrecer a los investigadores una herramienta que les permita simultáneamente: (1) explotar las unidades de procesamiento gráfico (GPU) como arquitecturas alto rendimiento para acelerar operaciones matriciales especialmente costosas, centrándonos en las operaciones matriz dispersa vector; y (2) continuar expresando sus modelos matemáticos en entornos de uso extendido como MATLAB [1], que ofrece una extensa colección de librerías y recursos que permiten al científico expresar sus modelos con un lenguaje compacto de muy alto nivel.

MATLAB es un entorno de programación interactivo de alto nivel que permite expresar modelos matemáticos basados en la solución de problemas numéricos sin necesidad de escribir un programa en un lenguaje de más bajo nivel como Java, C, C+ + o Fortran. Por tanto, la productividad en el desarrollo de algoritmos es más alta con MATLAB frente a lenguajes de programación tradicionales de más bajo nivel, ya que no necesita la declaración de variables y asignación de memoria y además dispone de un amplio conjunto de funciones matemáticas. Sin embargo, este entorno en su versión estándar no dispone de recursos que permitan la explotación de arquitecturas de alto rendimiento como las GPUs sin un esfuerzo adicional de desarrollo, o sin disponer de un conjunto de recursos adicionales (‘toolbox”) que ofrecen funciones específicas desarrolladas para explotar distintos tipos de arquitecturas de alto rendimiento tales como multicore, GPU o clúster. Jacket es un ejemplo de recurso complementario para MATLAB que ofrece una gran variedad de funciones que son aceleradas en arquitecturas GPUs y/o multicore [2].

En este proyecto nuestro interés se centra en la operación producto matriz dispersa vector, por ser una operación clave en la implementación computacional de una gran variedad de modelos científicos y de ingeniería. El algoritmo de búsqueda en el que se basa el buscador de Google, (Pagerank [3]), simuladores basados en la solución numérica de PDEs [4] o algoritmos de reconstrucción tomográfica [5] son algunos ejemplos de aplicaciones en los que este tipo de operación está involucrada. Por tanto, es de gran interés disponer de implementaciones de alto rendimiento que se puedan integrar en la implementación de los modelos con facilidad.

Por otra parte, Compute Unified Device Architecture (CUDA) es el interface de programación desarrollado por NVIDIA para explotar las GPUs [6]. El desarrollo de códigos basados en este tipo de interfaces requiere de un esfuerzo de desarrollo muy superior al requerido en entornos de más alto nivel como MATLAB. Existen diversas rutinas de uso libre basadas en CUDA que implementan el producto matriz dispersa vector que pueden se integradas en códigos C [7, 8, 9,10]. La rutina ELLR-T [4] ha mostrado superar en rendimiento al resto de alternativas y además dispone de un recurso que permite configurar esta rutina de forma óptima de acuerdo con la matriz y la arquitectura GPU implicadas en el procesamiento de este tipo de operaciones [7]. Por este motivo, en el proyecto se plantea la integración de la rutina ELLR-T en el entorno MATLAB de forma transparente al usuario, sin necesidad de establecer los parámetros de configuración de la rutina, ni de conocer el modelo de programación CUDA.

Las claves para llevar a cabo la integración de ELLR-T en MATLAB son dos. Por una parte, la definición de ficheros MEX [11], que representan el interface de comunicación entre el entorno de MATLAB y código C, y por otra parte, la definición de arrays persistentes en la memoria de la GPU. Este recurso evita la penalización en la GPU de la copia de estructuras de datos persistentes cuando se llama la misma rutina varias veces. De esta forma es innecesaria la copia de datos que han sido previamente alocados y copiados en la memoria de la GPU. De este modo, sólo los datos que se han actualizado se vuelven a copiar en dicha memoria.

Como resultado final de este proyecto se generará una librería que denominamos MATELLR-T que permita realizar la operación SpMV explotando las GPUs en base a la rutina ELLR-T basada en CUDA que podrá ser integrada en un código MATLAB.

Además, se incluirá una guía completa de fácil uso de la librería implementada con el objetivo de que el usuario final, con un mínimo esfuerzo, pueda acelerar de forma sustancial todos los procedimientos donde la rutina MATELLR-T intervenga. Esta documentación será de especial interés en la actividad de los miembros del grupo Supercomputación-Algoritmos. En este proyecto nos proponemos explorar la integración total de rutinas CUDA con MATLAB.

Para desarrollar los objetivos de este proyecto se han completado las siguientes tareas:

1. Estudio del estado del arte de los métodos y librerías que resuelven las operaciones matriciales de tipo disperso.
2. Implementación de las rutinas necesarias para establecer la conexión ente CUDA y MATLAB del cálculo de operaciones matriciales de tipo disperso.
3. Integración de la rutina ELLR-T como rutina MATLAB.
4. Evaluación comparativa del rendimiento de la rutina nativa de MATLAB y de MATELLR-T.

Para el desarrollo de los objetivos de este proyecto se han utilizado principalmente los siguientes recursos tanto hardware como software:

- Plataforma de desarrollo:
  - La plataforma utilizada para la interface es MATLAB 2011
  - Plataforma Renoir,
    - 1 Intel Xeon Quad-core E5450, 3,00 GHz, 12 MB caché L2, RAM 8 GB DDR3 1.333 MHz
    - 1 GPU NVIDIA GTX 285. (240 cores)
    - Distribución de linux x64, CUDA SDK 2.3.
  - Plataforma DaVinci (Plataforma de evaluación) ,
    - 2 Intel Xeon (4 cores) E5640 2.67 GHz y 16 GB de RAM
    - 1 GPU Tesla C2050 (488 cores).
    - CUDA (Driver + Toolkit + SDK) 4.2.9
- Interfaz de programación
  - Compilador nvcc para los archivos MEX.
  - Editor de textos gedit, de Unix
  - Sistema operativo Fedora 7

2. GPU computing

El término GPU Computing hace referencia al uso de la GPU (unidad de procesamiento gráfico) junto con una CPU como acelerador de operaciones de cálculo científico o técnico de propósito general.

El cálculo en la GPU ofrece un rendimiento computacional sin igual al descargar en la GPU las partes de la aplicación que requieren gran capacidad computacional, mientras que el resto del código sigue ejecutándose en la CPU. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido.

Una plataforma computacional compuesta por CPU y GPU constituye una potente combinación porque la CPU está formada por varios núcleos optimizados para el procesamiento en serie, mientas que la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el rendimiento en paralelo. Las partes en serie del código se ejecutan en la CPU mientras que las paralelas se ejecutan en la GPU.

Figura 2. 1 Distribución del procesador CPU 4 cores y GPU 8SM

En la Figura 2.1: se pueden observar las localizaciones de las distintas unidades funcionales que constituyen una CPU y una GPU, respectivamente [18]. El diseño de una CPU está optimizado para aumentar el rendimiento de código secuencial haciendo uso de una sofisticada lógica de control que permite a las instrucciones de un hilo de programa ejecutarse en paralelo, o incluso fuera de orden, manteniendo la apariencia de una ejecución secuencial. Se disponen de memorias caché de gran tamaño con el fin de reducir la latencia en el acceso a los datos. Las nuevas generaciones de CPUs contienen cuatro, ocho o incluso varias decenas de núcleos de procesamiento diseñados para aumentar el rendimiento al atender simultáneamente a varias aplicaciones secuenciales, o aplicaciones paralelas cuyas ramas de ejecución pueden seguir diversas tazas.

En cambio, el diseño de la GPU se basa en disponer de decenas de unidades de procesamiento, organizadas en conjuntos denominados Streaming Multiprocessors, capaces de ejecutar decenas de *threads* que ejecuten la misma instrucción sobre datos

distintos, pero con la misma traza de ejecución por disponer de pequeñas unidades de control que no son capaces de gestionar simultáneamente distintas trazas de cada núcleo. Esta característica se puede apreciar en la figura ya que la mayor parte del chip está constituida por unidades de cálculo, en menor proporción se incluyen unidades de memoria y una pequeña fracción corresponde a circuitería de control.

Otro aspecto importante es el ancho de banda. Las GPUs tienen un ancho de banda muy superior a las CPUs, esto es debido a que las CPUs son procesadores de propósito general que tienen que satisfacer los requerimientos de diversos tipos de aplicaciones, sistemas operativos y dispositivos de entrada/salida, por lo que disponen de menor espacio para poder realizar cálculo y de ahí que su rendimiento para cálculo sea menor. Mientras que las GPUs destina la mayor parte del espacio a unidades de cálculo.

El aumento de la potencia de la GPU actualmente va ligado a la industria del video juego donde necesitan una gran capacidad de cómputo porque deben ser capaces de realizar operaciones en coma flotante de forma masiva. Este es uno de los motivos por los que la GPU destina más espacio a las unidades de cómputo y menos a lógica de control. Necesitan menor tamaño de lógica de control porque integra un sistema de gestión que controla qué datos están listos para su uso y cuáles no.

Por eso desde hace años el modelo computacional CPU-GPU está muy extendido en el ámbito científico, porque, aprovecha el control lógico de la CPU y la potencia de cálculo de la GPU. En la industria de las videoconsolas también han apostado por este modelo computacional, de este modo, la nueva generación de videoconsolas PS4 [19] y XboxOne [20] tienen una arquitectura híbrida en la que disponen de varias CPUs y varias GPUs donde reducen la latencia CPU-GPU con memorias compartidas muy rápidas.

2.1 Modelo computacional

En la Figura 2.2 se puede observar el modelo computacional de una aplicación CPU - GPU se basa que la GPU ejecute la parte paralela del código, actuando como coprocesador de la CPU.

En este contexto, la CPU se denomina *host* y es la que dirige el hilo de ejecución secuencial de la aplicación y la GPU se denomina *device*. El código paralelo se desarrolla en unidades separadas llamadas *kernels*. El código de la CPU sigue una ejecución secuencial mientras realiza llamadas a los *kernels* alojados en el *device* que se ejecutan en paralelo y de forma asíncrona en la GPU (a no ser que se establezcan barreras de sincronización).

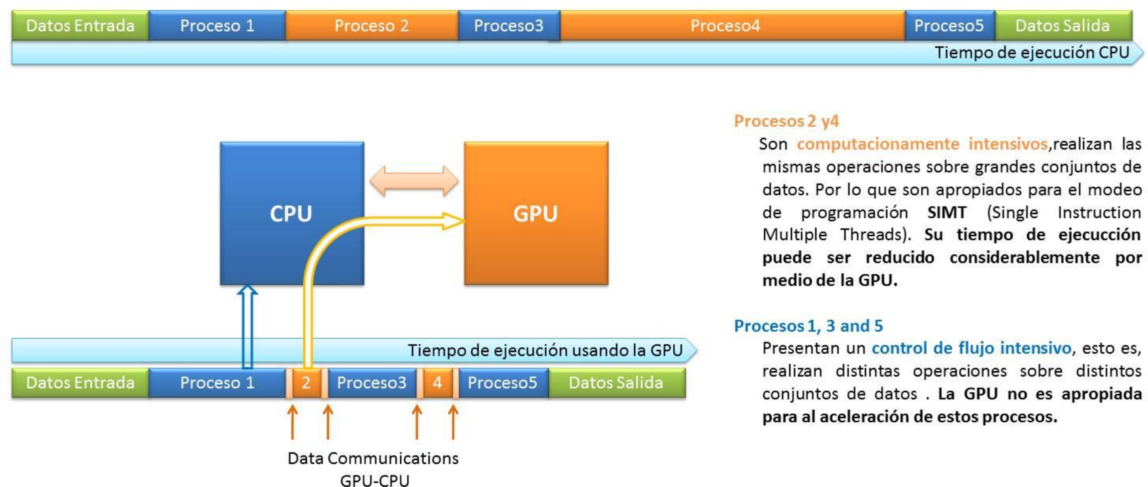


Figura 2. 2 Modelo computacional CPU vs GPU

Los *kernels* están implementados utilizando el lenguaje de programación CUDA, que contiene instrucciones que pueden ser ejecutadas en el *host*, en la GPU o en ambos. Dichas instrucciones se usan para la sincronización, alojamiento y liberación de memoria, copia de estructuras entre la memoria de la CPU y GPU y cálculo de operaciones.

La ejecución de un programa CUDA se basa en la ejecución simultánea de un conjunto de *thread* organizados en unidades unidimensionales, bidimensionales o tridimensionales llamadas bloques o *thread* blocks. Estos bloques a su vez, se organizan en otras unidades mayores (igualmente de una, dos o tres dimensiones) denominadas grid. Durante la ejecución, cada bloque se divide en conjuntos de 32 *threads* denominados *warps*, de tal forma que la arquitectura puede conseguir en 4 ciclos de reloj (8 cores × 4) que los 32 *threads* completen la misma operación sobre distintos operandos. Este modelo computacional se denomina Single Instruction Multiple *Thread* (SIMT) y es una extensión al modelo computacional vectorial Single Instruction, Multiple Data (SIMD).

Una vez calculados los datos del grid, se realiza la distribución entre todos los SMs (streaming multiprocessor), de tal forma que un bloque completo, solo se ejecuta en un

determinado SM. El SM crea, maneja y ejecuta los *threads* de forma concurrente sin sobrecarga, porque no hay sobrecarga en el cambio de contexto entre *warps*.

El programador debe tener en cuenta varias características arquitectónicas, como la topología de los multiprocesadores y la gestión de la jerarquía de memoria. La arquitectura permite a la GPU a emisión de una serie de invocaciones a *kernels.* Cada *kernel* se ejecuta como una cola de bloques de hilos organizados como una red de bloques. Cada bloque se le asigna a cada SM. Todos los hilos que pertenecen a un *warp* ejecutan el mismo conjunto de instrucciones con diferentes datos. El tamaño de cada bloque es definido por el programador. El rendimiento máximo se obtiene cuando todos los hilos del *warp* ejecutan la misma secuencia de instrucciones, ya que cualquier instrucción de control de flujo puede causar que los hilos sigan caminos diferentes en su ejecución. Si esto ocurre, se secuencializará la actuación de los hilos del *warp* y el tiempo de ejecución aumentará [35, 53].

A continuación se describen los aspectos que el programador debe tener en cuenta para el desarrollo de código CUDA.

2.1.1 Topología del GRID

Para el desarrollo de un *kernel* es importante decidir la topología que van a tener los *grids* y bloques, intentándolo hacer que exista una correspondencia con la propia topología de los datos que se van a manejar. Una vez diseñadas tales estructuras, es necesario saber la identificación de cada *threads* dentro del bloque y del *grid.*

2.1.2 Divergencia

Todos los *threads* que componen un *warp* comienzan su ejecución en la misma traza de programa, pero pueden realizar saltos condicionales y seguir distintas trazas de ejecución. Por lo tanto, si los *threads* de un *warp* ejecutan conjuntos de instrucciones con trazas distintas estas se serializan, ejecutando en cada instante una traza determinada y parando los *threads* que no siguen esa traza, bajando significativamente el rendimiento global [48]. Sin embargo, se consigue la mayor eficiencia cuando todos los *threads* del *warp* realzaran la misma instrucción a la vez. El programador ha de intentar minimizar las situaciones de divergencia del *warps.*

2.1.3 Ocupación

Los rendimientos teóricos de las GPU son elevadísimos, pero en el uso cotidiano no rinden cómo deberían. Esto se debe a que existe una alta latencia a la hora de acceder a la memoria global. La métrica que mide el número de *warps* que pueden estar activos en un SM se llama ***ocupación*** y determina con qué efectividad el hardware se mantiene ocupado. La ocupación es el ratio entre el número de *warps* activos por SM y el número máximo de éstos. Una alta ocupación siempre interfiere directamente con la cantidad de registros y de memoria *shared* que necesita un bloque para ejecutarse. Cuantos menos recursos se necesiten, más *warps* podrán estar activos y más alto porcentaje de ocupación se obtendrá. La ocupación liga el número de recursos que necesita el *kernel* con los recursos disponibles de la GPU, que vienen determinados por su capacidad de cómputo.

2.1.4 Gestión de la memoria

La GPU dispone de una jerarquía de memoria en la que intervienen los siguientes tipos de dispositivos de almacenamiento, en la tabla 2.1 se puede observar las características de cada memoria:

Memoria	Localización	Caché	Acceso	Alcance
Registros	On-chip	No	lectura/escritura	Un *Thread*
Shared	On-chip	N/A	lectura/escritura	Todos los *Thread* en un bloque
Global	Off-chip	Yes	lectura/escritura	Todos los *threads*+ host
Constantes	Off-chip	Yes	lectura	Todos los *threads*+ host
Texturas	Off-chip	Yes	lectura/escritura	Todos los *threads*+ host

Tabla 2. 1 Tipos de memoria y características

- **Registros:** Es una memoria (on-chip), su acceso es el más rápido de la GPU. Es la memoria con mayor ancho de banda y baja latencia de la GPU. Generalmente, el acceso a un registro no genera ningún ciclo de reloj extra, pero puede ocurrir un retardo debido a las dependencias de lectura después de escritura y a conflictos en el acceso a dicha memoria.
- **Memoria shared:** Es una memoria (on-chip) con un tamaño máximo de 64 Kb, su acceso es más rápido que la memoria global y si no existe conflictos en el acceso a un registro es tan rápido como el acceso a registro. Para su mayor aprovechamiento se divide en bancos de memoria para que todos puedan acceder a él. Una desventaja de la memoria shared es que cuando existen conflictos se penaliza tanto los accesos a registros como de bancos de memoria. Por eso es importante tener en la memoria shared todos los datos necesarios para realizar el cómputo y reducir los accesos a la memoria global lo máximo posible, ya que penaliza considerablemente el rendimiento global de la aplicación.
- **Memoria de constantes**: Es una memoria caché de solo lectura (off-chip) con un tamaño máximo de 64 Kb, en la que mientras no exista fallo de caché, su coste de acceso a los datos es mínimo.
- **Memoria de texturas**: Es una memoria caché de solo lectura (off-chip) con un tamaño de 8Kb por SM, cuando ocurre un fallo se obtiene de la memoria global, con la latencia pertinente. Es un tipo de memoria que se utiliza cuando existe una reutilización de datos, su uso mejora el rendimiento comparado con leer los datos desde memoria global.
- **Memoria global:** Es una memoria (off-chip) mucho más lenta y grande que la memoria shared ya que se encuentra fuera del chip y por lo tanto es mucho más importante realizar peticiones de forma eficiente. Por eso es importante centrarse en la reutilización de datos dentro del SM y caches para evitar las limitaciones del ancho de banda de la memoria. Desde la perspectiva del desarrollador el acceso a memoria global necesita que sea coalescente, esto significa combinar las solicitudes de memoria de los *threads* dentro una única operación de memoria.

  El mayor rendimiento se produce cuando la dirección de memoria está alineada y un *warp* accede a todos los datos dentro de una región contigua, lo que significa que la

transacción tiene una eficiencia del 100% ya que cada byte recuperado de la memoria es utilizado.

La actualización de los datos a la memoria global ocurre por la invalidación de la cache L1 o cuando se está escribiendo en la cache L2. Solo cuando es desalojado se actualizan los datos de la cache L2 en la memoria global.

La memoria global fue diseñada para transmitir rápidamente bloques de memoria de datos al SM. Por ese motivo esta memoria es la encargada de almacenar la persistencia de las matrices en formato ELLR-T evitando así la penalización continua de la transmisión de la matriz entre la memoria del host y la memoria global. La limitación de la memoria global depende de la arquitectura de la GPU. Hoy día el espacio de memoria global para GPU de HPC suele ser superior a 6Gb.

Las instrucciones de memoria son operaciones en las que se lee/escribe en memoria global, shared o local. Debido a la latencia entre la memoria global y la shared, lo ideal es siempre mover desde la memoria global hasta la memoria shared la información con la que se va a operar. Pero antes es necesario que los datos se transfieran a la memoria global desde la memoria del host (memoria Mapped). Se puede observar en la tabla 2.2 la diferencia de velocidad entre los distintos anchos de banda de las distintas memorias.

Memoria de registro	8000 GB/s
Memoria Shared	1600 GB/s
Memoria global	177 GB/s
RAM del Host	8 GB/s

Tabla 2. 2 Ancho de banda de las memorias de la GPU

La mayor reducción de ancho de banda se encuentra en los accesos de la GPU a la memoria del host. Esto hace que la operación **cudaMemcpy** sea la que mayor coste tiene y la que origina un mayor consumo del tiempo de ejecución. Este es uno de los principales motivos por los que la operación SpMV en MAT-ELLRT se hace con persistencia de datos (Sección 4), para que solo penalice una vez el alojamiento de los datos a la memoria global de la GPU.

2.2 Arquitecturas GPU

La arquitectura de una GPU se compone de una serie de unidades de procesamiento denominadas *streaming multiprocessor* (SM) o multiprocesadores, cada SM contiene al menos 8 procesadores escalares segmentados (SP) o cores que comparten la lógica de control y caché de instrucciones. El número de SMs y de SPs varía en cada generación de arquitectura. Cada SM contiene:

- Un conjunto de registros de 32 bits por SP
- Un espacio de memoria de lectura/escritura compartida entre todos los SPs denominada *shared*, mencionada en el apartado 2.1.4.
- Dos áreas de memoria de sólo lectura (memoria de constantes y memoria de texturas) compartidas entre todos los SPs de todos los SMs.

A continuación, veremos las diferencias entre las últimas arquitecturas de GPUs para HPC.

2.2.1 Arquitectura Fermi

Figura 2. 3 Arquitectura Fermi

La arquitectura Fermi [49] contiene 32 Scalar proccesor (SPs) en cada SM, para modelos son 16SMs haciendo un total de 512 cores con el estándar IEE 754-2008. Contiene memoria ECC, para la tolerancia de datos en la GPU.

Se puede observar en la Figura 2.3 el diseño de la arquitectura Fermi con 16 SMs, divididos en 4 bloques, cada bloque contiene 4 SMs y una unidad de rasterización. Cada SM consta de una caché de instrucciones, un *warps scheduler,* un conjunto de registros para todos los cores, también dispone de unidades de carga y almacenamiento de datos, unidades de operaciones especiales (raíces, operaciones exponenciales, trigonométricas…), unidades de tratamiento de texturas.

Todos los SMs acceden a una memoria caché L2 que permite compartir los datos entre los demás SMs, además, dispone de unidades de control y corrección de fallos.

La arquitectura Fermi dispone también de un sistema Nvidia Giga*thread*, que es capaz de atender concurrentemente a varios los *kernels* gracias a que integra un administrador de *kernel* inteligente que intenta hacer concurrente para que no haya tareas sin ejecutar existiendo recursos para ellas.

2.2.2 Arquitectura Kepler

La arquitectura Kepler [17] es la arquitectura más actual del mercado de la supercomputación. Proporciona una mayor velocidad de procesamiento y eficiencia gracias a un SM nuevo llamado SMX.

Figura 2. 4 Arquitectura kepler

En la Figura 2.4 se puede observar la distribución de la arquitectura Kepler, donde tienen numerosos SMX, el diseño es similar a la arquitectura Fermi, cada SMX consta de una caché de instrucciones, un *warps scheduler,* un conjunto de registros para todos los cores, también dispone de unidades de carga de datos y almacenamiento de datos, unidades de operaciones especiales (raíces, operaciones exponenciales, trigonométricas…), unidades de tratamiento de texturas. La diferencia básica con respecto a la arquitectura Fermi son los SMs con los SMX de Kepler, que dedica más espacio más unidades de cálculo que a lógica de control Figura 2.5.

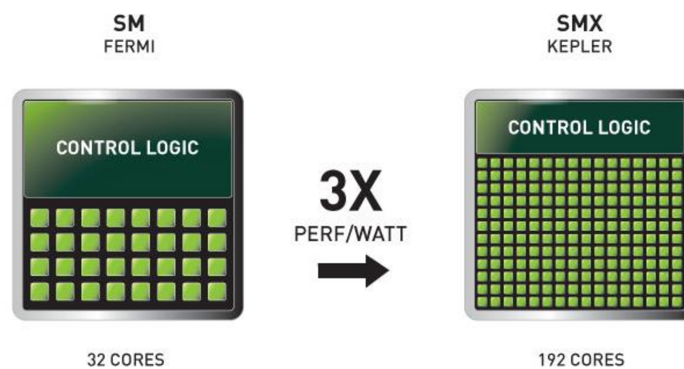


Figura 2. 5 Diferencia entre Fermi y Kepler

Su actual configuración incrementa el rendimiento de aceleración de bucles anidados paralelos. Esto significa que una GPU puede iniciar nuevos subprocesos de forma dinámica por sí misma, sin necesidad de volver a la CPU.

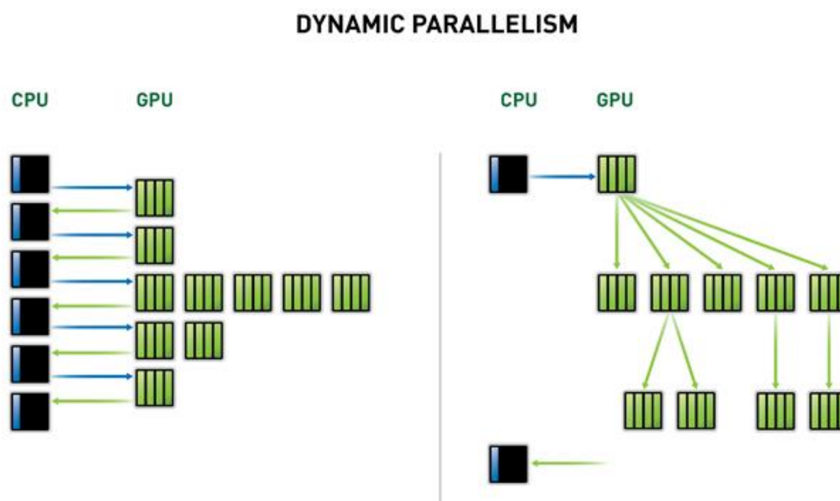


Figura 2. 6 GPU dinámica

En la figura 2.6 se puede observar que varios núcleos de la CPU utilizan la misma GPU Kepler (con programación MPI), lo que mejora drásticamente la programabilidad y la

eficiencia. También simplifica la programación en la GPU ya que facilita la aceleración de bucles anidados paralelos, lo que significa que una GPU puede iniciar nuevos subprocesos de forma dinámica por sí misma, sin necesidad de volver a la CPU Figura 2.6

2.3 CUDA como interfaz de programación

En primera instancia, la programación de procesadores gráficos fue complicada debido a la falta de entornos de desarrollo para que los programadores pudiesen implementar aplicaciones numéricas. Los lenguajes de programación usados eran OpenGL [21] y Direct3D [22]. Posteriormente, surge CUDA una API de computación paralela de propósito general para las GPUs de NVDIA [23,24]. Incluye un modelo de programación paralelo y un conjunto de instrucciones que aprovechan la capacidad computacional paralela de la GPUs de NVIDIA. CUDA pone a disposición de los desarrolladores y programadores un entorno de programación de alto nivel como C, sin necesidad de usar APIs gráficas.

Existen numerosos lenguajes para poder explotar el máximo rendimiento de las GPUs, tales como CUDA C, OpenCL, etc. (véase Figura 2.7). Todas estas plataformas de desarrollo tienen un lenguaje de programación de alto nivel, donde es posible explotar la GPUs de distintos fabricantes y modelos.

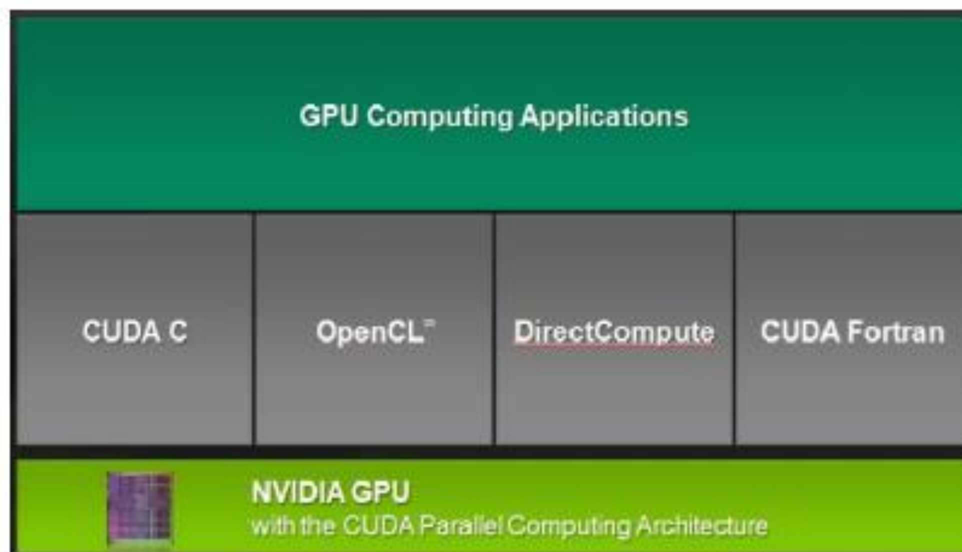


Figura 2. 7 Interfaces de programación para la GPU

En el mercado existe una amplia diversidad de plataformas de arquitectura HPC basadas en tarjetas gráficas, NVIDIA [25] ATI [50] e INTEL [51] que facilitan a los desarrolladores entornos de programación adaptados para sus propias arquitecturas.

El lenguaje en el que se ha desarrollado este proyecto es CUDA, un lenguaje diseñado por NVIDIA para conseguir aplicaciones que se ejecuten de forma paralela y que sean capaces, de forma transparente, de obtener una buena escalabilidad dependiendo la arquitectura de la GPU fabricada por NVIDIA y la correcta configuración de parámetros necesarios para su explotación. Cabe destacar, que actualmente CUDA junto con las

GPUs de NVIDIA son las plataformas de GPU computing más extendidas en HPC de propósito general.

2.3.1 Estructura de un programa CUDA

La estructura del programa CUDA está relacionada con la arquitectura y el modelo computacional que se han descrito en las secciones previas. El programador prepara todos los datos que necesitará la GPU, debe inicializar la GPU, realizar las reservas de memoria en el *host* y el *device*, realizar si es oportuno un volcado de memoria del *host* al *device* o del *device* al *host.* Una correcta estructuración del programa evita funcionamientos incorrectos por problemas de memoria o rendimiento. Es importante que el programador conozca su arquitectura para poder sacar el máximo rendimiento de la GPU.

Un aspecto a tener en cuenta es que el *kernel* es invocado desde un lenguaje de alto nivel. Eso implica que el programador debe asegurarse que la integridad de los datos no se altere durante todo el proceso. Para el paso de parámetros y variables entre el *host* y el *device* existen numerosas funciones. En la figura 2.8 aparece un ejemplo básico de una multiplicación en la GPU con sus correspondientes llamadas.

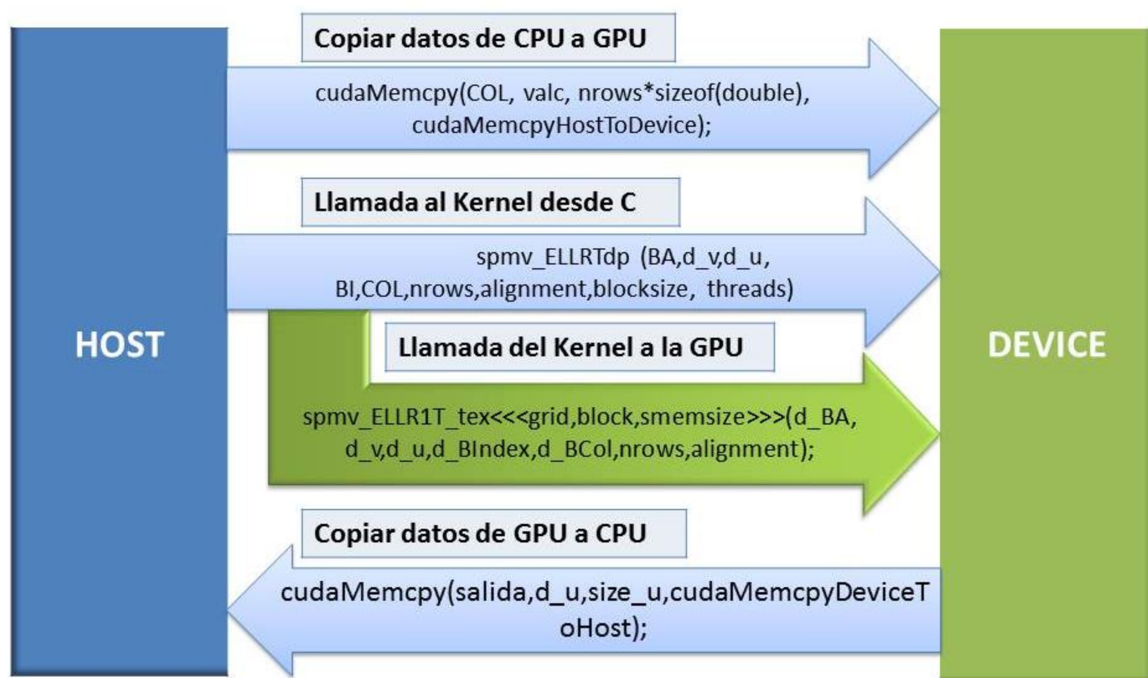


Figura 2. 8 Llamadas del *host* al *device*

En la Figura 2.8 se pueden observar las distintas llamadas que existen entre el *Host* (CPU) y el *Device* (GPU) y los datos que intervienen en la computación acelerada en la GPU. Las reservas de memoria en el *device* deben realizarse previamente en el *host* antes del envío de los datos ya que pasarán a estar alojados el *device.* Todo dato que se genere en la GPU será almacenado en la GPU por lo que será necesario realizar su reserva de memoria para para su utilización posterior en la GPU o su envío al *host* para la continuación de la ejecución del programa. CUDA ofrece las rutinas necesarias para

implementar las copias de memoria entre la RAM de la CPU (*host*) y la memoria global de la GPU (*device*). La Figura 2.8 ilustra la estructura de un código CUDA que se ejecuta en el *host* y que lanza la ejecución de un *kernel* en el *device* junto con las correspondientes copias de datos entre las memorias. Más concretamente se pueden distinguir las siguientes llamadas a rutinas específicas de CUDA:

- **Copiar datos de CPU a GPU:** Esta llamada se realiza con la función,

  cudaMemcpy(destino,origen,tamaño,cudaMemcpyHostToDevice);

  donde la variable **destino** es el nombre de la variable del *device* en la que se realizará la copia desde el *host*, **origen** es el nombre de la variable del *host* a realizar la copia a la variable del *device,* **tamaño** indica el tamaño total del array y **cudaMemcpyHostToDevice** es el nombre de la rutina CUDA que permite realizar una copia de una variable del *host* al *device.*

- **Llamada al *kernel* desde C:** Consiste en la invocación del *kernel* desde un programa escrito en C indicándole los parámetros necesarios para utilizar la GPU (número de *thread*, tamaño del bloque y los vectores a utilizar). Ésta función realiza la llamada del *kernel* al *device* donde le indica el grid , tamaño del bloque y tamaño de la memoria a usar (grid,blocks,smemsize).

- **Copiar datos de GPU a CPU:** Esta llamada se realiza con la función,

  cudaMemcpy(destino,origen,tamaño,cudaMemcpyDeviceToHost);

  donde la variable **destino** es el nombre de la variable que es copiada del *device* al *host*, **origen** es el nombre de la variable del *device* a realizar la copia a la variable del *host,* **tamaño** indica el tamaño total del array y **cudaMemcpyDeviceToHost)** es el nombre de la rutina CUDA que permite realizar una copia de una variable del *device* al *host* .

Desde el punto de vista del programador un *kernel* es ejecutado por un conjunto de *threads* que depende de la dimensión de las estructuras de datos involucradas en la computación. De forma automática gracias a la intervención de CUDA, la arquitectura atiende al conjunto de *threads* organizándolos en un grid de bloques, cada uno de los bloques está estructurado en conjuntos de 32 *threads* (*warps*). Cada bloque se ejecuta en un SM de manera que cada SM ejecuta de forma concurrente varios *warps*, de acuerdo con los recursos de memoria consumidos por los mismos.

Aunque este proceso de mapeo de *threads* en la arquitectura de la GPU está automatizado por CUDA, el programador debe definir algunos de los parámetros que definen la estructuración de los *threads* en un grid de bloques. La Figura 2.9 representa esta estructuración en un grid de bloques bidimensional donde cada caja representa un bloque y cada flecha del bloque corresponde con un *thread* que es identificado por una pareja de

índices. El programador puede establecer el número de dimensiones en función de las estructuras de datos que se procesen.

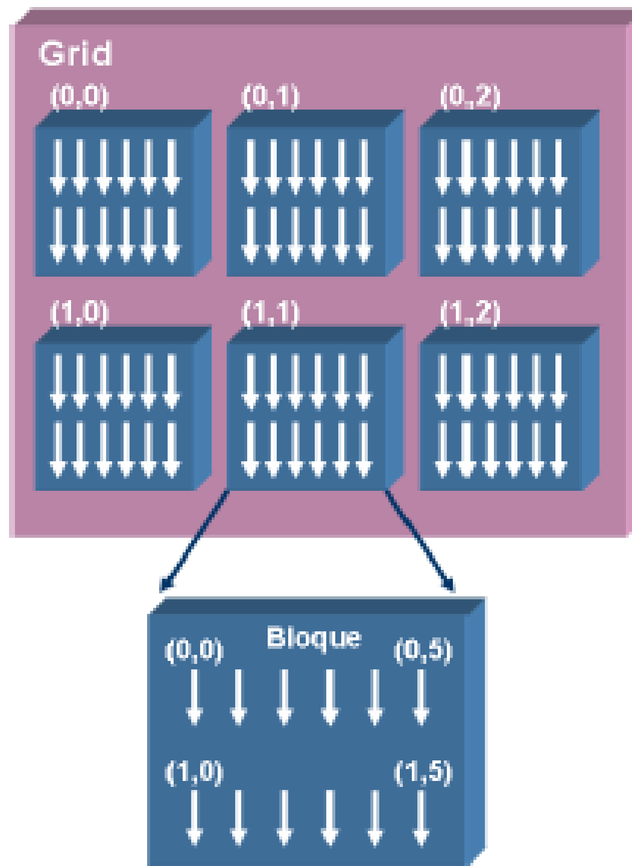


Figura 2. 9 Arquitectura de un Grid

De esta forma en la sintaxis de CUDA se establecen los siguientes identificadores asociados con la estructuración de los *threads*:

Identificadores

Índice (x,y,z) del thread dentro del bloque	int threadIdx.x, threadIdx.y, threadIdx.z
Índice (x, y, z) del bloque dentro del grid	int blockIdx.x, blockIdx.y, blockIdx.z
Tamaño (x, y, z) del bloque	int blockDim.x, blockDim.y, blockDim.z

Figura 2. 10 Identificadores de los *threads*

En la figura 2.10 se observa un ejemplo de la obtención del índice de un *threads* en un array tridimensional.

Con la combinación de estos indicadores se puede identificar un *thread* que está dentro de un bloque y de un grid, lo que equivale a que se obtiene el identificador global del *thread* en una de las dimensiones x.

Int idx =blockIdx.x* blockDim.x+ threadIdx.x;

En la figura 2.11 se observa de forma ilustrativa la comparación entre el código CPU y el código GPU. Mientras que en el programa escrito en C se realiza el incremento de la variable sin preocuparse de cómo se ejecuta, en el programa CUDA es necesario que indicar el grid y tamaño de bloque así como tener identificado en todo momento el *thread* dentro del bloque en el que se ejecuta.

Programa CPU

```
Void increment_cpu(float*a, floatb, intN)
{
        for (intidx = 0; idx<N; idx++)
          a[idx] = a[idx] + b;
}


Void main()
{
.....
        increment_cpu(a, b, N);
}
```

Programa CUDA

```
__global__ void increment_gpu(float*a, float b,
int N)
{
  intidx =blockIdx.x* blockDim.x+ threadIdx.x;
   if (idx < N)
    a[idx] = a[idx] + b;
}
Void main()
{
.....
Dim3 dimBlock (blocksize);
Dim3 dimGrid( ceil( N / (float) blocksize) );
increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Figura 2. 11 Diferencia entre código C y CUDA

En la figura 2.11 se puede observar que la sintaxis CUDA para la configuración y utilización de la GPU es sencilla y versátil ya que permite el cambio de diversos parámetros (tamaño del grid y tamaño del bloque). Estos datos solo se indican una vez a la hora de la utilización de la GPU. También se puede observar que el lenguaje en CUDA es similar a lenguajes como C o C++, por lo que, para los programadores es más sencillo.

En la rutina SpMV la dimensión del grid es unidimensional debido al formato ELLRT que almacena los datos en arrays unidimensionales por lo que la distribución de los bloques también es unidimensional. El máximo que permite la arquitectura actual son dimensiones de grid y bloques tridimensionales.

2.4 Integrando GPU computing (MEX-file)

CUDA es un interface de programación que facilita la programación en GPU en el contexto de aplicaciones generales. Como se ha analizado en la sección anterior, está diseñado para facilitar la adaptación de programas expresados en C al entorno de ejecución paralelo de la GPU. Por tanto, en principio un proceso debe ser definido a nivel de lenguaje C para que se pueda acelerar en una GPU.

Sin embargo, existen multitud de aplicaciones implementadas en entornos de desarrollo de un nivel de abstracción superior y que demandan una elevada cantidad de recursos computacionales que pueden beneficiarse de la explotación de las GPUs. MATLAB [26] es uno de estos entornos de trabajo que está más extendido entre los científicos e ingenieros. Por tanto, es de gran interés disponer en MATLAB de rutinas capaces de explotar las GPUs sin que el científico o ingeniero disminuya de nivel de abstracción para expresar sus modelos.

MATLAB es un lenguaje de alto nivel y así como un entorno interactivo para el cálculo numérico, la visualización y la programación

MATLAB se puede utilizar en una gran variedad de aplicaciones, tales como procesamiento de señales y comunicaciones, procesamiento de imagen y vídeo, sistemas de control, pruebas y medidas, finanzas computacionales y biología computacional. Más de un millón de ingenieros y científicos de la industria y la educación utilizan MATLAB, el lenguaje del cálculo técnico.

Entre sus prestaciones básicas se hallan: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. El paquete MATLAB dispone de dos herramientas adicionales que expanden sus prestaciones, Simulink (plataforma de simulación multidominio) y GUIDE (editor de interfaces de usuario - GUI). Además, se pueden ampliar las capacidades de MATLAB con las cajas de herramientas (toolboxes); y las de Simulink con los paquetes de bloques (blocksets).

Existen varias tools que son capaces de aprovechar la potencia de la GPU para algebra desde el entorno Maltab como Jacket [27] y CuMat [32]. Son recursos independientes integrados con MATLAB de una forma fácil y sencilla para el usuario debido al uso de una nomenclatura y sintaxis similar a la de MATLAB.

Jacket dispone de una librería para desarrolladores llamada ArrayFire para la explotación de la GPU con lenguajes de alto nivel como C, C++ o Fortran. Dispone de dos versiones, una básica y otro profesional, la básica te permiten el acceso a todas las funcionalidades pero sólo con una GPU, la profesional, incorpora funciones de multinúcleo y multiGPU. Jacket incorpora un entorno de desarrollo [28] y un entorno de trabajo capaz de paralelizar de forma sencilla sin conocimientos de la GPU. Sin embargo, actualmente no dispone de rutinas para procesar matrices de tipo disperso en las que se centra este proyecto.

Hasta la fecha, el uso de la GPU para acelerar el cálculo siempre ha ido asociado a plataformas de desarrollo para programadores especializados, no para usuarios sin conocimientos previos de la arquitectura o la programación de GPUs como se ha comentado anteriormente. No obstante, existen estrategias que permiten usar rutinas desarrolladas en lenguajes C, C + +, Fortran, o incluso CUDA en el contexto MATLAB. La estrategia más extendida se basa en la utilización de los denominados ficheros MEX. Esta es la manera de que MATLAB pueda explotar la GPU mediante CUDA.

Por lo tanto, surge un problema ya que CUDA no soporta el lenguaje MATLAB, y en cambio sí podría aprovechar todas las ventajas para el cálculo que ofrece. Es en este punto cuando empiezan a ser necesarios los ficheros MEX [29].

Un archivo MEX proporciona una interfaz entre MATLAB y subrutinas escritas en otros lenguajes de programación. Cuando se compilan los ficheros MEX, se crea una función envoltorio que permite que sean pasados y devueltos tipos de datos de MATLAB, de este modo, desde MATLAB se pueden cargar de forma dinámica y ser invocados códigos en CUDA, C, C++ o Fortran como si fuesen funciones integradas del propio MATLAB. Para apoyar el desarrollo de los archivos MEX, MATLAB ofrece funciones de interfaz que facilitan la transferencia de datos entre archivos MEX y MATLAB. Por tanto, en este proyecto los ficheros MEX son la clave para integrar códigos CUDA en el entorno MATLAB.

Los archivos MEX tienen una estructura estándar que siempre se aconseja completar.

- La rutina debe contener el código implementado del MEX. Dónde se especifica el número de entradas y salida desde la interfaz de MATLAB.

- En la función MEX las estructuras de datos que se conectan en los dos entornos son identificado por prhs, nrhs, plhs, nlhs, donde prhs es un array de entrada de argumentos, nrhs es el número de entradas argumentos, plhs es un array de salida de argumentos, y nlhs es el número de salidas de los argumentos.

En los ficheros MEX es posible el manejo de tipos de datos mxArray, esta estructura es la utilizada para la manipulación de los datos con códigos de más bajo nivel (C, C++, CUDA, Fortran) dentro del fichero MEX con la que se puede manipular estos datos desde un código C.

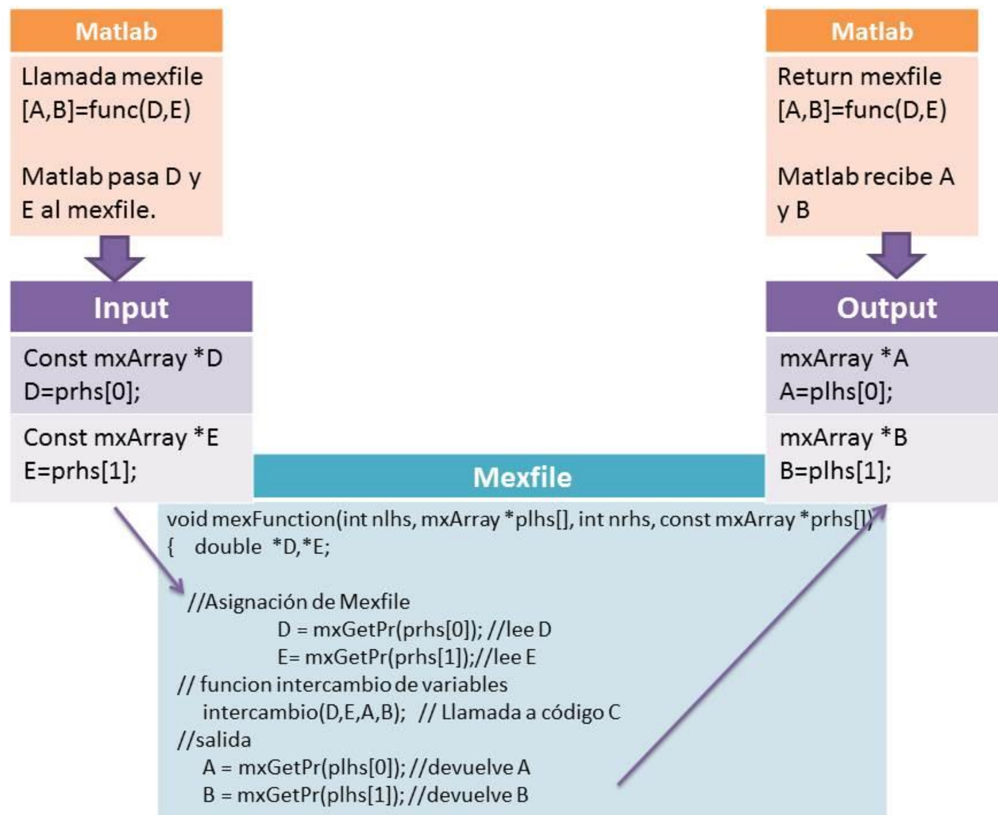


Figura 2. 12 Entradas y salidas de datos a través del MEX

La ventaja principal del uso de los MEX es la reducción de la latencia en el envío y la recepción de datos. La latencia del envío de datos a las rutinas invocadas es mínima en cambio la recepción de los datos es más costosa si son tamaños de datos grandes.

En la figura 2.12 aparece un pequeño ejemplo de paso de parámetros a través de un MEX. Donde la variables de entrada procedentes de MATLAB son D y E que son de tipo mxArray y se convierten en double gracias a la función mxGetPr. Posteriormente se realiza una llamada a una función de intercambio de valores, donde A y B son los array devueltos de tipo double que se deben convertir en tipo mxArray con la función mxGetPr. El formato de entrada/salida de todos los parámetros a MATLAB es de tipo mxArray, por lo que el programador debe utilizar la función que desee para la conversión de datos para su utilización en el código C. La numeración de prhs[0] y prhs[1] indica el orden de los parámetros de entrada desde MATLAB y plhs[0] , plhs[1] indica el orden de los parámetros de salida hacia MATLAB.

3. Algebra dispersa

El producto Matrix-Vector (MV) es una operación clave para una amplia variedad de aplicaciones científicas, como el procesamiento de imágenes, simulación, ingeniería de control, etc [36]. La relevancia de este tipo de operación en las ciencias computacionales se constata por el constante esfuerzo dedicado a optimizar el cálculo de MV en las plataformas computacionales de cada momento, que van desde las primeras computadoras en la década de los setenta a las más modernas arquitecturas multi-núcleo [37], [38], [39], [40] . Como prueba de este esfuerzo, cabe destacar que MV es una rutina de nivel 2 de BLAS (Basic Linear Algebra Subroutines) y esta librería ha sido mejorada y optimizada a medida que han evolucionado las arquitecturas [41], [42], [43].

Para muchas aplicaciones basadas en MV, las matrices involucradas son grandes y dispersas, y están relacionadas con la solución de sistemas lineales, del problema de autovalores o de ecuaciones diferenciales parciales de un amplio espectro de disciplinas científicas y de ingeniería. Para estos problemas, la optimización del producto matriz dispersa vector (SpMV) es un reto debido a la irregularidad de la computación asociada. Esta irregularidad se debe al hecho de que la localidad de acceso a los datos no se mantiene y que el paralelismo de grano fino de los bucles no es explotado [44]. Por lo tanto, es necesario un esfuerzo adicional para que la computación de SpMV explote los distintos niveles de paralelismo que ofrecen los modernos procesadores multicore o las GPUs. Este esfuerzo se centra en el diseño de estructuras de datos para almacenar la matriz dispersa, ya que el rendimiento de SpMV está directamente relacionado con el formato utilizado.

Algunos ejemplos de librerías que incluyen la implementación optimizada de operaciones dispersas en diversas plataformas modernas son MAGMA [30,31], MLIB, MKL [33], CULAsparse[34], SpBLAS [32].

3.1 Almacenamiento por coordenadas (COO)

El esquema de almacenamiento por coordenadas (COO) para comprimir una matriz dispersa es una transformación directa del formato denso. Sea Nz el número total elementos no nulos de la matriz. Una implementación típica del formato COO define tres arrays unidimensionales de dimensión Nz. Uno de ellos de tipo float que almacena los elementos no nulos de la matriz y los otros de tipo entero que almacenan los índices correspondientes de filas y columnas. El rendimiento de SpMV puede deteriorarse debido a que se pierde información sobre el orden de las coordenadas.

3.2 Compressed Row Storage (CRS)

Es el formato más extendido para almacenar las matrices dispersas en procesadores superescalares. La Figura 3.1 muestra los detalles de CRS. Sean N y Nz el número de filas

de la matriz y el número total de entradas no nulas de la matriz, respectivamente; la estructura de datos se compone de los siguientes conjuntos:

1. A [] matriz dimensión Nz de tipo float, que almacena las entradas,
2. J [] matriz de enteros de dimensión Nz, que almacena su índice de columna,
3. Start [] matriz de enteros dimensión de N, que almacena los punteros del principio de cada fila de A [] y j [].

El código para calcular SpMV en base al formato CRS es el siguiente:

Hay varios inconvenientes que dificultan la optimización de la ejecución de este código en arquitecturas superescalares. En primer lugar, la localidad del acceso al vector v [] no se mantiene debido al direccionamiento indirecto. En segundo lugar, el paralelismo de grano fino no es explotado debido a que el número de iteraciones del bucle interno es pequeño y variable [45].

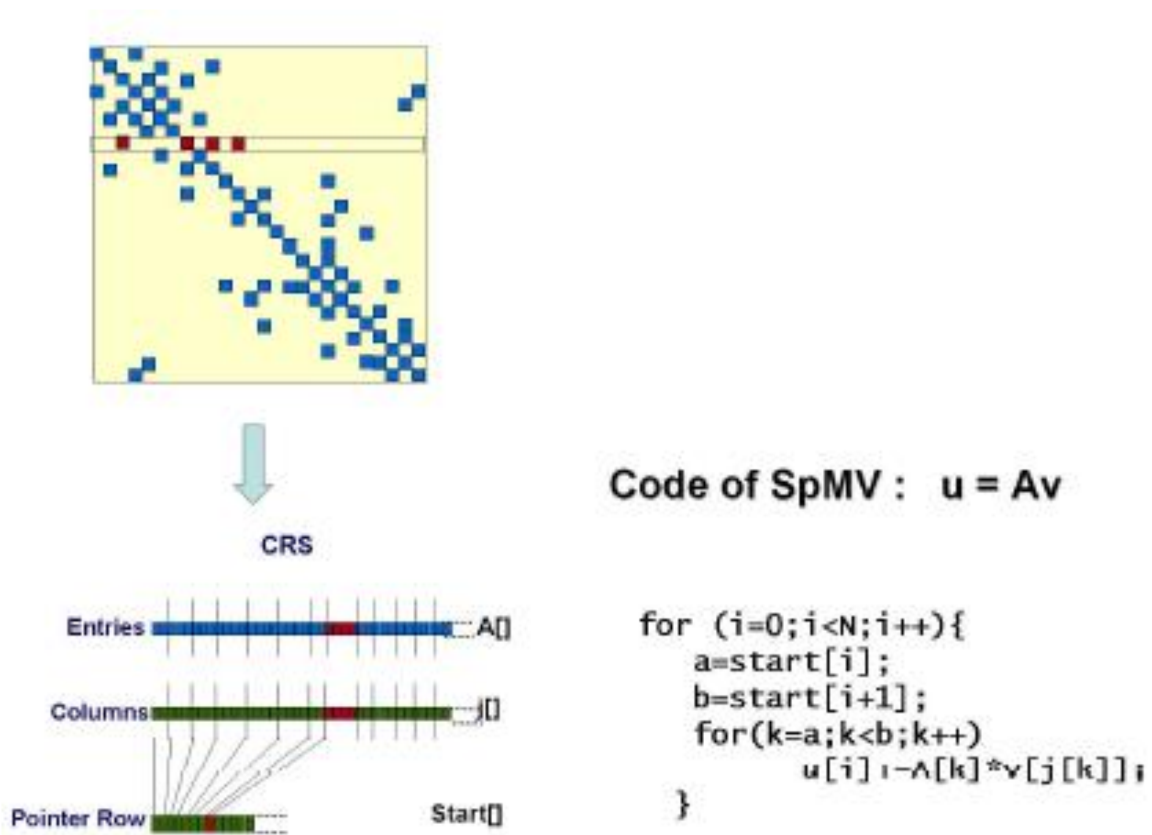


Figura 3. 1 Ilustración del formato CRS

3.3 ELLPACK-R

ELLPACK o ITPACK [46] se presentó como un formato para comprimir la matriz dispersa con el fin de resolver grandes sistemas lineales con las subrutinas ITPACKV en

arquitecturas vectoriales. Este formato almacena la matriz dispersa en dos matrices, una de tipo float, para guardar las entradas, y otra de tipo entero, para guardar las columnas de cada entrada. Ambos conjuntos son de dimensión por lo menos N x MaxEntriesbyRows , donde N es el número de filas y MaxEntriesbyRows es el número máximo de elementos no nulos por fila en la matriz. Cabe destacar que el tamaño de todas las filas de estos conjuntos comprimidos A [] y j[] es el mismo, ya que cada fila se rellena con ceros, como se ve en la Figura 3.2.

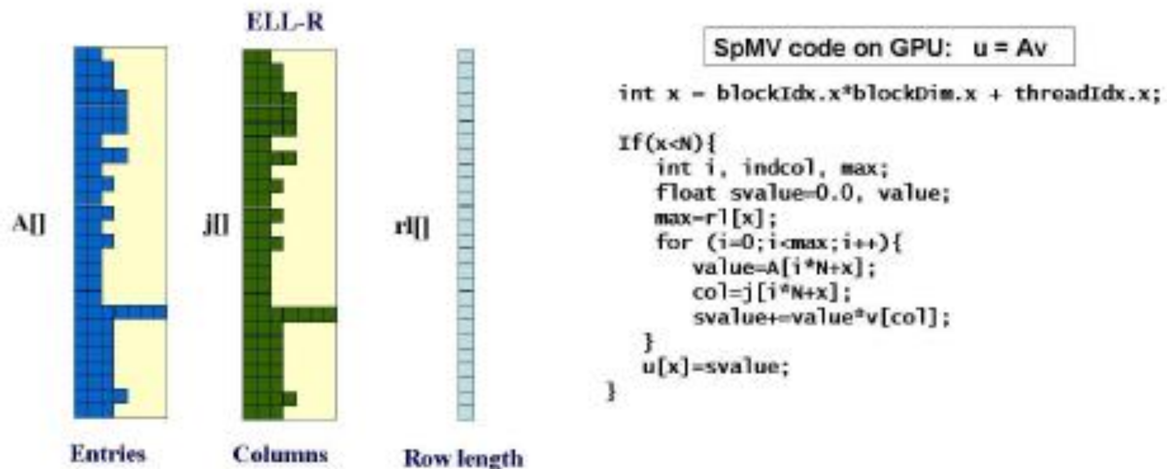


Figura 3. 2 Formato ELLPACK-R y *kernel* para computar SpMV en GPUs

Por lo tanto, ELLPACK puede considerarse como una estrategia para definir la matriz dispersa en una estructura de datos similar a la de una matriz densa. En consecuencia, este formato es adecuado para calcular las operaciones con matrices dispersas en las arquitecturas vectoriales. Sin embargo, si el porcentaje de ceros es alto en la estructura de datos ELLPACK entonces el rendimiento disminuye. ELLPACK-R es una variante de este formato que se adapta especialmente a la computación en GPU.

ELLPACK-R es una variante del formato ELLPACK e incluye dos matrices, A [] (float) y j [] (entero) de dimensión N x MaxEntriesbyRows y, además, se incluye un vector lineal entero adicional llamado *rl* [] de dimensión de N (es decir, el número de filas) con el propósito de almacenar la longitud real de cada línea, independientemente del número ceros de relleno. Un punto importante es el hecho de que las matrices se almacenan por columnas. Estas estructuras de datos ofrecen las ventajas siguientes:

(1) El acceso coalescente a memoria global, gracias a que se han almacenado las matrices por columnas en las estructuras de datos. Entonces, el hilo identificados por índice x accede a los elementos de la fila x : A [x + i * N] con (0 < i <*rl* [x]) donde i es el índice de la columna y *rl* [x] es el número total de no-ceros en la fila x . En consecuencia, dos hilos consecutivos x y x +1, acceden a direcciones de memoria consecutivas, con lo que cumplen las condiciones de acceso coalescente a memoria global.

(2) No es necesario sincronización en la ejecución de los diferentes bloques de hilos. Cada bloque de hilos pueden completar su cálculo, sin sincronización con otros bloques, porque cada hilo calcula un elemento del vector u (es decir, el resultado de la operación SpMV), y no hay dependencias de datos en el cálculo de los diferentes elementos de u.

(3) La reducción del tiempo de espera o el desequilibrio entre los hilos de un *warp*. La Figura 3.3 muestra un ejemplo de histograma de una pequeña matriz, y se considera un hipotético *warp* pequeño de ocho hilos, con el objetivo de ilustrar la ventaja de ELLPACK-R. La carga computacional de cada ocho hilos del *warp* es diferente y es proporcional al tamaño de la fila más larga dentro del correspondiente subconjunto de filas de la matriz. Teniendo en cuenta el *kernel* de SpMV con ELLPACK-R, la zona oscura es proporcional al tiempo de ejecución de cada hilo, y la zona gris es proporcional al tiempo de espera de cada hilo. Por lo tanto, sólo los *warps* relacionados con conjuntos de filas de longitudes muy diferentes están penalizados con tiempos de espera más largos, como puede verse en la Figura 3.3.

(4) Computación regular dentro de los hilos de un *warp*. Los hilos que pertenecen a un *warp* no divergen cuando el *kernel* de ejecución calcula SpMV. Ya que, el código no incluye instrucciones de control de flujo que provocan la serialización, de forma que cada hilo ejecuta el mismo bucle, pero con diferente número de iteraciones. Cada hilo se detiene tan pronto como termine su bucle, y los otros hilos en el *warp* continúan con la ejecución y mantienen las condiciones de acceso coalescente a la memoria global (véase la Figura 3.3. Esta característica tiene un impacto significativo en el rendimiento.

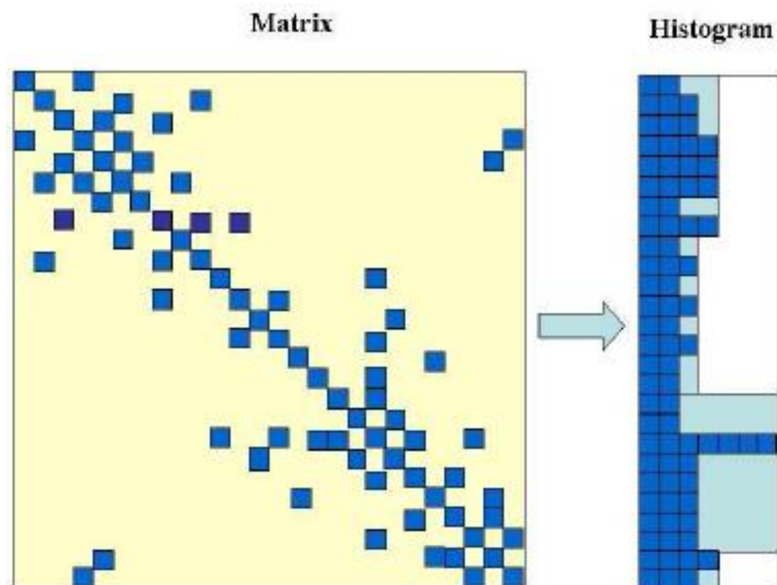


Figura 3. 3 Histograma de un ejemplo de una matriz pequeña que sirve como entrada hipotéticos pequeños *warps* de ocho hilos que colaboran en la computación de SpMV en la GPU

Recientemente, diferentes propuestas de *kernels* para calcular SpMV se han propuesto y analizado [31, 33,34]. Los *kernels* asociados al formato llamado HYB (híbrido), propuesto por [31] parecen obtener los mejores rendimientos en GPU. Este formato combina los formatos ELLPACK y COO para diferentes conjuntos de filas. Sin embargo, este formato requiere un preprocesamiento inicial, consistente en la reordenación de las filas para obtener un mejor rendimiento. Este preprocesamiento de la matriz es un inconveniente que puede producir una importante penalización, debido a las invocaciones de varios tipos de *kernels*, especialmente para las matrices de grandes dimensiones. Otros *kernels* basados en el formato CRS también han mostrado un elevado rendimiento y son la clave de la librería CUSPARSE [34]. CUSPARSE define múltiples bloques, donde cada bloque está a cargo del procesamiento de un grupo de filas y cada fila se asigna a un grupo de hilos. Por otra parte, se han implementado algunas estrategias adicionales para la mejora del rendimiento basadas en: (1) ajuste del número de hilos por fila para minimizar el desequilibrio entre los hilos, (2) patrón de acceso coalescente a la memoria global para leer la matriz y (3) el uso compartido y la memoria de textura [17]. Sin embargo, CUSPARSE no permite al usuario seleccionar el valor del parámetro de configuración BS para lograr el rendimiento óptimo.

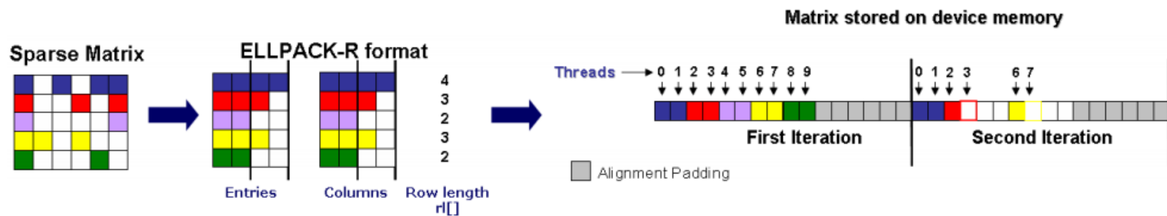
La rutina ELLR-T en la que se centra este proyecto, se basa en variantes del formato ELLPACK-R descrito previamente. El estudio comparativo del rendimiento de esta librería descrito en [22,24, 52] muestra que el rendimiento de ELLR-T supera al obtenido por las otras estrategias en la mayoría de las matrices evaluadas. A continuación se pasa describir el formato y mapeo de los *threads* en la GPU. Además también de definen los parámetros que el programador puede controlar para optimizar el rendimiento de ELLR-T de acuerdo con las características particulares de la matriz y arquitectura GPU.

3.4 ELLR-T

ELLR-T consiste en dos arrays, A [] (float) y J[] (integer) de dimensión NxMax_Nz y además un array adicional *rl[]* de dimensión N (número de filas) con el propósito de almacenar la longitud de cada fila sin importar el número de ceros rellenados.

Según el mapeo de los *threads* en la computación de cada fila, se pueden desarrollar distintas implementaciones de SpMV basados en ELLPACK-R, como se muestra en la parte inferior de la Figura 3.4.
Así, cuando T *threads* calculan el elemento *u[i]* accediendo a la fila i-ésima, la implementación se divide en grupos de T elementos.
Entonces, con el fin de calcular el elemento *u[i],* T *threads* calculan [*rl[i]/T]* iteraciones del bucle interno de SpMV. Cada *threads* almacena su cálculo parcial en la memoria compartida de la GPU. Por último, para generar el valor de *u[i]* se completa, una reducción de los valores calculados y almacenados en la memoria compartida. El valor del parámetro T se puede ajustar con el fin de obtener el mejor rendimiento con todo tipo de matrices dispersas. La Figura 3.4 ilustra las características del código ELLR-T destacando en la parte superior el almacenamiento para la matriz dispersa en la memoria del dispositivo para garantizar que el acceso a la memoria del dispositivo de cada grupo de T=2 *threads*

sea coalescente y alineados. Esta característica es muy relevante para ELLR-T debido al alto acceso a la memoria relacionada con el cálculo SpMV.




```
idx=(blockIdx.x*blockDim.x+threadIdx.x) # Thread index (BS=blockDim.x)
idb = threadIdx.x;   # Thread index into the block of index blockIdx.x
idp = idb % T;       # Thread index into S_i (set of T Threads related to the i-th row )
i = idx / T;         # Row index

if (i < N) {
  svalue=0.0;
  max=ceil(rl[i]/T);
  for(int k = 0;k < max;k++){
    value=A[k*(N'*T)+(i*T)+idp];# N'=new dimension for the memory alignment requirements
    col=J[k*(N'*T)+(i*T)+idp];
    svalue+=value*v[col];
  }
  shared[idb]=svalue;
  if(idp < 16){         # Reduction on shared memory to compute u[i] for T = 32
      shared[idb]+=shared[idb+16];
      if (idp < 8) shared[idb]+=shared[idb+8];
      if (idp < 4) shared[idb]+=shared[idb+4];
      if (idp < 2) shared[idb]+=shared[idb+2];
      if(idp == 0) u[i]=shared[idb]+shared[idb+1];
  }
}
```

Figura 3. 4 El almacenamiento de memoria ELL-T de la matriz dispersa cumplimiento la coalescencia y las condiciones de alineación para T = 2.

El algoritmo ELLR-T para el cálculo de SpMV con GPUs tiene en cuenta uno de los principales problemas de la computación en CUDA, que es el correcto uso de la memoria y el acceso a ella (véase 2.1.4) de la siguiente forma.

1. Coalescencia y acceso a memoria global. ELLRT representa la matriz mediante los arrays *A* y *J* de dimensión N × max. Requiere el uso del array *rl* de dimensión *N* para poder almacenar en una estructura regular la matriz dispersa, por lo que necesita un relleno de ceros hasta *max* en cada fila para obtener esa regularidad.

2. Computación homogénea de los *threads* de cada *warps*. Cada *thread* de un *warps* no diverge cuando se ejecuta el *kernel* SpMV. El código no incluye instrucciones de flujo que causan la serialización en *warps* desde cada subproceso ejecuta el mismo bucle,

pero con distintos números de iteraciones. Cada *threads* se detiene tan pronto como su bucle concluye y los hilos que no han concluido continúan la ejecución.

3. Reducción de la computación y desequilibrio de los *threads* de un *warps*. Un grupo de *threads* colaboran en el cálculo de *u[i]*, el número de iteraciones del bucle alcanza su máximo valor en $k=[rl[i]T]<[Max-nzr/T]$.

El modelo para determinar los parámetros de configuración BS y T se basa en considerar que operación SpMV está dominada por los accesos a memoria para la lectura de matriz. El modelo evalúa para distintas combinaciones de valores posibles BS y T los accesos a memoria que efectúa cada SM de acuerdo con el esquema de ejecución de la GPU teniendo en cuenta el número de SM, su política de Scheduling, y el número de elementos no nulos de la matriz. De esta forma a partir de la información particular sobre la arquitectura GPU y la matriz que se estén considerando se determinan de forma automática los parámetros de configuración de ELLR-T.

4. MAT-ELLR-T

4.1 Introducción

MAT-ELLR-T es una librería que permite procesar la operación del producto matriz dispersa vector (SpMV) con GPUs desde MATLAB. Esta librería se basa en la rutina ELLR-T, descrita en la sección anterior y desarrollada en CUDA para ser usada en programas codificados en C o C++. Una de las principales ventajas que aporta el desarrollo de MAT-ELLRT es que las rutinas pueden ser invocadas desde MATLAB de forma transparente al usuario. De este modo, se está aprovechando, por un lado, la rapidez de cómputo proporcionada por la GPU y, por otro, las ventajas del entorno de programación MATLAB.

Por tanto, la operación SpMV que es clave en el desarrollo de una gran diversidad de modelos y aplicaciones científicas y de ingeniería [3, 4, 5] podrá ser invocada desde MATLAB y además puede ser acelerada en la GPU en base a la rutina CUDA denominada ELLR-T. La estrategia de diseño que se optó para MAT-ELLR-T fue un desarrollo modular para que el usuario pueda seleccionar lo que necesite de forma rápida y cómoda.

Otro aspecto importante ha sido la dificultad de ensamblar el nivel de abstracción de la programación de alto nivel de MATLAB con el nivel de abstracción de más bajo nivel de CUDA. Haciendo trasparente para el usuario las reservas de memoria de la GPU, la coalescencia de los datos, operaciones internas y las comunicaciones entre las plataformas gracias a los ficheros MEX. De este modo se aprovecha por un lado, la rapidez de cómputo proporcionada por la GPU y, por otro, las ventajas del entorno MATLAB.

Es necesario incluir una etapa de preproceso antes de realizar la operación SpMV porque la GPU necesita tener la matriz en formato ELLR-T ya que se debe almacenar los valores en estructuras de datos regulares. Esta es la única función que utiliza la GPU, los demás módulos son solo preprocesos de datos necesarios para su utilización.

Los MEX son los encargados del flujo de datos de entrada y salida MATLAB /C, porque, reciben los datos desde MATLAB, reservan la memoria correspondiente para su uso en C. Desde el código C se realizan las reservas de memorias para CUDA y se invocan a los *kernels* de CUDA para activar la computación en la GPU. Para el cálculo de SpMV en la GPU se necesita tener alojado en la memoria de la GPU la matriz en formato ELLR-T. Además, es necesario definir los datos de la matriz de forma persistente con el objetivo de que sigan alojados en memoria incluso después de que concluya la ejecución del *kernel* SpM y no sea necesario efectuar nuevas copias de la matriz si se vuelve a efectuar operaciones con la misma matriz en la GPU.

Para ello se realiza la persistencia de la matriz en formato ELLR-T en la GPU. Este proceso consiste en copiar la matriz en la GPU para solo tener una única penalización por el copiado de los datos a la GPU si se vuelve a utilizar la matriz dispersa en otras llamadas desde MATLAB. La utilización de la rutina SpMV suele estar unida a la resolución de sistemas de ecuaciones. Todos estos procesos utilizan algoritmos iterativos de forma que en cada iteración se calcula al menos un producto de la matriz dispersa del sistema por un

vector que va evolucionando con las iteraciones. Por tanto el uso de la persistencia supone un ahorro considerable de recursos en cada iteración de este tipo de algoritmos (véase Sección 5). La copia de la matriz a la GPU se realiza desde el código C que recibe la matriz a través del MEX. La persistencia de la matriz solo se realiza una vez, el sistema detecta si hay alguna matriz alojada en la GPU y de ser así, no realizará la copia. Al hacer persistente la matriz en la memoria de la GPU, existe el límite físico de la memoria de la GPU.

Los consumos de tiempo más altos son la carga de matrices, la determinación de parámetros de configuración GPU y el almacenamiento de la matriz en formato ELLR-T, mientras que el cálculo de SpMV tiene un rendimiento muy bueno. Por eso, para evitar la penalización del resto de los módulos se podrán realizar en bloque una sola vez y almacenar los datos, para poder usar el cálculo SpMV.

Es aconsejable el uso del formato .mat de MATLAB para el almacenamiento interno de los datos obtenidos de los módulos, porque permite guardar datos para un uso posterior sin necesidad de ejecutar nuevamente módulos de MAT-ELLRT lo que supone un ahorro de recursos.

Se puede observar en la Figura 4.1 que el flujo de los módulos que constituyen MAT-ELLRT. Esta figura representa el flujo de ejecución de MAT-ELLRT desde que se carga una matriz desde un fichero .mtx. No obstante, cuando la matriz se genera en un código MATLAB o se lee de un fichero .mat se puede evitar el uso de algunos de los módulos de MAT-ELLRT.

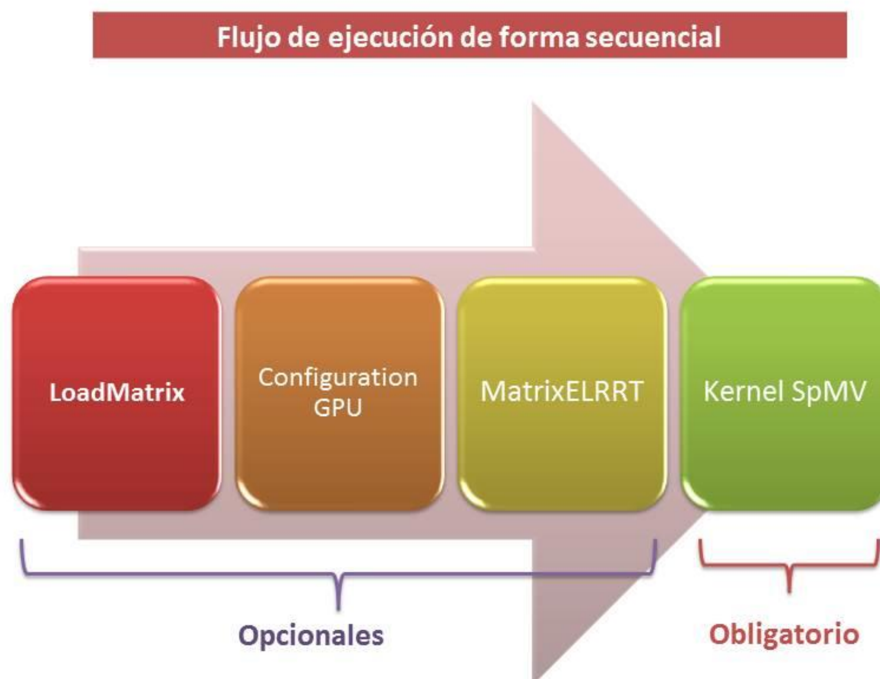


Figura 4. 1 Imagen del flujo de ejecución

Como ya se mencionará posteriormente todos los datos de los módulos se pueden guardar para un posterior uso. Por ese motivo se puede estructurar los módulos en dos bloques como se muestra en la Figura 4.2.

- **Bloque de obtención de datos (ópcionales):** Permite obtener todas las estructuras de datos necesarias para la utilización de SpMV.
- **Bloque de Cálculo(Obligatorio):** Realiza la operación SpMV en la GPU.

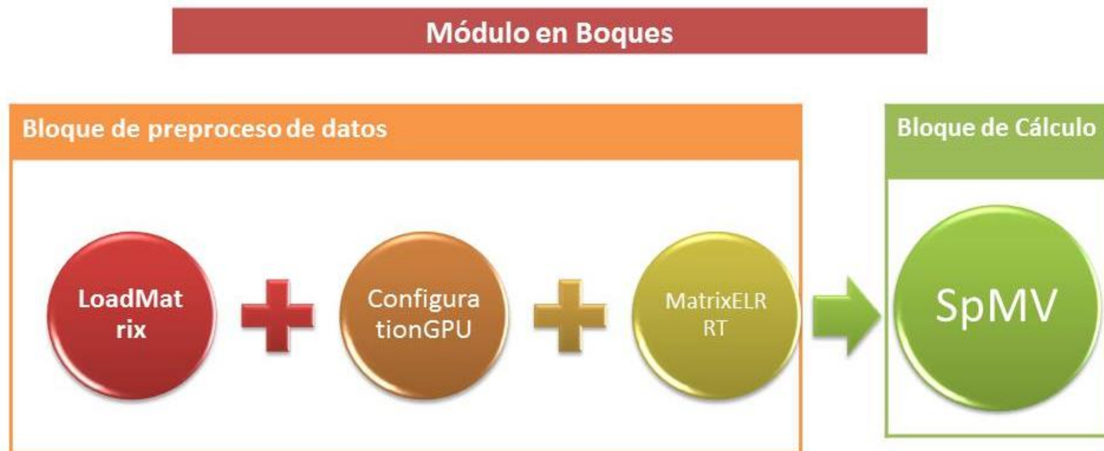


Figura 4. 2 Imagen de los módulos por bloques

La estructura de la librería puede ser fácilmente modificable, adaptando los módulos en código .m de MATLAB. Se ha realizado así porque cada usuario puede obtener las matrices de distintas formas o que su forma de trabajo sea distinta al flujo de módulos de la librería. Por ejemplo:

Está ideada la librería para que todo empiece con una matriz en formato COO en un archivo .mtx y poder ejecutar todos los módulos de forma secuencial como mínimo la primera vez, a partir de ahí, existe numerosas variantes de uso, como guardar todos los resultados de los módulo para no tener que volver a calcularlo o solo guardar algunos datos. . Además parte de eso, el usuario puede generar su matriz desde MATLAB y usar MAT-ELLRT con una sencilla adaptación al módulo LoadMatrix .

MAT-ELLRT se ha estructurado en los siguientes módulos:

- **Carga de matrices:** Este módulo permite realizar la carga de una matriz del Matrix Market en formato COO (simétrica o no simétrica), también sirve para cualquier fichero con la matriz en formato COO. Para las matrices simétricas el módulo debe convertirla en una matriz completa, ya que, las matrices simétricas del Matrix Market solo guardan la mitad de la matriz. Una penalización sobre las matrices de grandes dimensiones suelen ser que tienen tiempos de carga muy altos.
- **Determinación de parámetros de configuración de GPU:** Este módulo determina los parámetros adecuados para la configuración de la GPU dependiendo de la arquitectura y de la distribución de la matriz (definición de los parámetros BlockSize y T el número de *threads* que intervienen en el producto de cada fila de la matriz por el vector).

- **Almacenar la matriz en el formato ELLR-T:** Este módulo convierte la matriz en el formato ELLR-T, que son 3 arrays (BA, Bindex y BCol).
- **Kernel SPMV:** Este módulo es más importante porque es el que realiza la operación en la GPU si, se puede llamar tantas veces como haga falta sin necesidad de llamar a las fases anteriores. Este módulo realiza la persistencia de la matriz en formato ELLR-T en la GPU, gracias a eso, solo se realiza la copia de la matriz a la GPU una sola vez.

La Figura 4.3 muestra un breve esquema del MAT-ELLRT en el que se identifican cada uno de los módulos que constituyen la librería así como la función que desempeñan.

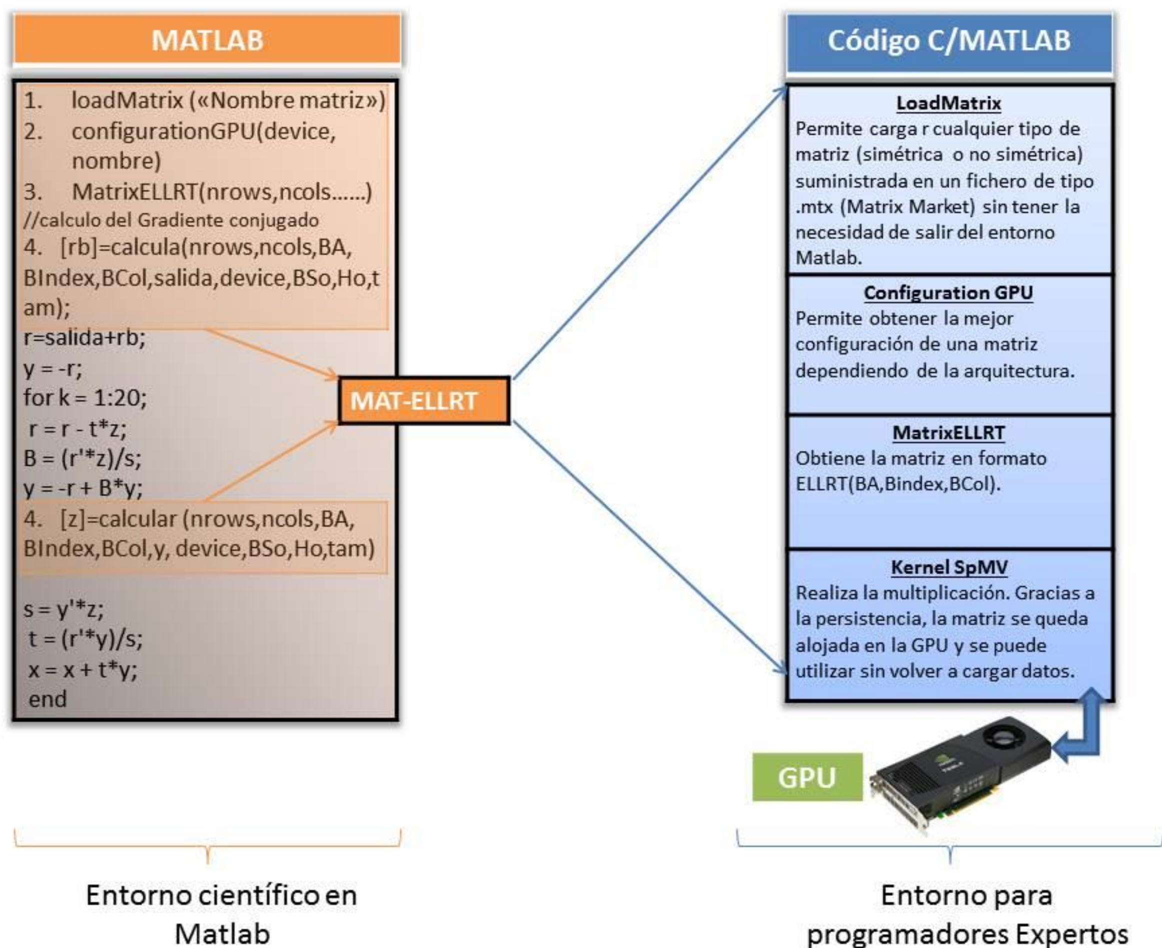


Figura 4. 3 Esquema de la integración de MAT-ELLRT por módulos

4.2 Load Matrix

El objetivo de este módulo es la carga de matrices (simétrica o no simétrica) suministradas en un fichero .mtx (Matrix Market) de tipo texto sin tener la necesidad de salir del entorno MATLAB y almacenándolas en un fichero binario con un formato COO.

Para las matrices simétricas, los ficheros .mtx solo se guardan la mitad de la matriz por lo que MAT-ELLRT debe convertir la matriz en una matriz completa.

La plataforma Matrix Market [47] proporciona un repositorio de datos de prueba para su uso en estudios comparativos de algoritmos de álgebra lineal numérica. Se proporcionan matrices dispersas, así como software y servicios de generación de matrices, a partir de problemas de sistemas lineales, mínimos cuadrados, y cálculos de valores propios implicados en una amplia variedad de disciplinas científicas y de ingeniería. También se incluyen herramientas para la navegación a través de la recolección o para la búsqueda de matrices con propiedades especiales.

Cada matriz tiene su propia "página de inicio", que proporciona información sobre las propiedades de la matriz, la visualización de la estructura de la matriz, y los permisos de descarga de la matriz en uno de varios formatos de archivo de texto. Del mismo modo, cada generador de matriz tiene una página de inicio que describe sus propiedades. Esto permite que de una forma sencilla se obtenga la matriz para su utilización posterior. La matriz se almacena en un fichero .mtx con formato COO.

En la siguiente imagen se puede observar la llamada que se ejecuta desde MAT-ELLR-T hacia *Module_LoadMatrix*, esta a su vez invoca a *Simetric_N* que realiza la carga de matrices no simétricas o a *Simetric_S* , que esta, realiza una llamada a su vez al código C que realiza la simetría.

Este módulo consta de dos submódulos Figura 4.4:

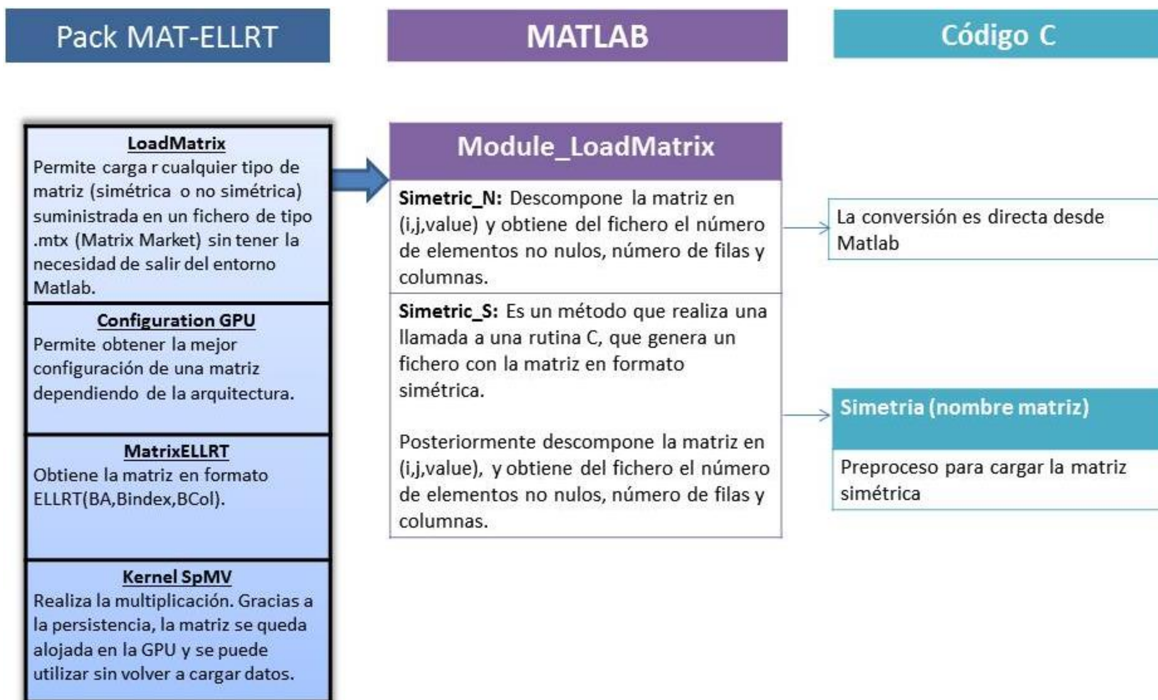


Figura 4. 4 Estructura de las rutinas de LoadMatrix

Los datos de entrada necesarios son:

- **Nombre:** Nombre de la matiz
- **Tipo**: Indicar si es simétrica o no

A través del nombre de la matriz, el módulo es capaz de descomponer la matriz para obtener los siguientes valores:

- **Nrows**: Número de filas de la matriz.
- **Ncols**: Número de columnas.
- **Nonull**: Número de elementos no nulos
- **I**: Un array con los valores de la componente I de la matriz, con un tamaño de nonull.
- **J**: Un array con los valores de la componente J de la matriz, con un tamaño de nonull.
- **Value**: Un array con los valores de la matriz, con un tamaño de nonull.

Las matrices simétricas necesitan ser completas para que MAT-ELLRT pueda trabajar con ellas como de una matriz estándar se tratara, el encargado de esta conversión es el submódulo Simetric_S, que realiza la conversión y genera un fichero con nombre:

simetric_"nombre de la matriz"

El objetivo de la generación del fichero de la simetría es un método de ahorro de recursos, ya que el usuario podría ejecutar el módulo de Simetric_N indicándole el nombre del fichero sin necesidad de realizar otra vez la simetría, que tiene un coste de cómputo mayor que Simetric_N.

Este módulo también es útil para cualquier trabajo con MATLAB en especial en el ambiente del algebra lineal, debido a las numerosas funciones que necesitan la componente (i,j,value,nrows,ncols y el número de no nulos).

Como una opción después de la primera carga si esta ha sido computacionalmente muy costosa, sería guardar los datos en un fichero .mat, para no tener que repetir esta tarea en caso de que se efectuaran más operaciones con la matriz. De esta forma se tendría la matriz en todo momento a disposición del usuario. Esta opción, es compatible con el funcionamiento de MAT-ELLR-T y para todos sus módulos.

Una vez ejecutado el módulo, se eliminan todos los datos auxiliares generados para la obtención de datos.

4.3 Configuration GPU

El objetivo de este módulo es la determinación de los parámetros adecuados para la configuración del *kernel* SpMV en la GPU dependiendo de la arquitectura (número de streaming multiprocessors) y de la distribución de los elementos no nulos en las filas de la matriz (vector *rl*). De esta forma el usuario no tiene que preocuparse de ajustar manualmente los dos parámetros de ajuste que se han descrito previamente *BS* y *T*, ya que, el módulo lo genera de forma automática. Recuérdese que *BS* denota el tamaño de los bloques y *T* el número de *threads* que se encargan de ejecutar el producto de cada fila de la matriz.

Para una explotación total de la GPU en la operación SpMV, es necesario, una correcta configuración tanto del tamaño de bloque como del número de hilos a utilizar por fila, de acuerdo con las distribución de los elementos no nulos de la matriz y de los recursos de la GPU. Estos parámetros dependen:

- **Hilos (T)**: Depende de la longitud promedio de las filas (array *rl*). Si , cada fila está poco llena utilizará pocos hilos para efectuar el producto, en cambio, si está muy llena utilizará un mayor número de hilos.

- **Tamaño del bloque (BS)**: También depende del Array *rl*, pero en este caso, sirve para organizar la ejecución concurrente de hilos en la arquitectura.

La combinación de valores de T y BS que permita una distribución más equilibrada de la carga en el conjunto de Streaming Multiprocessors de la GPU es la que obtiene un mejor rendimiento de la operación SpMV en la GPU. El modelo matemático que se utiliza para la determinación de estos parámetros se describe en [53].

En la Figura 4.5 se puede observar la llamada que se ejecuta desde MAT-ELLR-T hacia *Module_ConfigurationGPU*, primero inicializa la GPU con *InizialiteGPU indicando la GPU,* seguidamente se lanza *Module_ConfigurationGPU* que obtiene el tamaño del bloque (*BS*) y el número de *thread* (*T*), tras la aplicación del mencionado modelo.

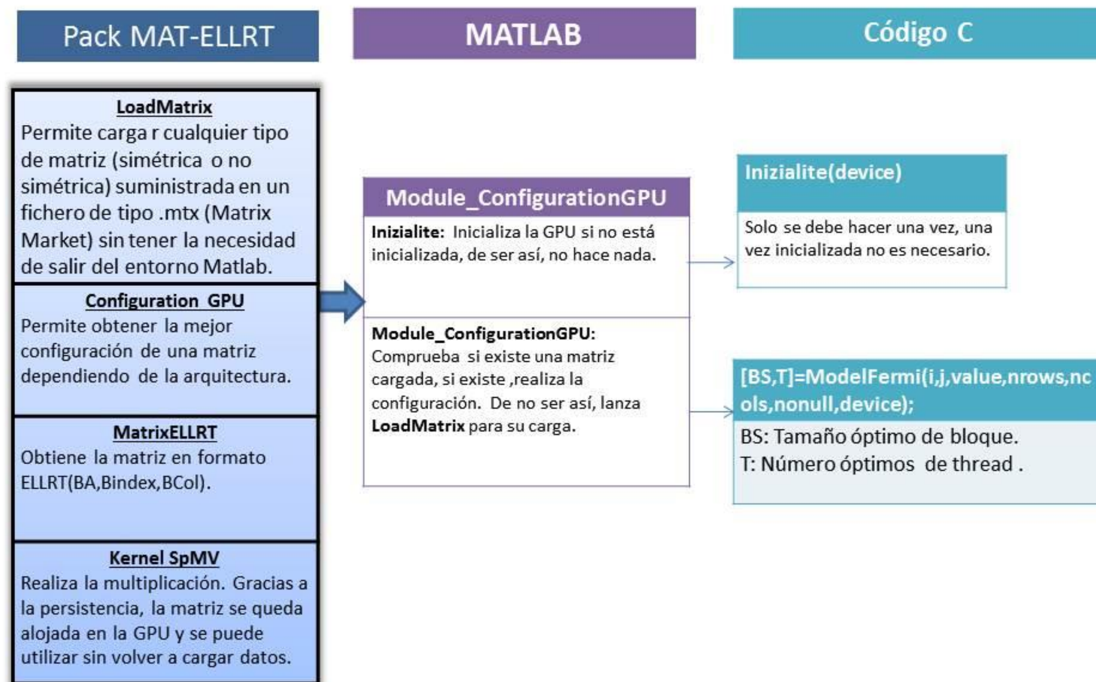


Figura 4. 5 Descomposición por rutinas de Configuration GPU

El módulo necesita

Los datos de entrada necesarios son:

- **Nrows**: Número de filas de la matriz.
- **Ncols**: Número de columnas.

Los datos de salida son:

- **BS:** Tamaño óptimo de bloque.
- **T:** Número de *thread* óptimos.

Esta rutina es de fácil aplicación, se puede lanzar antes de la ejecución para guardar los datos para su posterior uso, o simplemente insertarlo en un .m más complejo. Ya que las aplicaciones que realizan SpMV suelen trabajar siempre con la misma matriz y lo que cambian es el vector a multiplicar.

Una vez obtenidos los parámetros, es aconsejable guardarlo para no tener que lanzar nuevamente el módulo, debido a su coste de recursos.

4.4 Matrix ELLRT

El objetivo de este módulo es la obtención de la matriz en formato ELLR-T necesario para la ejecución de SpMV.

Internamente Matrix ELLR-T, trabaja con los arrays BA, BIndex y BCol que corresponden con los arrays. A, J y rl, mencionados en la Sección 3.3.

En la Figura 4.6 se puede observar la llamada que se ejecuta desde MAT-ELLRT hacia *Module_matrixELLRT*, este a su vez invoca primero a *Calculo_formato1* que a su vez lanza un código C que obtiene el tamaño de los vectores BA y Bindex, seguidamente se lanza *calculo_formato2* y este a su vez lanza un código C que obtiene los vectores del formato ELLR-T (BA,Bindex,BCol).

El módulo, genera y guarda los valores de la matriz en formato ELLR-T con 3 arrays.

- **BA:** Proporciona los valores no nulos de la matriz en una estructura de dos dimensiones regular con el mismo número de filas que la matriz y tantas columnas como el número máximo de elementos no nulos en todas las filas de la matriz. Las posiciones de la estructura que no corresponde a ningún elemento no nulo de la matriz dispersa se rellena con ceros.
- **Bindex:** Proporciona las coordenadas de la matriz en una estructura regular de la matriz dispersa sobre los elementos no nulos, las posiciones de la estructura que no corresponde a ningún elemento de la matriz dispersa se rellenar con ceros
- **BCol:** Proporciona el número máximo de elementos no nulos por fila. Este valor es muy útil para no realizar operaciones sobre elementos no nulos y ajustar lo máximo posible el tamaño de BA y BIndex.

Los detalles completos sobre el formato ELLR-T se pueden encontrar en la Sección 3.3. Únicamente destacamos aquí que el orden en el que se almacenan los datos en la estructura de datos depende del valor de T, de ahí la necesidad de usar este parámetro como entrada para generar el formato ELLR-T de la matriz.

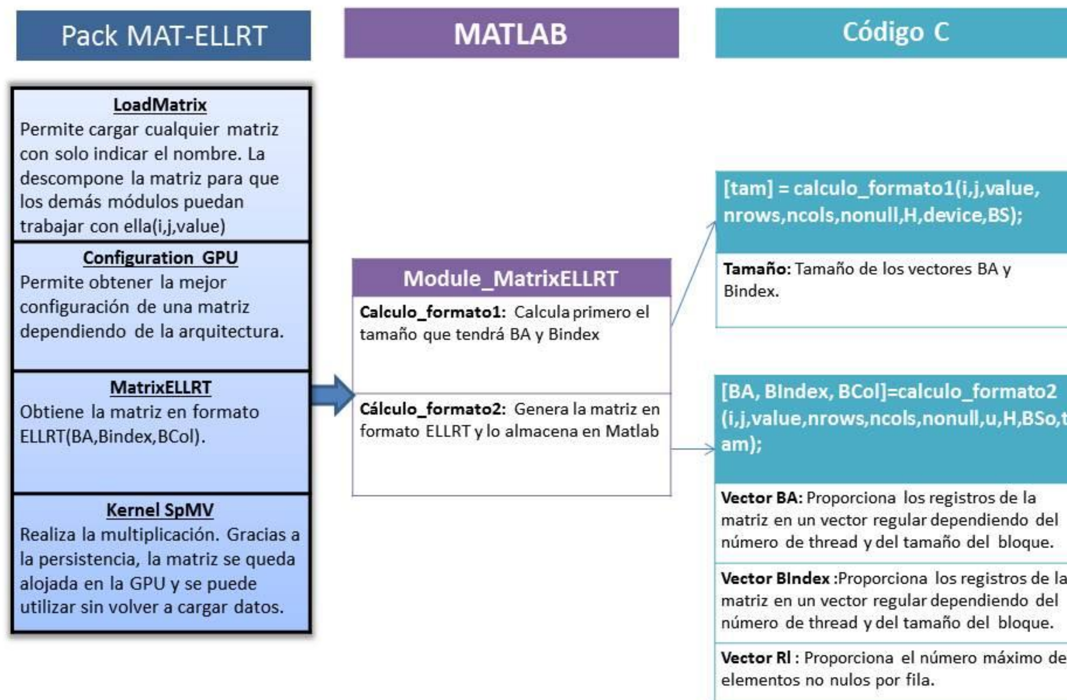


Figura 4. 6 Descomposición por rutinas de MatrixELLR-T

Datos de entrada:

- **Nrows**: Número de filas de la matriz.
- **Ncols**: Número de columnas.
- **Nonull**: Número de elementos no nulos
- **I**: Un array con los valores de la componente I de la matriz, con un tamaño de nonull.
- **J**: Un array con los valores de la componente J de la matriz, con un tamaño de nonull.
- **Value**: Un array con los valores de la matriz, con un tamaño de nonull.
- **BS:** Tamaño de bloque.
- **T:** Número de *thread.*

Este módulo contiene dos llamadas:

- **calculo_formato1**: calcula la dimensión que tendrá BA y Bindex

```
[tam] = calculo_formato1 (i,j,value,nrows,ncols,nonull,T ,BS);
```

- **calculo_formato2**: genera la matriz en formato ELLR-T y lo almacena en MATLAB

```
[BA, BIndex, BCol]=calculo_formato2 (i,j,value,nrows,ncols,nonull,T, BS,tam);
```

Hay que destacar que la actuación de estas rutinas para generar la matriz en formato ELLRT puede ser computacionalmente muy costosa. Una opción después de la primera ejecución si esta ha sido muy costosa, consiste en guardar los datos en un fichero .mat, para no tener que repetir esta tarea. De esta forma se tendría los arrays necesarios para realizar el cálculo SpMV en todo momento a disposición del usuario. Así de esta forma, no sería necesario almacenar nada en memoria de los módulos anteriores, algo recomendable para liberar memoria para el cálculo.

Esta opción, es compatible con el funcionamiento de MAT-ELLR-T y para todos sus módulos. Una vez almacenado el archivo con formato .mat se podría lanzar tantas veces se quisiese el módulo de SpMV el único inconveniente sería el posterior acceso al fichero cuyo tamaño puede ser considerable.

Uno de los problemas de rendimiento de este módulo no es el coste computacional de calcular los arrays, sino del volcado de memoria a MATLAB, porque dependiendo del tamaño de la matriz puede rondar entre 1Gb y 5Gb lo que equivale de 10sg a 60sg en términos de rendimiento.

Una vez obtenidos los arrays, no es necesario mantener memoria los vectores i, j y value, por defecto se eliminan para disponer del máximo de memoria libre.

4.5 Kernel SpMV

El objetivo de este módulo es realizar el cálculo del producto de la matriz dispersa por el vector utilizando la GPU desde MATLAB.

Es el módulo más importante y más complejo de todos ellos porque los datos pasan por todas las plataformas hasta alojar la matriz en formato ELLR-T en la GPU de manera persistente. La importancia de la persistencia es evitar la penalización de la copia de datos del *host* al *device* y viceversa, visto en el apartado 2.1.4

En la siguiente imagen se puede observar la llamada que se ejecuta desde MAT-ELLR-T hacia *Module_SpMV*, primero inicializa la GPU con *InizialiteGPU indicando la GPU,* seguidamente se lanza *CalcularSpmv* que lanza el código C *calcular_spmv* que a su vez, llama la función *spmv_ELLR-Tdp* que realiza la operación en la GPU y devuelve la salida, Figura 4.7.

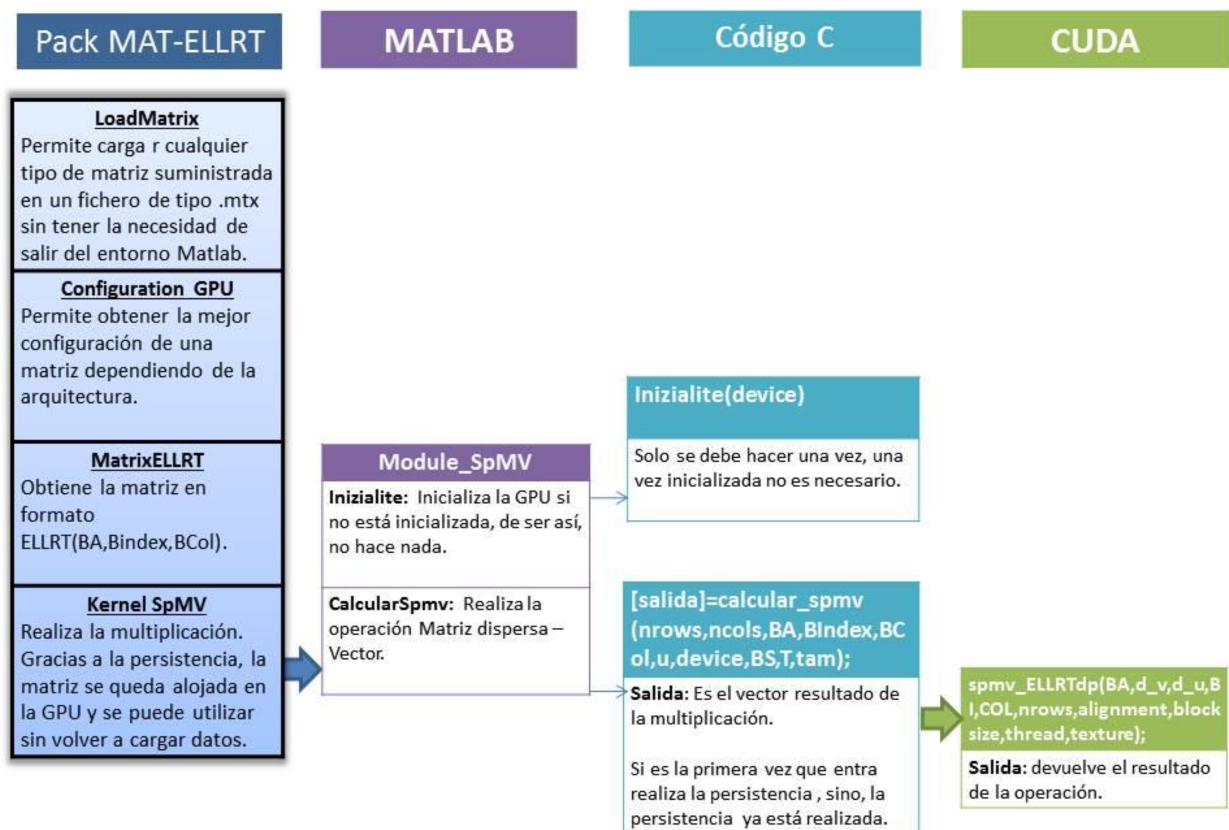


Figura 4. 7 Descomposición por rutinas de SpMV

Cuando se ejecuta el módulo SpMV, se realiza una llamada a un código C, que detecta si es la primera vez que es llamado o no. Esta diferencia la hace para realizar la persistencia de datos o no.

- Si el programa detecta que es la primera vez que se ejecuta este módulo, crea la persistencia de la matriz en formato ELLR-T con los arrays BA, BIndex y BCol en la

memoria de la GPU para no tener que almacenarse repetidamente los datos si se utiliza en diversas operaciones SpMV. Está técnica mejora considerablemente el tiempo de ejecución, porque, aprovecha totalmente el potencial de la GPU disminuyendo la penalización a memoria al mínimo, ya que solo se copia una vez los arrays.
- Si el programa detecta que se ha inicializado anteriormente el módulo, tendrá la matriz alojada en la GPU y así solo se volcará a la memoria de la GPU el vector a multiplicar.

La persistencia se crea y se gestiona desde el MEX.

Importante: El programa no detecta si la matriz que está alojada en la GPU corresponde con la de la próxima ejecución, por eso, es recomendable destruir la persistencia de los datos cuando no se vaya a trabajar con ella. De esta forma también se liberará memoria, porque, la persistencia tiene un consumo elevado de memoria RAM.

4.6 Interfaz de usuario

La interfaz es un recurso adicional que se ha desarrollado para ayudar al usuario a utilizar y familiarizarse con MAT-ELLRT.

Gracias a la modularidad de MAT-ELLRT, se puede adaptar perfectamente a la necesidad de cada usuario de una forma sencilla, ya que, el usuario dispone de una interfaz de entrenamiento guiada para el proceso del cálculo SpMV. Además se ilustra el uso de MAT-ELLRT para la solución de un sistema lineal definido por una matriz dispersa mediante el método del Gradiente Conjugado que es de tipo iterativo, a partir del código MATLAB que se suministra el usuario puede adaptarlo fácilmente para la implementación de otros métodos iterativos.

Es importante que el usuario comprenda todo el proceso de la rutina, por eso la interfaz está separada por módulos para que el usuario pueda utilizarlos de forma separada o integrarlos en su sistema.

En este apartado se explicará la interfaz guiada de MATLAB que tiene como objetivo familiarizarse del flujo de la ejecución secuencial visto en 4.1.

El primer paso para la integración es descomprimir MAT-ELLRT.zip en el workspace donde se vaya a trabajar.

Una vez integrado MAT-ELLRT en el workspace de MATLAB ya se puede ejecutar.

El nombre del ejecutable es: **PackELLRT** Figura 4.8

```
Welcome to MAT-ELLRT
_____________________
 Data Processing
 ________________
  1: Load Matrix
  2: Configurate GPU
  3: MatrixELLRT
  4: Automatic step (1-2-3)
 __________________________
 Data Calculation (You need to create the variables before you continue)
 _________________
  5: SpMV
  6: Calculate CGS
  7: SpMV Basic(whitout persistence)
  8: Exit
__________________________________
     Choose:
```

Figura 4. 8 Interfaz de PackELLR-T

La Figura 4.9 es un esquema de la integración de todos los módulos de MAT-ELLRT que se explicarán a continuación. Esto facilitará la compresión del flujo de ejecución inicial de MAT-ELLRT para que sean capaces de su integración en otros sistemas.

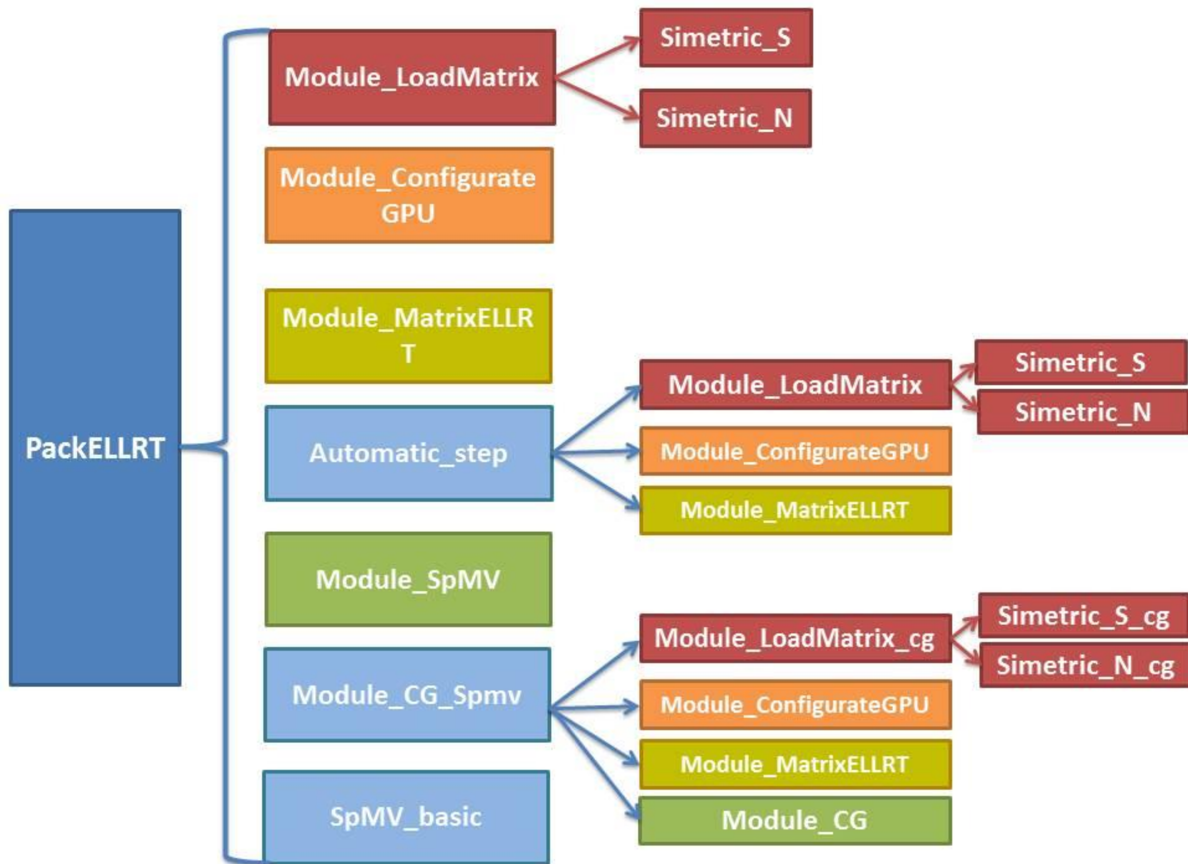


Figura 4. 9 Diagrama de MAT-ELLRT desglosados por módulos en .m

Consta de 8 funciones dividido en dos bloques:

Bloque de procesamiento de datos

1. LoadMatrix (Módulo para cargar una matriz a MATLAB)
2. Configurate GPU (Módulo obtener el tamaño del bloque y el número de *threads* óptimo)
3. MatrixELLR-T (Genera en formato ELLRT la matriz cargada)
4. Atomatic step (Genera de forma secuencial y automática el paso 1,2 y 3)

Bloque de cálculo

5. SpMV (permite realizar la operación SpMV con persistencia)
6. Automatic CG (permite realizar el cálculo de CG)
7. SpMV Basic (permite realizar la operación SpMVsin persistencia)

4.6.1 Bloque de procesamiento de datos

1. **LoadMatrix**

LoadMatrix permite obtener la matriz en el formato adecuado para su uso.
El formato de la matriz Figura 4.10 debe ser .mtx y tener formato COO:

```
comentarios      %%MatrixMarket matrix coordinate real ge
nrows,ncols      2048 2048 10114 nonull
                 1 1  4.3892986335356e-01
                 2 1  2.1867987492099e-03
                 33 1  7.3181090110147e-04
i  j value       1 2  2.1867987492099e-03
                 2 2  4.3777970117382e-01
                 3 2  3.3493014241990e-03
```

Figura 4. 10 Formato de entrada del fichero .mtx de la matriz

También se puede lanzar LoadMatrix con un fichero de distinta extensión, siempre y cuando cumpla con el formato anterior.
Una vez ejecutado desde la interfaz con la opción 1, se deberá indicar si la matriz es de tipo simétrica o no y su nombre Figura 4.11.

```
    Choose: 1
IF the matrix is symmetric push 1, ELSE push 0: 1
Put the name of Symmetric Matrix:  cant.mtx
```

Figura 4. 11 Interfaz de ejecución del modulo Simetria_s

El módulo cargará los datos a MATLAB, si la matriz es muy grande tardará unos segundos.

Si el usuario desea agilizar el proceso porque solo usa matrices simétricas puede integrar en su programa con el submódulo **simetria_S**, si fuese al contrario, que las matrices que usa no son simétricas, integrará el submódulo **simétrica_N.**

Si desea integrar ambas funciones, debe llamar a **module_LoadMatrix.**

Importante: Al lanzar este módulo siempre se cargará una matriz nueva, no contempla si existe una matriz cargada con anterioridad. Es el único módulo que no hace esta distinción. Por eso, cada vez que se quiera cambiar de matriz es necesario ejecutarlo.

2. **Configurate GPU**

ConfigurateGPU obtiene el tamaño del bloque y el número de *thread* adecuados para efectuar el producto de cada fila de la matriz por el vector de acuerdo con la arquitectura de la GPU y la distribución de los elementos no nulos de la matriz.

Al ejecutar Configurate GPU con la opción 2 del menú realiza una comprobación, que es: si existe una matriz cargada o no Figura 4.12.

```
1 -     disp('Configurate GPU')
2 -     disp('_______________')
3 -     device=input('Put the device number: ');
4 -     if v1==1;
5 -     disp('If you wish chance the matrix, run LoadMatrix ')
6 -     [BS,T]=ModelFermi(i,j,value,nrows,ncols,nonull,device);
7 -     elseif v3==1;
8 -         disp('If you wish chance the matrix, run LoadMatrix ')
9 -         [BS,T]=ModelFermi(i,j,value,nrows,ncols,nonull,device);
.0
.1 -    else
.2 -        disp('there isn´t a matrix loaded ')
.3 -        module_LoadMatrix
.4 -        [BS,T]=ModelFermi(i,j,value,nrows,ncols,nonull,device);
.5 -    end
```

Figura 4. 12 Código que verifica si existe una matriz cargada

De existir una matriz cargada con el módulo 1 o 3 se lanza solo la llamada *ModelFermi* que necesita el parámetro *device*, que será pedido por pantalla, sino existe ninguna matriz cargada, se lanza *module_LoadMatrix* y posteriormente *ModelFermi* de forma automática.

Si un usuario quisiera integrar este módulo en un módulo más complejo le bastaría con llamar a *Model_Fermi:*

[BS, T]=ModelFermi(i,j,value,nrows,ncols,nonull,*device*);

Y obtendrá las variables BS (BlockSize) y T (*Thread*).

Importante: Si desea obtener la configuración para otra matriz es necesario lanzar LoadMatrix o salirse de PackMAT-ELLR-T y volver a ejecutarlo. De esta forma ya no detectará que existe una matriz cargada.

3. MatrixELLR-T

Este módulo convierte la matriz en el formato ELLR-T (BA, Bindex y BCol).

Al ejecutar MatrixELLR-T con la opción 3 del menú comprueba si se ha ejecutado antes ConfigurateGPU. Esto es debido a la necesidad de BS y T para la realización del formato ELLR-T. Si no ha sido ejecutado, se deberá introducir manualmente un BS y un T, aunque lo recomendable es realizar primero el paso 2 y luego el 3 Figura 4.13.

```
1 -      disp('MatrixELLRT')
2 -      if v2==1
3 -      ELLRT
4 -      disp('Created Arrays')
5 -      end
6 -      if v2==0
7 -          disp('You dont run ConfigurateGPU, you must input manual :')
8 -          disp('First Load Matrix')
9 -          module_LoadMatrix
10 -         T=input('Put the threads numbers (1,2,4,8,16) ');
11 -         BS=input('Put the Block Size (128,256,512)');
12 -         ELLRT
13
14 -     end
```

Figura 4.13 Código Maltab para el guiado del cálculo de la matriz en formato ELLRT

Si un usuario quisiera integrar este módulo en un módulo más complejo bastará con llamar a *ELLRT*, que este módulo contiene las llamadas: *calculo_formato1 y calculo_formato2* Figura 4.14*:*

```
%ELLRT
[tam] = calculo_formato1(i,j,value,nrows,ncols,nonull,T,BS);
[BA, BIndex, BCol]=calculo_formato2 (i,j,value,nrows,ncols,nonull,T,BS,tam);
```

Figura 4. 14 Llamada a las funciones que integran la rutina ELLR-T

Estas dos funciones deben ir juntas y no deben separarse, porque *calculo_formato1* obtiene el tamaño que tendrán BA y BIndex en *calculo_formato2.* Por eso se han unificado en el módulo ELLR-T

Importante: En la introducción de los parámetros de forma manual no usar datos distintos a los mostrados por pantalla, porque el sistema NO dará error en ese momento, pero al ejecutar el módulo SpMV o *CalculateCG* si lo hará.

4. Automatic 1-2-3

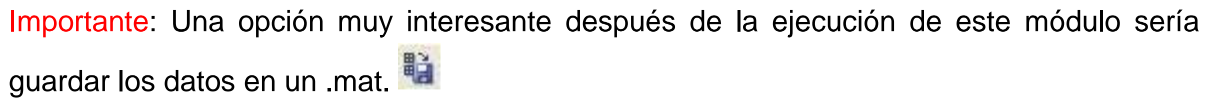
Este módulo permite la realización de LoadMatrix, configurateGPU y MatrixELLR-T de forma automática.

Esto es una prueba de las ventajas que tiene MAL-ELLR-T que desde un solo módulo permite obtener todos los datos necesarios para la realización de SpMV, Figura 4.15.

```
1 -          v1=1;
2 -          module_LoadMatrix
3 -          module_configurationGPU
4 -          v2=1;
5 -          module_MatrixELLRT
6 -          v3=1;
```

Figura 4. 15 Llamadas que permite realizar el paso 1,2 y 3 con una sola llamada

NOTA: Las variables v1, v2 y v3, son variables auxiliares que controlan por los módulos que han pasado para mostrar unas opciones u otras.

Importante: Una opción muy interesante después de la ejecución de este módulo sería guardar los datos en un .mat. 

4.6.2 Bloque de cálculo

5. SpMV

Este módulo realiza la operación Matriz dispersa –Vector en la GPU.

Para el uso de este módulo, deben de estar los datos en MATLAB. Por eso lo recomendable es la obtención de la matriz dispersa paso a paso mediante módulos o con el módulo *automatic1-2-3* o cargar desde un fichero .mat.

Es necesario tener el vector a multiplicar antes de la ejecución de MAT-ELLR-T, porque se pedirá por pantalla Figura 4.16.

Inicializa la GPU, por si no hubiera sido inicializada en algún módulo previo.

```
inicializarGPU(device)
aux=input(' Put the variable to multiplicate ');
u=aux;
[salida]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,u,device,BS,T,tam);
disp('Name of output : Salida ')
```

Figura 4. 16 Uso de la función SpMV

El resultado será el vector *Salida*.

Importante: El vector a multiplicar debe ser una columna de tamaño nrows (1, nrows), si no es así dará fallo. Un error muy común es tener el vector como una fila (nrows, 1), de ser así, basta con hacerle la traspuesta al vector, *vector=vector'*.

Si una vez realizado una multiplicación, no es aconsejable realizar un clear sin destruir los arrays persistentes, porque dará fallo el sistema antes de lanzr MAT-ELLRT.

6. CalculateCG

El módulo de cálculo del gradiente conjugado ejecuta un algoritmo para resolver numéricamente los sistemas de ecuaciones lineales.

Para el uso de este módulo es necesario disponer del vector resultado en el sistema.

En MAT-ELLR-T va incluido la función conjgrad.m, que es un archivo MATLAB que realiza la operación y tiene integrado la rutina SpMV para acelerar las operaciones Matriz dispersa –Vector.

En este módulo no se puede ejecutar directamente con los datos obtenidos de los otros módulos porque, la ejecución de este módulo es necesario que la Matriz esté multiplicada por su traspuesta A=A*A';

Por eso se ha creado un nuevo módulo (*module_LoadMatrix_cg*) que es una copia modificada de *module_LoadMatrix* Figura 4.17.

```
%Modificación para el cálculo CG
A=sparse(i,j,value,nrows,ncols);
A = A * A';
[i,j,value]=find(A);
nonull=nnz(A);
clear A;
```

Figura 4. 17 Preprocesado de la matriz para poder realizar el CG.

Esta variación está al final de los ficheros *simetría_N_cg* y en *simetría_S_cg.*

Para el usuario que quiera utilizarla basta con cargar los datos ejecutando module_LoadMatrix_cg el resto de la secuencia permanecería intacta. La secuencia para el cálculo de CG Figura 4.18 sería:

```
module_LoadMatrix_cg
module_configurationGPU
ELLRT
aux=input(' Put the variable to multiplicate ');
vcg=aux;
[result]= conjgrad(nrows,ncols,BA,BIndex,BCol,vcg,device,BS,T,tam);
disp(' ')
disp('Name of output: result ')
```

Figura 4. 18 Uso de la función conjgrad (integra la rutina SpMV para el cálculo cg)

Este archivo es el claro ejemplo de la flexibilidad y potencia de MAT-ELLR-T.

Se puede adaptar al formato necesario en un lenguaje de alto nivel. La variable *vcg* corresponde al vector que se desea calcular el CG.

La siguiente imagen muestra la facilidad de la integración de la función *calcular_spmv* Figura 4.19.

```
function [x] = conjgrad(nrows,ncols,BA,BIndex,BCol,salida,device,BS,T,tam)

% By Yi Cao at Cranfield University, 18 December 2008.
%
tol=10^-10;


    [rb]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,salida,device,BS,T,tam);
    r=salida+rb;
    y = -r;

    [z]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,y,device,BS,T,tam);
    s = y'*z;
    t = (r'*y)/s;
    x = -salida + t*y;

     for k = 1:100;
       r = r - t*z;
       if( norm(r) <tol )
            return;
       end
       B = (r'*z)/s;
       y = -r + B*y;
        [z]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,y,device,BS,T,tam);

       s = y'*z;
       t = (r'*y)/s;
       x = x + t*y;
    end


end
```

Figura 4. 19 Función conjgrad con la integración de la rutina SpMV

Donde se ha sustituido del archivo original las operaciones con la matriz.

Donde era z= A*y, por:

[z]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,y,device,BS,T,tam);

Dónde Z es el vector resultante de la multiplicación entre la matriz en el formato ELLR-T (BA,Bindex,BCol) por Y, el vector a multiplicar.

Importante: La matriz estará alojada en la memoria GPU de forma persistente, es importante no realizar el comando *clear* de MATLAB una vez ejecutado este módulo porque desvinculará los punteros de MATLAB a la GPU.

7. SpMV Basic

Este módulo permite realizar la operación SpMV sin persistencia de datos en la GPU.

Este módulo tiene la ventaja con respecto al de la persistencia de datos el ahorro de memoria RAM, debido a la realización del formato ELLR-T de la matriz de forma interna. El inconveniente es que debe realizar el formato ELLR-T en cada llamada, lo que ralentiza considerablemente su uso.

Por eso, la utilización de este módulo está destinada a cálculos aislados, con pocas iteraciones. Otra ventaja es que con matrices grandes gestiona mejor la memoria que MATLAB, por lo que tiene un rendimiento mayor.

Para su utilización solo sería necesario ejecutar LoadMatrix e indicar T y BS de forma manual, o ejecutar configurateGPU y obtenerlos de forma automática Figura 20.

```
inicializarGPU(device)
module_LoadMatrix
module_configurationGPU

[salida]=spmvELLRTdp (i,j,value,nrows,ncols,nonull,w,T,device,B);
disp('Name of output : Salida ')
```

Figura 4. 20 Código MATLAB para el cálculo de SpMV utilizando MAT-ELLRT

Otro ejemplo de facilidad y adaptabilidad de MAT-ELLR-T.

Otra ventaja del uso de este módulo es que no es necesario destruir la persistencia si se efectúan operaciones SpMV con matrices distintas, por lo que se puede utilizar tanta veces como se quiera y cambiar la matriz sin ningún problema.

Importante: No es necesaria la ejecución de ELLR-T.

8. Exit

Esta función devuelve el control de la ejecución a MATLAB.

5. Evaluaciones

5.1 Introducción

El objetivo de este apartado es la evaluación con la rutina SpMV en un entorno de prueba y en un entorno real.

- Evaluación de SpMV: Es un entorno de prueba donde se realiza una multiplicación iterativa de las matrices a evaluar por un vector aleatorio de tamaño ncols. La multiplicación se realiza 100 veces cambiando el vector a multiplicar en cada iteración.

Salida=A*salida;

- Evaluación de CG: Es un entorno real donde se evaluará la rutina CG (Conjugate gradients squared) de MATLAB contra la rutina congrad (rutina modificada con la integración de SpMV), con un número de 100 iteraciones y una tolerancia de 10^-10.

5.2 Recursos para la evaluación

Plataforma DaVinci,

- 2 Intel Xeon (4 cores) E5640 2.67 GHz y 16 GB de RAM
- 1 GPU Tesla C2050 (488 cores) con 6Gb de memoria RAM DDR5.
- CUDA (Driver + Toolkit + SDK) 4.2.9

5.3 Matrices a evaluar

Las matrices para la evaluación han sido obtenidas de la plataforma MatrixMarket [15] elegidas de distintas disciplinas cientificas.

La plataforma MatrixMarket es un repositorio de test de datos usado para la comparativa de diversos estudios de algoritmos en el algebra dispersa.

Estas son las matrices que se han decidido evaluar tabla 5.1, ordenadas por los elementos no nulos manera ascendente.

Matriz	Simétrica	N	Nz	Disciplina
rbs480a	No	480	17.088	Cinemática robótica
qh1484	No	1484	6110	Sistema de energía hidroeléctrica
dw2048	No	2048	10114	Guía de onda dieléctrica
bcsstk24	Si	3562	159910	Ingeniería estructural BCS
e20r4000	No	4241	131556	Cavidad de impulso
gemat12	No	4929	33111	Flujo de potencia sistema bus 2400
dw8192	No	8192	41746	Guia de onda dieléctrica
pdb1HYS	Si	36417	4344766	Banco de proteinas
rma10	No	46835	2374001	FEM/Harbor
qcd5_4	No	49152	1916928	QCD
cant	Si	62451	4007384	FEM/Cantiveler
consph	Si	83334	6010481	FEM/Spheres
cop20k_A	Si	121192	2624331	FEM/Accelerator
shipsec1	Si	140874	7813404	FEM/Ship
mac_econ	No	206500	1273389	Modelo macro económico
pwtk	Si	217918	11634425	Túnel del viento
mc2depi	No	525825	2100225	Epidemiología

Tabla 5. 1 Tipo y formato de las matrices

Se han elegido una variedad de matrices heterogénea, donde, la proporción de número de filas y elementos no nulos no es lineal. Así se puede comprobar cómo se comporta MAT-ELLT con todo tipo de matrices.

Se puede apreciar que el uso de la algebra dispersa está en un gran número de disciplinas científicas, por lo que, MAT-ELLR-T tiene un gran número de áreas en las que se pudiera integrar.

5.4 Evaluaciones de SpMV

El objetivo de esta evaluación es ver la respuesta que tiene MAT-ELLR-T con las matrices a evaluar sin alterar su estructura Tabla 5.2.

Matrices	Simétrica	N	T. MATLAB	BS	T	T.GPU	Ganancia
rbs480a	No	480	0,018	64	16	0,1	0,18
qh1484	No	1484	0,009	64	2	0,1	0,09
dw2048	No	2048	0,0085	64	2	0,045	0,19
bcsstk24	Si	3562	0,09	64	2	0,11	0,82
e20r4000	No	4241	0,12	128	8	0,12	1,00
gemat12	No	4929	0,02	64	4	0,11	0,18
dw8192	No	8192	0,031	64	1	0,11	0,28
pdb1HYS	Si	36417	4,5	256	8	0,79	5,70
rma10	No	46835	1,9	128	8	0,46	4,13
qcd5_4	No	49152	3	128	1	0,46	6,52
cant	Si	62451	3,57	128	4	0,67	5,33
consph	Si	83334	6,4	128	1	0,97	6,60
cop20k_A	Si	121192	5,67	64	1	1,39	4,08
shipsec1	Si	140874	7	512	2	1,22	5,74
mac_econ	No	206500	2,59	128	4	1,13	2,29
pwtk	Si	217918	8,29	64	1	1,51	5,49
mc2depi	No	525825	2,17	128	1	1,08	2,01

Tabla 5. 2 Evaluaciones de 100 iteraciones con SpMV sin alterar las matrices

En esta tabla podemos observar, datos relevantes sobre las matrices:

- **Sim**: Indica si la matriz es simétrica.
- **N**: Número de filas.
- **Nz**: Número de elementos o nulos.
- **T.MATLAB**: Tiempo de ejecución en MATLAB.
- **BS**: Tamaño del bloque dado por el módulo configurateGPU.
- **T:** Número de *thread* dado por el módulo configurateGPU.
- **T.GPU**: Tiempo de ejecución con SpMV.
- **Ganancia**: Muestra la ganancia entre T.MATLAB y T.GPU.

Se puede apreciar en la Figura 5.1 que con matrices pequeñas (N < 206500 y Nz<1273389) y muy dispersas gana MATLAB. Esto es debido a que la penalización de la copia de los datos en la memoria de la GPU a las comunicaciones, y a la poca explotación de la arquitectura GPU cuando la matriz es muy dispersa o de reducidas dimensiones. Por eso en estos casos, no es aconsejable su utilización.

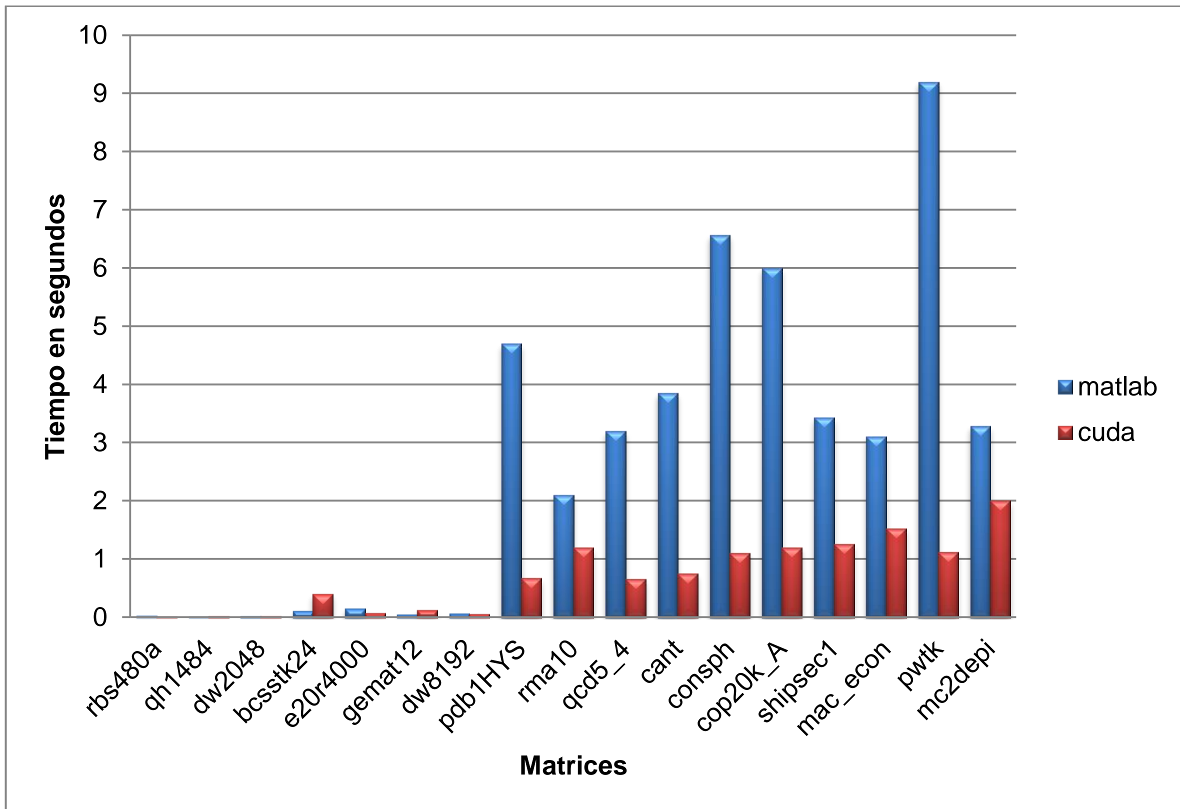


Figura 5. 1 Gráfica del cálculo de 100 iteraciones con SpMV en tiempo de ejecución en segundos CPU con Intel Xeon (4 cores) E5640 y GPU con Tesla C2050 (488 cores)

Con matrices medianas o grandes, se puede observar en la figura 5.2 que la función SpMV obtiene una ganancia mínima de 2X y máxima de 7X, por lo tanto, es aconsejable su uso. Cuanto mayor sea el número de multiplicaciones que se realice la ganancia en GPU es mayor debido a que el porcentaje de la penalización de comunicaciones frente al tiempo total de computación es menor, esto es debido a la persistencia de datos en GPU que una vez cargada la matriz en formato ELLR-T en la memoria de la GPU no es necesario volver a copiarla, por lo tanto, el tiempo restante solo es cálculo puro en la GPU y volcado del resultado, mientras que el tiempo en MATLAB para realizar una multiplicación es constante y no tiene ninguna ventaja en procesos iterativos ya que siempre tarda lo mismo por multiplicación.

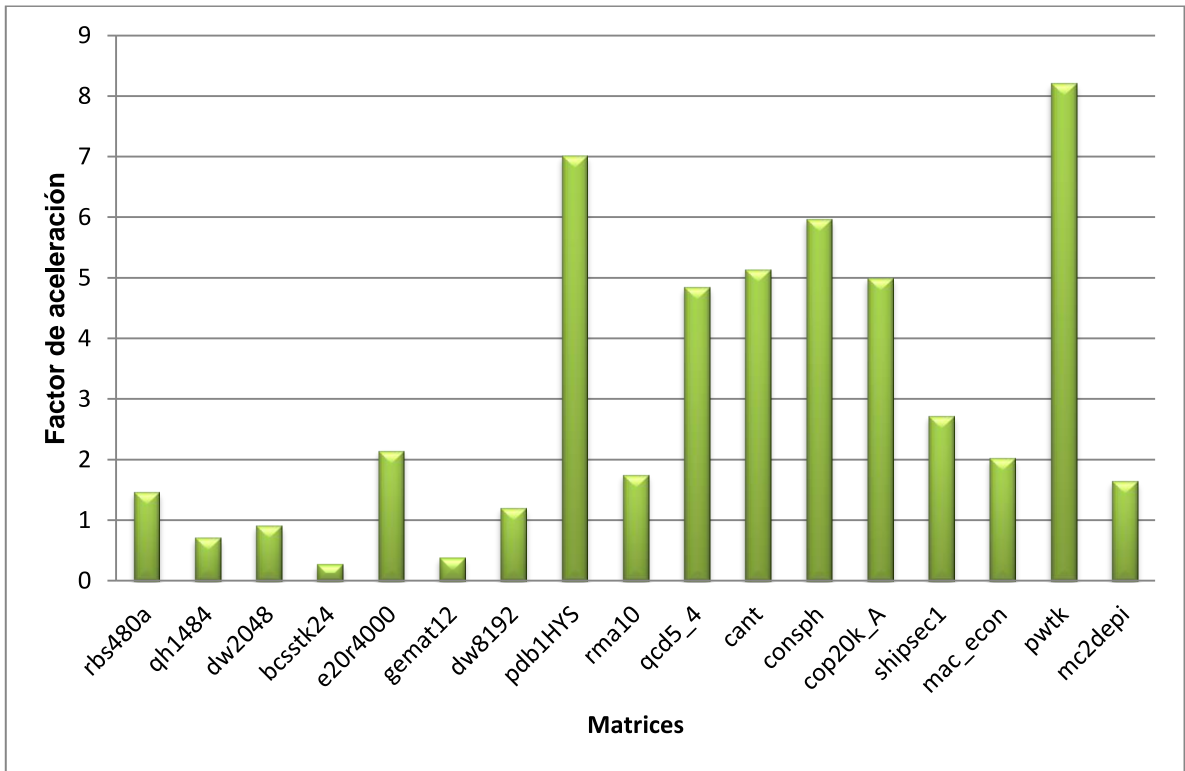


Figura 5. 2 Gráfica de ganancias de MAT-ELLRT frente al SpMV de MATLAB ejecutadas en una GPU Tesla C2050 (488 cores) y una CPU Intel Xeon (4 cores) E5640

5.5 Evaluaciones de CG

El objetivo de esta evaluación es ver la respuesta que tiene la integración de la rutina SpMV en un método numérico para la solución de sistemas de ecuaciones. Centramos nuestra atención en el método del Gradiente Conjugado que es un método iterativo, especialmente adecuado para la solución de sistemas dispersos que son demasiado grandes para ser tratados por métodos directos como la descomposición de Cholesky cuya matriz es simétrica y definida positiva. Tales sistemas surgen frecuentemente cuando se resuelve numéricamente las ecuaciones en derivadas parciales [16].

Los valores han sido tomados desde MATLAB con la función CG, con 100 iteraciones y una tolerancia de error de 10^{-10}.

Para realizar este método se necesita que la matriz sea simétrica definida positivas, por eso es necesario realizar un preproceso de la matriz. $A=A*A^T$ Figura 5.3, esto no hace que sea definida positiva pero sí que sea simétrica y se pueda ejecutar el algoritmo y ya que dependa de la distribución y tipo de la matriz.

```
%Modificación para el cálculo CG
A=sparse(i,j,value,nrows,ncols);
A = A * A';
[i,j,value]=find(A);
nonull=nnz(A);
clear A;
```

Figura 5. 3 Modificación de la matriz para la utilización de CG

Una vez realizado esta multiplicación, la matriz cambia de distribución por lo que es necesario ejecutar configutareGPU nuevamente para ejecutarlo con la mejor configuración Tabla 5.3.

Ese es el motivo para que la configuración de BS y T sean distintas a la tabla anterior Tabla 5.2.

Matrices	Simétrica	N	T.MATLAB	BS	T	T.GPU	Ganancia
rbs480a	No	480	0,025	256	32	0,017	1,47
qh1484	No	1484	0,015	256	2	0,021	0,71
dw2048	No	2048	0,02	256	4	0,022	0,91
bcsstk24	Si	3562	0,11	256	8	0,4	0,28
e20r4000	No	4241	0,15	256	16	0,07	2,14
gemat12	No	4929	0,046	256	8	0,12	0,38
dw8192	No	8192	0,065	256	2	0,054	1,20
pdb1HYS	Si	36417	4,7	256	16	0,67	7,01
rma10	No	46835	2,1	128	8	1,2	1,75
qcd5_4	No	49152	3,2	128	1	0,66	4,85
cant	Si	62451	3,85	256	8	0,75	5,13
consph	Si	83334	6,57	256	8	1,1	5,97
cop20k_A	Si	121192	6	64	1	1,2	5,00
shipsec1	Si	140874	3,43	256	8	1,26	2,72
mac_econ	No	206500	3,1	512	16	1,53	2,03
pwtk	Si	217918	9,2	256	8	1,12	8,21
mc2depi	No	525825	3,29	256	1	2	1,65

Tabla 5. 3 Evaluación de CG con 100 iteraciones

En la Figura 5.4 se puede apreciar que al realizar $A=A*A^T$ la distribución de la matriz varia porque con el mismo número de filas y columnas aumentan el número de elementos no nulos. Eso beneficia a la rutina SpMV porque permite aprovechar mejor la arquitectura de la GPU teniendo tamaños de bloques mayores y aumentando el número de *thread* por filas.

Para MATLAB esta modificación de la matriz es muy costosa computacionalmente debido a que aumenta el número de elementos no nulos de la matriz, lo que equivale, a más operaciones y mayor consumo de recursos. Un inconveniente de MATLAB es que cuando se queda sin recursos aumenta el tiempo de ejecución.

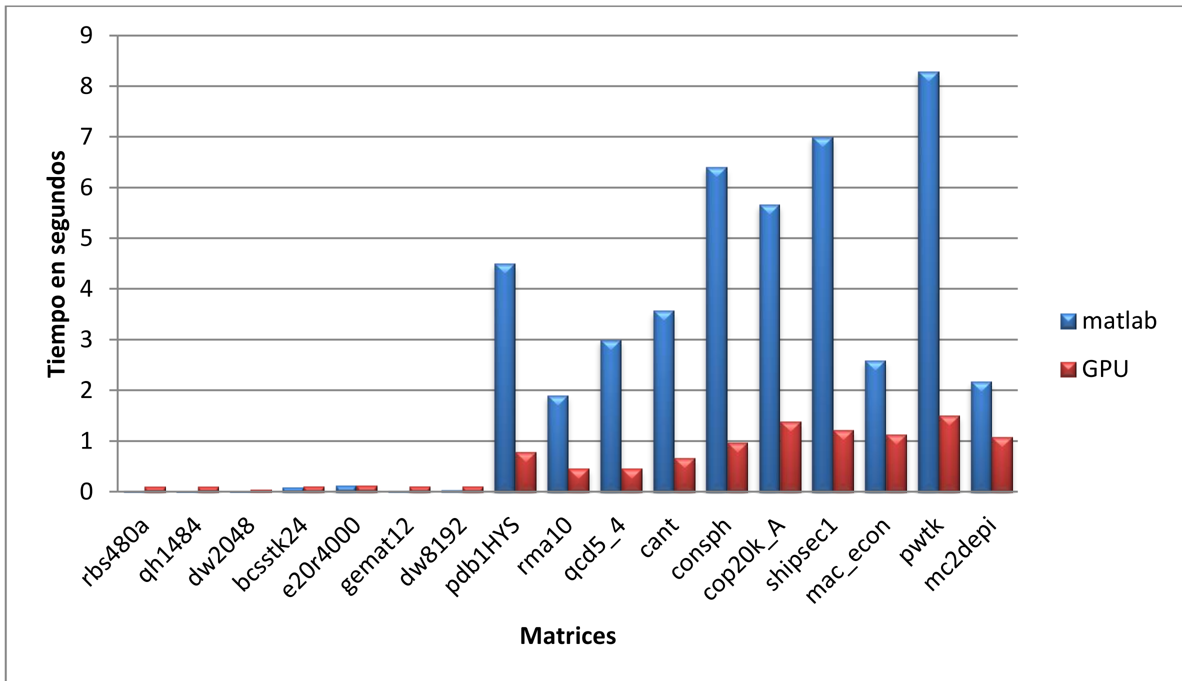


Figura 5. 4 Gráfica de 100 iteraciones de CG en segundos en CPU con Intel Xeon (4 cores) E5640 y GPU con Tesla C2050 (488 cores)

Este es uno de los motivos principales de la ganancia de la GPU, MATLAB no tiene penalización alguna por el acceso a variables desde MATLAB, pero la multiplicación tiene un coste constante por iteración. La GPU por lo contrario tiene una penalización en la comunicación y en la copia de datos a la GPU, pero esta penalización solo ocurre la primera vez, cuando se realiza la persistencia de los datos una vez alojados los datos en la GPU, la penalización de comunicación es casi nula, en la multiplicación tiene también un coste constante por iteración pero gracias a la potencia de la GPU es muy inferior al tiempo de MATLAB en CPU. Ese es el motivo principal de la ganancia, el uso de la potencia de la GPU desde un programa de alto nivel como MATLAB Figura 5.5.

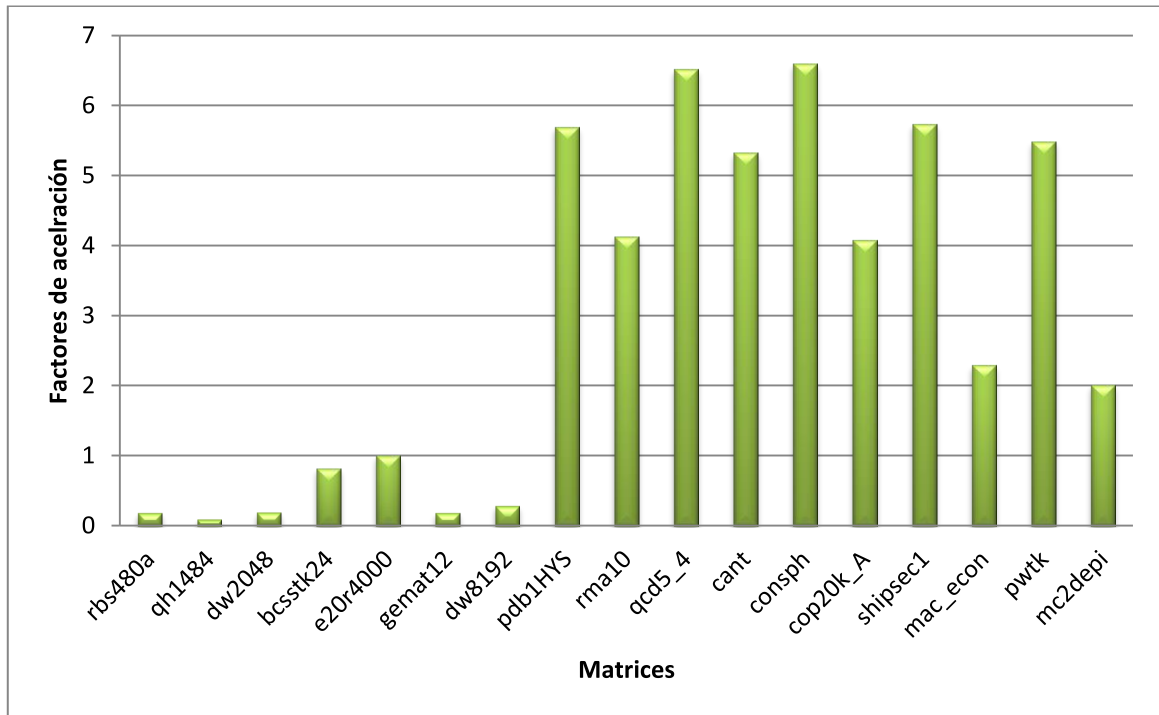


Figura 5. 5 Gráfica de ganancia con 100 iteraciones de CG de MAT-ELLRT frente al SpMV de MATLAB ejecutadas en una GPU Tesla C2050 (488 cores) y una CPU Intel Xeon (4 cores) E5640

El único inconveniente que para el uso de MAT-ELLRT o de la rutina SpMV es necesario realizar un preproceso de los datos para que estén en el formato adecuado para su uso, eso requiere tiempo y recursos. Por eso, se aconseja para matrices de uso habitual tener los datos guardados en formato .mat.

6. Conclusiones y trabajos futuros

La importancia de la operación SpMV ha sido constatada a lo lardo de todo este proyecto y de la necesidad de llevarla al ámbito científico de una manera sencilla para el usuario como es MAT-ELLRT.

MAT-ELLRT es una librería modular integrada en MATLAB que permite al usuario realizar operaciones de carga de datos, configuración de la GPU, obtención del formato ELLR-T de una matriz y realizar la operación SpMV (con persistencia de datos en la GPU) aprovechando el potencial de la GPU en un entorno familiar como MATLAB obtenido un rendimiento muy superior a las herramientas que proporciona MATLAB para las operaciones con algebra dispersa. MAT-ELLRT permite además su integración invocando las llamadas desde MATLAB

MAT-ELLRT también incorpora la posibilidad de realizar la operación CG utilizando la función SpMV. Esto es un claro ejemplo de la versatilidad que tiene la librería, porque se puede observar la facilidad de integración de la rutina SpMV en cualquier sistema sacándole así el máximo rendimiento a la GPU y obteniendo un tiempo de ejecución menor. Se ha demostrado que MAT-ELLRT con matrices pequeñas no explota todo su potencial debido a la penalización de las comunicaciones entre el *host* y el *device,* mientras que en la CPU con MATLAB no sucede.

Todos los resultados han sido obtenidos desde la plataforma DaVinci donde se ha comparado el tiempo de ejecución de la CPU desde MATLAB con respecto al tiempo de ejecución de su GPU desde la librería MAT-ELLRT en realizar el mismo cálculo obteniendo en la mayoría de las operaciones excelentes resultados en términos de rendimiento, cuando las dimensiones de las matrices son suficientemente elevadas.

Los posibles trabajos futuros en torno a MAT-ELLRT se integrarán otras variantes del formato ELLRT para efectuar SpMV en la GPU. Así por ejemplo merece especial interés el formato Sliced COO [54] que tiene asociado un preproceso de ordenación de las filas de la matriz, de forma que se pueden clasificar distintas partes de la matriz de acuerdo al llenado de las filas y así tratar el relleno de cada conjunto de filas de forma independiente. Este tipo de formatos son especialmente adecuados para matrices cuyo llenado de las filas es muy irregular y el formato ELLPACK contiene un elevado porcentaje de elementos cero de relleno.

Otra línea de trabajo futuro es la integración en el entorno MATLAB de plataformas multiGPU siguiendo la misma metodología desarrollada en este proyecto.

7. Bibliografía

1. Mathworks, “MATLAB, Primer R2013a” http://www.mathworks.es/help/pdf_doc/MATLAB/getstart.pdf.
2. Getting Started Guide. Jacket v1.3 http://hpc.icc.ru/documentation/MATLAB/GettingStartedGuidev1.3.pdf
3. S. Brin, L. Page. "The anatomy of a large-scale hypertextual Web search engine". Computer Networks and ISDN Systems. Vol 30 pp 107–117. 1998
4. S. Tabik, L.F. Romero, E.M. Garzon, and J.I. Ramos. On a model of three-dimensional bursting and its parallel implementation. Computer Physics Communications, Vol. 178, n. 7, pp. 471-485; 2008
5. F. Vazquez, E.M. Garzon and J.J. Fernandez. A matrix approach to tomographic reconstruction and its implementation on GPUs. Journal of Structural Biology, Vol. 170, pp. 146-151. 2010.
6. NVIDIA (2010) Next generation CUDA architecture. Fermi architecture
7. F. Vazquez, J.J. Fernandez, E.M. Garzon. Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach, Parallel Computing, Vol 38(8) pp 408-420 August 2012
8. NVIDIA (2010) CUDA CUSPARSE library. Tech rep. http://www.nvidia.com/content/GTC-2010/pdfs/2070_GTC2010.pdf
9. NVIDIA (2010) Cusp library. Tech rep
10. Choi JW, Singh A, Vuduc R (2010) Model-driven autotuning of sparse matrix–vector multiply n GPUs. In: PPoPP’10, pp 115–126
11. MEX http://www.mathworks.es/es/help/MATLAB/build-cc-mex-files.html
12. Rob H. Bisseling. Parallel Scientific Computation: A Structured Approach using BSP and MPI, Oxford University Press, March 2004.
13. *The Matrix Market.* Available at: http://math.nist.gov/MatrixMarket [June 2009].
14. The University of Florida Sparse Matrix Collection http://www.cise.ufl.edu/research/sparse/matrices/
15. http://math.nist.gov/Matrix/
16. Kendell A. Atkinson (1988), An introduction to numerical analysis (2ª ed.), Section 8.9, John Wiley and Sons. ISBN 0-471-50023-2.
17. http://www.nvidia.es/object/nvidia-kepler-es.html
18. NVIDIA. Programming guide 2.3.1. 2009. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf
19. http://es.playstation.com/ps4/features/techspecs/

20. http://www.microsoftstore.com/store/mseea/es_ES/html/pbPage.PDP/productID.282414600
21. SGI. Opengl reference manual 1.4. http://www.amazon.com/exec/obidos/ASIN/032117383X/khongrou-20.
22. Microsoft. The directx software development kit. http://msdn.microsoft.com/en-us/library/ee416796(v=VS.85).aspx
23. M. A. Gray. Getting started with gpu programming. IEEE Computing in Science & Engineering, 11-4:61_64, 2009.
24. D. B. Kirk and W. W. Hwu. Programming Massively Parallel Processors. Elsevier,2010.
25. http://www.nvidia.es/object/gpu-programming-es.html
26. http://www.mathworks.es/products/MATLAB/?s_cid=books_spot_cat17_products
27. http://www.accelereyes.com/
28. http://wiki.accelereyes.com/wiki/index.php/Jacket_SDK
29. http://www.mathworks.es/es/help/MATLAB/create-mex-files.html
30. https://developer.nvidia.com/magma
31. http://icl.cs.utk.edu/magma/
32. https://developer.nvidia.com/cublas
33. https://developer.nvidia.com/cusp
34. http://www.culatools.com/sparse/
35. NVIDIA, *CUDA Programming guide. Version 1.1*, April, 2009 http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf
36. R.H. Bisseling Parallel Scientific Computation, Oxford Univ. Press, 2004.
37. A.T. Ogielski, W. Aiello Sparse matrix computations on parallel processor arrays SIAM Journal on Scientific Computing,14 (1993) 519–530.
38. S. Toledo Improving the memory-system performance of sparse-matrix vector multiplication IBM Journal of Research and Development,41 (6) (1997) 711–725
39. J. Mellor-Crummey, J. Garvin Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam Intl. J. High Performance Comput. App.,18 (2004) 225–236
40. S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel Optimization of sparse matrix-vector multiplication on emerging multicore platforms Parallel Computing,35 (3) (2009) 178–194
41. C. Lawson, R. Hanson, D. Kincaid, F. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, ACM Trans. Mathematical Software, 5 (1979) 308–325.
42. J. Baldeschwieler and R. Blumofe and E. Brewer ATLAS: An Infrastructure for Global Computing In Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, (1996).
43. NVIDIA, CUDA CUBLAS Library. PG-00000-002.V2.1 September, 2008. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUBLAS_Library_ 2.1.pdf
44. J. Kurzak, W. Alvaro and J. Dongarra Optimizing matrix multiplication for a short-vector SIMD architecture- CELL processor Parallel Computing,35 (3) (2009) 138–150
45. Kronos Group, OpenCL - The open standard for parallel programming of heterogeneous systems http://www.khronos.org/developers/library/overview/opencl_overview.pdf
46. D.R. Kincaid, T.C. Oppe, D.M. Young, ITPACKV 2D User's Guide. CNA-232 1989 http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/
47. http://math.nist.gov/MatrixMarket/info.html
48. NVIDIA. NVIDIA CUDA. CUDA C Best Practices Guide 3.2. 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.
49. http://www.nvidia.es/object/fermi_architecture_es.html
50. http://sites.amd.com/uk/business/it-solutions/compute-intensive-hpc/Pages/compute-intensive-hpc.aspx
51. http://www.intel.com/content/www/us/en/high-performance-computing/server-reliability.html

52. G. Ortega, E. M. Garzón, F. Vázquez and I. García. The BiConjugate gradient method on GPUs. The Journal of Supercomputing. 64:49–58, 2013. DOI: 10.1007/s11227-012-0761-2
53. F. Vázquez, E.M. Garzón, J.A. Martínez, J.J. Fernández Acelerando el producto matriz dispersa vector en GPUs In Actas de las XX Jornadas de Paralelismo, pages 313–318, Coruña, Septiembre 2009.
54. HV. Dang, B. Schmidt. The Sliced COO Format for Sparse Matrix-Vector Multiplication on CUDA-enabled GPUs, Procedia Computer Science, Volume 9:57-66, 2012 DOI: 10.1016/j.procs.2012.04.007

Anexos

Es este apartado están los código más importantes en código MATLAB de MAT-ELLRT. La mayoría de los módulos realizan llamadas a rutinas C.

Código MATLAB del módulo LoadMatrix Simetric_N

```
nombrematriz=input('Put the name of  Matrix (NO symmetric): ','s');
%disp('carga de datos')

data=load(nombrematriz);
nrows=data(1,1);
ncols=data(1,2);
null=data(1,3);

data(1,:)=[ ];

i2=data(:,1);
j=data(:,2);

value=data(:,3);
clear data;

A=sparse(i2,j,value,nrows,ncols);
A = A * A';

[i,j,value]=find(A);
nonull=nnz(A);
%clear A;
disp('Wait .....')
disp('_____________')
```

Código MATLAB del módulo LoadMatrix Simetric_S

```
nombrematriz=input('Put the name of Symmetric Matrix:  ','s');
[fsalida]=simetria(nombrematriz);
disp(' ')
disp('Created symmetric file ')
disp(fsalida)
disp('Wait .....')
disp('_____________')

data=load(fsalida);
nrows=data(1,1);
ncols=data(1,2);
```

```
null=data(1,3);
data(1,:)=[ ];

i2=data(:,1);
j=data(:,2);
value=data(:,3);
clear data;

A=sparse(i2,j,value,nrows,ncols);
A = A * A';
[i,j,value]=find(A);
nonull=nnz(A);
clear A;
```

Código MATLAB del módulo Load Matrix Simetric_N_cg

```
nombrematriz=input('Put the name of  Matrix (NO symmetric): ','s');
disp('carga de datos')

data=load(nombrematriz);
nrows=data(1,1);
ncols=data(1,2);
nonull=data(1,3);

data(1,:)=[ ];

i=data(:,1);
j=data(:,2);

value=data(:,3);
clear data;
%value=2*rand(null,1);
A=sparse(i,j,value,nrows,ncols);
A = A * A';


[i,j,value]=find(A);
nonull=nnz(A);
%clear A;
disp('Wait .....')
disp('____________')
```

Código MATLAB del módulo LoadMatrix Simetric_S_cg

```
nombrematriz=input('Put the name of Symmetric Matrix:  ','s');
[fsalida]=simetria(nombrematriz);
disp(' ')
disp('Created symmetric file ')
disp(fsalida)
disp('Wait .....')
disp('____________')

data=load(fsalida);
nrows=data(1,1);
```

```
ncols=data(1,2);
nonull=data(1,3);

%Matrices vacias, cargar en matlab y descomentar
%val=2*rand(1,null);
%value=val';

data(1,:)=[ ];

i=data(:,1);
j=data(:,2);
%%comentar cuando es matriz vacia
value=data(:,3);
clear data;

%Modificación para el cálculo CGS
A=sparse(i,j,value,nrows,ncols);
A = A * A';
[i,j,value]=find(A);
nonull=nnz(A);
%clear A;
```

Código MATLAB del módulo Configuration GPU

```
disp('Configurate GPU')
disp('________________')
device=input('Put the device number: ');
if v1>0;
disp('If you wish chance the matrix, run LoadMatrix ')
[BS,T]=ModelFermi(i,j,value,nrows,ncols,nonull,device);
else
   disp('there isn´t a matrix loaded ')
   modulo_carga_datos
   [BS,T]=ModelFermi(i,j,value,nrows,ncols,nonull,device);
End
```

Código MATLAB del módulo Matrix ELLR-T

```
disp('MatrixELLRT')
[tam] = calculo_formato1(i,j,value,nrows,ncols,nonull,T,BS);
[BA, BIndex, BCol]=calculo_formato2 (i,j,value,nrows,ncols,nonull,T,BS,tam);
disp('Created Arrays')
```

Código MATLAB del módulo Kernel SpMV

```
[salida]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,u,device,BS,T,tam);
```

Código MATLAB del módulo module_CG_spmv

```
module_LoadMatrix_cg
module_configurationGPU
ELLRT
%vcg=2*rand(nrows,1);
aux=input(' Put the variable to multiplicate ');
vcg=aux;
[result]= conjgrad(nrows,ncols,BA,BIndex,BCol,vcg,device,BS,T,tam);
disp(' ')
disp('Name of output: result ')
```

Código MATLAB del módulo CG original CPU

```
function [x] = conjgrad2(A,b)
% By Yi Cao at Cranfield University, 18 December 2008.
%
tol=10^-10;

   r = b + A*b;
   y = -r;
   z = A*y;
   s = y'*z;
   t = (r'*y)/s;
   x = -b + t*y;

   for k = 1:100;
     r = r - t*z;
     if( norm(r) < tol )
         return;
     end
     B = (r'*z)/s;
     y = -r + B*y;
     z = A*y;
     s = y'*z;
     t = (r'*y)/s;
     x = x + t*y;
   end

end
```

Código MATLAB del módulo CG utilizado para la integración de la rutina SpMV, GPU.

```
function [x] = conjgrad(nrows,ncols,BA,BIndex,BCol,salida,device,BS,T,tam)

% By Yi Cao at Cranfield University, 18 December 2008.
%
tol=10^-10;
   [rb]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,salida,device,BS,T,tam);
   r=salida+rb;
   y = -r;
   [z]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,y,device,BS,T,tam);
```

```
    s = y'*z;
    t = (r'*y)/s;
    x = -salida + t*y;

     for k = 1:100;
       r = r - t*z;
       if( norm(r) <tol )
            return;
       end
       B = (r'*z)/s;
       y = -r + B*y;
        [z]=calcular_spmv (nrows,ncols,BA,BIndex,BCol,y,device,BS,T,tam);

       s = y'*z;
       t = (r'*y)/s;
       x = x + t*y;
    end
end
```