Trabajo Fin de Grado
Grado en Ingeniería en Tecnologías Industriales

Implementación paralela de algoritmos de control y optimización en CUDA

Autor: Eduardo Mayoral Briz
Tutor: Daniel Rodríguez Ramírez

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado
Grado en Ingeniería en Tecnologías Industriales

Implementación paralela de algoritmos de control y optimización en CUDA

Autor:
Eduardo Mayoral Briz

Tutor:
Daniel Rodríguez Ramírez
Profesor titular de Universidad

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Trabajo Fin de Grado: Implementación paralela de algoritmos de control y optimización en CUDA

Autor: Eduardo Mayoral Briz

Tutor: Daniel Rodríguez Ramírez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Me gustaría agradecer a toda mi familia el apoyo y cariño recibido durante estos años, sin el cual no podría haber llegado a escribir estas líneas.

Gracias a todos mis amigos y compañeros de estudio, que han estado en los malos y en los buenos momentos durante estos duros años.

También agradezco la ayuda a mi tutor, Daniel, quien me ha orientado durante el trabajo y me ha abierto las puertas a nuevos conocimientos.

Eduardo Mayoral Briz

Sevilla, 2018

Resumen

Tradicionalmente, el desarrollo de algoritmos siempre ha estado sujeto a la recepción secuencial de instrucciones de los ordenadores. Aunque en ciertas máquinas ha sido posible elaborar programas con cierto nivel de concurrencia, nunca se han alejado demasiado de lo conocido hasta entonces.

De la mano de NVIDIA, aparece en el año 2007 CUDA, una nueva plataforma de computación en paralelo. Introducía un nuevo género en la informática, GPGPU, del inglés *General Purpose Graphics Processing Units*. Tal y como su nombre indica, trataba de acercar al público las cualidades de cómputo de una GPU, para tareas comunes más allá del procesamiento gráfico.

En primer lugar, en este trabajo se trata de reunir las características principales de la plataforma. Se intenta hacer una descripción de su modelo de programación y de su arquitectura. A su vez, se hace un seguimiento para establecer una configuración básica de CUDA en un sistema operativo Microsoft Windows.

A continuación, se propone la paralelización de un algoritmo de factorización de Cholesky, enfocado a la resolución de sistemas de ecuaciones. Aquí se abordan dos métodos diferentes, el uso de lenguaje C estándar en la CPU y de CUDA en la GPU. Se profundiza en el comportamiento del programa cuando se ejecuta en la unidad de procesamiento gráfico, así como se realiza una comparación entre ambos métodos.

Por último, se desarrolla otro algoritmo para el ajuste de una recta por mínimos cuadrados, usando el *bagging* como técnica estadística, el cual introduce cierto nivel de paralelismo. También se verán las posibles ventajas e inconvenientes del uso de este método en este contexto.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 DESCRIPCIÓN DE CUDA

1.1 Descripción general

CUDA (Compute Unified Device Architecture) es una arquitectura de cálculo paralelo desarrollada por NVIDIA, con el fin de sacar el máximo provecho de las GPUs (Graphics Processing Unit) en labores más allá de las tareas gráficas que siempre han desempeñado. Desde su creación, ha proporcionado un incremento en el rendimiento de diferentes aplicaciones enfocadas a vídeo, biología, química computacional, machine learning o dinámica de fluidos, entre otros.

Gracias a CUDA, en las tareas anteriormente mencionadas, y en muchas otras, se ha optado por utilizar una GPU como alternativa a la tradicional CPU, gracias a su gran capacidad de programación en paralelo. Mientras que en una CPU actual encontramos un numero de núcleos en torno a 4-8, en las GPUs más modernas podemos llegar a ver hasta 4000 núcleos. Estos, aunque tienen una capacidad computacional mucho menor, se benefician de su trabajo en paralelo para superar a la CPU en determinados tipos de tarea.

El lenguaje de programación puede ser C/C++, Fortran, o incluso algunas directivas de lenguaje abierto como OpenACC. Dispondremos de dos partes a la hora de programar, el llamado *host* (anfitrión, asociado a la CPU) y el *device* (dispositivo, relacionado con la GPU).

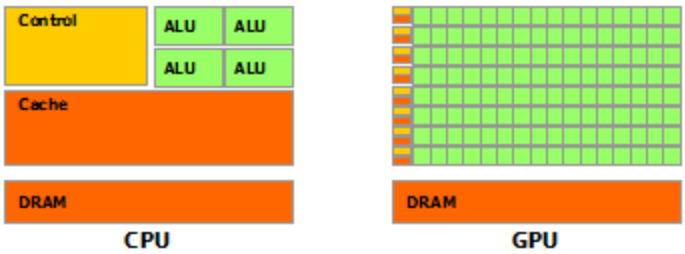

Figura 1-1. Arquitectura de CPU y GPU.

1.2 Modelo de programación

1.2.1 Kernels

Los kernels toman el papel de una función de C, pero son ejecutados por la GPU. Por lo general, se ejecutan desde el *host*, aunque también puede hacerse desde el *device* mediante el uso del paralelismo dinámico. Se ejecutan un determinado número de veces en paralelo, a diferencia de una función en la CPU, que solo se ejecuta una vez. Definimos un kernel anteponiéndole la declaración `__global__`, y lo llamamos especificando los parámetros de ejecución, el número de bloques e hilos que se utilizarán en su ejecución, entre <<<…>>>, de la forma <<< *número de bloques*, *número de hilos* >>>. Dichos parámetros pueden ser de tipo `int` o `dim3`.

A continuación, se propone un kernel de ejemplo, donde se realiza la suma de dos vectores en paralelo:

Ejemplo. *Suma de dos vectores en paralelo.*

```
// Definicion del kernel
__global__ void VecAdd(float* A, float* B, float* C)
{
	int i = threadIdx.x;
	C[i] = A[i] + B[i];
}

int main()
{
```

```
        ...
        // Invocacion del kernel con N hilos
        VecAdd << <1, N >> >(A, B, C);
        ...
}
```

1.2.2 Jerarquía de hilos

Cada uno de los kernels mencionados anteriormente se ejecutarán en tantos hilos como se desee. Es tarea del programador establecer el número de hilos, así como la organización y estructura de estos a la hora de lanzar el kernel.

Los hilos (*threads)* forman parte de bloques *(blocks)*, mientras que un grupo de estos constituye una rejilla *(grid)*. Un hilo puede ser definido con un índice de hasta 3 dimensiones, al igual que cada uno de los bloques que contienen a estos hilos. Este índice se representa con `threadIdx`, y lo acompañamos de la dimensión correspondiente usando `threadIdx.x` (por ejemplo, para la dimensión x). De igual forma indexamos los bloques, con la variable `blockIdx`.

Se pueden definir tantos índices como se deseen, y el kernel trabajará en todos y cada uno de los hilos indicados. Sin embargo, existen limitaciones en el número de hilos por bloque, así como en el número de bloques por rejilla, dependiendo de los recursos de la GPU que se trate. Por lo general, se tienen como máximo 1024 hilos por bloque. Aun así, existe la posibilidad de lanzar varios bloques de igual cantidad de hilos para alcanzar un número de hilos en ejecución mayor al límite mencionado anteriormente.

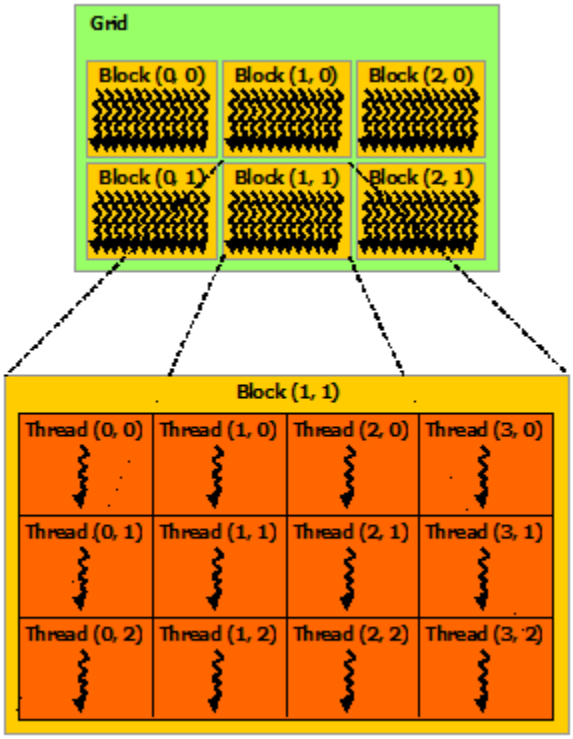

Figura 1-2. Rejilla de bloques de hilos bidimensional.

A la hora de programar, podemos conocer este límite de hilos por bloque con la variable `blockDim`. Al igual que las anteriores, irá seguida de su dimensión (`blockDim.x`), ya que en cada dimensión los bloques tendrán diferentes limitaciones.

Un kernel de ejemplo para la suma de dos matrices, usando dos índices, uno para filas y otro para columnas, es el siguiente:

Ejemplo. *Suma de dos matrices en paralelo con dos índices.*

```
// Definicion del kernel
__global__ void MatAdd(float A[N][N], float B[N][N],
       float C[N][N])
{
       int i = threadIdx.x;
       int j = threadIdx.y;
       C[i][j] = A[i][j] + B[i][j];
}

int main()
{
       ...
       // Invocacion del kernel con un bloque de N*N hilos
       int numBlocks = 1;
       dim3 threadsPerBlock(N, N);
       MatAdd << <numBlocks, threadsPerBlock >> >(A, B, C);
       ...
}
```

Cada uno de los hilos se ejecutará de forma independiente, aunque pueden compartir datos de memoria si están en el mismo bloque. Esto es gracias a una memoria compartida, *shared memory*, de una latencia muy baja, que coordina los accesos de cada hilo a la memoria. Debemos hacer uso de la llamada `__syncthreads()`, que actúa como barrera para que los hilos se coordinen, y no se de la operación por finalizada hasta que acaben todos los hilos.

1.2.3 Jerarquía de memoria

En CUDA, los hilos pueden hacer accesos a diferentes tipos de memoria. Cada hilo tiene una **memoria local** privada asociada, mientras que cada bloque de hilos tiene una **memoria compartida** que es visible ante todos los hilos que componen el bloque. Todos los hilos de este bloque tendrán acceso a dicha memoria compartida. Además, tienen acceso a los espacios de memoria de sólo lectura *constant* y *texture*.

También tienen acceso a la memoria global de la GPU, la memoria asociada al *device*. Sin embargo, aunque está a disposición de cualquier hilo, los accesos son más lentos que si se usa la memoria compartida. El uso de dicha memoria *shared* puede darnos en ocasiones un rendimiento mayor que la global, aunque añadirá cierta complejidad al problema, llegando a casos en los que su uso es inviable, debido a su pequeño espacio.

Para alojar espacio en la memoria global, disponemos de funciones como `cudaMalloc()` o `cudaFree()`, análogas a las ya conocidas `malloc()` o `free()` en C/C++ estándar. Si queremos transferir datos desde el *host* al *device*, debemos usar alguna función del tipo `cudaMemcpy()`. Esto es algo fundamental, ya que no podemos acceder directamente al contenido de la memoria global del *device* sin antes transferirla al *host*.

Es interesante mencionar que algunas funciones como `cudaMemcpy()` son síncronas, es decir, no vuelven para seguir ejecutando instrucciones hasta que esta finalice. En muchas ocasiones, estaremos desaprovechando tiempo si las siguientes instrucciones en la cola de ejecución no tienen relación con esta y están paradas. Para ello se usan funciones asíncronas, como en este caso `cudaMemcpyAsync()`. Esta función hace la transferencia de memoria que se le indique, mientras ya ha dado paso a la siguiente instrucción.

Sin embargo, el uso de estas funciones asíncronas tiene algunos inconvenientes. Por un lado, el uso de cualquier dato que dependa de una función que aún no ha terminado puede desencadenar errores. Por otro, el programa puede volverse más complejo, ya que en muchos casos es necesario el uso de objetos auxiliares, como los *streams*.

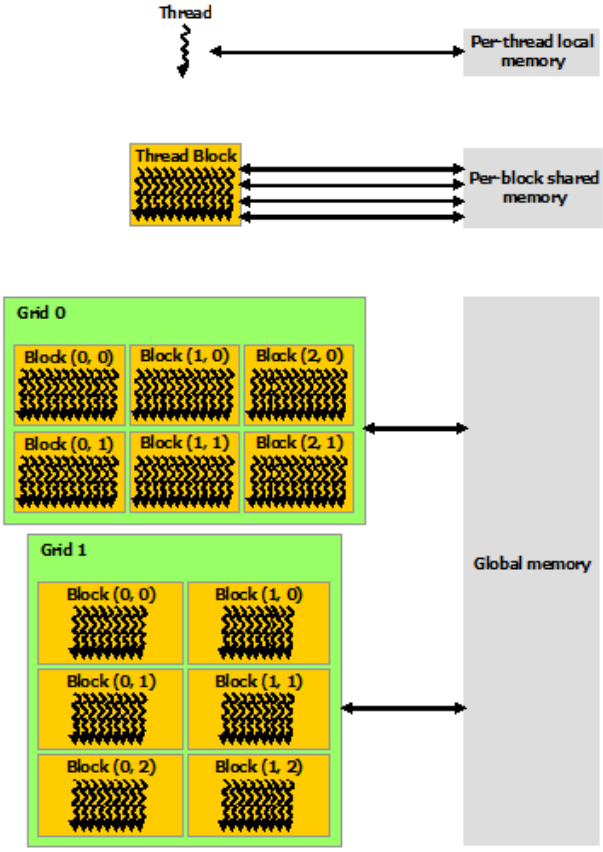

Figura 1-3. Tipos de memoria en el *device*.

1.2.4 Programación heterogénea

El modelo de programación de CUDA asume que los hilos se ejecutan en un dispositivo físicamente separado (el llamado *device*). Los kernels lanzados serán ejecutados en la GPU, mientras que el resto de código C se ejecuta en la CPU. El programa escrito es el que gestiona los accesos a los diferentes tipos de memoria, así como las transferencias de datos, mediante el llamado *CUDA Runtime* (Figura 1-4).

El programa escrito en C se ejecutará de forma secuencial. Así mismo, las órdenes correspondientes a la CPU, también lo harán de forma secuencial. Sin embargo, a la hora de ejecutar una función asociada a la GPU, como puede ser el lanzamiento de un kernel, está se ejecutará en paralelo y dará paso a la siguiente instrucción.

Existen algunas funciones que, aunque se ejecutan en el device, son asíncronas. Es responsabilidad del programador evitar que las funciones que se ejecuten asíncronamente no usen datos aun no disponibles de funciones anteriores. De lo contrario, se podrían producir problemas y la forma de actuar del programa no estaría definida.

1.2.5 Capacidad computacional

La capacidad computacional de un dispositivo, o *compute capability*, identifica a cada GPU, definiendo las características soportadas por cada una. Es usada por las aplicaciones en tiempo de ejecución para determinar que características de hardware e instrucciones están disponibles en la GPU que se trate. Se representa por un número *X.Y*, donde X refleja la arquitectura de la GPU: 7 para arquitecturas *Volta*, 6 para *Pascal*, 5 para *Maxwell*, 3 para *Kepler*, 2 para *Fermi* y 1 para *Tesla*. No debe ser confundido con la versión de CUDA.

En ocasiones, muchas actualizaciones del Toolkit introducidas por NVIDIA tienen limitaciones de hardware. Mediante la capacidad computacional, establecen umbrales en los que crean grupos de GPUs que podrán o no disfrutar de dichas actualizaciones. Un claro ejemplo es el paralelismo dinámico, disponible únicamente a partir de la capacidad computacional 3.5.

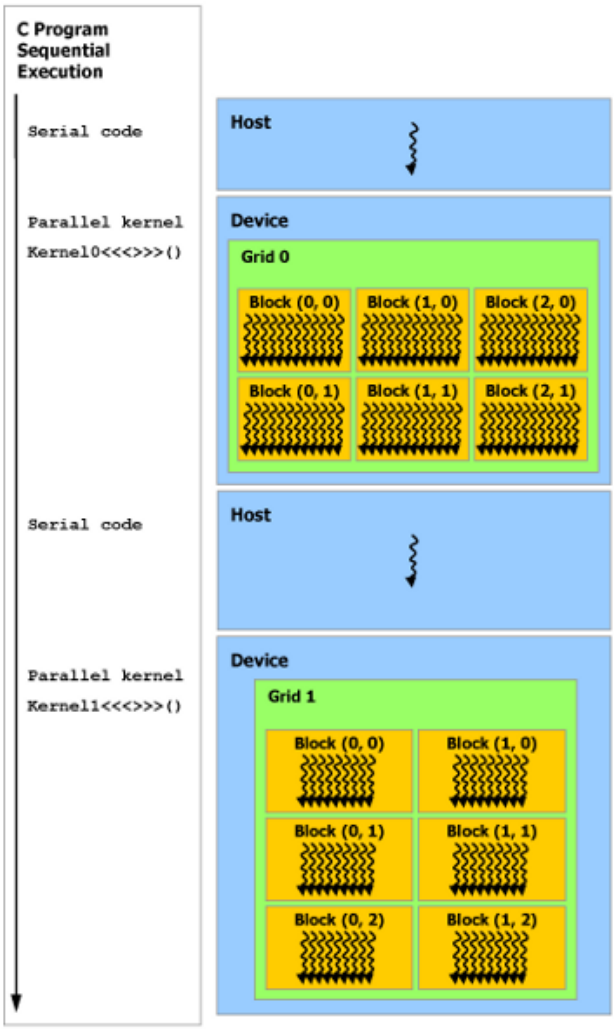

Figura 1-4. Ejecución de un programa CUDA C.

1.3 Arquitectura

Una GPU NVIDIA está construida en torno a unos procesadores multihilo, llamados *streaming multiprocessors (SMs)*. Cuando un programa en CUDA invoca un kernel, los bloques de un *grid* (red, rejilla) son enumerados y distribuidos a lo largo de los multiprocesadores con capacidad de ejecución disponible. Los hilos de un bloque de hilos se ejecutan concurrentemente en un multiprocesador, así como múltiples bloques de hilos son ejecutados en un multiprocesador. Cuando estos bloques terminan, se lanzan nuevos bloques en los multiprocesadores vacantes (Figura 1-5).

Un multiprocesador está diseñado para ejecutar cientos de hilos concurrentemente. Para gestionar tal cantidad de hilos, se emplea una arquitectura llamada *SIMT (Single-Instruction, Multiple-Thread)*. Las diferentes instrucciones son encoladas y, a diferencia de las CPUs, no se puede predecir en qué orden serán ejecutadas.

Cada multiprocesador crea, ejecuta y gestiona los hilos en grupos de 32, llamados *warps* (urdimbres). Los hilos que componen un *warp* comienzan a la vez, en la misma dirección del programa, aunque cada hilo tiene su propio registro y contador de instrucciones, por lo que son libres para ejecutarse independientemente.

Cuando un multiprocesador recibe uno o más bloques de hilos para ser ejecutados, los reparte en *warps* y hace una esquematización de la ejecución. La forma en que los bloques se reparten en *warps* es siempre la misma: cada *warp* contiene hilos consecutivos y de identificador creciente, comenzando por el hilo 0. Cada *warp* ejecuta una instrucción al mismo tiempo, por lo que la máxima eficiencia se alcanza cuando los 32 hilos del *warp* concuerdan en la orden de ejecución.

En particular, cada multiprocesador tiene un conjunto de registros de 32 bits distribuidos a lo largo de los

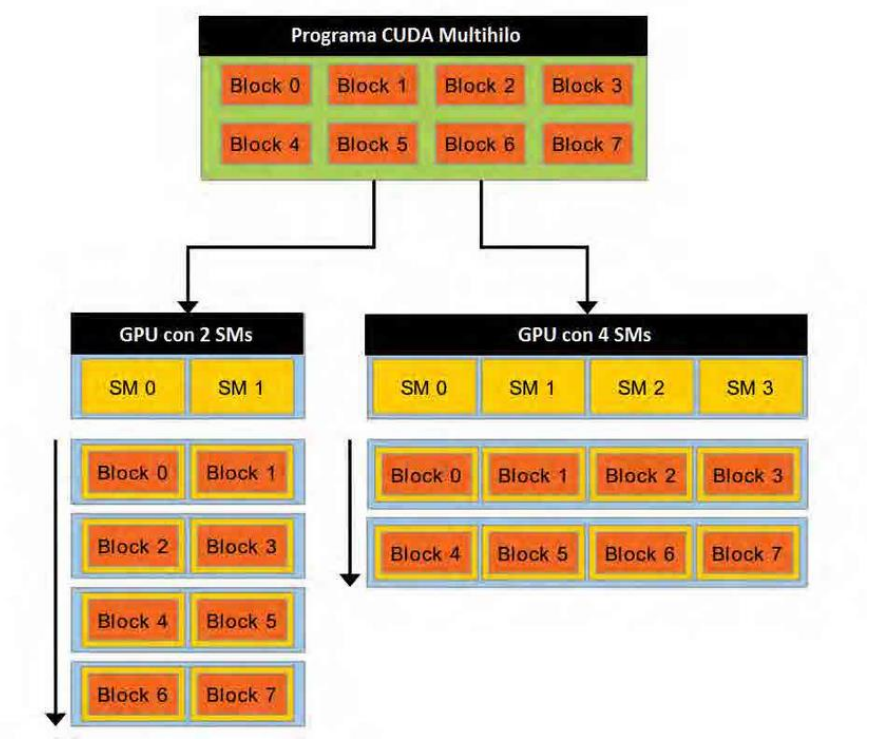

Figura 1-5. Distribución de bloques por SMs.

warps, y una memoria compartida *(shared memory)* distribuida a lo largo de los bloques. El número de bloques e hilos que pueden alojarse y ser procesados a la vez en un multiprocesador para un kernel determinado, depende de la cantidad de registros y memoria compartida usados por el kernel y de la cantidad de registros y memoria compartida disponibles en el multiprocesador. También hay un límite de *warps* y bloques de hilos por cada multiprocesador.

Todos estos límites, así como la cantidad de registros y memoria compartida, viene dado por la capacidad computacional del dispositivo. Si en un momento dado no existieran suficientes registros o memoria compartida disponible para procesar al menos un bloque, el kernel fallaría al ser lanzado.

La ocupación es una relación establecida entre los registros y memoria disponibles, y los utilizados. Teóricamente, una alta ocupación hará que los multiprocesadores estén siempre trabajando, lo que ocultará mucho mejor la latencia, y aportará resultados más eficientes. El Toolkit de CUDA nos provee de una calculadora de ocupación teórica, asi como de otras herramientas para calcular la ocupación real, que serán tratadas más adelante.

2 CONFIGURACIÓN DE CUDA EN WINDOWS

2.1 Instalación de CUDA

CUDA está disponible para distintas plataformas, aunque nos centraremos en su instalación en Windows, sistema operativo donde se ha usado para este proyecto. Para usar CUDA en Windows debemos contar con los siguientes recursos básicos:

- Una GPU que soporte CUDA. Casi cualquiera de los últimos años no debería presentar problemas de compatibilidad, más allá de las capacidades computacionales de cada una.
- Un sistema operativo Windows compatible. Esto es variable según la versión de CUDA en cuestión. La versión más reciente, CUDA 9.2, podría ejecutarse sin problemas en Windows 10, Windows 8 y Windows 7.
- Una versión de Microsoft Visual Studio compatible.
- El Toolkit de NVIDIA CUDA. Se puede descargar de forma gratuita desde la web de desarrolladores de NVIDIA, tanto el más reciente como versiones anteriores.

Una vez descargado el Toolkit, se procede a ejecutar el instalador. Este se encargará de instalar CUDA, actualizar drivers, librerías y códigos de ejemplo, entre otros. Una vez finalizado, debemos asegurarnos de que se ha instalado correctamente. Una forma de hacerlo es abrir una ventana de comandos, e introducir `nvcc -V`, comando que comprobará la versión que hemos instalado. Usaremos `nvcc` siempre que queramos usar el compilador de CUDA. En caso de obtener un error, puede deberse a una mala instalación o a una mala configuración del *path* de CUDA en las variables de entorno del sistema.

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.1\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 870M"
  CUDA Driver Version / Runtime Version          9.2 / 9.1
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 3072 MBytes (3221225472 bytes)
  ( 7) Multiprocessors, (192) CUDA Cores/MP:     1344 CUDA Cores
  GPU Max Clock rate:                            967 MHz (0.97 GHz)
  Memory Clock rate:                             2500 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 393216 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.2, CUDA Runtime Version = 9.1, NumDevs = 1
Result = PASS
Presione una tecla para continuar . . .
```

Figura 2-1. Resultados tras ejecutar *DeviceQuery*.

Además, debemos compilar y ejecutar algunos de los ejemplos preinstalados con CUDA, para verificar que todo funciona correctamente. Los ejemplos estarán, si se ha realizado una instalación por defecto, en la ubicación `C:\ProgramData\NVIDIA Corporation\CUDA Samples`. Como ejemplo, se ha compilado el proyecto *deviceQuery*, que nos proporciona información acerca de nuestra GPU, a la vez que realiza una prueba para indicarnos si nuestra tarjeta gráfica es compatible con la versión de CUDA instalada. El resultado obtenido, para una GPU GeForce GTX 870m, se puede observar en la Figura 2-1.

En caso de que nuestra GPU sea compatible con la versión instalada, y podamos compilar y ejecutar los diferentes ejemplos, podemos decir que hemos realizado una instalación correcta de CUDA en nuestro PC.

2.2 Configuración del proyecto en Visual Studio

A lo largo del desarrollo del proyecto, se han ido usando algunas librerías de NVIDIA compatibles con CUDA, como son cuBLAS o cuRAND. Aunque depende de la versión del Toolkit de CUDA instalado, por lo general vienen preinstaladas, y solo es necesario hacer alusión a ellas en las opciones del proyecto para que sean reconocidas por el compilador. Más adelante se profundizará en dichas librerías, mientras que ahora nos centraremos en la configuración genérica para cualquier librería.

En concreto, dentro de un proyecto en Visual Studio 2013 para Windows 10, habrá que dirigirse a la pestaña *PROYECTO*, y dentro de ella hacer click en *Propiedades*. En el listado de la izquierda, iremos a *Propiedades de configuración*, *Vinculador*, y *Entrada*. En dicha ventana, incluiremos en el apartado *Dependencias adicionales* cada una de las librerías que vayamos a usar (Figura 2-2).

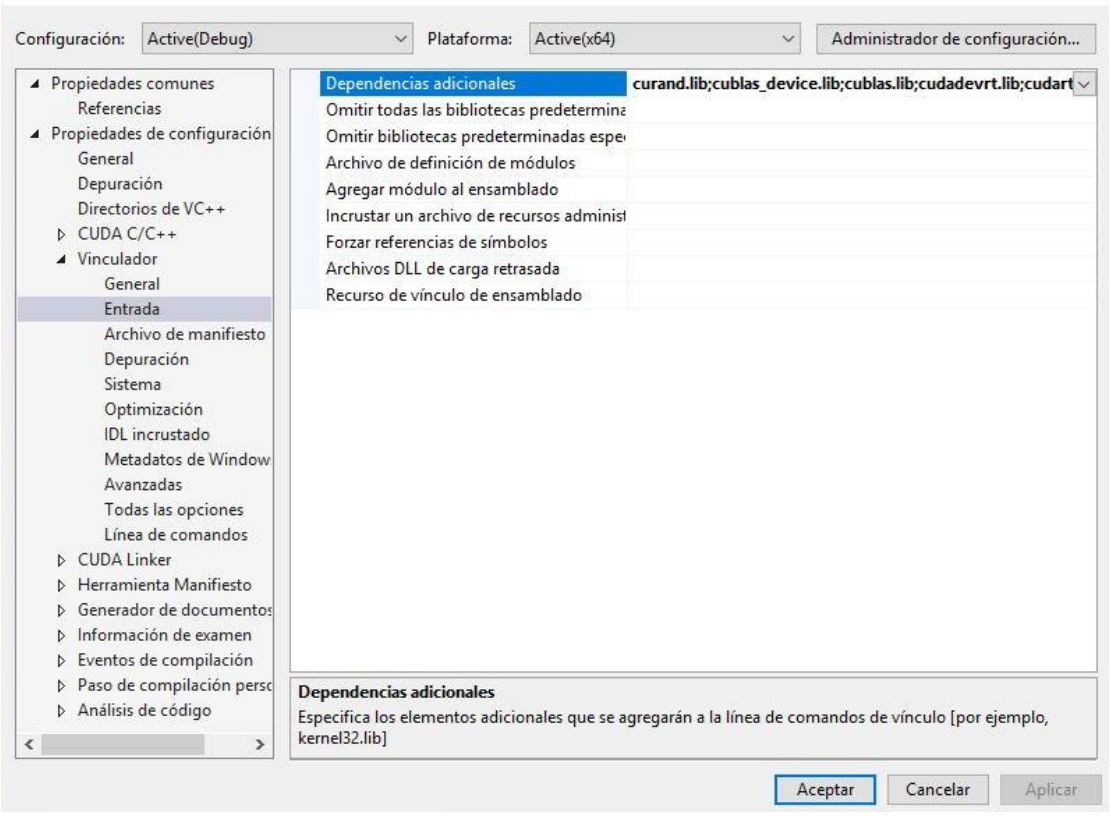

Figura 2-2. Configuración del proyecto: librerías.

En nuestro caso, hemos tenido que introducir ***curand.lib*** (asociado a la librería cuRAND), ***cublas.lib*** (asociado a cuBLAS) y ***cublas_device.lib***. Este último paso es muy importante, ya que sin dicha librería será imposible utilizar funciones de cuBLAS dentro de otros kernels, es decir, cuBLAS no funcionaría usando paralelismo dinámico.

Por otro lado, hay otros ajustes necesarios para el correcto funcionamiento del proyecto. Volvemos a *Propiedades de configuración*, *CUDA C/C++*, *Common*. Para utilizar paralelismo dinámico y algunas otras funciones, necesitaremos activar la compilación separada. Esto lo haremos marcando la opción ***Sí (-rdc=true)*** en el apartado *Generate Relocatable Device Code* (Figura 2-3). Además, el proyecto debe estar configurado en una plataforma de 64 bits. Para ello, hacemos click en el botón superior derecho *Administrador de configuración*, y elegimos ***x64*** en *Plataforma de soluciones activas*.

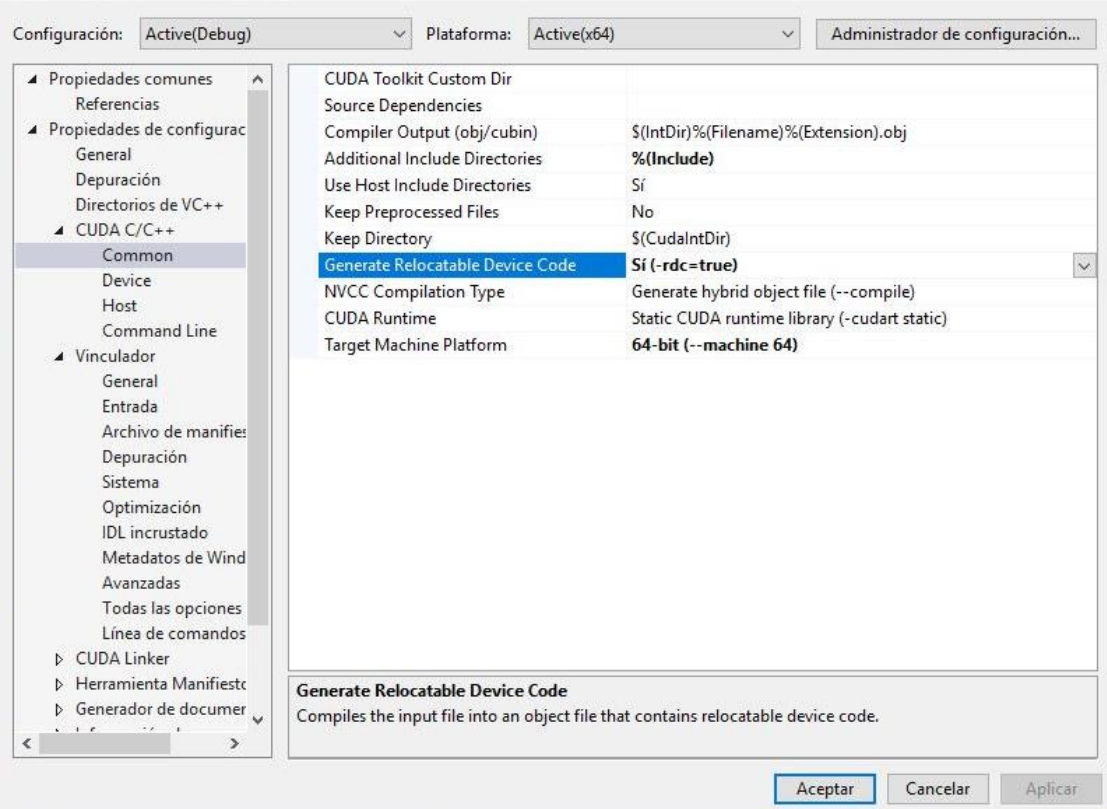

Figura 2-3. Configuración del proyecto: compilación separada.

Por último, hay que asegurarse de que se está compilando con la capacidad computacional (compute capability) adecuada a la GPU en cuestión. Para ello, en *Propiedades de configuración*, *CUDA C/C++*, *Device*, elegiremos dicha capacidad en el apartado *Code Generation*. Se definirá en parejas del tipo ***compute_XX, sm_XX;*** donde ***XX*** corresponde a una capacidad computacional ***X.X***. Es decir, si trabajamos con una GPU de capacidad computacional 3.5 (el mínimo requerido para hacer funcionar código de paralelismo dinámico), debemos asegurarnos de que en dicho apartado se incluya ***compute_35, sm_35*** (Figura 2-4).

Hay otros ajustes no obligatorios, pero que pueden ser muy útiles a la hora de depurar el proyecto y comprobar posibles errores. Entre ellos, en la misma pestaña *Device* que se trató antes, están *Generate GPU Debug Information*, equivalente al flag **-G** en la línea de comandos, que proporciona información acerca de la compilación en la GPU. También es útil marcar la opción ***Sí*** en el apartado *Verbose PTXAS Output*, equivalente a complementar la línea de comandos al compilar con la orden **–ptxas-options=-v**.

Puede haber otros parámetros que también proporcionen ayuda a la hora de detectar errores o querer obtener información de los procesos que se siguen, aunque en los párrafos anteriores ya se han mencionado los más importantes y útiles. A continuación, se hablará un poco acerca de las librerías usadas, así como de su funcionamiento e inclusión en nuestro proyecto.

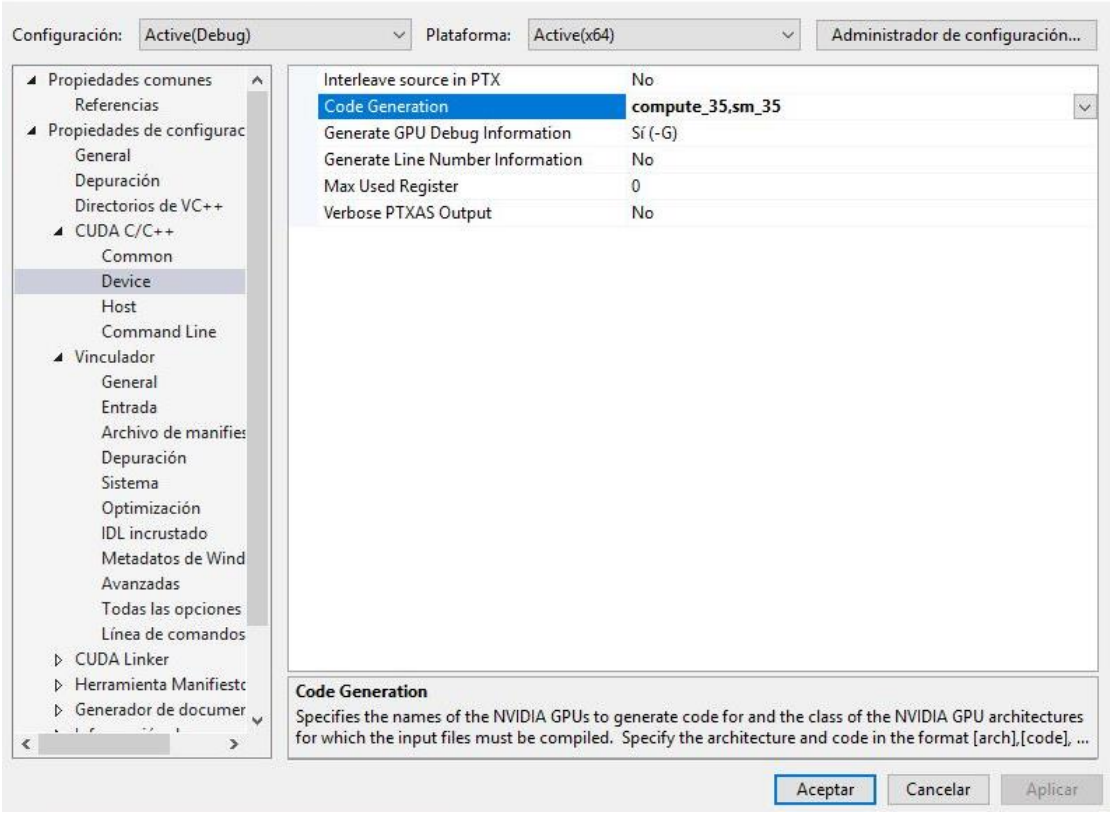

Figura 2-4. Configuración del proyecto: capacidad computacional.

2.3 Librerías usadas

2.3.1 cuRAND

La librería cuRAND nos proporciona una serie de funciones enfocadas a la generación de números pseudoaleatorios, siendo una muy buena aproximación a un conjunto aleatorio de números. En concreto, el prefijo *cu-* aquí es indicativo de su compatibilidad con CUDA.

Esta librería está formada por dos partes, una enfocada al uso en la CPU *(host API)* y otra para la GPU *(device API)*. Para la definición de ambas en nuestro proyecto, se han incluido en la cabecera con las sentencias `#include <curand.h>` e `#include <curand_kernel.h>`, respectivamente. Una vez hecho esto, podremos usar las funciones que nos proporciona cuRAND, y enfocarlas a una dirección de memoria de *host* o *device*.

Como es lógico, si trabajamos en el *host* tendrá que ser con punteros que apunten a direcciones de memoria del *host*, y viceversa para el *device*. Si por cualquier motivo esto no se cumpliera, por ejemplo, pasando como argumento un puntero de memoria *device* a una función del *host*, el funcionamiento no estaría definido y sería totalmente erróneo.

2.3.1.1 Host API

Para trabajar en la CPU, las llamadas se realizan en la CPU, y todos los números generados son guardados en la memoria del *host*. Los pasos básicos que seguir se suelen repetir al usar esta librería:

- Se crea un **generador** con `curandCreateGenerator()`, el cual puede ser de diferentes tipos.
- Se configuran las diversas opciones, como por ejemplo la semilla de generación con la función `curandSetPseudoRandomGeneratorSeed()`.
- Se aloja memoria con `malloc()` para los resultados, si no se ha hecho antes.

- Con cada llamada a la función `curandGenerate()` o similares se generarán secuencias de números. En nuestro caso, se ha usado la función `curandGenerateUniform()` para una generación con distribución aleatoria uniforme de flotantes con valores en el intervalo (0,1]. Entre llamada y llamada, podremos usar los resultados obtenidos.
- Por último, cuando se acabe la generación y no se vuelva a usar, se destruye el generador con la función `curandDestroyGenerator()`.

Es posible trabajar con diferentes generadores al mismo tiempo, ya que cada uno de ellos trabajará de forma independiente. Además, si las opciones impuestas no varían (como, por ejemplo, una determinada semilla de generación), las secuencias obtenidas tras cada ejecución serán siempre las mismas.

2.3.1.2 Device API

El uso de las funciones enfocadas al dispositivo es similar al mencionado para el *host* con algunas diferencias. En este caso, las llamadas pueden ser realizadas tanto desde el *host* como desde el *device*, pero será la GPU quien se encargue de la computación y alojamiento de los datos.

Por ello, en esta ocasión siempre tendremos que reservar memoria en la GPU en lugar de la CPU, mediante la orden `cudaMalloc()`. Hay que tener en cuenta que al lanzar un kernel que contenga una función del tipo `curandGenerate()`, se produce un retorno asíncrono, por lo que si se quisieran usar esos datos generados y la función no hubiera acabado, estaríamos ante un problema. Es por eso por lo que hay que utilizar una gestión de datos asíncrona, bien sincronizando los hilos del kernel lanzado con la llamada `cudaThreadSynchronize()` o `__syncthreads()`, o bien haciendo uso de *streams* o *events*.

2.3.2 cuBLAS

Esta librería proviene de BLAS *(Basic Linear Algebra Subprograms)*, y está adaptada al entorno de CUDA para trabajar con la GPU. Al contrario que cuRAND, cuBLAS tiene su origen en un uso del tipo *host API*, es decir, desde un inicio se diseñó para realizar llamadas desde el *host* y procesar y obtener los resultados en el *device*. Posteriormente, se diseñó una librería del tipo *device API*, que hace uso del paralelismo dinámico disponible a partir de una capacidad computacional igual o superior a 3.5.

Aunque no se ha hecho uso de ella en este proyecto, también es destacable la existencia de una librería de cuBLAS enfocada al trabajo con varias GPUs, *cuBLASXT API*. Sólo es soportada por plataformas de 64 bits, y es capaz de gestionar por sí sola la reserva de memoria en las diferentes GPUs de la forma más eficiente posible, únicamente indicando un espacio del *host* donde ya estén guardados los datos.

Algunas de las particularidades del uso de la librería cuBLAS son las siguientes:

- Todas las funciones devuelven un estado de error del tipo `cublasStatus_t`.
- Para utilizar las funciones, es necesario crear un *handle*. Es del tipo `cublasHandle_t` y es inicializado con la función `cublasCreate()`. Cuando se haya finalizado el trabajo, debemos destruirlo mediante la función `cublasDestroy()`.
- En la mayoría de los casos, como por ejemplo la multiplicación de matrices *(gemm)*, existen diversos tipos de funciones que realizan la misma operación. Para el caso de la multiplicación, la función general sería *cublas<t>gemm()*, donde *<t>* indica el tipo de función que usemos (S, D, C, Z, H). Dicho tipo hace referencia al tipo de dato tratado: S trabaja con matrices de datos *float*, D con *double*, C con datos tipo *cuComplex*…

Al igual que en BLAS, en cuBLAS existen funciones de 3 niveles. El nivel 1 abarca funciones para operaciones con escalares y vectores; el 2, para operaciones matriz-vector; y el 3, para operaciones matriz-matriz. También existen diversos tipos de datos, que serán argumentos para llamadas a funciones de cuBLAS, y que tendrán gran utilidad, como por ejemplo realizar una operación matricial con una de ellas traspuesta *(CUBLAS_OP_T)* o rellenar una parte triangular de la matriz *(CUBLAS_FILL_MODE_LOWER* para la inferior o *UPPER* para superior*)*.

Por otro lado, cuBLAS tiene una serie de curiosidades dignas de mención, como por ejemplo el tratamiento de las matrices. A diferencia de C estándar, donde las matrices se tratan en forma de filas *(row-major)*, cuBLAS

las trata por columnas *(column-major)*. Esto puede introducir algunas dificultades, sobre todo a la hora de trabajar con datos externos cuyo formato es distinto al usado por cuBLAS. Sin embargo, existen multitud de métodos incluso macros para alterar el orden de las matrices y adaptarlo a cualquiera de los dos ámbitos sin ningún problema.

3 PARALELIZACIÓN DE ALGORITMOS: FACTORIZACIÓN DE CHOLESKY

3.1 Introducción

En este apartado se expone cómo paralelizar un algoritmo determinado, a la vez que se comparará con otro no paralelizado. Se ha desarrollado un código que implementa la factorización de Cholesky en un número de matrices de tamaño determinado, tanto en la CPU como en la(s) GPU(s). En general, el código sería muy similar para hacer cualquier otra función un determinado número de veces en paralelo.

A diferencia de la programación habitual en la CPU, a la hora de hacerlo en la GPU hay diferentes parámetros variables que influyen de forma directa en los tiempos obtenidos para la factorización: número de GPUs usadas, **tamaño** de cada submatriz, **número** de submatrices, y, sobre todo, **número de bloques e hilos** del kernel.

Se han escrito dos scripts diferentes, cada uno con la finalidad de trabajar en la CPU o en la GPU. Esto es debido a que, al trabajar en varias GPUs, debemos pasar la información como estructuras para poder dividirla equitativamente en cada GPU. En la CPU esto no es necesario, y basta con tratar la información con un puntero por cada matriz.

Es necesario aclarar que es bastante complejo dar con el nivel máximo de eficiencia en cuanto a tiempos de factorización. Aquí se parte de una paralelización sencilla, en la que cada hilo se encarga de realizar la factorización de una submatriz de todo el conjunto de matrices dado. A partir de ahí, se hace un estudio para comprobar la influencia de las variables del problema en los tiempos obtenidos.

Quizá fuera posible alcanzar tiempos algo mejores si se cambiara la forma en la que actúan hilos y bloques. Por ejemplo, dividiendo cada submatriz en partes iguales, y haciendo que cada hilo procesara dicha parte en lugar de la submatriz entera. Sin embargo, esto añade una gran complejidad al problema, y en ocasiones no aportaría siquiera alguna mejora en los resultados.

A continuación, se hablará detalladamente de las funciones y utilidades usadas en ambos scripts, así como algunos de los resultados obtenidos más interesantes. Para ello, nos centraremos por un lado en la factorización realizada en la CPU, y por otro lado en la GPU, realizando posteriormente una comparativa entre ambas.

3.2 Funciones generales

Como nos hemos centrado en el proceso de factorización, se ha creado un problema a medida en el cual conocemos la solución. Para ello, hemos creado matrices originales a partir de otras ya factorizadas, haciendo así que dichas matrices sean definidas positivas, y sean factorizables. Por otro lado, se han generado una serie de términos independientes a medida, función de los coeficientes, de forma que la solución de las incógnitas de todos los sistemas sea igual a la unidad.

Algunas de estas funciones son necesarias para demostrar el funcionamiento del programa, como, por ejemplo, `crea_mat_sdp` o `crea_term_indep`. En cualquier caso, si estos datos fueran introducidos (porque se conocieran las matrices que se quieren factorizar) no habría que usar dichas funciones auxiliares.

En primer lugar, `crea_mat_sdp` es una función que crea una matriz definida positiva de tamaño `dim`. Recibe también tres punteros para almacenar tanto matrices auxiliares como la definitiva. Funciona en tres pasos: crea una matriz aleatoria con ceros en la diagonal superior; se guarda la misma matriz en otro puntero, pero traspuesta; por último, se multiplican para obtener la matriz definida positiva.

Para crear todas las matrices que necesitamos, basta con crear un bucle *for* que vaya almacenándolas en la matriz grande. Es recomendable ir cambiando la semilla de generación si queremos obtener números distintos en cada matriz. Además, hay que dejar una columna vacía por matriz para dejar espacio para los términos independientes, que serán introducidos más tarde.

Con `crea_term_indep`, generamos los términos independientes de las matrices. Como es una función auxiliar, igual que la anterior, se generan en función de los coeficientes de la matriz factorizada (por eso se hace tras la factorización, simplemente por ahorrar coste computacional).

La forma de generarlos es la siguiente: recorre los coeficientes de cada fila, y los va sumando. De esta forma, genera una serie de términos independientes tales que todas las soluciones del sistema serán 1. Esto facilita mucho la revisión y la comprobación de la factorización.

Por último, con la función `comprueba_soluciones` comprobaremos que se ha factorizado y resuelto con éxito. Esta función es únicamente de uso auxiliar, y solo funcionará con éxito si previamente se ha usado `crea_term_indep`. Si es así, revisa que todas las soluciones valgan 1, y en ese caso imprime un mensaje diciendo que se ha factorizado con éxito. En caso de haber algún número distinto de 1, sale y avisa de que ha habido fallos.

3.3 Factorización en la CPU

Como ya se mencionó antes, las matrices se almacenarán en punteros. Cada puntero hará referencia a todas las submatrices de cada tipo. Por tanto, basta con utilizar un índice para movernos dentro de cada matriz "grande" y localizar la submatriz deseada.

Para medir los tiempos de ejecución, utilizaremos distintos métodos para CPU y GPU. En este caso, haremos uso de las funciones `QueryPerformanceCounter` y `performancecounter_diff` para establecer puntos de inicio y final, y calcular el tiempo transcurrido entre ambos puntos, respectivamente.

Código. *Medición de tiempos en rutinas CPU.*

```
double performancecounter_diff(LARGE_INTEGER *a, LARGE_INTEGER *b) {

	LARGE_INTEGER freq;
	QueryPerformanceFrequency(&freq);
	return (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
}


void main(){
	QueryPerformanceCounter(&iniTime);	// primera llamada

	cholesky_cpu(h_m, h_Bfact, tam);	// función que se ejecutará

	QueryPerformanceCounter(&endTime);	// segunda llamada

	printf("Tiempo en crear %d submatrices de dimensiones %d x %d: %.16g
milisegundos\n", num, tam, tam, 1000 * performancecounter_diff(&endTime, &iniTime));
}
```

La función principal encargada de la factorización es `cholesky_cpu`. Los parámetros que recibe la función son dos matrices (`*m` y `*Bfact`) que a su vez contienen un total de *num* submatrices. En `*m` se guardan las matrices originales, mientras que en `*Bfact` se guardan las matrices tras la factorización. También recibe `dim`, que es el tamaño de cada submatriz.

Esta función se encarga de realizar la factorización cada submatriz en bucle, en el que el índice `idx_sub` se encarga de recorrer todas las submatrices, desde 0 hasta `num`. Los pasos que sigue son poner a 0 los términos por encima de la diagonal en `Bfact` y ejecutar el algoritmo de la factorización.

Por otro lado, la función `resuelve_sist_cpu` se encarga de resolver cada sistema de ecuaciones. Agrupa todas las soluciones en una matriz de dimensiones `num_submats * filas_submats`, es decir, devuelve un puntero a una matriz `sols` que contiene todas las soluciones para cada incógnita. Es una función prácticamente auxiliar, ya que sirve para comprobar las soluciones y verificar que la factorización se ha realizado con éxito.

3.4 Factorización en la GPU

Para la GPU se ha desarrollado un programa diferente al anterior, que nos da opción a usar múltiples GPUs. En cualquier caso, el programa es capaz de detectar el número de GPUs disponibles, para no ejecutarse si se le pide que lo haga usando más de las disponibles. Es necesario definir un tipo de estructura que abarque las variables con independencia para cada GPU, como la que aparece en el código a continuación.

Código. *Estructura y órdenes para ejecución en varias GPU.*

```
typedef struct{
	int tams;		//número de elementos de cada matriz
	int num_submats;	//número de submatrices en esta GPU
	int fila_submats;	//longitud de la fila en esta GPU
	int num_bloq;
	int num_hilos;
	float *sols;
	float *h_m;
	float *h_Bfact;
	float *d_m;
	float *d_Bfact;
	cudaStream_t stream;
} TGPUplan;

int detected_GPUs;
// buscamos el numero de GPUs disponibles
cudaError_t error = (cudaGetDeviceCount(&detected_GPUs));
if (error != cudaSuccess) {
	printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
	cudaGetErrorString(error));
	exit(0);}

if (NUM_GPUs > detected_GPUs){
	printf("El numero de GPUs escogido (%d) es mayor al numero de GPUs instaladas
		(%d)\nSolo se trabajara con %d GPUs\nSaliendo...\n", NUM_GPUs,
		detected_GPUs, detected_GPUs);
	exit(EXIT_FAILURE);}

TGPUplan plan[NUM_GPUs];
```

`NUM_GPUs` es un macro que es definido por el usuario, donde se indica el número de GPUs que se desea usar. Previamente a la creación de la estructura en cuestión, se verifica que el usuario no haya introducido un número mayor de GPUs al existente, mediante la función `cudaGetDeviceCount`.

3.4.1 Uso de múltiples GPUs

En algunas ocasiones, podemos encontrarnos con un PC con más de una GPU instalada. Mediante ciertas funciones de CUDA seremos capaces de exprimir esta cualidad, haciendo que ambas trabajen a la vez, con la correspondiente mejora de rendimiento que en la mayoría de los casos supone.

Para desempeñar el cálculo en varios dispositivos, es interesante recalcar el uso de la función `cudaSetDevice(int disp)`, que nos permitirá elegir la GPU donde queremos que se desarrollen las funciones que se mencionen a continuación de esta. El parámetro `disp` será un entero que representa el número de la GPU, comenzando en 0.

En general, a lo largo del programa se usará un bucle del estilo `for (i = 0; i < NUM_GPUs; i++)`, dentro del cual escribiremos el código que queramos que se desarrolle en cada GPU. La única diferencia en cada bucle es que al comienzo habrá que llamar a `cudaSetDevice(i)`, para seleccionar la GPU donde queremos ejecutar las órdenes.

El uso de varios dispositivos, aparte de darnos un extra en rendimiento, nos añadirá un tanto de complejidad.

Por ejemplo, a la hora de lanzar un kernel, tendremos que especificar al sistema en qué dispositivo queremos que se lleve a cabo la ejecución. Para ello, tendremos que hacer uso de **streams**, los cuales nos permitirán añadir un nivel más de concurrencia. Gracias a esto, asociando cada GPU a un stream diferente, haremos que cada una haga su trabajo por separado y no interfieran entre ellas.

3.4.2 Medición de tiempos

Se utilizan funciones de la librería de CUDA que tienen un funcionamiento similar al mencionado en el apartado de la CPU. Estas funciones servirán siempre que queramos medir tiempos en CUDA. Para comenzar, debemos crear dos eventos, uno para comenzar la cuenta y otra para finalizar. Esto lo haremos con la función `cudaEventCreate`.

Para definir los puntos o eventos de comienzo-final, utilizamos la función `cudaEventRecord`, que darán comienzo y final a la cuenta cuando se llamen. Se introducirá como argumento el evento que se desee, y previamente creado con `cudaEventCreate`. Estas funciones requieren de un stream, al igual que para el cálculo en múltiples GPUs. Sin embargo, en esta ocasión vale con introducir un 0 como argumento, usando así el stream por defecto.

Tras llamar dos veces a `cudaEventRecord`, es necesario sincronizar los eventos con la función `cudaEventSynchronize`. Por último, con la función `cudaEventElapsedTime` almacenamos en la variable `ms` el valor del tiempo medido en milisegundos.

3.4.3 Transferencias de memoria

Al ser el kernel ejecutado en la GPU, necesitamos transferir los datos desde el *host* al *device*, y viceversa. Para esto, aunque se suele usar la función `cudaMemcpy`, se usará la función `cudaMemcpyAsync`. La diferencia es que esta última es asíncrona respecto al *host*, y por tanto volverá antes de finalizar la copia, lo que implica un ahorro de tiempo.

Otra diferencia es que es necesario crear un stream al que asociar la función. Este debe ser creado en cada estructura, ya que habrá un stream diferente para cada llamada de `cudaMemcpyAsync` y, por tanto, para cada estructura. Es por esto por lo que cada estructura, y por tanto cada GPU, estará asociada a un stream diferente.

Código. *Funciones asociadas a la transferencia de memoria síncrona y asíncrona.*

```
error = cudaMemcpy(h_m, d_m, tam*fila * sizeof(float), cudaMemcpyDeviceToHost);
if (error != cudaSuccess) { … }

error = cudaMemcpyAsync(plan[disp].d_m, plan[disp].h_m, plan[disp].tams * sizeof(float),
       cudaMemcpyHostToDevice, plan[disp].stream);
if (error != cudaSuccess) { … }
```

Es conveniente también mencionar que, para evitar problemas tras el uso de la función `cudaMemcpyAsync`, debemos utilizar la función `cudaStreamSynchronize`. Esta función establece un punto de interrupción donde se esperará que todos los streams hayan acabado con las tareas asociadas a ellos. Gracias a esto evitaremos que se trabaje con zonas de memoria que aún no ha sido transferidas por `cudaMemcpyAsync` y se produzcan problemas indeseados y difíciles de localizar.

3.4.4 Kernels usados

3.4.4.1 Kernel de la factorización: *cholesky_gpu*

Tiene un trabajo similar a la función `cholesky_cpu`, pero con las particularidades de un kernel. La primera diferencia es que un kernel no trabaja en un bucle secuencial con un índice, sino que lo hace en paralelo en un determinado número de hilos y bloques elegidos por el usuario.

El índice en este caso es `int idx_sub = blockIdx.x * ThreadsPerBlock + threadIdx.x`, y cada valor corresponderá a una submatriz. Para no causar problemas por desborde, todo el procesamiento que sigue a

continuación se realizará dentro de una sentencia `if (idx_sub < num_submats) {…}`.

Los pasos que sigue son los mismos que en la función `cholesky_cpu`. La única diferencia es que se utilizan los valores `num_submats` y `fila_submats`, los cuales son diferentes a `num` y `fila`. Esto es debido a que en cada GPU puede haber un numero diferente de submatrices, en caso de que el total de submatrices no fuera múltiplo del número de GPUs utilizadas. Por ello, el kernel recibe como parámetros dichos valores independientes para cada estructura.

Otra particularidad del kernel es la necesidad de usar la función `__syncthreads()` , que establece un punto de interrupción hasta que lleguen todos los *warps*, para evitar problemas indeseados de sincronización entre hilos.

El lanzamiento del kernel se realiza de la siguiente forma: `cholesky_gpu <<< BlocksPerGrid, ThreadsPerBlock, 0, plan[disp].stream >>>(plan[disp].d_m, plan[disp].d_Bfact, plan[disp].num_submats, plan[disp].fila_submats , tam)`. Entre comillas se definen las características de lanzamiento: número de bloques, de hilos, y *stream* asociado al lanzamiento. El *stream* no sería necesario si se hubiera usado la función `cudaMemcpy` en vez de `cudaMemcpyAsync`. Entre paréntesis se incluyen los argumentos, como una función normal.

3.4.4.2 Kernel de la resolución de sistemas: *resuelve_sist_gpu*

Resuelve los sistemas en paralelo en la GPU, como alternativa a la CPU. El prototipo es igual, si obviamos las diferencias que existen entre una llamada a una función y el lanzamiento de un kernel: `resuelve_sist_gpu`.

El indexado es igual al kernel de la factorización *cholesky_gpu*, ya que estamos haciendo un tratamiento a cada una de las submatrices que componen el conjunto de submatrices. La única diferencia está en el cuerpo, el cual sí que es prácticamente igual al de la función *resuelve_sist_cpu*.

El kernel se lanza con la siguiente sentencia: `resuelve_sist_gpu <<< BlocksPerGrid, ThreadsPerBlock, 0, plan[disp].stream >>>(plan[disp].d_Bfact, plan[disp].d_sols, plan[disp].num_submats, plan[disp].fila_submats, tam)`. Como es lógico, para lanzar el kernel con éxito habrá que haber reservado y transferido la memoria necesaria, tal y como se explica en apartados anteriores.

3.4.5 Análisis de las variables

Como se ha mencionado en puntos anteriores, a la hora de ejecutar nuestro programa en la GPU existen determinadas variables. Es evidente que cualquier modificación de ellas hará que obtengamos tiempos de factorización diferente. Por tanto, se hará un breve análisis para intentar estudiar la influencia de cada uno de los parámetros más destacables en nuestro kernel de factorización.

Para estos análisis se ha utilizado una CPU Intel Core i7 a 2,5GHz, y diferentes dispositivos gráficos. Por un lado, se ha usado una GPU NVIDIA GeForce GTX 870M, con 3GB de VRAM, 7 SMs (multiprocesadores) y 1344 CUDA Cores. Por otro lado, se han hecho análisis con 2 dispositivos NVIDIA GeForce GTX TITAN, de 6GB de VRAM, 15 SMs y 2880 CUDA Cores cada uno. Como cada dispositivo de este tipo presenta 2 GPUs, en algunos casos se ha llegado a hacer un uso a la vez de hasta 4 GPUs.

3.4.5.1 Influencia del tamaño de cada submatriz

Partimos de un conjunto de submatrices cuadradas cuya dimensión es una de las variables de nuestro problema. Por el estilo de nuestro kernel, en el cual cada hilo hace un procesamiento de factorización individual de cada submatriz, podemos esperar que aumentar el tamaño de cada una de las submatrices implique un aumento del tiempo de factorización en una determinada proporción. El resultado de la comparación en una GTX 870M se puede apreciar en la Figura 3-1.

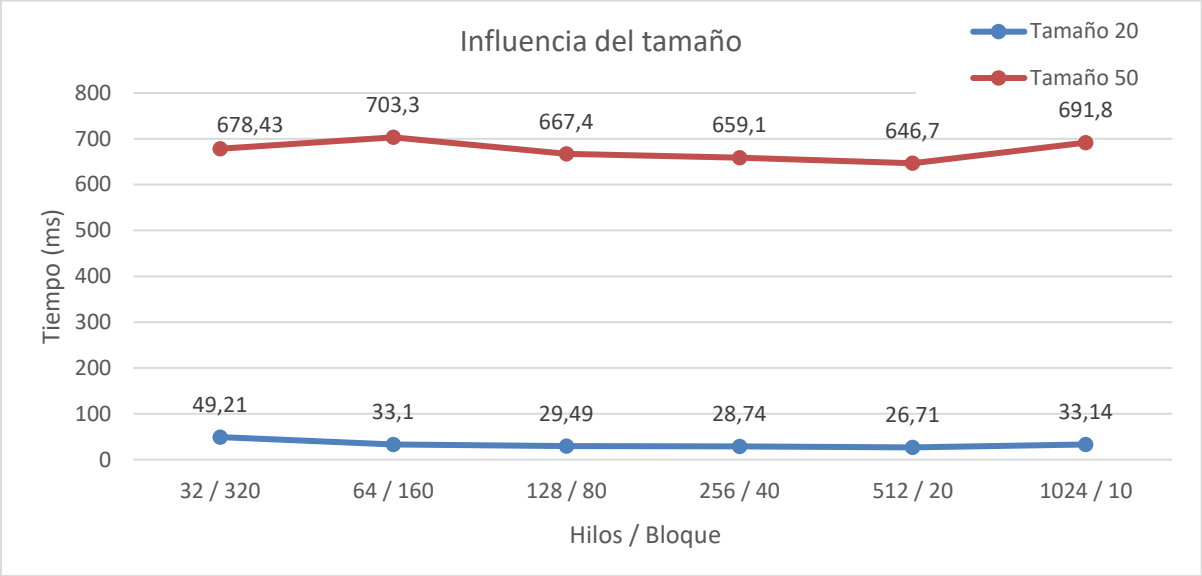

Figura 3-1. Influencia del tamaño.

Tal y como se esperaba, encontramos un comportamiento casi proporcional, en el que aumenta el tiempo de factorización considerablemente. Esto es debido a que cada uno de los hilos que procesa una submatriz tiene una capacidad de computación muy baja, si lo comparamos con una CPU. Es decir, hacer que un hilo tenga mucha carga computacional nos puede llevar a tiempos demasiado altos, por lo que lo ideal para tamaños más altos sería repartir la carga de un kernel a varios hilos.

3.4.5.2 Influencia del número de hilos de lanzamiento

Como se mencionó en el apartado de la arquitectura de CUDA, la información es procesada por los multiprocesadores en grupos de 32 hilos, llamados *warps*. Esto es algo clave si queremos observar como influye la variación del número de hilos en los tiempos obtenidos. Para comenzar, hemos obtenido una serie de tiempos en función del número de hilos, quedando de la siguiente forma (Tabla 3-1):

Tabla 3-1. Influencia del numero de hilos en la factorización.

Hilos	**Tiempo por matriz (µs)**
20	7.69
32	4.81
40	4.92
50	3.31
64	3.23
100	4.0
128	2.89
200	3.88
256	2.81
400	3.78
512	2.63
1000	3.33
1024	3.24

Lo que nos decía la teoría de la arquitectura se cumple con creces. Vemos que, si vamos alternando el número

de hilos, conseguimos tiempos mejores para aquellos números múltiplos de 32, el tamaño de un *warp*. Como ya se dijo, esto es debido a que los multiprocesadores trabajan en grupos de 32 hilos, y el hecho de hacerlo con un número de hilos que no hagan *warps* completos hará que algunos SMs se ejecuten de forma ineficiente.

Por ello, a la hora de ejecutar nuestro kernel, o alguno que tenga una estructura similar y desarrolle tareas diferentes, la elección óptima estará entre los valores marcados en rojo en la Figura 3-2. Sin embargo, esta tendencia de mejor eficiencia a mayor número de hilos no se puede generalizar. En otro problema, y con otra GPU, podemos encontrar mejores resultados con 1024 hilos.

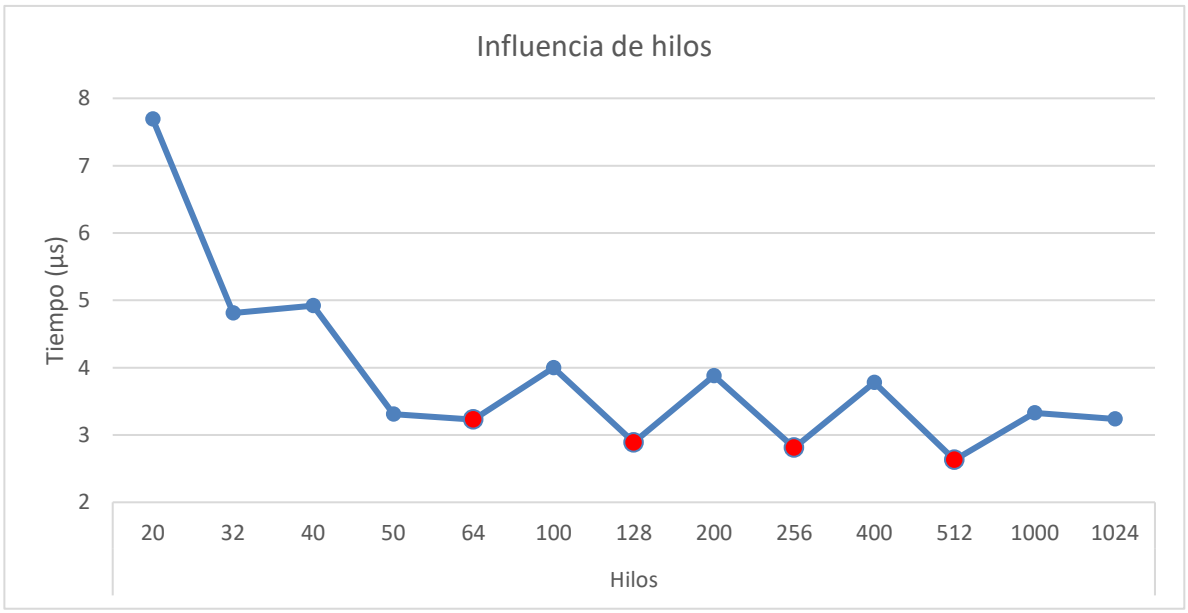

Figura 3-2. Influencia del número de hilos.

3.4.5.3 Influencia del número de bloques

Cada bloque se irá asignando a un SM, que lo procesará. Teóricamente, mientras más warps maneje cada SM, tendrá más facilidad para ocultar la latencia, y con ello conseguir tiempos mejores. Esto es debido a que, si un *warp* determinado provoca un bloqueo, el SM tiene que elegir otro. Por tanto, mientras más haya, tendrá más elecciones posibles, y con ello estará menos tiempo bloqueado.

Intentaremos hacer un análisis viendo si utilizar un número de bloques distinto de un múltiplo de 7 puede influir. Para ello, fijaremos el número de hilos utilizados en 1024, e iremos variando el número de bloques en el lanzamiento, para ver las variaciones que sufre (Figura 3-3).

Vemos que, al llegar a 7 bloques, obtenemos un tiempo mejor que si pasamos a 8. Esto es debido a que cuando elegimos 7, estamos ocupando todos los SMs a la vez. Sin embargo, al pasar a 8, habrá un instante en el que solo un SM esté trabajando, y el kernel no finaliza hasta que este acabe. Por ello encontramos ahí un incremento de tiempo.

Esta teoría no se mantiene si nos vamos a los siguientes múltiplos de 7. Aún así, puede tener una explicación bastante sencilla, y es el orden en que se van asginando los bloques a los SMs. Que se asignen 14 bloques, no implica que esto vaya a ser en dos "tandas" de 7. Cuando observamos que en 15 se mantienen los tiempos obtenidos en 14, corroboramos esto. Y es que, si ha quedado algún SM vacante, se le ha asignado el bloque sobrante. Es decir, se han conseguido procesar los 15 bloques en sólo 2 "tandas".

Esto toma más sentido si vemos la tendencia de la curva obtenida hacia un número infinito de bloques: a medida que hay más bloques, hay más posibilidades de ocupar SMs vacantes, y con ello mejorar los tiempos. Esto conlleva una reducción de la latencia, que se mencionaba en el apartado de la arquitectura. Esta curva, obtenida de la misma forma, pero en un alto número de bloques, debe ser prácticamente horizontal.

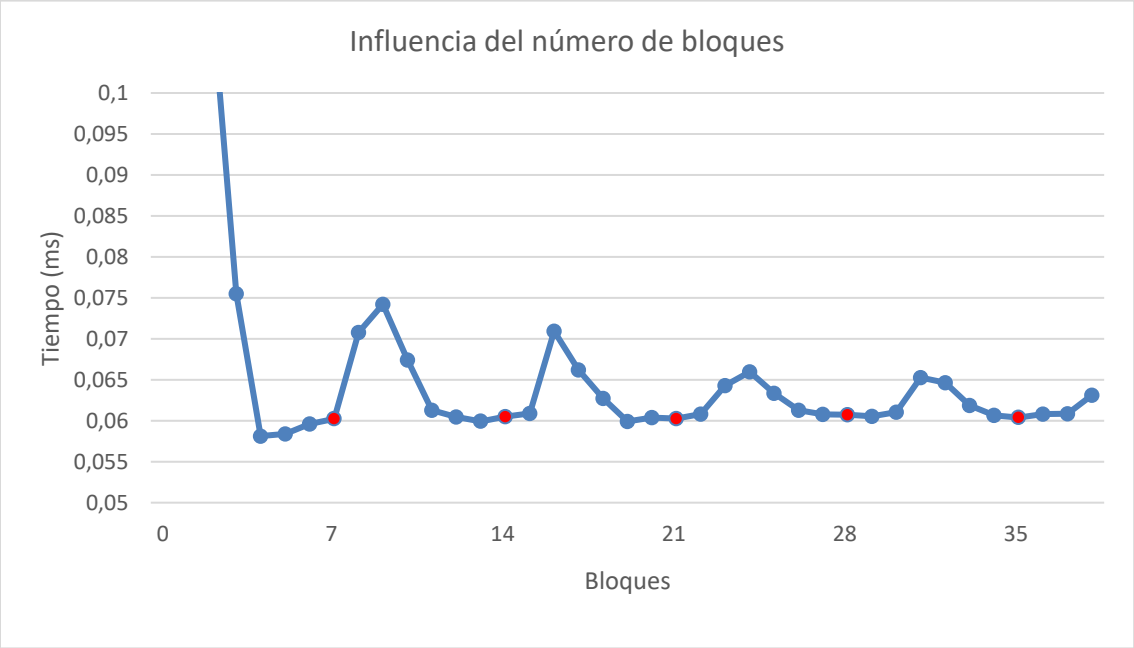

Figura 3-3. Influencia del número de bloques.

3.4.5.4 Ocupación

Como ya sabemos, cada dispositivo tiene una serie de límites físicos a la hora de almacenar registros y memoria en cada SM. A su vez, según los parámetros que establezcamos nosotros, entre ellos bloques e hilos, conseguiremos que cada SM almacene un determinado número de registros y memoria compartida. La relación porcentual entre los registros y memoria usadas en cada momento, respecto a los límites establecidos por el dispositivo, representa la ocupación.

Teóricamente, una ocupación alta es algo positivo, ya que ocultamos bien la latencia y mejoran los tiempos. Por el contrario, una baja nos debe dar peores resultados. Sin embargo, en la práctica esto no siempre es así, y muchas veces los tiempos óptimos se consiguen con una ocupación media-baja.

Para el estudio de la ocupación en nuestro kernel hemos usado dos herramientas. Estas nos permiten calcular la ocupación teórica y la ocupación real según el número de hilos usados en nuestro kernel. La teórica se calcula mediante una hoja Excel proporcionada por NVIDIA, descargable desde la web oficial, llamada *CUDA_Ocuppancy_calculator.xls*.

Por otro lado, para la real, utilizamos la herramienta *nvprof* en la línea de comandos del símbolo del sistema de Windows. Esta herramienta es un *profiler* (perfilador), que nos proporciona información sobre nuestro archivo, como por ejemplo sobre la ejecución del kernel, transferencias de memoria o configuración de la memoria. En concreto, podemos elegir que comprobar añadiendo el flag `--metrics`, y los parámetros

```
PS E:\OneDrive - UNIVERSIDAD DE SEVILLA\4 GITI\TFG\GPU.30.03.18\GPU.30.03.18> nvprof --metrics achieved_occupancy,executed_ipc GPU3
==3008== NVPROF is profiling process 3008, command: GPU3

Generando 8192 matrices de dimensiones 50 x 50...

Factorizando 8192 matrices de dimensiones 50 x 50...
Tiempo en factorizar 8192 submatrices de dimensiones 50 x 50: 476.10455 milisegundos

Exito al resolver el sistema!

==3008== Profiling application: GPU3
==3008== Profiling result:
==3008== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "GeForce GTX 870M (0)"
    Kernel: cholesky_gpu(float*, float*, int)
          1                        achieved_occupancy                        Achieved Occupancy    0.571820    0.571820    0.571820
          1                                   ipc                              Executed IPC    0.021333    0.021333    0.021333
```

Figura 3-4. Resultado tras aplicar *nvprof* al kernel de factorización.

deseados. El resultado de aplicar esta herramienta al kernel se puede consultar en la Figura 3-4.

Reuniendo la información necesaria de ambas ocupaciones para la ejecución del programa con diferentes números de hilos, se obtiene la gráfica que podemos ver en la siguiente figura (Figura 3-5):

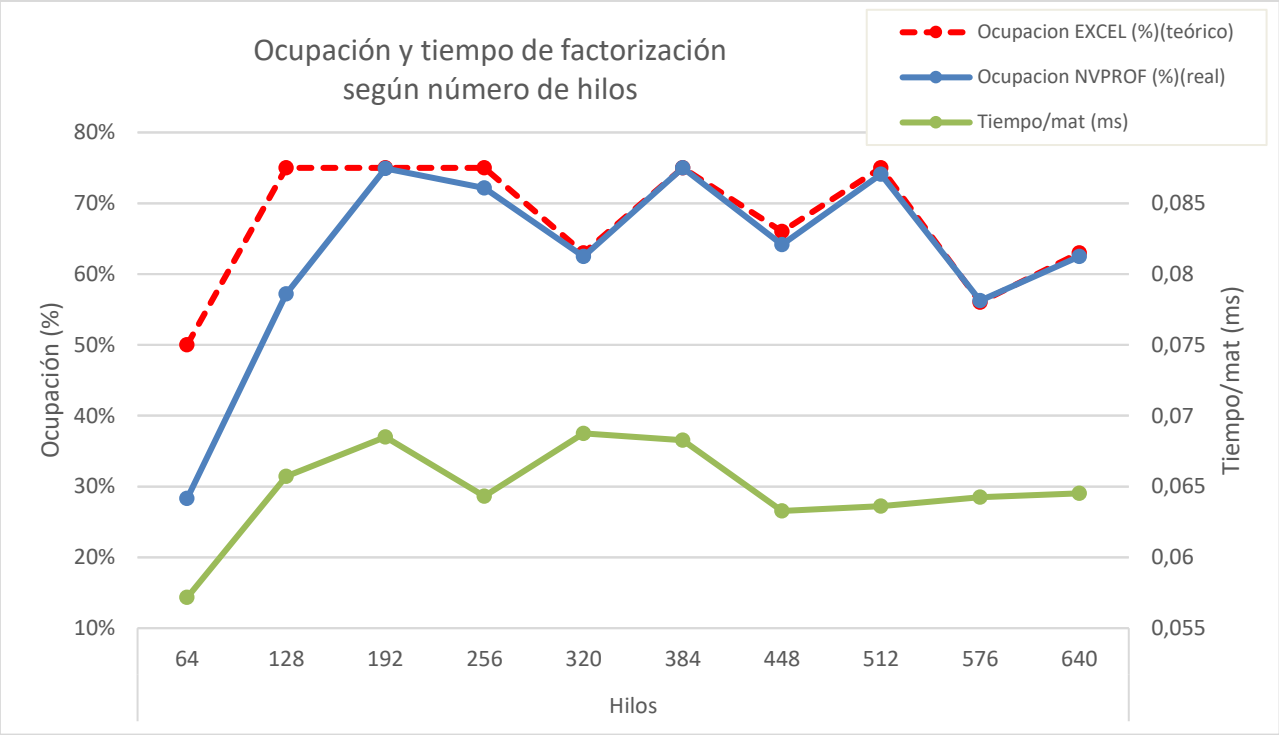

Figura 3-5. Ocupación.

Como podemos ver, no siempre una alta ocupación implica mejores tiempos, ni una baja ocupación malos. Es algo bastante complejo como generalizarlo con un simple caso, ya que es algo muy variable según el tipo de problema y dispositivos que estemos usando.

3.5 Comparativa entre CPU y GPU

Vistas las características del algoritmo tanto en la CPU como en la GPU, y tras haber profundizado un poco en el funcionamiento del kernel, procedamos a la comparación de ambos métodos. En la mayoría de los casos, elaborar un algoritmo en código C estándar es mucho más sencillo que en CUDA. Muchas menos líneas de código, menos complejidad y más soporte hacen que la programación en la CPU sea nuestra primera opción.

Cuando nos decantamos por el uso de CUDA y la programación en GPU, es porque esto nos va a dar una optimización y rendimiento mayor que la CPU. Sin embargo, pueden existir casos en los que, tras horas y horas de programación en CUDA, obtengamos resultados incluso peores que con un simple programa en C. En este punto se va a hacer una comparativa entre ambos métodos, y se intentará explicar cuando es conveniente usar cada uno, así como los beneficios y contras que nos aportan.

Cuando hacemos una ejecución de ambas formas, intentando hacer que los códigos y parámetros de ejecución sean lo más parecidos posibles, obtenemos los resultados esperados: la factorización en la GPU da resultados mucho mejores a partir de un determinado umbral. En este caso, para submatrices de tamaño 20, a partir de 1.000 submatrices comienza a compensar el uso de la GPU frente a la CPU (Figura 3-6). Este umbral será variable en función del tipo de kernel que tratemos. Es posible incluso encontrar funciones que sean totalmente inviables en CPU o GPU, ya que del modo contrario siempre da mejor resultado.

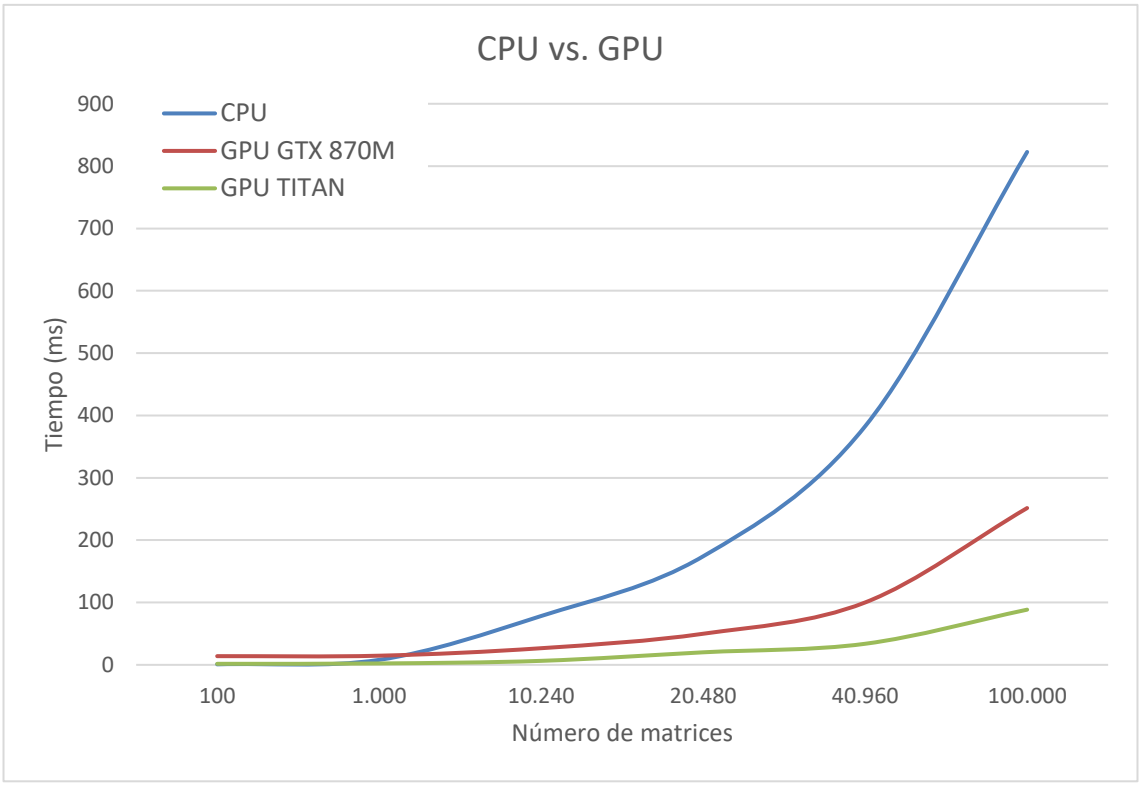

Figura 3-6. Comparativa CPU vs GPU.

En nuestro caso, este límite será distinto según el número o tamaño de las matrices, como es lógico, y será la aplicación la que marque qué es mejor usar, si CPU o GPU. Por ejemplo, para factorizar 50 matrices, compensaría usar CPU, ya que los costes de *overhead* en la GPU superan a los costes de cálculo de la CPU. Estos costes de *overhead* abarcan el lanzamiento del kernel, transferencias de memoria, y creación y destrucción de ciertos objetos que no serían necesarios en una CPU, entre otras cosas.

Esto se ve de forma clara si hacemos zoom en la zona baja de la gráfica anterior (Figura 3-7). Podremos observar como es el comportamiento de la factorización para un número de matrices bajo, y como en este caso sí que no compensaría desarrollar y usar el kernel, ya que hay un nivel bajo de paralelización, y un código secuencial en CPU sería incluso más efectivo.

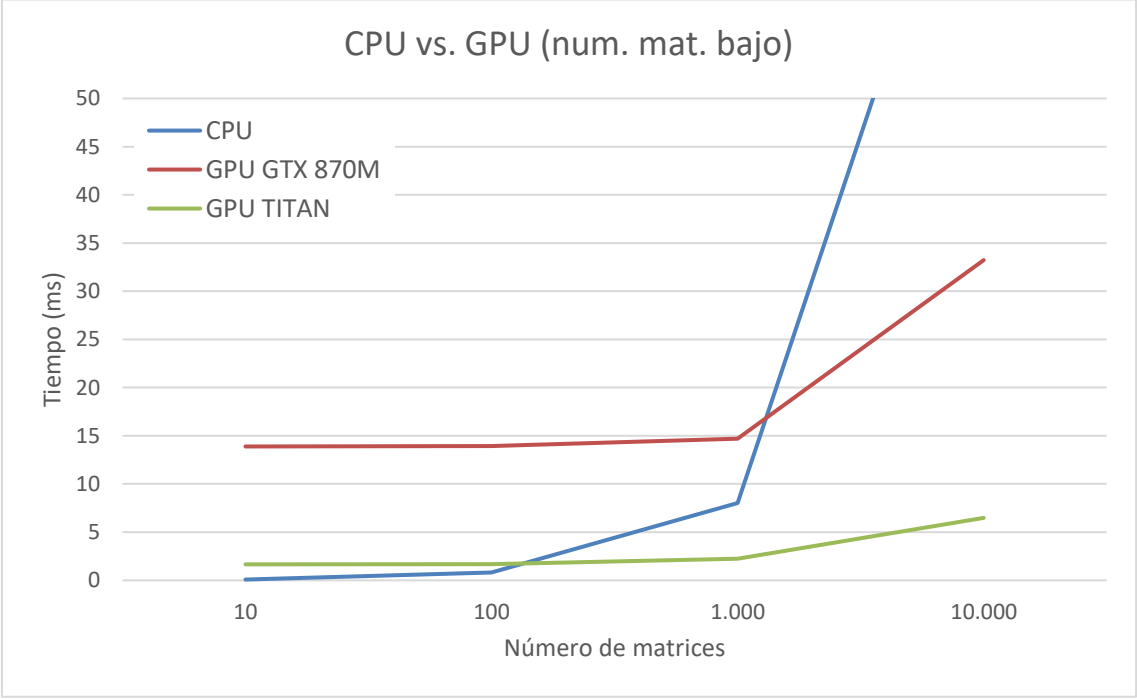

Figura 3-7. Comparativa CPU vs GPU para número de matrices bajo.

En esta última figura, vemos que cuando trabajamos con pocas matrices tenemos varios escenarios diferentes. Para un número muy bajo (<100), la opción de la CPU es la que obtiene los mejores tiempos. Para tratar menos de 1000 matrices aproximadamente, y suponiendo que nuestra GPU fuera la GTX 870M, la opción de la CPU seguiría siendo más viable.

Sin embargo, para un número mayor a partir de ahí, la opción de la GPU es la que predomina, sea de gama media o alta. La CPU toma tiempos del orden de 20 veces más, por lo que sería difícil encontrar algún caso en este contexto en el que mereciese la pena usarla.

4 BAGGING EN AJUSTE POR MÍNIMOS CUADRADOS

4.1 Introducción

A continuación, se explica de forma breve un ejemplo de paralelización de algoritmos en CUDA, el *bagging* para el ajuste por mínimos cuadrados. Esto consistirá en dividir el total de datos de entrenamiento en varios conjuntos, que serán procesados de forma individual y posteriormente promediados, en lugar de procesarse como un solo conjunto. Gracias a esto se reducirá considerablemente la varianza en la estimación de mínimos cuadrados.

Los conjuntos extraídos tienen la particularidad de que pueden contener datos repetidos, es decir, no solo estamos repartiendo el total de datos en conjuntos, si no creando conjuntos formados por valores aleatorios del total de conjuntos.

El ejemplo que se propone está enfocado a la regresión lineal, por lo que se trabajará con el par de coordenadas x e y de un conjunto de puntos. El objetivo es encontrar los parámetros α y β que satisfagan una recta de mínimos cuadrados de la forma

$$y = \alpha + \beta x$$

Para ello, se procederá a buscar el vector θ a partir de la ecuación siguiente, el estimador de mínimos cuadrados,

$$\theta = \begin{bmatrix}\alpha \\ \beta\end{bmatrix} = [M^T(N)M(N)]^{-1} M^T(N)Y(N)$$

donde la matriz M contiene los diversos puntos x y el vector Y está formado por las coordenadas y. Así, tendremos los parámetros de la pendiente y el término independiente para calcular la recta objetivo en un conjunto de N puntos. La forma de M e Y para nuestro problema, al ser lineal, es la siguiente:

$$M = \begin{bmatrix}1 & x(n) \\ 1 & x(n+1) \\ \vdots & \vdots \\ 1 & x(N-1) \\ 1 & x(N)\end{bmatrix}, \qquad Y = \begin{bmatrix}y(n) \\ y(n+1) \\ \vdots \\ y(N-1) \\ y(N)\end{bmatrix}$$

Como se ha comentado antes, se procesarán los conjuntos de forma paralela e individualmente, por lo que habrá que calcular dichos parámetros para cada uno de los conjuntos que se formen. Aquí entra en juego la paralelización, ya que cada una de las operaciones necesarias (multiplicación, inversión, trasposición) se harán a la vez para todos los subconjuntos mediante un kernel de operación.

4.2 Paralelismo dinámico

Para resolver cada una de las operaciones que componen el estimador de mínimos cuadrados, y así hallar los parámetros de nuestra recta, hemos utilizado cuBLAS, librería ya mencionada anteriormente. La particularidad que surge es que cuBLAS debe ser ejecutado dentro de otro kernel, ya que debemos hacer tantas ejecuciones de cuBLAS como conjuntos estemos procesando en paralelo.

Actuando de esta forma, estaremos intentando ejecutar un kernel dentro de otro, ya que las funciones de la librería cuBLAS son kernels. Esto solo es posible utilizando el **paralelismo dinámico** de CUDA (Figura 4-1). Esta tecnología sólo está presente en GPUs cuya capacidad computacional es igual o superior a 3.5.

Si resulta tedioso buscar la máxima eficiencia a un programa desarrollado en CUDA, el uso del paralelismo agrava aún más esta dificultad. Al usarlo, estamos haciendo sobre la GPU lo mismo que esta hace sobre la

CPU en condiciones normales. Esto implica que también estamos añadiendo costes por *overhead* a los kernels que usen paralelismo dinámico, y haciendo más complejo el control de los tiempos.

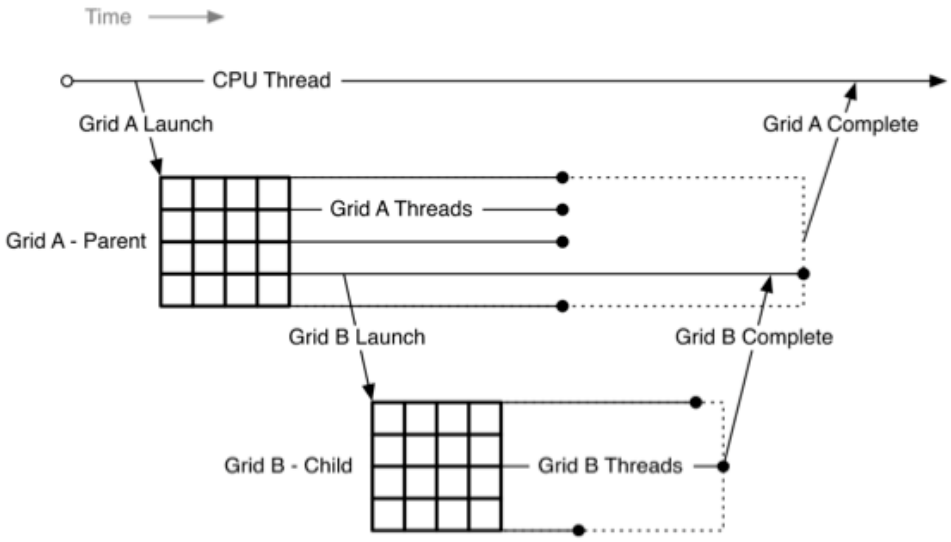

Figura 4-1. Paralelismo dinámico en CUDA.

Es por esto por lo que el uso de paralelismo dinámico tampoco implica siempre una mayor eficiencia. Se puede ver como una analogía a lo hablado en la comparación entre CPU y GPU. Habrá casos en los que el nivel de paralelismo sea alto, y nos ofrezca grandes resultados, mientras que en otros pueda incluso duplicarnos un mal resultado. A continuación, se expone un ejemplo en el que se realiza una multiplicación de matrices con cuBLAS dentro de un kernel distinto:

Ejemplo. *Uso del paralelismo dinámico en multiplicación de matrices.*

```
__global__ void mult_kernel(int nvect, float *Ag, int Afilas, int Acols, float *Bg, int
Bcols, float *Cg){
              /* Ag, Bg y Cg son matrices 'grandes' que contienen una matriz A, B y C por
cada conjunto creado. Por eso hay que asignar un conjunto a cada hilo. */
              int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
              float *A, *B, *C;
              int tam_A = Afilas*Acols;
              int tam_B = Acols*Bcols;
              int tam_C = Afilas*Bcols;
              if (idx < nvect){
                     /* asignacion de conjunto corresp. a cada hilo */
                     A = Ag + idx*tam_A;
                     B = Bg + idx*tam_B;
                     C = Cg + idx*tam_C;

                     const float alpha = 1.0f;
                     const float beta = 0.0f;
                     cublasHandle_t hdl;
                     cublasStatus_t status = cublasCreate_v2(&hdl);

                     status = cublasSgemm_v2(hdl, CUBLAS_OP_N, CUBLAS_OP_N, Bcols, Afilas,
Acols, &alpha, B, Bcols, A, Acols, &beta, C, Bcols);
                     /* operación de multiplicación de matrices en CUBLAS */
                     __syncthreads();
                     if (status != 0) printf("mult_kernel idx %d, status: %d\n", idx,
status);
                     cublasDestroy_v2(hdl); }
```

4.3 Estructura del proyecto

4.3.1 Cabeceras

En el proyecto se incluyen diferentes librerías, desde algunas básicas en C como `<stdio.h>` a `<cuda_runtime.h>` o `<curand.h>`, pasando por las creadas personalmente, como `"subconjuntos.cuh"`.

También se definen, bien en forma de macro o bien en forma de variable global, el numero de hilos por bloque de cada kernel, que se utilizará a la hora de lanzar los diferentes kernels.

Por último, se definen macros para la detección y depuración de errores, tanto para CPU como para GPU, así como para las diferentes librerías. Por ejemplo, para cualquier función de CUDA se puede usar la siguiente macro de depuración:

Ejemplo. *Macro usada para depurar errores en CUDA.*

```
#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { \
printf("Error at %s:%d\n",__FILE__,__LINE__);\
return EXIT_FAILURE;}} while(0)
```

4.3.2 Archivo mincuad.cu

Es el archivo .cu principal, donde está el `main`. Aparte, se definen los cuatro kernel de operación necesarios para el estimador de mínimos cuadrados, así como dos funciones auxiliares para lanzar de forma ordenada el resto de kernels y funciones.

4.3.2.1 Funciones auxiliares

La primera función es `int crea_subconj_y_mat(int d, int m, int num_sub, int tam_sub, int datomax, float *d_M, float *d_Y)`. Hace uso de la librería subconjuntos.cuh y su correspondiente archivo subconjuntos.cu, los cuales han sido creados personalmente desde cero, e incluye las funciones necesarias para crear los subconjuntos requeridos a partir del conjunto total de puntos. Dentro de dicha función, se hacen llamadas a otras funciones y kernels. Las más destacables son:

- `int gen_datos_x(int d, float *d_dato)`. Genera `d` datos aleatorios en el puntero `*d_dato` alojado en la memoria de la GPU, que serán las coordenadas *x* del conjunto de puntos. Hace uso de la librería cuRAND. Para las coordenadas *x*, se ha hecho uso del *Host API*, es decir, se han usado funciones que son llamadas desde la CPU y generan datos en la GPU. Los datos generados por la librería usada serán flotantes cuyo valor estará entre 0 y 1, por lo que deberán ser escalados con otra función.
- `__global__ void escala_datos(int dato_max, int ndatos, float *d_dato)`. Kernel que se encarga de escalar los datos aleatorios a números en el conjunto [0, `ndatos`], y asigna valores a estos datos entre [0, `dato_max`], como se menciona en la función anterior.
- `__global__ void gen_datos_y(int d, float *x, float *y, curandState *state)`. Este kernel usa una función para generar las coordenadas *y* a partir de las *x*. En concreto, se ha usado la función $y = x$ para generar estos datos, resultando muy sencilla la comprobación de los cálculos realizados a la hora de evaluar los resultados de la recta de mínimos cuadrados. También se introduce una componente de ruido, que para los casos evaluados ha sido de un máximo del 20%.
- `__global__ void crea_subconj(int num_vect, int ndatos, int tam_subconj, int num_subconj, int *prueba_indice, float *prueba_dato, float *d_dato_x, float *d_dato_y, curandState *state)`. Es uno de los kernels con más carga computacional. Se usa la *Device API* de la librería cuRAND, ya que es llamada desde la GPU. Aquí se realiza la selección de datos que compondrán cada uno de los conjuntos. Todos los subconjuntos se guardarán en `*prueba_dato`, aunque también se plantea la opción de guardar los índices de cada dato en `*prueba_indice` para trabajar con sus direcciones de memoria y evitar reservas de memoria redundantes. Sin embargo, es algo más complicado e innecesario en nuestro caso.
- `__global__ void crea_mat(int num_vect, int num_subconj, int tam_subconj, float *prueba_dato, float *M, float *Y)`. Este kernel crea las matrices necesarias para el estimador de mínimos cuadrados, *M* e *Y*.

La otra función es `float* mat_op(int nvect, int num_subconj, int tam_subconj, float *d_M, float *d_Y)`. Esta función contiene y gestiona todos los kernels de operación para el estimador de mínimos cuadrados, y devuelve el conjunto de vectores θ que contienen los parámetros de las rectas. Hay que tener en cuenta que cuBLAS debe recibir los punteros de matrices individuales, no tal y como están guardadas en M e Y (todas juntas). Para ello se intentará que, en cada kernel, cada hilo procese un vector distinto. Los kernels que contiene esta función son los siguientes:

- `__global__ void inv_kernel(int nvect, float *a_ig, float *c_og, int n)`. Kernel para realizar la inversa de una matriz de dimensión `n`. Recibe un puntero a la matriz a invertir, así como otro donde alojará la matriz inversa. Como es lógico, obtendremos un mayor rendimiento para mayores valores de `n`.
- `__global__ void mult_kernel(int nvect, float *Ag, int Afilas, int Acols, float *Bg, int Bcols, float *Cg)`. Este kernel se encarga de multiplicar dos matrices. Es importante destacar que cuBLAS recibe las matrices en formato *column-major*, al contrario que C estándar, cuyo tratamiento es *row-major*. No hay que preocuparse por hacer un cambio ya que basta con trabajar con las traspuestas para cambiar el formato en cuestión.
- Por otro lado, el estimador necesita otras operaciones de multiplicación en las cuales algunas de las matrices que participan son traspuestas. La función de cuBLAS que se encarga de la multiplicación (*cublasSgemm*) tiene unos *flags* en los argumentos dos y tres, de la forma CUBLAS_OP_'X'. Dicha 'X' puede tomar, entre otros valores, 'N' para no alterar su contenido, o 'T' para considerarla traspuesta. Además, activar el flag CUBLAS_OP_T implica que habría que modificar los argumentos restantes de la función para ajustar las dimensiones. Debido a esto, se han hecho otros dos kernel: `__global__ void mult_trans_kernel(int nvect, float *Mg, int Mfilas, int Mcols, float *Cg)` y `__global__ void mult_trans_kernel_2(int nvect, float *Ag, int Afilas, int Acols, float *Bg, int Bcols, float *Cg)`. El propósito de esto es tener a mano las tres posibles operaciones entre dos matrices *A* y *B*: AB, A^tB y AB^t.

4.3.2.2 Main

La función principal comienza con la declaración de variables básicas para el problema: número de datos generados, número de conjuntos, valor máximo de cada dato, etc. Todos estos parámetros serán recogidos por las funciones auxiliares para desarrollar los conjuntos y operaciones. También se realiza una reserva de memoria para los punteros de datos de las matrices, así como la creación de eventos de medida de tiempo del tipo `cudaEvent_t`.

Para la medida de los tiempos, se ha ejecutado la función que realiza todas las operaciones dentro de un bucle *for*, un número de veces determinado. Después, se ha obtenido un tiempo promedio de la ejecución de las operaciones. Este tiempo también incluye la transferencia de memoria del vector de parámetros desde la GPU a la CPU.

4.3.3 Archivos subconjuntos.cu/cuh

Como ya se ha mencionado antes, en este archivo separado se encuentran las funciones necesarias para elaborar un determinado número de conjuntos de tamaño concreto a partir de un conjunto de datos, así como también para crear dicho conjunto de datos de forma aleatoria si se desea.

El objetivo de crear este archivo es, aparte del orden a la hora de trabajar, tener separadas las funciones relativas a los subconjuntos con las de la estimación por mínimos cuadrados. De esta forma, se puede usar este fichero en otro proyecto que requiera de una agrupación en conjuntos similar pese a que el objetivo de uso sea distinto.

Al igual que en el lenguaje C estándar, basta con incluir el archivo en el proyecto en cuestión, e incluirlo en la cabecera del fichero principal con `#include "subconjuntos.cuh"`. Por otro lado, en el archivo .cuh se utiliza la sentencia `#ifndef` para evitar errores de definición reiterada de la librería:

Ejemplo. *Inclusión de librerías propias con #ifndef*

```
#ifndef SUBCONJUNTOS_CUH
#define SUBCONJUNTOS_CUH

int gen_datos_x(int d, float *d_dato);
	…
__global__ void crea_mat(int num_vect, int num_subconj, int tam_subconj, float
	*prueba_dato, float *M, float *Y);


#endif
```

5 CONCLUSIONES Y LÍNEAS DE CONTINUACIÓN

5.1 Conclusiones

Gracias a CUDA, hoy en día tenemos las herramientas necesarias para llevar a cabo la paralelización del algoritmo que queramos. Es evidente que el uso de estas va a ser mayor con el paso de los años, sobre todo en ramas de conocimientos aún por descubrir. La capacidad de hacer una computación en paralelo supone algo nuevo con respecto a lo que conocíamos hasta hace unos años.

Sin embargo, por sencilla que sea la cuestión, siempre estará presente la duda de cuál será la forma más eficiente de lograr nuestros objetivos. Esto es algo que, a nivel básico, se ha intentado describir a lo largo del presente trabajo. Y es la propia aplicación que tengamos entre manos, la que definirá si es conveniente usar una CPU, una GPU, o ambas a la vez.

Aquí se ha trabajado con una determinada función, que bien puede ser secuencial o incluso única, a la que se le da cierto nivel de paralelismo, como puede ser la factorización de Cholesky para la resolución de sistemas de ecuaciones. A partir de ahí, se exponen diferentes casos y situaciones en los que descubrimos que todos los métodos usados son útiles. Lo único necesario es saber en qué momento y de qué forma usarlos.

Para la factorización de Cholesky hemos encontrado que la CPU nos puede ayudar para factorizar un número bajo de matrices. Si quisiéramos hacerlo a miles de ellas, tendríamos un grave problema en cuanto a tiempos de ejecución. Esto se soluciona con la programación en paralelo mediante CUDA, llevando a cabo una operación para miles en el mismo tiempo que una CPU convencional se toma en hacerlo en apenas 100.

Por otro lado, también hemos descubierto la gran influencia que tiene conocer la arquitectura interior de CUDA. Desarrollar un programa en paralelo requiere de habilidad y conocimiento para distribuir los recursos, y esto resulta imposible si no conocemos como se comportan estos. Además, hemos verificado como las suposiciones teóricas en cuanto a la estructura de dichos recursos son ciertas para la aplicación que le hemos dado.

Hemos complementado todo este estudio con el máximo uso posible de diferentes alternativas, como pueden ser las librerías de NVIDIA (cuBLAS, cuRAND), a la vez que también hay partes donde se desarrollan estas funciones con un código propio.

Lo mismo sucede con el paralelismo dinámico, el cual se ha comprobado no ser lo suficientemente eficiente para las labores en las que lo necesitamos. Sin embargo, se ha hecho uso de él ya que se considera que, si bien en esta aplicación no es lo óptimo, en muchas otras sí que lo puede ser. Aporta ventajas que no pueden ser alcanzadas si no es con su uso.

También se ha desarrollado un caso particular, el cual encaja a la perfección con la forma de los kernel que se han desarrollado en este trabajo. El bagging para la identificación por mínimos cuadrados nos exige dar un nivel de paralelismo más para conseguir los objetivos, que influye de manera positiva en términos estadísticos.

En este caso, no queda comprobado que el uso del bagging sea algo necesario para la estimación por mínimos cuadrados de una recta. Esto es debido a que dicho estimador trabaja de forma óptima en el caso lineal, por lo que el bagging no mejora con creces los resultados que se obtendrían con un solo gran conjunto.

Sin embargo, esto no quiere decir que el bagging no sea útil para lo que se propone. Será necesario verificar si esto nos proporciona mejoras en sistemas de grados mayores, más allá de una estimación lineal. Seguramente ahí, donde el uso del estimador de mínimos cuadrados ya no está tan optimizado, el bagging nos aporte sus ventajas estadísticas.

5.2 Líneas de continuación

El carácter de este trabajo deja muy bien definidas las líneas de continuación. Por un lado, es necesario llevar todo el conocimiento de la arquitectura CUDA a un nivel más, desarrollando funciones en las que se busque

mucha más optimización y eficiencia, más allá de sentar las bases de este lenguaje de programación y dar a conocer los aspectos básicos. Con ello, también se debería intentar incorporar una concurrencia GPU-CPU, en la que podamos observar que ocurre si combinamos las ventajas de ambos métodos.

Esto mismo también es aplicable al bagging. Sería oportuno implementar mejoras en el código, que permitan llevar la estimación por mínimos cuadrados a sistemas más complejos, más allá de lo lineal. A su vez, también se podría trabajar más en el nivel estructural de este, haciendo que se reparta mejor la computación, y no quede toda la carga de cada conjunto en hilos independientes, ya que estos tienen muy poca capacidad.

Todo lo mencionado anteriormente no sería posible si no se hubieran sentado las bases aquí escritas, ya que las conclusiones obtenidas pueden ser el origen de otros proyectos más complejos.

ANEXOS

A. Código de la factorización de Cholesky en CPU/GPU

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <time.h>
#include <Windows.h>
#include "cublas_v2.h"

#define BlocksPerGrid 5           // valor maximo: 65535 por dimension
#define ThreadsPerBlock 30        // valor maximo: 1024 (x,y) / 64 (z)

#define tam 200                                 // tamaño de cada submatriz
#define num (BlocksPerGrid * ThreadsPerBlock)   // ---PARA GPU---    numero de
submatrices. se genera como combinación de hilos y bloques para evitar problemas de
indexado al lanzar el kernel
//#define num 40960                     // ---PARA CPU---    numero de submatrices.

#define fila (tam * num + num)                  // longitud de una fila en una matriz
grande (que contiene muchas submatrices)

double performancecounter_diff(LARGE_INTEGER *a, LARGE_INTEGER *b) {
	/* Funcion para medir tiempos de ejecucion durante la factorizacion en la CPU. */
	LARGE_INTEGER freq;
	QueryPerformanceFrequency(&freq);
	return (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;
}

int * crea_mat_sdp(int *randmat, int *randmattransp, int *sdpmat, int dim) {

	int i, j, k;
	float sum = 0;

	//	Generación de matriz triangular inferior aleatoria 'randmat'
	//  NOTA: términos de la diagonal >0 para evitar indeterminaciones
	for (i = 0; i < dim; i++)
		for (j = 0; j < dim; j++)
			if (i >= j)
				*(randmat + i*dim + j) = 1 + (rand() % 5);
			else
				*(randmat + i*dim + j) = 0;

	//	Traspuesta de 'randmat'
	for (i = 0; i < dim; i++)
		for (j = 0; j < dim; j++)
			*(randmattransp + i*dim + j) = *(randmat + j*dim + i);

	//	Matriz producto (Matriz definida positiva)
	for (i = 0; i < dim; i++) {
		for (j = 0; j < dim; j++) {
			for (k = 0; k < dim; k++) {
				sum += (*(randmat + dim*i + k)) * (*(randmattransp + k*dim +
j));
			}
			*(sdpmat + i*dim + j) = sum;
```

```
                        sum = 0;
                }
        }

        return sdpmat;

}

void crea_term_indep(float *h_Bfact, int *term, int dim) {

        int i, j, k;
        for (k = 0; k < num; k++) {
                for (i = 0; i < tam; i++) {
                        *(term + i) = 0;
                        for (j = 0; j <= i; j++) {
                                *(term + i) = *(term + i) + *(h_Bfact + i*fila + k*dim + k +
j);
                        }
                        *(h_Bfact + i*fila + k*dim + k + tam) = *(term + i);
                }
        }

}

float * resuelve_sistema(float *h_Bfact, float *sols, int dim) {

        // resolucion de los sistemas de ecuaciones
        // conjunto de soluciones de tamaño tam*num
        int i, j, k;
        float sum;

        for (k = 0; k < num; k++) {
                for (i = 0; i < dim; i++) {
                        sum = 0;
                        for (j = 0; j < i; j++) {
                                sum = sum + (*(sols + j*num + k) * *(h_Bfact + i*fila + k*dim
+ k + j));
                        }
                        *(sols + i*num + k) = (*(h_Bfact + i*fila + k*dim + k + dim) - sum) /
*(h_Bfact + i*fila + k*dim + k + i);
                }
        }

        return sols;
}

void comprueba_soluciones(float *sols) {
        // Comprueba que todas las soluciones del sistema valen 1.
        // Este comprobador solo es válido si se usa la funcion crea_term_indep para...
        // ... generar los terminos independientes de las matrices.

        int i, j, aux = 0;

        for (i = 0; i < tam; i++) {
                for (j = 0; j < num; j++) {
                        if (*(sols + i*num + j) != 1) {
                                printf("ERROR: Error al resolver el sistema\n");
                                aux = 0;
                                break;
                        }
                        else {
                                aux = 1;
                        }
```

```
		}
	}

	if (aux == 1)
		printf("Exito al resolver el sistema!\n");

}

__global__ void cholesky_gpu(float *m, float *Bfact, int dim)
{
	// -- m es una matriz de tamaño 'tam*tam*num', que contiene a 'num' submatrices
	// -- Bfact es una matriz de tamaño 'tam*tam*num', que contiene a 'num'
submatrices factorizadas
	// -- dim es la dimensión de cada submatriz (definida cte. por 'tam')

	int i = 0, j = 0, k = 0;
	double s = 0, t = 0;

	// idx_sub es el número de la submatriz que queremos factorizar [0, (num-1)]
	// es el elemento para indexar la paralelización
	int idx_sub = blockIdx.x * ThreadsPerBlock + threadIdx.x;

	if (idx_sub < num)
	{
		//	para poner a 0 los términos por encima de la diagonal
		for (i = 0; i < dim; i++) {
			for (j = 0; j < dim + 1; j++) {
				if (j > i)
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = 0;
				if (j == dim)
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = *(m +
i*fila + idx_sub*dim + j + idx_sub);}}

		__syncthreads();

		//	algoritmo de la factorización
		for (i = 0; i < dim; i++) {
			for (j = 0; j < (i + 1); j++){
				if (i == j) {
					s = 0;
					for (k = 0; k < j; k++) {
						s += (*(Bfact + i*fila + idx_sub*dim + k
+ idx_sub)) * (*(Bfact + i*fila + idx_sub*dim + k + idx_sub));
					}
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) =
sqrt(*(m + i*fila + idx_sub*dim + j + idx_sub) - s);
				}
				if (i != j) {
					t = 0;
					for (k = 0; k < j; k++) {
						t += (*(Bfact + i*fila + idx_sub*dim + k
+ idx_sub)) * (*(Bfact + j*fila + idx_sub*dim + k + idx_sub));
					}
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) =
(*(m + i*fila + idx_sub*dim + j + idx_sub) - t) / *(Bfact + j*fila + idx_sub*dim + j +
idx_sub);
				}
			}
		}

		__syncthreads();

	}
```

```
}

void cholesky_cpu(float *m, float *Bfact, int dim) {

	int i = 0, j = 0, k = 0, idx_sub;
	double s = 0, t = 0;

	// idx_sub es el número de la submatriz que queremos factorizar [0, (num-1)]

	for (idx_sub = 0; idx_sub < num; idx_sub++)
	{

		//	para poner a 0 los términos por encima de la diagonal
		for (i = 0; i < dim; i++) {
			for (j = 0; j < dim + 1; j++) {
				if (j > i)
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = 0;
				if (j == dim)
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = *(m +
i*fila + idx_sub*dim + j + idx_sub);
			}
		}

		//	algoritmo de la factorización

		for (i = 0; i < dim; i++) {
			for (j = 0; j < (i + 1); j++) {
				if (i == j) {
					s = 0;
					for (k = 0; k < j; k++) {
						s += (*(Bfact + i*fila + idx_sub*dim + k +
idx_sub)) * (*(Bfact + i*fila + idx_sub*dim + k + idx_sub));
					}
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) =
sqrt(*(m + i*fila + idx_sub*dim + j + idx_sub) - s);
				}
				if (i != j) {
					t = 0;
					for (k = 0; k < j; k++) {
						t += (*(Bfact + i*fila + idx_sub*dim + k +
idx_sub)) * (*(Bfact + j*fila + idx_sub*dim + k + idx_sub));
					}
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = (*(m +
i*fila + idx_sub*dim + j + idx_sub) - t) / *(Bfact + j*fila + idx_sub*dim + j + idx_sub);
				}
			}
		}

	}

}

void main(void)
{
	int i, j, k;
	//clock_t t_ini, t_fin;				// tipos para cálculo de tiempos en
CPU
	//LARGE_INTEGER iniTime, endTime;	// tipos para cálculo de tiempos en CPU
	//double t_crear, t_fact;
	cudaError_t error;
	cudaEvent_t start, stop;	// tipos para cálculo de tiempos en GPU
	float tiempo;
```

```
	// declaracion y reserva de memoria para matrices en 'host'  y 'device'

	float *h_m = (float *)malloc(tam*fila * sizeof(float)),
		*h_Bfact = (float *)malloc(tam*fila * sizeof(float)),
		*sols = (float *)malloc(tam*num * sizeof(float));

	float *d_m, *d_Bfact;

	error = cudaMalloc((void **)&d_m, tam*fila * sizeof(float));
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaMalloc((void **)&d_Bfact, tam*fila * sizeof(float));
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	int *A = (int *)malloc(tam*tam * sizeof(int)),
		*B = (int *)malloc(tam * sizeof(int)),
		*randmat = (int *)malloc(sizeof(int)*tam*tam),
		*randmattransp = (int *)malloc(sizeof(int)*tam*tam);

	if (h_m == NULL || h_Bfact == NULL || A == NULL || B == NULL || sols == NULL ||
randmat == NULL || randmattransp == NULL) {
		printf("ERROR: imposible reservar memoria (h_m, h_Bfact, A, B, sols)\n");
		exit(EXIT_FAILURE);
	}

	// generacion de matrices apropiadas para la factorizacion

	printf("\nGenerando %d matrices de dimensiones %d x %d...\n", num, tam, tam);

	for (k = 0; k < num; k++) {
		srand(2 * k + time(NULL));		// Cambio de semilla para generar
matrices siempre diferentes
		A = crea_mat_sdp(randmat, randmattransp, A, tam);
		for (i = 0; i < tam; i++) {
			for (j = 0; j < (tam + 1); j++) {
				if (j == tam) {
					*(h_m + i*fila + k*tam + k + j) = 0;
				}
				else {
					*(h_m + i*fila + k*tam + k + j) = *(A + i*tam + j);
				}
			}
		}
	}

	// copia de datos desde 'host' a 'device'

	error = cudaMemcpy(d_m, h_m, tam*fila * sizeof(float), cudaMemcpyHostToDevice);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}
```

```
	error = cudaMemcpy(d_Bfact, h_Bfact, tam*fila * sizeof(float),
cudaMemcpyHostToDevice);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	// lanzamiento del kernel
	printf("\nFactorizando %d matrices de dimensiones %d x %d...\n", num, tam, tam);

	error = cudaEventCreate(&start);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaEventCreate(&stop);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaEventRecord(start, 0);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	cholesky_gpu << < BlocksPerGrid, ThreadsPerBlock >> >(d_m, d_Bfact, tam);

	// QueryPerformanceCounter(&iniTime);

	//cholesky_cpu(h_m, h_Bfact, tam);

	//QueryPerformanceCounter(&endTime);

	//printf("Tiempo en crear %d submatrices de dimensiones %d x %d: %.16g
milisegundos\n", num, tam, tam, 1000*performancecounter_diff(&endTime, &iniTime));

	error = cudaEventRecord(stop, 0);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaEventSynchronize(start);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaEventSynchronize(stop);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}
```

```
	error = cudaEventElapsedTime(&tiempo, start, stop);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	printf("Tiempo en factorizar %d submatrices de dimensiones %d x %d: %3.5f
milisegundos\n\n", num, tam, tam, tiempo);

	error = cudaDeviceSynchronize();
	if (error != cudaSuccess) {
	printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
	exit(0);
	}

	// copia de datos de 'device' a 'host'

	error = cudaMemcpy(h_m, d_m, tam*fila * sizeof(float), cudaMemcpyDeviceToHost);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaMemcpy(h_Bfact, d_Bfact, tam*fila * sizeof(float),
cudaMemcpyDeviceToHost);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	// creacion de los terminos independientes
	// se generan como combinacion lineal de los coeficientes
	crea_term_indep(h_Bfact, B, tam);

	// resolucion de los sistemas
	sols = resuelve_sistema(h_Bfact, sols, tam);

	// comprobacion de los resultados
	comprueba_soluciones(sols);

	// liberacion de memoria reservada
	free(A);
	free(B);
	free(h_m);
	free(h_Bfact);
	free(sols);
	free(randmat);
	free(randmattransp);

	error = cudaFree(d_m);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaFree(d_Bfact);
	if (error != cudaSuccess) {
```

```
            printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
            exit(0);
      }

      printf("\n");

}
```

B. Código de la factorización de Cholesky en múltiples GPUs

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <time.h>
#include <Windows.h>

#define BlocksPerGrid 10                // valor maximo: 65535 por dimension
#define ThreadsPerBlock 10        // valor maximo: 1024 (x,y) / 64 (z)
#define tam 250                                 // tamaño de submatrices tam*tam
#define NUM_GPUs 1                      // numero de GPUs involucradas
//#define num 40960
      // ---PARA CPU---    numero de submatrices.
#define num (NUM_GPUs * BlocksPerGrid * ThreadsPerBlock)     // ---PARA GPU---    numero
de submatrices.
#define fila (tam * num + num)


double performancecounter_diff(LARGE_INTEGER *a, LARGE_INTEGER *b) {
      LARGE_INTEGER freq;
      QueryPerformanceFrequency(&freq);
      return (double)(a->QuadPart - b->QuadPart) / (double)freq.QuadPart;

}

int * crea_mat_sdp(int *randmat, int *randmattransp, int *sdpmat, int dim) {

      int i, j, k;
      float sum = 0;

      //    Generación de matriz triangular inferior aleatoria 'randmat'
      //  NOTA: términos de la diagonal >0 para evitar indeterminaciones
      for (i = 0; i < dim; i++)
            for (j = 0; j < dim; j++)
                  if (i >= j)
                        *(randmat + i*dim + j) = 1 + (rand() % 5);
                  else
                        *(randmat + i*dim + j) = 0;

      //    Traspuesta de 'randmat'

      for (i = 0; i < dim; i++)
            for (j = 0; j < dim; j++)
                  *(randmattransp + i*dim + j) = *(randmat + j*dim + i);

      //    Matriz producto (Matriz definida positiva)

      for (i = 0; i < dim; i++) {
            for (j = 0; j < dim; j++) {
                  for (k = 0; k < dim; k++) {
```

```
                        sum += (*(randmat + dim*i + k)) * (*(randmattransp + k*dim +
j));
                }
                *(sdpmat + i*dim + j) = sum;
                sum = 0;
            }
        }

        return sdpmat;

}
void crea_term_indep(float *h_Bfact, int *term, int num_submats, int filas_submats, int
dim) {

        int i, j, k;
        for (k = 0; k < num_submats; k++) {
            for (i = 0; i < tam; i++) {
                *(term + i) = 0;
                for (j = 0; j <= i; j++) {
                    *(term + i) = *(term + i) + *(h_Bfact + i*filas_submats +
k*dim + k + j);
                }
                *(h_Bfact + i*filas_submats + k*dim + k + tam) = *(term + i);
            }
        }

}

float * resuelve_sist_cpu(float *h_Bfact, float *sols, int num_submats, int
filas_submats, int dim) {

        // resolucion de los sistemas de ecuaciones
        // conjunto de soluciones de tamaño tam*num
        int i, j, k;
        float sum;

        for (k = 0; k < num_submats; k++) {
            for (i = 0; i < dim; i++) {
                sum = 0;
                for (j = 0; j < i; j++) {
                    sum = sum + (*(sols + j*num_submats + k) * *(h_Bfact +
i*filas_submats + k*dim + k + j));
                }
                *(sols + i*num_submats + k) = (*(h_Bfact + i*filas_submats + k*dim +
k + dim) - sum) / *(h_Bfact + i*filas_submats + k*dim + k + i);
            }
        }

        return sols;
}

__global__ void resuelve_sist_gpu(float *h_Bfact, float *sols, int num_submats, int
filas_submats, int dim){

        float sum;
        int i, j;
        int idx_sub = blockIdx.x * ThreadsPerBlock + threadIdx.x;

        if (idx_sub < num_submats){

            for (i = 0; i < dim; i++) {
                sum = 0;
                for (j = 0; j < i; j++) {
```

```
                        sum = sum + (*(sols + j*num_submats + idx_sub) * *(h_Bfact +
i*filas_submats + idx_sub*dim + idx_sub + j));
                    }
                    *(sols + i*num_submats + idx_sub) = (*(h_Bfact + i*filas_submats +
idx_sub*dim + idx_sub + dim) - sum) / *(h_Bfact + i*filas_submats + idx_sub*dim + idx_sub
+ i);
            }
            __syncthreads();
        }

}

void comprueba_soluciones(float *sols, int num_submats, int disp){

        int i, j, aux = 0;

        for (i = 0; i < tam; i++) {
            for (j = 0; j < num_submats; j++) {
                if (*(sols + i*num_submats + j) != 1) {
                    printf("ERROR: Error al resolver el sistema  en GPU %d\n",
disp);
                    aux = 0;
                    break;
                }
                else {
                    aux = 1;
                }
            }
        }
        if (aux == 1)
            printf("Exito al resolver el sistema en GPU %d!\n", disp);

}

__global__ void cholesky_gpu(float *m, float *Bfact, int num_submats, int fila_submats,
int dim)
{
        // -- m es una matriz de tamaño 'tam*tam*num', que contiene a 'num' submatrices
        // -- Bfact es una matriz de tamaño 'tam*tam*num', que contiene a 'num'
submatrices factorizadas
        // -- dim es la dimensión de cada submatriz (definida cte. por 'tam')
        // -- num_submats: numero de submatrices factorizadas en el kernel
        // -- fila_submats: tamaño de fila de num_submats matrices

        int i = 0, j = 0, k = 0;
        float s = 0, t = 0;

        // idx_sub es el número de la submatriz que queremos factorizar [0, (num-1)]
        // es el elemento para indexar la paralelización
        int idx_sub = blockIdx.x * ThreadsPerBlock + threadIdx.x;

        if (idx_sub < num_submats)
        {

            //     para poner a 0 los términos por encima de la diagonal
            for (i = 0; i < dim; i++) {
                for (j = 0; j < dim + 1; j++) {
                    if (j > i)
                        *(Bfact + i*fila_submats + idx_sub*dim + j + idx_sub) =
0;
                    if (j == dim)
                        *(Bfact + i*fila_submats + idx_sub*dim + j + idx_sub) =
*(m + i*fila_submats + idx_sub*dim + j + idx_sub);}}
```

```
		__syncthreads();

		//	algoritmo de la factorización

		for (i = 0; i < dim; i++) {
			for (j = 0; j < (i + 1); j++){
				if (i == j) {
					s = 0;
					for (k = 0; k < j; k++) {
						s += (*(Bfact + i*fila_submats + idx_sub*dim + k
+ idx_sub)) * (*(Bfact + i*fila_submats + idx_sub*dim + k + idx_sub));
					}
					*(Bfact + i*fila_submats + idx_sub*dim + j + idx_sub) =
sqrt(*(m + i*fila_submats + idx_sub*dim + j + idx_sub) - s);
				}
				if (i != j) {
					t = 0;
					for (k = 0; k < j; k++) {
						t += (*(Bfact + i*fila_submats + idx_sub*dim + k
+ idx_sub)) * (*(Bfact + j*fila_submats + idx_sub*dim + k + idx_sub));
					}
					*(Bfact + i*fila_submats + idx_sub*dim + j + idx_sub) =
(*(m + i*fila_submats + idx_sub*dim + j + idx_sub) - t) / *(Bfact + j*fila_submats +
idx_sub*dim + j + idx_sub);
				}
			}
		}

		__syncthreads();
	}
}

void cholesky_cpu(float *m, float *Bfact, int dim) {

	int i = 0, j = 0, k = 0, idx_sub;
	double s = 0, t = 0;

	// idx_sub es el número de la submatriz que queremos factorizar [0, (num-1)]

	for (idx_sub = 0; idx_sub < num; idx_sub++)
	{

		//	para poner a 0 los términos por encima de la diagonal
		for (i = 0; i < dim; i++) {
			for (j = 0; j < dim + 1; j++) {
				if (j > i)
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = 0;
				if (j == dim)
					*(Bfact + i*fila + idx_sub*dim + j + idx_sub) = *(m +
i*fila + idx_sub*dim + j + idx_sub);

			}
		}

		//	algoritmo de la factorización

		for (i = 0; i < dim; i++) {
			for (j = 0; j < (i + 1); j++) {
				if (i == j) {
					s = 0;
					for (k = 0; k < j; k++) {
```

```
                                        s += (*(Bfact + i*fila + idx_sub*dim + k +
idx_sub)) * (*(Bfact + i*fila + idx_sub*dim + k + idx_sub));
                                }
                                *(Bfact + i*fila + idx_sub*dim + j + idx_sub) =
sqrt(*(m + i*fila + idx_sub*dim + j + idx_sub) - s);
                        }
                        if (i != j) {
                                t = 0;
                                for (k = 0; k < j; k++) {
                                        t += (*(Bfact + i*fila + idx_sub*dim + k +
idx_sub)) * (*(Bfact + j*fila + idx_sub*dim + k + idx_sub));
                                }
                                *(Bfact + i*fila + idx_sub*dim + j + idx_sub) = (*(m +
i*fila + idx_sub*dim + j + idx_sub) - t) / *(Bfact + j*fila + idx_sub*dim + j + idx_sub);
                        }
                }
        }

    }

}

void main(void)
{
    // declaración de variables

    int i, j, k, detected_GPUs, aux;
    clock_t t_ini, t_fin;                       // tipos para cálculo de tiempos en
CPU
    LARGE_INTEGER iniTime, endTime;         // tipos para cálculo de tiempos en CPU
    double t_crear, t_fact;
    cudaError_t error;                          // tipo para comprobacion de
errores en GPU
    cudaEvent_t start, stop;                // tipo para cálculo de tiempos en GPU
    float tiempo;

    typedef struct{
        int tams;                   //numero de elementos de cada matriz mediana
(matriz "grande" de cada GPU)
        int num_submats;    //numero de submatrices en esta GPU
        int fila_submats;   //longitud de la fila en esta GPU
        int num_bloq;
        int num_hilos;
        float *h_sols;
        float *h_m;
        float *h_Bfact;
        float *d_sols;
        float *d_m;
        float *d_Bfact;
        cudaStream_t stream;
    } TGPUplan;

    error = (cudaGetDeviceCount(&detected_GPUs));
    if (error != cudaSuccess) {
        printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
        exit(0);
    }

    if (NUM_GPUs > detected_GPUs){
        printf("El numero de GPUs escogido (%d) es mayor al numero de GPUs
instaladas (%d)\nSolo se trabajara con %d GPUs\nSaliendo...\n", NUM_GPUs, detected_GPUs,
detected_GPUs);
```

```
		exit(EXIT_FAILURE);
	}

	TGPUplan plan[NUM_GPUs];

	for (i = 0; i < NUM_GPUs; i++){
		plan[i].tams = (num / NUM_GPUs) * tam*(tam + 1);
		plan[i].num_submats = num / NUM_GPUs;
	}

	// num no multiplo de NUM_GPUs
	for (i = 0; i < num % NUM_GPUs; i++){
		plan[i].tams += tam*(tam + 1);
		plan[i].num_submats++;
	}

	for (i = 0; i < NUM_GPUs; i++){
		plan[i].fila_submats = plan[i].num_submats * (tam+1);
		plan[i].num_bloq = BlocksPerGrid;
		plan[i].num_hilos = ThreadsPerBlock;
	}

	// declaracion y reserva de memoria para matrices en 'host'

	int *A = (int *)malloc(tam*tam * sizeof(int)),
		*B = (int *)malloc(tam * sizeof(int)),
		*randmat = (int *)malloc(sizeof(int)*tam*tam),
		*randmattransp = (int *)malloc(sizeof(int)*tam*tam);

	if (A == NULL || B == NULL || randmat == NULL || randmattransp == NULL) {
		printf("ERROR: imposible reservar memoria (h_m, h_Bfact, A, B, sols)\n");
		exit(EXIT_FAILURE);
	}

	// declaracion y reserva de memoria para matrices en 'host' y 'device'

	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaStreamCreate((&plan[disp].stream));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMalloc((void**)&plan[disp].d_m, plan[disp].tams *
sizeof(float));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMalloc((void**)&plan[disp].d_Bfact, plan[disp].tams *
sizeof(float));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
```

```
		error = cudaMalloc((void**)&plan[disp].d_sols, tam*plan[disp].num_submats *
sizeof(float));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMallocHost((void**)&plan[disp].h_m, plan[disp].tams *
sizeof(float));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMallocHost((void**)&plan[disp].h_Bfact, plan[disp].tams *
sizeof(float));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMallocHost((void**)&plan[disp].h_sols,
tam*plan[disp].num_submats * sizeof(float));
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}

	// generacion de matrices apropiadas para la factorizacion

	printf("\nGenerando %d matrices de dimensiones %d x %d en %d GPUs...\n", num, tam,
tam, NUM_GPUs);

	for (int disp = 0; disp < NUM_GPUs; disp++){
		srand((disp+4) * 2 + time(NULL));
		for (k = 0; k < plan[disp].num_submats; k++) {
			srand((disp+k) * (k+2) + time(NULL));		// Cambio de semilla
para generar matrices siempre diferentes
			A = crea_mat_sdp(randmat, randmattransp, A, tam);
			for (i = 0; i < tam; i++) {
				for (j = 0; j < (tam + 1); j++) {
					if (j == tam) {
						*(plan[disp].h_m + i*plan[disp].fila_submats +
k*tam + k + j) = 0;
					}
					else {
						*(plan[disp].h_m + i*plan[disp].fila_submats +
k*tam + k + j) = *(A + i*tam + j);
					}
				}
			}
		}
	}

	// copia de datos desde 'host' a 'device'

	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
```

```
                        exit(0);
                }

                error = cudaMemcpyAsync(plan[disp].d_m, plan[disp].h_m, plan[disp].tams *
sizeof(float), cudaMemcpyHostToDevice, plan[disp].stream);
                if (error != cudaSuccess) {
                        printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                        exit(0);
                }

                error = cudaMemcpyAsync(plan[disp].d_Bfact, plan[disp].h_Bfact,
plan[disp].tams * sizeof(float), cudaMemcpyHostToDevice, plan[disp].stream);
                if (error != cudaSuccess) {
                        printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                        exit(0);
                }
        }

        printf("Factorizando %d matrices de dimensiones %d x %d...\n", num, tam, tam);

        error = cudaEventCreate(&start);
        if (error != cudaSuccess) {
                printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                exit(0);
        }

        error = cudaEventCreate(&stop);
        if (error != cudaSuccess) {
                printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                exit(0);
        }

        error = cudaEventRecord(start, 0);
        if (error != cudaSuccess) {
                printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                exit(0);
        }

        // lanzamiento del kernel
        for (int disp = 0; disp < NUM_GPUs; disp++){
                error = cudaSetDevice(disp);
                if (error != cudaSuccess) {
                        printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                        exit(0);
                }
                cholesky_gpu << < BlocksPerGrid, ThreadsPerBlock, 0, plan[disp].stream >>
>(plan[disp].d_m, plan[disp].d_Bfact, plan[disp].num_submats, plan[disp].fila_submats ,
tam);
        }

        error = cudaEventRecord(stop, 0);
        if (error != cudaSuccess) {
                printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
                exit(0);
        }
```

```
	error = cudaEventSynchronize(start);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaEventSynchronize(stop);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	error = cudaEventElapsedTime(&tiempo, start, stop);
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	printf("\nTiempo en factorizar %d submatrices de dimensiones %d x %d: %3.5f
milisegundos\n", num, tam, tam, tiempo);

	error = cudaDeviceSynchronize();
	if (error != cudaSuccess) {
		printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
		exit(0);
	}

	printf("\n");


	// copia de datos de 'device' a 'host'
	for (int disp = 0; disp < NUM_GPUs; disp++){

		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMemcpyAsync(plan[disp].h_m, plan[disp].d_m, plan[disp].tams *
sizeof(float), cudaMemcpyDeviceToHost, plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}

		error = cudaMemcpyAsync(plan[disp].h_Bfact, plan[disp].d_Bfact,
plan[disp].tams * sizeof(float), cudaMemcpyDeviceToHost, plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}

	// liberacion de memoria reservada para la factorizacion
	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
```

```
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaStreamSynchronize(plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaFree(plan[disp].d_m);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}

	// creacion de los terminos independientes
	// se generan como combinacion lineal de los coeficientes
	printf("Generando terminos independientes en CPU...\n");
	for (int disp = 0; disp < NUM_GPUs; disp++){
		crea_term_indep(plan[disp].h_Bfact, B, plan[disp].num_submats,
plan[disp].fila_submats, tam);
	}


	// resolucion de los sistemas en GPU

	// copia de datos desde 'host' a 'device'
	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMemcpyAsync(plan[disp].d_sols, plan[disp].h_sols,
tam*plan[disp].num_submats * sizeof(float), cudaMemcpyHostToDevice, plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMemcpyAsync(plan[disp].d_Bfact, plan[disp].h_Bfact,
plan[disp].tams * sizeof(float), cudaMemcpyHostToDevice, plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}

	printf("Resolviendo sistemas en la GPU...\n\n");

	// lanzamiento del kernel
	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
```

```
		}
		error = cudaStreamSynchronize(plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		resuelve_sist_gpu << < BlocksPerGrid, ThreadsPerBlock, 0, plan[disp].stream
>> >(plan[disp].d_Bfact, plan[disp].d_sols, plan[disp].num_submats,
plan[disp].fila_submats, tam);
	}

	// copia de datos de 'device' a 'host'
	for (int disp = 0; disp < NUM_GPUs; disp++){

		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaMemcpyAsync(plan[disp].h_sols, plan[disp].d_sols,
tam*plan[disp].num_submats * sizeof(float), cudaMemcpyDeviceToHost, plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}

		error = cudaMemcpyAsync(plan[disp].h_Bfact, plan[disp].d_Bfact,
plan[disp].tams * sizeof(float), cudaMemcpyDeviceToHost, plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}

	// liberacion de memoria reservada
	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaStreamSynchronize(plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaFree(plan[disp].d_Bfact);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaFree(plan[disp].d_sols);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
```

```
		}
		error = cudaStreamDestroy(plan[disp].stream);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}


	// comprobacion de soluciones
	aux = 0;
	for (int disp = 0; disp < NUM_GPUs; disp++){
		comprueba_soluciones(plan[disp].h_sols, plan[disp].num_submats, disp);
	}

	// liberación del resto de memoria alojada

	free(A);
	free(B);
	free(randmat);
	free(randmattransp);

	for (int disp = 0; disp < NUM_GPUs; disp++){
		error = cudaSetDevice(disp);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaFreeHost(plan[disp].h_m);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaFreeHost(plan[disp].h_Bfact);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaFreeHost(plan[disp].h_sols);
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
		error = cudaDeviceReset();
		if (error != cudaSuccess) {
			printf("Cuda failure %s:%d: '%s'\n", __FILE__, __LINE__,
cudaGetErrorString(error));
			exit(0);
		}
	}
	printf("\n");

}
```

C. Código del bagging: *mincuad.cu*

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <curand.h>
#include <curand_kernel.h>
#include <cuda.h>
#include <device_launch_parameters.h>
#include "subconjuntos.cuh"

//#define ThreadsPerBlock 32
const int ThreadsPerBlock = 1024;
#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)
#define PERR(call) \
  if (call) {\
   fprintf(stderr, "%s:%d Error [%s] on "#call"\n", __FILE__, __LINE__,\
      cudaGetErrorString(cudaGetLastError()));\
   exit(1);\
    }
#define ERRCHECK \
  if (cudaPeekAtLastError()) { \
    fprintf(stderr, "%s:%d Error [%s]\n", __FILE__, __LINE__,\
       cudaGetErrorString(cudaGetLastError()));\
    exit(1);\
    }

__global__ void inv_kernel(int nvect, float *a_ig, float *c_og, int n)
{
       int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
       float *a_i, *c_o;
       if (idx < nvect){
              a_i = a_ig + idx*n*n;
              c_o = c_og + idx*n*n;

              int *p = (int *)malloc(n * sizeof(int));
              int *info = (int *)malloc(sizeof(int));
              int batch;
              cublasHandle_t hdl;
              cublasStatus_t status = cublasCreate_v2(&hdl);
              //printf("handle %d n = %d\n", status, n);

              info[0] = 0;
              batch = 1;
              float **a = (float **)malloc(sizeof(float *));
              *a = a_i;
              const float **aconst = (const float **)a;
              float **c = (float **)malloc(sizeof(float *));
              *c = c_o;

              status = cublasSgetrfBatched(hdl, n, a, n, p, info, batch);
              __syncthreads();
              if (status!=0) printf("inv_kernel idx %d, status: rf %d info %d\n", idx,
status, info[0]);    //Depuracion

              status = cublasSgetriBatched(hdl, n, aconst, n, p,
                     c, n, info, batch);
              __syncthreads();
```

```
		if (status != 0) printf("inv_kernel idx %d, status: ri %d info %d\n", idx, status, info[0]);	//Depuracion
		cublasDestroy_v2(hdl);
		//printf("exito\n"); //Depuracion
		free(p); free(info); free(a); free(c);
	}
}
__global__ void mult_kernel(int nvect, float *Ag, int Afilas, int Acols, float *Bg, int Bcols, float *Cg){
	/* Kernel que realiza la multiplicacion matricial INV * M' = RES mediante CUBLAS
	NOTA: CUBLAS usa como dimension principal las COLUMNAS (al contrario que C estandar, que usa filas)
	Para usar la misma notacion que en C, basta con intercambiar M' e INV en los argumentos de la llamada cublasSgemm */
	int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
	float *A, *B, *C;
	int tam_A = Afilas*Acols;
	int tam_B = Acols*Bcols;
	int tam_C = Afilas*Bcols;
	if (idx < nvect){
		/* asignacion de vector corresp. a cada hilo */
		A = Ag + idx*tam_A;
		B = Bg + idx*tam_B;
		C = Cg + idx*tam_C;

		const float alpha = 1.0f;
		const float beta = 0.0f;
		cublasHandle_t hdl;
		cublasStatus_t status = cublasCreate_v2(&hdl);
		//printf("handle %d\n", status);

		status = cublasSgemm_v2(hdl, CUBLAS_OP_N, CUBLAS_OP_N, Bcols, Afilas, Acols, &alpha, B, Bcols, A, Acols, &beta, C, Bcols);

		__syncthreads();
		if (status != 0) printf("mult_kernel idx %d, status: %d\n", idx, status);

		cublasDestroy_v2(hdl);
	}
}
__global__ void mult_trans_kernel(int nvect, float *Mg, int Mfilas, int Mcols, float *Cg){
	/*	Kernel que realiza la multiplicacion matricial M' * M = C mediante CUBLAS
		NOTA: CUBLAS usa como dimension principal las COLUMNAS (al contrario que C estandar, que usa filas)
		Para usar la misma notacion que en C, basta con intercambiar M' y M en los argumentos de la llamada cublasSgemm */
	int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
	float *M, *C;
	int tam_M = Mfilas*Mcols;
	int tam_C = Mcols*Mcols;
	if (idx < nvect){
		/* asignacion de vector corresp. a cada hilo */
		M = Mg + idx*tam_M;
		C = Cg + idx*tam_C;

		const float alpha = 1.0f;
		const float beta = 0.0f;
		cublasHandle_t hdl;
		cublasStatus_t status = cublasCreate_v2(&hdl);
		//printf("handle %d\n", status);
		status = cublasSgemm_v2(hdl, CUBLAS_OP_N, CUBLAS_OP_T, Mcols, Mcols, Mfilas, &alpha, M, Mcols, M, Mcols, &beta, C, Mcols);
```

```
		/* Parametros:
				- Supongo la matriz M almacenada en filas. Como CUBLAS usa el
almacenamiento en columnas, nuestra operacion
				M' * M se la pasaremos a CUBLAS como M * M'. Por ello marcamos
el flag CUBLAS_OP_T en el segundo argumento */
		__syncthreads();

		if (status != 0) printf("mult_trans_kernel idx %d, status: %d\n", idx,
status);

		cublasDestroy_v2(hdl);
		}
	}
__global__ void mult_trans_kernel_2(int nvect, float *Ag, int Afilas, int Acols, float
*Bg, int Bcols, float *Cg){
	/* Kernel que realiza la multiplicacion matricial INV * M' = RES mediante CUBLAS
	NOTA: CUBLAS usa como dimension principal las COLUMNAS (al contrario que C
estandar, que usa filas)
	Para usar la misma notacion que en C, basta con intercambiar M' e INV en los
argumentos de la llamada cublasSgemm */
	int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
	float *A, *B, *C;
	int tam_A = Afilas*Acols;
	int tam_B = Acols*Bcols;
	int tam_C = Afilas*Bcols;
	if (idx < nvect){
		/* asignacion de vector corresp. a cada hilo */
		A = Ag + idx*tam_A;
		B = Bg + idx*tam_B;
		C = Cg + idx*tam_C;

		const float alpha = 1.0f;
		const float beta = 0.0f;
		cublasHandle_t hdl;
		cublasStatus_t status = cublasCreate_v2(&hdl);
		//printf("handle %d\n", status);

		status = cublasSgemm_v2(hdl, CUBLAS_OP_T, CUBLAS_OP_N, Bcols, Acols,
Afilas, &alpha, B, Afilas, A, Acols, &beta, C, Bcols);

		__syncthreads();
		if (status != 0) printf("mult_trans_kernel_2 idx %d, status: %d\n", idx,
status);

		cublasDestroy_v2(hdl);
	}
}

int crea_subconj_y_mat(int d, int m, int num_sub, int tam_sub, int datomax, float *d_M,
float *d_Y)
{
	/* Necesaria libreria "subconjuntos.cuh" */
	/* Variables del problema */
	float *d_datos_x;			// Puntero a los datos (coord x)(GPU)
	float *d_datos_y;			// Puntero a los datos (coord y)(GPU)
	int *d_prueba_indice;			// Puntero a los subconjuntos (indice)
	(GPU)
	float *d_prueba_dato;			// Puntero a los subconjuntos (dato)
	(GPU)
	//int datomin = 10;			// Dato minimo para evitar redondeos a 0
al generar los puntos X

	curandState *d_states_ygen;
```

```
	curandState *d_states_subconj;				// states para cuRAND. Se crean tantos como vectores haya

	/* Reserva de memoria */
	CUDA_CALL(cudaMalloc((void **)&d_datos_x, d*sizeof(float)));
	CUDA_CALL(cudaMalloc((void **)&d_datos_y, d*sizeof(float)));
	CUDA_CALL(cudaMalloc((void **)&d_prueba_indice, num_sub*tam_sub*m*sizeof(int)));
	CUDA_CALL(cudaMalloc((void **)&d_prueba_dato, num_sub * 2 * tam_sub*m*sizeof(float)));
	CUDA_CALL(cudaMalloc((void **)&d_states_subconj, m*sizeof(curandState)));
	CUDA_CALL(cudaMalloc((void **)&d_states_ygen, d*sizeof(curandState)));

	//printf("Trabajando en la GPU:\n\tSe generaran %d vectores de %d subconjuntos\n\tValor maximo de cada dato: %d\n\tTotal de puntos generados: %d\n\n", m, num_sub, datomax, d);

	/* Se genera un puntero de datos de m vectores y n datos por vector
	Se utilizan ordenes desde el host (host API)	*/
	gen_datos_x(d, d_datos_x);

	/* Se escalan estos datos entre [0, datomax] */
	escala_datos << < ((d) + ThreadsPerBlock - 1) / ThreadsPerBlock, ThreadsPerBlock >> > (datomax, d, d_datos_x);
	cudaDeviceSynchronize();
	ERRCHECK;

	/* Se generan las coordenadas Y para cada dato */
	gen_datos_y << < ((d) + ThreadsPerBlock - 1) / ThreadsPerBlock, ThreadsPerBlock >> > (d, d_datos_x, d_datos_y, d_states_ygen);
	cudaDeviceSynchronize();
	ERRCHECK;

	/* Se copia al host y se imprimen (también en archivo CSV) */
	//copia_datos(d, d_datos_x, d_datos_y);

	/* Se crean los subconjuntos */
	crea_subconj << < (m + ThreadsPerBlock - 1) / ThreadsPerBlock, ThreadsPerBlock >> > (m, d, tam_sub, num_sub, d_prueba_indice, d_prueba_dato, d_datos_x, d_datos_y, d_states_subconj);
	cudaDeviceSynchronize();
	ERRCHECK;

	/* Se copian al host y se imprimen */
	//copia_subconj(m, num_sub, tam_sub, d_prueba_indice, d_prueba_dato);

	/* Inicializacion de matrices M e Y */
	crea_mat << < (m + ThreadsPerBlock - 1) / ThreadsPerBlock, ThreadsPerBlock >> > (m, num_sub, tam_sub, d_prueba_dato, d_M, d_Y);
	cudaDeviceSynchronize();
	ERRCHECK;

	/* Copia M e Y al host e imprime por pantalla */
	//copia_mat(m, num_sub, tam_sub, d_M, d_Y);



	CUDA_CALL(cudaFree(d_datos_x));
	CUDA_CALL(cudaFree(d_datos_y));
	CUDA_CALL(cudaFree(d_prueba_indice));
	CUDA_CALL(cudaFree(d_prueba_dato));
	CUDA_CALL(cudaFree(d_states_subconj));
	CUDA_CALL(cudaFree(d_states_ygen));
```

```
	printf("\n");
	return(0);

}
float* mat_op(int nvect, int num_subconj, int tam_subconj, float *d_M, float *d_Y){
	/* Realiza las operaciones matriciales para el estimador de minimos cuadrados */
	/* Hay que tener en cuenta que CUBLAS debe recibir los punteros de matrices
	individuales, no tal y como estan guardadas en M e Y (todas juntas)
	Para ello se intentara que cada en cada kernel, cada hilo procese un vector  */

	/* Multiplicacion  MULT = [Mtras * M] */
	int Mfil = num_subconj;
	int Mcol = tam_subconj;
	int Ycol = 1;

	//float *M;
	//PERR(cudaMallocHost(&M, nvect*num_subconj*tam_subconj*sizeof(float)));
	//PERR(cudaMemcpy(M, d_M, nvect*num_subconj*tam_subconj*sizeof(float),
cudaMemcpyDeviceToHost));
	//
	//for (int k = 0; k < nvect; k++){
	//	printf("\nMatriz M[%d]:\n\n", k);
	//	for (int i = 0; i < num_subconj; i++){
	//		for (int j = 0; j < tam_subconj; j++){
	//			printf("%7.4f  ", *(M + k*num_subconj*tam_subconj +
i*tam_subconj + j));
	//		}
	//		printf("\n");
	//	}
	//	printf("\n");
	//}
	//PERR(cudaFreeHost(M));


	float *d_MULT;
	//float *MULT;
	PERR(cudaMalloc(&d_MULT, nvect*tam_subconj*tam_subconj*sizeof(float)));
	//PERR(cudaMallocHost(&MULT, nvect*tam_subconj*tam_subconj*sizeof(float)));

	mult_trans_kernel << <((nvect)+ThreadsPerBlock - 1) / ThreadsPerBlock,
ThreadsPerBlock >> > (nvect, d_M, Mfil, Mcol, d_MULT);

	/* Se han intercambiado filas y columnas del primer argumento, ya que se
	ha activado el flag para trasponer la matriz dentro del kernel */
	PERR(cudaDeviceSynchronize());
	ERRCHECK;

	//PERR(cudaMemcpy(MULT, d_MULT, nvect*tam_subconj*tam_subconj*sizeof(float),
cudaMemcpyDeviceToHost));

	//for (int k = 0; k < nvect; k++){
	//	printf("\nMatriz MULT = [Mtras * M] num %d:\n\n", k);
	//	for (int i = 0; i < tam_subconj; i++){
	//		for (int j = 0; j < tam_subconj; j++){
	//			printf("%7.4f  ", *(MULT + k*tam_subconj*tam_subconj +
i*tam_subconj + j));
	//		}
	//		printf("\n");
	//	}
	//	printf("\n");
	//}
	/* Inversa INV = [MULT]^-1 */
```

```
	float *d_INV;
	//float *INV;
	PERR(cudaMalloc(&d_INV, nvect*tam_subconj*tam_subconj*sizeof(float)));
	//PERR(cudaMallocHost(&INV, tam_subconj*tam_subconj*sizeof(float)));

	inv_kernel << <((nvect)+ThreadsPerBlock - 1) / ThreadsPerBlock, ThreadsPerBlock >>
> (nvect, d_MULT, d_INV, tam_subconj);
	PERR(cudaDeviceSynchronize());
	ERRCHECK;

	//PERR(cudaMemcpy(INV, d_INV, tam_subconj*tam_subconj*sizeof(float),
cudaMemcpyDeviceToHost));

	//printf("\nMatriz INV = [MULT]^-1:\n\n");
	//for (int i = 0; i < tam_subconj; i++){
	//	for (int j = 0; j < tam_subconj; j++){
	//		printf("%7.4f  ", *(INV + i*tam_subconj + j));
	//	}
	//	printf("\n");
	//}
	//printf("\n");
	//PERR(cudaFreeHost(INV));

	PERR(cudaFree(d_MULT)); // ya no necesitamos esta matriz auxiliar

	/* Multiplicacion RES = [INV * Mtras] */
	int INVfil = tam_subconj;
	int INVcol = tam_subconj;

	float *d_RES;
	//float *RES;
	PERR(cudaMalloc(&d_RES, nvect*tam_subconj*num_subconj*sizeof(float)));
	//PERR(cudaMallocHost(&RES, tam_subconj*num_subconj*sizeof(float)));

	mult_trans_kernel_2 << <((nvect)+ThreadsPerBlock - 1) / ThreadsPerBlock,
ThreadsPerBlock >> > (nvect, d_INV, INVcol, INVfil, d_M, Mfil, d_RES);
	/* Kernel para multiplicar A*B'. El 5º parametro deben ser las filas de M, es
decir, las columnas de M'. */

	PERR(cudaDeviceSynchronize());
	ERRCHECK;

	//PERR(cudaMemcpy(RES, d_RES, tam_subconj*num_subconj*sizeof(float),
cudaMemcpyDeviceToHost));

	//printf("\nMatriz RES = [INV * Mtras]:\n\n");
	//for (int i = 0; i < tam_subconj; i++){
	//	for (int j = 0; j < num_subconj; j++){
	//		printf("%7.4f  ", *(RES + i*num_subconj + j));
	//	}
	//	printf("\n");
	//}
	//printf("\n");
	//PERR(cudaFreeHost(RES));

	PERR(cudaFree(d_INV));

	/* Multiplicacion THETA = [RES * Y] */
	int RESfil = tam_subconj;
	int REScol = num_subconj;

	float *d_THETA;
	PERR(cudaMalloc(&d_THETA, nvect*tam_subconj*sizeof(float)));
```

```
	mult_kernel << <((nvect)+ThreadsPerBlock - 1) / ThreadsPerBlock, ThreadsPerBlock
>> > (nvect, d_RES, RESfil, REScol, d_Y, Ycol, d_THETA);
	/* Kernel para multiplicar A*B. */

	PERR(cudaDeviceSynchronize());
	ERRCHECK;

	PERR(cudaFree(d_RES));
	return(d_THETA);
}

int main(int argc, char **argv)
{
	/* decalaracion variables MAIN */
	float *d_M;
	float *d_Y;
	float *d_THETA, *THETA;

	FILE *fp;
	fp = fopen("tiempos_bagging.csv", "w+");

	cudaEvent_t start, stop;
	float tiempo;

	/* declaracion variables SUBCONJUNTOS */
	int d = 20000;                                  // Numero de datos generados
(numero de puntos del problema general)
	int num_vect = 2000;                    // Numero de vectores
	int num_sub = 30000;                    // Numero de subconjuntos por vector
	int tam_sub = 2;                        // Tamaño de cada subconjunto (num.datos,
debe ser PAR)
	int datomax = 1000000;                          // Valor maximo de cada dato en
*d_datos_x   //tener en cuenta el RUIDO
	int nreps = 25;

	/* reserva de memoria matrices SUBCONJUNTOS */
	PERR(cudaMalloc((void **)&d_M, tam_sub*num_sub*num_vect*sizeof(float)));
	PERR(cudaMalloc((void **)&d_Y, num_sub*num_vect*sizeof(float)));
	PERR(cudaMallocHost(&THETA, num_vect*tam_sub*sizeof(float)));

	/* creacion de SUBCONJUNTOS */
	crea_subconj_y_mat(d, num_vect, num_sub, tam_sub, datomax, d_M, d_Y);

	printf("Total datos analizados: %d;\n\n", d);
	fprintf(fp, "Total datos analizados:\n%d\nNumero de conjuntos:\n%d\nPuntos por
conjunto:\n%d\n\nTHREADS_PER_BLOCK:\n%d\n\n", d, num_vect, num_sub, ThreadsPerBlock);
	fprintf(fp, "Se harán %d repeticiones y se promediaran los tiempos obtenidos\n\n",
nreps);

	PERR(cudaEventCreate(&start));
	PERR(cudaEventCreate(&stop));
	PERR(cudaEventRecord(start, 0));

	for (int nreps = 0; nreps < 25; nreps++){

		/* operaciones matematicas para la obtencion del vector de parametros
*/
		d_THETA = mat_op(num_vect, num_sub, tam_sub, d_M, d_Y);

		/* transferencia al host y comprobacion */
		PERR(cudaMemcpy(THETA, d_THETA, num_vect*tam_sub*sizeof(float),
cudaMemcpyDeviceToHost));
```

```
                    /* analisis de los datos obtenidos */
                    double sum_a = 0, sum_b = 0;
                    //printf("Num. conjuntos: %d     Puntos por conjunto: %d;\n",
num_vect, num_sub);
                    //fprintf(fp, "Num. conjuntos: %d     Puntos por conjunto: %d;\n",
num_vect, num_sub);
                    for (int j = 0; j < num_vect*tam_sub; j = j + 2){
                           //printf("%7.4f     ", THETA[j]);
                           //fprintf(fp, "%7.4f,", THETA[j]);
                           sum_a += THETA[j];
                           //printf("%7.4f\n\n", THETA[j + 1]);
                           //fprintf(fp, "%7.4f\n\n", THETA[j + 1]);
                           sum_b += THETA[j + 1];
                    }
                    //printf("\n");

                    long double med_a = sum_a / num_vect;
                    //printf("\nValor medio de a: %7.6f;     ", med_a);
                    //fprintf(fp, "\nValor medio de a: %7.6f;     ", med_a);
                    long double med_b = sum_b / num_vect;
                    //printf("\nValor medio de b: %7.6f;\n\n", med_b);
                    //fprintf(fp, "\nValor medio de b: %7.6f;\n\n", med_b);

                    PERR(cudaFree(d_THETA));

             }
             //}

             PERR(cudaEventRecord(stop, 0));
             PERR(cudaEventSynchronize(start));
             PERR(cudaEventSynchronize(stop));
             PERR(cudaEventElapsedTime(&tiempo, start, stop));

             printf("Tiempo total:\n%7.4f\n\nTiempo promedio: %7.4f ms\n\n", tiempo,
tiempo/(float)nreps);
             fprintf(fp, "Tiempo total:\n%7.4f\n\nTiempo promedio:\n%7.4f\n\n", tiempo, tiempo
/ (float)nreps);

             PERR(cudaFree(d_M));
             PERR(cudaFree(d_Y));
             PERR(cudaFreeHost(THETA));

             fclose(fp);

             PERR(cudaDeviceReset());

             return 0;

}
```

D. Código del bagging: *subconjuntos.cu*

```
#include <curand.h>
#include <curand_kernel.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <device_launch_parameters.h>
#include <time.h>
#include "subconjuntos.cuh"
```

```
#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)

//#define ThreadsPerBlock 32
const int ThreadsPerBlock = 1024;

int gen_datos_x(int d, float *d_dato){

       /*                          [host API] ---> se trabaja con ordenes en la CPU
       Genera m vectores de n numeros pseudoaleatorios en la direccion *d_dato de la
GPU.*/

       int i, j;
       curandGenerator_t gen;

       /* Create pseudo-random number generator */
       CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT));

       /* Set seed */
       CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen, 1111ull));

       /* Generate n floats on device */
       CURAND_CALL(curandGenerateUniform(gen, d_dato, d));

       //curandSetGeneratorOffset();

       /* Cleanup */
       CURAND_CALL(curandDestroyGenerator(gen));

       return EXIT_SUCCESS;
}
int copia_datos(int d, float *d_datos_x, float *d_datos_y){

       /* Copia los datos de d_dato al host y los imprime por pantalla y en el fichero FP
*/

       int i, j;
       /* Para comprobar en el host */
       float *h_datos_x;
       float *h_datos_y;
       FILE *fp;
       fp = fopen("varianza_bagging_X_Y.csv", "w+");

       /* Reserva de memoria para host */
       h_datos_x = (float *)malloc(d * sizeof(float));
       if (h_datos_x == NULL){
              printf("Error al reservar memoria para h_datos_x\nError en %s:%d\n",
__FILE__, __LINE__);
              return EXIT_FAILURE;
       }
       h_datos_y = (float *)malloc(d * sizeof(float));
       if (h_datos_y == NULL){
              printf("Error al reservar memoria para h_datos_y\nError en %s:%d\n",
__FILE__, __LINE__);
              return EXIT_FAILURE;
       }
```

```
	/* Copy device memory to host */
	CUDA_CALL(cudaMemcpy(h_datos_x, d_datos_x, d * sizeof(float),
cudaMemcpyDeviceToHost));
	CUDA_CALL(cudaMemcpy(h_datos_y, d_datos_y, d * sizeof(float),
cudaMemcpyDeviceToHost));

	/* Show result */
	//printf("Datos generados:\n");
	//fprintf(fp, "Datos generados:\n");
	//printf("Comp. X:\n");
	//fprintf(fp, "Comp. X:\n");
	//for (j = 0; j < d; j++){
	//	printf("%7.4f   ", *(h_datos_x + j));
	//	fprintf(fp, "%7.4f   ", *(h_datos_x + j));
	//}
	//printf("\n");
	//fprintf(fp, "\n");
	//printf("Comp. Y:\n");
	//fprintf(fp, "Comp. Y:\n");
	//for (j = 0; j < d; j++){
	//	printf("%7.4f   ", *(h_datos_y + j));
	//	fprintf(fp, "%7.4f   ", *(h_datos_y + j));
	//}
	//printf("\n\n");
	//fprintf(fp, "\n\n");

	fprintf(fp, "Datos generados:\n");
	fprintf(fp, "Comp. X:, Comp. Y:\n");
	for (j = 0; j < d; j++){
		fprintf(fp, "%7.4f, %7.4f\n", *(h_datos_x + j), *(h_datos_y + j));
	}
	fprintf(fp, "\n\n");

	fclose(fp);

	free(h_datos_x);
	free(h_datos_y);
}
int copia_subconj(int num_vect, int num_subconj, int tam_subconj, int *prueba_indice,
float *prueba_dato){

	/* Copia los datos de d_dato al host y los imprime por pantalla */

	int i, j, k;
	int total_datos = num_vect * num_subconj * tam_subconj;
	int *h_prueba_indice;	/* Para comprobar en el host */
	float *h_prueba_dato;

	/* Allocate n floats on host */
	h_prueba_indice = (int *)malloc(total_datos * sizeof(int));
	if (h_prueba_indice == NULL){
		printf("Error al reservar memoria para h_prueba_indice\nError en %s:%d\n",
__FILE__, __LINE__);
		return EXIT_FAILURE;
	}
	h_prueba_dato = (float *)malloc(total_datos * sizeof(float));
	if (h_prueba_indice == NULL){
		printf("Error al reservar memoria para h_prueba_dato\nError en %s:%d\n",
__FILE__, __LINE__);
		return EXIT_FAILURE;
	}

	/* Copy device memory to host */
```

```
	CUDA_CALL(cudaMemcpy(h_prueba_indice, prueba_indice, total_datos * sizeof(int),
cudaMemcpyDeviceToHost));
	CUDA_CALL(cudaMemcpy(h_prueba_dato, prueba_dato, total_datos * sizeof(float),
cudaMemcpyDeviceToHost));


	/* Show result */
	printf("Subconjuntos obtenidos de forma aleatoria:\n");
	for (i = 0; i < num_vect; i++){
		printf("\nVector numero %d:\n\n", i);
		for (j = 0; j < num_subconj; j++){
			printf("\tSubconj num %d:  ", j);
			for (k = 0; k < tam_subconj; k = k + 2){
				printf("X[%d]: %7.4f;   ", *(h_prueba_indice +
i*num_subconj*tam_subconj + j*tam_subconj + k), *(h_prueba_dato +
i*num_subconj*tam_subconj + j*tam_subconj + k));
				printf("Y[%d]: %7.4f;   ", *(h_prueba_indice +
i*num_subconj*tam_subconj + j*tam_subconj + k), *(h_prueba_dato +
i*num_subconj*tam_subconj + j*tam_subconj + k + 1));
				printf("\n");
			}
			printf("\n");
		}
	}

	free(h_prueba_indice);
	free(h_prueba_dato);
}
int copia_mat(int num_vect, int num_subconj, int tam_subconj, float *M, float *Y){
	/* Copia las matrices M e Y al host y las imprime */
	/* Reserva de memoria auxiliar para el host */
	float *h_M, *h_Y;
	h_M = (float *)malloc(2 * num_subconj*num_vect * sizeof(float));
	if (h_M == NULL){
		printf("Error al reservar memoria para h_M\nError en %s:%d\n", __FILE__,
__LINE__);
		return EXIT_FAILURE;
	}
	h_Y = (float *)malloc(num_subconj*num_vect * sizeof(float));
	if (h_Y == NULL){
		printf("Error al reservar memoria para h_Y\nError en %s:%d\n", __FILE__,
__LINE__);
		return EXIT_FAILURE;
	}

	/* Copia de memoria al host */
	CUDA_CALL(cudaMemcpy(h_M, M, 2 * num_subconj*num_vect * sizeof(float),
cudaMemcpyDeviceToHost));
	CUDA_CALL(cudaMemcpy(h_Y, Y, num_subconj*num_vect * sizeof(float),
cudaMemcpyDeviceToHost));

	/* Imprime el resultado */
	for (int i = 0; i < num_vect; i++){
		printf("Matriz M[%d]\n", i);
		for (int j = 0; j < num_subconj; j++){
			for (int k = 0; k < 2; k++){
				printf("%7.4f  ", *(h_M + i*tam_subconj*num_subconj + 2 * j +
k));
			}
			printf("\n");
		}
		printf("\n");
	}
```

```
	for (int i = 0; i < num_vect; i++){
		printf("Matriz Y[%d]\n", i);
		for (int j = 0; j < num_subconj; j++){
			printf("%7.4f\n", *(h_Y + i*num_subconj + j));
		}
		printf("\n");
	}

	free(h_M); free(h_Y);
}

__global__ void escala_datos(int dato_max, int ndatos, float *d_dato){

	/* Reescala los datos aleatorios a numeros en el conjunto [0, ndatos]
	y asigna valores a estos datos entre [0, dato_max]	*/

	int idx = blockIdx.x * ThreadsPerBlock + threadIdx.x;

	if (idx < ndatos){
		*(d_dato + idx) = *(d_dato + idx) * dato_max;
	}
	__syncthreads();
}
__global__ void gen_datos_y(int d, float *x, float *y, curandState *state){

	/* Funcion para "obtener" y en funcion de x */
	/* Introducimos ruido en el termino independiente */

	int idx = blockIdx.x * ThreadsPerBlock + threadIdx.x;

	unsigned long seed = 1;
	float ruido;
	float escala_ruido = 200000;
	curand_init(seed, idx, 0, &state[idx]);
	curandState localState = state[idx];
	/* El ruido será un flotante entre [ 0, 2*escala_ruido ] */
	/* Se resta escala_ruido para dejarlo entre [ -escala_ruido , escala_ruido ] */
	ruido = (curand_uniform(&localState) * 2 * escala_ruido) - escala_ruido;

	if (idx < d){
		/* funcion de la recta que forman los puntos (x,y) */
		*(y + idx) = (/*2 **/ *(x + idx)) + ruido;
	}
	__syncthreads();
	state[idx] = localState;
}
__global__ void crea_subconj(int num_vect, int ndatos, int tam_subconj, int num_subconj,
int *prueba_indice, float *prueba_dato, float *d_dato_x, float *d_dato_y, curandState
*state){

	/*		[device API] ---> se trabaja en la GPU
	Crea 'num_subconj' conjuntos de tamaño 'tam_subconj'.
	[[Se intentara que los subconjuntos se identifiquen por direccion de memoria.
	Asi evitamos repetir y ocupar memoria al copiar el dato]]	*/

	int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
	unsigned long seed = 1;
	unsigned int x;
	curand_init(seed, idx, 0, &state[idx]);
	curandState localState = state[idx];
	/* Cada hebra procesa un vector */
	if (idx < num_vect){
```

```
		/* Creacion de los subconjuntos */
		for (int i = 0; i < num_subconj; i++){
			for (int j = 0; j < tam_subconj; j = j + 2){

				/* Se asigna el flotante obtenido de curand_uniform
				al entero x, redondeando hacia el entero mas bajo.
				curand devuelve un flotante en [0,1] y lo escalamos
				con tam_vect para elegir una posicion del vector de datos */
				x = (curand_uniform(&localState) * ndatos);

				/* La siguiente asignacion es clave. Es donde podría
				asignarse la direccion de memoria en vez del dato (?)
				Las dos asignaciones a continuacion necesitan reservar
				en memoria dichos punteros (querriamos evitarlo) */

				/* Se almacena el indice de cada valor */
				*(prueba_indice + idx*num_subconj*tam_subconj + i*tam_subconj
+ j) = x;

				/* Se almacena el propio dato */
				/* Dato X */
				*(prueba_dato + idx*num_subconj*tam_subconj + i*tam_subconj +
j) = *(d_dato_x + x);
				/* Dato Y */
				*(prueba_dato + idx*num_subconj*tam_subconj + i*tam_subconj +
j + 1) = *(d_dato_y + x);
				/* Tenemos todos los puntos (x,y) guardados en prueba_dato */

			}
		}
	}
	__syncthreads();
	state[idx] = localState;

}
__global__ void crea_mat(int num_vect, int num_subconj, int tam_subconj, float
*prueba_dato, float *M, float *Y){
	/* Inicializacion de las matrices M e Y  */
	int idx = blockIdx.x*ThreadsPerBlock + threadIdx.x;
	if (idx < num_vect){
		for (int i = 0; i < num_subconj; i++){
			for (int j = 0; j < tam_subconj; j = j + 2){
				/* ------------------------- MATRICES ----------------------
--- */
				/* Se inicializan las matrices M e Y para el calculo de
parametros */

				*(M + idx*tam_subconj*num_subconj + i*tam_subconj + j) = 1;
				*(M + idx*tam_subconj*num_subconj + i*tam_subconj + j + 1) =
*(prueba_dato + idx*num_subconj*tam_subconj + i*tam_subconj + j);
				*(Y + idx*num_subconj + i) = *(prueba_dato +
idx*num_subconj*tam_subconj + i*tam_subconj + j + 1);

				/* ------------------------------------------------------------
--- */
			}
		}
	}
	__syncthreads();
}
```

E. Código del bagging: *subconjuntos.cuh*

```
#include <device_launch_parameters.h>

#ifndef SUBCONJUNTOS_CUH
#define SUBCONJUNTOS_CUH

int gen_datos_x(int d, float *d_dato);
int copia_datos(int d, float *d_datos_x, float *d_datos_y);
int copia_subconj(int num_vect, int num_subconj, int tam_subconj, int *prueba_indice,
float *prueba_dato);
int copia_mat(int num_vect, int num_subconj, int tam_subconj, float *M, float *Y);
__global__ void gen_datos_y(int d, float *x, float *y, curandState *state);
__global__ void escala_datos(int dato_max, int ndatos, float *d_dato);
__global__ void crea_subconj(int num_vect, int ndatos, int tam_subconj, int num_subconj,
int *prueba_indice, float *prueba_dato, float *d_dato_x, float *d_dato_y, curandState
*state);
__global__ void crea_mat(int num_vect, int num_subconj, int tam_subconj, float
*prueba_dato, float *M, float *Y);

#endif
```

REFERENCIAS

[1] Sanders, J. y Kandrot, E. *CUDA by example, An Introduction to General-Purpose GPU Programming.* 2010.

[2] James, G., Witten, D., Hastie, T., Tibshirani, R. *An Introduction to Statistical Learning.* 2013.

[3] Hastie, T., Tibshirani, R., Friedman, J. *The Elements of Statistical Learning*. 2009.

[4] Zeller, C. *CUDA C/C++ Basics, Supercomputing 2011 Tutorial*. 2011.

[5] NVIDIA Corporation. *CUDA Toolkit Documentation. CUDA C Programming Guide*. [En línea] Consultado el 14 de febrero de 2018. Disponible en: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[6] Stack Overflow. *nvidia simpleDevLibCUBLAS error.* [En línea] Consultado el 5 de junio de 2018. Disponible en: https://stackoverflow.com/questions/37653654/nvidia-simpledevlibcublas-error

[7] Stack Overflow. *CUDA 5.0 separate compilation and linking option in Visual Studio.* [En línea] Consultado el 12 de junio de 2018. Disponible en: https://stackoverflow.com/questions/15192787/cuda-5-0-separate-compilation-linking-option-and-visual-studio-2010

[8] NVIDIA Corporation. *CUDA Toolkit Documentation. cuBLAS*. [En línea] Consultado el 20 de mayo de 2018. Disponible en: https://docs.nvidia.com/cuda/cublas/

[9] NVIDIA Corporation. *CUDA Toolkit Documentation. cuRAND*. [En línea] Consultado el 13 de mayo de 2018. Disponible en: https://docs.nvidia.com/cuda/curand/

[10] Rosetta Code. *Cholesky decomposition*. [En línea] Consultado el 26 de marzo de 2018. Disponible en: https://rosettacode.org/wiki/Cholesky_decomposition

[11] Terraludus. *Consejos C++: midiendo tiempos*. [En línea] Consultado el 1 de abril de 2018. Disponible en: http://terraludus.blogspot.com/2012/06/consejos-c-ii-midiendo-tiempos.html

[12] Stack Overflow. *Different occupancy between calculator and nvprof*. [En línea] Consultado el 2 de mayo de 2018. Disponible en: https://stackoverflow.com/questions/23469180/different-occupancy-between-calculator-and-nvprof

[13] Stack Overflow. *Re-seeding cuRAND host API.* [En línea] Consultado el 1 de junio de 2018. Disponible en: https://stackoverflow.com/questions/42072377/re-seeding-curand-host-api

GLOSARIO

CUDA: Compute Unified Device Architecture. Tecnología de cálculo paralelo desarrollada por NVIDIA y destinada a aprovechar el perfil de las tarjetas gráficas en tareas comunes 1

GPU: Graphics Processing Unit 1

CPU: Central Processing Unit. Responsible de cálculos, control y supervisión de otras partes en un ordenador1

C/C++, Fortran: Lenguajes de programación de propósito general 1

OpenACC: Estándar de programación de informática paralela 1

Host: En programación paralela, anfitrión, lado asociado a la CPU 1

Device: Al contrario que host, dispositivo, lado asociado a la GPU 1

Kernel: Función que se ejecuta en el dispositivo, en un determinado número de bloques e hilos 1

Thread: Hilo de ejecución en CUDA 2

Block: Bloque formado por hilos en CUDA 2

Grid: Conjunto que abarca varios bloques CUDA 2

SM: Streaming Multiprocessor. Cada uno de los multiprocesadores que componen una GPU 5

SIMT: Single-Instruction, Multiple-Thread. Arquitectura empleada en los multiprocesadores de una GPU 6

Warp: Conjunto de hilos que son procesados a la vez en un multiprocesador de una GPU con una sola instrucción 6

Host API: Hace referencia al conjunto de funciones que son ejecutadas en el lado del host 10

Device API: Hace referencia al conjunto de funciones que son ejecutadas en el lado del device 11