UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Tesis Fin de Master en Computación Paralela y Distribuida

Algoritmos Paralelos para la resolución de problemas de mínimos cuadrados basados en transformaciones ortogonales sobre GPUs y multiprocesadores

Presentado por Carla Ramiro Sánchez
Dirigido por Dr.Antonio M. Vidal Maciá

Diciembre 2010

Agradecimientos

Esta tesis de master ha sido financiada por los siguientes proyectos: PROMETEO/2009/013 (Generalitat Valenciana), TEC2009-13741 (Ministerio de Ciencia e Innovación, España), TIN2008-06570-C04-02 y PAID-05-09

Resumen

La resolución de sistemas de ecuaciones lineales sobredeterminados es un problema que se presenta con frecuencia en la computación científica. Algunos ejemplos pueden encontrarse en campos como el procesado de señal, resolución de problemas en electromagnetismo, simulación de dinámica molecular, econometría etc. La modelización de estos problemas da lugar a sistemas de ecuaciones lineales o problemas lineales de mínimos cuadrados con matrices densas, a veces enormes. Uno de los métodos que se utiliza habitualmente para resolver sistemas de ecuaciones lineales sobredeterminados es el de mínimos cuadrados.
Los procedimientos más fiables para resolver este problema conllevan la reducción de la matriz a alguna forma canónica mediante transformaciones ortogonales, como por ejemplo: la descomposición de Cholesky, descomposición en valores singulares o descomposición QR, siendo esta última la más comúnmente utilizada.

En la actualidad, las plataformas multicore, entre ellas las GPUs, lideran el mercado de los computadores, El rápido avance, tanto en la programabilidad de los procesadores gráficos como en su flexibilidad, ha permitido utilizarlos para resolver un amplio rango de complejos problemas con altas necesidades computacionales. Es lo que se conoce como GPGPU (General-Purpose Computing on the GPU).

En la presente tesis, se han implementado distintos algoritmos para la resolución de problemas de mínimos cuadrados: mínimos cuadrados ordinarios, generalizados, ponderados, Modelos de Ecuaciones Simultáneas y mínimos cuadrados sobre conjuntos discretos con aplicaciones en sistemas MIMO. Para ello se han utilizando distintos entornos como UPC, OMP, CUDA y librerías como LAPACK y CULA. Nuestros algoritmos están basados en la descomposición QR calculada mediante rotaciones de Givens, aunque también se han utilizado librerías como LAPACK, CULA o MAGMA que utilizan transformaciones de Householder para obtener esta descomposición.

Abstract

Solving overdetermined systems of linear equations is a problem that frequently occurs in scientific computing. Examples of this can be found in fields like signal processing, electromagnetism, molecular dynamics simulation, econometrics etc.
The modeling of these problems leads to linear systems and least squares problems, with dense matrices, sometimes huge. One of the methods commonly used to solve overdetermined linear systems is the least squares method.
The most reliable procedures for solving this problem involve the transformation of the matrix into some canonical form by using orthogonal transformations, for example, Cholesky decomposition, Singular Value Decomposition, QR decomposition, the latter being the most commonly used.

Nowadays, multicore platforms, including GPUs, dominate the computer market. The rapid advance in both the programmability of graphics processors as its flexibility has allowed use them to solve a wide range of complex problems with high needs computer, giving rise to the GPGPU (General-Purpose Computing on the GPU) concept.

In this work, we have implemented algorithms for solving least squares problems: lineal least squares problem, generalized least squares problem, weighted least squares problem, simultaneous equations models and least squares problem on discrete sets with applications in MIMO systems. We have used different settings such as UPC, OpenMP, CUDA and libraries such as LAPACK and CULA. Our algorithms are based on QR decomposition obtained by means of Givens rotations, but we have also used libraries such as LAPACK, CULA or MAGMA where QR decomposition is computed by means of Householder transformations.

Índice General

Capítulo 1
Introducción

En este trabajo se desarrollan, estudian y comparan algoritmos basados en la descomposición QR mediante rotaciones de Givens, para la resolución de distintos problemas de mínimos cuadrados, como son mínimos cuadrados generalizados, ponderados, en dos etapas y problemas de decodificación de señales en sistemas MIMO.
En este primer capítulo se muestra la motivación del trabajo desarrollado, se plantean los objetivos que se pretenden cubrir, así como la estructura del resto de capítulos.

1.1 Planteamiento

La resolución de sistemas de ecuaciones lineales sobredeterminados es un problema que se presenta con frecuencia en la computación científica. Algunos de estos ejemplos pueden encontrarse en campos como el procesado de señal, resolución de problemas de electromagnetismo, simulación de dinámica molecular, econometría etc. La modelización de estos problemas da lugar a sistemas de ecuaciones lineales o problemas lineales de mínimos cuadrados con matrices densas a veces enormes.
Uno de los métodos que se utiliza habitualmente para resolver sistemas de ecuaciones lineales sobredeterminados es el de mínimos cuadrados. Los procedimientos más fiables para resolver este problema conllevan la reducción de la matriz a alguna forma canónica mediante transformaciones ortogonales, como por ejemplo: la descomposición LU, descomposición en valores singulares, descomposición QR [1], siendo esta última la más comúnmente utilizada.

El reciente desarrollo de arquitecturas paralelas, principalmente los procesadores multinúcleo, ha propiciado la aparición de nuevos algoritmos de resolución que permiten explotar las características particulares de estas plataformas. Uno de los mayores exponentes de este tipo de arquitecturas son las GPUs (Graphical Processing Unit) que gracias a su capacidad para procesar datos en paralelo de forma masiva han experimentado un gran éxito estos últimos años, convirtiéndose hoy en día en plataformas de bajo coste para la computación de propósito general (GPGPU, General-Purpose Computacion on Graphics Hardware[2]).

Podemos encontrar algoritmos paralelos para la factorización QR en la librería LAPACK[3] que pueden ser utilizados para resolver el problema de mínimos cuadrados, en arquitecturas de tipo memoria compartida o en ScaLAPACK[14] distribuida. También los hay que utilizan aceleradores gráficos para su resolución[4,5] estos algoritmos son híbridos ya que realizan parte del cálculo en la tarjeta gráfica.

Nos proponemos abordar en profundidad problemas de mínimos cuadrados y su uso en aplicaciones como decodificación en sistemas mimo, problemas de mínimos cuadrados generalizados y en modelos de ecuaciones simultáneas, sobre GPUs y multicores.

1.2 Objetivos

En este trabajo se tiene como objetivo diseñar, implementar y evaluar algoritmos paralelos para resolver de forma eficiente el problema de mínimos cuadrados con transformaciones ortogonales, sobre arquitecturas actuales como son los multiprocesadores y unidades de procesamiento gráfico (GPGPUs).

Este objetivo general puede refinarse en los siguientes objetivos específicos:

- Estudio del problema matemático de mínimos cuadrados y su resolución mediante transformaciones ortogonales en concreto mediante la factorización QR.
- Paralelizar los algoritmos con el uso de herramientas software UPC, CUDA y CULA.
- Aplicar las técnicas basadas en la descomposición QR para resolver el problema de decodificación en sistemas MIMO basado en el ordenamiento por gradiente.
- Resolver el problema de mínimos cuadrados generalizados y ponderados de manera eficiente reduciendo el tiempo de ejecución con el uso de aceleradores gráficos.
- Resolver modelos de ecuaciones simultáneas con el uso de la factorización QR.
- Analizar y evaluar las prestaciones y escalabilidad de los algoritmos paralelos implementados.

1.3 Estado del arte

Durante muchos años los fabricantes de hardware basaban las mejoras de los computadores en el aumento de los ciclos de reloj del procesador. Este modelo de mejora del hardware se estancó a mediados del 2003 debido a los altos consumos de energía que producían las altas frecuencias de reloj, esta disipación de energía limitaba así el nivel de actividad que podía realizarse en cada ciclo de reloj en una CPU.
Desde ese momento los fabricantes de microprocesadores optaron por modelos de diseños multi-core y many-core, en el que existen varias unidades de procesos en un mismo chip, de esta forma se aumenta la capacidad de proceso sin aumentar el consumo de energía.

En la actualidad las plataformas multicore lideran el mercado de los computadores. Gracias a la industria de los videojuegos, las tarjetas gráficas han evolucionado de tal manera que se han convertido en procesadores con una gran capacidad de cómputo.
Por otra parte, la programación ha sido, hasta hace bien poco, una tarea de expertos puesto que sólo se disponía de APIs gráficas, como OpenGL y DirectX. En este sentido, los avances realizados en el modelo de programación y las herramientas de programación de los procesadores gráficos han resultado determinantes para su éxito. Recientemente, tanto AMD como NVIDIA han puesto a disposición de los desarrolladores de GPGPU nuevos modelos de programación. Cabe destacar CUDA[7], de NVIDIA, que proporciona una sintaxis como la del lenguaje C.

Por todos estos motivos los aceleradores gráficos han evolucionado notablemente en los últimos años, superando en rendimiento a las CPUs (ver figura 1.1). De hecho las GPUs actuales [8] tienen un rendimiento máximo teórico de 1TFlop/s en simple precisión.

En la actualidad, los principales fabricantes de chips, como Intel, AMD, IBM y NVIDIA, hacen que sea evidente que los futuros diseños de los microprocesadores y los grandes sistemas HPC serán híbridos de naturaleza heterogénea, basándose en la integración de dos grandes tipos de componentes:

1. Multicores, donde el número de núcleos seguirán aumentando.
2. Hardware de propósito especial y aceleradores, especialmente GPUs.

Existe una gran cantidad de software de calidad que calcula las factorizaciones más típicas en álgebra lineal numérica. En lo que respecta a software desarrollado para la factorización de matrices, podemos encontrar algoritmos por bloques, secuenciales y paralelos en la librería LAPACK [3], por ejemplo factorización de Cholesky (xPOTRF), factorización LU con pivotamiento (xGETRF) o factorización QR (xGEQRF).

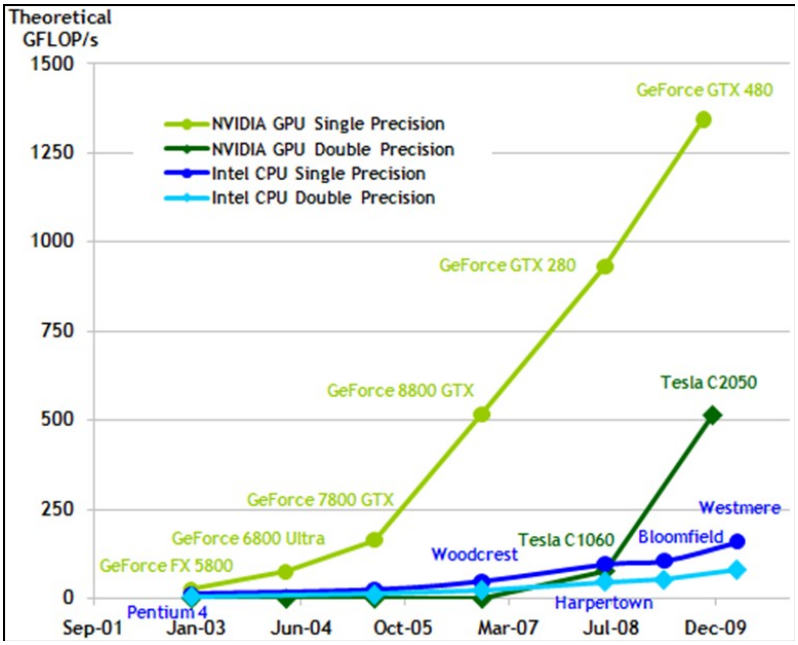


Figura 1.1 Comparativa rendimiento en GFLOPS entre CPUs y GPUs

En esta tesis se ha utilizado la factorización QR para resolver todos los tipos de problemas de resolución de mínimos cuadrados, por lo que nos vamos a centrar en lo referente a este tipo de descomposición. El algoritmo que implementa la rutina xGEQRF, es un algoritmo por bloques basado en la técnica WY que acumula transformaciones de Householder [6]. Las librerías MAGMA[4] (magma_xgeqrf) y CULA[5] (culaXgeqrf), utilizan el mismo algoritmo que LAPACK pero en este caso la computación es dividida entre la GPU y la CPU, de manera que sea posible solapar cálculo con algunas comunicaciones entre la CPU y la GPU. Sin embargo, no es frecuente encontrar algoritmos que calculen la descomposición QR utilizando transformaciones de Givens.

LAPACK tiene rutinas para resolver problemas de mínimos cuadrados generalizados (xGGGLM) y con restricciones de igualdad (xGGLSE), sin embargo, CULA únicamente resuelve problemas de mínimos cuadrados con restricciones de igualdad (culaXgglse).

1.4 Organización del documento

El resto del trabajo está organizado de la siguiente forma:

En el capítulo 2 se presentan las herramientas de computación paralela utilizadas durante el desarrollo de la investigación. Se describen las características de cada una de las herramientas hardware, y se describen un conjunto de librerías y entornos utilizados en la programación de las rutinas.

El capítulo 3 muestra los aspectos matemáticos y computacionales del problema de mínimos cuadrados y sus métodos de solución. Se presentan una implementación para la resolución de mínimos cuadrados mediante la factorización QR, haciendo uso de aceleradores gráficos. Se presentan también pruebas numéricas y comparaciones de las prestaciones del algoritmo desarrollado frente a la versión en secuencial.

En el capítulo 4 se plantea la formulación del problema de mínimos cuadrados dentro del campo de las comunicaciones inalámbricas MIMO. Se abordan varios métodos haciendo uso de rutinas implementadas en cuLAPACK, aceleradores gráficos y con lenguajes que trabajan con entornos de memoria compartida como OpenMP y UPC.

En el capítulo 5 se define la factorización QR generalizada y su aplicación en problemas de mínimos cuadrados generalizados y se analiza la posibilidad de resolver de forma similar modelos de ecuaciones simultáneas. Se presentan los métodos secuenciales que lo resuelven, y se presenta un algoritmo paralelo que utiliza GPUs; además se evalúan sus prestaciones con respecto a LAPACK.

En el capítulo 6 se dan las conclusiones de este trabajo y las perspectivas de investigación que se desprenden del mismo.

Capítulo 2
Arquitecturas y entornos actuales

En este capítulo se repasan las características de las arquitecturas de computadores actuales que han sido utilizadas para evaluar los algoritmos diseñados e implementados en esta tesis. También se describen las herramientas software utilizadas.

2.1 Multiprocesadores y Procesadores multinúcleo

Multiprocesadores

Los multiprocesadores con memoria compartida (o simplemente multiprocesadores) son plataformas compuestas por varios procesadores completos que disponen de una memoria común accesible mediante instrucciones hardware (en código máquina). Existen dos posibilidades en cuanto la forma en que se realiza el acceso a la memoria en estas plataformas:

- SMP (Symmetric Multiprocessor): El coste de acceso a cualquier posición de la memoria es uniforme.
- NUMA(Non-Uniform Memory Acces): La memoria está físicamente distribuida entre los procesadores, de modo que cada procesador dispone de una memoria local de acceso más rápido que el acceso a la memoria local de otro procesador.

Debido a la simplicidad del principio en el que están fundamentados (replicación completa de procesadores), los multiprocesadores fueron una de las primeras arquitecturas paralelas que se diseñaron. La facilidad de su programación, basada en el paradigma de variables compartidas (habitualmente, accesible mediante hebras de ejecución o herramientas de alto nivel para la paralelización de bucles) hace prever que seguirán siendo una plataforma paralela válida en los próximos años. La principal crítica que se hace a los multiprocesadores es su escasa escalabiliadad desde el punto de vista hardware. A medida que aumenta el número de procesadores, la memoria en el caso de SMP o la red de interconexión en las arquitecturas NUMA, se transforma en un cuello de botella. Sin embargo el número de procesadores que pueden soportar eficientemente estas plataformas es más que suficiente para muchas aplicaciones.

Procesadores multinúcleo

Los procesadores multinúcleo (multicore processors o chip multiprocessors) incluyen en un sólo chip varias unidades de proceso independientes, denominadas núcleos (cores). Cada núcleo es una unidad operacional completa, es decir, puede ser un procesador con técnicas sofisticadas de paralelismo y/o segmentación. Los problemas de disipación del calor y consumo de energía de la tecnología actual provocaron que a partir de 2005, los principales fabricantes de procesadores comenzaran a incorporar diseños multinúcleo para seguir transformando las mejoras en la escala de integración dictada por la ley de Moore, en un mayor rendimiento de sus productos.
En la actualidad, los procesadores multinúcleo incluyen un reducido número de núcleos (entre cuatro y ocho en la mayoría de los casos)[9], pero se espera que en un futuro próximo esta cantidad

se vea aumentada considerablemente. Los procesadores multinúcleo, si bien pueden programarse del mismo modo que los multiprocesadores, poseen características propias que los hacen diferentes:

- Las tendencias de diseño apuntan a procesadores multinúcleo heterogéneos, con núcleos de distinta capacidad y/o naturaleza integrados en el mismo sistema.
- Las previsiones para los procesadores multinúcleo apuntan a cientos de núcleos en un chip, frente a los multiprocesadores que, en sus configuraciones comerciales más frecuentes, no superan los 16 ó 32 procesadores.
- La organización habitual de los procesadores multinúcleo implica comunicaciones entre procesos mucho más eficiente (menor latencia, mayor ancho de banda y consumo más reducido) cuando los datos residen dentro del propio chip. Esta economía no es posible en los multiprocesadores y tampoco en los procesadores multinúcleo cuando los datos residen en niveles de memoria fuera del chip.

2.2 Aceleradores gráficos (Graphics processing unit: GPU)

Los procesadores gráficos (Graphics Processing Units o GPU) son arquitecturas que, en un principio, fueron diseñadas para el procesamiento de imágenes gráficas. En sus orígenes, estaban formados por un cauce segmentado cuyas etapas realizaban tareas fijas, explotando así el paralelismo a nivel de tareas y también el paralelismo a nivel de datos, ya que en cada etapa se trabajaba sobre varios datos a la vez. El rápido avance, tanto en la programabilidad de los procesadores gráficos como en su flexibibilidad, ha permitido utilizarlos para resolver un amplio rango de complejos problemas con altas necesidades computacionales. Es lo que se conoce como GPGPU (General-Purpose Computing on the GPU).

La figura 2.1 muestra la arquitectura típica de una GPU actual. Está compuesta por un número escalable de multiprocesadores paralelos (SMs). Estos multiprocesadores se agrupan de tres en tres (o de dos en dos en arquitecturas más antiguas) en lo que se llama Cluster de Procesado de Hilos (TPC). El número de SMs varía desde las arquitecturas más antiguas (1), hasta las más modernas y de mayor gama (30) [10].

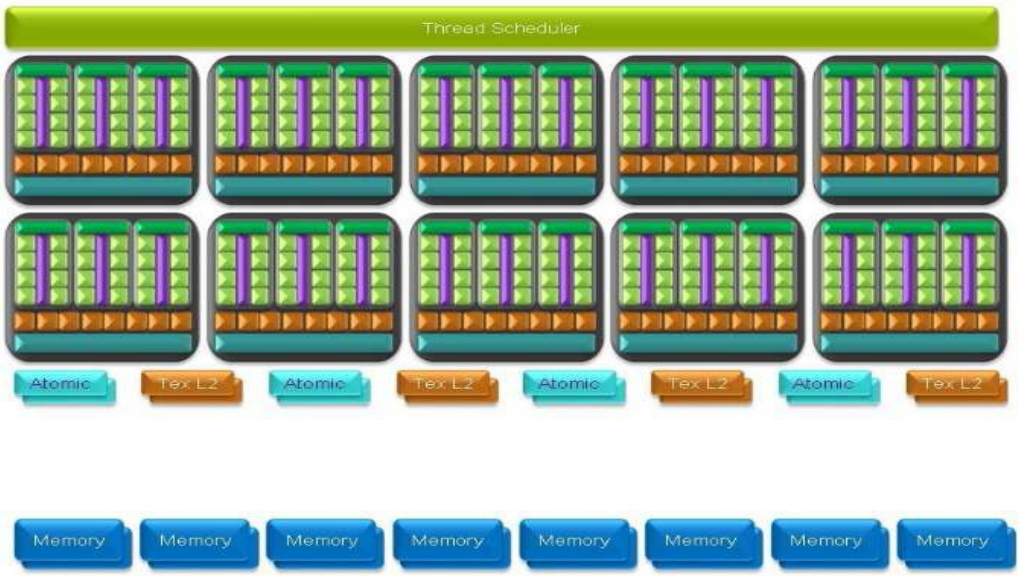


Figura 2.1: Arquitectura de una GPU moderna (GeForce GTX 280)

El diseño interno de cada SM es similar para todas las versiones, cada SM cuenta con 8 procesadores escalares (SPs), lo que hace un total de 240 núcleos en las tarjetas más modernas; 2 Unidades Especiales (SFUs), capaces de realizar operaciones en punto flotante. También cuenta con una unidad de multiplicación y suma (MAD) y una unidad adicional de multiplicación (MUL). Los ocho procesadores de un multiprocesador comparten la unidad de búsqueda y lanzamiento de instrucciones de forma que se ejecuta la misma instrucción al mismo tiempo en los ocho procesadores. La capacidad de computo de los procesadores varía en función del modelo, pero suelen funcionar a 1.3Ghz en la mayoría de modelos. Desde el punto de vista de la memoria, cada SM cuenta con tres módulos de memoria on-chip. La primera, una memoria compartida de lectura/escritura que ofrece un tiempo de acceso similar al de un registro. La segunda y tercera se corresponden con dos cachés de solo lectura: una de constantes y otra de texturas. Estos elementos se muestran en la figura 2.2.

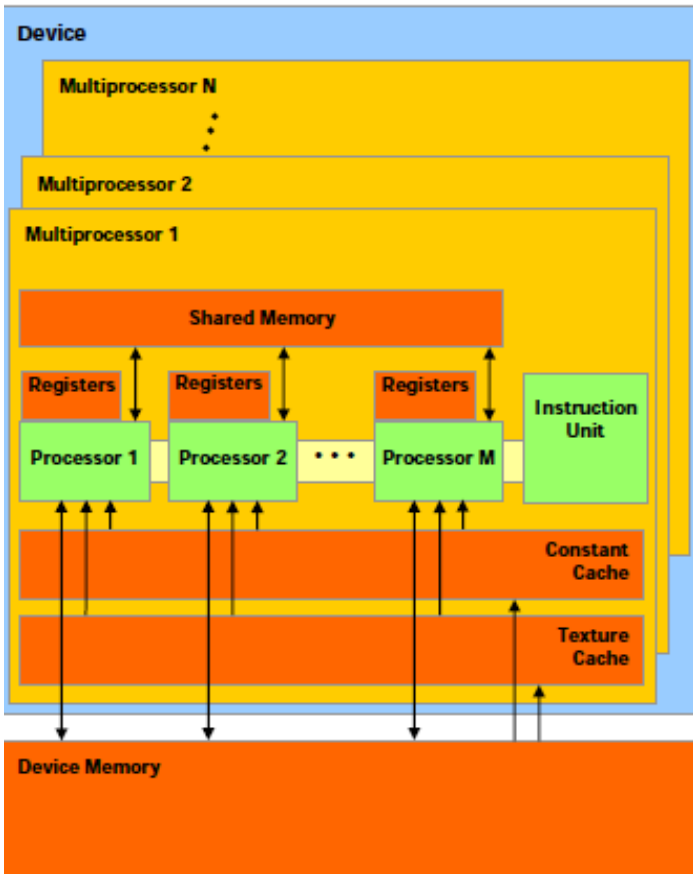


Figura 2.2: Arquitectura interna de un SM (Streaming Multiprocessor)

A nivel global, la tarjetas gráficas cuenta hasta con 6 GB de memoria dedicada, en los nuevos modelos Fermi [11]. Actualmente la comunicación de la GPU con la CPU se realiza a través de un bus PCI-Express.

La ejecución de los hilos en la GPU no se lleva a cabo de forma independiente. El planificador de hilos mostrado en la figura 2.3 planifica y distribuye bloques de hilos entre los SM. Los threads son asignados a los SMs en grupos llamados bloques (máximo 8 bloques por SM). El multiprocesador crea, gestiona y ejecuta hilos concurrentes en el hardware sin sobrecoste de planificación o cambios de contexto. Mapea cada hilo a un procesador, y cada hilo se ejecuta de forma independiente con su

propia dirección de instrucción y registros de estado. Este nuevo modelo de ejecución se llama SIMT (*Single Instruction Multiple Thread*) (ver figura 2.4) .
En primer lugar, el multiprocesador divide los bloques de hilos en grupos de 32 hilos llamados *warp*. A la hora de lanzar una nueva instrucción, la unidad de planificación, selecciona un warp disponible y lanza la misma instrucción para todos los hilos de ese *warp*.

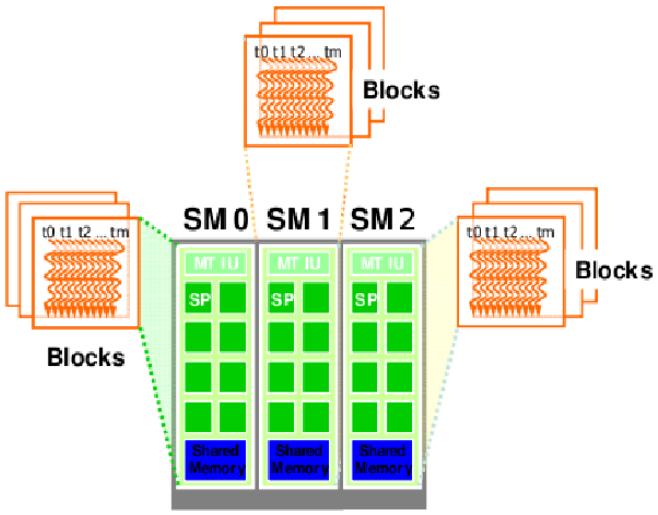


Figura 2.3: Asignación de bloques a los SMs (Streaming Multiprocessor) de un TPC

Ya que hay 8 SPs por SM, se tardan cuatro ciclos en ejecutar una instrucción de un *warp*. Las instrucciones de salto suponen un problema ya que hilos de un mismo *warp* pueden tomar caminos de ejecución distintos. En este caso, la ejecución se serializa, ejecutando primero los hilos de un camino y después los hilos del otro. Además existen instrucciones para sincronizar todos los hilos de un mismo bloque[1], haciendo que *warps* enteros detengan su ejecución hasta que todos los *warps* del bloque alcancen el mismo punto de ejecución.

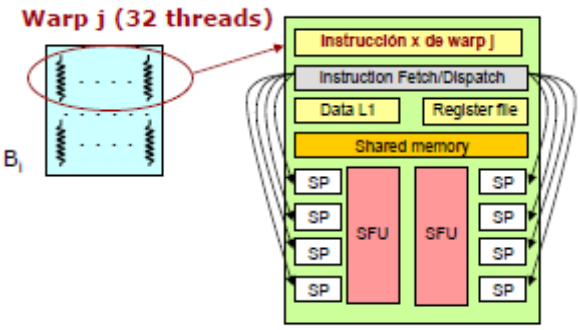


Figura 2.4: Single Instruction Multiple Thread

La arquitectura SIMT es similar a la arquitectura vectorial SIMD (*Single Instruction Multiple Data*) en el sentido en que una instrucción controla varios elementos de procesado.
Sin embargo, una diferencia clave es que la organización de los vectores SIMD exponen el ancho del vector al software, mientras que desde el punto de vista SIMT, las instrucciones especifican la ejecución y comportamiento de un único hilo. A diferencias de las máquinas SIMD, SIMT permite al programador describir paralelismo a nivel de hilo para hilos independientes, así como paralelismo a nivel de datos para hilos coordinados. El programador puede ignorar el comportamiento SIMT para el correcto funcionamiento de su código, sin embargo es un detalle muy importante para el

1 La sincronización es a nivel de hilos de un mismo bloque, ya que los bloques de hilos son independientes entre si y su sincronización no es posible.

rendimiento.
Las nuevas GPU de NVIDIA llamadas Fermi (ver figura 2.5) , esta implementada con 3 millones de transistores, cuenta con hasta 512 núcleos CUDA organizados en 16 SM de 32 núcleos cada uno. Un núcleo es capaz de ejecutar una instrucción de punto flotante o entera por ciclo de reloj de un hilo. A diferencia de arquitecturas anteriores es capaz de ejecutar varios kernels en paralelo añadiendo otro planificador, tiene un sistema de detección de errores, una nueva memoria caché para implementar algoritmos que no podían hacer un uso eficiente de la memoria compartida, mayor memoria compartida y global, etc. Fermi proporciona la capacidad de un superordenador por tan sólo una décima parte del coste y una vigésima parte del consumo de energía de los tradicionales servidores basados en CPUs.

Puede verse más información sobre el funcionamiento de una GPU en [12].

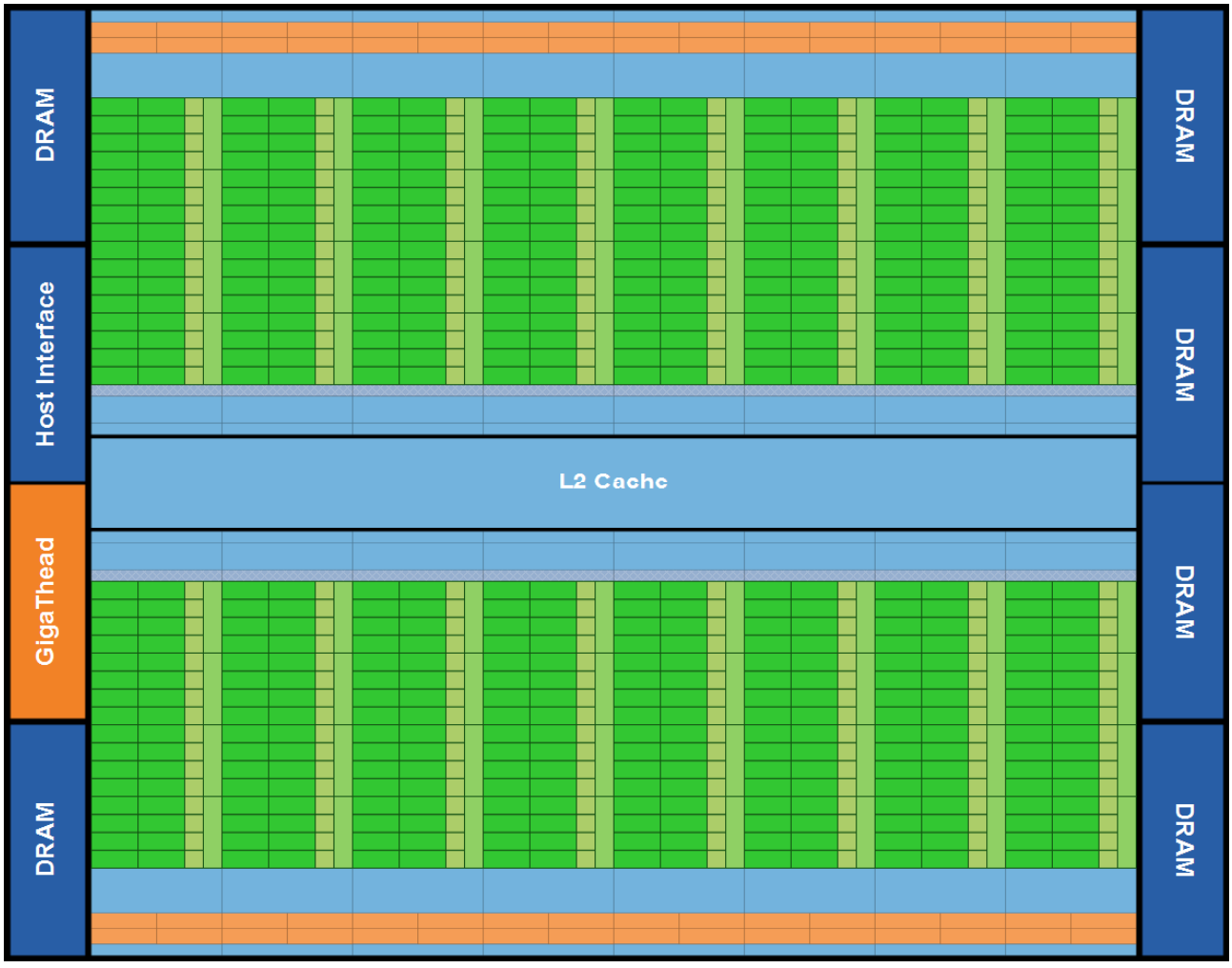


Figura 2.5: Arquitectura GPU Fermi . 16 SM rodeados de una memoria caché L2 común.

2.3 Herramientas software

Los modelos de programación utilizados para la implementación de los algoritmos han sido CUDA para las versiones que utilizan la GPU; también se han utilizado librerías matemáticas y UPC para el modelo de memoria compartida. Para evaluar los algoritmos implementados con UPC se han

implementado con el API openMP.

2.3.1 Modelo de programación CUDA

El modelo de programación de CUDA [7] supone que los hilos CUDA se ejecutan en una unidad física distinta, que actúa como coprocesador (*device*), al procesador (*host*) donde se ejecuta el programa (figura 2.6). CUDA C es una extensión del lenguaje de programación C, que permite al programador definir funciones C, llamadas *kernels*, que, al ser llamadas, se ejecutan en paralelo por *N hilos* diferentes. Los kernels se ejecutan de forma secuencial en el *device*[2].

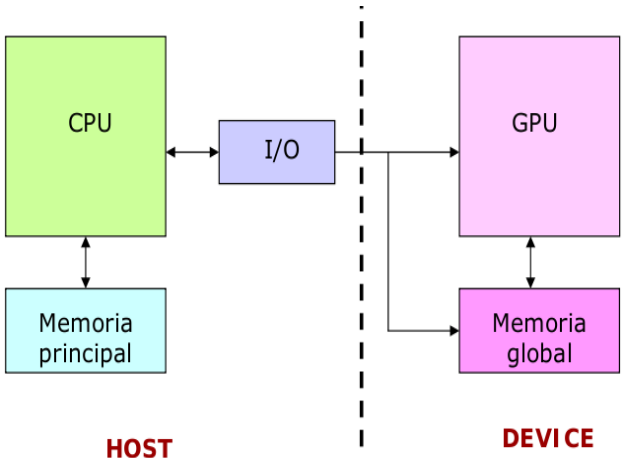


Figura 2.6: Arquitectura CPU-GPU

Como ejemplo, el siguiente código muestra cómo se define un kernel y cómo se llama desde un programa:

```
// Kernel d e f i n i t i o n
__global__ void vecAdd( f loat A, f loat B, f loat C)
{
        int i = threadIdx . x ;
        C[ i ] = A[ i ] + B[ i ] ;
}
int main ( )
{
        . . .
        // Kernel inv o c a t i on wi th N t h r e a d s
        vecAdd<<<1, N>>>(A, B, C) ;
}
```

Existe una jerarquía perfectamente definida sobre los hilos de CUDA. Los hilos se agrupan en vectores a los que se les llama *bloques*, estos vectores pueden ser de una, dos o tres dimensiones, de forma que definen bloques de hilos de una, dos o tres dimensiones.

Los hilos del mismo bloque pueden cooperar entre si, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, hilos de distintos bloques no pueden cooperar. Los bloques a su vez, se organizan en un *grid* de bloques. Este grid, de nuevo puede ser de una o dos dimensiones. Los valores entre <<< ... >>> que aparecen en código anterior se conocen como la configuración del kernel, y definen la dimensión del *grid* y el número de hilos de cada bloque. La figura 2.7 muestra como se organizan los hilos en un grid de 2x3 bloques y 3x4 hilos cada uno.

2 La nueva arquitectura Fermi permite ejecutar kernels en paralelo

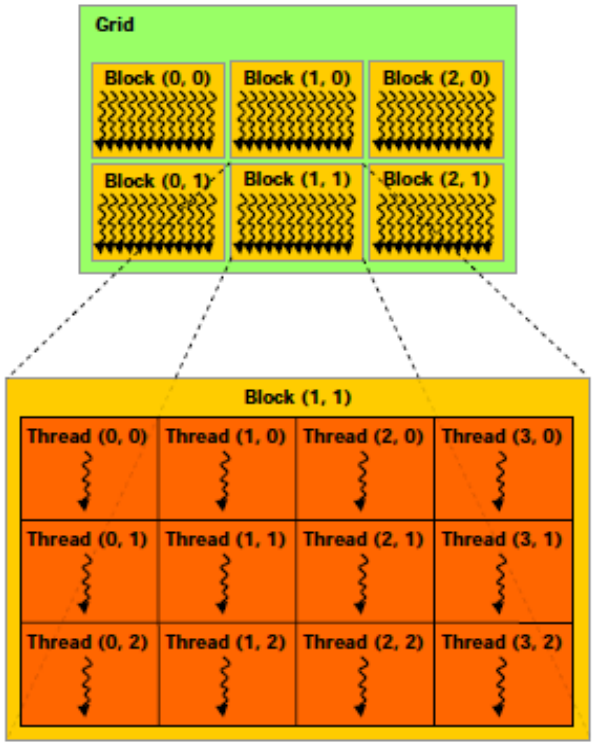


Figura 2.7 Grid de Bloques de hilos

Como puede verse, cada hilo queda perfectamente identificado por un *ID* de bloque y el *ID* del propio hilo dentro del bloque. Estos IDs suelen usarse como índices para definir que porciones de los datos procesa cada hilo. Esto puede verse en el código anterior.
Cada hilo tiene acceso a distintos tipos de memoria. En primer lugar una *memoria local* al propio hilo, esta memoria se aloja en la memoria principal de la GPU (off-chip). Además todos los hilos de un mismo bloque comparten una región de *memoria compartida* (on-chip) para comunicarse entre ellos, la memoria compartida tiene el mismo tiempo de vida que el bloque de hilos. Por último, todos los hilos tienen acceso a la misma memoria global (*device memory*).
Para más información se puede consultar el manual de CUDA [7].

2.3.2 Librerías matemáticas

A continuación se enumeran las librerías matemáticas secuenciales y paralelas más importantes. Algunas de ellas ya incorporan computación sobre GPU, como CULA y MAGMA y es en éstas en las que nos centraremos a lo largo del trabajo para realizar diversas comparativas.

BLAS (Basic Linear Algebra Subprograms). Colección de rutinas para realizar operaciones básicas a bloques sobre matrices densas y vectores [13].

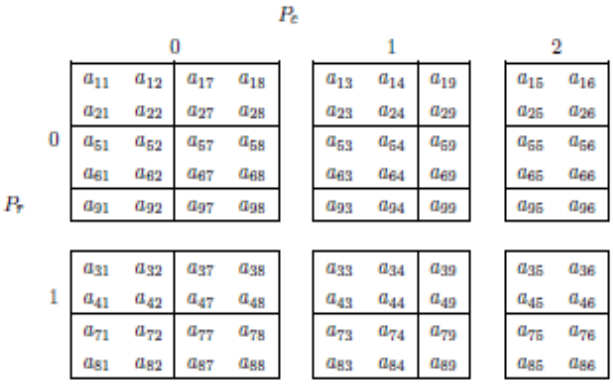
LAPACK (Linear Algebra PACKage). Biblioteca que contiene rutinas para resolver sistemas de ecuaciones lineales, problemas de mínimos cuadrados lineales, problemas de valores propios y problemas de valores singulares, sobre matrices densas [3]. Incluye una serie de rutinas para la factorización de matrices, además de rutinas que permiten resolver los sistemas obtenidos a partir de estas factorizaciones, entre otras. Las rutinas de LAPACK que realizan las factorizaciones de matrices densas son las siguientes:

_POTRF: Algoritmo por bloques para calcular la factorización de Cholesky.
_GETRF: Algoritmo por bloques para calcular la factorización LU con pivotamiento parcial de filas.
_GEQRF: Algoritmo por bloques para calcular la factorización QR.

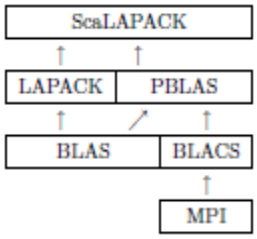
LAPACK ofrece dos rutinas para dos tipos de problemas de mínimos cuadrados generalizados, con restricciones de igualdad y generalizados. Las rutinas que resuelven estos problemas son:

_GGLSE: Resuelve el problema de mínimos cuadrados con restricciones de igualdad usando la factorización RQ generalizada (GRQ).
_GGGLM: Resuelve el problema de mínimos cuadrados generalizados usando la factorización QR generalizada.

ScaLAPACK (Scalable LAPACK). Librería que implementa un conjunto de rutinas LAPACK en multicomputadoras de memoria distribuida con mecanismos de comunicación de paso de mensajes. Asume que las matrices que maneja se descomponen en bloques cíclicos bidimensionales sobre una malla logica de Prc = PrxPc procesadores (Pr numero de procesadores renglón, Pc numero de procesadores columna). Por ejemplo, si la malla de procesadores consta de 2 renglones y 3 columnas (Pr = 2, Pc = 3, Prc = 6), se espera que una matriz de tamaño 9x9 y un vector de tamaño 9, que se descomponen en bloques de tamaño 2x2, queden distribuidos en la malla de procesadores como sigue [14]:



La jerarquía de software de las distintas librerías LAPACK/ScaLAPACK es la siguiente:



MKL(Math Kernel Library). Es la biblioteca que contiene funciones matemáticas optimizadas para aplicaciones científicas e ingenieriles optimizadas para procesadores Intel[15]. Incluye: BLAS, LAPACK, PARDISO, biblioteca matemática vectorial y biblioteca estadística vectorial.

CUBLAS. Es una implementación de los núcleos computacionales BLAS acelerados con el uso de GPU.[16]

CULA. Es una implementación de la librería LAPACK con el uso de GPU[5]. Tiene funciones para resolver sistemas lineales, factorizaciones ortogonales, funciones para calcular valores propios etc.

MAGMA. Es otra implementación de la librería LAPACK para ser utilizada sobre la GPU[4]. La nueva versión de la librería MAGMA (versión 0.2) incluye factorizaciones de matrices one-sided y solvers basados en esto. Las factorizaciones pueden ser en las 4 precisiones, simple, doble,compleo simple y complejo doble. Para cada función hay dos interfaces del estilo de LAPACK. La primera de ellas, es la interfaz de la CPU, coge la entrada y produce el resultado en la memoria de la CPU. La segunda de ellas, es la interfaz de la GPU, coge la entrada y produce el resultado en la memoria de la GPU. El nombre del algoritmo es el mismo que el utilizado en LAPACK, añadiendo el prefijo magma_ y en el caso de la interfaz de la GPU con el sufijo _gpu.

2.3.3 Unified Parallel C

Unified Parallel C (UPC) [17] es un lenguaje que sigue el modelo de Espacio de Direccionamiento Global Particionado (PGAS)[18] con un alto rendimiento y portabilidad en un amplio rango de arquitecturas, tanto de memoria compartida como distribuida.
Al igual que ocurre en memoria compartida, en PGAS todos los hilos comparten un mismo espacio de direccionamiento global. No obstante este espacio está dividido entre los hilos, como ocurre en el modelo de memoria distribuida. Así, el programador puede explotar la localidad espacial de los datos para aumentar el rendimiento de los programas y beneficiarse de la facilidad de programación de los modelos de memoria compartida.
Los lenguajes PGAS constituyen una solución de compromiso entre facilidad de uso y buen rendimiento gracias a la explotación de la localidad de los datos. De ahí que hayan sido varios los lenguajes PGAS desarrollados hasta la fecha, como Titanium, Co-Array Fortran y UPC.

Unified Parallel C es una extensión paralela del estándar de C. El lenguaje adopta un modelo de programación SIMD, pues todos los hilos ejecutan el mismo programa pero cada uno de ellos sobre un conjunto de datos locales. Como ocurre en otros modelos cada hilo tiene un identificador único contenido en la variable *MYTHREAD*, y se puede saber con cuanto hilos se está ejecutando el programa consultando la variable *THREADS*. Cada hilo tiene su parte privada de datos, pero además UPC dispone de un espacio compartido de direccionamiento para facilitar el paso de mensajes entre los hilos. El programador puede definir datos en este espacio compartido con el tipo de datos shared. Por lo tanto un dato en la parte privada de un determinado hilo sólo puede ser leído y escrito por sí mismo mientras que la memoria compartida puede ser accedida por todos los hilos para leer o escribir.

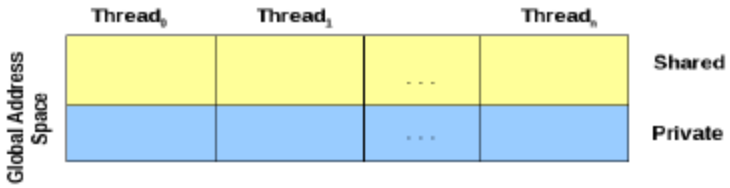


Figura 2.8: Esquema de la división del Espacio de Direccionamiento Global en UPC

Como puede verse en la figura 2.8 la memoria compartida está a su vez dividida entre todos los hilos por lo tanto aparece un nuevo concepto, el de memoria local compartida (local shared space). Los datos que estén en la parte de memoria local compartida de un determinado hilo se dice que tienen *afinidad* con ese hilo. Por lo tanto los compiladores utilizarán esta información para explotar la localidad de los datos y reducir los costes de comunicación.

2.3.4 OpenMP

La librería OpenMP[30] es un API que permite añadir concurrencia a los procesoss mediante paralelismo de memoria compartida. Con OpenMP es posible desarrollar programas a un nivel superior que haciéndolo creando hilos y controlando la concurrencia de éstos. Con las directivas de OpenMP se hacen transparentes las operaciones de concurrencia de hilos así como la creación, inicialización, destrucción, etc, de hilos.
OpenMP consta de tres componentes API principales: directivas del compilador, rutinas de librería de tiempo de ejecución y variables de entorno. Es una librería portable, pues está especificada para los lenguajes C/C++ y Fortran y se ha implementado en múltiples plataformas incluyendo Unix y Windows.

2.4 Herramientas hardware

A continuación se describen brevemente las computadoras utilizadas en los experimentos. Se trata de dos máquinas, ambas con tarjetas gráficas de NVIDIA.

En este trabajo se ha utilizado el multiprocesador Golub perteneciente al grupo INCO2 del departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia. Golub es un multicore compuesto por 4 núcleos. El procesador es un Intel Xeon E5430 a 2.66 GHz y con 6MB de memoria caché. La tarjeta gráfica que tiene instalada es una NVIDIA Quadro FX 5800, con 30 multiprocesadores con 8 procesadores cada uno, lo que hace un total de 240 cores. Estas unidades trabajan a 1.35 Ghz. Tiene 4GB de memoria DRAM. La tarjeta soporta ejecución y copia en paralelo (concurrent copy and execution) y el driver de CUDA que tiene instalado es el 3.1. Cuenta con multitud de librerías como MKL, CUBLAS, CULA, MAGMA, y soporte para trabajar con OMP y UPC.

Otro computador utilizado es Locomotora también del grupo INCO2 y ubicado en el ITEAM (Universidad Politécnica de Valencia). Es un multicore compuesto por 4 núcleos con hyperthreading activado, lo que hace un total de 8 cores lógicos. El procesadores es un Intel Core i7 CPU 950 a 3.07GHz y con 8MB de memoria caché. El computador tiene instaladas dos tarjetas NVIDIA; una de ellas es una Tesla con las mismas características que una Quadro FX 5800, y la otra es una GeForce GTX 285 que a diferencia de las otras dos tarjetas tiene una mamoria de 2 GB. También tiene instaladas las librerías CUBLAS, CULA, y soporte para trabajar con OMP y UPC.

2.5 Conclusiones

En este capítulo hemos descrito las herramientas de las que disponemos para la implementación de los algoritmos paralelos. A continuación se va a describir brevemente como se van a utilizar todos

los recursos de los que se dispone para paralelizar los algoritmos.
Los aceleradores gráficos se van a utilizar para programar rutinas eficientes para la factorización QR, y QR generalizada basadas en las rotaciones de Givens. Con estas rutinas y realizando llamadas a librerías ya optimizadas como CUBLAS se van a desarrollar rutinas capaces de resolver varios problemas de mínimos cuadrados, como son : mínimos cuadrados ordinarios, generalizados, ponderados y en dos etapas.

Para resolver el problema en sistemas MIMO que veremos en el capítulo 4, se van a utilizar varios entornos, por un lado se utilizarán las GPUs y la librería CULA, y también, se implementarán en memoría compartida utilizando openMP y UPC.

Estos algoritmos y rutinas se evaluarán en base a algoritmos secuenciales y rutinas de la librería LAPACK y CULA.

Capítulo 3
Problemas de mínimos cuadrados y transformaciones ortogonales

3.1 Descripción matemática

Uno de los métodos que se utilizan habitualmente para resolver sistemas de ecuaciones lineales sobredeterminados es el de mínimos cuadrados. Dado el sistema $Ax = b$, con $A \in \mathbb{R}^{mxn} con\, m \geq n\, y\, b \in \mathbb{R}^m$, su resolución mediante este método consiste en encontrar el vector $x \in \mathbb{R}^n$ que minimiza $\|Ax-b\|_2$. Los procedimientos más fiables para resolver este problema conllevan la reducción de la matriz A a alguna forma canónica mediante transformaciones ortogonales. Entre estos procedimientos el más comúnmente utilizado es la factorización QR[1], que descompone la matriz $A \in \mathbb{R}^{mxn}$ en el producto A = QR, donde $Q \in \mathbb{R}^{mxm}$ es ortogonal y $R \in \mathbb{R}^{mxn}$ es triangular superior. Una vez factorizada la matriz en el producto A=QR, la resolución del problema de mínimos cuadrados se lleva a cabo calculando, en primer lugar, $y = Q^T b$ y resolviendo, después, el sistema triangular superior $R_1 x = y_1$ con

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Existen varios métodos para calcular la factorización QR. Así, los hay que se basan en transformaciones de Householder, otros se basan en rotaciones de Givens, y por ultimo están los métodos que realizan una ortogonalización vía Gram-Schmidt o via Gram-Schmidt modificado.

A continuación describimos en que se basan los métodos de Householder y de Givens. Este último método será que el que utilizaremos para implementar los algoritmos que se presentan en esta tesis.

3.2 Algoritmos básicos

En este apartado se presentan los algoritmos para la factorización QR con el uso de los dos métodos vistos en el apartado anterior. Además se describe el algoritmo por bloques que se utiliza en Lapack y también en las librerías MAGMA y CULA basada en la técnica WY de acumulación de transformaciones de Householder.

3.2.1 Transformaciones de Householder

Una transformación de Householder o reflexión de Householder[1] es una transformación que refleja el espacio con respecto a un hiperplano determinado. Esta propiedad se puede utilizar para realizar la transformación QR de una matriz si tenemos en cuenta que es posible diseñar la matriz de Householder de manera que un vector elegido quede con una única componente no nula tras ser transformado (es decir, premultiplicando por la matriz de Householder). Gráficamente, esto significa que es posible reflejar el vector elegido respecto de un hiperplano de forma que el reflejo

quede sobre uno de los ejes de la base cartesiana.
La elección del hiperplano de reflexión y la construcción de la matiz de Householder asociada, es de la forma siguiente: Sea $x \in R^n$ un vector columna arbitrario tal que $||x||_2 = |\alpha|$, donde α es un escalar. Entonces, siendo e_1 el vector $(1, 0, ..., 0)^T$, se define $u = x \pm \alpha e_1$, $v = \frac{u}{||u||_2}$ y $Q = I - 2vv^T$, donde v es un vector unitario perpendicular al hiperplano de reflexión elegido, Q es una matriz de Householder asociada dicho hiperplano y $Qx = (\alpha, 0, \cdots, 0)^T$. En el Algoritmo 3.2 puede verse una descripción del proceso en detalle.

Algoritmo 3.1 (house) Dado $x \in \mathbb{R}^n$, esta función calcula $v \in \mathbb{R}^n$ con v(1) = 1 y $\beta \in \mathbb{R}^n$ de manera que $P = I_n - \beta vv^T$ es ortogonal y $Px = \|x\|_2 e_1$.

Entrada: $x \in \mathbb{R}^n$
Salida: $v \in \mathbb{R}^n$ y $\beta \in \mathbb{R}^n$
1: $n = length(x)$
2: $\sigma = x(2:n)^T x(2:n)$
3: $v = \begin{bmatrix} 1 \\ x(2:n) \end{bmatrix}$
4: **si** σ = 0 **entonces**
5: $\beta = 0$
6: **sino**
7: $\mu = \sqrt{x(1)^2 + \sigma}$
8: **si** x(1) <= 0 **entonces**
9: $v(1) = x(1) - \mu$
10 **sino**
11: $v(1) = -\sigma / (x(1) + \mu)$
12 **fin si**
13: $\beta = 2v(1)2/(\sigma + v(1)2)$
14: $v = v/v(1)$
15: **fin si**

Algoritmo 3.2 (Householder QR) Dado $A \in \mathbb{R}^{mxn}$ con $m \geq n$ el siguiente algoritmo encuentra las matrices de Householder $H_1,...,H_n$ tales que $Q = H_1 \cdots H_n$ se verifica que $Q^TA = R$ es triangular superior. La parte triangular superior de A es sobrescrita por la para triangular superior de R y las componentes *j+1:m* de *j-ésimo* vector de Householder son almacenadas en *A(j+1:m,j), j<m*.

1: **Para** j=1:n **hacer**
2: $[v, \beta] = house(A(j:m, j))$
3: $A(j:m, j:n) = (I_{m-j+1} - \beta vvT) A(j:m, j:n)$
4: **si** j<m **entonces**
5: $A(j+1:m, j) = v(2:m-j+1)$
6: **fin si**
7: fin para

Para aclarar como se sobrescribe la matriz A consideramos el siguiente ejemplo:

$$A=\begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ v_2^{(1)} & r_{22} & r_{23} & r_{24} & r_{25} \\ v_3^{(1)} & v_3^{(2)} & r_{33} & r_{34} & r_{35} \\ v_4^{(1)} & v_4^{(2)} & v_4^{(3)} & r_{44} & r_{45} \\ v_5^{(1)} & v_5^{(2)} & v_5^{(3)} & v_5^{(4)} & r_{55} \\ v_6^{(1)} & v_6^{(2)} & v_6^{(3)} & v_6^{(4)} & v_6^{(5)} \end{bmatrix}$$

Este algoritmo requiere $2n^2(m-n/3)$ flops. Si la matriz $Q = H_1 \cdots H_n$ es requerida, puede calcularse con el método de acumulación , cuyo coste es $4(m^2n-mn^2+n^3/3)$ flops.

3.2.2 Rotaciones de Givens

Las descomposiciones QR pueden calcularse utilizando una serie de rotaciones de Givens. Cada rotación anula (hace cero) un elemento en la subdiagonal de la matriz, formando de este modo la matriz R. La concatenación de todas las rotaciones de Givens realizadas, forma la matriz ortogonal Q.
El procedimiento de rotación de Givens es útil en situaciones donde sólo pocos elementos fuera de la diagonal necesitan ser anulados.

Las rotaciones de Givens son modificaciones de la matriz identidad de rango 2 de la forma.

$$G(i,k,\theta)=\begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix}$$
$$\begin{matrix} & & i & & k & & \end{matrix}$$

donde $c=cos(\theta)$ y $s=sin(\theta)$ para algun θ.
El efecto de una rotación de Givens sobre un vector *x es el siguiente*

$$G(i,k,\theta)\begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_i \\ \vdots \\ X_k \\ \vdots \\ X_n \end{pmatrix}=\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ cx_i-sx_k \\ \vdots \\ sx_i+cx_k \\ \vdots \\ x_n \end{pmatrix}$$

Eligiendo el ángulo θ apropiado, podemos hacer un cero en la componente k, por ejemplo, podemos hacer $y_k = 0$ tomando:

$$c=\frac{x_i}{\sqrt{x_i^2+x_k^2}} \quad s=\frac{-x_k}{\sqrt{x_i^2+x_k^2}}$$

Algoritmo 3.3 Dado dos escalares a , b, esta función calcula *c=cos(θ)* y *s=sin(θ)* de manera que $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix}=\begin{bmatrix} r \\ 0 \end{bmatrix}$

***Function**: [c,s] = **givens**(a,b)*

1: **si** b = 0 **entonces**
2: c=1;s=0
3: **sino**
4: **si** |b| > |a| **entonces**
5: r = -a/b; $s=1/\sqrt{1+r^2}$;c=sr
6: **sino**
7: r=-b/a; $c=1/\sqrt{1+r^2}$;s=cr
8: **fin si**
9: **fin si**

Algoritmo 3.4 (Givens QR) Dado $A\in\mathbb{R}^{mxn}\, con\, m\geqslant n$ el siguiente algoritmo sobrescribe A con $Q^TA=R$, donde R es triangular superior y Q es ortogonal .

1: **para** j=1:n **hacer**
2: **para** i=m:-1:j+1 **hacer**
3: [c,s] = givens(A(i-1,j),A(i,j))
4: A(i-1:i,j:n) = $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T$ A(i-1:i,j:n)
5: **fin para**
6: **fin para**

Este algoritmo requiere $3n^2(m-n/3)$ flops. Una rotación (c,s) puede se puede codificar en un único número ρ y ser almacenado en A(i,j) de la siguiente manera.

***if** c = 0*
ρ = 1
***elseif** |s| < |c|*
ρ = sign(c)s/2
else
ρ=2sign(s)/c
end

De manera que podemos reconstruir la rotación como:

***if** ρ=1*
c=0;s=1
***elseif** |p| < 1*
s=2 ρ; $c=\sqrt{1-s^2}$
else
c=2/p; $s=\sqrt{1-c^2}$

Este procedimiento permite almacenar y recuperar las rotaciones con una pérdida mínima de

información asociada a errores de redondeo y cancelaciones catastróficas.

3.2.3 Representación WY de los productos de las matrices de Householder

La representación WY [6] permite calcular la factorización QR de una matriz por bloques utilizando fundamentalmente operaciones de tipo BLAS 3. Si tenemos una matriz A dividida en bloques de la siguiente forma

$$A=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

y consideramos las $P_{1,}P_2,\ldots,P_b$ reflexiones de Householder que triangularizan $\begin{pmatrix} A_{11} \\ A_{12} \end{pmatrix}$,éstas pueden expresarse como

$$P_{1,}P_2,\ldots,P_b=I-\begin{bmatrix} V_{11} \\ V_{22} \end{bmatrix}\left[T_{11}\right]\left[V_{11}^T V_{21}^T\right], \textit{con } T_{11} \textit{ triangular superior.}$$

Por tanto

$$P_{1,}P_2,\ldots,P_b=\left[I-\begin{bmatrix} V_{11} \\ V_{22} \end{bmatrix}\left[T_{11}\right]\left[V_{11}^T V_{21}^T\right]\right]^T\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$P_{1,}P_2,\ldots,P_b=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}-\begin{bmatrix} V_{11} \\ V_{22} \end{bmatrix}\left[T_{11}\right]\left[V_{11}^T A_{11}+V_{21}^T A_{21} \quad V_{11}^T A_{12}+V_{21}^T A_{22}\right]=\begin{bmatrix} R_{11} & \bar{A}_{12} \\ 0 & \bar{A}_{22} \end{bmatrix}$$

A continuación se sigue aplicando la misma técnica sobre la matriz $\bar{A}_{22.}$ para conseguir la triangularización completa de A. Esta técnica se utiliza, por ejemplo en las librerías LAPACK, MAGMA y CULA.

La subrutina de LAPACK DGEQR2 está basada en este método, produce *b* reflectores de Householder (V_{*1}) y una matriz triangular superior R_{11} de tamaño *bxb*, que es una porción de la *R* final. La matriz triangular T_{11} se calcula con la subrutina de LAPACK DLARFT. V_{11} es una matriz triangular inferior de tamaño *bxb*. Los vectores V_{*1} y R_{11} no necesitan almacenamiento extra para ser almacenadas ya que sobrescriben A_{*1}. Sin embargo si que es necesario almacenamiento extra para $T_{11.}$

La operación

$$\begin{pmatrix} R_{11} \\ \tilde{A}_{12} \end{pmatrix}=\left(I-\begin{pmatrix} V_{11} \\ V_{22} \end{pmatrix}\cdot(T_{11})\cdot(V_{11}^T V_{21}^T)\right)\begin{pmatrix} A_{11} \\ A_{22} \end{pmatrix}$$

se calcula con la subrutina del LAPACK DLARFB, y produce una porción R_{12} de la matriz *R* final de tamaño *bx(n-b)* y la matriz $\tilde{A}_{22}$.

El tamaño del bloque $b << m,n$ es inicializado por defecto a 32 en LAPACK y puede ser modificado a un valor más apropiado dependiendo de las características de la arquitectura del computador donde se ejecute.

MAGMA y CULA utilizan algoritmos híbridos donde la computación es dividida entre la GPU y la

CPU. En general el cálculo de V_{11} y T_{11} se hace en la CPU y la actualización del resto de la matriz en la GPU. Estas técnicas son usadas para solapar el trabajo y algunas comunicaciones de la CPU y la GPU. La figura 3.1 ilustra esta cuantificación de solapamiento de la QR en simple precisión en el algoritmo usado por MAGMA.

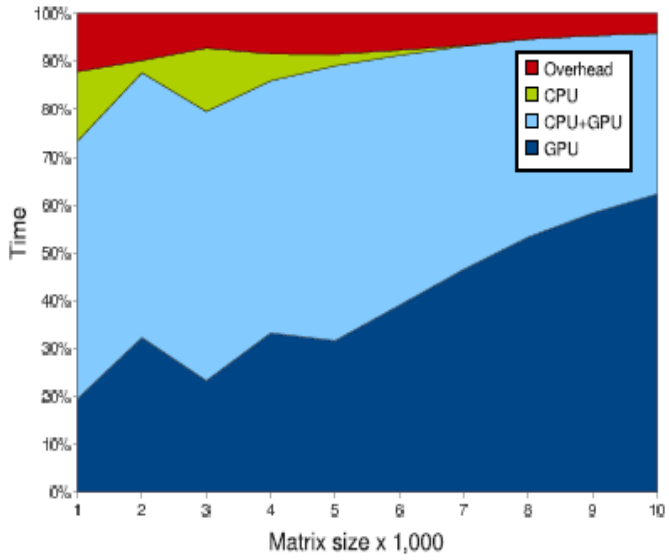


Figura 3.1: Tiempo para QR hibrida de MAGMA en precision simple aritmetica sobre una GPU GTX280 y un Intel Xeon 2.33GHz CPU.

En ambas interfaces la matriz a factorizar reside en la memoria de la GPU, y las transferencias entre la CPU-GPU están asociadas al cálculo de V_{11} y T_{11}. El resultado de la matriz es acumulado (en la CPU o GPU dependiendo de la interfaz utilizada) durante el cálculo. En la interfaz de CPU, la transferencia de la matriz original a la GPU se solapa con el primer cálculo de V_{11} y T_{11}.

3.3 Factorización QR en CUDA

Se ha implementado un algoritmo que calculan la descomposición QR de una matriz, que se basa en triangularizar la matriz por diagonales; la CPU realiza parte del cálculo ya que es la encargada de calcular los índices de la nueva diagonal y enviarlos a la memoria de la GPU. Al contrario que los algoritmos utilizados en las librerías antes mencionadas, el algoritmo que se presenta en esta tesis utiliza rotaciones de Givens.

El algoritmo se basa en realizar múltiples rotaciones en una misma fase, de manera que puedan realizarse múltiples ceros con la condición de que se apliquen en filas distintas. Para ello las rotaciones se realizan por diagonales, como puede verse en la figura 3.7 para una matriz de 8x4 puede calcularse la triangulación en 10 fases; por ejemplo en la fase 7 se pueden calcular 4 ceros en paralelo.

$\bar{x}$	x	x	x
1	$\bar{x}$	x	x
2	3	$\bar{x}$	x
3	4	5	$\bar{x}$
4	5	6	7
5	6	7	8
6	7	8	9
7	8	9	10

Figura 3.7 Ejemplo de rotaciones por diagonales

Estas rotaciones se aplican sobre el elemento diagonal de la columna donde se esté calculando el cero (denotado por $\bar{x}$ en la figura 3.7).
En el Algoritmo 3.7 se describen los pasos para calcular la QR con este método; $Nf = n+m-2$ es el número de fases necesarias para llevar a cabo la triangulación (10 en el ejemplo anterior). Los *indices* que se envían a la GPU son almacenados en un array de enteros dimensión 2**Nc*, donde *Nc* es el número de ceros a hacer en cada fase. En este array se almacena la fila y la columna donde se aplica la rotación, y se utilizará en el kernel *calcularRotaciones* para que cada hilo calcule el cero correspondiente a un elemento de una fila y columna determinada y en el kernel *aplicarRotaciones* para que los hilos que deben aplicar al resto de columnas sepan qué rotación deben aplicar.

Algoritmo 3.7 : (DQR) Algoritmo que calcula la Factorización QR por diagonales en la GPU

Entrada: $A \in R^{mxn}$

Salida: $Q \in R^{mxm}$ *y* $R \in R^{mxn}$

1: Para *i = 1:Nf* **hacer**
2: indices ← calcula índices de rotaciones
3: copiar indices a memoria constante de la GPU
4: calcularRotaciones(A) *(calcula las rotaciones de la diagonal i en la GPU dimensión del grid* $(1 x \lceil \frac{Nc}{bx} \rceil)$ *)*
5: aplicarRotaciones(A) *(aplica las rotaciones de la diagonal i en la GPU* $(Nc x \lceil \frac{n}{bx} \rceil)$ *)*
6: fin Para

Algoritmo 3.8 (calcularRotaciones) Calcula las rotaciones de una diagonal

```
1: index ← blockIdx.x*bx+threadIdx.x;
2: si index <= diag-1 entonces
3:    f ← indices[index*2]
4:    c ← indices[index*2+1]
5:    [cos,sen] ← getGivens(t1,t2)
6:    A(c,c) ← cos*A(c,c)-sen*A(f,c)
7:    A(f,c) ← getPacks(cos,sen)
8: fin si
```

Algoritmo 3.9 (aplicarRotaciones) Aplica las rotaciones de una diagonal

```
1:  index ← blockIdx.x*bx+threadIdx.x;
2:  f ← indices[blockIdx.y*2]
3:  c ← indices[blockIdx.y*2+1]
4:  si threadIdx.x = 0 entonces
5:        [cos,sen] ← unpack(A(f,c))  (* la rotación se almacena en memoria compartida *)
6:  fin si
7:   __syncthreads();
8:   si index < g.BX y index > c entonces
9:        A(c,c) ← cos*A(c,index) - sen*A(f,index)
10:       A(f,c) ← sen*A(f,index) - cos*A(f,index)
11:  fin si
```

A continuación se muestra cómo se realiza esta factorización con el método visto en el Algoritmo 3.7. Para ello vamos a coger como ejemplo el visto en la figura 3.7.
La figura 3.8 da una representación gráfica del Algoritmo 3.7 con una matriz de 8x4. Con un tamaño de bloque de hilos de 2, es decir por cada bloque en el grid se crean dos hilos que aplicarán el kernel. La dimensión del grid para el caso del kernel calculaRotaciones es lineal $\lceil \frac{Nc}{bx} \rceil$. Y el tamaño del grid en para el kernel aplicaRotaciones es bidimensional, de tamaño $Nc \, x \lceil \frac{n}{bx} \rceil$.

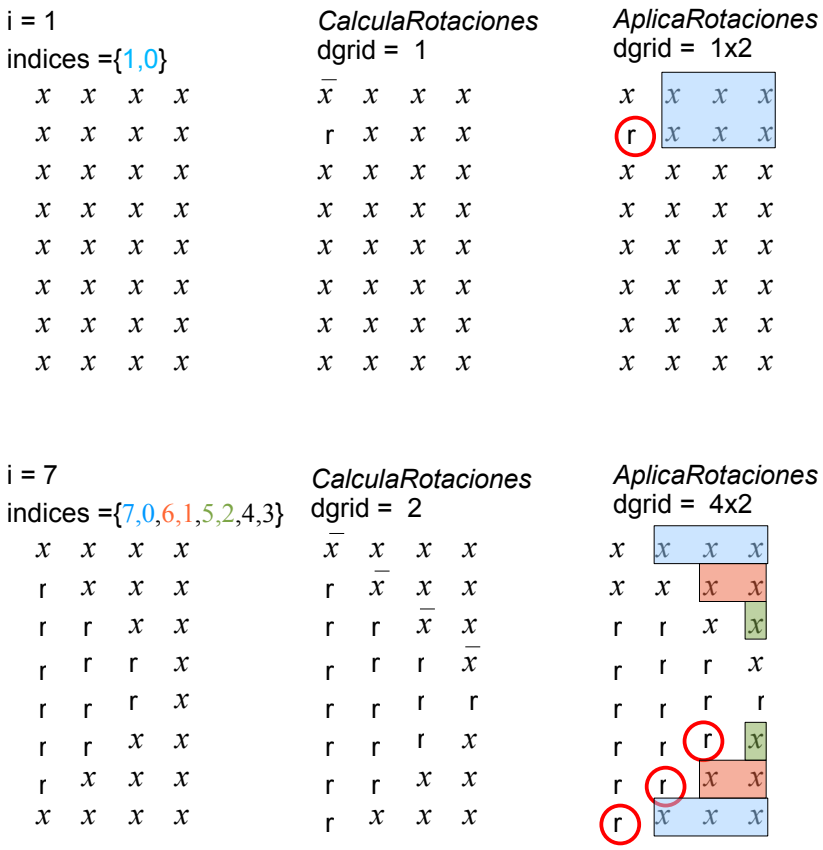


Figura 3.8 Representación gráfica del Algoritmo 3.3.2

Cuando i = 7, el kernel calculaRotaciones se lanza con un total de 2x2 hilos, cada uno de estos hilos calcula una rotación para una determinado elemento que extrae del array de *indices*, en cambio el kernel AplicaRotaciones se lanza con 4x(2x2) hilos, el hilo 0 de cada bloque es el encargado de acceder a la rotación previamente calculada y guardarla en una estructura de datos de tipo *__shared__*, de esta manera estamos asegurando que todos lo hilos que tengan que aplicar esa rotación accedan a ella a través de su memoria compartida y no a la global cuyo acceso conllevaría más coste.

3.5 Resultados

La tabla 3.1 muestra una comparación entre los tiempos de ejecución y Speed-Up de la factorización QR en secuencial vista en el Algoritmo 3.4 y la implementada por diagonales con el uso de CUDA. Las pruebas se han realizado con matrices aleatorias de tamaño MxN, donde M = 8192. El bloque de hilos seleccionado para calcular la factorización QR en la GPU es 128, ya que es el que mejores resultados obtenía.

N	Secuencial	Por Diagonales (Bloque 128)	Speed-Up
64	4,70E-02	5,43E-01	0,09
256	6,83E-01	8,40E-01	0,81
512	2,41E+00	1,48E+00	1,63
1024	8,67E+00	2,75E+00	3,15
2048	3,16E+01	5,95E+00	5,31
4096	1,10E+02	1,60E+01	6,88
8192	3,51E+02	4,45E+01	7,89

Tabla 3.1 Tiempos de ejecución (en segundos) y Speed-Up de los algoritmos que calculan la factorización QR en secuencial y por diagonales en Golub

Como puede observarse el speed-up aumenta a medida que vamos aumentando la dimensión de la matriz.

3.6 Resolución del problema lineal de mínimos cuadrados en CUDA

Una vez vistos el método desarrollado para calcular la factorización QR, la resolución del sistema de ecuaciones es trivial. El objetivo es minimizar la función $\|Ax-b\|_2$. Para ello se calcula la descomposición QR de A , de manera que Q^TAx - $Q^Tb = 0$, esto es equivalente a resolver el sistema de ecuaciones $R_1x = c_1$, con $R=\begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ $c=Q^Tb=\begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$.

Una técnica para evitar construir la matriz Q y así evitar el realizar el producto Q^Tb, es ir aplicando simultáneamente las rotaciones calculadas para triangularizar A al vector b. La manera en la que se consigue es construyendo una matriz de la forma [A | b] . Ahora, los hilos que actualizan la matriz tendrán que repartirse *n+1* columnas y aplicar en ellas todas las rotaciones. En el algoritmo 3.10 se muestra los cambios realizados a *DQR* (Algorimto 3.7) para realizar estas operaciones, como puede observarse la única diferencia es que ahora el *grid* de hilos con el que se llama a *aplicarRotaciones*,

debe tener una dimensión $(Nc\,x\lceil\frac{n+p}{bx}\rceil)$, en el caso que nos ocupa en este apartado $p = 1$, que es la dimensión del vector al que queremos aplicarle las rotaciones. En el capítulo 5 veremos como se utiliza este algoritmo para calcular la Factorización QR generalizada de una matriz.

Algoritmo 3.10 : (qr_span) Algoritmo que calcula la Factorización QR de una matriz Extendida

---

Entrada: $A \in R^{m\,x\,n+p}$ donde $A = [A_1 \mid A_2 \mid ... \mid A_k]$, n = *columnas de* A_1 , p = *columnas de* $A_2 + ... + A_k$

Salida: $Q^T[A_1 A_2, ..., A_k] = [R \;\; Q^T A_{2,...}, Q^T A_k]$

1: Para $i = 1:Nf$ **hacer**
2: indices ← calcula indices de rotaciones
3: copiar indices a memoria constante de la GPU
4: calcularRotaciones(A) *(calcula las rotaciones de la diagonal i en la GPU dimensión del grid* $(1\,x\lceil\frac{Nc}{bx}\rceil)$ *)*
5: aplicarRotaciones(A) *(aplica las rotaciones de la diagonal i en la GPU* $(Nc\,x\lceil\frac{n+p}{bx}\rceil)$ *)*
6: fin para

Una vez calculada R y c, ya podemos resolver el sistema triangular superior $R_1 x = c_1$, para ello utilizamos la rutina *cublasStrsv* implementada en CUBLAS, que resuelve un sistema de ecuaciones de la forma $(Ax = b)$.

Capítulo 4

Aplicación en Sistemas MIMO

4.1 Descripción del Problema

El problema en el cual se centra este capítulo es el problema de mínimos cuadrados en espacios discretos:

$$\min_{s \in A^m} \|x - Hs\|^2 \qquad (1)$$

donde $x \in \mathbb{R}^{nx1}$, $H \in \mathbb{R}^{nxm}$ y A es un conjunto de valores discretos enteros, por lo que A^m es un subconjunto isomorfo del espacio entero m-dimensional $\mathbb{Z}^m$.

Usualmente este problema se describe en términos de retículas (*lattices*). Si los elementos de A están igualmente espaciados, el conjunto A^m forma una retícula rectangular como la mostrada en la figura 4.1(a). Cuando los elementos de A^m son multiplicados por la matriz **H** entonces la retícula sufre una especie de deformación, y puede resultar similar a como se muestra en la figura 4.1 (b). El problema (1) es equivalente al problema de encontrar en una retícula generada por una matriz **H** el punto más cercano a un punto dado *x*. Este problema es conocido por las siglas CVP(Closest Vector Problem), y se conoce que es un problema NP-completo [19].

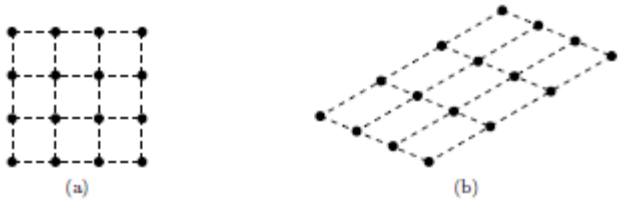


Figura 4.1 (a) Retícula Rectangular (b) Retícula deformada

Este tipo de problemas aparecen en el campo de las comunicaciones inalámbricas, donde las señales de comunicación digitales son enviadas a través de sistemas con múltiples antenas de envío y recepción (multiple input-multiple output o sistemas MIMO) y deben ser correctamente decodificadas[20,21].

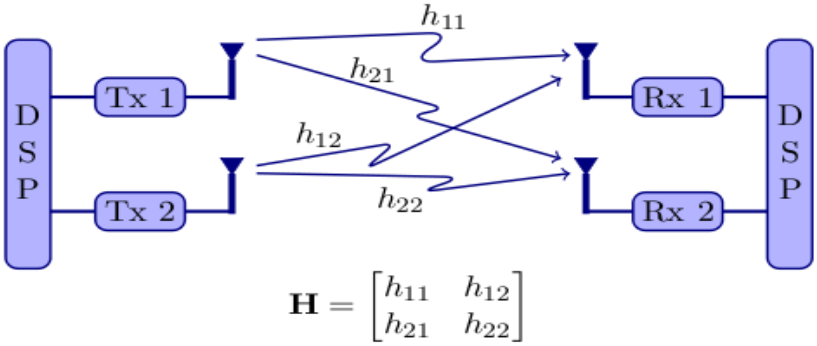


Figura 4.2 Sistema MIMO 2x2

Estos sistemas están compuestos por M antenas transmisoras y N antenas receptoras, a través de las

cuales una señal $s=[s_{1,}s_{2,\ldots},s_M]^T\in\mathbb{C}^m$ es transmitida. En la figura 4.2 se muestra un sistema MIMO 2x2 , es decir, con dos antenas de transmisión y dos antenas de recepción. La señal se propaga a través de varios canales, pues existe un canal entre cada antena transmisora y cada antena receptora. La propagación a través de este sistema se puede representar a través de una matriz de canal, denotada por **H.** El elemento h_{ij} representa la función de transferencia compleja entre la antena transmisora *j* y la antena receptora *i*. La parte real e imaginaria de cada componente de la señal transmitida pertenece a un conjunto discreto A, y finito (|A| = L) que se llama constelación o alfabeto de símbolos.

En la figura 4.3 se muestra el modelo general del proceso de transmisión-recepción de un sistema MIMO. La señal recibida $x\in\mathbb{C}^N$ es una combinación lineal de la señal transmitida *s* más un ruido *v* captado en las antenas receptoras.

$$x=\boldsymbol{H}s+v \qquad (2)$$

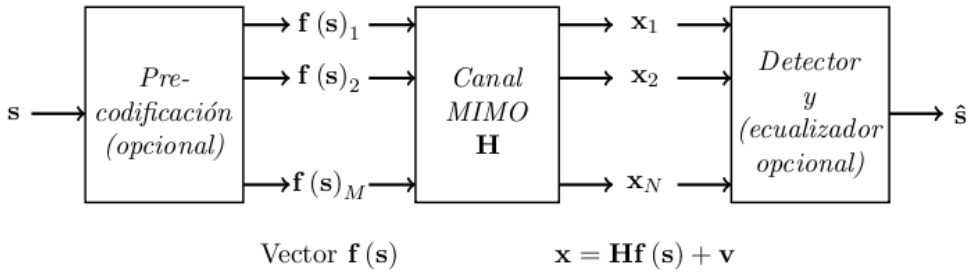


Figura 4.3 Modelo general de un sistema de comunicaciones MIMO

En este caso, la matriz de canal **H** es una matriz general de valores complejos con N filas y M columnas que modela la respuesta del canal MIMO. Con el fin de evitar la aritmética compleja, normalmente se transforma el sistema (2) a un modelo real, donde el vector *s* de dimension m=2M, y los vectores *x* y *v* de dimension n=2N se definen como :

$$s=[\ \Re(s)^T\ \Im(s)^T\ ]^T, x=[\ \Re(x)^T\ \Im(x)^T\ ]^T,$$
$$v=[\ \Re(v)^T\ \Im(v)^T\ ]^T$$

y la matriz **H** de dimensión nxm como:

$$H=\begin{bmatrix}\Re(H) & \Im(H)\\ -\Im(H) & \Re(H)\end{bmatrix}$$

Existen una gran variedad de métodos que obtienen una solución exacta o aproximada al problema (1). Los llamados métodos sub-óptimos o heurísticos obtienen una solución aproximada basándose en la inversión de la matriz de canal y presentan una complejidad temporal cúbica. Por otro lado los llamados métodos óptimos sí obtienen la solución exacta al problema pero a un coste computacional mayor, pues basan su funcionamiento en el recorrido de un árbol de posibles soluciones siguiendo una estrategia *Branch-and-Bound.* Los métodos Sphere-Decoding[22,23] pertenecen a esta última categoría y es el método que se utilizará en este capítulo.

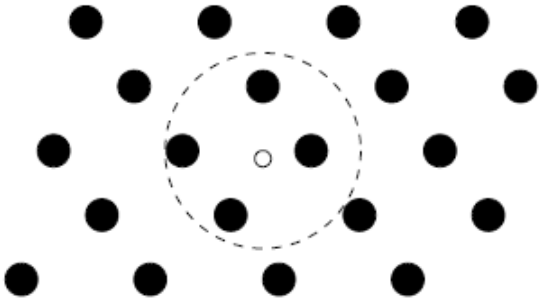


Figura 4.4 Idea del Sphere Decodong

Los métodos de decodificación esférica (SD-Sphere-Decoding) son métodos que siguen un esquema de Ramificación y Poda. La idea del algoritmo es intentar buscar sólo aquellos puntos de la retícula que están dentro de la esfera con centro en el vector dado *x*, y un radio *r* inicialmente escogido (ver figura 4.4). De esta manera se reduce el espacio de búsqueda y por tanto la complejidad computacional. Por lo tanto hay que encontrar todos los vectores $s \in A_L^m$ tales que

$$r^2 \geqslant \|x - Hs\|^2 \quad (3)$$

y posteriormente seleccionar el que minimice la función objetivo.

Para determinar los puntos de una retícula que se encuentran en el interior de una hiper-esfera m-dimensional se parte de un caso base con m=1, donde determinar los puntos en su interior es equivalente a determinar los puntos en el interior de un intervalo. Podemos usar esta observación para ir de la dimensión k a la k+1.
Supongamos que hemos determinado todos los puntos k-dimensionales que están en una esfera de radio r. Luego, de esos puntos k-dimensionales, el conjunto de valores admisibles de la k+1-dimension del mismo radio r forman un intervalo. Esto significa que podemos determinar todos los puntos de una esfera m-dimensional, de radio r , determinando sucesivamente todos los puntos solución en esferas de dimensiones menores 1,2,...,m y el mismo radio r.

Para subdividir el problema en estos diversos subproblemas, se realiza previamente la descomposición QR de la matriz H, donde R es una matriz triangular de dimensiones n x m y $R_1 \in R^{mxm}$ con los elementos de la diagonal todos positivos y $Q = [Q_1 Q_2]$ es una matriz ortogonal m x m. Las matrices Q_1 y Q_2 representan las primeras n y las últimas m-n columnas ortogonales de Q. La condicion (3) se puede escribir como

$$r_2 \geq \left\| x - [Q_1 Q_2] \begin{bmatrix} R_1 \\ 0 \end{bmatrix} s \right\|^2 = \|Q_1^T x - R_1 s\|^2 + \|Q_2^T x\|^2$$

Para determinar los puntos en una esfera m-dimensional el algoritmo construye un árbol donde las ramas en el k-ésimo nivel del árbol corresponden a los puntos en el interior de la esfera de radio r y dimensión k (ver figura 4.5). Por lo tanto la complejidad de este algoritmo dependerá del tamaño del árbol, es decir, del número de puntos visitados en cada una de las dimensiones. Sin embargo, con la aplicación de algoritmos alternativos de preprocesado/ordenación se puede conseguir una complejidad menor.

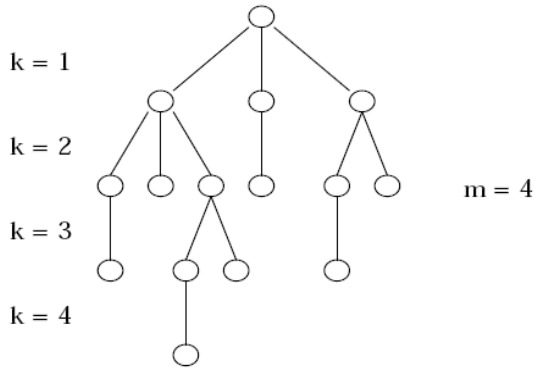


Figura 4.5 Ejemplo de árbol generado para determinar puntos de la retícula en una esfera de 4 dimensiones

El preprocesado y ordenación estándar consiste en el cálculo de la factorización QR sobre la matriz de canal H y la ordenación de las componentes de la señal transmitida de forma natural pero empezando por el último $s_m, s_{m-1}, \ldots, s_1$. El preprocesado consiste entonces en encontrar una matriz de permutación P, cuyas componentes permitan modificar el sistema de la siguiente forma:

$$x = Hs + v = HPP^{-1}s + v = \tilde{H}z + v$$

En [24] se presenta un algoritmo de ordenación basado en el gradiente y en la función objetivo. El gradiente devuelve la tasa de cambio de la función objetivo respecto a cada componente. El gradiente para cualquier punto x de la función objetivo $||Hs - x||^2$ puede ser calculado como $H^T(Hs-x)$. Las componentes con mayor valor absoluto del gradiente son las que implican un mayor cambio en el valor de la función objetivo. Usando un razonamiento similar al usado en [25], proponen el siguiente procedimiento: dada una matriz de canal H y una señal recibida x.

1)Obtener un vector valido de la constelación s (el ordenamiento será mejor si s esta cerca de la solución)
2)Calcular el gradiente $g = H^T(Hs-x)$[3].
3)Ordenar las componentes (columnas de H) dependiendo de los valores absolutos de el gradiente g.

La desventaja que introduce este método es que depende de la señal recibida, por lo que para cada nueva señal hay que recalcular la factorización QR.

4.2 Aproximaciones

En el método basado en la ordenación del gradiente, hay que utilizar la matriz de canal y la señal a decodificar para calcular la ordenación óptima. Esto tiene un inconveniente y es que para cada nueva señal hay que recalcular la factorización QR.

Teniendo en cuenta esta característica podemos seguir dos estrategias:

1. Recalcular en tiempo real la factorización QR para cada nueva señal.
2. Añadir una fase de preproceso en la que se calculen todas las posibles factorizaciones, es decir, una factorización por cada uno de las ordenaciones posibles de las columnas de la matriz de canal H. Seguir esta estrategia supone calcular para una matriz de canal de n columnas, n! factorizaciones, por lo tanto hay que conseguir minimizar al máximo este

3 $g = H^T x$, puesto que el gradiente únicamente depende de la señal recibida y no del punto escogido en la retícula.

tiempo de cómputo para que sea viable utilizar esta aproximación. Es obvio que el valor de n debe ser pequeño para que esta estrategia tenga sentido. Pero planteamos la estrategia para utilizarla en problemas 4x4 u 8x8. Existen sistemas MIMO en aplicaciones reales cuyas dimensiones no son superiores a las descritas, y consideran un logro llegar a sistemas MIMO con matrices de canal 8x8 [26,27].

A continuación se describen los dos algoritmos que realizan el proceso de decodificación de una señal de entrada con el uso del *Sphere-Decoding* (SD) según las dos aproximaciones anteriormente vistas.

Algoritmo 4.1: Aproximación 1

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema

1: b ← señal recibida a decodificar
2: gradiente ← H^Tb
3: HP ← matriz H permutada con el vector gradiente
4: [Q,R] ← Factorización de la matriz HP
5: SD(Q,R,y,constelación,radio)

En el Algoritmo 4.1 vemos como no es necesario una fase de preproceso previa a la ordenación del gradiente, puesto que conforme el sistema va recibiendo señales y se lleva a cabo una nueva factorización QR. Esto supone que para dos señales b_1, b_2 que dieran lugar a una misma ordenación habría que calcular de nuevo la factorización QR, aunque ya hubiera sido calculada para señales anteriores (b_1), desaprovechando de esta manera tiempo de cómputo.
Sin embargo el Algoritmo 4.2 añade una capa de prepoceso anterior a la puesta en marcha del sistema. En esta fase para cada una de las posibles permutaciones de la matriz de canal H se calcula una factorización. De manera que una vez se comience a recibir señales, solamente será necesario calcular la ordenación óptima y llamar al Sphere-Decoding con la QR correspondiente.

Algoritmo 4.2: Aproximación 2

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema
1: [Q,R] ← Factorización QR de H en formato compacto
2: Calcular P: conjunto de permutaciones 1,2,...,n!
3: Calcular las n! Factorizaciones QR de la matriz H
4: b ← señal recibida a decodificar
5: gradiente ← H^Tb
6: g ← Obtener la permutación correspondiente al gradiente
7: SD(Q_g,R_g,y,constelación,radio)

4.3 Algoritmos Paralelos

En este apartado se presentan lo algoritmos diseñados para resolver el problema de decodificación de MIMO visto en la sección anterior, utilizando diferentes entornos paralelos: aceleradores hardware (GPU), memoria compartida (UPC y OMP) y utilizando librerías algebraicas eficientes

como es el cuLAPACK.

4.3.1Algoritmos en CUDA

En este apartado se presentan 3 algoritmos implementados en CUDA, dos de ellos se basan en la aproximación 2 vista en el apartado anterior, el otro sin embargo se basa en la primera aproximación y utiliza la librería cuLAPACK para calcular las factorizaciones QR. Los dos algoritmos implementados en CUDA que se basan en la aproximación 2, vista en el apartado anterior, difieren únicamente en el momento en el que se copian los datos de la GPU a la CPU.

Método I – Factorización con cuLAPACK

El algoritmo implementado con el uso de la librería cuLAPACK sigue la Aproximación 1 vista en el punto 4.1. En este caso la factorización de la matriz permutada se realiza en la GPU llamando a la función que nos ofrece dicha librería para la factorización QR, esta función se llama *culaDeviceSgeqrf,* una vez finalizado el cálculo, se realiza la copia entre las memorias como puede observarse en la figura 4.6.

Algoritmo 4.3: Método I para la resolución con CULA

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema
Salida: Estimación de la señal transmitida $s \in A^m$
1: **mientras** *se reciban señales* **hacer**
2: b ← señal recibida a decodificar
3: gradiente ← H^Tb
4: HP ← matriz H permutada con el vector gradiente
5: Enviar HP a la GPU
6: [Q,R] ← Factorización de la matriz HP llamada a *culaDeviceSgeqrf*
7: z ←sphdec(Q,R,y,constelación,radio)
8: s ← z permutada con gradiente
9: **finmientras**

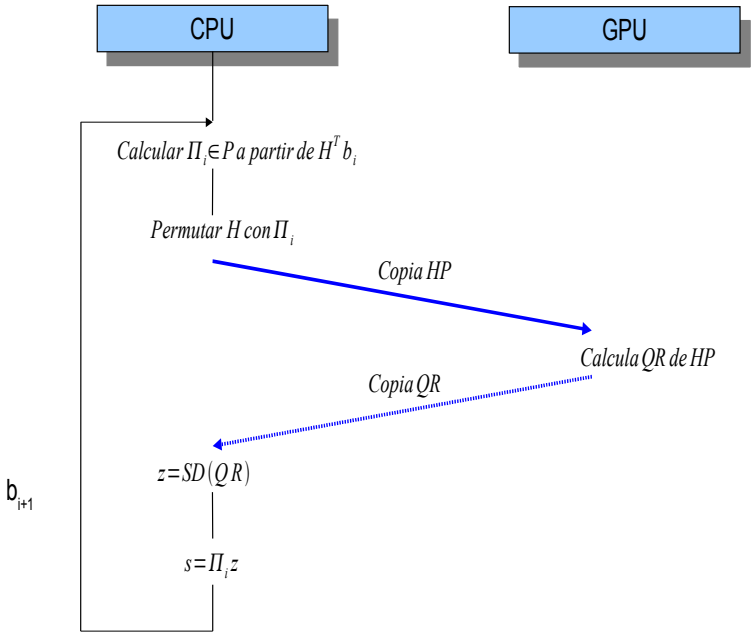


Figura 4.6 Esquema de envío de datos entre CPU y GPU en el Método I

Método II – Copia directa

En el Método II, en la capa de preproceso las n! factorizaciones se calculan en la GPU y posteriormente se copian a la memoria de la CPU. Una vez el sistema está preparado para recibir señales para su decodificación ya no es necesario interactuar con la tarjeta gráfica si no que se trabaja únicamente con la CPU. En la figura 4.7 puede verse con mayor claridad el momento en que se realiza dicha copia.

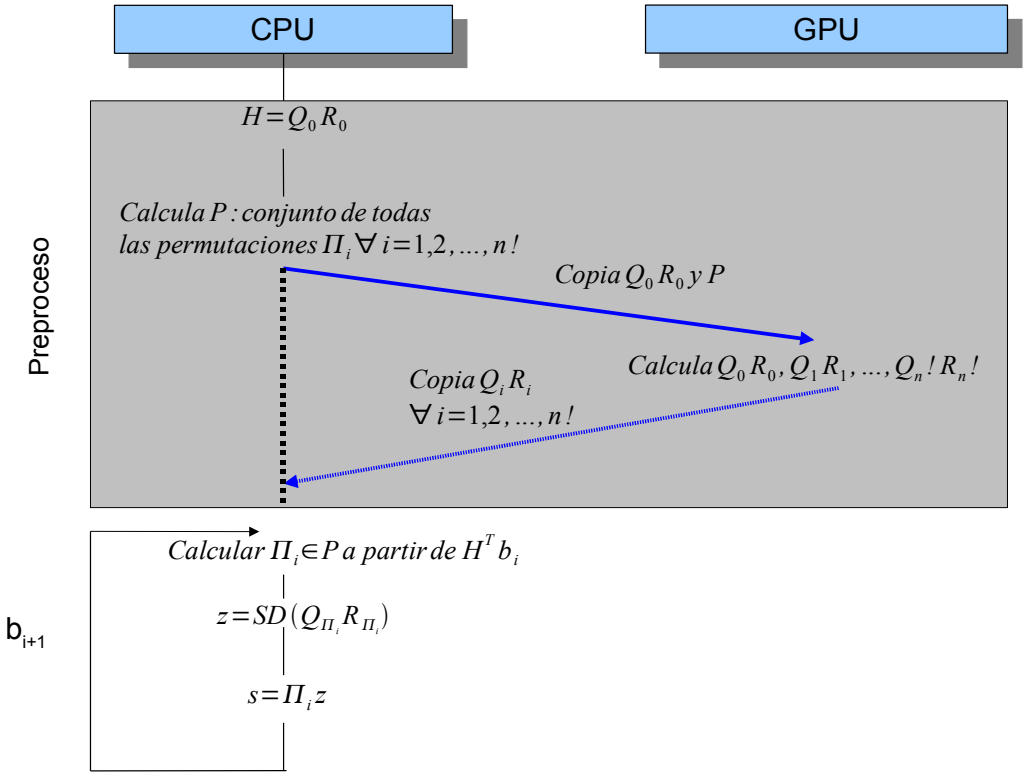


Figura 4.7 Esquema de envío de datos entre CPU y GPU en Método II

Estructuras	Número de Elementos en doble precisión			
Q_0R_0 (QR matriz de canal)	1.6e1	3.6e1	6.4e1	1.0e2
P	9.6e1	4.3e3	3.2e5	3.63e7
$Q_iR_{i\ i=0,\ldots,n!}$	3.8e2	2.6e4	2.6e6	3.63e8
Espacio de trabajo auxiliar	1,2e2	2,6e2	4,6e2	7,2e2
Total Dobles	6.2e2	3.1e4	2.9e6	4.0e8
Memoria Requerida	4.81 Kb	238.53 Kb	22.15 Mb	2.97 Gb
m=n	**4**	**6**	**8**	**10**

Tabla 4.1 Requerimientos de memoria tanto en la CPU como en la GPU del algoritmo 4.4

El tamaño de la matriz de canal será un factor crítico a la hora de optar por esta aproximación, debido a la gran cantidad de factorizaciones que hay que tener almacenadas en memoria. En la tabla 4.1 puede verse el total de memoria requerida tanto para la CPU como para la GPU según el tamaño de la matriz de canal. Como la GPU tiene 4GB de memoria se han tomado medidas hasta un tamaño

de matriz de 10x10.

Algoritmo 4.4: Método II para la resolución con GPU

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema
Salida: Estimación de la señal transmitida $s \in A^m$

1: [QR] ← Factorización QR de H en formato compacto
2: Calcular P: conjunto de permutaciones 1,2,...,n!
3: Enviar QR y P a la GPU
4: *calculaQR (* calcula las n! Descomposiciones QR de H; se ejecuta en la GPU, ver apartado *)*
5: copiar todas las n! QRs de la GPU
6: **mientras** *se reciban señales* **hacer**
7: b ← señal recibida a decodificar
8: gradiente ← H^Tb
9: g ← Obtener la permutación correspondiente al gradiente
10: [Q,R] ← desempaquetar QR_g
11: z ←SD(Q,R,y,constelación,radio)
12: s ← z permutada con gradiente
13: **finsi**

Método III – Copia en diferido

Por el contrario en el Método III las factorizaciones llevadas a cabo en el dispositivo no son copiadas a la CPU, sino que a medida que van procesándose las señales se copia la factorización correspondiente a memoria de la CPU. En la figura 4.8 se puede observar la diferencia con el método anterior.

Cuando la CPU calcula la QR que quiere copiar de la GPU en el paso 8, no es necesario ningún envío de información a la GPU para que le envía la QR correspondiente, ya que es la CPU la que explícitamente realiza una copia de m*n elementos desde la GPU a la CPU con la función

*cudaMemcpy(QR,d_QR+(gradiente*m*n),m*n*sizeof(double),cudaMemcpyDeviceToHost)*

En este caso los requerimientos de memoria requerida en la CPU es inferior al del Algoritmo 4.4, en la Tabla 4.2 puede verse como solo es necesario un espacio de unos 300Mb para matrices 10x10. Sin embargo que la GPU mantiene los requerimientos vistos en la Tabla 4.1.

Estructuras	**Número de Elementos en doble precisión**			
Q_0R_0	1.6e1	3.6e1	6.4e1	1.0e2
P	9.6e1	4.3e3	3.2e5	3.63e7
Espacio de trabajo auxiliar	1,2e2	2,6e2	4,6e2	7,2e2
Total Dobles	6.2e2	3.1e4	2.9e6	4.0e8
Memoria Requerida	1.81 Kb	36.0 Kb	2.46 Mb	276.9 Mb
m=n	4	6	8	10

Tabla 4.2 Requerimientos de memoria en la CPU del algoritmo 4.5

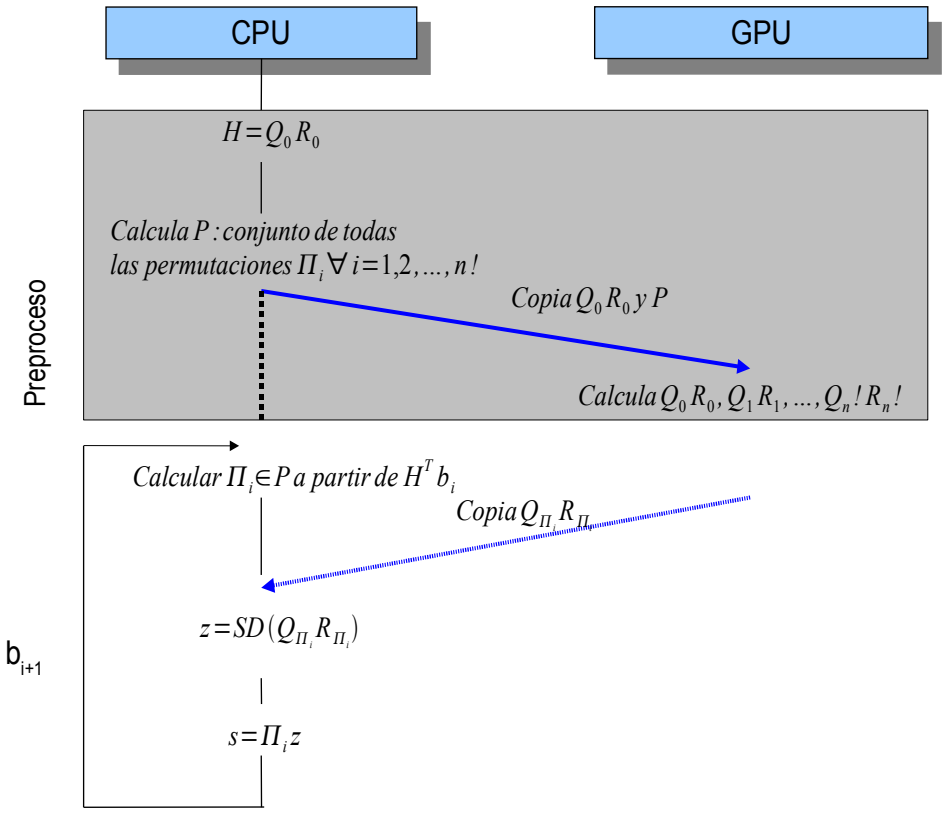


Figura 4.8 Esquema de envío de datos entre CPU y GPU en Método III

Algoritmo 4.5: Método III para la resolución con GPU

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema
Salida: Estimación de la señal transmitida $s \in A^m$

1: [QR] ← Factorización QR de H en formato compacto
2: Calcular P: conjunto de permutaciones 1,2,...,n!
3: Enviar QR y P a la GPU
4: *calculaQR* *(* calcula las n! Descomposiciones QR de H se ejecuta en la GPU, ver apartado *)*
5: **mientras** *se reciban señales* **hacer**
6: b ← señal recibida a decodificar
7: gradiente ← H^Tb
8: g ← Obtener la permutación correspondiente al gradiente
9: copiar QR_g de la GPU
10: [Q,R] ← desempaquetar QR_g
11: z ←SD(Q,R,y,constelación,radio)
12: s ← z permutada con gradiente
12: **finsi**

4.3.2 Algoritmos en Unified Parallel C

El Algoritmo 4.6 implementado en UPC, también sigue la Aproximación 2. Las n! factorizaciones se calculan en paralelo por los hilos de la ejecución. Una vez comienzan a recibirse señales, un

único hilo calcula el gradiente, el hilo que calculó la factorización asociada a ese gradiente es el encargado de ejecutar el decodificador ya que es el que tiene en su memoria local compartida esta QR y por tanto accederá a los datos necesarios con mayor rapidez que el resto de hilos.

Algoritmo 4.6: Método I para la resolución con UPC

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema
Salida : Estimación de la señal transmitida $s \in A^m$
1: [QR] ← Factorización QR de H en formato compacto
2: Calcular P: conjunto de permutaciones 1,2,...,n!
3: **En paralelo**:
4: **para** i=1,2,...,n! **hacer**
5: HP ← matriz H permutada con P(i)
6: [QR_i] ← Factorización QR de HP en formato compacto
7: **finpara**
8: **finparalelo**
5: **mientras** *se reciban señales* **hacer**
6: b ← señal recibida a decodificar
7: gradiente ← H^Tb
8: g ← Obtener la permutación correspondiente al gradiente
9: **si** p tiene *afinidad* con Qr_g **entonces** (* véase apartado 2.3.3)
10: [Q,R] ← desempaquetar QR_g
11: z← SD(Q,R,y,symbset,radius)
12 s ← z permutada con gradiente
13: **finsi**
14: **finsi**

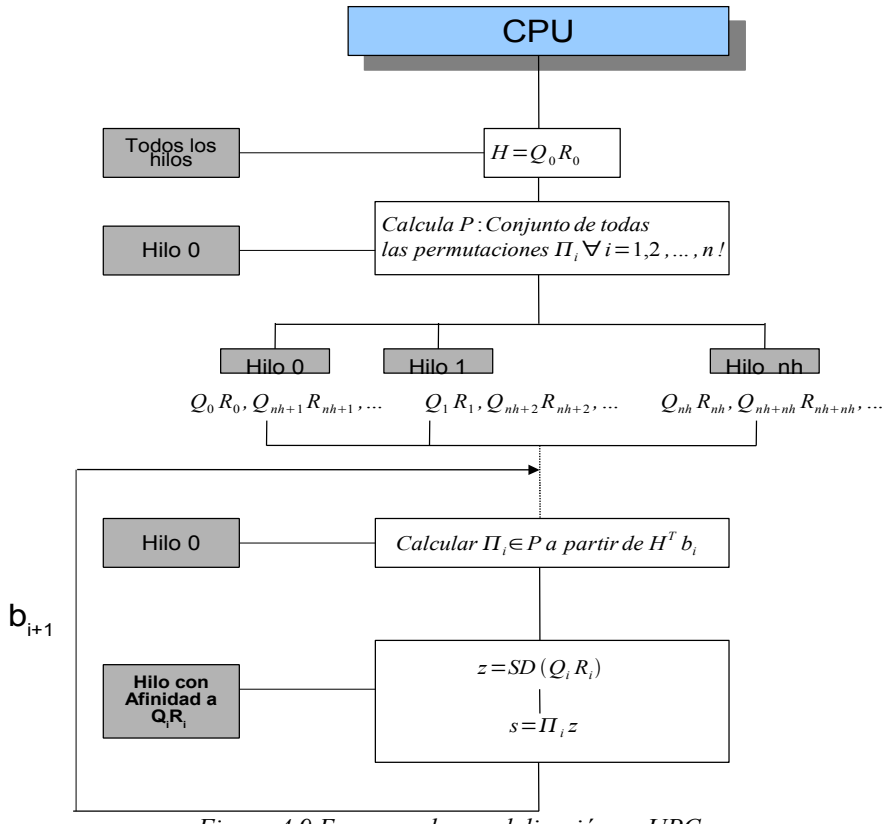


Figura 4.9 Esquema de paralelización en UPC

En la figura 4.9 puede verse como se distribuye el trabajo entre los distintos hilos, como puede observarse la parte de la paralelización se realiza en el momento del calculo de las n! factorizaciones, pero en la parte del decodificador se aprovecha de la característica que ofrece UPC de la localidad de datos.

En los sistemas de decodificación es frecuente que las señales se almacenen primero en un buffer conforme se van recibiendo, para ser procesadas a continuación. Hemos paralelizado también la decodificación de *k* señales que se conocen, y están almacenadas en un buffer (ver figura 4.10). Esta aproximación obtendría mucho beneficio en los algoritmos planteados. Las señales pueden almacenarse en una matriz B de tamaño *nxk,* entonces el cálculo de los gradientes de las *k* señales se calcula con una multiplicación de matrices $H^T B$. Cada una de las columnas de la matriz resultante será una permutación de la matriz de canal, sólo hace falta obtener el índice de las permutaciones en cuestión (ver algoritmo 4.16) y hacer que los hilos que tienen afinidad con estas descomposiciones apliquen el decodificador.
Existen dificultades para implementar este esquema en CUDA debido a que el algoritmo SD puede necesitar una memoria excesiva para cada señal.

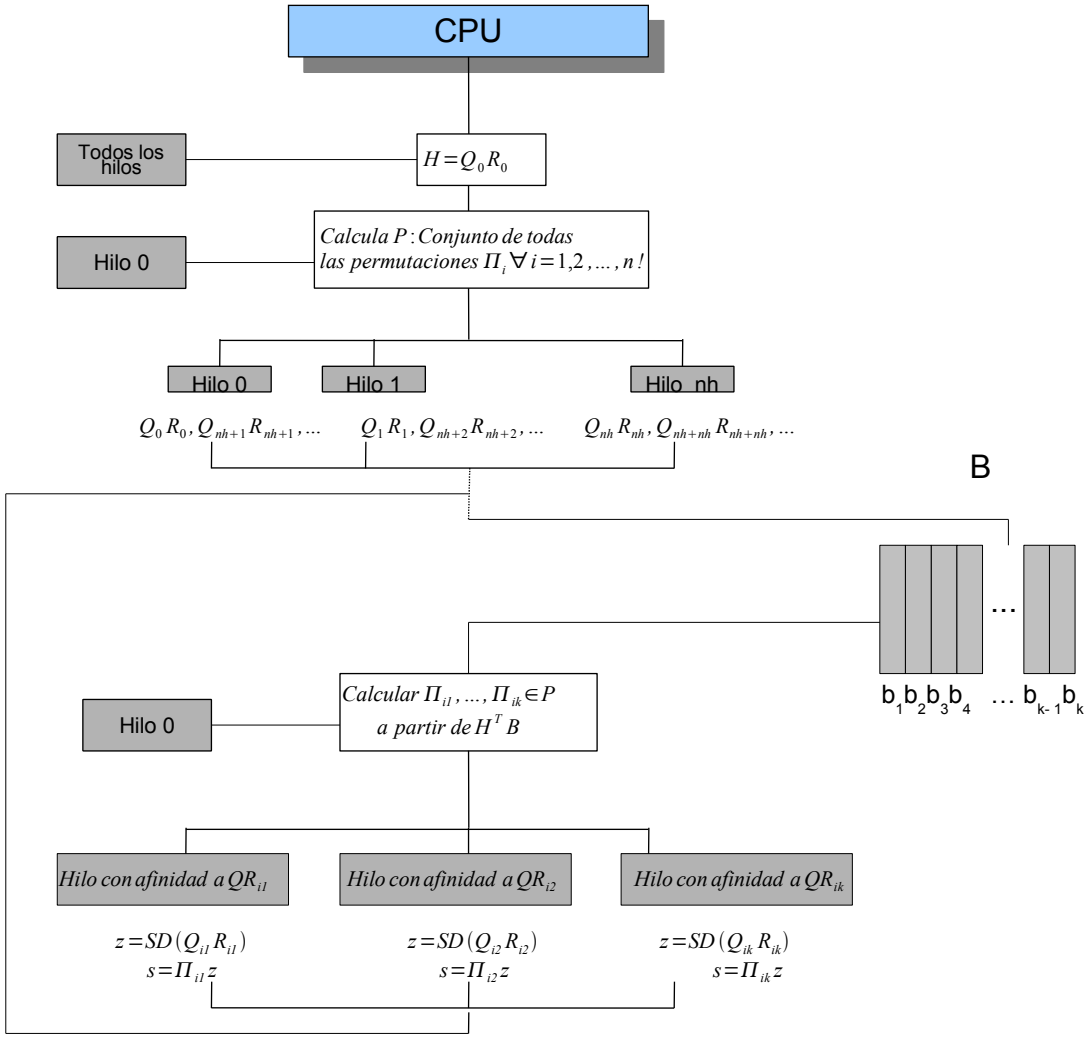


Figura 4.10 Esquema de paralelización en UPC con un buffer de señales

4.4 Implementación

A continuación se presentan los detalles de implementación de los algoritmos diseñados en el apartado anterior. La implementación de las funciones es común para todas las versiones realizadas independientemente de la entorno paralelo usado.
Estas subrutinas se pueden englobar en grupos los siguientes grupos

Factorización QR en formato compacto		
Algoritmo	**Nombre**	**Descripción**
4.7	CQR	Calcula la factorización QR en formato compacto de una matriz.
4.8	getGivens	Obtiene el coseno y seno de una Rotación de Givens
4.9	applyGivens	Aplica una rotación a una matriz.
4.10	getPacks	Obtiene una rotación en formato compacto.
4.11	unPack	Decompacta una rotación.
4.12	desempaquetarR	Obtiene la matriz R de una matriz QR en formato compacto.
4.13	desempaquetarQ	Obtiene la matriz Q de una matriz QR en formato compacto.
Problema MIMO		
Algoritmo	**Nombre**	**Descripción**
4.14	setPermutacion	Calcula el conjunto de todas las permutaciones de una matriz.
4.15	getGradiente	Calcula el gradiente de una señal
4.16	shell_short:	Ordena un vector en valor absoluto de forma decreciente
4.17	getPermutacion	Obtiene el indice de la permutación.
Decodificación de la Señal		
Algoritmo	**Nombre**	**Descripción**
4.18	SD	Sphere Decoder
4.19	sphdec_core	Subrutina principal del decodificador

Algoritmo 4.7: (CQR) Factorización QR en formato compacto

Entrada: $H \in R^{mxn}\ m \geq n$
Salida: H, factorización QR de H en formato compacto

```
1:  para i=1,2,...,n hacer
2:     para j=m,m-1,...,i+1 hacer
3:           si H(i,j) ≠ 0 entonces
4:                 [c,s] ← getGivens(H(i-1,j),H(i,j))
5:                 H ← applyGivens(H,i-1,i,c,s,j)
6:                 H(i,j) ← getPacks(c,s);
7:              finsi
8:        finpara
9:  finpara
```

Debido a la gran cantidad de espacio que ocupaba en memoria el conjunto de todas las posibles factorizaciones $QR_{i\ i=1,2,...,n!}$ se decidió utilizar el formato compacto visto en el apartadp 3.2.2. La

matriz R es almacenada en la parte triangular superior de la matriz resultado, en el resto se almacenan las rotaciones de givens en formato compacto. El coste de este algoritmo[1] es de $3n^2(m-\frac{n}{3})$ flops . Este coste no contempla los 2 flops necesarios para empaquetar cada una de las rotaciones (ver Algoritmo 4.10).

Algoritmo 4.8: (getGivens) Obtiene el coseno y seno de una Rotación de Givens

Entrada: $a, b \in R$
Salida: cos ,sen , coseno y seno de la Rotación de Givens
1: **si** |b| = 0 **entonces**
2: cos ← 1
3: sen ← 0
4: **sino**
5: **si** |b| > |a| **entonces**
6: t ← -(a/b)
7: sen ← $\frac{1}{\sqrt{1+t^2}}$
8: cos ← t*sen
9: **sino**
10: t ← -(b/a)
11: cos ← $\frac{1}{\sqrt{1+t^2}}$
12: sen ← t*cos
13: **finsi**
14: **finsi**

Algoritmo 4.9: (applyGivens) Aplica una rotación a una matriz.

Entrada: $H e Rmxn m \geq n$ **,** matriz de canal de sistema
$i, k, col \in Z$ y cos, $sen \in R$, rotación de Givens
Salida: Matriz H tras aplicar la rotación
1: **para** i=col,col+1,...,n **hacer**
2: t1 ← H(i,j)
3: t2 ← H(k,j)
4: H(i,j) ← cos*t1-sen*t2
5: H(k,j) ← sen*t1+cos*t2
6: **finpara**

Algoritmo 4.10: (getPacks) Obtiene una rotación en formato compacto.

Entrada: cos, $sen \in R$ **,** rotación de Givens
Salida: $p \in R$, rotación de Givens en formato compacto
1: **si cos = 0 entonces**
2: p ← 0
3: **sino**
4: **si** |sen| < |cos| **entonces**
5: p ←sign(cos)*sen/2

```
6:    sino
7:          p ← 2*sign(sen)/cos
8:    finsi
9: finsi
```

Algoritmo 4.11:(unPack) : Descompacta una rotación

Entrada: $a \in R$
Salida: cos, $sen \in R$, rotación de Givens

```
1: si a = 0 entonces
2:   cos ← 0
3:   sen ← 1
3: sino
4:    si |a| < 1 entonces
5:       sen ← 2*a
6:       cos ← √(1 − sen²)
7:    sino
8:       cos ← 2/a
9:       sen ← √(1 − cos²)
8:    finsi
9: finsi
```

Para decodificar la señal es necesario proporcionarle al decodificador las matriz Q ortogonal y R triangular superior. Para ello se han implementado dos rutinas que a partir de una factorización QR en formato compacto devuelve las matrices Q y R.

Algoritmo 4.12: (desempaquetaR) Obtiene la matriz R de una matriz QR en formato compacto.

Entrada: $A \in R^{mxn}$, matriz compacta
Salida: $R \in R^{mxn}$, matriz triangular superior

```
1: para i=1,2,...,n hacer
2:    para j=1,2,...,i+1 hacer
3:       R(j,i) = A(j,i)
4:    finpara
5: finpara
```

Algoritmo 4.13: (desempaquetaQ) Obtiene la matriz Q de una matriz QR en formato compacto.

Entrada: $A \in R^{mxn}$, matriz compacta
Salida: $Q \in R^{mxm}$, matriz triangular superior

```
1:  Q ← I^mxm
2:  para i=1,2,...,n hacer
3:     para j=m,m-1,...,i+1 hacer
4:           si H(i,j) ≠ 0 entonces
5:                 [c,s] ← unPack(A(i,j))
6:                 Q ← applyGivens(Q,i-1,i,c,s,0)
```

```
7:          finsi
8:      finpara
9:  finpara
```

Para poder paralelizar el calculo de todas las factorizaciones es necesario calcular todo el conjunto de permutaciones P de manera que cada procesador trabaje sobre un subconjunto de éstas. Para ello se ha implementado un método que dado el número de columnas de la matriz de canal, calcula todas las posibles permutaciones de éstas, además todas ellas están ordenadas ascendentemente por el número de índice de la columna.

Algoritmo 4.14 : (setPermutacion) Calcula el conjunto de todas las permutaciones de una matriz.

Entrada: $n \in Z$, numero de columnas de la matriz
Salida: $v \in R^{n!xn}$ **,** conjunto de permutaciones

```
1: repite ← true
2: para i=1,2,...,n hacer
3:    v(i) = i
4: finpara
5: vAll ← v
6: cont ← 1
7: mientras repite = true hacer
8:    i ← n–2
9:    mientras (v(i) > v(i+1)) y (i > 0)  hacer
10:       i ← i-1
11:   finmientras
12:   si (i = 0) y (v(i) > v(i+1)) entonces
13:      repite = false
14:   sino
15:      j ← n-1
16:      mientras v(i) > v(j)  hacer
17:      j ← j-1
18:      finmientras
19:      temp ← v(j)
20:      v(j) ← v(i)
21:      v(i) ← temp
22:      r ← n-1
23:      s ← i+1
24:      mientras r > s  hacer
25:         temp ← v(s)
26:         v(s) ← v(r)
27:         v(r) ← temp
28:         r ← r-1
29:         s ← s+1
30:      finmientas
31:      vAll(cont) ← v
32:      cont ← cont+1
33:   finsi
```

34: **finmientas**

Algoritmo 4.15 (getGradiente) Calcula el gradiente de una señal

---

Entrada: $H \in R^{mxn}$, matriz de canal del sistema
$b \in R^{m}$, señal recibida
Salida: $g \in Z^{n}$,vector gradiente
1: z ← H^{T}b
2: g ← shell_sort(|z|)

Algoritmo 4.16 (shell_short) Ordena un vector en valor absoluto de forma decreciente.

Entrada: $z \in R^{n}$ gradiente
Salida: G , vector con los índices de los valores de z en orden decreciente
1: in ← size(z)/2
2: **mientras** in > 0 **hacer**
3: **para** i=in,in+1,...,size **hacer**
4: **j** ← i
5: temp ← z(i)
6: tempP ← G(i)
7: **mientras** (j > in) y (z(j-in) >temp) **hacer**
8: z(j) ← z(j-in)
9: G(j) ← G(j-in)
10: j ← j – in
11: **finmientras**
12: z(j) ← temp
13: G(j) ← tempP
14: **finpara**
15: in ← in/2
16: **finmientras**

Cuando hemos calculado el gradiente es necesario saber que determinado índice de permutación le corresponde. Para ello se ha realizado una rutina que dado un determinado orden de las columnas de la matriz de canal devuelve el índice de permutación que le corresponde en el conjunto de las permutaciones P.

Algoritmo 4.17 (getPermutacion) Obtiene el indice de la permutación.

Entrada: $z \in R^{n}$ **,** vector gradiente
Salida: ind , índice de la permutación

1: ind ← z(0)*(n-1)!
2: **para** i=1,2,...,n **hacer**
3: **para** j=1,2,...,i **hacer**
4: **si** z(i) > z(j) **entonces**
5: num ← num +1
6: **finsi**

7: **finpara**
8: ind ← ind +(z(i)-num)*(n-i-1)!
9: num ← 0
10: **finpara**

El algoritmo utilizado para la decodificación de las señales es el Sphere-Decoding . En este algoritmo se utilizan las siguiente variables globales:

SPHDEC_RADIUS e R
RETVAL e R^m
TMPVAL e R^m
SEARCHFLAG e Z

Algoritmo 4.18 (SD) Sphere Decoder

Entrada: $Q \in R^{mxm}$, matriz ortogonal
$R \in R^{mxn}$, matriz triangular superior
$y \in R^m$, señal recibida
$symbset \in R^t$, constelación
$radius \in R$, radio de búsqueda
Salida: La estimación de la señal transmitida $s \in A^m$

1: z ← Q^Ty
2: SPHDEC_RADIUS ← radius
3: sphdec_core(z,R,symbset,n,0)
4: **si** SEARCHFLAG > 0 **entonces**
5: r ← RETVAL
6: **sino**
7: r ← 0
8: **finsi**

Algoritmo 4.19 (sphdec_core) Subrutina principal del decodificador

---

Entrada: $z \in R^m$, señal recibida
$R \in R^{mxn}$, matriz triangular superior
$symbset \in R^t$, constelación
$layer \in Z$
$dist \in R$

1: **si** layer = 1 **entonces**
2: **para** i=1,2,...,t **hacer**
3: TMPVAL(1) ← symbset(i)
4: d ← |z(1)-R(1,:)*TMPVAL|^2 + dist
5: **si** d ≤ SPHDEC_RADIUS **entonces**
6: RETVAL ← TMPVAL
7: SPHDEC_RADIUS ← d
8: SEARCHFLAG ← SEARCHFLAG + 1

```
9:        finsi
10:    finpara
11: sino
12:    para i=1,2,...,t hacer
13:        TMPVAL(layer) ← symbset(i)
14:        d ← |z(layer)-R(layer,[layer:end])*TMPVAL(layer:end)|^2 + dist
15:        si d ≤ SPHDEC_RADIUS entonces
16:            sphdec_core(z,R, symbset,layer-1,d);
17:        finsi
18:    finpara
19: finsi
```

4.4.1 Detalles de Implementación con CUDA

A continuación vamos a describir el kernel *calculaQR* que se utiliza en los Algoritmos 4.4 y 4.5. Esta función será ejecutada por cada uno de los hilos que hayamos instanciado anteriormente a la llamada y tiene la siguiente forma.

Algoritmo 4.20: (calculaQR) Calcula n! Factorizaciones QR en la GPU

Entrada: $H \in R^{mxn} Rmxn$ m≥n matriz de canal de sistema
$P \in R^{n!xn}$ conjunto de todas las permutaciones
Salida: Calculo de la QR_{id}

```
1: idx ← blockIdx.x*blockDim.x+threadIdx.x;
2: id ← (blockDim.x*gridDim.x)*(blockDim.y*blockIdx.y) + threadIdx.y*(blockDim.x*gridDim.x)+idx;
3: si id < n! entonces
4:     HP ← matriz H permutada con el vector gradiente P(id)
5:     [QR_i] ← Factorización QR de HP en formato compacto
6: finsi
```

En el paso 3 vemos como es necesario una condición para poder realizar el cálculo, esto es necesario porque cuando creamos la dimensión del grid es posible que el número de hilos que hayamos creado sea superior a n!. Si dejamos a todos estos hilos realizar la operación su identificador de hilo creará un error de overflow al querer almacenar una factorización en un espacio de memoria que no está reservado.

Una vez todos los hilos han terminado las factorizaciones calculadas quedan almacenadas en la memoria de la GPU, por lo tanto para poder utilizarlas en el programa que corre sobre la CPU es necesario hacer una copia del dispositivo a la CPU de la misma manera que hacíamos al mandar la matriz H y las permutaciones a la GPU para que pudiera trabajar con estos datos.

4.4.2 Detalles de Implementación con UPC

En el Algoritmo 4.6 implementado con UPC vemos como en el paso 9 se utiliza el concepto de afinidad de una determinada posición de memoria o variable con un hilo (ver apartado 2.3.3).

La forma en la que se distribuyen los datos a través de la memoria se puede observar en la figura 4.10.

- La zona de memoria donde se almacenan las factorizaciones se distribuye en bloques de *mxn* elementos (*doubles en este caso*), es decir, repartimos las n! factorizacines cíclicamente entre los hilos.
- P está alojado en la memoria local compartida del hilo 0, por lo que para optimizar el proceso los hilos hacen una copia de la permutación a calcular a su memoria privada.
- Todos ellos tienen la factorización de la matriz de canal H original en su memoria privada *($Q_0 R_0$)* ya que será necesaria para realizar todas las factorizaciones.

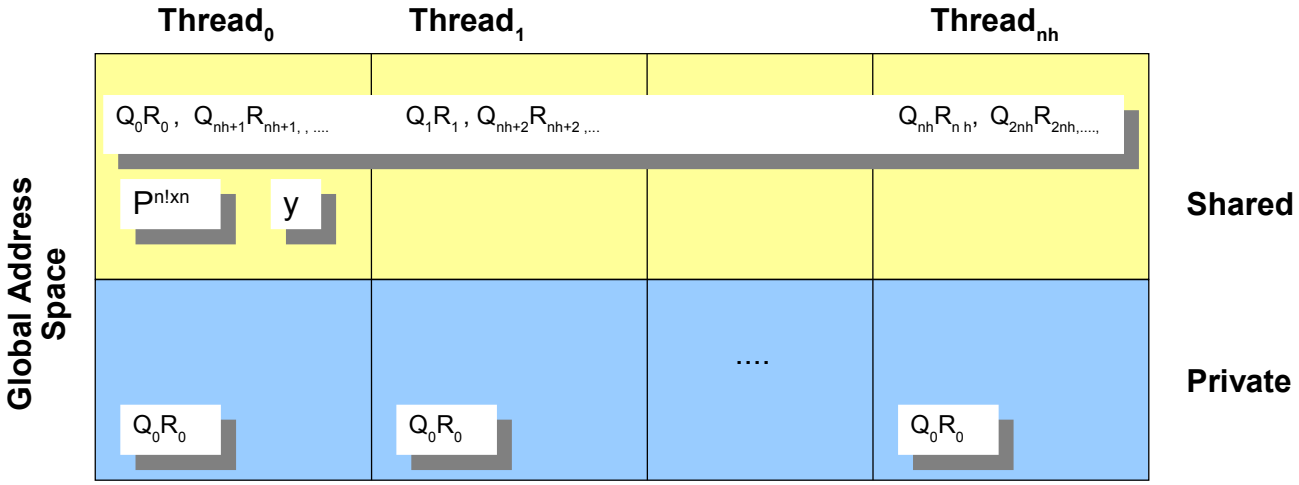


Figura 4.10 : Distribución de las variables a través de los hilos del Algoritmo 4.20

A continuación se muestra el Algoritmo 4.6 detallado, se han añadido las transferencias de datos entre las memorias compartida y local de los hilos necesarias para poder realizar las optimizaciones oportunas.

Algoritmo 4.21 : Método I para la resolución con UPC detallado

---

Entrada: $H \in R^{mxn}$ m≥n matriz de canal de sistema
Salida: Estimación de la señal transmitida $s \in A^m$

```
1:  [QR] ← Factorización QR de H en formato compacto
2:  P ← setPermutación(n)
3:  En paralelo:
4:     para i=1,2,...,n! Hacer
5:           copiar a memoria privada P(i)
6:           HP ← matriz H permutada con P(i)
7:           [QRi] ← Factorización QR de HP en formato compacto
8:           copiar [QRi] a memoria compartida
9:     finpara
10:  finparalelo
11:  mientras se reciban señales hacer
12:     b ← señal recibida a decodificar
13:     gradiente ← H^T b
```

14: g ← Obtener la permutación correspondiente al gradiente
15: **si** p tiene *afinidad* con Qr_g **entonces**
16: *copiar y a memoria privada*
17: *copiar Qr_g a memoria privada*
18: [Q,R] ← desempaquetar QR_g
19: z ← sphdec(Q,R,y,symbset,radius)
20: s ←z permutada con g
21: **finsi**
22: **finmientras**

A continuación presentamos un ejemplo de los movimientos entre las memorias para el cálculo de las *nh* primeras factorizaciones. En la figura 4.11 vemos como cada hilo copia a su memoria privada la permutación de las columnas correspondiente a esa factorización. A continuación con el uso de $Q_0 R_0$ calculan la QR. Una vez calculada se copia a memoria *shared.*

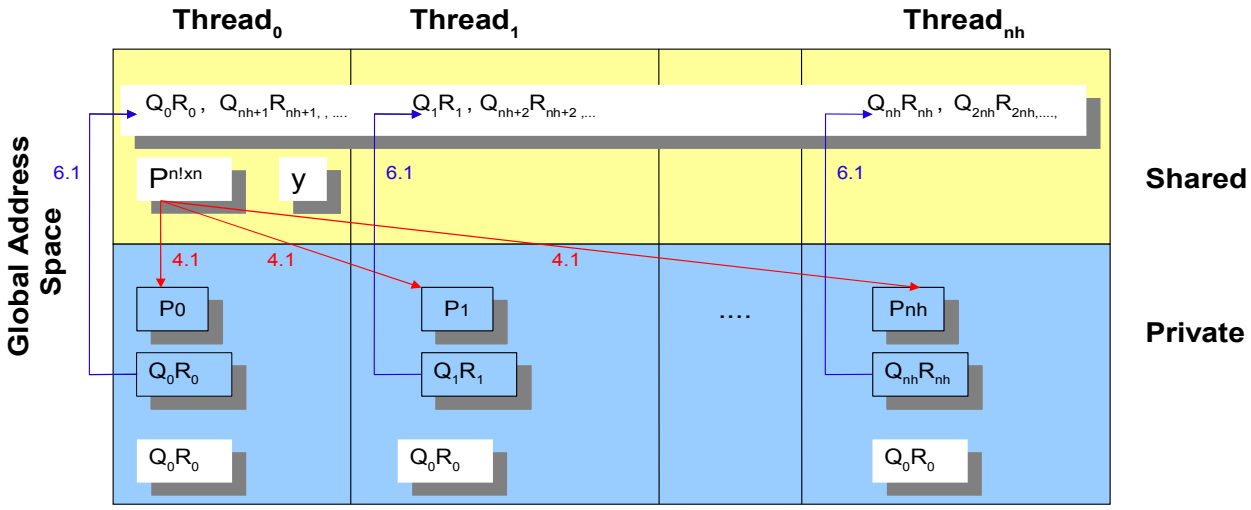


Figura 4.11: Movimientos entre memorias en la fase de preproceso

Ahora supongamos que el índice de la permutación del gradiente calculado es 1. Entonces el Hilo 1 es el encargado de realizar la decodificación ya que es el que tiene afinidad con $Q_1 R_1$. Para ello se hace una copia de la señal *y* a su memoria privada. En la figura 4.12 puede verse como también se hace una copia de la factorización $Q_1 R_1$, aunque el hilo 1 tenga afinidad siempre son más rápidos los accesos a memoria privada que a la compartida, a este proceso se le llama privatización.

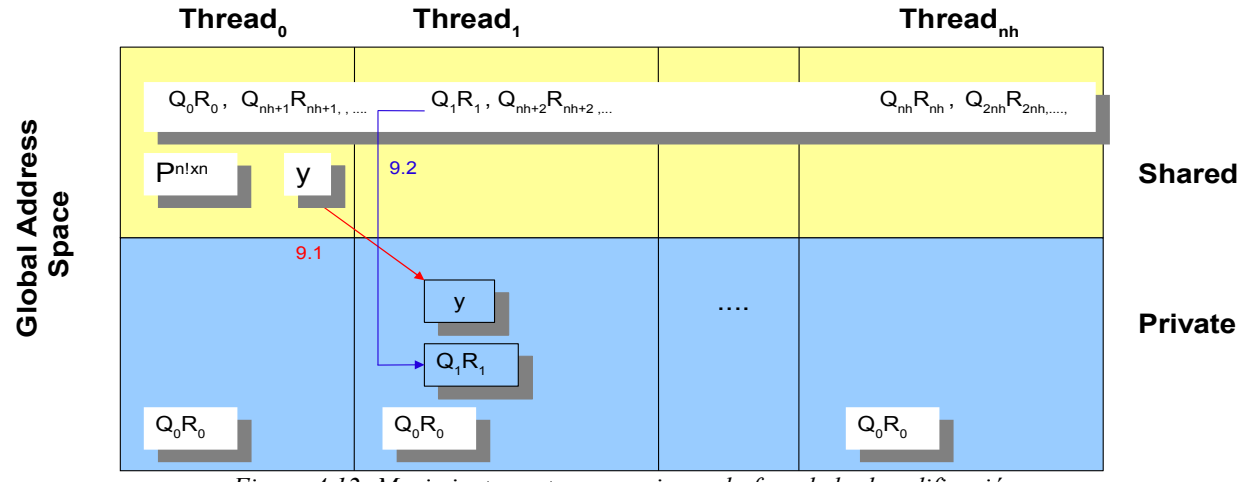


Figura 4.12: Movimientos entre memorias en la fase de la decodificación

4.5 Resultados

Los algoritmos paralelos han sido implementados y evaluados en las máquinas paralelas con memoria compartida, y con GPUs de NVIDIA Golub y Locomotora vistas en el apartado 2.4. Los experimentos se han realizado con matrices aleatorias de tamaños 4x4, 6x6, 8x8 y 10x10.

Estos experimentos tienen dos objetivos:

- Evaluar los costes de la parte de preproceso en memoria compartida y con tarjetas gráficas.
- Evaluar la paralelización en UPC de la decodificación de señales, si suponemos que el sistema MIMO utiliza un buffer de señales.

4.5.1 Evaluación de la fase de Preproceso

La tabla 4.3 muestra los tiempos de ejecución y Speed-up de los algoritmos en memoria compartida con UPC y openMP y el kernel implementado en los Métodos II y III en CUDA (*calculaQR)* frente al algoritmo secuencial en el cual se calculan las n! descomposiciones con un solo procesador. Únicamente se han tomado tiempos de la parte de preproceso, sin tener en cuenta el tiempo de la estimación de la señal, centrándonos en el objetivo de este trabajo.

N=M		UPC				OMP				CUDA (*calculaQR*)			
	Secuencial	UPC 4 hilos		UPC 8 hilos		OMP 4 hilos		OMP 8 hilos		T. Ej			
	T.ej	T.ej	Sp	T.ej	Sp	T.ej	Sp	T.ej	Sp	Envio	Calculo	Total	Sp
4	1,00E-05	3,00E-05	0,33	4,00E-05	0,25	4,96E-03	0,002	6,41E-02	0,0002	5,20E-05	1,02E-04	1,54E-04	0,06
6	6,90E-04	1,60E-04	4,31	1,00E-04	6,9	8,63E-03	0,08	1,23E-01	0,01	7,40E-05	3,65E-04	4,39E-04	1,57
8	5,45E-02	1,70E-02	3,2	7,81E-03	6,97	2,97E-02	1,83	1,38E-01	0,39	1,78E-03	1,32E-02	1,49E-02	3,65
10	5,66E+00	1,42E+00	3,99	1,14E+00	4,96	1,66E+00	3,41	1,33E+00	4,26	2,24E-01	2,64E+00	2,86E+00	1,98

Tabla 4.3 Tiempos de ejecución (en segundos) en Locomotora con hyperthreading activado y Speed-up correspondientes a la parte de preproceso de los algoritmos desarrollados en OpenMP , UPC y CUDA.

Como puede observarse en la figura 4.13 el algoritmo desarrollado con UPC ofrece mayores prestaciones que OMP , esto es debido al control que posee UPC sobre la localidad de la memoria y los accesos de los hilos a los datos son menos costosos. Los mejores resultados los obtenemos cuando utilizamos 8 hilos con UPC, aunque el Speed-up solo crece hasta un tamaño de matriz de 8x8.

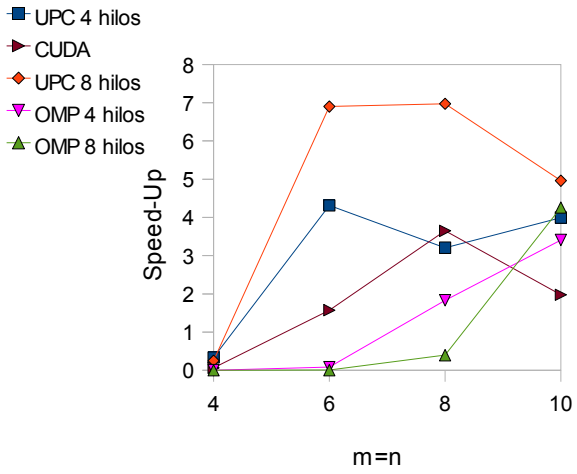


Figura 4.13 . Speed-up de los algoritmos desarrollados en OpenMp y Unified Parallel C

4.5.2 Evaluación de la fase de de Decodificación

Para evaluar la paralelización en UPC de la fase de decodificación de las señales, se han tomado tiempos simulando que el sistema MIMO recibe de 100.000 señales y con varios tamaños (k) de buffer . La tabla 4.4 muestra el Speed-Up conseguido en la decodificación de señales utilizando el esquema de paralelización visto en la figura 4.10.

M=N	k=64	Sp	k=128	Sp	k=256	Sp	k=512	Sp	k sin límite	Sp	Secuencial
4	0,13	2,8	0,23	1,58	0,19	1,92	0,09	4,04	0,12	3,02	0,36
6	1,17	1,39	0,41	3,98	0,39	4,18	0,57	2,86	0,36	4,57	1,63
8	3,06	1,8	2,84	1,94	1,23	4,48	1,09	5,06	1,02	5,39	5,51
10	4,42	7,81	4,03	8,57	3,3	10,47	3,24	10,66	2,98	11,6	34,54

Tabla 4.4 Tiempos de ejecución (en segundos) para la decodificación de 100000 señales con distintos tamaños de buffer utilizando en algoritmo 4.6 modificado.

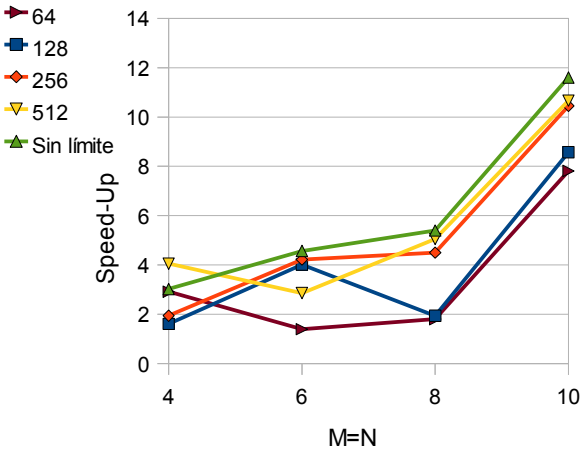


Figura 4.13 . Speed-up para la decodificación de señales con distintos tamaños de buffers utilizando UPC.

Como vemos el Speed-up aumenta cuanto mayor es el tamaño del buffer, esto sucede porque cada vez que un buffer de señales es consumido, los hilos deben sincronizarse y además tiene que recalcularse la matriz de gradientes realizando la multiplicación H^TB. En el caso ideal de que tuviéramos un buffer enorme en el que cupieran todas las señales a decodificar, los hilos tendrían siempre señales que tratar sin tener que esperar la siguiente tanda ni quedarse ociosos.

4.5.3 Conclusiones

En este problema en concreto, el entorno UPC ofrece mejores resultados, tanto en la fase de preproceso, donde se calculan las descomposiciones, como también en la decodificación de la señal donde conseguimos una eficiencia muy buena, principalmente gracias a la gestión de la memoria que podemos conseguir utilizando UPC.
Por el contrario, el entorno de CUDA no consigue tantas prestaciones, principalmente porque todos los hilos trabajan sobre la memoria global de la GPU y resulta complicado adaptar el algoritmo para que haga un uso eficiente de la memoria compartida.
La aproximación que hemos planteado con el uso de CULA, no es aconsejable para este problema,

ya que la matrices son tan pequeñas que la función implementada en esta librería no realiza ningún tipo de cálculo sobre la GPU, si no que simplemente se limita a realizar una llamada a LAPACK.
Por lo tanto la solución con UPC es, en global, la que mejores prestaciones ofrece. La evaluación de este algoritmo nos permite decir que es viable utilizar un sistema MIMO basado en la Aproximación 2, puesto que el tiempo de puesta en marcha del sistema es muy pequeño. Si seguimos esta aproximación, evitamos recalcular en tiempo real la descomposición QR para cada nueva señal y además obtendremos una mejora considerable en la fase de decodificación de la señal.

Capítulo 5
Aplicación de la QR en otros problemas de mínimos cuadrados

5.1 Descripción del problema

El término factorización QR generalizada (generalized QR factorization) GQR, usado por Hammarling [28] y Paige [29], corresponde a una factorización ortogonal que simultáneamente transforma una matriz A, *mxn,* y una B, *mxp,* a la forma triangular. Esta descomposición corresponde a la factorización de $B^{-1}A$ cuando B es cuadrada y no singular. Por ejemplo, si $m \geqslant n \, y \, m \leqslant p$ entonces la factorización GQR de A y B es de la forma

$$Q^T A = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \quad Q^T BV = [0 \quad S],$$

donde Q, *mxm,* es una matriz ortogonal, V, *pxp,* es una matriz ortogonal, R_1, nxn, es una matriz triangular superior, y S, pxp, es una matriz triangular superior.

Al igual que la factorización QR es una herramienta muy potente para la resolución de problemas de mínimos cuadrados y relacionados con problemas de regresión lineal, de la misma manera la factorización GQR se puede usar para resolver distintos problemas de mínimos cuadrados como:

- Problema de mínimos cuadrados con restricciones de igualdad $\min_{Bx=d} \|Ax-b\|_2$
- Problema de mínimos cuadrados generalizados $\min_{x,u} u^T u \quad sujeto\,a \quad b = Ax + Bu$
- Problema de mínimos cuadrados ponderados $\min_{x} \|B^{-1}(Ax-b)\|_2$
- Modelos de ecuaciones simultáneas $\min_{\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}} \left\| [X \; Y] \begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix} - Y \right\|_2$

En este capítulo abordamos algunos de ellos, en particular, el problema de mínimos cuadrados generalizados, ponderados y resolución de modelos de ecuaciones simultáneas.

5.2 Factorización QR generalizada

Siendo A una matriz *mxn*, B una matriz *mxp*, y suponiendo que $m \geqslant n$, entonces existen una matrices Q (m*xm*) y V (*pxp*) tales que

$$Q^T A = R, \quad Q^T BV = S,$$

donde

$$R=\begin{bmatrix} R_{11} \\ 0 \\ {\scriptstyle n} \end{bmatrix}_{m-n}^{n},$$

con R_{11} (nxn) triangular superior, y

$$S=\underset{p-m}{\begin{bmatrix} 0 & S_{11} \end{bmatrix}}_{m} \quad si\ m \leqslant p$$

donde la matrix S_{11} *mxm* es triangular superior, o

$$S=\begin{bmatrix} S_{11} \\ S_{21} \\ {\scriptstyle p} \end{bmatrix}_{p}^{m-p}, \quad si\ m>p,$$

donde la matriz S_{21} es triangular superior.

La prueba es fácil y constructiva. Se puede ver en [31].
Dada factorización QR de A tenemos

$$Q^T A=\begin{bmatrix} R_{11} \\ 0 \\ {\scriptstyle n} \end{bmatrix}_{m-n}^{n}$$

basta multiplicar Q^TB, y construir una matriz ortogonal V que triangularice (Q^TB) al postmultiplicarla (Descomposición RQ)

La factorización RQ de Q^TB tiene la forma

$$(Q^T B)\,V=\begin{bmatrix} S_{11} \\ S_{21} \\ {\scriptstyle p} \end{bmatrix}_{p}^{m-p} \quad \text{si} \quad m \leqslant p$$

$$(Q^T B)\,V=\underset{p-m}{\begin{bmatrix} 0 & S_{11} \end{bmatrix}}_{m} \quad \text{si} \quad m>p$$

Algoritmo 5.1 (GQR) Algoritmo que calcula la factorización GQR

Entrada : $A \in R^{mxn}\ y\ B \in R^{mxp}$
Salida: $Q \in R^{mxm},\ R \in R^{mxn}\ y\ S \in R^{mxp}$
1: $R = Q^TA$
2: $B_1 \leftarrow Q^TB$
3: $S = B_1V$

5.3 Problemas de mínimos cuadrados ponderados

En ocasiones, se necesita calcular la factorización QR de $V^T(B^{-1}A)$, por ejemplo, para resolver el problema de mínimos cuadrados ponderados

$$\min_{x} \left|\left| B^{-1}(Ax-b) \right|\right|,$$

Para evitar la formación de B^{-1} y $B^{-1}A$, la factorización GQR de A y B implícitamente proporciona la factorización QR de $B^{-1}A$. En efecto, si se calculan V y Q ortogonales tales que A=QR y $Q^TBV=S$ se tiene

$$V^T(B^{-1}A)=\begin{bmatrix}T\\0\end{bmatrix}=S^{-1}\begin{bmatrix}R_{11}\\0\end{bmatrix},$$

La parte triangular superior T de la factorización QR de $B^{-1}A$ se puede calcular resolviendo el sistema triangular en T,

$$S_{11}T=R_{11}$$

donde S_{11} es la parte superior izquierda, de tamaño *mxm*, de la matriz S. De esta manera, las dificultades numéricas que conlleva usar implícitamente o explícitamente, la factorización QR de $B^{-1}A$ se resumen en el condicionamiento de S_{11} .

$$\min_x \left\| B^{-1}(Ax-b)\right\|_2=\min_x \left\| V^T B^{-1}(Ax-b)\right\|_2=\min_x \left\| \begin{bmatrix}T\\0\end{bmatrix}x-S^{-1}Q^Tb\right\|_2$$

Algoritmo 5.2 : (PMC) Algoritmo que resuelve el problema de mínimos cuadrados ponderado.

Entrada: $A\in R^{mxn}$, $B\in R^{mxp}$ y $b\in R^n$

Salida: $x\in R^n$ que cumple $\min_x \left\| B^{-1}(Ax-b)\right\|$

1: [Q,R,S] ← GQR (A,B)
2: c ← Q^Tb
3: **Resolver** Sz = c
4: **Resolver** y=$S_{11}z_1$
5: **Resolver** R_1x = y

5.4 Problemas de mínimos cuadrados generalizado

En el problema de mínimos cuadrados ponderados se exige la invertibilidad de la matriz B. Se puede resolver un problema equivalente cuando B no sea invertible. Este es el caso de los estimadores de *Mínimos cuadrados generalizados* (MCG). La estimación de *x es la* solución al siguiente problema de mínimos cuadrados

$$\min_{x,u} u^T u \quad sujeto\,a \quad b=Ax+Bu,$$

Por conveniencia supondremos que $m\geqslant n, m\geqslant p$ que es el caso que más frecuentemente ocurre. Cuando B = I , es justo el problema lineal de mínimos cuadrados ordinario, y si B es invertible se transforma en el problema de mínimos cuadrados ponderados.

Utilizando la factorización GQR tenemos que

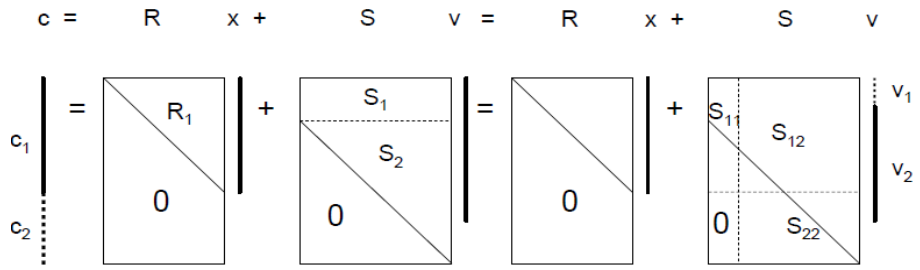
$$Q^T A = R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}_{m-n}^{n}, \qquad Q^T B V = S = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix}_{m-p}^{p}$$

Si llamamos $c = Q^T b, v = V^T u$ entonces tenemos que $Q^T b = Q^T Ax + Q^T BVV^T u \rightarrow c = Rx + Sv$

y $\min_{b=Ax+Bu} u^T u = \min_{c=Rx+Sv} (Vv)^T (Vv) = \min_{c=Rx+Sv} v^T v$

Podemos encontrar dos casos en función del tamaño de p,

En el caso de que $p \geqslant m-n$



$$c = Rx + Sv \rightarrow \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} s + \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \quad (1)$$

donde

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_{m-n}^{p-m+n} \quad S = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} S_{11} & S_{22} \\ 0 & S_{22} \end{bmatrix}_{p-m+n \;\; m-n}^{n}$$

Entonces de la ecuación superior de (1), se tiene

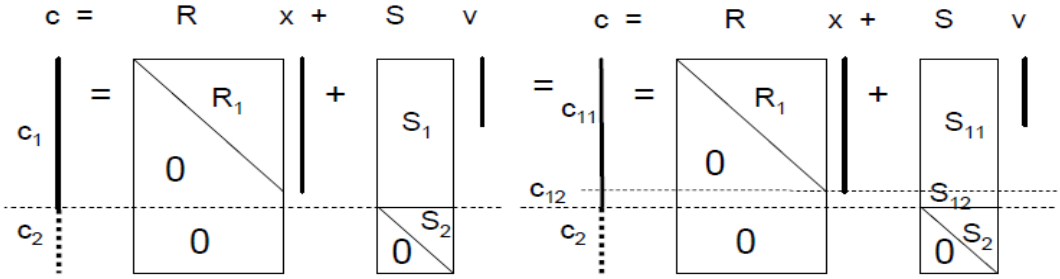
$$c_1 = R_1 x + S_{11} v_1 + S_{12} v_2$$

y de la ecuación inferior de (1) podemos calcular v_2 resolviendo un sistema triangular

$S_{22} v_2 = c_2$ (p.4 Algoritmo 5.3)

La solución al problema es $x = R_1^{-1}(c_1 - S_{12} v_2)$ (p.5-6 Algoritmo 5.3)

Si por el contrario p < m-n, el método de resolución cambia. En este caso



$$c = Rx + Sv \rightarrow c = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} s + \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} v \qquad (2)$$

y

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} R_1' \\ 0 \end{bmatrix} x + \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} v \qquad con\ R_1' = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}_{m-n-p}^{n} \ y\ S = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix}_{p}^{m-p}$$

Entonces de la ecuación inferior de (2), podemos calcular *v* resolviendo un sistema triangular

$S_2 v = c_2$ (p.9 Algoritmo 5.3)

La ecuación superior de (2) podemos expresarla como

$$c_1 = \begin{bmatrix} c_{11} \\ c_{12} \end{bmatrix} = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x + S_1 v = \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x + \begin{bmatrix} S_{11} \\ S_{12} \end{bmatrix} v$$

y por tanto

$c_{11} = R_1 x + S_{11} v$ (p.10-11 Algoritmo 5.3)

La solución al problema es $x = R_1^{-1}(c_{11} - S_{11} v)$

El algoritmo 5.3 resuelve el problema de mínimos cuadrados generalizado

---

Algoritmo 5.3 : (GMC) Algoritmo que resuelve el problema de mínimos cuadrados generalizado.

Entrada: $A \in \mathbb{R}^{mxn}, m \geq n, B \in \mathbb{R}^{mxp}, m \geq p\ y\ b \in R^n$

Salida: $x \in R^n$ que cumple $\min_{x,u} u^T u \quad sujeto\ a \quad b = Ax + Bu,$

1: [Q,R,S] ← GQR(A,B)
2: c ← Q^Tb
3: **si** $p \geq m - n$ **entonces**
4: **Resolver** $S_{22}v_2 = c_2$
5: y ← $S_{12}v_2$
6: **Resolver** $R_1x = c_1$-y
7: **finsi**
8: **si** $p<m-n$ **entonces**
9: **Resolver** $S_2v = c_2$
10: y ← $S_{11}v$
11: **Resolver** $R_1x = c_{11}$-y
12: **finsi**

5.5 Resolución de Modelos de Ecuaciones Simultáneas

Los Modelos de Ecuaciones Simultáneas (SEM) son una técnica estadística utilizada tradicionalmente en el mundo de la economía. Sin embargo, hoy por hoy es usada en diferentes campos como son la econometría [32], medicina [33], en el estudio del ratio de divorcios, etc. SEM puede ser resuelta por una gran variedad de métodos como son Mínimos Cuadrados Indirectos (MCI), Mínimos cuadrados en Dos Etapas (MC2E), Mínimos Cuadrados en tres Etapas (MC3E), etc. El método MC2E es uno de los estimadores más usados en los modelos de ecuaciones simultáneas, y es el que se estudia en este apartado.

5.5.1 Modelo de Ecuaciones Simultáneas

Consideremos N variables vectoriales independientes (variables endógenas) que dependen de K variables vectoriales independientes (variables exógenas). Suponemos que estas variables pueden ser expresadas como una combinación lineal de las variables endógenas, las variables exógenas, y un ruido que representa la interferencia estocástica.
Entonces, un sistema de N ecuaciones lineales donde las N variables endógenas son expresadas linealmente como una función de otras variables endógenas, exógenas y un ruido [34],toma la forma:

$$\begin{aligned} y_1 &= B_{1,2}\,y_2 + B_{1,3}\,y_3 + \ldots + B_{1,N}\,y_N + \Gamma_{1,1}\,X_1 + \ldots + \Gamma_{1,K}\,X_K + u_1 \\ y_2 &= B_{2,2}\,y_2 + B_{2,3}\,y_3 + \ldots + B_{2,N}\,y_N + \Gamma_{2,1}\,X_1 + \ldots + \Gamma_{2,K}\,X_K + u_2 \\ y_N &= B_{N,2}\,y_2 + B_{N,3}\,y_3 + \ldots + B_{N,N-1}\,y_N - 1 + \Gamma_{N,1}\,X_1 + \ldots + \Gamma_{N,K}\,X_K + u_N \end{aligned} \qquad (1)$$

donde $x_1,x_2,...,x_K$ son variables exógenas, $y_1,y_2,...,y_N$ son variables endógenas, y $u_1,u_2,...,u_N$ son variables aleatorias. Todas estas variables son vectores con una dimensión dx1, donde d es el tamaño de la muestra.

La ecuación (1) puede ser representada de forma matricial como:

$$B\,Y^T + \Gamma\,X^T + u^T = 0 \quad (2)$$

donde

$$Y = (y_{1,}\,y_{2,\ldots},\,y_N),\; X = (x_{1,}\,x_{2,\ldots},\,x_K),\; u = (u_{1,}\,u_{2,\ldots},\,u_N)$$

$$B = \begin{pmatrix} \beta_{1,1} & \cdots & \beta_{1,N} \\ & \cdots & \\ \beta_{N,1} & \cdots & \beta_{N,N} \end{pmatrix},\quad \Gamma = \begin{pmatrix} \gamma_{1,1} & \cdots & \gamma_{1,K} \\ & \cdots & \\ \gamma_{N,1} & \cdots & \gamma_{N,K} \end{pmatrix}$$

siendo $\beta_{i,i} = -1\;\forall\, i = 1,..,N.$ El objetivo es estimar $\gamma_{(1,1)},\ldots,\gamma_{(N,K)},\beta_{(1,2)},\beta_{(1,3)},\ldots,\beta_{(N,N-1)}$ a partir de datos conocidos experimentalmente en cada una de las variables x e y.

Es decir, se dispone de d datos de las K variables exógenas $x_1,x_2,...,x_K$ y d datos de las N variables endógenas $y_1,y_2,...,y_N$ de un modelo de ecuaciones simultáneas, y lo que se pretende es estimar a partir de estos datos, los coeficientes del modelo que las relaciona. Estos coeficientes son las componentes de las matrices $B\ y\ \Gamma$.

5.5.2 Mínimos cuadrados en dos etapas

Habitualmente existe un problema de correlación entre los valores de las variables *Y* y el ruido *u*. Para evitar este problema el estimador MC2E obtiene un conjunto de variables, llamadas proxy $(denotadas\ por\ \hat{Y})$, que están próximas a las variables endógenas. Para el cálculo de las variables proxy se aproxima B=-I_N *(*suponiendo que en la primera etapa solo dependen de las variables exógenas) *y* se resuelve el problema de mínimos cuadrados $\min_{\Gamma^T} \|X\Gamma^T - Y\|_{2,}$ de este modo se obtiene $\hat{\Gamma}^T$. Las variables proxy pueden estimarse por la expresión $\hat{Y} = X\hat{\Gamma}^T$.
Las condiciones físicas de los problemas donde se suelen aplicar los MES imponen, a veces, que las variables endógenas no dependan de ciertas variables endógenas o exógenas. Esto se traduce en que aparecen ceros, conocidos de antemano, en las matrices $B\ y\ \Gamma$. Definimos una matriz asociada a la i-ésima ecuación, denotada por $Z_i = [X_i | \hat{Y}_i]$, donde X_i es una matriz formada por las columnas de X que corresponden a la variables exógenas cuyos coeficientes no son nulos en la i-ésima ecuación, y $\hat{Y}_i$ es una matriz formada por las columnas de $\hat{Y}$ que corresponden a la variables exógenas cuyos coeficiente no son cero en la i-ésima ecuación excepto las variables endógenas principales. Las condiciones estadísticas del problema obligan a que Z_i sea una matriz con mayor o igual número de filas que de columnas.

Para resolver la ecuación i-ésima , $y_i = X_i b_i + Y_i \Gamma_i + \varepsilon_i$, siendo y_i la variable endógena principal de la ecuación), primero se sustituyen las variables endógenas que hay en dicha ecuación (salvo la principal) por las variables proxys estimadas $(Y_i\ por\ \hat{Y}_i)$, entonces $B_i\, y\, \Gamma_i$ pueden ser estimadas resolviendo el problema de mínimos cuadrados $\min_{\eta_i} \|y_i - Z_i \eta_i\|_2$, donde y_i es la variable endógena principal y $\eta_i^T = [\Gamma_{i,:} | B_{i,:}] \in R^{k_i + n_i - 1}$.

En definitiva, se trata de resolver el problema

$$Y = X\Gamma^T + YB^T + u \qquad \min_{\substack{\Gamma^T \\ B^T}} \left\| [X \;\; Y] \begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix} - Y \right\|_2$$

Suponemos que en la primera etapa Y solo dependen de X (variables exógenas)

$$Y = X\Gamma^T + u \qquad \min_{\hat{\Gamma}^T} \|X\Gamma^T - Y\|_2$$

para resolver este problema de mínimos cuadrados realizamos la factorización QR de X de forma que

$$X = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \qquad Y_1 = Q^T Y = \begin{bmatrix} Y_{11} \\ Y_{22} \end{bmatrix} \qquad \hat{\Gamma}^T = R_1^{-1} Y_{11}$$

A continuación calculamos las variables proxys

$$\hat{Y} = X\hat{\Gamma}^T = X R_1^{-1} Y_{11} = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} R_1^{-1} Y_{11} = Q \begin{bmatrix} Y_{11} \\ 0 \end{bmatrix}$$

Planteamos de nuevo el problema como

$$Y = X\Gamma^T + \hat{Y} B^T + u = [X \;\; \hat{Y}] \begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix} + u$$

Ahora para cada columna i de Y, formamos Z_i haciendo uso de una matriz de selección S_i

$$Zi=\begin{bmatrix} X & \hat{Y} \end{bmatrix} S_i = \left[Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \; Q \begin{bmatrix} Y_{11} \\ 0 \end{bmatrix} \right] S_i = Q \begin{bmatrix} R_1 & \hat{Y}_{11} \\ 0 & 0 \end{bmatrix} S_i$$

y resolvemos el siguiente problema de minimos cuadrados

$$\min_{\beta_i} \left\| Q \left[\bar{R}_1 | \bar{Y}_{11} \right] \beta_i - (Y_1)_i \right\|$$

El algoritmo 5.4 muestra un esquema para el estimador MC2E. Como se observa se aplican dos veces Mínimos Cuadrados Ordinarios para resolver cada una de las ecuaciones del sistema, una en el cálculo de las variables proxy , y la otra en la resolución de cada una de las ecuaciones.

Algoritmo 5.4 : (MES) Algoritmo que resuelve Modelos de Ecuaciones Simultáneas

---

Entrada: $X \in \mathbb{R}^{dxk}, Y \in \mathbb{R}^{dxN}, seleccion \in Ce^{N \; x \; k+n} \; Ce = \{0,1\}$

Salida: $\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}$ tal que $\min_{\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}} \left\| \begin{bmatrix} X \; Y \end{bmatrix} \begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix} - Y \right\|_2$

1: [Q,R] ← QR(X) , $R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$

2: $Y_1 = Q^T Y = \begin{bmatrix} Y_{11} \\ Y_{21} \end{bmatrix} \begin{matrix} k \\ d-k \end{matrix}$
N

3: para i = 1:N **hacer**

4: $Z_i = selección(R_1, Y_{11}) \in R^{k \; x \; ri}, \quad 1 \leqslant ri \leqslant k$

5: [H, U] = QR(Z_i)

6: $c_i = H^T (Y_1)_i \in R^{k \; x 1}$

7: Resolver $U \beta_i = c_i$

8: Sustituir $\beta_i \rightarrow \Gamma^T, B^T$

9: fin para

5.6 Implementación en CUDA

Se muestra ahora detalles de implementación de los algoritmos GQR, PMC, GMC y MES en CUDA y utilizando la librería CULA.

5.6.1 Implementación de la factorización QR generalizada

La forma en la que se ha implementado la factorización QR generalizada sobre la GPU se puede

observar en la figura 5.1. Como vemos se hace uso del algoritmo 3.10 *qr_span,* para ello la matriz B y se copian tras A, y las transformaciones ortogonales que van calculando para triangularizar A se aplican tanto a A como a la matriz B. Una vez calculada $B_1 = Q^T B$, calculamos la traspuesta de B_1 y se calcula su descomposición RQ usando la función implementada en CULA *culaDeviceSgeqrf.*

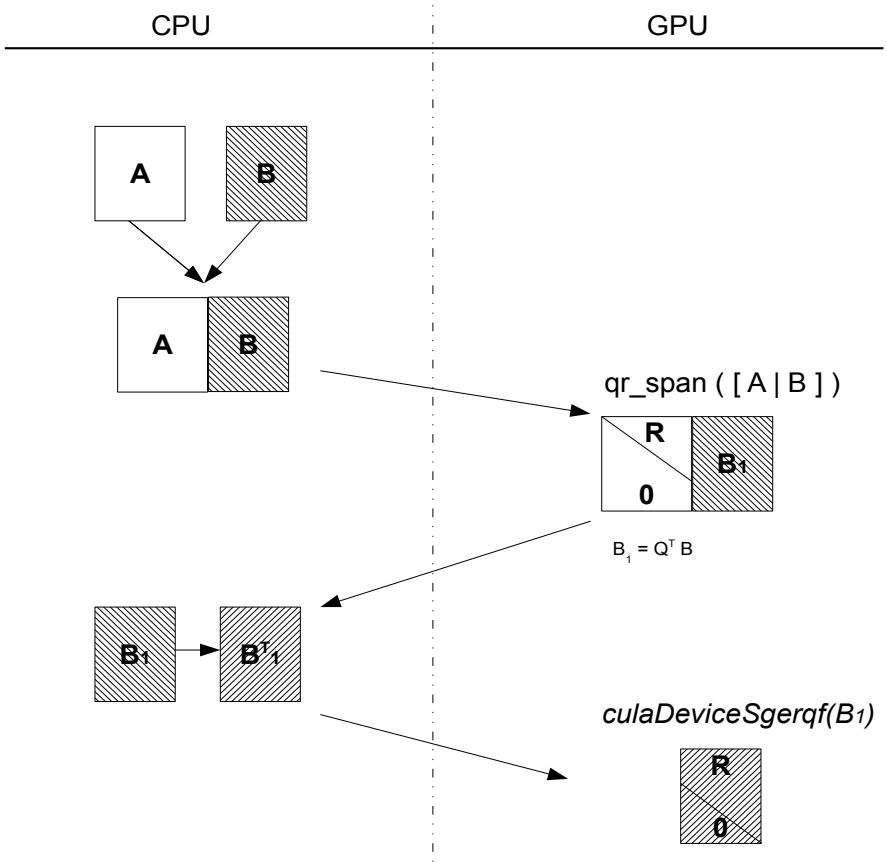


Figura 5.1. Esquema del cálculo de la Factorización QR Generalizada

5.6.2 Implementación del problema de mínimos cuadrados ponderados y generalizados

La implementación de los problemas de mínimos cuadrados generalizados y ponderados se ha basado en el uso de la función QR generalizada, pero en este caso, el vector de términos independientes *b* también se añade a A, de manera que calculamos la QR generalizada sobre la matriz [A | B | b] . Ahora, las rotaciones calculadas para descomponer la matriz A , se aplican a B y b, tal y como se muestra en la Figura 5.2.

La resolución de los sistemas de ecuaciones, multiplicaciones matriz vector y suma de vectores utilizados en los algoritmos 5.2 y 5.3 se han resuelto llamando a las siguientes funciones implementadas en la librería CUBLAS[16]:

- ***cublasStrsv:*** Resuelve un sistemas de ecuaciones (*Ax = b)*
- ***cublasSgemv*****:** Multiplica una matriz por un vector ($y \leftarrow \alpha A + \beta y$)
- **cublasSaxpy:** Multiplica un vector *x* por un escalar α y suma el resultado a otro vector *y.* ($y \leftarrow x\alpha + y$)

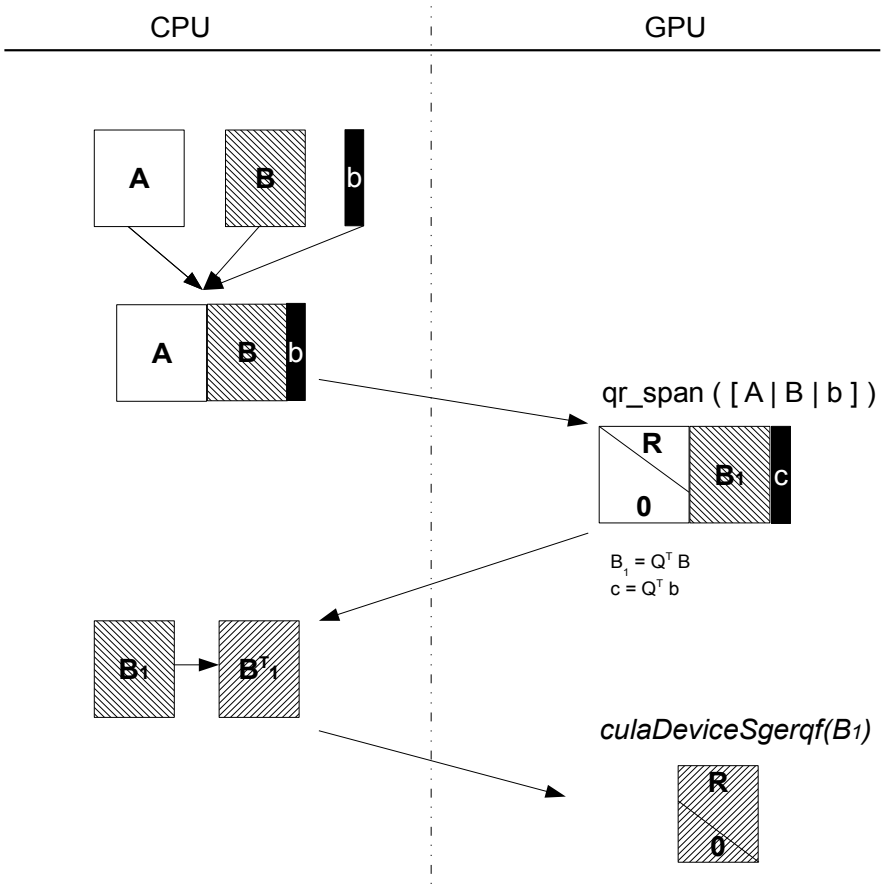


Figura 5.2. Esquema del cálculo de la Factorización QR Generalizada

5.6.3 Implementación del problema de Modelos de ecuaciones simultáneas

Los pasos 1,2 del algoritmo 5.4 se han realizado con una llamada a la rutina qr_span, presentada en el capítulo 3.

En el capítulo 3 presentábamos la descomposición QR mediante rotaciones de Givens, pero no teníamos en cuenta la estructura de la matriz, suponíamos que casi todos sus elementos eran distintos de cero, ya que no sabíamos de ante mano si la matriz era dispersa o densa. Sin embargo en este problema, conocemos que la estructura de la matriz Z_i es casi triangular, ya que esta formada en parte por columnas de R_1 que es triangular superior. Para beneficiarnos de esta estructura, mientras se forma la matriz Z_i vamos almacenando los índices de los elementos que hay que anular, y se añaden a una lista que los almacena por diagonales.

Por ejemplo, si $R_1 \in R^{6x6}$ y $Y_{11} \in R^{6x5}$:

$$R_1 = \begin{bmatrix} r & r & r & r & r & r \\ 0 & r & r & r & r & r \\ 0 & 0 & r & r & r & r \\ 0 & 0 & 0 & r & r & r \\ 0 & 0 & 0 & 0 & r & r \\ 0 & 0 & 0 & 0 & 0 & r \end{bmatrix} \qquad Y_{11} = \begin{bmatrix} y & y & y & y & y \\ y & y & y & y & y \\ y & y & y & y & y \\ y & y & y & y & y \\ y & y & y & y & y \\ y & y & y & y & y \end{bmatrix}$$

y seleccionamos las columnas 1,3,5 de R_1 y dos columnas cualquiera de Y_{11}, la matriz Z_i tiene la

siguiente estructura:

$$Z_i = \begin{bmatrix} r & r & r & y & y \\ r & r & r & y & y \\ 0 & r & r & y & y \\ 0 & r & r & y & y \\ 0 & 0 & r & y & y \\ 0 & 0 & r & y & y \end{bmatrix}$$

Ahora si distribuimos los elementos a anular por diagonales vemos como hay una diagonal (2) en la que no hay que hacer ningún cero, y otras en las que el número de elementos a anular es inferior al número de elementos que habría que anular si la matriz fuera completa. De manera que mientras se va construyendo la matriz Z_i se puede llevar la cuenta de cuantos elementos habrá en cada una de las diagonales y los índices de los elementos a anular.

$$Z_i = \begin{bmatrix} r & r & r & y & y \\ 1 & r & r & y & y \\ 0 & 3 & r & y & y \\ 0 & 4 & 5 & y & y \\ 0 & 0 & 6 & 7 & 8 \\ 0 & 0 & 7 & 8 & 9 \end{bmatrix}$$

índices = {(1,0) (2,1) (3,1) (3,2) (4,2) (4,3) (5,2) (4,4) (5,3) (5,4)}
d = {1,0,1, 1, 1,1,2,2,1} guarda el número de elementos a anular de la diagonal

La ventaja de este proceso es que ahora se realiza una fase menos en el proceso de triangularización. Por otra parte, solo crearíamos los bloques de hilos necesarios para anular los índices anotados y no para todos de índices de las diagonales. En el algoritmo 5.5 se muestra como únicamente se procesan aquellas diagonales que tengan algún elemento que anular.

Algoritmo 5.5 : (Dqr_span) Algoritmo que calcula la Factorización QR por diagonales en la GPU adaptado para el Modelo de Ecuaciones Simultáneas

---

Entrada: $A \in R^{mxn}$

Salida: $Q \in R^{mxm}$ *y* $R \in R^{mxn}$

1: Para *i = 1:Nf* **hacer**
2: **si** d(i) != 0 **entonces**
3: copiar d(i)-indices a memoria constante de la GPU
4: calcularRotaciones(A) *(calcula las rotaciones de la diagonal i en la GPU dimensión del grid* $(1 x \lceil \frac{d(i)}{bx} \rceil)$ *)*

5: aplicarRotaciones(A) *(aplica las rotaciones de la diagonal i en la GPU* $(d(i) x \lceil \frac{n}{bx} \rceil)$ *)*

6: **fin si**
7: fin para

El algoritmo 5.4 que resuelve el problema de Modelos de ecuaciones simultáneas utilizando las rutinas qr_span y dqr_span presentadas en esta tesis.

Algoritmo 5.5 : (DQR-MES) Algoritmo que resuelve Modelos de Ecuaciones Simultáneas usado las rutinas qr_span, dqr_span y CUBLAS.

Entrada: $X \in \mathbb{R}^{dxk}, Y \in \mathbb{R}^{dxN}, seleccion \in Ce^{N\ x\ k+n} Ce = \{0,1\}$

Salida: $\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}$ tal que $\min_{\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}} \left\| [X\ Y] \begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix} - Y \right\|_2$

1: [R,Y_1] = qr_span([X | Y])
2: para i = 1:N **hacer**
3: $Z_i = selección(R_1, Y_{11}) \in R^{k\ x\ ri}, \quad 1 \leqslant ri \leqslant k$
4: [U,c_i] = dqr_span([Z_i | $(Y_1)_i$])
5: [β_i] = cublasStrsv(U,c_i)
6: Sustituir $\beta_i \rightarrow \Gamma^T, B^T$
7: fin para

También se ha desarrollado un algoritmo implementado íntegramente con llamadas a funciones de CULA y de CUBLAS. Como se puede observar el algoritmo 5.6 conlleva construir explícitamente las matrices ortogonales Q y H ; así como también el cálculo de los productos de estas matrices por Y_1 y C_i respectivamente.

Algoritmo 5.6: (CULA-MES) Algoritmo que resuelve Modelos de Ecuaciones Simultáneas utilizando CULA y CUBLAS

Entrada: $X \in \mathbb{R}^{dxk}, Y \in \mathbb{R}^{dxN}, seleccion \in Ce^{N\ x\ k+n} Ce = \{0,1\}$

Salida: $\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}$ tal que $\min_{\begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix}} \left\| [X\ Y] \begin{bmatrix} \Gamma^T \\ B^T \end{bmatrix} - Y \right\|_2$

1: [R] = culaDeviceSgeqrf(X)
2: [Q] = culaDeviceSorgqr(R)
3: [Y_1] = cublasSgemm(Q^T,Y)
4: para i = 1:N **hacer**
5: $Z_i = selección(R_1, Y_{11}) \in R^{k\ x\ ri}, \quad 1 \leqslant ri \leqslant k$
6: [U] = culaDeviceSgeqrf(Z_i)
7: [H] = culaDeviceSorgqr(U)
8: [c_i] = cublasSgemm(H^T,$(Y_1)_i$)
9: [β_i] = cublasStrsv(U,c_i)
10: Sustituir $\beta_i \rightarrow \Gamma^T, B^T$
11: fin para

5.7 Resultados

A continuación se evalúan los algoritmos implementados para la resolución de los problemas de mínimos cuadrados vistos en el apartado anterior. Estas pruebas se ha llevado a cabo en Golub, comparando los tiempos de ejecución con versiones secuenciales implementadas con el uso de la librería LAPACK.

5.7.1 Evaluación del problema de mínimos cuadrados generalizados

Para evaluar el algoritmo que resuelve el problema de mínimos cuadrados generalizados se ha comparado el tiempo de ejecución del algoritmo **GMC**, con M= 3000, P=9000 y variando el tamaño de N y un tamaño de bloque de hilos de 128, contra la función ya implementada en LAPACK que lo resuelve (sggglm).

N	GMC	LAPACK (SGGGLM)
64	9,179	18,42
256	9,432	14,21
512	9,959	15,68
1024	11,547	18,30
2048	16,625	22,40

Tabla 5.1 Tiempos de Ejecución en segundos del problema de mínimos cuadrados generalizados con M=3000,P=9000

5.7.2 Evaluación del problema de mínimos cuadrados ponderados

Para evaluar el algoritmo que resuelve el problema de mínimos cuadrados generalizados se ha comparado el tiempo de ejecución del algoritmo **PMC**, con M=P=6000 y variando el tamaño de N y un tamaño de bloque de hilos de 128, contra la función ya implementada en LAPACK que lo resuelve (sggglm).

N	PMC	LAPACK (SGGGLM)
64	18,886	30,92
256	19,348	24,25
512	20,429	26,39
1024	22,82	31,55
2048	29,54	39,97
4096	49,02	55,34

Tabla 5.2 Tiempos de Ejecución en segundos del problema de mínimos cuadrados ponderados para M=P=6000

5.7.3 Evaluación del problema de Modelos de Ecuaciones Simultáneas

Se ha realizado una versión para la resolución de Modelos de Ecuaciones Simultáneas con el uso de LAPACK, para tomar medidas y compararlo con las versiones implementadas en CUDA.
La tabla 5.3 muestra una comparativa entre los tiempos de ejecución de la resolución de un M.E.S usando el algoritmo **DQR-MES** que realiza las triangularizaciones con rotaciones de Givens, el algoritmo **CULA-MES** que realiza las triangularizaciones con llamadas al CULA con transformaciones de Householder y el algoritmo implementado con rutinas de LAPACK, que hace resuelve el problema con llamadas a las funciones de la librería en secuencial.

Se puede observar que el tiempo de resolución de los dos algoritmos presentados en esta tesis son menores para casi todos los casos en comparación con el algoritmo implementado con LAPACK.

N	K	D	Lapack	DQR-MES	CULA-MES
400	400	1000	6,330	12,666	10,340
400	400	1500	6,400	12,640	10,559
400	600	1000	15,029	20,571	14,928
400	600	1500	15,093	20,682	15,027
800	800	2000	84,180	75,439	45,812
800	800	2500	85,126	75,199	45,933
800	1000	2000	137,512	102,006	63,176
800	1000	2500	138,116	102,230	63,434
1000	1000	2500	197,350	140,272	84,506
1000	1200	3000	295,960	182,838	109,526
1200	1200	2500	395,570	239,294	138,759
1200	1200	3000	395,570	240,166	138,825

Tabla 5.3 Tiempos de ejecución (en segundos) en Golub para el algoritmo DQR-MES y CULA-MES y utilizando LAPACK, variando el número de variables endógenas (N), el número de variables exógenas (K) y el tamaño de la muestra (D).

5.7.4 Conclusiones

Como se puede observar en las tablas 5.1 y 5.2, los tiempos de ejecución con el uso de la librería LAPACK en problemas de mínimos cuadrados generalizados y ponderados es algo superior a los conseguidos por el algoritmo presentado en esta tesis. Si observamos los tiempos, se aprecia que no hay una diferencia excesiva entre los dos métodos, pero cabe recordar que LAPACK utiliza un algoritmo basado en transformaciones de Householder y el método que se presenta en esta tesis utiliza las rotaciones de Givens, con costes $2n^2(m-\frac{n}{3})$ y $3n^2(m-\frac{n}{3})$ (más el sobrecoste de trabajar con el formato compacto de las rotaciones) flops respectivamente[1].
Esta diferencia se vería aumentada con matrices estructuradas, con pocos elementos que anular, ya que una de las ventajas de Givens frente a Householder es que es capaz de hacer ceros selectivamente, en cambio Householder hace ceros todos los elementos de un vector sin considerar si algunas de sus componentes ya son nulas.

Respecto al problema de mínimos cuadrados en Modelos de ecuaciones simultáneas, vemos como se consiguen mejores tiempos con el uso de CULA y de los métodos vistos en esta tesis, sin embargo, pese a lo que cabía esperar el método que utiliza rotaciones de Givens no mejora el tiempo de la rutina de CULA con Householder; esto es lógico ya que la función culaDeviceSgeqrf utiliza un algoritmo híbrido que hace parte de la computación en la CPU mediante llamadas a LAPACK.

Capítulo 6
Conclusiones y Trabajos Futuros

En este último capítulo resumimos las principales conclusiones que se pueden extraer del trabajo realizado con el fin de tener una visión general de los resultados obtenidos.

6.1 Conclusiones

Como se ha visto en el capítulo de introducción, la factorización QR es un método muy eficiente para la resolución de múltiples problemas relacionados con la resolución de mínimos cuadrados. La mayoría de implementaciones que se pueden encontrar en las librerías más conocidas, obtienen la descomposición a través de reflexiones de Householder y no de Givens, puesto que el coste es el doble. Sin embargo, utilizar Givens es útil cuando la estructura de la matriz del problema es dispersa, por lo que conviene centrarse en hacer ceros sólo aquellos elementos no nulos.

En esta tesis hemos desarrollado distintos algoritmos para la resolución de problemas de mínimos cuadrados: mínimos cuadrados ordinarios, generalizados, ponderados, Modelos de Ecuaciones Simultáneas y mínimos cuadrados sobre conjuntos discretos con aplicaciones en sistemas MIMO, utilizando distintos entornos como UPC, OMP, CUDA y librerías como LAPACK y CULA.

Se ha desarrollado una versión paralela de la factorización QR por diagonales que obtiene unas prestaciones razonables frente a la versión secuencial; aun así el algoritmos implementado se presta a futuras optimizaciones variando el esquema de paralelización y poniendo más énfasis en su posible uso sobre matrices estructuradas.

Se han presentado varias soluciones para la aplicación de mínimos cuadrados en sistemas MIMO, comparando las prestaciones obtenidas con UPC, con OpenMP y CUDA. Se ha probado que con el uso de UPC es viable seguir el modelo de un sistema MIMO con preproceso con la finalidad de acelerar el proceso de decodificación.

Se ha desarrollado un algoritmo que resuelve problemas de mínimos cuadrados generalizados con el uso de aceleradores gráficos, no implementada todavía por la librería CULA, y se han obtenido mejores tiempos que con LAPACK.

Por último se han presentados dos algoritmos que resuelven Modelos de Ecuaciones Simultáneas, uno de ellos ha sido implementado con rutinas desarrolladas en esta tesis y el otro método con llamadas a librerías como CUBLAS y CULA. Los tiempos obtenidos para ambos son inferiores a los obtenidos con la versión en LAPACK implementada para evaluar los algoritmos.

Analizando las prestaciones obtenidas en las evaluaciones de los distintos algoritmos, se aprecia que UPC es una API más evolucionada que CUDA, puesto que ofrece mejores tiempos, mayor facilidad de aprendizaje y permite aprovechar de una manera más eficiente los recursos de un computador.
En cambio CUDA tiene más recorrido que UPC, ya que continuamente está en evolución. Sin embargo, no es fácil aprovechar toda la potencia que ofrecen las GPUs para obtener soluciones

óptimas porque resulta complicado aprovechar las ventajas de la jerarquía de memoria y acoplar su modelo SIMT a los algoritmos. No obstante, parte de estos problemas se solucionan con la nueva arquitectura Fermi, con nuevas jerarquías de memoria, y paralelismo en nivel de kernel, que no se ha tenido disponible para la elaboración de esta tesis.

6.2 Trabajos Futuros

■ Optimizar el código implementado en CUDA sobre las nuevas tarjetas Fermi recientemente adquiridas por el grupo. Tras analizar los resultados obtenidos en los algoritmos implementados en CUDA, se observa que en la mayoría de los casos, que los accesos a memoria son un factor crítico. Las nuevas tarjetas Fermi, añaden una memoria caché a la memoria global, lo que nos permitirá optimizar el código y los accesos de los hilos a la memoria.

■ Realizar un modelo de programación y de coste para CUDA con el fin de predecir resultados en fase de diseño. Uno de los problemas de los SIMD es que la fracción escalar en los programas es grande, por lo que resulta necesario un modelo que permita mejorar el algoritmo en fase de diseño.

■ Refinamiento de los algoritmos paralelos para la descomposición QR basada en rotaciones de Givens. Estudiar otras formas de aprovechar el paralelismo, por ejemplos, triangularizar por diagonales principales , por bloques, etc, y resolver los problemas de las divergencias de los hilos de un *warp* para evitar la secuencialización.

■ Modelo de Ecuaciones Simultáneas. Se tratará de aprovechar al máximo la estructura de la matriz Z_i, para que la descomposición QR necesaria para cada una de las ecuaciones se calculen lo más rápido posible. Estas mejoras irán en dos sentidos:

- Árbol de Mínimo Coste: Se trata de aplicar la teoría de árboles para obtener el orden óptimo en el que deben resolverse las ecuaciones, de manera que se pueda aprovechar la triangularización de una Z_i para triangularizar otra Z_j con menor coste.

- Mejorar el paralelismo en el algoritmo por diagonales implementado en esta tesis, de manera que puedan hacerse un número mayor de ceros en paralelo.

Por otra parte se estudiará la posibilidad de cambiar el paralelismo benefiándonos de las nueva característica de la tarjeta Fermi capaz de lanzar varios kernels simultáneamente, de esta forma se podría resolver varias ecuaciones en paralelo.
Parte de este trabajo, ya en marcha, está dando lugar a un artículo donde se describen las técnicas utilizadas: **"Resolución de Modelos de Ecuaciones Simultáneas sobre GPUs"**, D.Giménez, J.J. López, C.Ramiro y A.M Vidal. Está previsto enviar este artículo a la revista *Computational Statistics and Data Analysis.Ed. Elsevier.*

Bibliografía

[1] G.H. Golub and C.F. van Loan. *Matrix computations*. Johns Hopkins University Press, 1996.

[2] GPGPU. *General-Purpose Computation on Graphics Hardware*, http://gpgpu.org

[3] E. Anderson, Z. Bai, C. Bishof, Demmel J., and Dongarra J. *LAPACK User Guide*; Second edition. SIAM, 1995.

[4] S. Tomov, R. Nath, P. Du, J. Dongarra, *MAGMA version 0.2 Users' Guide*, November 2009. http://icl.cs.utk.edu/magma

[5] CULA, tools. *CULA Reference Manual* <www.culatools.com>

[6] R.Schreiber and C. van Loan. A storage-efficient WY representation for products of Householder transformations. SIAM J. Sci. Stat. Comput.,10(1): 53-57, 1989

[7] NVIDIA, *NVIDIA CUDA Programming Guide*, Version 2.0, 06/07/2008.

[8] Fermi Compute Architecture. www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[9] Procesadores Intel® para estaciones de trabajo http://www.intel.com/cd/products/services/emea/spa/workstation/processor/344494.htm

[10] NVIDIA GeForce http://www.nvidia.es/object/geforce_family_es.html

[11] Tesla C2050 / C2070 .http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html

[12] NVIDIA. http://www.nvidia.es/page/home.html

[13] S. Hammarling, J. Dongarra, J. Du Croz, and Richard J. Hanson. *An extended set of fortran basic linear algebra subroutines*. ACM Trans. Mathemathical Software, 1988

[14] L.S. Blackford, J. Choi, and A. Clearly. *ScaLAPACK User's Guide*. SIAM, 1997.

[15] Intel Corporation. Intel (R) Math Kernel Library. *Reference Manual*. 2004.

[16] CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf

[17] UPC Consortium, UPC Language Specifications, ,v1.2 2005 http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf

[18] D.E.Culler, A.C.Arpaci-Dusseau, S.C.Goldstein,A.Krishnamurthy,S. Lumetta, T.Eicken,

K.A.Yelick. Parrallel Programing in Split-C, Proceedings of Supercomputing 1993,p. 262-273.

[19] D. Micciancio and S. Goldwasser, Complexity of Lattice Problems, Kluwer Academic Publishers, Norwell, MA, USA, 2002.
[20] G. J Foschini, Layered space-time architecture for wireless communications in a fading environment when using multi-element antennas, Bell Labs Technical Journal, vol. 1, pp. 41-59, 1996.
[21] I. E. Telater, Capacity of multi-antenna gaussian channels, Europ. Trans. Telecommun., pp. 585-595, 1999.

[22] K. Su, Efficient Maximum-Likelihood detection for communication over Multiple Input Multiple Output channels, tech. report, Department of Engineering, University of Cambridge, 2005.
[23] X. Guo. Sphere Decoder for MIMO Systems. [Online]. Available: http://www.mathworks.de/matlabcentral/fileexchange/22890

[24] R.A Trujillo, V.M García, A.M Vidal, S.Roger y A.Gonzalez. A Gradient-Based ordering for MIMO Decoding.

[25] K. Su and I.J. Wassell, "A new Ordering for Efficient Sphere Decoding", in International Conference on Communications, Mayo 2005

[26] Caio Masakazu Kinoshita, Yusuke Sasaki,Tadashi Fujino.An Improved Lattice Reduction Aided Detection Based on Gram-Schmidt Procedure.Proc IEEE Advanced Technologies for Communications.Conf. Ho Chi Minh City, Viet Nam,Oct.2010 pp.83-88

[27] Hidekazu Negishi,Wei Hou,and Tadashi Fujino. An MMSE Detector Applying Reciprocal-Lattice Reduction in MIMO Systems.Proc IEEE Advanced Technologies for Communications.Conf. Ho Chi Minh City, Viet Nam,Oct.2010 pp.89-94

[28] S. Hammarling, *The numerical solution of the general Gauss-Markov linear model*, in Mathematics in Signal Processing (T.S. Durrani ,Eds) Clarendon Press, Oxford, 1986.

[29] C.PAige, *Some aspects of generalized QR factorization*, in Reliable Numerical Computations (M.Cox and S.Hammarling, Eds,),Clarendon Press, Oxford, 1990

[30] L.Dagum, R.Menon. *OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE.*

[31] E.Anderson, Z.Bai, J.Dongarra. *Generalized QR Factorization and Its Applications*. 1991

[32] R.J Epstein. *A history of econometrics* . North-Holland, 1987

[33] A. Gonschorek, I. Lu, J. Halliwill, J. A. Beightol, J. A. Taylor, H.Painter, and D. Eckberg. *Influence of respiratory motor neurone activity on human autonomic and haemodynamic rhythms*. Clinical Physiology, 21(3):323-334,2001

[34] G. Gujarati. Basic Econometrics. McGraw Hill,1995