UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Optimización en procesadores ARM de la descomposición matricial QR para Beamforming

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Enric Ruiz Andrés
Tutor: Pedro Alonso Jordá

Curso académico 2016/2017

Resumen

Los procesadores de propósito general y bajo consumo ARM que se montan en una amplía gama de dispositivos móviles y sistemas embebidos están revolucionando el mundo de la tecnología, que hoy en día implica casi cualquier sector imaginable. Las series ARMv7 y posteriores, tienen en su diseño la inclusión de unidades de cómputo avanzadas SIMD (*Simple Instruction Multiple Data*), proporcionando un recurso brillante para el procesamiento paralelo de datos. El campo del procesamiento digital de audio es el contexto de este trabajo y es donde aprovechamos las citadas capacidades en un proceso inherentemente de tiempo real. Dicho proceso consiste en la adecuada selección y descarte de señales acústicas en base a un cómputo iterativo en la actualización de las mismas. El modelo se simula y está implementado de manera eficiente mediante el uso del lenguaje de programación C, el uso de librerías de altas prestaciones como BLAS/LAPACK optimizado (ATLAS), y a la herramienta de computación paralela OpenMP. Todo ello bajo un entorno de aprovechamiento máximo del rendimiento respecto del consumo (*Flop/Watt*), que haga factible su uso en plataformas portátiles.

Palabras clave: descomposición y actualización QR, procesadores ARM, NEON *Intrinsics*, OpenMP, *Beamforming*.

Abstract

ARM's general purpose and low-power consumption processors, which are assembled in a wide range of mobile devices and embedded systems are revolutionizing the world of technology, which nowadays involves almost any sector imaginable. The ARMv7 and later series, have in their design the inclusion of advanced SIMD (*Simple Instruction Multiple Data*) units, providing a brilliant feature for parallel data processing. The field of digital audio processing is the context of this work and it is where we take advantage of the aforementioned capabilities in an inherently real-time process. This process consists in the proper acoustic signals selection and discard, based on an iterative compute in this updating signals. The model is simulated and it is implemented by using the C programming language, the optimized BLAS/LAPACK (ATLAS) high performance library and the OpenMP framework. All this under an environment of maximum profit of the performance against the consumption (Flop/Watt), doing feasible its use in mobile platforms.

Keywords: QR factorization and updating, ARM processors, NEON Intrinsics, OpenMP, Beamforming.

Dedicado a mi hermano Matías, quien comparte la misma ilusión por el campo de la informática. También a mis padres por haber hecho posible mi dedicación al estudio una gran parte de mi vida.

Mostrar mi agradecimiento a Pedro Alonso, por el tiempo y empeño invertido para la realización del trabajo presente.

Tabla de contenidos

Portada **1**

Resumen **3**

Español 3
Inglés 5

Prefacio **8**

Capítulo 1. Introducción **10**

1.1. Motivación 11
2.1. Objetivo TFG 12

Capítulo 2. Presentación del Problema **14**

2.1. Modelo de la señal 14
2.2. *Beamforming* 15
2.2.1. Algoritmos *Beamforming* 16
2.3. Algoritmo *QRupdating* 18
2.3.1. Costes 25

Capítulo 3. Descripción del Entorno **27**

3.1. Arquitectura ARM 27
3.2. Procesador ARM Cortex-A15 28
3.2.1. Tecnología NEON 29
3.2.1.1. Tipos de datos 29
3.2.1.2. NEON Intrinsincs 30
3.3. Hardware NVIDIA 31
3.4. Software 33
3.4.1. Lenguajes de programación, Librerías y Compiladores 33
3.4.2. Herramientas de Paralelización de datos y Concurrencia 34
3.4.3. Programas 35

Capítulo 4. Aplicación y Análisis **37**

4.1. Versión con BLAS/LAPACK 38
4.2. Versión con NEON *Intrinsics* 40
4.3. Versión con OpenMP 42
4.4. Comparativas 49
4.4.1. Acceso Contiguo contra Acceso No Contiguo en Q^T 50
4.4.2. Acceso Contiguo en R contra Acceso No Contiguo en R^T 50

4.4.3. BLAS/LAPACK vs. NEON.....54
4.4.4. NEON vs. OMP.....56
4.4.5. BLAS/LAPACK vs. NEON vs. OMP.....59

Capítulo 5. Conclusiones.....62

Bibliografía.....65

Prefacio

La creciente necesidad de productividad y entretenimiento móvil dio lugar a una clase relativamente nueva de dispositivos: *smartphones* y *tablets*.
Si bien es cierto que ya a mediados y finales de los años 80 habían muchos diseños, llegando algunos de ellos a producción y formando un pequeño mercado de dispositivos denominados PDA (*Personal Digital Assistant*) en los años 90, no fue hasta entrado el sigo XXI cuando vimos una auténtica revolución del mercado de los dispositivos móviles, que aunó muchos de los conceptos de las propias PDA y el auge de los teléfonos móviles.

En este nuevo panorama y el IOT (*Internet of Things*) en escena, los procesadores de bajo consumo se presentan como el actor principal.

El trabajo se estructura en cinco capítulos y un apartado descriptivo de las fuentes de las diversas referencias bibliográficas que se realizan a lo largo del mismo. Se expone brevemente a continuación el contenido de los capítulos:

El Capítulo 1 sirve de introducción a la temática que tratamos y con qué herramientas lo abordaremos, esto es, el procesamiento digital de audio y los procesadores con arquitectura ARM.

A continuación, en el Capítulo 2 se presenta de forma global el problema a tratar, su naturaleza matemática y los algoritmos que se manejan para resolverlo. Asimismo, se distingue el verdadero escenario de trabajo, discerniendo de forma clara el objetivo real de optimización que se persigue.

En el Capítulo 3 se hace una pequeña descripción de los componentes clave que componen el hardware utilizado, haciendo hincapié en aquellos que establecen la base para nuestra optimización. También se detalla el software empleado para la programación, gestión, diseño e implementación del trabajo.

La exposición de las diferentes soluciones aportadas son vistas en el Capítulo 4, donde se enuncian y detallan pudiendo constatar una evolución progresiva de rendimiento en cada una de ellas a partir de gráficas y datos de análisis.

El Capítulo 5 se reserva para las conclusiones sacadas tras el estudio y desarrollo del trabajo.

La Bibliografía sigue el esquema de referencias *IEEE Citation Style* [1].

Capítulo 1 Introducción

Las técnicas de procesamiento digital de señales han ido reemplazando a los métodos analógicos de procesamiento de señales de manera ascendente a lo largo de más de tres décadas en muchos campos como en el de la voz y el habla, el del radar y el *sonar*, las señales biomédicas, las telecomunicaciones o las señales geofísicas.

El procesamiento digital de audio es un tipo de DSP (siglas en inglés de *Digital Signal Processing*), que es la manipulación matemática de una señal de información con el fin de modificarla o mejorarla en algún sentido. En el caso del procesamiento digital del audio, empieza con la codificación/decodificación digital de una señal eléctrica a partir de una onda sonora mediante una secuencia de valores enteros, bits en nuestro ámbito. Esto se consigue a partir de dos procesos: el muestreo y la cuantificación digital de la señal eléctrica.

Para tal fin se utilizan procesadores o microprocesadores en circuitos integrados para aplicaciones específicas o ASIC (*Application-Specific Integrated Circuit*). Son los denominados procesadores digitales de señales o DSP (*Digital Signal Processor*).

En general, los procesadores DSP ejecutan pequeños algoritmos altamente optimizados de procesamiento de audio o vídeo a alta velocidad. Normalmente, se evita el uso de *caches* porque su comportamiento debe ser altamente reproducible. Mediante instrucciones SIMD (*Single Instruction Multiple Data*) y VLIW (*Very Large Intruction Word*) brindan procesamiento en paralelo ofreciendo, en ocasiones, diferentes memorias de datos en espacios de direccionamiento distintos para aminorar el impacto que comporta las dificultades de direccionamiento de memoria en este tipo de procesamiento. Todo ello siguiendo por lo general, una arquitectura Harvard que usa memorias separadas para instrucciones y datos.

Sin embargo, hay varios inconvenientes con estos componentes. Por un lado, la dificultad que desempeña en el caso de ser posible, la programación en ensamblador en este tipo de arquitecturas con juegos de instrucciones muy irregulares y complejos basados en VLIW. Por otro lado, suelen ser sistemas propietarios y de acceso nulo o limitado a la plataforma de desarrollo, donde algunas de las compañías más relevantes implicadas en el mercado tecnológico móvil y/o de procesamiento digital de señales ha optado por implementar sus propias soluciones al respecto o usar las de compañías especializadas.

Así por ejemplo, en el campo de la tecnología móvil, la compañía Qualcomm Technologies® tiene su DSP Hexagon™ [2] embebido en sus conocidos procesadores Snapdragon™. La empresa Samsung® se encomendaba a la solución ofrecida por Qualcomm Technologies®. Desde el año 2010, la empresa sur coreana diseña y fabrica sus propios procesadores Exynos™ y en su producto Samsung Galaxy ® S7 optó por la tecnología provista por el fabricante de chipsets DSP Group® [3] y licenciada por CEVA® [4]. Huawei® también opta por los chipsets de DSP Group® y los incorpora en sus procesadores Kirin™. Y para finalizar con este pequeño resumen de compañías, fabricantes y diseñadores, tenemos a Apple® que diseña sus propios procesadores para terminales móviles y *tablets* (aunque es Samsung quien los manufactura, actualmente) y se deduce una implementación del procesamiento digital de audio propia.

Como puede verse, las diferentes implementaciones de los DSP no hace sino que el estudio y la programación de tales componentes sea, con el añadido de los inconvenientes ya citados, un marco muy cerrado al sistema final en cuestión.

Con todo, hay un punto en común en todos los procesadores nombrados anteriormente que trabajan con los diferentes DSP, son procesadores de propósito general basados en diversas revisiones de la arquitectura ARM. La documentación y manuales técnicos acerca de su arquitectura y de sus unidades de cómputo es detallada y extensa.

ARM Holdings es la empresa que promueve el uso de la arquitectura ARM, y lo hace mediante la venta de licencias que otorgan el permiso al acceso de "núcleos de propiedad intelectual" para su utilización, implementación y manufacturación, entre otros, de sus diseños de procesadores. Como se ha indicado, ARM provee una basta documentación a los posibles programadores de sus arquitecturas, sin costes. Esto último, ha provocado en gran medida que exista una amplia comunidad de desarrolladores en torno a esta arquitectura. Actualmente, dominan el mercado de los procesadores en las plataformas móviles.

Nosotros abordamos el procesamiento digital de audio y el manejo de datos que comporta desde la perspectiva que nos aporta ARM, alejándonos pues del enfoque óptimo en prestaciones que nos ofrecería teóricamente la implementación sobre los DSP, aunque no ausente de problemas e inconvenientes como hemos visto. A cambio, podemos programar en lenguaje de alto nivel y más importante todavía, se desarrolla sobre una arquitectura bien conocida y usada en una amplísima gama de dispositivos, por lo que se asegura un alto grado de portabilidad.

1.1. MOTIVACIÓN

Aproximadamente 4 mil millones de personas, el 60% de la población mundial, toca cada día un dispositivo como usuario final que lleva un chip ARM [5] [6], además del creciente interés por la implantación de servidores basados en esta arquitectura en ciertos entornos [7]. Se presenta pues, una clara oportunidad tanto en el ámbito académico como laboral, donde experiencias previas como esta son una indudable ventaja.

Asimismo, el abanico de aplicaciones para las que el procesamiento de señales de audio sobre procesadores ARM puede ser de utilidad, es tan amplio como el número de campos descritos en la Introducción.

En concreto y a destacar, el cómputo y análisis en tiempo real de las señales acústicas en base a la capacidad selectiva de qué señales queremos y cuáles no, puede ser de gran ayuda en el campo de la ciencia médica, en la rama de la audiología en particular.

Cada vez que escuchamos cualquier sonido, las ondas sonoras entran por el oído y recorren nuestro conducto auditivo externo hasta llegar al tímpano, que hace vibrar ya en el oído medio, la cadena de huesecillos: *martillo*, *yunque* y *estribo*. Estas vibraciones llegan a la ventana oval y de ahí al *caracol* o *cóclea*, en el oído interno. En la cóclea se encuentra el *órgano de Corti*, el cual es un verdadero órgano terminal de la audición. En esta zona se encuentran unas células ciliadas que transforman el sonido en impulsos nerviosos que se transmiten al cerebro a través del nervio auditivo. Si algún elemento falla en este complejo proceso, se producen pérdidas de audición en mayor o menor grado [8].

Una primera solución para la pérdida parcial de audición son los audífonos, muy útiles cuando las anomalías están presentes en el oído externo o medio. Tienen el inconveniente de que amplifican todo el sonido que envuelve al individuo impidiendo en ocasiones una escucha selectiva adecuada no obteniendo una audición concorde y fidedigna. Además, puede que el paciente no disponga de una cóclea bien formada por lo que cualquier amplificación puede ser inútil [9].

Para estos casos, en los que existen malformaciones o ausencia de partes del oído interno, una solución más invasiva pero a la vez más efectiva es el implante coclear, que consiste en un transductor que transforma las señales acústicas en señales eléctricas que estimulan el nervio auditivo. Estas señales son procesadas mediante diferentes partes del implante (entre las que se encuentra un procesador DSP), algunas de las cuales se colocan en el interior del cráneo y otras en el exterior. Los inconvenientes para esta solución son dos principalmente: la operación quirúrgica que no está exenta de riesgos, y el precio al que puede ascender [10].

No se pretende aquí elaborar una solución directa a lo indicado ni mucho menos, sino más bien un preámbulo para ello. No hay que olvidar que, aún siendo cada vez más potentes, los procesadores ARM no son capaces actualmente de tratar de forma óptima y en tiempo real los procesamientos digitales de audio necesarios para el problema que aquí se plantea.

El reto de optimizar al máximo cada parte del código pertinente a estos procesamientos de audio es el motivo de este trabajo, para en un futuro próximo, ser capaces de profundizar acerca del interés de las plataformas móviles como recurso añadido a las soluciones ya existentes en el déficit de la capacidad auditiva.

Sin duda, también es emocionante comprobar como la tecnología móvil se acerca en prestaciones a rendimientos computacionales no tan distantes en el tiempo en plataformas de escritorio, empleando para ello menor consumo energético a un coste de fabricación más reducido.

1.2. OBJETIVO TFG

El propósito de este trabajo radica en el post-proceso de señales acústicas emitidas por altavoces y recogidas por micrófonos. Procesar y ser capaces de separar cada una de estas señales de acuerdo a su origen requiere de una potencia de cómputo de la plataforma destino para resultados en tiempo real, que solo conseguiremos si aunamos algunas técnicas de ingeniería como implementaciones óptimas con librerías de altas prestaciones o el acceso eficiente a memoria de datos, el uso de los componentes SIMD de los procesadores que contiene la propia plataforma, reunidos bajo el apodo de tecnología NEON [11], y el empleo de la herramientas de computación paralela OpenMP [12]. Recalcar que el desafío se encuentra en el procesamiento en tiempo real.

Para ello, nos centraremos en optimizar una pequeña parte del proceso descrito en el Modelo de la Señal, relativo a la actualización de las matrices resultado de la descomposición de la matriz primaria que almacena los datos captados por los micrófonos. Todo ello bajo el marco para la aplicación de filtros *beamforming* [13, págs. 23 a 32] para conseguir una señal resultante lo más parecida posible a una de las señales origen, como se explica y en consonancia a lo expuesto en la Presentación del Problema.

Capítulo 2 Presentación del Problema

Se dispone una sala con M altavoces que emiten sendas señales: $s_1(k), s_2(k), \ldots, s_M(k)$. Al otro lado de la habitación, existen N micrófonos que recogen las señales acústicas emitidas por los altavoces. La cuestión reside en cómo recuperar las señales emitidas por los altavoces y procesarlas según las grabaciones realizadas por los micrófonos [14], para ser capaces de distinguir y seleccionar una señal origen u otra según nos convenga. La idea presentada se puede visualizar en la Figura 2.1:

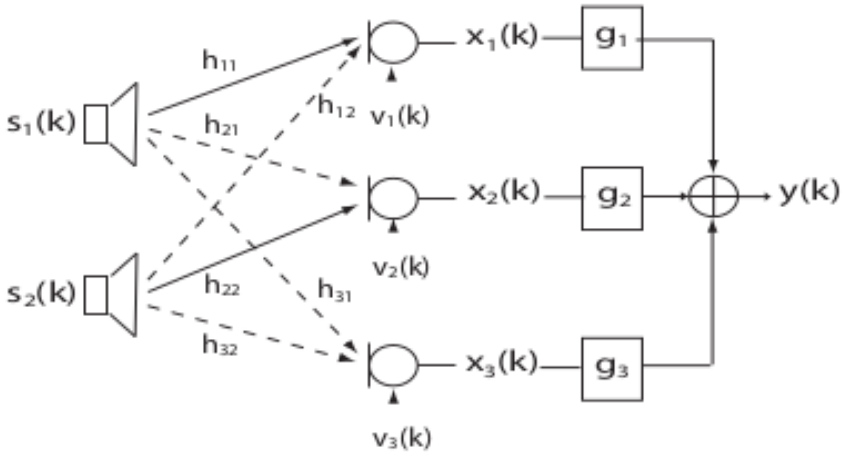


Figura 2.1*: Dos altavoces emitiendo señales acústicas, tres micrófonos registrándolas y una serie de filtros dan como resultado una salida lo más parecida posible a una de las señales emitidas en primera instancia por uno de los altavoces.*

2.1. MODELO DE LA SEÑAL

De acuerdo a la Figura 2.1, la salida de los micrófonos vendrá determinada por:

$$x_n(k) = \sum_{m=1}^{M} \sum_{j=1}^{L_h} h_{nm}(j) \cdot s_m(k-j) + v_n(k)\,, \qquad \text{(EQ 2.1)}$$

donde *m* toma valores desde uno hasta *M,* y *n* toma valores desde uno hasta *N*, siendo *M*=3

altavoces y N=2 micrófonos, para nuestro caso particular. L_h es la mayor de las longitudes de entre todas las respuestas impulsionales en los canales acústicos h_{nm} que recorren la habitación quedando registrados por los micrófonos. Así pues, h se define como la representación característica de la sala donde se realiza el proceso descrito. A mayor longitud de L_h, la precisión de las grabaciones en los micrófonos es mayor pero al mismo tiempo supone un agravio para el procesamiento en tiempo real al tener que manejarse más datos.

Como hemos mencionado, s_m hace referencia a la señal emitida por cada uno de los altavoces, y por lo que se refiere al ruido de la señal v_n, no se considerará en aras de mayor claridad. Aprovechando este descarte y en virtud de mejorar la eficiencia computacional, la EQ 2.1 puede ser reescrita de la siguiente forma:

$$x_n(k) = \sum_{m=1}^{M} \boldsymbol{h}_{nm}^{T} \cdot \boldsymbol{s}_m(k) \quad , \qquad \text{(EQ 2.2)}$$

donde $\mathbf{s}_m(k)$ es el vector columna definido como $\mathbf{s}_m(k) = [s_m(k) \;\; s_m(k-1) \;\; \cdots \;\; s_m(k-L_h+1)]^T$, $\mathbf{h}_{nm}$ es el vector del canal acústico desde el altavoz m hasta el micrófono n en $\Re^{(L_h x 1)}$. El símbolo $()^T$ denota la traspuesta de un vector o matriz.

Reflexionando ahora sobre el problema de recuperar las señales fuente $s_m(k)$ desde las señales obtenidas $x_n(k)$, se diseñan los filtros *beamforming* g_n de la Figura 2.1 en términos en los que la señal de salida $y(k)$ sea una buena estimación de $s_m(k)$, esto es, $y(k) = \hat{s}_m(k - \tau)$ con un mínimo error. Dada una longitud máxima L_g para cada uno de los filtros g_n, el ancho de banda de la señal de salida se expresa de una forma similar a la EQ 2.2 :

$$y(k) = \sum_{n=1}^{N} \boldsymbol{g}_n^{T} \cdot \boldsymbol{x}_n(k) \quad , \qquad \text{(EQ 2.3)}$$

donde $\mathbf{g}_n$ es el vector en $\Re^{(L_g x 1)}$ que contiene los consecuentes filtros *beamforming* de forma ordenada de la Figura 2.1, y $\mathbf{x}_n(k)$ es el vector columna definido como $\mathbf{x}_n(k) = [x_n(k) \;\; x_n(k-1) \cdots x_n(k-L_g+1)]^T$.

Con el propósito de calcular el vector entero $\mathbf{x}_n(k)$ usado como matriz en la EQ 2.3, la EQ 2.2 se reescribe en forma compacta redefiniendo $\mathbf{h}_{nm}$ como matrices de Sylvester. Para más detalles consultar [15].

2.2. BEAMFORMING

El *Beamforming* o conformación de haces, es una técnica de procesado de señal usada para distinguir entre las propiedades espaciales de una señal objetivo y el ruido de fondo. El dispositivo utilizado para realizar dicha tarea es denominado “conformador de haces” o *beamformer*. Entiéndase el concepto de *beamformer* como un conjunto de elementos (en nuestro caso, micrófonos) que captando por separado conforman una única señal a analizar.

Un *beamformer* en la recepción de las señales puede aumentar la sensibilidad en la dirección de la señal útil (máximo del *beamformer*) y reducirla en la dirección de señales de interferencia (nulos en esa dirección). También se puede conseguir en la transmisión, estableciendo un máximo en la dirección deseada y nulos en direcciones que no vayan al destino deseado [16]. En nuestro caso no se cuenta con la ayuda de *beamformers* en el origen sino que los altavoces emiten el sonido de forma dispersa. Es en el destino, en las señales captadas por los micrófonos donde se aplica el *beamforming*.

Este sistema es el que se utiliza en numerosas aplicaciones en radares, *sonars*, sismología,

comunicaciones *wireless,* radio astronomía, acústica o biomedicina entre otras. Nosotros la utilizaremos de forma simulada para el cálculo de resultados y tiempos en este trabajo.

Cabe resaltar que para la simulación de dicha técnica se hará uso del material aportado por el tutor del proyecto y Francisco J. Alventosa Rueda, basado en su Tesis de Máster [13] y no se ahondará en detalles referentes a la implementación de dicha parte. No obstante, es necesario hacer un repaso acerca del paradigma que permite la simulación del *Beamforming* para posteriormente explicar qué queremos optimizar en dicho proceso.

2.2.1. ALGORITMOS BEAMFORMING

En [15], Benesty et al. presenta un excelente repaso de los principales algoritmos en procesamiento de señal. En [13], Francisco J. Alventosa se centra en el algoritmo LCMV (*Linear Constrained Minimum Variance*) basado en correlación de matrices, puesto que ofrece un mejor rendimiento a nivel computacional.

El algoritmo LCMV calcula los filtros *beamforming* de acuerdo a la EQ 2.4:

$$g^{LCMV} = \hat{R}_x^{-1} H_{:m} \left[H_{:m}^T \hat{R}_x^{-1} H_{:m} \right]^{-1} \boldsymbol{u}_m \quad , \tag{EQ 2.4}$$

donde g^{LCMV} está formado por la concatenación de los filtros g_n , esto es, $g^{LCMV} = [\, g_1^T \cdots g_N^T \,]^T$, la matriz $H_{:m}$ es una partición de la matriz de impulso del canal que solo incluye las respuestas al impulso de las *m*-ésimas fuentes de los N micrófonos [17] utilizados en la matriz de Sylvester para las dimensiones $[N \; L_g \cdot L_h - 1]$. La matriz $\hat{R}_x$ es la matriz de correlación de las señales capturadas y $\mathbf{u}_m$ es un vector de ceros a excepción de una de las componentes del vector con el fin de compensar el retardo de la respuesta al impulso de la habitación.

Buscando la implementación más eficiente y precisa del LCMV, utilizamos un método basado en la descomposición QR de la matriz X^T , definida como $X \in \mathbb{R}^{[N \, L_g \cdot K]}$:

$$X = \frac{1}{\sqrt{K}} \begin{pmatrix} x_1(k) & x_1(k+1) & \cdots & x_1(k+K-1) \\ x_2(k) & x_2(k+1) & \cdots & x_2(k+K-1) \\ \vdots & \vdots & \cdots & \vdots \\ x_N(k) & x_N(k+1) & \cdots & x_N(k+K-1) \end{pmatrix} \quad , \tag{EQ 2.5}$$

donde, $K > N\, L_g$ es el número de muestras utilizadas.

Esta forma, $X^T = Q \cdot R$, donde Q es una matriz ortogonal y R es una matriz triangular superior, permite la resolución rápida de sistemas algebraicos lineales empleando las rutinas de la librería LAPACK. Nosotros construimos la matriz X^T en la representación *Column Major Order* para poder aplicar directamente estas rutinas de forma eficiente.

Considerando la descomposición QR de las observaciones de los micrófonos, se puede redefinir $\hat{R}_x$ como:

$$\hat{R}_x = X \cdot X^T = R^T \cdot Q^T \cdot Q \cdot R = R^T \cdot R \quad .$$

A continuación se define por conveniencia la matriz W como $W = \hat{R}_x^{-1} H_{:m}$, así que el filtro *beamformer* LCMV (g^{LCMV}) definido en la EQ 2.4 queda de la siguiente forma:

$$g^{LCMV} = W \left[H_{:m}^T W \right]^{-1} \boldsymbol{u}_m \quad . \tag{EQ 2.6}$$

Se define la matriz Z como la solución al sistema lineal mostrado en la EQ 2.7:

$$R^T Z = H_{:m} \ , \qquad \text{(EQ 2.7)}$$

entonces, utilizando la descomposición QR de la matriz *X*, tenemos:

$$W = \hat{R}_x^{-1} H_{:m} = (R^T R)^{-1} H_{:m} = R^{-1} R^T H_{:m} = R^{-1} Z \ ,$$

donde se puede apreciar claramente que la matriz W es la solución al sistema lineal $RW = Z$.

La solución para obtener los filtros *beamforming* se obtiene de resolver el sistema lineal de la EQ 2.8:

$$A \boldsymbol{b}_m = \boldsymbol{u}_m \ , \qquad \text{(EQ 2.8)}$$

donde $A = H_{:m}^T W = H_{:m}^T R^{-1} Z = Z^T Z$. También aquí, la solución del sistema lineal para la EQ 2.8 es obtenida a partir de la factorización QR, en este caso, de la matriz *Z*. Ahora, $Z=Q^T R^T$ es la descomposición QR de la matriz *Z*, entonces el vector $\mathbf{b}_m$ puede ser calculado resolviendo los dos siguientes sistemas lineales triangulares:

$$R^T \boldsymbol{y} = \boldsymbol{u}_m \ , \qquad \text{(EQ 2.9)}$$

$$R^T \boldsymbol{b}_m = \boldsymbol{y} \ . \qquad \text{(EQ 2.10)}$$

Finalmente, se deduce que el cálculo del filtro *beamformer* de la EQ 2.6 puede ser calculado empleando los últimos cálculos realizados, esto es, *R*, *Z* y $\mathbf{b}_m$ de la forma que se muestra en la EQ 2.11:

$$g^{LCMV} = R^{-1} Z \boldsymbol{b}_m \ . \qquad \text{(EQ 2.11)}$$

Estos últimos cálculos se resuelven realizando un producto matriz-vector y resolviendo un sistema lineal triangular.

Una vez analizado matemáticamente por completo el modelo de la señal, se concluye en [13, pág. 45] que la mayor carga computacional del mismo, reside en la factorización QR de la matriz *X*, suponiendo entre un 60% y un 70% del coste total del algoritmo.

Reducir dicho coste en concreto, es la tarea de este trabajo y se empieza a detallar en el siguiente punto.

2.3. ALGORITMO QR UPDATING

La factorización QR se utiliza para resolver sistemas lineales de ecuaciones, pudiendo ser éstos sobredeterminados, lo que permite resolver el problema de mínimos cuadrados para obtener, por ejemplo, regresiones lineales.

Son tres las formas de descomposición QR más conocidas:

1. El método de ortogonalización de Gram-Schmidt.
2. Mediante el uso de reflexiones de HouseHolder.
3. Mediante rotaciones de Givens.

Para más detalle, consultar [18].

El procedimiento para actualizar la descomposición QR que se explica a continuación utiliza rotaciones de Givens y ha sido extraído y adaptado a partir de [19]. Siendo X una matriz $n \times m$ y su factorización $X = Q \cdot R$, donde $Q \in \mathbb{R}^{m \times m}$ es ortogonal y $R \in \mathbb{R}^{n \times m}$ es triangular superior.

La actualización de la descomposición QR consiste en el procesamiento iterativo de la factorización de una nueva matriz X_1 , siendo esta $X_1 = Q_1 R_1$, donde la matriz X_1 es el resultado de eliminar las primeras f filas de X_1 y añadir nuevas f filas al final de X. En todo este proceso, designado como *QRupdating*, asumimos que las matrices X y X_1 son de rango completo (regulares o invertibles).

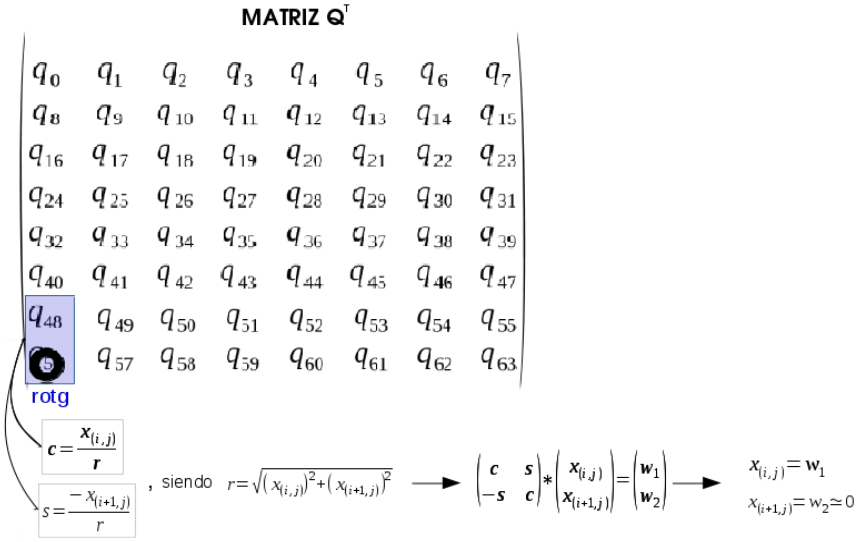
Siendo $H \in \mathbb{R}^{m \times m}$ una matriz ortogonal tal que $H^T Q^T = \begin{pmatrix} I_f & 0 \\ 0 & Q^T \end{pmatrix}$, es decir, H es el factor ortogonal de la descomposición QR de las primeras f columnas de la matriz Q^T, donde I_f es la matriz identidad de orden f. Entonces:

$$X = (QH)(H^T R) = \begin{pmatrix} I_f & 0 \\ 0 & Q^T \end{pmatrix} \begin{pmatrix} V \\ R^T \end{pmatrix} . \qquad \text{(EQ 2.12)}$$

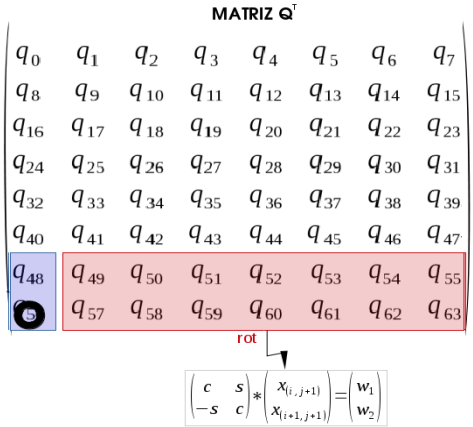
La matriz $V \in \mathbb{R}^{f \times n}$, contiene las primeras f filas de la matriz X, tal que $V = X_{1:f,1:n}$. Además, construyendo correctamente la matriz ortogonal H, no es difícil sacar la matriz R^T como una matriz triangular superior.

Esta parte del proceso *QRupdating* se muestra en el Algoritmo 1, se denomina *QRupdate: Eliminar Filas* y se describe gráficamente en la siguiente secuencia, tomando como ejemplo una matriz X^T de m=8 y n=4 ($Q^T \in \mathbb{R}^{8 \times 8} \wedge R^T \in \mathbb{R}^{8 \times 4}$), siendo dos filas a actualizar, esto es, f=2.

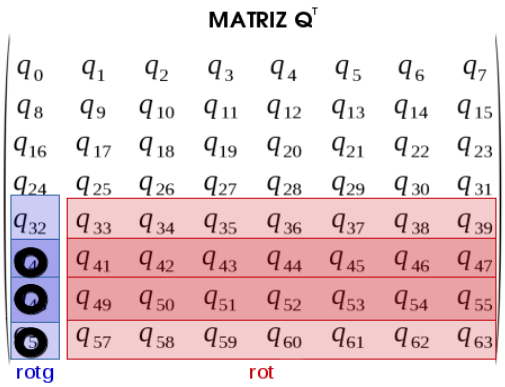
1. Se obtienen los senos y cosenos característicos de la matriz de Givens y se aplican mediante la rotación de Givens correspondiente sobre sobre la dupla de elementos en cuestión (*rotg*). Esto provoca que el elemento $x_{(i+1,j)}$ sea ahora un cero:



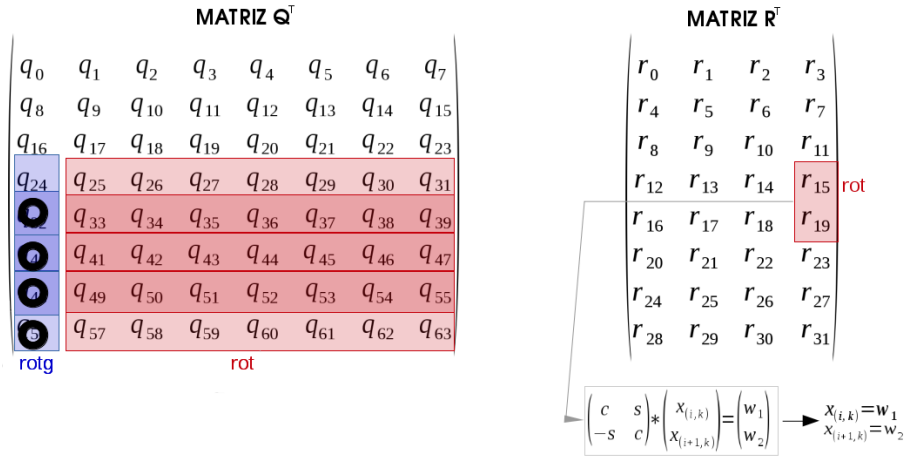
2. Se aplica las rotaciones de Givens al resto de las filas (*rot*) correspondientes de Q^T. También es una aplicación por duplas de elementos, pero a lo largo de ambas filas y a partir de la matriz de Givens calculada en el punto anterior.



3. Se itera el proceso descrito en 1.) y 2.) desde la fila *m* hasta la fila *n+j* , siendo *n*=4 y *j* tomando el valor de uno hasta *f* filas. Por lo que este caso la fila *n+j* es la quinta fila en esta primera iteración (*j*=1). La matriz Q^T quedaría así hasta este punto:

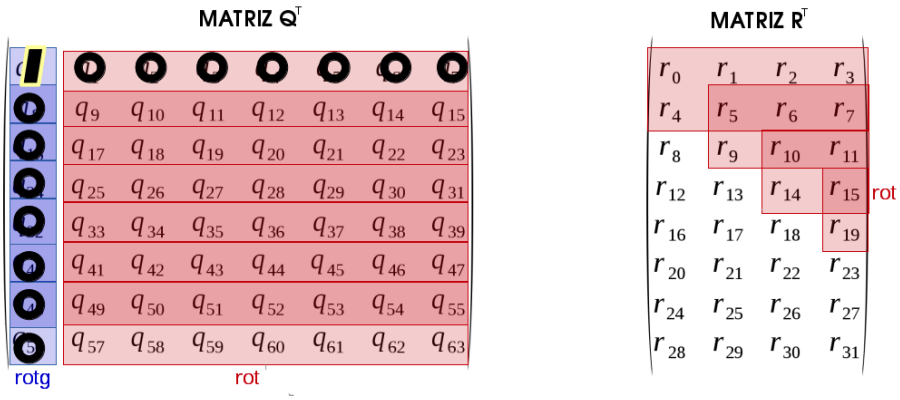


4. A partir de la fila *n+j-1* hasta primera fila, las rotaciones de Givens también afectarán a la matriz R^T:



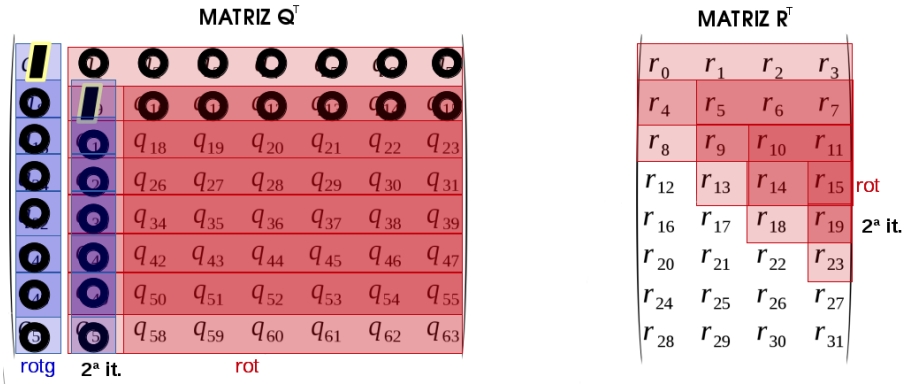
Ocurre así porque a partir de la fila *n+j* del Algoritmo 1 estamos modificando la que será la parte triangular superior de R^T, cuyos valores necesitaremos para el paso siguiente, denominado *QRupdate: Añadir filas*.

5. Siguiendo con el proceso hasta el final de esta primera iteración se obtiene:



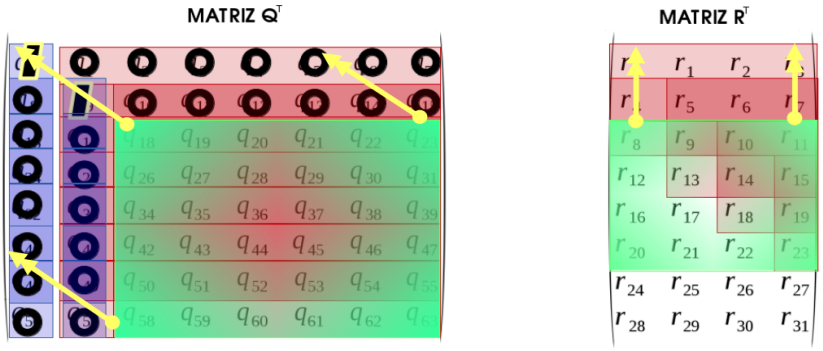
Como puede verse, al finalizar el proceso de generación de matrices de Givens (*rotg*) en Q^T, el primer elemento del primer vector columna pasa a ser la unidad, y la sucesiva aplicación de las rotaciones (*rot*) sobre el resto de esta matriz, provoca que el primer vector fila donde se han aplicado las rotaciones, sea una ristra de ceros.

6. La segunda iteración del proceso *QRupdate: Eliminar Filas* que corresponde con la segunda fila a actualizar quedaría como sigue:

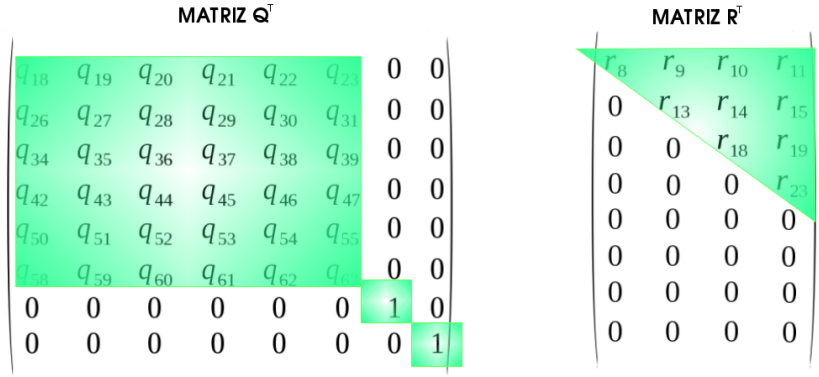


De nuevo, el primer elemento del segundo vector columna (que empieza en el segundo elemento de la segunda columna de la matriz Q^T), es la unidad tras *rotg* y el segundo vector fila se torna a ceros tras *rot*. De modo que de seguir el proceso las $m - f$ filas restantes se formaría la matriz identidad. Este caso, sin embargo, no ocurrirá en la práctica pues $f << m$.

7. Se deben ahora recolocar las matrices Q^T y R^T, preparándolas para la adición de f nuevas filas procedentes de los datos de las grabaciones realizadas por los micrófonos, esto es, de la matriz X.



8. Se desplazan los datos de Q^T que quedan sin ser anulados en el proceso anterior, y los datos en R^T a partir de la fila $f+1$ hasta $f+1+n$, es decir, quitando tantas filas como las que tenemos intención de añadir ($f=2$) y salvaguardando solamente las filas que contienen los datos que forman la nueva triangular superior de R^T:



Con estas traslaciones dejamos las últimas f filas y columnas de Q^T a cero a excepción de la diagonal unitaria para esas f filas, y ponemos a cero la parte triangular inferior de R^T . Acabamos pues, por deshacernos de las primeras f filas de la matriz X, esto es, eliminando V de la EQ 2.12.

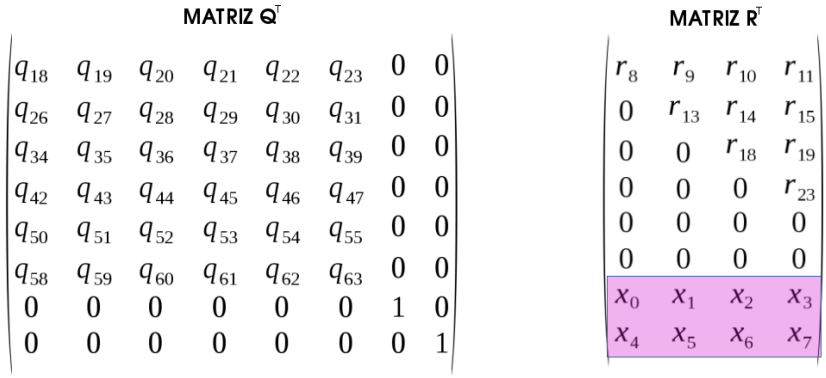
El siguiente paso se basa en el Algoritmo 2, *QRupdate: Añadir Filas,* donde en primera instancia se añaden f filas nuevas al pie de X^T, que denotaremos como la matriz $U \in \mathbb{R}^{f \times n}$. Usando el hecho que la descomposición QR de X^T es $X^T = Q^T R^T$, la matriz del sistema buscado puede representarse como:

$$X_1 = \begin{pmatrix} X^T \\ U \end{pmatrix} = \begin{pmatrix} Q^T & 0 \\ 0 & I_f \end{pmatrix} \begin{pmatrix} R^T \\ U \end{pmatrix} , \qquad \text{(EQ 2.13)}$$

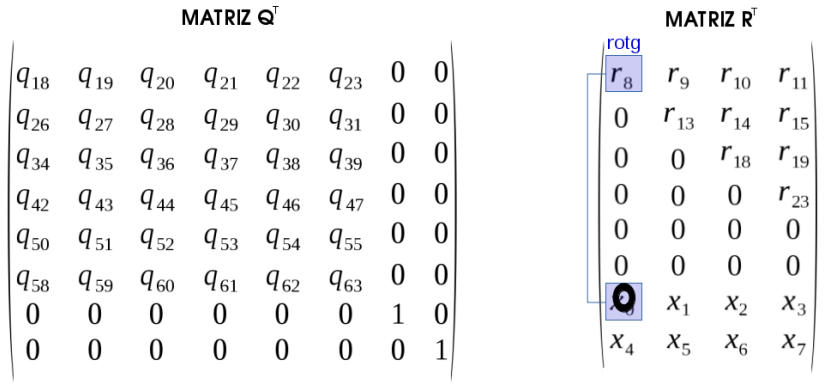
donde el objetivo reside en anular la matriz U completamente de forma que surja la nueva descomposición QR de X_1.

Se describe *QRupdate: Añadir Filas* de forma gráfica en la siguiente secuencia siguiendo con el ejemplo propuesto:

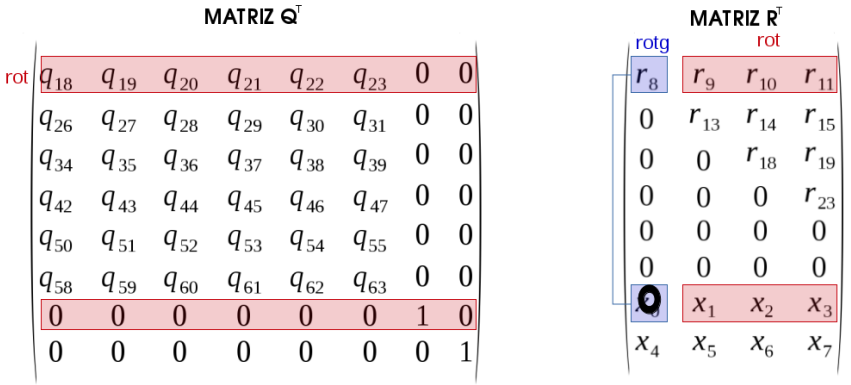
1. La matriz *U* de la EQ 2.13, resaltada en tono rosado, son los nuevos datos de X^T, a partir de los cuales y de la parte triangular superior de R^T se calcularán las matrices de Givens que luego aplicaremos a las correspondientes filas tanto de R^T como Q^T.



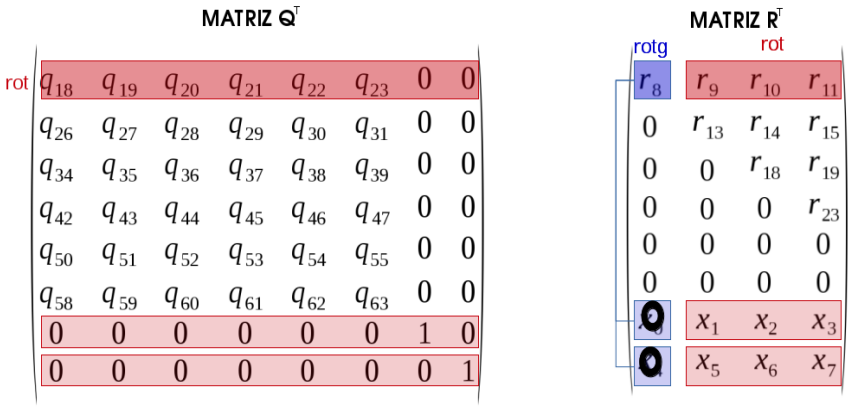
2. Si en la fase de *QRupdate:Eliminar Filas* se construían las matrices de Givens a partir de los elementos del primer vector columna de Q^T, ahora se calculan las matrices de Givens (*rotg*) a partir de los elementos de la diagonal de la parte superior de R^T y los nuevos elementos añadidos desde X^T:



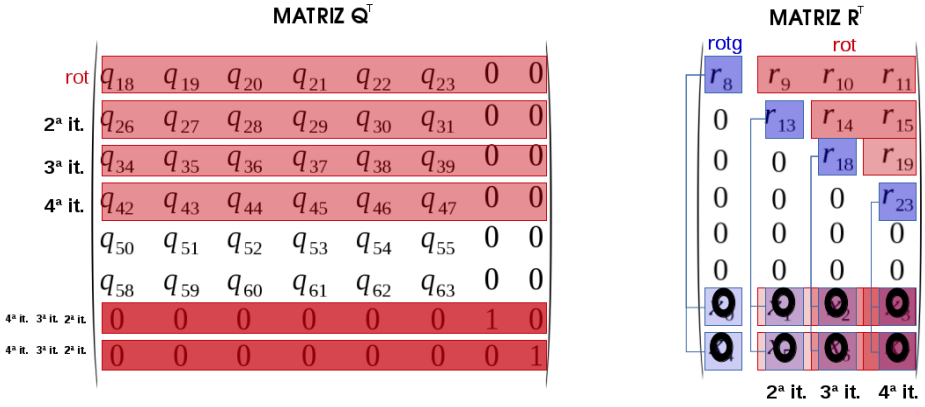
3. A continuación se aplican las rotaciones de Givens (*rot*) sobre las filas correspondientes de ambas matrices:



4. Se repite el proceso para las *f* filas restantes adheridas a R^T, en este caso, solo una fila más:



5. Se repite el proceso para las *n* columnas que tenga R^T:



Véase que las rotaciones sobre las filas en Q^T se producen en el mismo orden de filas que en R^T, donde se calculan y aplican las matrices de Givens. También indicar que, en esta fase hay un menor cálculo de matrices de Givens que en la fase de *QRupdate:Eliminar Filas.* Esto lo veremos reflejado en el cálculo de costes para una y otra fase.

Aquí termina el proceso de *QRupdating* para una iteración de *f* filas a actualizar, siendo ahora posible la aplicación conveniente de los filtros g_n presentados en la Figura 2.1, conforme a lo expuesto en los Algoritmos *Beamforming*. Se repetirá el proceso completo tantas veces como nuevas *f* filas puedan ser suministradas desde la matriz fuente *X*, donde están almacenadas las señales capturadas por los micrófonos. Esto, en realidad, debería suceder en tiempo real, es decir, conforme esté sucediendo la propia grabación de las señales emitidas por los altavoces.

Acerca de la secuencia mostrada de acuerdo al ejemplo planteado, indicar que las matrices que se forman a partir de los ficheros aportados para la simulación del *Beamforming*, son de tamaño mucho mayor. Se ha optado para el ejemplo por unos tamaños manejables, capaces de mostrar el proceso de manera gráfica, desmenuzando cada fase en cada uno de los pasos que consta, pero no son tamaños representativos de los casos reales.

En concreto, se nos aportan tres ficheros. Uno de ellos con las señales registradas por los micrófonos a partir del cual construimos en primera instancia la matriz *X* y otros dos ficheros de datos que representan dos tamaños distintos para la matriz *H* con los parámetros L_h y L_g prefijados, mediante la cual se calculan los filtros para aplicar en *X*.

En la simulación *Beamforming* se tiene que *m* es aproximadamente del orden de cuatro veces mayor a *n* y *f* es muy inferior a *n*, siendo *m* el número de filas, *n* el número de columnas y *f* el número de filas a actualizar.

Se detallan a continuación las funciones de generación de matrices de Givens (*rotg*) y aplicación de las rotaciones de Givens (*rot*), que tanto el Algoritmo 1 como el Algoritmo 2 utilizan. También se muestran los Algoritmos citados:

ALGORITMO 1
QRupdate: Eliminar Filas

Requiere: Q',R',f
Asegura: Q, R

```
 1: Q=Q';
 2: R=R';
 3: m = size( Q, 1 );
 4: n = size( R, 2 );
 5: for j = 1:f
 6:   for i = m-1:-1:n+j
 7:     [Q(i,j),Q(i+1,j),c,s] = rotg(Q(i,j),Q(i+1,j));
 8:     [Q(i,j+1:end),Q(i+1,j+1:end)] = rot(Q(i,j+1:end),Q(i+1,j+1:end),c,s);
 9:   end
10:   k = n;
11:   for i = n+j-1:-1:j
12:     [Q(i,j),Q(i+1,j),c,s] = rotg(Q(i,j),Q(i+1,j));
13:     [Q(i,j+1:end),Q(i+1,j+1:end)] = rot(Q(i,j+1:end),Q(i+1,j+1:end),c,s);
14:     [R(i,k:end),R(i+1,k:end)] = rot(R(i,k:end),R(i+1,k:end),c,s);
15:     k = k - 1;
16:   end
17: end
18: W=eye(m);
19: W(1:m-f,1:m-f) = Q(f+1:m,f+1:m);
20: Q=W;
21: R(1:n,:) = R(f+1:f+n,:);
22: R = triu(R);
```

ALGORITMO 2
QRupdate: Añadir Filas

Requiere: U, Q, R, f
Asegura: Q, R

```
 1: R(m-f+1:m,:) = U(1:f,:);
 2: m = size( Q, 1 );
 3: n = size( R, 2 );
 4: for  j = 1:n
 5:   for i = m-f+1:m
 6:     [R(j,j),R(i,j),c,s] = rotg(R(j,j),R(i,j));
 7:     [R(j,j+1:end),R(i,j+1:end)] = rot(R(j,j+1:end),R(i,j+1:end),c,s);
 8:     [Q(j,:),Q(i,:)] = rot(Q(j,:),Q(i,:),c,s);
 9:   end
10: end
```

FUNCIÓN ROTG
Generación de la matriz de Givens y aplicación de la misma

Requiere: p, q
Asegura: p, q, c, s

```
1: X = givens(p,q);
2: w = X*[p;q];
3: p = w(1);
4: q = w(2);
5: c = X(1,1);
6: s = X(1,2);
```

FUNCIÓN ROT
Aplicación de la rotación de Givens

Requiere: p, q, c, s
Asegura: p, q

```
1: X = [ [ c s ]; [ -s c ] ];
2: Y = X * [ p; q ];
3: p = Y(1,:);
4: q = Y(2,:);
```

2.3.1 COSTES

Tras el desglose visto en las secuencias para las dos partes que conforman el Algoritmo *QRupdating,* ambas fases basan la actualización de las matrices Q y R en la eliminación o en la adición de *f* filas de acuerdo al cálculo de matrices de Givens y sus consiguientes rotaciones. Sin embargo, se pueden distinguir dos esquemas de actuación que implican costes diferentes

En concreto, en *QRupdate:Eliminar Filas* tenemos que para la matriz Q^T, se calculan *m-j* matrices de Givens para cada una de las filas a actualizar, las cuales sirven para aplicar las rotaciones sobre duplas de filas adyacentes de *m-j* datos, siendo *j=1* hasta *f* filas a actualizar. En relación a la matriz R^T en esta misma fase, se aplican las rotaciones sobre *n-k* datos en cada una de las dos filas adyacentes pertinentes en cada iteración, siendo *k=n-1* hasta cero (en correlación al modo que actúa el algoritmo), repitiendo el proceso para *f* filas a actualizar.

En *QRupdate:Añadir Filas* y en relación a la matriz R^T, se calculan *f* matrices de Givens en *n* ocasiones, las cuales sirven para aplicar las rotaciones sobre *n-k* datos en la primera y en la última fila, siendo *k=0* hasta *n-1* e incrementando *k* en cada una de las siguientes columnas a tratar, cubriendo la diagonal superior de R^T y las *f* filas añadidas en R^T. En relación a la matriz Q^T en esta misma fase, se calculan $n \cdot f$ rotaciones de Givens sobre *n+f* filas de *m* datos cada una.

Teniendo en cuenta lo analizado, deducimos matemáticamente los costes de ambas fases por separado en base al número de operaciones que implican el cálculo de matrices y rotaciones de Givens. Esto nos ayudará a interpretar los resultados finales de acuerdo a las mejoras que implementamos en el trabajo.

- Costes de *QRupdate: Eliminar Filas:*
  - en la matriz R^T: $t_{R^T} = f \cdot \sum_{k=0}^{n-1} (n-k) = f \cdot (\frac{n^2+n}{2})$,
  - en la matriz Q^T: $t_{Q^T} = \sum_{j=0}^{f-1} (m-j)*(m-j-1) \approx fm^2 - f^2 m = f \cdot (m^2 - fm)$
  - total: $t_{eliminar} \approx f \cdot (m^2 - fm) + f \cdot (\frac{n^2+n}{2}) = f \cdot (m^2 - fm + \frac{n^2+n}{2})$.

- Costes de *QRupdate:Añadir Filas:*
  - en la matriz R^T : $t_{R^T} = f \cdot \sum_{k=0}^{n-1} (n-k) = f \cdot (\frac{n^2+n}{2})$,
  - en la matriz Q^T: $t_{Q^T} = \sum_{i=0}^{n-1} (f \cdot m) \approx f \cdot (m \cdot n)$,
  - total: $t_{añadir} = fmn + f(\frac{n^2+n}{2}) = f \cdot (mn + \frac{n^2+n}{2})$.

Podemos concluir, a raíz del análisis y los cálculos, que la fase *QRupdate:Eliminar Filas* es más costosa que *QRupdate:Añadir Filas.*

Capítulo 3 Descripción del Entorno

A continuación se describe de manera breve la arquitectura de la plataforma destino sobre la que se ha planteado y desarrollado el trabajo, esto es, la placa NVIDIA® Jetson TK1. También se detalla la misma de acuerdo a sus características y componentes más relevantes. Por último, se expone el software utilizado para la implementación de la tarea.

Como se ha indicado anteriormente, trataremos con procesadores de propósito general ARM, en concreto sobre los procesadores ARM Cortex A-15 basados en ARMv7. Esta versión, al igual que revisiones posteriores, posee propiedades muy atractivas desde la perspectiva del procesamiento de datos en paralelo. Nuestra placa en cuestión, cuenta con hasta cuatro *cores* AMR Cortex A-15 los cuales tienen capacidades SIMD (*Simple Instruction, Multiple Data*).

Esto, unido al uso de GPUs (*Graphics Processor Units*) en el mismo SoC (*System on Chip*) y un diseño de bajo consumo [20, pág. 28] hacen de la placa Jetson TK1 un sistema de interés alrededor del concepto *Flop/Watt.* El diseño de bajo consumo se sustenta en cuatro aspectos básicos [21] que se enumeran a continuación:

1. Cambios a bajas frecuencias de funcionamiento en situaciones de poca carga computacional.
2. Uso de menor número de transistores con tecnología de fabricación nanométrica.
3. Uso de *Sleep Mode* que detienen al procesador hasta nueva orden.
4. Juego de instrucciones simple, versátil y muy regular, donde las extensiones de instrucciones son a través de co-procesadores tratados a modo de memoria auxiliar.

3.1. ARQUITECTURA ARM

En la ingeniería de la computación, la arquitectura de una máquina identifica los componentes, así como sus interrelaciones y marca una serie de reglas y métodos que describen la funcionalidad, organización e implementación del sistema en cuestión.

ARM es una arquitectura RISC (*Reduced Instruction Set Computer*), las características más relevantes de la cual son:

- Instrucciones de tamaño fijo, mayormente sencillas y presentadas en un reducido número de formatos.
- Sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos con pocos modos de direccionamiento.
- Gran número de registros de uso general, cuya utilización es optimizada en el compilador.
- Unidad de control cableada.
- Especial énfasis en la segmentación.

Además, la arquitectura ARM provee una serie de mejoras [22, págs. 27 a 33], que explotan las características anteriores :

- Control sobre la Unidad Aritmético Lógica (ALU) y el desplazador en la mayoría de instrucciones de procesamiento de datos para maximizar la combinación de ambos (ALU y *shifter*).
- Modos de direccionamiento con auto-incremento y auto-decremento para optimizar los bucles de los programas.
- Carga/Almacenamiento de múltiples instrucciones para incrementar la salida de datos.
- Ejecución condicional de muchas instrucciones para aumentar la velocidad de ejecución final del programa.
- Extensiones del *Instruction Set Architecture* (ISA). Algunas de ellas comportan extensiones también a nivel de hardware.

El diseño basado en RISC en general y ARM en particular, implica un menor número de transistores que, por ejemplo, los procesadores x86 basados en arquitecturas CISC (*Complex Instruction Set Computer)*, que son el referente en ordenadores de sobremesa. Esto comporta a su vez una reducción de los costes, calor y energía. Es esta reducción energética relacionada con el rendimiento computacional lo que hace de este tipo de sistemas interesantes para la optimización de algoritmos y programas para alcanzar buenas cotas *Flop/Watt*.

En este texto, como se ha mencionado, trabajamos sobre la arquitectura ARMv7, en concreto sobre los procesadores ARM Cortex-A15 que implementan dicha arquitectura y nos centraremos en la extensión *Advnaced Single Instruction Multiple Data version 2* (SIMDv2) para operaciones sobre registros vectoriales de números de coma flotante.

3.2. PROCESADOR ARM CORTEX-A15

El procesador ARM Cortex-A15 suele presentarse en compañía de otros tres de su misma clase bajo un mismo chip, denominado por ARM como Cortex-A15 MPCore, como puede verse en la Figura 3.2.

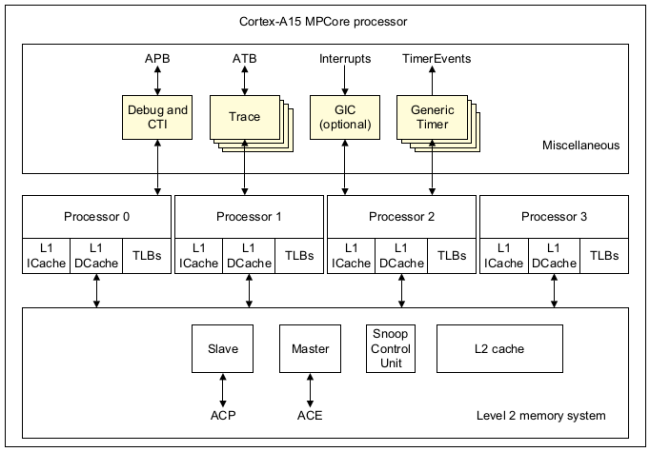


Figura 3.2*: ARM Cortex-A15MPCore* [23, pág. 13]

Se trata de un procesador superescalar con *out-of-order-pipeline*. Es capaz de hacer saltos predictivos dinámicos gracias a la implementación de *Branch Target Buffer* (BTB) y *Global History Buffer* (GHB). Posee tres *Translation Look-aside Buffers* (TLBs) de 32 entradas para instrucciones, carga de datos y almacenamiento de datos. El tamaño de las *caches* de L1 para datos e instrucciones es de 32KB cada una y de 512KB a 4MB(tamaño configurable) para una L2 compartida [23, pág. 16].

Además, cada procesador cuenta con una unidad SIMD avanzada de 128 bits, denominada por ARM como tecnología NEON. Dicha unidad es la base para el procesamiento de señales de vídeo y/o audio aumentando el rendimiento y disminuyendo el consumo.

El uso de este último componente es uno de los ejes principales para la mejora de prestaciones que se presenta en el trabajo.

3.2.1. TECNOLOGÍA NEON

Con el uso de esta unidad SIMD avanzada se nos presentan los *registros vectoriales*, que son como su nombre indica, vectores de elementos que se dividen en *carriles,* y cada carril contiene el valor de un elemento.

Por norma general, cada instrucción NEON resulta en **n** operaciones en paralelo, donde **n** es el número de carriles en los que están divididos los vectores con los que se trabaja. Cada operación está contenida en el propio carril y no hay posibilidad de desbordamiento de un carril a otro.

El número de carriles depende del tamaño del vector NEON con el que se trabaje. Así pues:

- Vectores NEON de 64-bit pueden contener:
  - Ocho elementos de 8-bit.
  - Cuatro elementos de 16-bit.
  - Dos elementos de 32-bit.
  - Un elemento de 64-bit.
- Vectores NEON de 128-bit pueden contener:
  - Dieciséis elementos de 8-bit.
  - Ocho elementos de 16-bit.
  - Cuatro elementos de 32-bit.
  - Dos elementos de 64-bit.

3.2.1.1. Tipos de datos

64-bit type (D-register)	128-bit type (Q-register)
int8x8_t	int8x16_t
int16x4_t	int16x8_t
int32x2_t	int32x4_t
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_t
poly8x8_t	poly8x16_t
poly16x4_t	poly16x8_t

Tabla 3.2.1*: Tipo de datos para cada clase de vector NEON* [11, pág. 77].

La unidad SIMD avanzada trata los vectores citados como registros de 64-bit y 128-bit, denominados *doubleword* (registro tipo D) y *quadword* (registro tipo Q), respectivamente.

Dependiendo del tipo de vector, la unidad NEON trabajará con un tipo u otro de registro. Podemos ver qué tipo datos y cuántos elementos puede albergar cada clase de vector NEON en la Tabla 3.2.1.

La nomenclatura que se sigue para los tipos de datos contenidos en los vectores es :

*<tipo><tamaño>x<número_de_carriles>*_t

En total se disponen de hasta dieciséis registros tipo Q {Q0-Q15} y treinta y dos de tipo D {D0-D31}.

3.2.1.2. NEON Intrinsincs

Las funciones que son reemplazadas por las instrucciones NEON apropiadas mediante el compilador son designadas por ARM como NEON *Intrinsincs*. Estas funciones, así como los tipos de datos descritos anteriormente proveen acceso a la funcionalidad de bajo nivel desde lenguaje C o C++. Esto permite crear variables C con asignación directa en los registros NEON, aunque deja al compilador el control del reparto y manejo entre los mismos. Además, el compilador puede optimizar el código referente a las *Intrinsincs* como código C/C++ normal, reemplazando partes del mismo por las secuencias de instrucciones ensamblador más eficientes posibles.

Cabe destacar que las NEON *Intrinsincs* son parte del ARM *Application Binary Interface*(ABI), por lo que se asegura la portabilidad entre el compilador ARM y GCC.

Las NEON *Intrinsincs* están definidas en el fichero cabecera `arm_neon.h`.

En la Tabla 3.2.2, se pueden apreciar un listado de funciones empleadas ya sea en nuestro código desarrollado como en el código de simulación *Beamforming* aportado.

Instruction	Description
`vmovq_n_s32(value)`	Initialize `int32x4_t` vector registers
`vmovq_n_s64(value)`	Initialize `int64x2_t` vector registers
`vmovq_n_f32(value)`	Initialize `flo32x4_t` vector registers
`vmov_n_s16(value)`	Initialize `int16x4_t` vector registers
`vmov_n_s32(value)`	Initialize `int32x2_t` vector registers
`vld1_s16(*data)`	Load `int16x4_t` vector registers
`vld1_s32(*data)`	Load `int32x4_t` vector registers
`vld1q_f32(*data)`	Load `flo32x4_t` vector registers
`vmls_s16(dest,data1,data2)`	Vector multiply subtract, `int16x4_t` dest. vector registers
`vmls_s32(dest,data1,data2)`	Vector multiply subtract, `int32x2_t` dest. vector registers
`vmlsq_f32(dest,data1,data2)`	Vector multiply subtract, `flo32x4_t` dest. vector registers
`vmlal_s16(dest,data1,data2)`	Vector multiply accumulate long, `int32x4_t` dest. vector registers
`vmlal_s32(dest,data1,data2)`	Vector multiply accumulate long, `int64x2_t` dest. vector registers
`vmlaq_f32(dest,data1,data2)`	Vector multiply accumulate, `flo32x4_t` dest. vector registers
`vst1_s16(dest,source)`	Store a single vector into memory, `*int16_t[4]` dest. registers
`vst1_s32(dest,source)`	Store a single vector Store a single, `*int32_t[4]` dest. registers
`vst1q_f32(dest,source)`	Store a single vector into memory, `*flo32_t[4]` dest. registers
`vget_high_s32(reg)`	Splitting vector registers, `int32x2_t` return vector registers
`vget_high_f32(reg)`	Splitting vector registers, `flo32x2_t` return vector registers
`vget_low_s32(reg)`	Splitting vector registers, `int32x2_t` return vector registers
`vget_low_f32(reg)`	Splitting vector registers, `flo32x2_t` return vector registers
`vpadd_s32(data1,data2)`	Pairwise add, `int32x2_t` dest. vector registers
`vpadd_f32(data1,data2)`	Pairwise add, `flo32x2_t` dest. vector registers
`vget_lane_s32(reg,pos)`	Extract lanes from a vector, `int32_t` dest. registers
`vget_lane_s64(reg,pos)`	Extract lanes from a vector, `int64_t` dest. registers
`vget_lane_f32(reg,pos)`	Extract lanes from a vector, `flo32_t` dest. registers

Tabla 3.2.2: *Algunas de las funciones NEON Intrinsics utilizadas y una breve descripción.*

3.3. HARDWARE NVIDIA

El hardware objetivo para el que se ha desarrollado y sobre el que se ha trabajado gran parte del proyecto es la placa Jetson TK1 de NVIDIA que puede verse en la Figura 3.2.

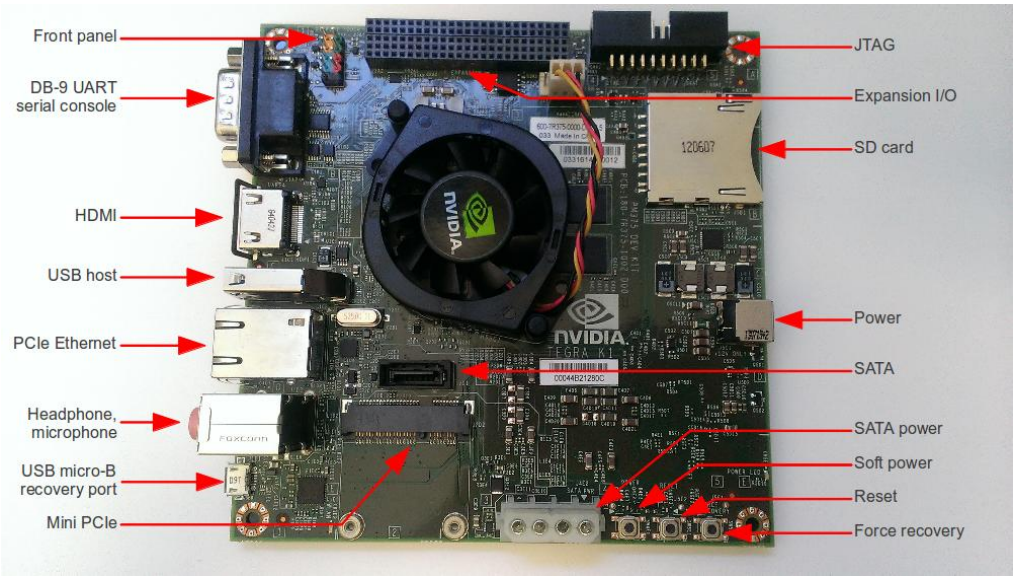


Figura 3.2*: NVIDIA Jetson TK1.*

Se detallan a continuación sus componentes y características [24]:

- **Dimensiones:** 5" x 5" (127mm x 127mm).
- **Tegra K1 SOC** (CPU+GPU+ISP en un chip, consumo entre 1 y 5 Watts):
  - **GPU:** NVIDIA Kepler "GK20a" GPU con 192 SM3.2 CUDA cores (hasta 326 GFLOPS).
  - **CPU:** NVIDIA "4-Plus-1" 2.32GHz ARM quad-core Cortex-A15 CPU con Cortex-A15 (*battery-saving shadow-core).*
- **DRAM:** 2GB DDR3L 933MHz EMC x16 de 64bits.
- **Almacenamiento:** 16GB *fast* eMMC 4.51 (routed to SDMMC4)
- Una entrada/salida para**:**
  - **mini-PCIe**.
  - **SD/MMC card**.
  - **USB 3.0**.
  - **USB 2.0**: micro-AB hembra (para conectar al PC, aunque puede ser usado como un USB 2.0 con el adaptador correspondiente: micro-B male to female Type-A adapter.
  - **Salida HDMI**.
  - **RS232**.
  - **Audio**: conectores de 3.5mm tanto para salida (p.ej. altavoces) como entrada (p.ej. micrófono) a través del códec ALC5639 Realtek HD Audio.
  - **Ethernet:** puerto LAN RTL8111GS Realtek 10/100/1000Base-T Gigabit.
  - **SATA:** puerto SATA3 con soporte para discos de 2.5" y 3.5".

- **JTAG:** puerto debug de 2x10-pin 0.1".
- **Power:** 12V DC barrel power jack and a 4-pin PC IDE power connector, using AS3722 PMIC
- **Ventilador:** funcionando a 12V (para permitir de forma segura sobrecargas de trabajo intensas, pero puede ser reemplazado por un disipador de calor).

Las siguientes señales están disponibles a través del puerto de expansión:

- **Camera ports:** 2 fast CSI-2 MIPI camera ports (one 4-lane and one 1-lane).
- **LCD port:** LVDS and eDP Display Panel.
- **Touchscreen ports:** Touch SPI 1 x 4-lane + 1 x 1-lane CSI-2.
- **UART**.
- **HSIC**.
- **I2C:** 3 ports.
- **GPIO:** 7 x GPIO pins (1.8V). Camera CSI pins can also be used for extra GPIO if you don't use both cameras.

Indicadores en el panel de conectores frontal:

- Verde - Power LED.
- Naranja - HDD LED.
- Rojo - Power Button.
- Violeta / Azul - Reset Button.

APIs de aceleración Hardware soportadas:

- **CUDA** 6.0 .
- **OpenGL** 4.4 .
- **OpenGL ES** 3.1 .
- **OpenMAX IL multimedia codec**.
- **NPP** (CUDA optimized NVIDIA Performance Primitives).
- **OpenCV4Tegra** (NEON + GLSL + quad-core CPU optimizations).
- **VisionWorks**.

Como vemos, la placa Jetson TK1 está capacitada con el ARM Cortex-A15 MPCore más un quinto procesador Cortex-A15 de características limitadas para la salvaguarda de batería en estados ociosos del sistema. Junto a estos, se dispone una GPU de 192 cores CUDA de la propia NVIDIA siguiendo una microarquitectura denominada como Kepler. Todo ello bajo un mismo chip bautizado como Tegra TK1, que podemos encontrar en algunos productos del mercado como la tablet NVIDIA SHIELD y Google Nexus 9 o en el portátil Acer Chromebook 13.

3.4. SOFTWARE

En los siguientes subapartados se listan: los lenguajes de programación, compiladores, librerías, herramientas y programas que hemos utilizado para el desarrollo del trabajo acompañados de una leve explicación.

3.4.1. LENGUAJES DE PROGRAMACIÓN, LIBRERÍAS Y COMPILADORES

Como hemos mencionado al comienzo del trabajo, utilizamos el lenguaje de programación C convirtiendo los Algoritmos desde código **MATLAB**, con el fin de emplear librerías de altas prestaciones y otras herramientas para optimizar el proceso reflejado en el apartado Algoritmo *QRupdating*.

Para la compilación del código traducido a C, hacemos uso del compilador de GNU `gcc`, compilador común en los sistemas operativos basados en Unix como el que opera el Jetson. Empleamos las opciones y parámetros adecuados para el uso de las siguientes librerías:

- **BLAS** [25] (*Basic Linear Algebra Subprograms*): Librería que provee rutinas que realizan operaciones básicas de álgebra lineal numérica sobre vectores y matrices. Las rutinas de esta librería están agrupadas en tres niveles: BLAS1, que contiene operaciones de coste lineal sobre vectores unidimensionales; BLAS2, que contiene operaciones de coste cuadrático que involucran matrices y vectores; y BLAS 3, para operaciones de coste cúbico sobre matrices.
- **LAPACK** [26] (*Linear Algebra PACKage*)*:* Librería desarrollada para resolver problemas de álgebra lineal numérica tales como la resolución de sistemas de ecuaciones lineales, resolución de sistemas de mínimos cuadrados, resolución problemas de valores propios y descomposición en valores singulares. Las rutinas de LAPACK utilizan llamadas a las rutinas de BLAS simplificando su implementación y aprovechando la eficiencia de BLAS en caso de que se disponga de una implementación optimizada.
- **ATLAS** [27] (*Automatically Tuned Linear Algebra Software*): No se trata de una librería propiamente dicha. Su función consiste en permitir configurar de forma empírica las librerías BLAS y LAPACK a las características del hardware sobre el que se han instalado, maximizando el rendimiento entre ambas y ayudando a reducir el tiempo computacional de nuestras aplicaciones.

Mencionar que el paquete ATLAS, ya se encontraba instalado en el Jetson conforme a [13, pág.20], y las librerías BLAS y LAPACK ya descargas.

A la hora de aprovechar el código de simulación *Beamforming* aportado por Francisco J. Alventosa, se hace uso del compilador propietario de NVIDIA `nvcc`, con las directivas y parámetros correspondientes para trabajar con las librerías ya citadas y dos más:

- **PLASMA** [28] (*Parallel Linear Algebra for Scalable Multi-core Architectures*): Es una implementación mejorada de la librería LAPACK, que permite explorar las características de las arquitecturas multi-núcleo en las funciones contenidas en ellas.

- **CUBLAS** [29] (*CUda Basic Linear Algebra Subroutines*): Librería que implementa las rutinas de BLAS pero empleando la GPU como hardware para los cálculos.

Estas dos últimas librerías no las hemos manejado directamente nosotros, sino que se usan en la parte de la implementación del código de simulación aportado.

3.4.2. HERRAMIENTAS DE PARALELIZACIÓN DE DATOS Y CONCURRENCIA

Para el uso de la funcionalidad de los componentes SIMD de cada procesador, esto es, la tecnología NEON y sus NEON *Intrinsics,* se utiliza el fichero cabecera `arm_neon.h` y se marcan las opciones con el compilador `gcc` para indicar al procesador que vamos a utilizar dicha característica. Es necesario utilizar el compilador `gcc` para crear el código ejecutable capaz de utilizar dicha tecnología. En el caso de querer utilizarlo junto con el compilador `nvcc` ,en primer lugar, habrá que crear un código objeto con `gcc` y enlazarlo a posteriori con el otro compilador.

El último paso del proceso de optimización realizado pasa por la utilización de la herramienta de computación paralela OpenMP para la paralelización del código y uso de las NEON de todos los procesadores de forma concurrente. Para ello, también será necesario especificarlo en la orden de compilación o fichero `Makefile`, para cualquiera de los dos compiladores mencionados.

OpenMP es una interfaz de programación de aplicaciones (API), para la programación multiproceso de memoria compartida. Esta permite añadir concurrencia a los programas escritos en C, C++ y Fortran, sobre las base del modelo de ejecución *fork-join*. Disponible en muchas arquitecturas, se compone de directivas de compilador, rutinas de biblioteca, y variables de entorno, con el objetivo mencionado. Es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible.

Se muestra en la Figura 3.3 el contenido de uno de los ficheros `Makefile` a partir del cual hemos compilado el programa resultado de la traducción del código MATLAB, utilizando la funcionalidad de las NEON y OpenMP, antes de ser "incrustado" en el programa de simulación *Beamforming*.

```
#*******************************************#
# QR Updating Optimization for Beamforming    #
# Autor: Enric Ruiz Andrés                    #
#*******************************************#

CC=gcc
CFLAGS=-DVERB -DDEBUG -DDUMPA -DDUMPQR2 -DDEBUGOMP -I/usr/local/openblas-0.2.19/include

BLAS=-L/usr/local/openblas-0.2.19/lib -lopenblas -lm
NEONFLAGS=-march=armv7 -mfpu=neon -L/usr/local/openblas-0.2.19/lib -lopenblas -lm
OMP=-fopenmp

PROGRAMNAME1=cblas_srotN
PROGRAMNAME=qrupdating

all: $(PROGRAMNAME)

$(PROGRAMNAME): $(PROGRAMNAME).c Makefile
        $(CC) -O3 -c $(PROGRAMNAME1).c -o $(PROGRAMNAME1).o $(NEONFLAGS)
        $(CC) $(OMP) $(CFLAGS) $(PROGRAMNAME).c -o $(PROGRAMNAME) $(PROGRAMNAME1).o $(BLAS)

clean:
        rm -f $(PROGRAMNAME) $(PROGRAMNAME1)
```

Figura 3.3*: Contenido de uno de los ficheros Makefile utilizados con las opciones, directivas y parámetros adecuados para compilar con las funcionalidades que nos aportan CBLAS, NEON y OpenMP.*

3.4.3. PROGRAMAS

Para tratar los Algoritmos aportados por el tutor del proyecto, escritos en **MATLAB**, se ha hecho uso del programa **Octave**, al ser código totalmente compatible.

El acceso al Jetson TK1 se ha hecho remotamente, haciendo uso del programa **vpnc** [30] para conectarse a la red interna de la U.P.V. (Universidad Politécnica de Valencia), para luego conectarse vía **ssh** [31] al dispositivo en cuestión. El acceso al Jetson queda restringido al uso por parte de alumnos de la U.P.V. que tengan una cuenta en el mismo, obviamente.

La programación en C se ha realizado mediante el editor de texto **Vim** [32]. Por un lado porque es la opción más simple por lo explicado en el párrafo anterior y por otro lado porque la longitud de código a tratar no ha sido demasiado extensa y la depuración del código ha podido hacerse explícitamente y mediante comparación directa con el código MATLAB.

Para el desarrollo de la memoria y la presentación se ha utilizado el paquete de ofimática **LibreOffice** [33], si bien para las gráficas se ha utilizado el programa **gnuplot** [34].

Capítulo 4 Aplicación y Análisis

En este capítulo se aborda el proceso seguido para la optimización del Algoritmo *QRupdating* partiendo de los códigos correspondientes, escritos en MATLAB.

Este algoritmo, como se ha explicado de forma analítica en su capítulo, calcula la descomposición QR de una matriz X dada y aplica una serie de actualizaciones en ambas matrices (Q y R), de modo que no sea necesario volver a calcular una nueva factorización QR completa, sino aquella que implica que solo se modifiquen f filas. Estas actualizaciones se basan en la eliminación y adición de nuevas filas.

Al proceso para realizar lo descrito, se le denota como *QRupdating* y consta de dos partes o fases: *QRupdate: Eliminar Filas* y *QRupdate: Añadir Filas* que corresponden con el Algorimo 1 y el Algoritmo 2, respectivamente.

La mejora de prestaciones se ha hecho de forma escalonada, de manera que cada evolución de la optimización se contempla como una versión en sí. Una vez comprobado el correcto funcionamiento de cada una, se introducen en el código de simulación *Beamforming* según las recomendaciones sugeridas por Francisco J. Alventosa y Pedro Alonso, y se pasa a la toma de tiempos para la comprobación y comparación de las distintas mejoras introducidas.

Son tres implementaciones las que se detallan a continuación, si bien han habido versiones preliminares que no se exponen aquí, pero que sí veremos reflejadas en el punto 4.4. Comparativas. Esto se comprende una vez analizada la primera implementación, donde se pone de manifiesto la importancia del acceso eficiente a memoria para las operaciones más costosas de *QRupdating*.

Esta primera versión consiste en la traducción del código de MATLAB a C y la propia implementación a partir del análisis a razón del estudio para un acceso eficiente a memoria.

En la segunda implementación se hace uso de la tecnología NEON de ARM que nos brindan los procesadores ARM Cortex-A15 con la base de mejora que presentamos en la primera versión.

La última versión consiste en el uso de la herramienta OpenMP para maximizar, con los cuatro *cores* que alberga la placa Jetson TK1, las mejoras que nos aportan las anteriores implementaciones.

A continuación se detallan los pasos dados y la aportación que propone cada versión.

4.1. VERSIÓN con BLAS/LAPACK

El primer paso es convertir el código MATLAB a lenguaje C haciendo uso de las funciones BLAS/LAPACK para el cálculo de una necesaria primera descomposición QR, la generación de matrices de Givens y la aplicación de sus rotaciones teniendo en cuenta, entre otras cosas, las diferencias de indexación entre un lenguaje y otro, así como el almacenamiento por columnas de las matrices para el uso correcto de las rutinas BLAS/LAPACK. En las siguientes Fichas se especifican qué funciones se han empleado.

Función	`int sgeqrf_(integer *m, integer *n, real *R, integer *lda, real *tau, real *work, integer *lwork, integer *info);`
Descripción	Realiza el cálculo de la factorización QR. En la entrada, *R* es un vector matriz de *m*×*n* y supone la matriz *X* a factorizar. En la salida, contiene en la parte superior de la diagonal y sobre ésta, la parte triangular superior de *R*, y en los elementos por debajo de la diagonal, junto con el vector *tau*, representa la matriz ortogonal *Q* como producto de los reflectores elementales (almacenados en *tau*). Se determina el *leading dimension* con el cuarto parámetro. El vector **tau** (parámetro de salida) son los factores escalares de los reflectores elementales que usa la rutina para el cálculo de la descomposición QR mediante reflexiones Householder. El vector **work** (parámetro de salida) representa el espacio de trabajo que necesita la función para el cálculo de la descomposición, siendo *lwork* su dimensión. El parámetro de salida *info* indica si la rutina ha tenido éxito en el cálculo.

Ficha 4.1

Para no sobrescribir la matriz origen X^T, haremos la llamada a **`sgeqrf`** con una copia de la misma, conscientes de que la parte superior de la matriz salida de la rutina será R^T.

Función	`int sorgqr_(integer *m, integer *n, integer *k, real *Q, integer *lda, real *tau, real *work, integer *lwork, integer *info);`
Descripción	Genera la matriz *Q* de tamaño *m*×*n*, con columnas ortogonales que son definidas como las primeras *n* columnas del producto de *k* reflectores elementales de orden *m* (en nuestro caso, *k*=*n*) . En la entrada Q debe contener en la *i*-ésima columna, el vector que define los reflectores elementales *H(i)*, para *i=1,2,...,n* , tal como nos devuelve la rutina **`sgeqrf`**. Se determina el *leading dimension* con el quinto parámetro. El vector **tau** (parámetro de entrada) debe contener los factores escalares de los reflectores elementales. El vector **work** (parámetro de salida) representa el espacio de trabajo que necesita la función para el cálculo de *Q*, siendo *lwork* su dimensión. El parámetro de salida *info* indica si la rutina ha tenido éxito en el cálculo.

Ficha 4.2

Con estas dos funciones suplimos eficientemente la instrucción MATLAB correspondiente a la descomposición QR.

La llamada a **`sorgqr`** se produce tras el uso de **`sgeqrf`**, como se nos sugiere en la descripción de la Ficha 4.2. Seguiremos la misma técnica empleada antes, salvaguardando ahora la matriz R^T para no sobrescribirla en la llamada, copiándola sobre la que será la matriz Q^T.

A continuación se indican las funciones utilizadas para la generación de matrices de Givens y la aplicación de las mismas.

Función	`void cblas_srotg (float *p, float *q, float *c, float *s);`
Descripción	Proporciona el coseno *c* y el seno *s* característicos de una matriz de Givens para dos elementos en coma flotante dados: *a* y *b*. También aplica la rotación de Givens sobre estos dos elementos.

Ficha 4.3

Función	`void cblas_srot (const int nE, float *A, const int incA, float *B, const int incB, const float c, const float s);`
Descripción	Aplica las rotaciones de Givens sobre *nE* elementos en coma flotante en las matrices *A* y *B*. Para indicar el *stride* o distancia entre los elementos de cada matriz se hace uso de los parámetros de entrada *incA* e *incB*. Los parámetros de entrada *c* y *s* son los valores de la matriz de Givens para el coseno y seno calculados previamente con **`cblas_srotg`**.

Ficha 4.4

Estas dos funciones reemplazan en nuestro código en C a las funciones *rotg* y *rot* de las que hacen uso los Algoritmos originales en MATLAB.

La llamada a **`cblas_srotg`** se producirá sobre la matriz Q^T si estamos en la fase *QRupdate: Eliminar Filas* o sobre la matriz R^T si nos encontramos en *QRupdate: Añadir Filas*.

Destacar que las matrices siguen una disposición *Column Major Order*, las cuales se construyen en base a la descomposición QR de la matriz X^T, la cual se forma directamente así según lo expuesto en [13]. Este hecho hace que nos beneficiemos en el cálculo de las rotaciones de Givens sobre la matriz Q^T, la cual, al estar almacenada por columnas se tiene un *stride* contiguo, es decir, los datos de los vectores columna **x** e **y** con los que operar, se encuentran de forma continua conforme al recorrido del proceso. Sin embargo, la aplicación de las rotaciones de Givens sobre R^T no tiene un *stride* contiguo, sino de *m* elementos, siendo así por la naturaleza del cálculo de dichas rotaciones en esta disposición de la matriz.

Se considera pues, al hilo de lo expuesto en el párrafo anterior y a raíz de la mejora en prestaciones que supone un *stride* contiguo, con acceso continuo y adyacente de los datos en memoria, que debe trasponerse la matriz R^T (véase el análisis de la Gráfica 4.4.1.1 y Gráfica 4.4.1.2, correspondiente a la comparativa entre la aplicación de las rotaciones de Givens sobre Q^T y Q , con uno u otro tipo de acceso). Se estima así, que trabajando sobre Q^T y la traspuesta de R^T, esto es, R, se consiguen los resultados de mayores prestaciones como veremos más adelante en las Compartivas.

Continuando con el desglose de funciones utilizadas en esta primera versión, para la traslación de elementos que ocurre en la parte final de *QRupdate:Eliminar Filas* (punto 7. y punto 8. de la secuencia) y que podemos ver reflejado en las líneas 18 a 22 del Algoritmo 1, así como para el resguardo de matrices y algunos volcados de datos de una localización de la memoria a otra, referentes a la adecuación de las matrices en cada iteración, se ha utilizado la función **`cblas_scopy`**.

Función	`void cblas_scopy (const int nE, float *a, const int incA, float *b, const int incB);`
Descripción	Copia *nE* elementos en coma flotante desde el vector **a** al vector **b**. Para indicar el *stride* o distancia entre los elementos de cada *array* se hace uso de los parámetros de entrada *incA* e *incB*.

Ficha 4.5

La comprobación de la correcta traducción desde un lenguaje a otro se ha conseguido cotejando los resultados obtenidos en una y otra implementación. Esto se ha hecho en todas y cada una de las diferentes versiones. Una de las funciones construidas con tal propósito se muestra en el Código 4.1.

```
1  void volcarAaFichero(float *A,unsigned int m,unsigned int n,unsigned int dimR){
2          FILE *fp;
3          int i,j,index=0;
4
5          fp=fopen("AtoMATLAB.m", "w+");
6          char *aEscribir=(char*)calloc(dimR*m,sizeof(char));
7          char tmp[]="0.0000";
8
9          strcat(aEscribir, "A=[");
10         for(i=0;i<m;i++){
11           if(i)//distinto de cero
12             strcat(aEscribir,";");
13           for(j=0,index=i;j<n;j++){
14             sprintf(tmp,"%.4f",A[index]);
15             strcat(aEscribir," ");
16             strcat(aEscribir,tmp);
17             index+=m;
18           }
19         }
20         strcat(aEscribir,"]\n\0");
21
22         fwrite(aEscribir,1,strlen(aEscribir),fp);
23         fclose(fp);
24 }
```

Código 4.1: *Función para volcar sobre un fichero los valores de los elementos de una matriz m×n con la nomenclatura correcta para cargar en el programa Octave.*

En esta conversión se persigue una programación inteligente sobre los mismos datos en memoria siempre que sea posible, siendo éstos los mínimos posibles y evitando en la medida de lo posible accesos en fallo a *cache*.

4.2. VERSIÓN con NEON INTRINSICS

Analizando el código y el proceso *QRupdating* en su conjunto, la operación más costosa y que desearíamos optimizar se trata sin duda de la relativa a la aplicación de las rotaciones de Givens en cada iteración. Además, es claramente un objetivo ideal para la vectorización de datos, es decir, la función **`cblas_srot`** posee unas características apropiadas, como se explica a continuación, para ser reescrita aprovechando las instrucciones SIMD que nos ofrece la tecnología NEON por medio de las funciones NEON *Intrinsics*.

La operación que realiza la función **`cblas_srot`** se define matemáticamente según las siguientes ecuaciones:

$$x_i := c * x_i + s * y_i \quad , \qquad \text{(EQ 4.1)}$$

$$y_i := c * y_i - s * x_i \quad . \qquad \text{(EQ 4.2)}$$

Teniendo a x_i e y_i como los *i*-ésimos elementos de los vectores columna **x** e **y** compuestos por *nE* elementos (mírese la Ficha 4.4) de las matrices Q^T o R (según a qué matriz apliquemos las rotaciones). Las variables *c* y *s* corresponden con el coseno y seno de la matriz de Givens calculada previamente con **`cblas_srotg`** (mírese la Ficha 4.3).

Así pues, se paraleliza el cálculo de **x** e **y** mediante la vectorización con las funciones NEON *Intrinsics* apropiadas para la suma y resta de productos que sostienen las ecuaciones definidas. De tal modo, conseguimos el resultado de cuatro operaciones, concernientes a x_i hasta x_{i+4} para la EQ 4.1 en un mismo instante de tiempo, y para y_i hasta y_{i+4} para la EQ 4.2 , en otro único instante de tiempo. Con ello, efectivamente, podríamos acelerar teóricamente hasta por cuatro estos cálculos. En la práctica no será así, como analizaremos más adelante en 4.4.3 BLAS/LAPACK vs. NEON.

Se muestra en el Código 4.2 la implementación de la rutina **`cblas_srotN`.**

```
1  #include <arm_neon.h>
2
3  void cblas_srotN(const int N, float *X, float *Y,const float *c,const float *s){
4    float stemp;
5    int i;
6    float32x4_t cN, sN, loadXi, loadYi;
7
8    cN = vmovq_n_f32(*c);
9    sN = vmovq_n_f32(*s);
10
11   for(i=0;i+4<N;i=i+4){
12     loadXi = vld1q_f32(&X[i]);
13     loadYi = vld1q_f32(&Y[i]);
14
15     vst1q_f32(&X[i],vmlaq_f32(vmulq_f32(cN,loadXi),sN,loadYi));
16     vst1q_f32(&Y[i],vmlsq_f32(vmulq_f32(cN,loadYi),sN,loadXi));
17   }
18
19   for(;i<N;i++){
20       stemp=*c*(X[i]) + *s*(Y[i]);
21       Y[i]=*c*(Y[i]) - *s*(X[i]);
22       X[i]=stemp;
23   }
24 }
```

Código 4.2: *Implementación de la rutina* **`cblas_srot`** *para el uso de la tecnología NEON en ARMv7.*

Como se adivina, en esta implementación no existe el parámetro para indicar el *stride* entre los elementos de los vectores columna de Q^T o R a tratar porque para ambos casos la distancia entre los datos a tratar es la unidad, es decir, el *stride* es contiguo como se ha explicado en la versión anterior. Por otro lado, solo tiene sentido aplicar las NEON *Intrinsics* cuando el *stride* es uno, porque solo de esta manera podemos cargar en los registros vectoriales cuatro datos consecutivos sin perder tiempo ni prestaciones. Por último, para los elementos restantes que no puedan ser tratados con instrucciones NEON, se operan secuencialmente como en la rutina original. Esto último se puede ver a partir de la línea 19 del Código 4.2.

A continuación se muestran las funciones NEON *Intrinsics* utilizadas, así como una breve descripción del uso en la Tabla 4.2.1:

NEON *Intrinsic*	Descripción del uso
`vmovq_n_f32``(value)`	Inicializa un vector de tipo `float32x4_t` con el valor del coseno, y otro con el valor del seno. Carga el mismo valor en los cuatro carriles del registro vectorial en cuestión.
`vld1q_f32``(*data)`	Carga un vector de tipo `float32x4_t` a partir de cuatro datos de tipo `float`. Cada dato se almacena en un carril del registro vectorial.
`vmulq_f32``(dataA, dataB)`	Realiza la operación *Vr[i]=Va[i]*Vb[i]* de dos vectores tipo `float32x4_t`, donde cada dato de cada carril del registro vectorial `dataA` se multiplica con su homónimo del otro registro vectorial `dataB`.
`vmlaq_f32``(dataA, dataB, dataC)`	Realiza la operación *Vr[i]=Va[i] + Vb[i]*Vc[i]*.
`vmlsq_f32``(dataA, dataB, dataC)`	Realiza la operación *Vr[i]=Va[i] – Vb[i]*Vc[i]*.
`vst1q_f32``(dest, source)`	Guarda el vector `source` de tipo `float32x4_t` que contiene el resultado de x_i a x_{i+4} y de y_i a y_{i+4}, acorde a EQ 4.2.1 y EQ 4.2.2, en los vectores `dest` **x** e **y** de tipo `float`.

Tabla 4.2.1

Podemos ver que las instrucciones con las que se trata, trabajan con los registros *quadcore* de 128 bits, siendo del tipo `float32x4_t` los datos albergados en ellos, concorde a lo expuesto en el apartado Tipos de Datos de la Tecnología NEON en el Capítulo 3: Descripción del entorno.

4.3. VERSIÓN con OpenMP

La versión con OpenMP no es fácil de implementar porque como se puede apreciar en las secuencias correspondientes tanto a la fase *QRupdate:Eliminar Filas* como *QRupdate: Añadir Filas*, en cada iteración de uno u otro proceso se modifican datos de la anterior iteración. Nos encontramos pues, ante Algoritmos inherentemente secuenciales por la condición descrita, donde existen dependencias de datos.

La dificultad que comportan estas dependencias no son una imposibilidad para el uso de la herramienta de programación concurrente OpenMP, pero sí una restricción.

Tenemos que adaptar y reorganizar el código relativo al Algoritmo 1 y al Algoritmo 2, para poder paralelizar la aplicación de las rotaciones de Givens que, como hemos visto ya en la versión anterior, se trata del proceso más pesado a nivel computacional. Esto se consigue siguiendo una estrategia de asignación por bloques para cada hilo, como veremos más adelante.

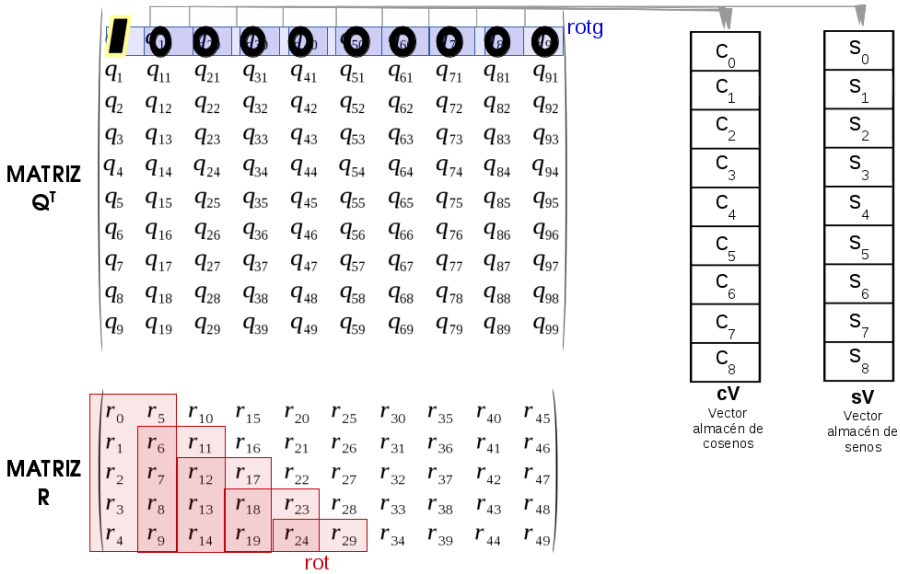
Siendo la matriz Q^T, la de mayor dimensión, y a expensas de saber si la paralelización es efectiva en algún caso, se ha optado por paralelizar solamente la aplicación del cálculo de las rotaciones sobre dicha matriz.

Resaltar que esta versión hace uso de la mejora que nos aporta la versión anterior. Al tener cada *core* su propia unidad de vectorización NEON, cada núcleo trabajará independientemente con su componente SIMD sobre el bloque de datos asignado.

La estrategia seguida para abordar la programación con el objetivo de utilizar las directivas OpenMP (de aquí en adelante, OMP) se visualiza en la siguiente secuencia, tomando para ello un ejemplo práctico con $Q^T \in \mathbb{R}^{10\times10} \wedge R \in \mathbb{R}^{5\times10}$, con una sola fila a actualizar (f=1), y dos hilos de ejecución.

En la fase *QRupdate: Eliminar Filas*:

1. Se calculan todas las matrices de Givens por duplas de datos a través de todas las columnas de Q^T a lo largo de la primera fila, aplicando las rotaciones sobre R cuando llegamos a aquellas que afecten a la diagonal superior de la que sería R^T. Al mismo tiempo, vamos guardando los senos y cosenos para cada matriz de Givens calculada.

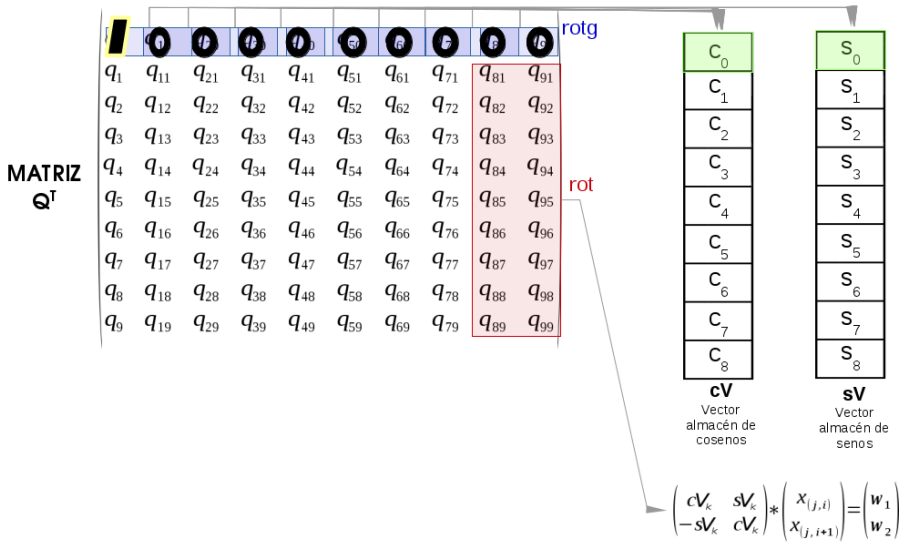


Los vectores que nos sirven de almacén **cV** y **sV** tendrán un tamaño variable para cada fila a

eliminar, de modo que su tamaño atiende a (*m-1)-(j-1)*, siendo *m* el número de columnas y *j* tomando valores desde uno hasta *f* filas a actualizar. En el ejemplo, el tamaño será de nueve elementos, puesto que estamos en la primera (y última) iteración para este ejemplo práctico, por lo que *j*=1.

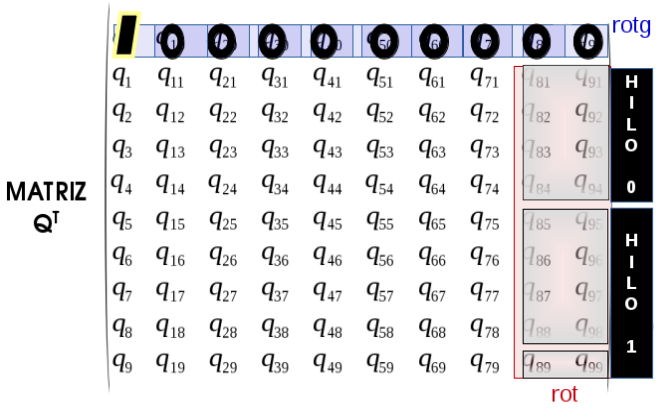
Nótese además, que el proceso en sí se encuentra traspuesto respecto del procedimiento original ya que la disposición de las matrices es por columnas en todas estas versiones implementadas, mientras que en el original es por filas.

2. Se procede ahora al cálculo de las rotaciones de Givens sobre el resto de la matriz Q^T, siguiendo el mismo recorrido que en el cálculo de las matrices de Givens, es decir, de forma iterativa desde la columna *m* hasta la primera.



Para el cálculo en cada iteración se utilizan los cosenos y senos almacenados en **cV** y **sV**. Aquí, al ser la primera iteración, se usan el coseno c_0 y el seno s_0.

3. En aras de utilizar OMP, se subdividen los vectores columna involucrados de manera que se forma un bloque de datos para que cada núcleo lo compute paralelamente.



El tamaño de bloque vendrá determinado por el reparto equitativo de datos albergados en los vectores entre el número de hilos. En el ejemplo:

$$lBlock = \left\lfloor \frac{(m-j)}{np} \right\rfloor = \left\lfloor \frac{9}{2} \right\rfloor = 4 \quad .$$

Si existe un resto en la división mostrada, el bloque de datos restante que representa es tratado por el hilo con el identificador más alto. En el ejemplo, el hilo uno.

Para que cada hilo acceda de forma correcta a su bloque de datos, se le asigna un nuevo índice para situarse sobre la fila correcta:

$$newJ = j + (th * lBlock) \ .$$

De manera que los índices para los hilos cero e hilo uno en esta primera iteración quedarían para el ejemplo como:

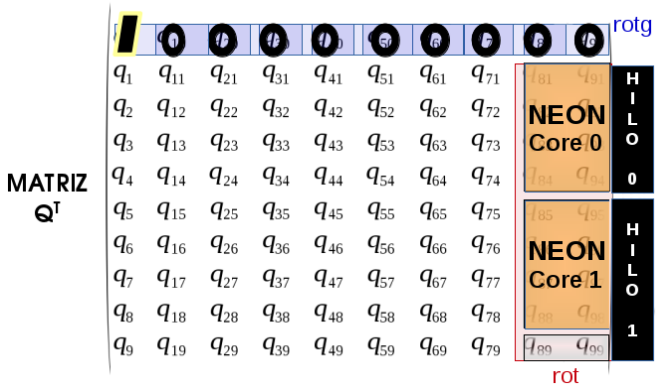
$$newJ_{hilo_0} = 1 + 0 * 4 = 1 \ ,$$

$$newJ_{hilo_1} = 1 + 1 * 4 = 5 \ .$$

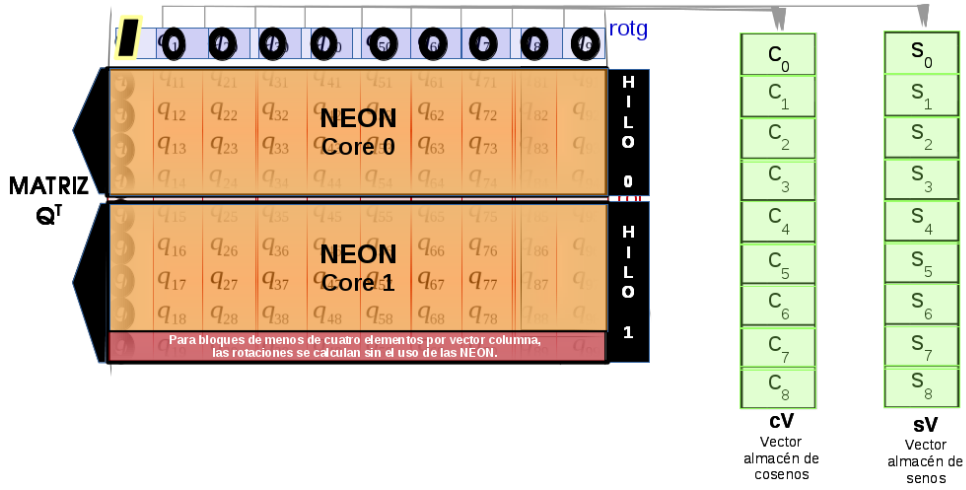
De este modo conseguimos que los hilos trabajen siempre con las mismas filas. Debe ser así porque como ya se ha argumentado, hay dependencias de datos en las sucesivas iteraciones, y esas dependencias ocurren para cada fila de Q^T.

Para aclarar el tratamiento de los índices, cabe indicar que los accesos al elemento q_{81} y q_{91} por parte del hilo cero en esta primera iteración son: *Q(newJ,i-1)* y *Q(newJ,i)*, respectivamente, donde *newJ* atiende a lo expuesto y el índice *i* toma valores desde *m-1* hasta *j*, marcando *j* la iteración sobre la fila a eliminar. En nuestro código se traduce como: `Q[newJ+(i-1)*m]` y `Q[newJ + i*m]`.

4. Cada *core* pondrá en funcionamiento su unidad de vectorización de datos NEON de acuerdo a lo expuesto en la anterior versión, según la implementación mostrada en el Código 4.2:



5. Se aplica el mismo procedimiento para el resto de rotaciones de Givens en las demás iteraciones, aplicando en cada una el seno y coseno pertinente a dicha iteración.



Como puede verse, la última fila es procesada por el hilo uno (el de identificador más alto, a raíz de lo explicado en el punto 3.). Dicha fila alberga los bloques sobrantes del reparto equitativo de *m-j* datos por vector columna para los dos hilos del ejemplo. De tal modo el hilo uno aplicará las rotaciones sin el uso de NEON pues este bloque restante está compuesto por un solo elemento por vector columna pues, *(m-j)%np* = 9%2 =1 . Por tanto, no se tienen los elementos suficientes para aprovechar la funcionalidad NEON ya que son menos de cuatro datos por vector columna, para este ejemplo.

6. Se completa la fase *QRupdate:Eliminar Filas*, recolocando los datos válidos de Q^T y *R* sobre su mismo espacio en memoria:

MATRIZ R

$$\begin{pmatrix} r_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ r_6 & r_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ r_7 & r_{12} & r_{17} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ r_8 & r_{13} & r_{18} & r_{23} & 0 & 0 & 0 & 0 & 0 & 0 \\ r_9 & r_{14} & r_{19} & r_{24} & r_{29} & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

MATRIZ Q^T

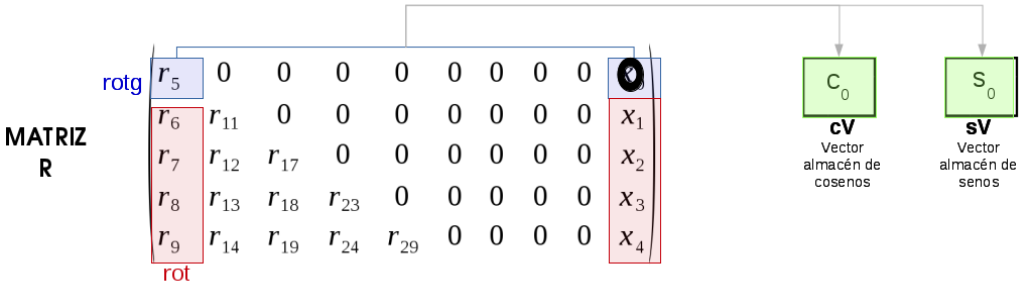
$$\begin{pmatrix} q_{11} & q_{21} & q_{31} & q_{41} & q_{51} & q_{61} & q_{71} & q_{81} & q_{91} & 0 \\ q_{12} & q_{22} & q_{32} & q_{42} & q_{52} & q_{62} & q_{72} & q_{82} & q_{92} & 0 \\ q_{13} & q_{23} & q_{33} & q_{43} & q_{53} & q_{63} & q_{73} & q_{83} & q_{93} & 0 \\ q_{14} & q_{24} & q_{34} & q_{44} & q_{54} & q_{64} & q_{74} & q_{84} & q_{94} & 0 \\ q_{15} & q_{25} & q_{35} & q_{45} & q_{55} & q_{65} & q_{75} & q_{85} & q_{95} & 0 \\ q_{16} & q_{26} & q_{36} & q_{46} & q_{56} & q_{66} & q_{76} & q_{86} & q_{96} & 0 \\ q_{17} & q_{27} & q_{37} & q_{47} & q_{57} & q_{67} & q_{77} & q_{87} & q_{97} & 0 \\ q_{18} & q_{28} & q_{38} & q_{48} & q_{58} & q_{68} & q_{78} & q_{88} & q_{98} & 0 \\ q_{19} & q_{29} & q_{39} & q_{49} & q_{59} & q_{69} & q_{79} & q_{89} & q_{99} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

La diferencia fundamental entre el procedimiento original y la versión detallada en esta secuencia radica en que las rotaciones de Givens se aplican sobre el resto de la matriz Q^T una vez hemos calculado todas las matrices de Givens para una fila, en vez de aplicar las rotaciones por cada matriz de Givens calculada. Esto puede verse reflejado en los puntos 1.) y 2.) de la secuencia mostrada. A partir del punto 3.) de la secuencia en adelante se plasma la estrategia OMP seguida y el uso de las NEON sobre la misma.

En la fase *QRupdate: Añadir Filas:*

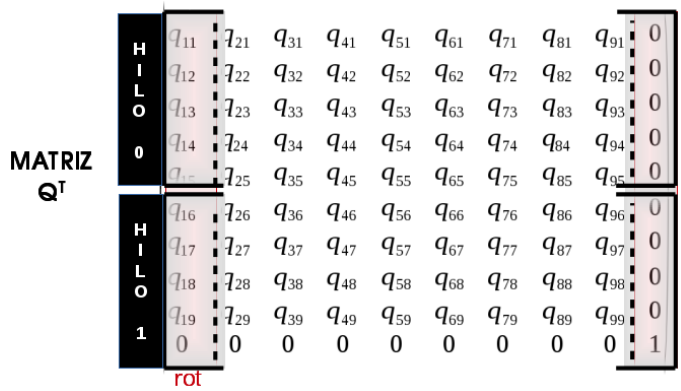
1. Se adhieren *f* filas nuevas desde X^T en la últimas *f* columnas de R .

2. Se procede al cálculo de las matrices de Givens en un proceso iterativo de *f* filas a través de tantas columnas como abarque la diagonal superior de la que sería R^T.



Almacenamos los senos y cosenos concernientes a la matriz de Givens calculada para la única fila añadida y aplicamos las rotaciones sobre los vectores columna correspondientes. El almacén de cosenos **cV** y de senos **sV** en *QRupdate:Añadir Filas* será de tamaño fijo, a diferencia de lo que ocurría en *QRupdate:Eliminar Filas* de esta versión. En el ejemplo práctico, al ser solo una fila a actualizar, solo habrá una iteración sobre el primer componente de la que sería la diagonal superior de R^T, y por lo tanto el tamaño de los almacenes es para un solo coseno en **cV** y para un solo seno en **sV**. Así pues, el tamaño fijo al que se hace referencia vendrá determinado por *f*.

En este mismo paso, se aplican las rotaciones sobre los vectores columna de Q^T correspondientes, y lo haremos con el uso de OMP de acuerdo a la misma estrategia utilizada en la fase previa, pero teniendo en cuenta las diferencias entre esta fase y la anterior.



Como se aprecia en la instantánea, se forman dos bloques de datos, uno para cada hilo, donde el tamaño del bloque viene determinado esta ocasión por:

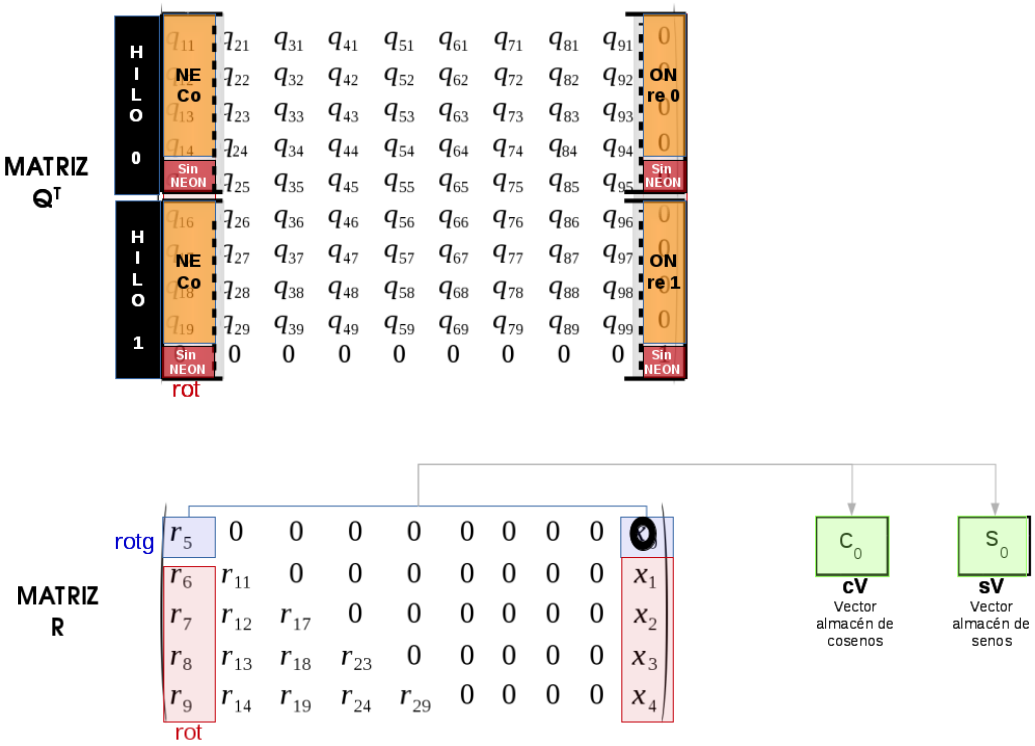
$$lBlock = \left\lfloor \frac{m}{np} \right\rfloor = \left\lfloor \frac{10}{2} \right\rfloor = 5 \quad .$$

La diferencia en el cálculo del tamaño del bloque respecto a la fase anterior reside en que, en cada iteración del proceso son siempre *m* datos sobre los que aplicar las rotaciones en lugar de *m- j* como ocurría en *QRupdate:Eliminar Filas* con OMP. Por este mismo motivo, tampoco se considera la iteración actual para el cálculo del índice que sirve a cada hilo para el acceso correcto a su bloque de datos:

$$newJ = th * lBlock \quad ,$$

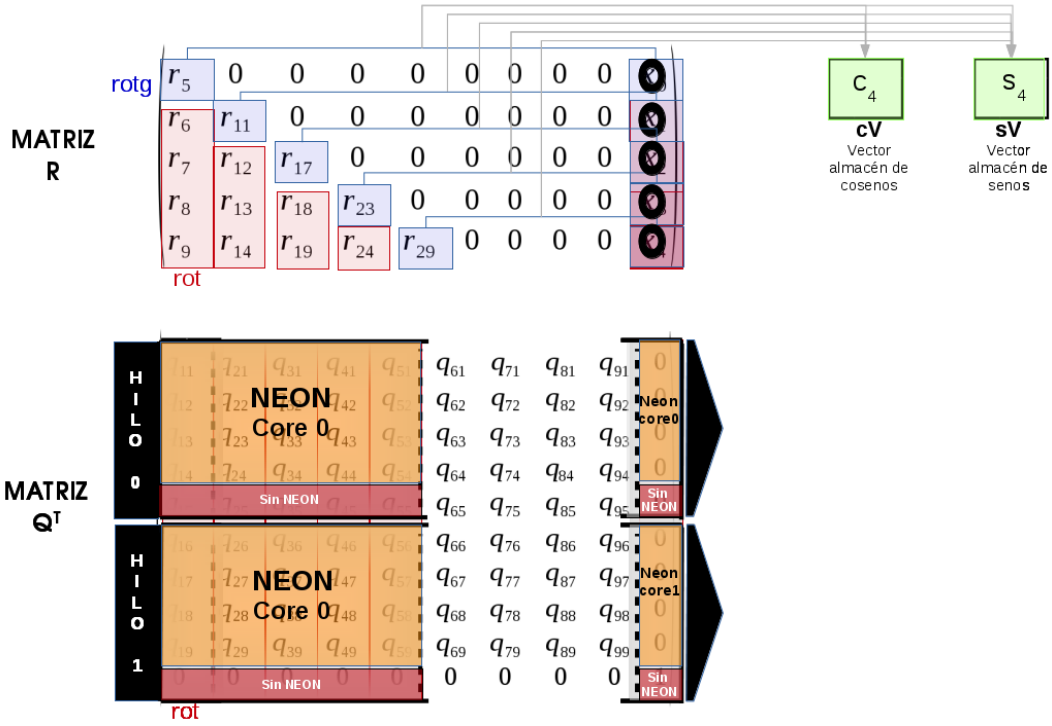
quedando determinado el nuevo índice simplemente por su identificador y el tamaño de bloque.

3. Se hace uso de la tecnología NEON de cada *core* sobre su bloque de datos asignado en aquellas rotaciones concernientes a dos vectores columna de cuatro datos consecutivos (dos registros vectoriales de tipo `float32x4_t` en el contexto de las NEON *Intrinsics*). En el caso de no quedar datos suficientes para conformar grupos de cuatro elementos por vector columna en los bloques de datos asignados a los hilos, estos se procesarán como en la rutina original **`cblas_srot`**, esto es, de forma secuencial sin la vectorización de datos que aporta NEON.



Las rotaciones sobre Q^T se computan a partir de los senos y cosenos guardados en **cV** y **sV**. En el ejemplo práctico, al haber solo una nueva fila añadida, solo se calcula una matriz de Givens y por ende solo hay almacenados un coseno en **cV** y un seno en **sV**. Aquí, y como ya se detalló en el cálculo de Costes, habrá menos resguardos de senos y cosenos porque hay un menor cálculo de matrices de Givens.

4. Se repite el proceso descrito para *n* columnas (aquellas que albergan la que sería la diagonal superior de R^T).



Aquí termina el proceso *QRupdate:Añadir Filas* y por consiguiente el proceso *QRupdating* para *f* filas a actualizar en una iteración con el uso de dos hilos (en el ejemplo) mediante la herramienta de paralelización OMP y el uso de la herramienta de vectorización de datos que nos proporciona la tecnología NEON de ARM en cada uno de sus núcleos.

A continuación se muestran el Código 4.3.1 y el Código 4.3.2 correspondientes a la parte reestructurada y paralelizada con la herramienta OpenMP de *QRupdate: Eliminar Filas* y *QRupdate: Añadir Filas*, respectivamente:

```
138     /*Aplicamos las rotaciones sobre Q con OMP*/
139     unsigned int lBlock; //longitud del bloque a tratar por cada hilo
140     unsigned int rest=(m-j)%nT;
141     lBlock=(m-j)/nT;
142     p=0;
143     #pragma omp parallel private(i) firstprivate(p,cV,sV)
144     {
145       unsigned int thread=omp_get_thread_num();
146       unsigned int newJ=j+(thread*lBlock);
147       if(rest){
148         for(i=mMo;i>=j;i--){
149            cblas_srotN(lBlock,&Q(newJ,i-1),&Q(newJ,i),&cV[p],&sV[p]);
150
151            if(thread==nT-1){ //El último hilo se encarga del resto:
152                              // -> hilo 3, en caso de ser cuatro en total: h0 - h1 - h2 -h3
153             cblas_srotN(rest,&Q(newJ+lBlock,i-1),&Q(newJ+lBlock,i),&cV[p],&sV[p]);
154             }
155
156             p++;
157         }//end for i=(m-1):j
158       }else{
159         for(i=mMo;i>=j;i--){
160           cblas_srotN(lBlock,&Q(newJ,i-1),&Q(newJ,i),&cV[p],&sV[p]);
161           p++;
162         }//end for i=(m-1):j
163       }
164     }//end omp parallel region
165   }//end for j=1:f
166 }//end function ELIMINAR
```

Código 4.3.1: *Paralelización OMP en la fase QRupdate: Eliminar Filas*

```
202     /*Aplicamos las rotaciones sobre Q con OMP*/
203     unsigned int lBlock; //longitud del bloque a tratar por cada hilo
204     unsigned int rest=m%nT;
205     lBlock=m/nT;
206     p=0;
207     #pragma omp parallel private(i) firstprivate(p,cV,sV)
208     {
209       unsigned int thread=omp_get_thread_num();
210       unsigned int newJ=(thread*lBlock);
211       if(rest){
212         for(i=dAux;i<m;i++){
213            cblas_srotN(lBlock,&Q(newJ,j),&Q(newJ,i),&cV[p],&sV[p]);
214
215            if(thread==nT-1){ //El último hilo se encarga del resto:
216                              // -> hilo 3, en caso de ser cuatro en total: h0 - h1 - h2 -h3
217             cblas_srotN(rest,&Q(newJ+lBlock,j),&Q(newJ+lBlock,i),&cV[p],&sV[p]);
218             }
219
220             p++;
221         }//end for i=(m-f):m
222       }else{
223         for(i=dAux;i<m;i++){
224           cblas_srotN(lBlock,&Q(newJ,j),&Q(newJ,i),&cV[p],&sV[p]);
225           p++;
226         }//end for i=(m-f):m
227       }
228     }//end omp parallel region
229   }//end for j=1:n
230 }//end function ANYADIR
```

Código 4.3.2*: Paralelización OMP en la fase QRupdate: Añadir Filas*

4.4. COMPARATIVAS

En esta sección confrontaremos las diferentes versiones implementadas y analizaremos las mejoras de rendimiento que se obtienen. En las gráficas podremos ver las diferencias de tiempo existentes entre versiones para una iteración del proceso completo *QRupdating*, pudiendo observar en algunas de ellas un desglose de tiempos para cada una de las fases que lo componen: *QRupdate:Eliminar Filas* y *QRupdate: Añadir Filas*. También se adjuntan los *SpeedUp* en diferentes tablas que nos indican el porcentaje de mejora logrado en cada versión respecto de la anterior.

Recalcar que los tiempos representados en las gráficas son para una sola iteración del proceso completo *QRupdating* con incrementos en el número de filas a actualizar.

El primer par de comparativas se ciernen sobre la idea de que cualquier optimización del algoritmo original debe asentarse sobre la base de un acceso eficiente a memoria. Es por ello que en estas dos primeras comparativas se reflejan unas versiones preliminares a la expuesta en el punto 4.1. Versión con BLAS/LAPACK, en la que los accesos para la matrices resultado de la factorización QR se realizan con un *stride* de tantos elementos como contiene cada columna de Q^T y R^T, esto es, *m*. Como se verá a continuación, ello repercute negativamente en los tiempos.

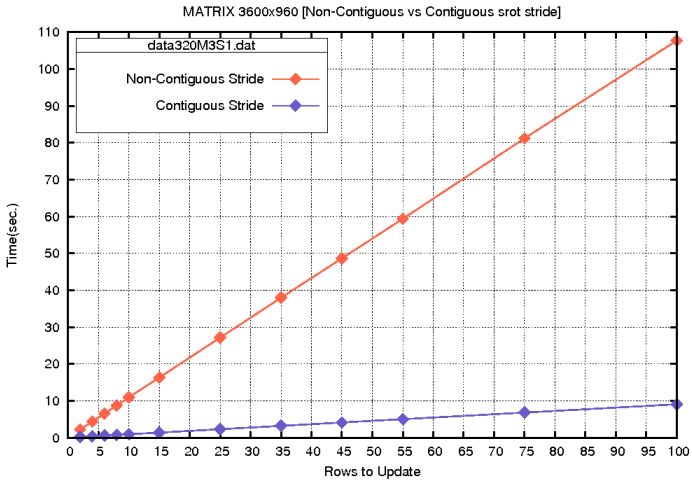
Para la toma de tiempos se ha contado con dos tamaños diferentes de la matriz origen que posee los datos captados por los micrófonos, esto es *X*, que vienen determinados por dos tamaños prefijados para L_h y L_g (para más detalle acerca del significado de cada parámetro ver el Modelo de la Señal y su repercusión en los Algoritmos *Beamforming*). Así, para una longitud prefijada de L_h=L_g=320, el tamaño de la matriz *X* es de *m*=3.600 filas y *n*=960 columnas y para una longitud prefijada de L_h=L_g=700, el tamaño de *X* es de *m*=8.400 filas y *n*=2.100 columnas.

En las comparativas se consideran ambos tamaños, siendo para el tamaño más pequeño, el más interesante, en términos de que el proceso pueda aplicarse para el mayor número de filas a actualizar en tiempo real (por debajo de un segundo en tiempo de ejecución). Nos referiremos al tamaño menor de matriz con L_g=320 y al mayor con L_g=700.

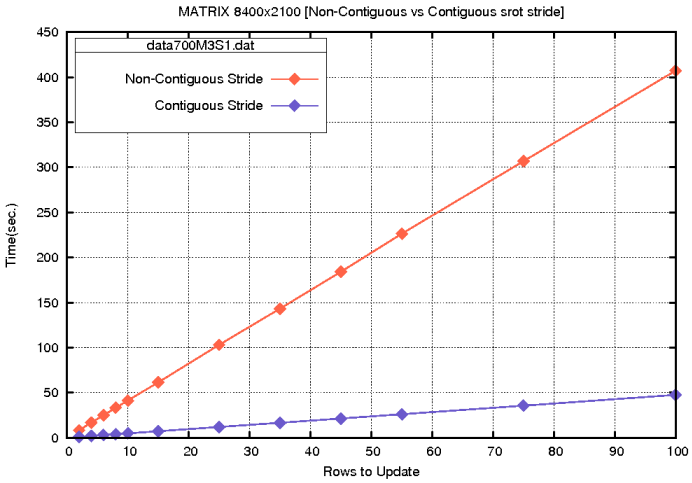
4.4.1. ACCESO CONTIGUO contra ACCESO NO CONTIGUO en Q^T

Se comparan aquí las diferencias de tiempo resultantes de las aplicación de las rotaciones de Givens con un *stride* contiguo sobre Q^T (que no sobre R^T), en azul en las gráficas, con respecto a un *stride* no contiguo en Q^T, en rojo en la gráficas.

En la Gráfica 4.4.1.1 se muestra la diferencia de tiempos para el tamaño de matriz pequeño (L_g=320) y en la Gráfica 4.4.1.2 para el tamaño de matriz grande (L_g=700).



Gráfica 4.4.1.1: *comparativa para L_g=320.*



Gráfica 4.4.1.2: *comparativa para L_g=700.*

En ambos casos, podemos ver cómo los tiempos con un acceso no continuo a memoria para el cálculo de las rotaciones puede llegar a ser ocho veces más lento con cien filas a actualizar para L_g=700 (Gráfica 4.4.1.2) y hasta diez veces para el mismo número de filas a actualizar para L_g=320 (Gráfica 4.4.1.1) .

La diferencia de prestaciones se debe al mero hecho de que la disposición de las matrices para el caso "*Non-Contiguous Stride*" de las gráficas, se encuentra por filas (*row-major order*), mientras que la rutina **`cblas_srot`** está implementada para trabajar en *column-major order*, por lo que si queremos utilizarla aún con la disposición por filas, debemos indicar un *stride* para la rutina de *m* elementos (véase la Ficha 4.4), la distancia entre datos de una misma fila considerando una disposición por columnas.

Este es el pretexto para la siguiente comparativa y en definitiva, tal como se ha argumentado al inicio de las Comparativas, la implementación de la Versión con CBLAS/LAPACK con el principio del acceso contiguo a memoria en ambas matrices en la aplicación de las rotaciones de Givens.

4.4.2. ACCESO CONTIGUO en R contra ACCESO NO CONTIGUO en R^T

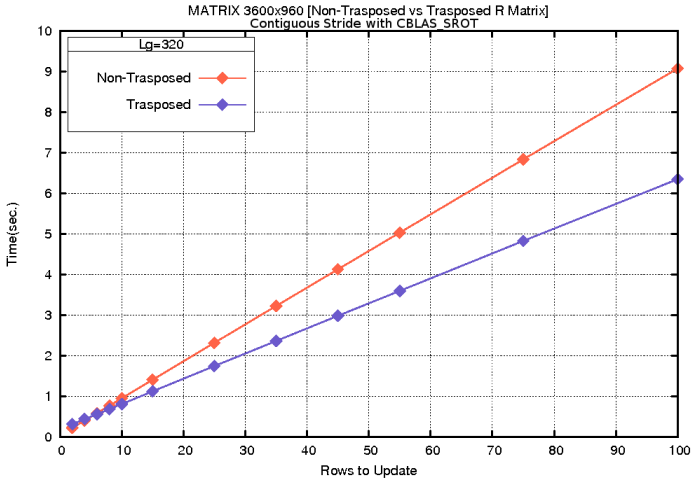
Como consecuencia de lo explicado en la comparativa anterior, nos queda pues encontrar la forma de utilizar la rutina **`cblas_srot`** sobre R^T de modo que tanto la fase *Qrupdate:Eliminar Filas* como *Qrupdate:Añadir Filas* se produzca con una distancia contigua entre los datos a tratar.

Realmente, tanto la matriz R^T como Q^T se forman a partir de la descomposición QR sobre X^T , y ésta ya se encuentra en *column-major order* como se nos explica en [13, pág. 35]. Gracias a ello, podemos aprovecharnos de un acceso contiguo sobre los datos de Q^T para el cálculo de las rotaciones de Givens, pero no siendo así sobre la matriz R^T. Esto es así, porque el proceso de las rotaciones de Givens que se aplican sobre R^T transcurre por filas, y estando almacenada por columnas, el *stride* entre elementos es igual a *m*.

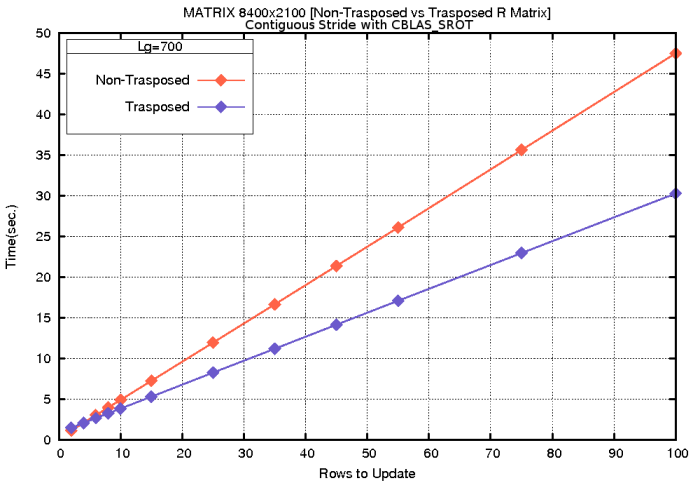
Por tanto, se opta por trasponer dicha matriz (pasando de R^T *a* R) antes de empezar con el proceso iterativo concerniente a *QRupdating,* durante el tiempo de inicialización del sistema, junto con otros cálculos como la primera y necesaria factorización QR completa.

Si bien es cierto que, actualmente el programa de simulación que se nos ha aportado para el trabajo para el cálculo de los filtros *Beamforming* sobre matrices de datos reales, en su rutina **`beamformerQRLCMV`** [13, pág. 42], necesita de la matriz R^T y esta rutina es utilizada en cada iteración, no consideraremos más allá de la Gráfica 4.4.2.3, el tiempo de destrasposición (trasponiendo la R actualizada a R^T al final de la fase *Qrupdate:Añadir Filas*), pues en futuras revisiones, dicha rutina puede ser adaptada para que trabaje con R. Por otro lado, también podríamos construir R en la inicialización del sistema, a partir de los datos de la matriz X en su forma normal, es decir, no traspuesta.

Excepcionalmente, en las siguientes gráficas (Gráfica 4.4.2.1 a Gráfica 4.4.2.3) se ha considerado el tiempo de destrasposición mencionado. Se contempla así, para evidenciar que aún añadiendo ese sobrecoste, el planteamiento de trabajar con R en vez de con R^T es el adecuado.

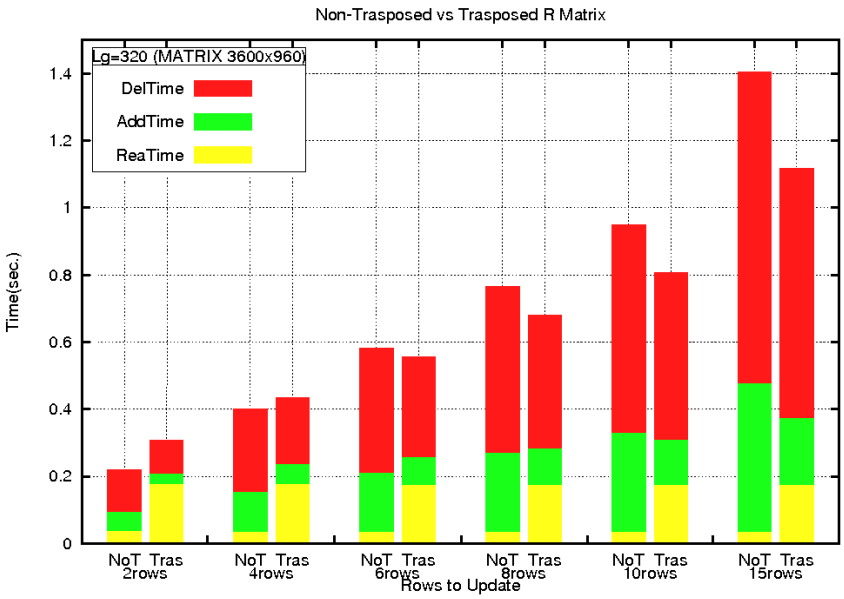


Gráfica 4.4.2.1: *comparativa para* L_g*=320.*



Gráfica 4.4.2.2: *comparativa para* L_g*=700.*

Más en detalle, en la Gráfica 4.4.2.3 y para un intervalo acotado entre 2 y 15 filas a actualizar con L_g=320, se observan los tiempos que representan cada una de las fases de *QRupdating* por separado. El tiempo "*ReaTime*" pertenece al tiempo de recolocación que ocurre en última instancia en *QRupdate: Eliminar Filas* (punto 7. y punto 8. de la secuencia original) y que podemos ver reflejado en las líneas 18 a 22 del Algoritmo 1. Siendo este último, un tiempo constante que no hemos optimizado en ninguna de las versiones al ser casi despreciable en relación al coste de las fases de eliminación y adición de filas, lo hemos plasmado por separado.

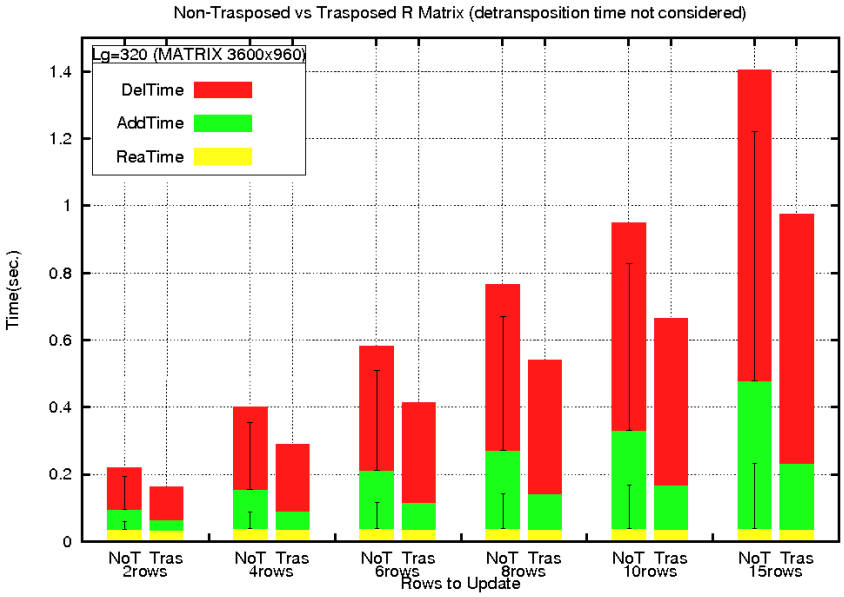


Gráfica 4.4.2.3: *comparativa para L_g=320.*

Etiquetas de barras:
NoT : Versión con la matriz R^T , *y*
Trans : Versión con la matriz *R*.

Si nos fijamos, vemos como el tiempo "*ReaTime*", en amarillo, para la versión con la matriz *R* es muy superior al de la versión con la matriz R^T. Esto es porque se ha incluido en "*ReaTime*", el tiempo de destrasposición al que nos vemos obligados debido a que la rutina *beamformer* **`beamformerQRLCMV`** tras la fase *QRupdate:Añadir Filas* trabaja con la matriz traspuesta R^T. Sin embargo, y como se ha argumentado no se tendrá en cuenta este suceso en las siguientes comparativas, al ser un sobrecoste que puede ser subsanado como también se ha explicado.

Sin considerar dicho sobrecoste, la comparativa queda como refleja la Gráfica 4.4.2.4, donde se han incluido líneas de diferencias en las barras para apreciar mejor el progreso entre versiones.



Gráfica 4.4.2.4: *comparativa para L_g=320.*

Como vemos en estas dos últimas gráficas, la mejora lograda al trasponer la matriz R^T es más notable en la fase de adición que en la fase de eliminación, y es lógico porque en la fase *QRupdate:Añadir Filas* los accesos sobre dicha matriz son más recurrentes y continuos que en *QRupdate:Eliminar Filas.*

También apreciamos lo visto en el apartado de Costes en el Capítulo 2, donde se analiza por qué la fase *QRupdate: Eliminar Filas* es más costosa que la fase *QRupdate: Añadir Filas.*

Se adjuntan dos tablas a continuación que indican el *SpeedUp* alcanzado para cada tamaño de matriz y cada fase para *f* filas a actualizar, que concuerdan con lo visto en las gráficas (a excepción del tiempo de recolocación, que aquí se considera como se ha explicado, constante y despreciable para un número de filas a actualizar). La columna "*#type*" que indica el tipo de implementación a la que pertenecen los tiempos de cada fila, acoge la etiqueta "**blas**" para la versión con la matriz R^T y "**blasRt**" para la versión mejorada con la matriz resultado de trasponer R^T.

Tabla 4.4.2.1: *SpeedUp para L_g=320.*

#RowsUp.	#AddTime.	#DelTime.	#ReaTime..	#TotTime.	#type
2	0,062317	0,12534	0,031787	0,219443	blas
2	0,033573	0,098982	0,026496	0,159051	blasRt
SpeedUp	85,62%	26,63%	19,97%	37,97%	
4	0,121466	0,248079	0,031573	0,401118	blas
4	0,060114	0,198515	0,026865	0,285494	blasRt
SpeedUp	102,06%	24,97%	17,52%	40,50%	
6	0,18075	0,371212	0,031177	0,58314	blas
6	0,086383	0,297984	0,026815	0,411182	blasRt
SpeedUp	109,24%	24,57%	16,27%	41,82%	
8	0,240248	0,494605	0,031316	0,766169	blas
8	0,11261	0,397355	0,026733	0,536699	blasRt
SpeedUp	113,35%	24,47%	17,14%	42,76%	
10	0,299583	0,618473	0,031188	0,949244	blas
10	0,138768	0,496208	0,026765	0,661741	blasRt
SpeedUp	115,89%	24,64%	16,53%	43,45%	
15	0,446372	0,927391	0,031254	1,405018	blas
15	0,202225	0,743731	0,026792	0,972748	blasRt
SpeedUp	120,73%	24,69%	16,65%	44,44%	
25	0,743327	1,542182	0,031205	2,316713	blas
25	0,333206	1,236061	0,026866	1,596133	blasRt
SpeedUp	123,08%	24,77%	16,15%	45,15%	
35	1,04162	2,150988	0,031912	3,224524	blas
35	0,464102	1,72562	0,02731	2,217031	blasRt
SpeedUp	124,44%	24,65%	16,85%	45,44%	
45	1,33951	2,758552	0,031524	4,129584	blas
45	0,594859	2,212505	0,027341	2,834705	blasRt
SpeedUp	125,18%	24,68%	15,30%	45,68%	
55	1,63655	3,364654	0,031577	5,032781	blas
55	0,72612	2,699647	0,027573	3,45334	blasRt
SpeedUp	125,38%	24,63%	14,52%	45,74%	
75	2,23422	4,574714	0,03171	6,840641	blas
75	0,98889	3,659352	0,02748	4,675722	blasRt
SpeedUp	125,93%	25,01%	15,39%	46,30%	
100	2,9852	6,066298	0,031354	9,082852	blas
100	1,32264	4,847825	0,027296	6,197756	blasRt
SpeedUp	125,70%	25,13%	14,87%	46,55%	

Tabla 4.4.2.2: *SpeedUp para L_g=700.*

#RowsUp.	#AddTime.	#DelTime.	#ReaTime.	#TotTime.	#type
2	0,36462	0,612619	0,146636	1,123875	blas
2	0,160062	0,466548	0,127884	0,754493	blasRt
SpeedUp	127,80%	31,31%	14,66%	48,96%	
4	0,69453	1,225278	0,148675	2,068489	blas
4	0,2853	0,934938	0,128979	1,349217	blasRt
SpeedUp	143,44%	31,05%	15,27%	53,31%	
6	1,02259	1,840615	0,147467	3,010675	blas
6	0,410039	1,401098	0,127541	1,938679	blasRt
SpeedUp	149,39%	31,37%	15,62%	55,30%	
8	1,35154	2,453148	0,148138	3,952822	blas
8	0,53491	1,868291	0,128178	2,531379	blasRt
SpeedUp	152,67%	31,30%	15,57%	56,15%	
10	1,68033	3,061901	0,147724	4,889956	blas
10	0,659467	2,333148	0,127603	3,120218	blasRt
SpeedUp	154,80%	31,23%	15,77%	56,72%	
15	2,49129	4,596067	0,147392	7,234745	blas
15	0,961152	3,498022	0,127469	4,586644	blasRt
SpeedUp	159,20%	31,39%	15,63%	57,74%	
25	4,13958	7,650886	0,146869	11,937335	blas
25	1,5858	5,82274	0,127306	7,53585	blasRt
SpeedUp	161,04%	31,40%	15,37%	58,41%	
35	5,80262	10,700066	0,147692	16,650373	blas
35	2,21124	8,140652	0,128025	10,479919	blasRt
SpeedUp	162,41%	31,44%	15,36%	58,88%	
45	7,48296	13,745375	0,146788	21,375124	blas
45	2,84053	10,455081	0,127792	13,4234	blasRt
SpeedUp	163,44%	31,47%	14,86%	59,24%	
55	9,18491	16,778622	0,147687	26,111221	blas
55	3,48175	12,763305	0,12758	16,372631	blasRt
SpeedUp	163,80%	31,46%	15,76%	59,48%	
75	12,6401	22,857237	0,14667	35,643955	blas
75	4,75859	17,360376	0,127653	22,246614	blasRt
SpeedUp	165,63%	31,66%	14,90%	60,22%	
100	17,0054	30,392273	0,146232	47,543858	blas
100	6,36144	23,084074	0,126454	29,571966	blasRt
SpeedUp	167,32%	31,66%	15,64%	60,77%	

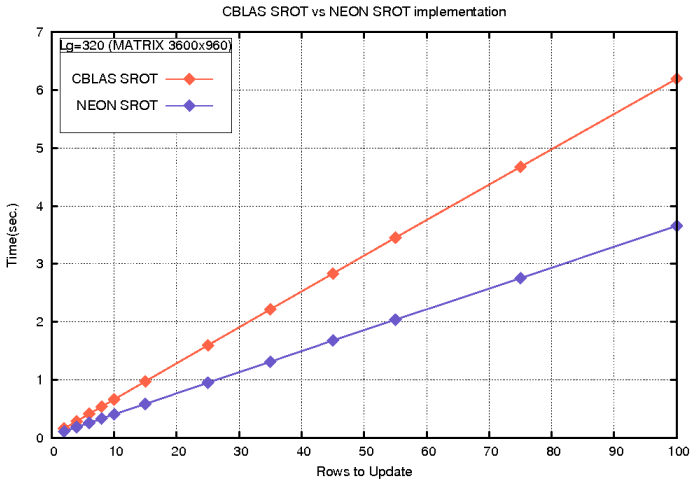
En base a los *SpeedUp* mostrados, indicar también que el coste asociado a la recolocación (*"ReaTime"*), es aproximadamente un 15% más rápida con *R* que con R^T.

Podemos estimar que la mejora total para L_g=320 es de alrededor del 45% a partir de quince filas en adelante, y de más de un 50% para L_g=700.

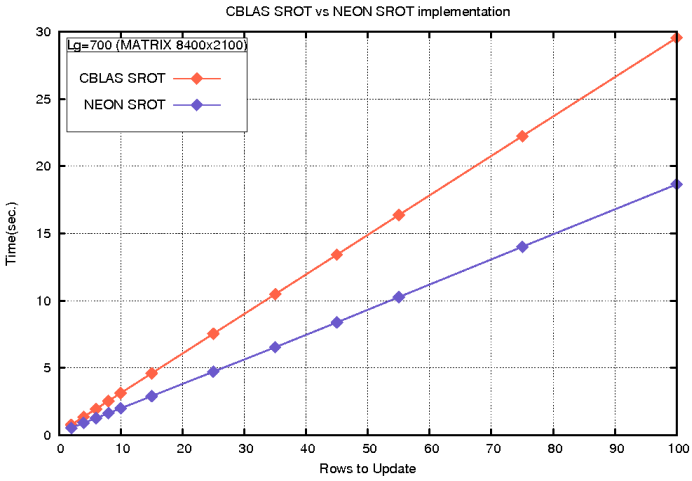
Vistas estas dos primeras comparativas, en las siguientes comparativas se trabaja siempre con *stride* o acceso contiguo sobre ambas matrices para el cálculo de las rotaciones de Givens, y se considera que el tiempo "*ReaTime*" es única y exclusivamente debido a la recolocación que ocurre entre la fase de eliminación y adición, y por tanto minúsculo y constante en comparación con las fases de eliminación y adición de filas.

4.4.3. BLAS/LAPACK vs. NEON

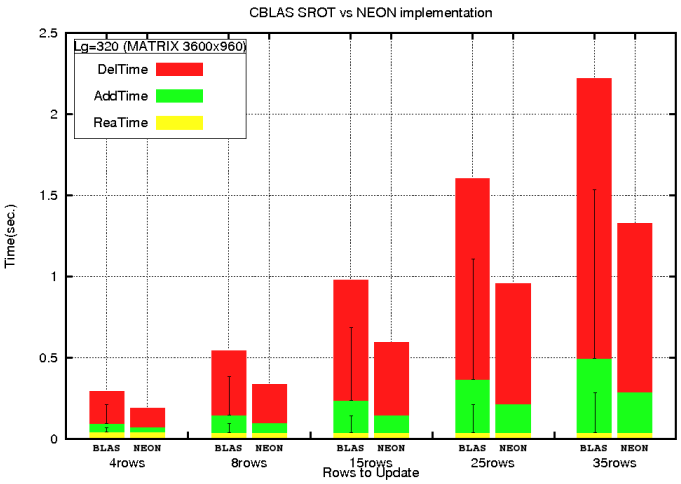
En las gráficas 4.4.3.1 y 4.4.3.2 se contempla una visión general de la ganancia que supone el uso de la herramienta de vectorización de datos NEON, mientras que en las gráficas 4.4.3.3 y 4.4.3.4 se puede apreciar con más detalle los tiempos relativos a cada fase por separado y apilados para un intervalo acotado de filas a actualizar.



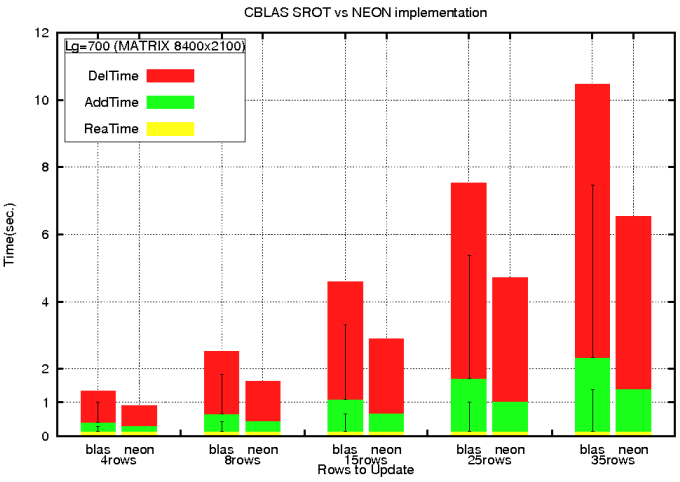
Gráfica 4.4.3.1: *comparativa para L_g=320.*



Gráfica 4.4.3.2: *comparativa para L_g=700.*



Gráfica 4.4.3.3: *comparativa para L_g=320.*



Gráfica 4.4.3.4: *comparativa para L_g=700.*

Podemos ver en estas dos últimas gráficas como la mejora es proporcional al coste de cada fase. Esto se puede contemplar de forma más precisa en las tablas con los *SpeedUp* relativos a esta comparativa, de nuevo para los dos tamaños de la matriz origen.

Tabla 4.4.3.1: *SpeedUp para L_g=320.*

#RowsUp.	#AddTime..	#DelTime.	#ReaTime.	#TotTime.	#type
2	0,098982	0,033573	0,026496	0,159051	blas
2	0,059077	0,022633	0,026606	0,108315	neon
SpeedUp	67,55%	48,34%	-0,42%	46,84%	
4	0,198515	0,060114	0,026865	0,285494	blas
4	0,118763	0,037243	0,026809	0,182815	neon
SpeedUp	67,15%	61,41%	0,21%	56,17%	
6	0,297984	0,086383	0,026815	0,411182	blas
6	0,177897	0,051416	0,026791	0,256104	neon
SpeedUp	67,50%	68,01%	0,09%	60,55%	
8	0,397355	0,11261	0,026733	0,536699	blas
8	0,237483	0,065581	0,026753	0,329817	neon
SpeedUp	67,32%	71,71%	-0,07%	62,73%	
10	0,496208	0,138768	0,026765	0,661741	blas
10	0,296401	0,079844	0,026695	0,40294	neon
SpeedUp	67,41%	73,80%	0,26%	64,23%	
15	0,743731	0,202225	0,026792	0,972748	blas
15	0,443122	0,113045	0,026692	0,582859	neon
SpeedUp	67,84%	78,89%	0,37%	66,89%	
25	1,236061	0,333206	0,026866	1,596133	blas
	0,73847	0,184057	0,026872	0,949398	neon
SpeedUp	67,38%	81,03%	-0,02%	68,12%	
35	1,72562	0,464102	0,02731	2,217031	blas
35	1,030181	0,255227	0,027322	1,31273	neon
SpeedUp	67,51%	81,84%	-0,04%	68,89%	
45	2,212505	0,594859	0,027341	2,834705	blas
45	1,324542	0,327026	0,027131	1,678699	neon
SpeedUp	67,04%	81,90%	0,77%	68,86%	
55	2,699647	0,72612	0,027573	3,45334	blas
55	1,612318	0,397919	0,02755	2,037787	neon
SpeedUp	67,44%	82,48%	0,08%	69,47%	
75	3,659352	0,98889	0,02748	4,675722	blas
75	2,184455	0,54217	0,027378	2,754004	neon
SpeedUp	67,52%	82,39%	0,37%	69,78%	
100	4,847825	1,32264	0,027296	6,197756	blas
100	2,896714	0,735124	0,027172	3,65901	neon
SpeedUp	67,36%	79,92%	0,46%	69,38%	

Tabla 4.4.3.2: *SpeedUp para L_g=700.*

#RowsUp.	#AddTime.	#DelTime.	#ReaTime.	#TotTime.	#type
2	0,160062	0,466548	0,127884	0,754493	blas
2	0,10832	0,290717	0,127649	0,526686	neon
SpeedUp	47,77%	60,48%	0,18%	43,25%	
4	0,2853	0,934938	0,128979	1,349217	blas
4	0,178344	0,590925	0,128536	0,897805	neon
SpeedUp	59,97%	58,22%	0,34%	50,28%	
6	0,410039	1,401098	0,127541	1,938679	blas
6	0,248334	0,878484	0,127685	1,254503	neon
SpeedUp	65,12%	59,49%	-0,11%	54,54%	
8	0,53491	1,868291	0,128178	2,531379	blas
8	0,318256	1,180923	0,127836	1,627015	neon
SpeedUp	68,08%	58,21%	0,27%	55,58%	
10	0,659467	2,333148	0,127603	3,120218	blas
10	0,387362	1,467493	0,127758	1,982613	neon
SpeedUp	70,25%	58,99%	-0,12%	57,38%	
15	0,961152	3,498022	0,127469	4,586644	blas
15	0,552325	2,210934	0,12726	2,890519	neon
SpeedUp	74,02%	58,21%	0,16%	58,68%	
25	1,5858	5,82274	0,127306	7,53585	blas
25	0,903431	3,679683	0,127086	4,7102	neon
SpeedUp	75,53%	58,24%	0,17%	59,99%	
35	2,21124	8,140652	0,128025	10,47991	blas
35	1,26495	5,141146	0,127618	6,533714	neon
SpeedUp	74,81%	58,34%	0,32%	60,40%	
45	2,84053	10,45508	0,127792	13,4234	blas
45	1,64542	6,605262	0,127585	8,378271	neon
SpeedUp	72,63%	58,28%	0,16%	60,22%	
55	3,48175	12,7633	0,12758	16,37263	blas
55	2,0698	8,06374	0,127356	10,2609	neon
SpeedUp	68,22%	58,28%	0,18%	59,56%	
75	4,75859	17,36037	0,127653	22,24661	blas
75	2,91477	10,96643	0,127395	14,0086	neon
SpeedUp	63,26%	58,30%	0,20%	58,81%	
100	6,36144	23,08407	0,126454	29,57196	blas
100	3,94837	14,57981	0,126217	18,6544	neon
SpeedUp	61,12%	58,33%	0,19%	58,53%	

La mejora en términos temporales aquí también es bastante constante al igual que en el progreso mostrado en la comparativa anterior en 4.4.2. Acceso Contiguo en R contra Acceso No Contiguo en R^T, y al igual que entonces el aumento de prestaciones se logra desde muy pocas filas a actualizar, beneficiándonos en mayor medida contra más filas a actualizar sean.

Se podría pensar en primera instancia que teórica e idealmente podríamos alcanzar cotas de mejora hasta cuatro veces más rápidas que la versión sin NEON, ya que supuestamente se consigue que hayan cuatro cálculos simultáneos en vez de uno solo, por lo que supone la vectorización de datos. Pero esto no es así, por dos motivos principalmente. El primero y más relevante es que se trata de un procedimiento *memory bound*, es decir, se encuentra limitado por el ancho de banda de la memoria. El segundo motivo se debe a que no todas las operaciones involucradas en el proceso *QRupdating* se realizan con las NEON *Intrinsics*, sino solo aquellas que atienden a la aplicación de las rotaciones de Givens.

Como puede verse para L_g=320 parece alcanzarse el pico de mejora de prestaciones en torno al 70% , y en torno al 60% para L_g=700. En general, se estima una mejora para L_g=*320* de más del 65% y de más del 55% para L_g=*700.*

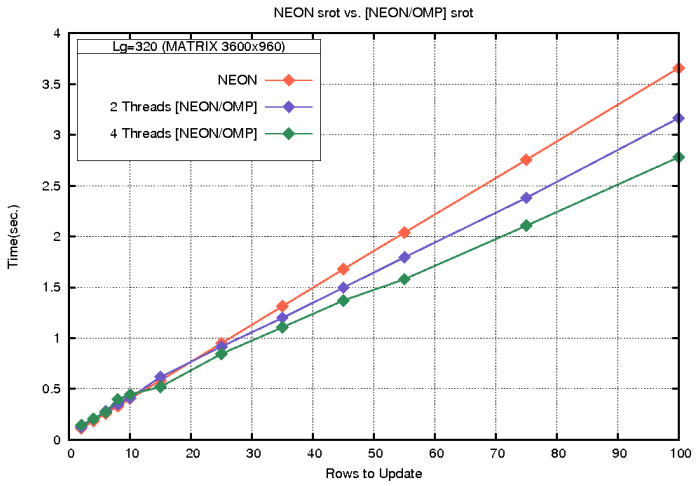
4.4.4. NEON vs. OMP

Comparamos aquí la versión del punto 4.2. Versión NEON con la del punto 4.3. Versión OMP. Para la versión con OpenMP, se han sacado tiempos tanto para dos como cuatro hilos de ejecución. Como veremos a continuación, las mejoras que otorga el uso de esta herramienta de paralelización son menores que en las implementaciones anteriores respecto de sus antecesoras.

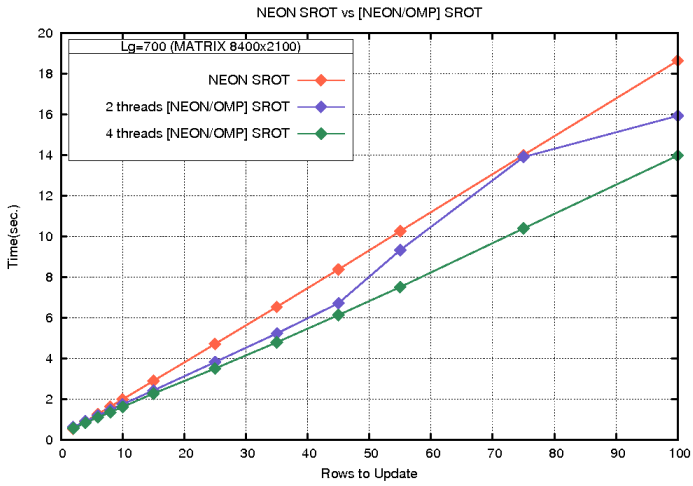
Habiendo varios *cores* trabajando sobre datos no homogéneos que surgen de un mismo espacio de memoria para todos ellos, se producen ciertas fluctuaciones en los tiempos obtenidos entre ejecuciones para un mismo número de filas a actualizar. La mejora no es tan estable como las anteriores.

Tampoco es tan buena en términos de eficiencia como la optimización resultante de la trasposición de R^T o la aplicación de la tecnología NEON. Esto también es debido en parte a que solo se paraleliza la aplicación de las rotaciones de Givens sobre la matriz de mayor dimensión, Q^T, y no sobre ambas. No obstante, arroja resultados sensiblemente más rápidos que la versión anterior.

Consideremos la nomenclatura [NEON/OMP] para designar a la versión OMP en las siguientes gráficas:

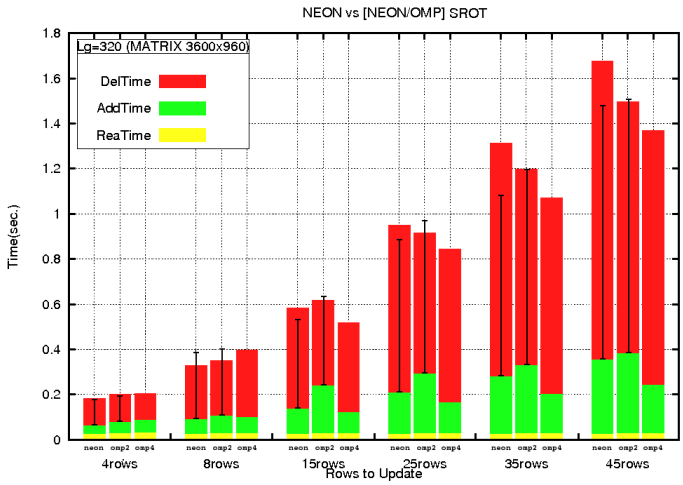


Gráfica 4.4.4.1:*comparativa para L_g=320.*

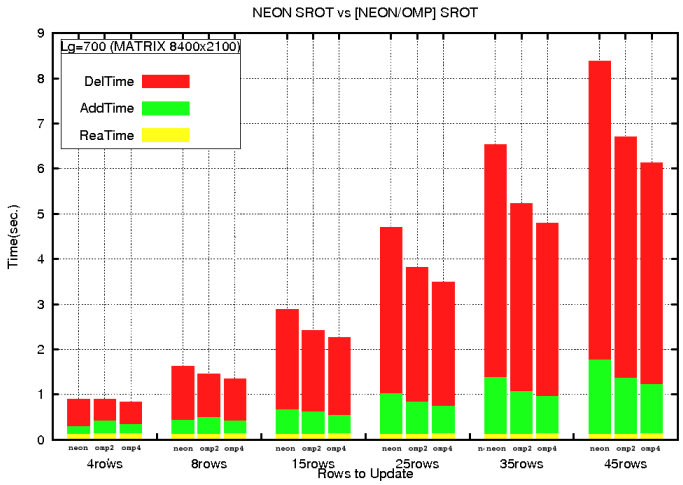


Gráfica 4.4.4.2: *comparativa para L_g=700.*

La Gráfica 4.4.4.1 y 4.4.4.2 sirven de visión general, reflejando que para L_g=700 será siempre mejor utilizar cuatro hilos que dos, pero para L_g=320, habrá casos en los que sería relativamente más rápido utilizar dos hilos que cuatro. Esto se puede apreciar mejor en las gráficas siguientes:



Gráfica 4.4.4.3: *comparativa para L_g=320.*

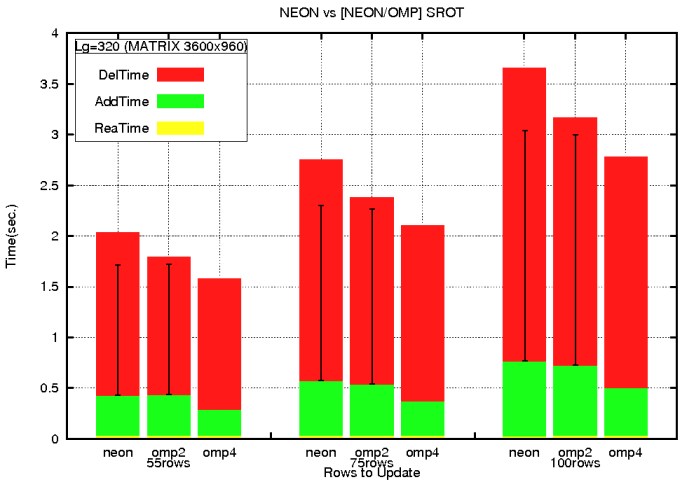


Gráfica 4.4.4.4: *comparativa para L_g=700.*

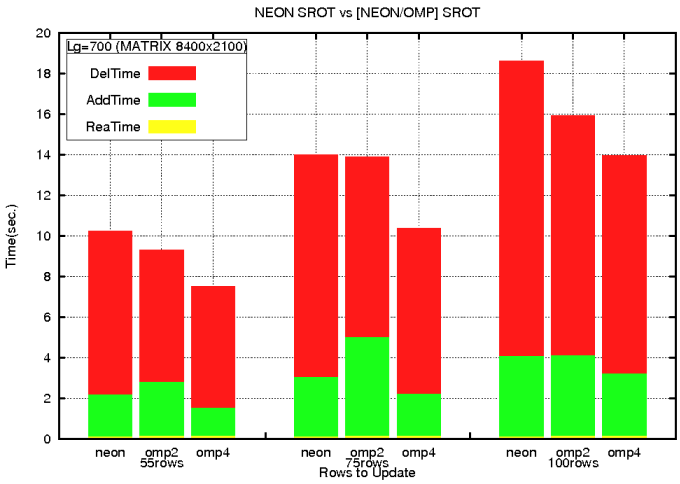
Como se puede observar en la Gráfica 4.4.4.3 para L_g=320, y pocas filas a actualizar (ocho o menos), los tiempos para las ejecuciones con dos hilos (barras etiquetadas con "**omp2**") son mejores que para cuatro hilos (barras etiquetada como "**omp4**"). Sin embargo, para estos casos y para este tamaño de matriz, los tiempos logrados por cualquiera de las dos versiones OMP son peores que sin su uso, es decir, sólo utilizando las NEON (barras etiquetadas como "**neon**"). Esto ocurre en base a lo comentado al inicio de esta comparativa, donde el sobrecoste por el manejo de varios hilos de ejecución penaliza los tiempos para estos casos, siendo peores que sin la utilización de la herramienta OpenMP. También se revela en esta misma gráfica que los tiempos logrados en la fase de eliminar para dos hilos son mejores que para cuatro hilos en la mayoría de casos comprendidos entre 4 y 45 filas a actualizar. Esto último, se puede apreciar mejor gracias a las líneas de diferencia sobre las barras "**neon**" y "**omp2**" que indican el tiempo de "**omp4**" en cada comparativa de tres barras (cada histograma), relativas a un número determinado de filas a actualizar.

Sin embargo para L_g=700 en la Gráfica 4.4.4.4, ya para muy pocas filas a actualizar el comportamiento para cuatro hilos es más rápido (en sus dos fases), que para dos hilos y que la versión sin paralelizar.

A raíz de lo comentado para la Gráfica 4.4.4.3, se expone la Gráfica 4.4.4.5 (también la 4.4.4.6), que contemplan los casos para un mayor número de filas:



Gráfica 4.4.4.5: *comparativa para L_g=320.*



Gráfica 4.4.4.6: *comparativa para L_g=700.*

Donde vemos que para L_g=320, y para 55 o más filas a actualizar, los tiempos sí que ya son mejores para **omp4** que para **omp2** en sus dos fases.

Hay dos lecturas más relativas al uso de OMP a raíz de las Gráficas 4.4.4.3 a 4.4.4.6. Por un lado, la fase *QRupdate: Añadir Filas s*e beneficia más de la paralelización del código que la fase *QRupdate: Eliminar Filas*. Por otro lado, puede ocurrir excepcionalmente que una fase tarde más en completarse con menos filas a actualizar que con más, tal como se ve en el caso concreto con 75 filas a actualizar para L_g=700 en la Gráfica 4.4.4.6.

Los datos que representan las gráficas, lo analizado, y los *SpeedUp* asociados a la comparativa de la versión NEON y la versión OMP con cuatro hilos, se contemplan en las siguientes tablas en mayor grado de precisión:

Tabla 4.4.4.1: *SpeedUp para L_g=320.*

#RowsUp.	#AddTime.	#DelTime.	#ReaTime.	#TotTime.	#type
2	0,022633	0,059077	0,026606	0,108315	neon
2	0,052065	0,062778	0,028842	0,143685	omp4
SpeedUp	-130,04%	-6,26%	-8,40%	-32,65%	
4	0,037243	0,118763	0,026809	0,182815	neon
4	0,054982	0,117456	0,032193	0,204631	omp4
SpeedUp	-47,63%	1,11%	-20,08%	-11,93%	
6	0,051416	0,177897	0,026791	0,256104	neon
6	0,06474	0,1722	0,031893	0,268832	omp4
SpeedUp	-25,91%	3,31%	-19,04%	-4,97%	
8	0,065581	0,237483	0,026753	0,329817	neon
8	0,070598	0,296107	0,028892	0,395596	omp4
SpeedUp	-7,65%	-24,69%	-8,00%	-19,94%	
10	0,079844	0,296401	0,026695	0,40294	neon
10	0,078638	0,337135	0,028894	0,444668	omp4
SpeedUp	1,53%	-13,74%	-8,24%	-10,36%	
15	0,113045	0,443122	0,026692	0,582859	neon
15	0,094948	0,395512	0,028942	0,519402	omp4
SpeedUp	19,06%	12,04%	-8,43%	12,22%	
25	0,184057	0,73847	0,026872	0,949398	neon
25	0,135853	0,680014	0,029198	0,845065	omp4
SpeedUp	35,48%	8,60%	-8,66%	12,35%	
35	0,255227	1,030181	0,027322	1,31273	neon
35	0,175395	0,802768	0,029537	1,02571	omp4
SpeedUp	45,52%	28,33%	-8,11%	27,98%	
45	0,327026	1,324542	0,027131	1,678699	neon
45	0,215707	1,124637	0,029492	1,369836	omp4
SpeedUp	51,61%	17,78%	-8,70%	22,55%	
55	0,397919	1,612318	0,02755	2,037787	neon
55	0,257324	1,292371	0,029828	1,579523	omp4
SpeedUp	54,64%	24,76%	-8,27%	29,01%	
75	0,54217	2,184455	0,027378	2,754004	neon
75	0,341136	1,735107	0,029875	2,106119	omp4
SpeedUp	58,93%	25,90%	-9,12%	30,76%	
100	0,735124	2,896714	0,027172	3,65901	neon
100	0,470551	2,280295	0,029665	2,780511	omp4
SpeedUp	56,23%	27,03%	-9,17%	31,59%	

Tabla 4.4.4.2: *SpeedUp para L_g=700.*

#RowsUp.	#AddTime.	#DelTime.	#ReaTime.	#TotTime.	#type
2	0,10832	0,290717	0,127649	0,526686	neon
2	0,165077	0,256613	0,139869	0,561558	omp4
SpeedUp	-52,40%	13,29%	-9,57%	-6,62%	
4	0,178344	0,590925	0,128536	0,897805	neon
4	0,210244	0,477019	0,140718	0,827981	omp4
SpeedUp	-17,89%	23,88%	-9,48%	8,43%	
6	0,248334	0,878484	0,127685	1,254503	neon
6	0,255727	0,696563	0,139477	1,091767	omp4
SpeedUp	-2,98%	26,12%	-9,24%	14,91%	
8	0,318256	1,180923	0,127836	1,627015	neon
8	0,293849	0,913567	0,139922	1,347338	omp4
SpeedUp	8,31%	29,27%	-9,45%	20,76%	
10	0,387362	1,467493	0,127758	1,982613	neon
10	0,339833	1,133239	0,139539	1,612610	omp4
SpeedUp	13,99%	29,50%	-9,22%	22,94%	
15	0,552325	2,210934	0,127260	2,890519	neon
15	0,418706	1,708165	0,139402	2,266272	omp4
SpeedUp	31,91%	29,43%	-9,54%	27,55%	
25	0,903431	3,679683	0,127086	4,710200	neon
25	0,616687	2,744565	0,139354	3,500606	omp4
SpeedUp	46,50%	34,07%	-9,65%	34,55%	
35	1,26495	5,141146	0,127618	6,533714	neon
35	0,836777	3,814126	0,140161	4,791062	omp4
SpeedUp	51,17%	34,79%	-9,83%	36,37%	
45	1,64542	6,605262	0,127585	8,378271	neon
45	1,09986	4,894615	0,139864	6,134343	omp4
SpeedUp	49,60%	34,95%	-9,62%	36,58%	
55	2,0698	8,063740	0,127356	10,260900	neon
55	1,40967	5,959545	0,139732	7,508950	omp4
SpeedUp	46,83%	35,31%	-9,72%	26,82%	
75	2,91477	10,966433	0,127395	14,008602	neon
75	2,10298	8,151725	0,139621	10,394331	omp4
SpeedUp	38,60%	25,67%	-9,60%	34,77%	
100	3,94837	14,579812	0,126217	18,654404	neon
100	3,07239	10,768564	0,138998	13,979948	omp4
SpeedUp	28,51%	35,39%	-10,13%	33,44%	

Como se ha ido analizando en base a las gráficas, la productividad de la versión OMP incrementa contra mayor es el número de filas a actualizar, donde la paralelización del código será más provechosa con un mayor número de rotaciones a calcular por hilo. Es decir, para que la opción con OMP funcione adecuadamente debe existir suficiente carga de trabajo para "alimentar" a cada hilo.

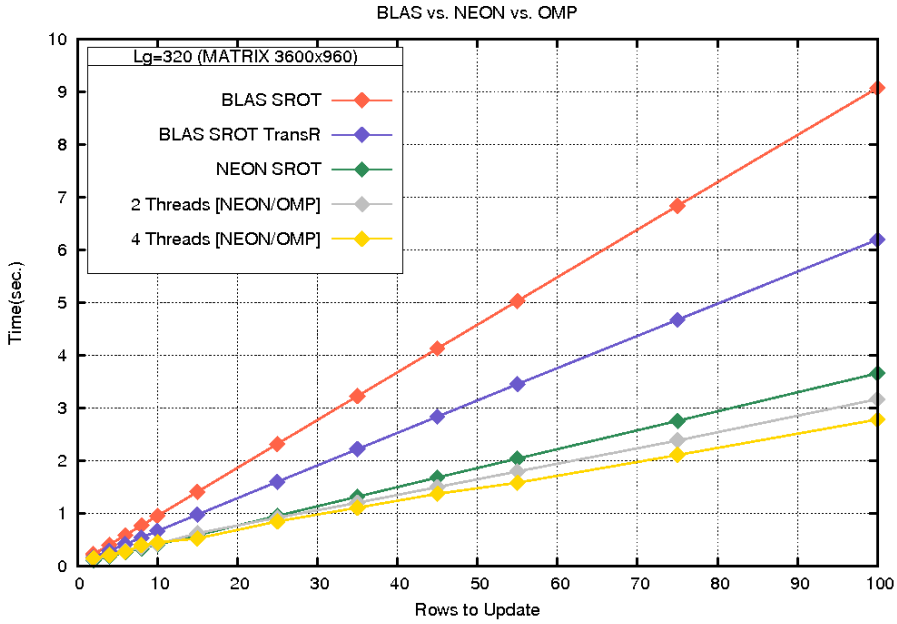
Los casos que más nos interesan son aquellos que están al filo de un segundo de ejecución, que serían aquellos que cumplen la premisa para ser considerados cálculos en tiempo real.

Así, para L_g=320 con 35 filas a actualizar nos situamos en un tiempo muy cercano al segundo de ejecución, consiguiendo una mejora de casi un 28% respecto de la versión NEON. Para L_g=700, con 6 filas a actualizar también nos situamos un poco por encima de un segundo, con una mejora de casi un 15%.

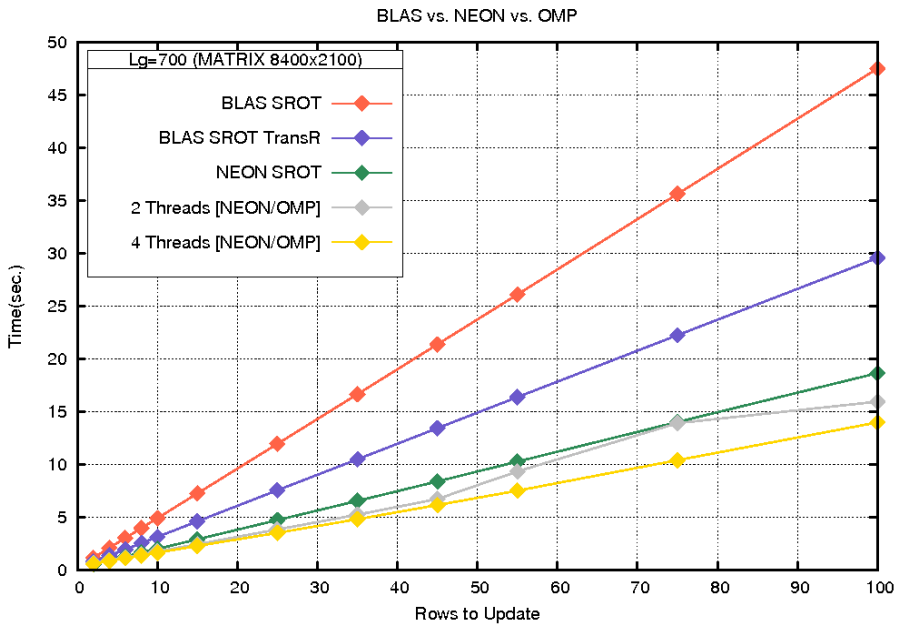
No obstante, para un máximo de 100 filas se llega a alcanzar un *SpeedUp* de algo más del 30%, en ambos tamaños.

4.4.5. BLAS/LAPACK vs. NEON vs. OMP

Las siguientes gráficas muestran las mejoras de rendimiento de las evoluciones que suponen las diferentes versiones implementadas.



Gráfica 4.4.5.1: *comparativa final para L_g=320 de las diferentes versiones.*



Gráfica 4.4.5.2: *comparativa final para L_g=700 de las diferentes versiones.*

Teniendo como referencia una primera versión "**BLAS SROT**" (color naranja en las gráficas), donde el acceso a los datos para las rotaciones de Givens sobre Q^T es contiguo y no contiguo sobre R^T vista en 4.4.2. Acceso Contiguo en R contra Acceso No Contiguo en R^T, se desprenden de las gráficas los siguientes puntos:

- Las mejoras más significativas son las relativas a "**BLAS SROT TransR**" (color violeta en las gráficas, vista en 4.1.Versión con BLAS/LAPACK, donde sí que se tiene un acceso contiguo a los datos para las rotaciones), y a "**NEON SROT**" (color verde en las gráficas, 4.2. Version con NEON *Intrinsics,* donde se hace uso del componente SIMD de ARM).

- La versión implementada con la herramienta OpenMP con uso de dos hilos "**2 Threads [NEON/OMP]**" y cuatro hilos "**4 Threads [NEON/OMP]**" (colores plata y oro, respectivamente, vistas en 4.3. Versión con OpenMP), es una optimización menos potente que "**BLAS SROT TransR**" y "**NEON SROT**" pero nos ayuda a alcanzar mejores tiempos, y con un número de filas a actualizar en *QRupdating* a partir de 15 filas en adelante para L_g=320 y de apenas 4 filas en adelante para L_g=700, posibilita en los tamaños de matrices más pequeños el cálculo en tiempo real de aproximadamente 35 filas, y en los tamaños más grandes, alrededor de 6 filas. Sin esta mejora, no es posible alcanzar estas cotas.

- A pesar de que en términos absolutos el proceso sea más costoso a mayor número de filas a actualizar, también la eficiencia de las optimizaciones incrementa proporcionalmente conforme a ello.

- El rendimiento de las mejoras implementadas es proporcionalmente mejor de acuerdo a un mayor número de datos a tratar y operaciones implicadas, es decir, para L_g=700 las ganancias son mayores que para L_g=320, si bien en términos absolutos es más costoso trabajar para el tamaño grande de matriz que para el pequeño.

- Desde "**BLAS SROT TransR**" a "**4 Threads [NEON/OMP]**" con 100 filas a actualizar se tiene un progreso de más de seis segundos para L_g=320, y de más de treinta segundos para L_g=700. Esto supone una optimización aproximada del 150% y 330%, respectivamente.

Capítulo 5 Conclusiones

En este trabajo final de grado se ha estudiado de forma global la arquitectura ARM, centrándonos en las unidades SIMD que poseen los procesadores en la versión ARMv7 (y versiones posteriores) con el objetivo de reducir los tiempos de cómputo del Algoritmo *QRupdating* que posibilita el cálculo de filtros *beamforming* para el procesamiento digital de audio. Con este mismo objetivo hemos hecho valer la herramienta de paralelización OpenMP para trabajar con una implementación concurrente que aproveche la característica de vectorización de datos que nos aporta la tecnología NEON en cada uno de los núcleos del SOC Tegra K1 de la plataforma destino Jetson TK1.

El proceso *QRupdating* se divide en dos partes o fases: *QRupdate*: *Eliminar Filas* y *QRupdate: Añadir Filas.* Esto nos ayuda a identificar y ubicar cada parte dentro del conjunto y poder referirnos a cada una con distinción según la lógica de su funcionamiento.

El uso de la tecnología NEON ha requerido un estudio de las diferentes funciones que brinda ARM en las denominadas NEON *Intrinsics,* para comprender cuáles eran las oportunas para cubrir las necesidades del cálculo de las rotaciones de Givens en base a los registros vectoriales (y sus variantes). La optimización del cálculo de esas rotaciones son el fundamento del trabajo, y la tecnología NEON, el principal instrumento para conseguirlo mediante el principio de un acceso eficiente a los datos en memoria.

Este principio de acceso eficiente a memoria es el cimiento a partir del que hemos implementado las diferentes versiones. Sin este tratamiento sobre los datos en memoria que se ven implicados en las rotaciones, haciendo que el procesador tome y procese bloques de memoria contiguos, no sería posible alcanzar las cotas de eficiencia conseguidas. Esta idea es la que conlleva el replanteamiento de la disposición de las matrices resultado de una primera factorización QR, no en base a una disposición particular por columnas o filas, sino conforme a la naturaleza del proceso de generación de matrices de Givens y la aplicación de las rotaciones en sí, en una y otra matriz.

En base a los conocimientos previos de las asignaturas de *Computación Paralela* y *Lenguajes y Entornos de Programación Paralela*, además de un buen enfoque, hemos logrado una versión concurrente para la actualización de la descomposición QR, que utiliza cada componente NEON de cada *core* disponible.

A raíz de los resultados obtenidos, podemos estar satisfechos de haber mejorado en cada evolución implementada. Sin embargo, se es consciente que aún queda margen de mejora, sobre todo en cuanto a la implementación con OpenMP, que tiene aún potencial por explotar.

En esta implementación concurrente, solo hemos paralelizado la aplicación de las rotaciones de Givens sobre la matriz de mayor dimensión, esto es, Q^T, y aunque la estrategia para la paralelización del código es acertada como vemos en los resultados, aún podría “afinarse” más. Es decir, por una parte quedaría paralelizar las rotaciones sobre la matriz R, y por otra parte, y a raíz del

análisis de la versión implementada, habría que tener en consideración un posible ajuste sobre el número de matrices de Givens a calcular antes de la aplicación concurrente en las rotaciones en la fase *QRupdate: Eliminar Filas.* Esta conclusión se basa atendiendo a los resultados de los *SpeedUp* para esta versión, donde podemos ver que el beneficio con la paralelización realizada en *QRupdate: Añadir Filas* puede llegar a ser de más de un 20% respecto de la paralelización en la fase de eliminación para un número de filas a actualizar que rondan el segundo en la ejecución completa (para L_g=320 en las ejecuciones de 25 a 35 filas, e incluso 45 filas), aunque también es cierto que la fase de adición sigue un esquema un tanto diferente a la fase de eliminación.

Por tanto, cabría hacer un estudio acerca de realizar un menor número de cálculos de matrices de Givens en la fase *QRupdate: Eliminar Filas*, pasando a la aplicación concurrente de las rotaciones de manera más asidua durante el transcurso de dicha fase, tal como ocurre en *QRupdate: Añadir Filas* por la naturaleza del proceso allí. Este mismo estudio podría comprobarse a la inversa, es decir, ver el comportamiento en *QRupdate: Añadir Filas* haciendo una mayor salvaguarda de senos y cosenos (previo cálculo de las matrices de Givens), para aplicarlos concurrentemente sobre un mayor número de rotaciones.

También sería interesante un análisis acerca de la conveniencia de la utilización de dos o cuatro hilos en la fase *QRupdate: Eliminar Filas* a tenor de los resultados obtenidos, con las expectativas de uso en tiempo real en la implementación actual. Habrían combinatorias de uso con dos hilos en la fase de eliminación y con cuatro hilos para la parte de *QRupdate: Añadir Filas* que resultarían en ejecuciones más rápidas a priori para L_g=320 en los casos de 25 a 45 filas, que además nos resultan de más interés por estar en disposición de realizarse en tiempo real o muy cerca de ello, a poco que mejoremos mínimamente las prestaciones. Por falta de tiempo, se nos escapan estos estudios, implementaciones y comprobaciones.

Nos encontramos ante un problema donde el factor limitante es la velocidad de acceso a memoria. A expensas de experimentar con un montaje del sistema real para saber de forma empírica cuál es un número aceptable de filas a actualizar por el proceso para el funcionamiento en tiempo real, podemos asegurar que siendo un código portable a otras revisiones ARM posteriores, como pudiera ser el procesador ARM Cortex-A57 del Jetson TX2 (ARMv8) , con un ancho de banda superior, capacidades mayores de memoria principal y *caches,* y una capacidad superior en el proceso de instrucciones por ciclo, se conseguirán prestaciones mejores a las mostradas aquí, sin mayores complicaciones que la compilación del código en la plataforma ARM (v7 en adelante) que consideremos oportuna.

Bibliografía

[1] Institute of Electical and Electronics Engineers. “IEEE Citation Reference” [Online/PDF]. Available: https://www.ieee.org/documents/ieeecitationref.pdf

[2] “Hexagon DSP Processor.” Internet: https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor [Oct. 5, 2016].

[3] “Chipmaker DSP Group to Benefit from Galaxy S7 Sales” Internet: https://www.androidheadlines.com/2016/03/chipmaker-dsp-group-to-benefit-from-galaxy-s7-sales.html Mar. 7, 2016 [Oct. 5, 2016].

[4] “Samsung Galaxy S7 released in Exynos and Snapdragon variants, both using CEVA's DSPs!” Internet: http://www.ceva-dsp.com/ourblog/samsung-galaxy-s7-released-in-exynos-and-snapdragon-variants-both-using-cevas-dsps/ Mar. 18, 2016 [Oct. 5, 2016].

[5] “Ericsson Mobility Report” [Online/PDF] Vol. November 2016. [Jan. 21, 2017].

[6] A. Vance. “ARM Designs One of the World's Most-Used Products. So Where's the Money?” Internet: https://www.bloomberg.com/news/articles/2014-02-04/arm-chips-are-the-most-used-consumer-product-dot-where-s-the-money Feb. 4, 2014 [Jan. 21, 2017].

[7] N.Hajdarbegovic. “ARM Servers: Mobile CPU Architecture For Datacentres?” Internet: https://www.toptal.com/back-end/arm-servers-armv8-for-datacentres, [Jan. 21, 2017].

[8] Conxa Trallero Flix. “El oído musical” [Online/PDF]

[9] “Cochlear Implant v Hearing Aid.” Internet: https://cochlearimplanthelp.com/journey/needing-a-cochlear-implant/cochlear-implant-v-hearing-aid/ [May. 14, 2017].

[10] ”Cochlear implant.” Internet: https://en.wikipedia.org/wiki/Cochlear_implant [Apr. 14, 2017].

[11] “NEON™ Programmer's Guide.” [Online/PDF] Version: 1.0.

[12] “The OpenMP API specification for parallel programming.” Internet: http://www.openmp.org/, [Sep. 25, 2016].

[13] F.J. Alventosa. “Optimización de aplicaciones de procesado de señales digitales empleando como plataforma hardware NVIDIA Jetson TK1”. Tesis de Máster en Computación Paralela, Universitat Politècnica de València, Sep. 2015.

[14] J. Lorente, G. Piñero, A.M. Vidal, J.A. Belloch & A. González. “Parallel Implementations of Beamforming Design and Filtering for Microphone Array Aplications”. *19Th European Signal Processing Conference*, Aug. 2011.

[15] J. Benesty, J. Chen, Y.Huang & J.Dmochowski. “On microphone-array beamforming from a MIMO acoustic signal processing perspective”. *IEEE Trans. On Audio, Speech and Lenguage Processing,* vol. 15, no. 3, pp. 1053-1065, Mar. 2007.

[16] Adrián Herbera González. “Estudio de la capacidad de sistemas de comunicaciones inalámbricos multiantena multiusuario.” Proyecto Final de Carrera en la Escuela Técnica Superior de Ingenieros de Telecomunicación, Universitat Politècnica de València, Oct. 2007.

[17] D. Theodoropoulos, G. Kuzmanov y G. Gaydadjiev, “Multi-core Platforms for Beamforming and Wave Field Synthesis”. *IEEE Transactions on Multimedia 99,* Dec. 2010.

[18] “Factorización QR.” Internet: https://es.wikipedia.org/wiki/Factorización_QR [Sep. 3, 2016]

[19] Sven Hammarling and Craig Lucas, “Updating the QR factorization and the least squares problem” *MIMS EPrint*, 2008.111.

[20] Steve Furber. “ARM System on Chip Architecture”. Second Edition.

[21] Ken Masterson. “What makes ARM-based chips relatively power efficient and what is the trade-off for power consumption?” Internet: https://www.quora.com/What-makes-ARM-based-chips-relatively-power-efficient Feb. 9, 2014 [Jan. 21, 2017]

[22] “ARM® Architecture Reference Manual” [Online/PDF] ARMv7-A and ARMv7-R edition.

[23] “ARM® Cortex-A15 MPCore Processor” [Online/PDF] Technical Reference Manual.

[24] “NVIDIA Jetson TK1.” Internet: http://elinux.org/Jetson_TK1 Mar. 11, 2017 [May. 14, 2017].

[25] “BLAS (Basic Linear Algebra Subprograms).” Internet: http://www.netlib.org/blas/ [Sep. 25, 2016].

[26] “LAPACK - Linear Algebra PACKage.” Internet: http://www.netlib.org/lapack/ [Sep. 25, 2016].

[27] “Automatically Tuned Linear Algebra Software (ATLAS).” Internet: http://math-atlas.sourceforge.net/, [Sep. 25, 2016].

[28] “PLASMA – Parallel Linear Algebra Software for Multicore Architectures” Internet: http://icl.cs.utk.edu/plasma/, [Sep. 25, 2016].

[29] “cuBLAS.” Internet: http://docs.nvidia.com/cuda/cublas/, [Sep. 25, 2016].

[30] “vpnc – client for cisco vpn concentrator.” Internet: https://www.unix-ag.uni-kl.de/~massar/vpnc/, [May. 14, 2017].

[31] “OpenSSH.” Internet: https://man.openbsd.org/ssh.1, [May. 14, 2017].

[32] Vim - the editor, http://www.vim.org/, [May. 14, 2017].

[33] LibreOffice, https://es.libreoffice.org/, [May. 14, 2017].

[34] gnuplot, http://www.gnuplot.info/, [May. 14, 2017].