UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Departamento de Sistemas
Informáticos y Computación

MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA

TESIS DE MÁSTER

Optimización de aplicaciones de procesado de señales digitales empleando como plataforma hardware NVIDIA Jetson TK1

VALENCIA, A 21 DE SEPTIEMBRE DE 2015

TESIS DE MÁSTER PRESENTADA POR: FRANCISCO J. ALVENTOSA RUEDA
DIRIGIDA POR: PEDRO ALONSO JORDÁ

Resumen

Hoy en día, las tabletas y los smartphones están equipados con procesadores de bajo consumo energético, como las series de arquitecturas ARMv7 y ARMv8. Estos procesadores tienen una unidad de SIMD potente, que permiten explotar el paralelismo intrínseco de los datos que tienen la mayoria de aplicaciones multimedia y de procesamiento de la señal. En el procesamiento de la señal acústica, existen múltiples aplicaciones que requieren el uso de operaciones de filtrado como son entre otros, las ecualizaciones o los sintetizadores de señal. La mayoría de estas aplicaciones se pueden ejecutar en dispositivos móviles, sin embargo, una correcta gestión de los diferentes recursos computacionales CPU's and GPU's, así como las unidades SIMD (Simple Instruction - Multiple Data) que incorporan estos, se convierte en una ardua tarea para el programador. En este documento, hablamos sobre la ejecución eficiente de filtrado multi-canal de señales de audio. Con este fin, analizamos tres estructuras comunes de filtrado de audio: FIR, IIRI y PIIR. Además, se ha desarrollado una implementación en C del algoritmo Beamformer, empleando librerías de altas prestaciones como el BLAS y LAPACK optimizado (ATLAS), el PLASMA y el CUBLAS.

Palabras clave

Procesadores de bajo consumo, ARMv7, ARM Cortex-A15, intrinsics NEON, procesamiento de audio, álgebra lineal densa, computación de alto rendimiento, Jetson TK1.

Abstract

Tablets and smart phones are equipped nowadays with low power processor architectures such as the ARMv7 and the ARMv8 series. These processors integrate powerful SIMD units to exploit the intrinsic data-parallelism of most media and signal processing applications. In audio signal processing, there exist multiple applications that require the use of filtering operations such as equalizations or signal synthesizers, among others. Most of these applications can be executed today on mobile devices via efficient exploitation of computer resources such as CPU's, GPU's and CPU feature SIMD (simple-Instruction Multiple-Data) units. In this paper, we target the implementation of multi-channel filtering of audio signals on ARM architectures. To this end, three common audio filter structures are considered: FIR, IIRI and PIIR. In addition, an implementation has been developed in C of the algorithm Beamformer using libraries of high performance such as BLAS and LAPACK optimized (ATLAS), PLASMA and CUBLAS.

Key words

Low Power Processors, ARMv7, ARM Cortex-A15, NEON Intrinsics, Audio Processing, Dense linear algebra, High-performance computing, Jetson TK1.

Agradecimientos

Las sisuientes líneas, las quiero dedicar para dar un sincero agradecimiento a las personas que han puesto todo su empeño, dedicación y apoyo para que este trabajo haya podido hacerse realidad:

- Pedro Alonso Jordá
- Antonio Manuel Vidal Maciá
- Alberto González Salvador
- Gemma Piñero Sipán
- Jose Antonio Belloch Rodriguez

Sin olvidar a mis personas más allegadas y en especial a mi pareja Tamara Reig, que en estos meses de intenso trabajo me han ayudado y mostrado su apoyo incondicional.

Gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción general

En este primer capítulo de presentación de la memoria, se introduce al lector en la temática de la tesis de máster, empezando con la sección de motivación, en la que se describe el interés del tema. En el segundo apartado, motivación personal, se realiza un breve inciso de las causas que me han llevado a realizar este trabajo. En los siguientes apartados, se presentan los objetivos específicos del trabajo, el hardware y software empleado, así como las librerías de altas prestaciones que han servido en el desarrollo del mismo.

1.1. Motivación

En la era multimedia, las tabletas y los *smartphones* incorporan procesadores de bajo consumo que juegan un papel importante en una amplia variedad de aplicaciones. El ARM Cortex-A15 [1] es una implementación de la arquitectura ARMv7 que fue concebida como un procesador embebido para smartphones, entre otros dispositivos móviles. En particular, cada núcleo de este procesador integra una unidad SIMD [2] (Simple Instruction, Multiple Data) de 128-bit. Para las arquitecturas ARM, este tipo de instrucciones SIMD son conocidas bajo el nombre de instrucciones NEON, así como en el caso de las instrucciones SIMD de las arquitecturas Intel se llaman SSE [3]. Estas instrucciones SIMD, permiten aumentar sensiblemente el rendimiento y reducir el consumo de energía para cualquier aplicación que posea paralelismo de datos, como son las aplicaciones de procesamiento de señal y multimedia. Este aumento de rendimiento y reducción de consumo se debe en gran medida a que las instrucciones NEON permiten ejecutar una misma instrucción sobre varios datos al mismo tiempo. Desde la perspectiva del programador, el paralelismo de datos puede ser aprovechado empleando las instrucciones SIMD, en el caso de los ARM llamadas NEON.

En el campo del procesamiento de señales acústicas, existen varias aplicaciones que requieren el uso de operaciones de filtrado tales como la ecualización de audio [4], la sintetización de audio [5], etc. La implementación eficiente de estas aplicaciones puede mejorar significativamente la experiencia de escucha por parte del usuario [6]. El hecho de que estas aplicaciones puedan implementarse en un dispositivo móvil constituye una tarea difícil que, sin embargo, puede acercar estas mejoras al usuario independientemente de donde se encuentre.

En este trabajo, nos centramos en dos aspectos importantes del procesamiento de la señal. La primera trata sobre las operaciones de filtrado, en la que vamos a utilizar tres estructuras comunes del filtro conocidas por sus siglas FIR, IIRI y PIIR. En la segunda trataremos de la ”limpieza” de señales sonoras que han sido alteradas por ruido, reverberaciones y mezcladas con diferentes señales al ser difundidas por el ”ambiente” en un determinado espacio temporal común. Partiendo de dichas señales, trataremos de limpiar y separar alguna de estas señales en particular con el fin de poder percibirla lo más nítidamente posible [7].

Las implementaciones de filtrado propuestas utilizan las capacidades del motor NEON integrado en el procesador Cortex-A15 del ARM, lo que permite extraer paralelismo de datos ofrecer, así, un notable incremento en la cantidad de operaciones de filtrado que se pueden realizar por unidad de tiempo. En

cambio, para optimizar la aplicación de separación de señales, se ha trabajado a un nivel superior. En este caso se han empleado librerías de altas prestaciones como ATLAS [12], PLASMA [13] y CUBLAS [14] con el fin de mejorar el tiempo de cómputo empleando todos los elementos computacionales disponibles en el dispositivo: cores CPU y la GPU. Resaltar que la librería ATLAS, hace uso en su implementación de las instrucciones NEON.

1.2. Motivavión personal

A nivel personal, como estudiante de master en computación paralela y distribuida en la rama de computación científica y futuro alumno de doctorado en informática, la elección de este TFM viene dada por la increíble oportunidad que se me dio durante mis estudios de máster de poder formar parte en un grupo de investigación como es el INCO2 del DSIC y poder iniciarme de este modo en el campo de la investigación. Dicho grupo de investigación mantiene una estrecha relación con el instituto iTeam (Instituto de Telecomunicaciones y Aplicaciones Multimedia), especializado en mejorar aplicaciones como las que he tratado en este trabajo.

1.3. Objetivos

El objetivo principal de esta investigación es el de emplear todas las técnicas y librerías de computación de altas prestaciones con el fin de reducir al máximo el tiempo de procesamiento de dos aplicaciones particulares de procesado de señal. Este objetivo principal se podría subdividir y detallar en los siguientes objetivos secundarios:

- Configurar los dispositivos hardware que se emplean, en este caso el NVIDIA Jetson TK1.
- Instalar y configurar los paquetes necesarios para emplear las librerías de altas prestaciones: CUBLAS, ATLAS y PLASMA.
- Reescribir y reorganizar código para mejorar el tiempo de computo. En particular, los códigos facilitados están escritos en Matlab y deben ser reescritos al lenguaje C/C++.
- Emplear las instrucciones SIMD de la arquitectura ARM (NEON) para reducir el tiempo de cómputo.
- Emplear librerías de computación de altas prestaciones como CUBLAS, ATLAS, PLASMA, para reducir el tiempo de cómputo.

1.4. Hardware

El hardware empleado en la realización de este proyecto, han sido las nuevas placas de NVIDIA llamadas Jetson [8]. Estas placas incorporan el chip Tegra K1 de NVIDIA (incluido en una gran variedad de dispositivos móviles como la tableta NVIDIA SHIELD) el cual incluye cuatro núcleos ARM Cortex-A15, una GPU de 192 núcleos Kepler y un quinto núcleo bastante limitado ARM para cuando la carga del sistema es muy baja reducir al máximo el consumo energético. Además, incluye dos GygaBytes de memoria principal compartida por los procesadores ARM y la gráfica, además de una gran variedad de conexiones, que se enumeran con más detalles a continuación, con el fin de hacer al Jetson de una herramienta lo más útil posible. Todas estas características, hacen al NVIDIA Jetson una herramienta muy útil para una gran variedad de aplicaciones que necesiten de un buen rendimiento computacional con un bajo consumo de energía.

A continuación se realiza una enumeración detallada de las características de la placa NVIDIA Jetson TK1 que podemos observar en la Figura 1.1.

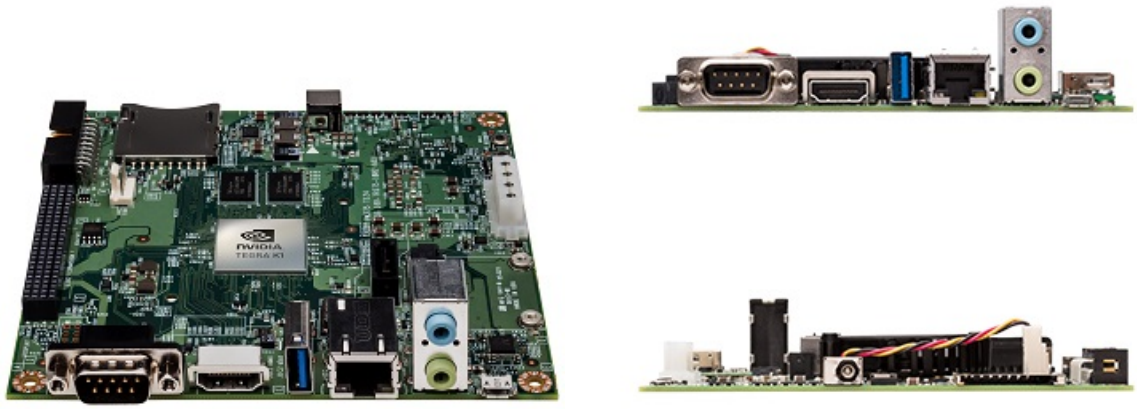

Figura 1.1: NVIDIA Jetson TK1.

- Tegra K1.
  - GPU Kepler 192 núcleos.
  - CPU 4+1 procesadores ARM Cortex-A15 (ARMv7).
- Memoria Principal 2GB.
- Disco Duro 16GB eMMC.
- 1 mini-PCIE.
- 1 SSD/MMC.
- 1 HDMI.
- 1 USB 2.0 micro AB.
- 1 USB 3.o.
- 1 RS232 serie.
- 1 SATA3.
- 1 GigaEthernet.
- 1 ALC5639 Audio in/out . . .

1.5. Software

Las aplicaciones desarrolladas se han implementado en lenguaje C y se ha empleado el compilador de GNU `gcc`, compilador común en los sistemas operativos basados en unix como el que opera en el Jetson. También se ha utilizado el compilador `nvcc`, compilador propietario de NVIDIA empleado para la compilación del código diseñado para las GPUs de NVIDIA escrito en el lenguaje CUDA [9] y también para compilar código que incorpora llamadas a rutinas que hacen uso de estas GPUs.

1.6. Librerias

En el desarrollo de este trabajo, se han empleado cuatro librerías de altas prestaciones con el fin de mejorar al máximo las prestaciones de las aplicaciones desarrolladas; estas librerías son: BLAS, LAPACK, PLASMA y CUBLAS. Además, se ha empleado el paquete ATLAS para la instalación optimizada de BLAS y LAPACK.

- **BLAS** [10] (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 BLAS3, que contiene operaciones de coste cúbico sobre matrices.

- **LAPACK** [11] (Linear Algebra PACKage): Librería desarrollada para resolver problemas de álgebra lineal numérica tales como resolución de sistemas de ecuaciones lineales, resolución de sistemas de mínimos cuadrados, resolución de problemas de valores propios y descomposiciones en valores singulares. Las rutinas de LAPACK utilizan llamadas a las rutinas de BLAS simplificando su implementación y aprovechando la eficiencia del BLAS en caso de que se disponga de una implementación optimizada.

- **ATLAS** [12] (Automatically Tuned Linear Algebra Software): No es una librería propiamente dicha, sino que permite configurar de forma empírica las librerías BLAS y LAPACK a las características del hardware sobre el que se están instalando, maximizando así el rendimiento de ambas y por lo tanto, reduciendo el tiempo computacional de nuestras aplicaciones.

- **PLASMA** [13] (Parallel Linear Algebra for Scalable Multi-core Architectures): Es una implementación mejorada de la libería LAPACK, que permite explotar las características de las arquitecturas multi-núcleos en las funciones contenidas en ella.

- **CUBLAS** [14] (CUda Basic Linear Algebra Subroutines): Librería que implementa las rutinas de BLAS pero empleando la GPU como hardware para los cálculos.

1.7. Estructura del trabajo

La memoria de la tesis de máster se estructura en los siguientes capítulos:

- **Configuración**

  En el Capítulo 2 se describen con detalle los pasos necesarios para actualizar y configurar de manera básica el Sistema Operativo que incorpora el NVIDIA Jetson, la distribución Ubuntu L4T de Linux. Además, se exponen las instalaciones y configuraciones de las librerías empleadas en el desarrollo de esta tesis, así como los problemas que surgieron al instalarlas y configurarlas.

- **Filtrado**

  En el Capítulo 3 se describen dos estructuras comunes empleadas para el filtrado de señales, como son el FIR (Finite Impulse Response) y el IIR (Infinite Impulse Response). Además de han realizado dos implementaciones diferentes de la estructura de filtrado IIR: IRRI (forma directa I) y PIIR (forma paralela). Las implementaciones han sido realizadas empleando instrucciones vectoriales NEON y tres tipos de datos, dos que trabajan con datos enteros y uno que trabaja con números reales: `INT16`, `INT32` y `FL032`. Al final del capítulo se realizan pruebas con las diferentes versiones y se analizan los resultados obtenidos.

- **Beamformer**

  En el Capítulo 4 se describe el algoritmo de *Beamformer*, empleado en señales digitales para separar una señal de otra/s de un conjunto en el que han sido mezcladas varias de ellas por un medio de transmisión con el objeto de extraer la original. Se han desarrollado diferentes versiones empleando librerías y herramientas de computación de altas prestaciones. Al final del capítulo se muestran los resultados obtenidos por las diferentes versiones y se analizan los mismos.

- **Conclusiones**

  En el Capítulo 5 se hace una recopilación de las ideas más relevantes obtenidas durante la elaboración de la tesis, tratando de enfatizar los aspectos más interesantes, los conceptos aprendidos y la aportación en la temática de procesamiento de señal digital en tiempo real. Además al final de este capítulo, se exponen las posibles mejoras o extensiones ha realizar en ambos trabajos expuestos en esta tesis, así como las futuras líneas de investigación por las que podemos encauzarnos a tenor de los resultados obtenidos.

Capítulo 2

Puesta en marcha y configuración NVIDIA Jetson TK1

En este segundo capítulo se abordan los primeros pasos que se han de llevar a cabo una vez tenemos en nuestras manos el Jetson TK1 [15], antes de pasar a implementar y ejecutar los códigos que vamos a desarrollar en los capítulos 3 y 4.

A la hora de configurar el NVIDIA Jetson TK1, nos hemos encontrado con un gran número de problemas que nos han reportado un tiempo extra para poder configurarlo de forma satisfactoria. Uno de los principales problemas que nos hemos encontrado, es que el sistema operativo del Jetson es una versión incompleta del sistema operativo Ubuntu para escritorio (L4T – Linux for Tegra), esto nos ha causado algunos problemas a la hora de configurarlo e instalar las librerías necesarias, ya que faltan algunos ficheros y/o paquetes de software que sí que se encuentran en una versión completa de Ubuntu.

Además, al ser un producto relativamente nuevo, tiene problemas de madurez en el sistema operativo que los técnicos de NVIDIA van solucionando en versiones sucesivas según los van detectando o son informados por parte de la comunidad de usuarios. Todo esto sin nombrar que, al tratarse de un producto nuevo y en fase de adquisición de usuarios y siendo la documentación bastante escueta en algunos apartados, el *feedback* a los usuarios aún es bastante escaso, lo que resulta en dificultades a la hora de encontrar soluciones a consultas acerca de cualquier problema surgido con el Jetson.

Para finalizar, comentar que no únicamente el software del Jetson carece de madurez a la hora de utilizarlo, sino que además, las librerías y/o aplicaciones no se encuentran adaptadas aún a algunas de sus peculiaridades por lo que surgen problemas añadidos al instalarlas (si es que se consiguen instalar correctamente en el Jetson), problemas que no se producen en equipos con un Ubuntu o Linux convencional. Por comentar algunos casos, destacar que no se ha podido hacer funcionar correctamente ni el compilador de ARM (`armcc`) ni la librería `MAGMA` que implementa las rutinas del `Lapack` haciendo uso de todos los núcleos computacionales disponibles en el hardware, tanto CPU's como GPU's.

En lo referente a la librería CUDA para el Jetson, comentar que la última versión disponible en el momento que se redactan estas líneas es la CUDA-6.5, mientras que para sistemas convencionales está disponible actualmente la CUDA-7.5. Además, reseñar que el paquete CUDA para el Jetson no dispone de la totalidad de las herramientas disponibles en una versión CUDA convencional.

2.1. Puesta en marcha

En esta primera sección, se van a detallar los pasos básicos para poner en marcha el NVIDIA Jetson, así como para obtener una configuración básica del mismo a partir de la cual podamos empezar a trabajar.

2.1.1. Conexiones

- Conectar un extremo del cable serie al puerto serie J1A2 UART4 de la placa NVIDIA Jetson TK1, y el otro extremo a un equipo Linux (Preferentemente el equipo Linux debe ser Ubuntu, que es el sistema Linux que incorporan los Jetson, en caso contrario pueden surgir problemas adicionales a los expuestos en este documento).
- Conectar el cable USB Micro-B (incluido en la caja) a la placa Jetson y por el otro extremo USB al equipo Linux.
- Conectar teclado y ratón al puerto USB de la placa, como la placa únicamente dispone de un único puerto USB, si se desea conectar ambos dispositivos es necesario disponer de un adaptador que saque más de un puerto USB.
- Conectar la salida HDMI de la placa Jetson a un monitor HDMI.
- Conectar un cable de red al puerto Ethernet de la placa por un extremo y por el otro a un switch/-router/etc que le proporcione acceso a internet al Jetson.
- Conectar el cable de alimentación del Jetson al puerto AC del mismo.

2.1.2. *Flashing*

La primera tarea que realizamos una vez recibidos los Jetson, fue actualizar el sistema operativo que incorporan a la versión más reciente disponible, tarea a la que llaman *flashing*. En este punto únicamente hacer un pequeño inciso, y es que en la guía se nos indica que el *flashing* se puede realizar desde cualquier equipo Linux, pero aprovechando nuestra experiencia, indicar que los sistemas operativos OSX no sirven para tal cometido, aunque bien sabemos que también están basados en Linux (distribución `Darwin`). Por lo tanto, decidimos finalmente emplear un equipo donde teníamos instalado Ubuntu como sistema operativo, que es el mismo sistema operativo en el que se basa NVIDIA para diseñar el de los Jetson. A continuación vamos a detallar paso por paso el procedimiento a seguir:

1. Descargar la última versión del software para el Jetson (L4T) de la url https://developer.nvidia.com/linux-tegra
2. Descomprimir el paquete descargado en una carpeta y ensamblar el sistema de ficheros.
```
sudo tar xpf ${RELEASE NAME}
cd Linux_for_Tegra/rootfs/
sudo tar xpf ../../Tegra_Linux_Sample-Root-Filesystem_Rxx.x.x_armhf.tbz2
cd ..
sudo ./apply_binaries.sh
```
3. Copiamos el sistema de ficheros a la memoria interna eMMC del Jetson.
   a) Poner la placa Jetson en `Recovery Mode` pulsando los botones reset y recovery de la placa y posteriormente soltando el botón de reset.
   b) Asegurarse que el equipo linux está conectado a la placa con el cable USB para el *flashing*.
   `sudo ./flash.sh -S 16GiB ${Jetson-tk1} mmcblk0p1`
   En unos minutos el sistema del Jetson abrá sido reinstalado por completo.
4. Instalamos el sistema gráfico, después de haber configurado la red, y reiniciamos el sistema.
```
sudo apt-get update
sudo apt-get install ubuntu-desktop
reboot
```

2.1.3. Configuraciones básicas recomendadas

Una vez tenemos realizado el *flashing* del Jetson por completo y el sistema ha sido reiniciado, procedemos a realizar unas configuraciones básicas en el sistema antes de proceder a instalar ningún otro software. Para ello, debemos abrir un terminal o consola del sistema Ubuntu del Jetson y seguir los pasos siguientes:

1. Es muy importante que se le diga al comando `apt` que no sobrescriba el archivo `libglx.so` si vamos a actualizar el sistema. El archivo `libglx.so` es un archivo de controladores gráficos de NVIDIA que podría ser reemplazado por una versión incorrecta al actualizar el sistema Ubuntu del Jetson, y esto causaría que no pudiera ser arrancado el entorno gráfico del Jetson. Por esta razón, es conveniente ejecutar el comando `sudo apt-mark hold xserver-xorg-core` antes de conectar a Internet y/o actualizar el sistema Ubuntu del Jetson.
2. Añadir a la lista de repositorios del sistema Ubuntu del Jetson el repositorio `universe` con los comandos `sudo apt-add-repository universe` y `sudo apt-get update`, ya que es común que se requieran paquetes contenidos en este repositorio cuando estamos desarrollando códigos.
3. Instalar el paquete `bash-completion`, permite que se auto-complete los comandos shell y te da sugerencias sobre los paquetes a instalar si se escribe un comando que no se encuentra en el sistema.
4. Instalar los paquetes `gfortran` y `gfortran-multilib` ya que las liberias `BLAS` y `LAPACK` están escritas en el lenguaje `Fortran` por lo que es necesario un compilador de fortran para instalarlas con el paquete ATLAS.

2.1.4. Instalación de CUDA

Como siguiente paso una vez hecho el *flashing* del Jetson a la última versión de software disponible, se procede a instalar las herramientas CUDA que nos permitirán hacer uso de la GPU incorporada en el chip TK1 del Jetson. A continuación detallaremos los pasos que se han se seguir para dicha instalación:

1. Descargar la última versión de CUDA disponible para sistemas L4T de la url `http://developer.download.nvidia.com/compute/cuda/release_version/rel/installers/cuda-repo-l4t-rXX.X-X-X-prod_X.X-XX_armhf.deb`.
2. Añadimos el repositorio descargado al sistema de paquetes del Jetson con el comando `sudo dpkg -i cuda-repo-l4t-rXX.X-X-X-prod_X.X-XX_armhf.deb`.
3. Actualizamos la lista de paquetes que hay en los repositorios con el comando `sudo apt-get update`.
4. Instalamos el paquete CUDA-toolkit de la versión descargada con el comando `sudo apt-get install cuda-toolkit-X-X`.
5. Añadimos los usuarios que deban tener acceso a la GPU al grupo video para que así puedan acceder con el comando `sudo usermod -a -G video usuario`.
6. Añadimos los path del CUDA instalado en el fichero bashrc de los usuarios, recargamos el fichero bashrc para que se apliquen las actualzaciones y comprobamos que podemos acceder al compilador de NVIDIA `nvcc` con los comandos:

```
echo ‘‘# Add CUDA bin & library paths:’’ >> ~/.bashrc
echo ‘‘export PATH=/usr/local/cuda/bin:$PATH’’ >> ~/.bashrc
echo ‘‘export LD_LIBRARY_PATH=/usr/local/cuda/lib:$LD_LIBRARY_PATH’’ >> ~/.bashrc
source ~/.bashrc
nvcc -V
```

2.1.5. Instalación de ATLAS

Una vez tenemos instalado el `CUDA-toolkit`, procedemos a la instalación del paquete `ATLAS` que nos hará una instalación óptima para nuestro equipo de las librerías `BLAS` y `LAPACK`. Para ello expondremos a continuación los pasos necesarios para una correcta configuración y instalación de la misma. Únicamente comentar, que si posteriormente se va ha hacer uso de otras librerías como el `PLASMA`, es necesario realizar la instalación del ATLAS con la opción `hard`, esto a diferencia que la opción `soft`, indica que la arquitectura dispone de unidades hardware para realizar las operaciones con números reales, mientras que la opción `soft` es para las arquitecturas que carecen de este tipo de unidades implementadas por hardware y lo que hacen es realizar las operaciones con números reales por software. En el caso del Jetson, el procesador dispone de estas unidades hardware, por lo que no tenemos mayores problemas para instalar la librería con una opción u otra, pero es necesario instalarla con la opción `hard` si vamos a utilizar otras librerías como en nuestro caso el `PLASMA`.

A continuación se detallan con esmero los pasos a seguir para una correcta configuración e instalación del paquete `ATLAS` en el Jetson:

1. Activar el modo performance en el Jetson
```
echo 0 > /sys/devices/system/cpu/cpuquiet/tegra_cpuquiet/enable
echo 1 > /sys/devices/system/cpu/cpu0/online
echo 1 > /sys/devices/system/cpu/cpu1/online
echo 1 > /sys/devices/system/cpu/cpu2/online
echo 1 > /sys/devices/system/cpu/cpu3/online
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

2. Crear una carpeta y posicionarnos dentro de ella
```
mkdir ATLAS
cd ATLAS
```

3. Descargar el paquete `ATLAS` de la url http://sourceforge.net/projects/math-atlas/files/.

4. Descomprimir fichero
```
tar -xjvf atlas3.XX.X.tar.bz2
rm -r atlas3.XX.X.tar.bz2
```

5. Descargar y configurar opción `hard`
```
wget http://math-atlas.sourceforge.net/fixes/armhardfp_archdef.tar
tar xvf armhardfp_archdef.tar
rm -r armhardfp_archdef.tar
vi ./CONFIG/src/atlcomp.txt
:g/=softfp/s//=hard/g
```

6. Instalar `ATLAS` con todas las funciones `Lapack`
```
wget http://www.netlib.org/lapack/lapack-3.5.0.tgz
```

7. Configurar, construir, comprobar e instalar `ATLAS`
```
mkdir srcATLAS
cd srcATLAS
../ATLAS/configure -D c -DATL_ARM_HARDFP=1 -Ss ADdir $CURRENT_PATH/ATLAS/ARMHARDFP
--with-netlib-lapack-tarfile=$CURRENT_PATH/lapack-3.5.0.tgz
sudo make build
sudo make check
sudo make ptcheck
sudo make install
```

2.1.6. Instalación de PLASMA

Para finalizar con las librerías ha instalar, procedimos a la instalación de la librería PLASMA la cual nos proporciona el poder realizar las funciones que incorpora la librería LAPACK, haciendo uso de los cuatro núcleos computacionales (CPU's) del Jetson, y no únicamente de uno. A continuación mostramos los comandos necesarios para la configuración e instalación de la librería PLASMA

1. Descargar la librería PLASMA desde la url http://icl.cs.utk.edu/projectsfiles/plasma/pubs/plasma-installer_X.X.X.tar.gz.

2. Descomprimir la librería
   ```
   tar -xzvf plasma-installer_X.X.X.tar.gz
   cd plasma-installer_X.X.X
   ```

3. Configuración e instalación
   ```
   sudo ./setup.py --prefix=/usr/lib/ --build=/usr/local --cc=gcc --fc=gfortran
   --cflags=‘‘-O2’’ --ldflags_c=-L/usr/local/atlas/lib/ --ldflags_fc=-L/usr/local/
   atlas/lib/ --blaslib=‘‘-lf77blas -lcblas -latlas’’ --lapacklib=‘‘-llapack’’
   --lapclib=‘‘-llapacke’’ --nbcores=4
   Pulsamos ‘‘d’’
   Pulsamos ‘‘d’’
   ```

Capítulo 3

Filtrado de sonido: FIR, IIRI y PIIR

ESTE capítulo aborda el primero de los dos problemas tratados con el dispositivo Jetson de NVIDIA y que consiste en el filtrado de señales digitales acústicas. Los filtros son un componente básico en todo procesamiento de señal digital y en los sistemas de telecomunicaciones. En este trabajo, nos centramos en filtros de tipo LTI (Linear Time-Invariant), es decir, filtros cuya señal producida es una combinación lineal de las entradas y sus coeficientes no varían con el tiempo. Concretamente se ha trabajado en tres modelos de filtros digitales, los cuáles se describen en las siguientes dos secciones. En la Sección 3.3 se explica la configuración e implementación de los algoritmos en nuestro sistema. El análisis de los resultados experimentales se muestra en la Sección 3.4.

En el dominio del tiempo la relación entrada-salida de un filtro lineal de tiempo discreto es dada por la ecuación de diferencia lineal EQ 3.1

$$y(m) = \sum_{k=1}^{N} a_k y(m-k) + \sum_{k=0}^{M} b_k x(m-k), \tag{3.1}$$

donde a_k y b_k son los coeficientes del filtro. La salida $y(m)$ es una combinación lineal de las N muestras de salida anteriores, la muestra actual $x(m)$ y las M muestras de entrada anteriores. Para un filtro invariante en el tiempo, ambos coeficientes (a_k y b_k) son constantes y calculados para obtener un propósito determinado como una ecualización específica, un efecto de audio espacial específico,... Existen diferentes factores como la estabilidad, el coste computacional, el desbordamiento aritmético, entre otros, que afectan a la hora de diseñar un filtro. Además, estos requisitos se dan siempre en función de una aplicación específica.

Dependiendo de la existencia de `feedback`, podemos distinguir entre dos tipos de estructuras de filtrado: FIR y IIR. En las siguientes dos secciones se describen ambos tipos, respectivamente.

3.1. Filtros FIR

Los filtros FIR (Finite Impulse Response), que proporcionan una respuesta al impulso finita, no poseen realimentación o `feedback` de manera que la salida es función de la entrada únicamente y no de salidas anteriores. La relación que guardan los datos de entrada y los de salida viene dada por la EQ 3.2.

$$y(m) = \sum_{k=0}^{M} b_k x(m-k), \tag{3.2}$$

donde la salida $y(m)$ de un filtro FIR es una función a la que únicamente le afectan los datos de entrada $x(m)$, como se muestra en la Figura 3.1. La respuesta de un filtro FIR al impulso consiste en un secuencia finita de $M+1$ muestras, donde M es el orden del filtro. Hay que tener en cuenta que el retardo de cada unidad viene representado por el operador z^{-1} en notación Z-transformada [16].

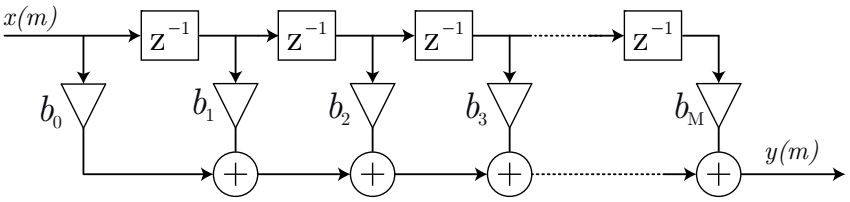

Figura 3.1: Estructura filtro FIR.

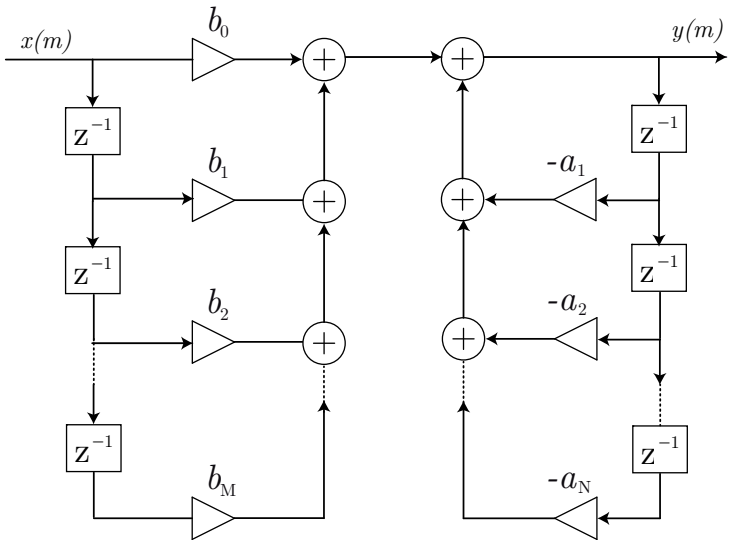

Figura 3.2: Estructura filtro IIRI.

Los filtros FIR son totalmente estables debido a que las salidas de estos son una suma finita de los múltiplos finitos de los valores de entrada [16]. Sin embargo, el coste computacional es mayor que el de un filtro con realimentación (tipo IIR) que tenga la misma finalidad.

3.2. Filtros IIR

Por su parte, los filtros IIR (Infinite Impulse Response) presentan un bucle de `feedback` que le proporcionan una estructura recursiva tal como muestra la Figura 3.2. Los filtros con una estructura IIR, se caracterizan por el hecho de que una vez la señal ha sido alterada por un impulso, la señal de salida ya no vuelve al estado de reposo nunca. Los filtros IIR necesitan de un número menor de coeficientes de filtrado para filtrar una señal que un filtro FIR. Esto implica una necesidad de almacenamiento y de cómputo menor que las necesarias para un filtro FIR equivalente. Sin embargo, un filtro FIR es siempre estable, mientras que un filtro IIR puede convertirse en inestable [16].

Existen diferentes estructuras para filtros IIR dependiendo de los requisitos relacionados con la estabilidad de los filtros y el desbordamiento en la operaciones aritméticas para calcularlos. En primer lugar, analizamos la estructura generalizada de los filtros IIR, conocida como forma directa y denotada como IIRI. Como segunda alternativa para los filtros IIR, proponemos la forma paralela PIIR.

La estructura de un filtro IIRI se puede apreciar en la EQ 3.1 y en la Figura 3.2. Los filtros de orden superior IIRI (con un número alto de coeficientes) tienen el inconveniente de que se pueden volver inestables fácilmente. Para reducir esa inestabilidad se puede descomponer el filtro IIRI en múltiples módulos de segundo orden [17] que pueden ser calculados de forma paralela, obteniendo por tanto una

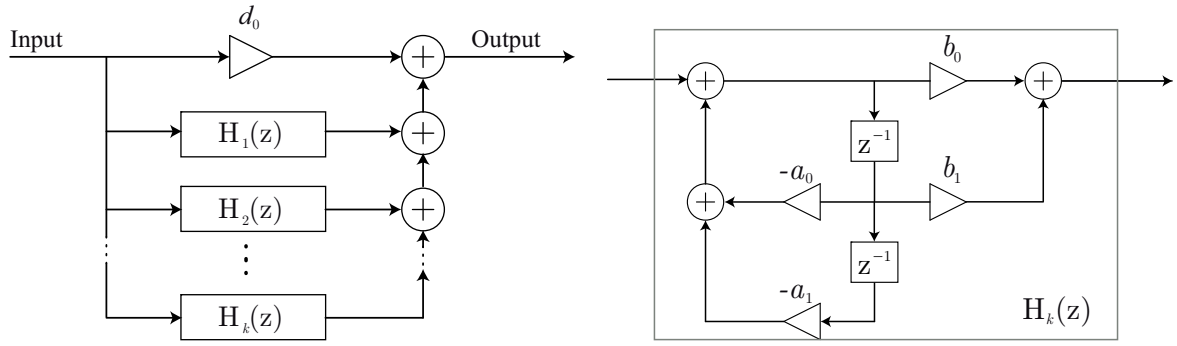

Figura 3.3: Estructura filtro PIIR.

mayor eficiencia en el cálculo. La estructura del filtro IIRI adopta pues una estructura paralela compuesta por varios módulos de segundo orden que denotamos como estructura PIIR.

La forma general de calcular un filtro PIIR consiste en un conjunto paralelo de módulos de segundo orden y un camino de coeficientes de filtro, tal como se puede ver en la EQ 3.3,

$$H(z^{-1}) = \sum_{k=1}^{K} \frac{b_{k,0} + b_{k,1} z^{-1}}{1 + a_{k,1} z^{-1} + a_{k,2} z^{-2}} + d_0 \ , \tag{3.3}$$

donde k es el número de módulos de segundo orden que tiene el filtro. La estructura del filtro y la estructura de los módulos de segundo orden se muestran en la Figura 3.3.

3.3. Configuración del sistema e implementación

La arquitectura ARMv7 y el juego de instrucciones ISA (Instruction Set Architecture) implementados en el procesador ARM Cortex-A15 además de otros, da soporte para el juego de instrucciones SIMD (Simple Instruction Multiple Data) conocidas como NEON. Las instrucciones SIMD permiten aplicar una misma instrucción a un conjunto de datos de forma paralela aprovechando de este modo el paralelismo de datos. Para poder soportar este juego de instrucciones, el procesador Cortex-A15 incorpora varios registros SIMD de 128 bits, los cuales pueden almacenar hasta dos datos reales de 64-bits doble precisión, dos datos enteros de 64-bits, cuatro datos de 32-bits reales de simple precisión, cuatro datos enteros de 32-bits, ocho datos enteros de 16-bits o dieciséis datos enteros de 8-bits.

Las instrucciones intrínsecas NEON de la arquitectura ARMv7 soportan todas las funcionalidades del juego de instrucciones NEON, incluyendo la definición de tipos de datos vectoriales [18]. Este tipo de datos es el que emplearemos para que sean cargados en los registros vectoriales del procesador y son nombrados de la siguiente manera `"type"x"number_of_lines"_t`, por ejemplo: `int16x4_t`, `int32x4_t` or `flo32x4_t`.

3.3.1. Configuración del sistema

Realizamos diferentes estructuras de filtro para ejecutar en el procesador quad-core Cortex-A15 a 2,32 GHz que incorpora el Jetson TK1. En una aplicación en tiempo real, la tarjeta de audio proporciona nuevos datos cada t_{buff} sec. En este test se ha empleado un tamaño de buffer de 1024 muestras con una frecuencia de muestreo de $f_s = 44{,}1$ KHz. Esto significa que cada 23.22 ms ($L = 1024$) el buffer está lleno de muestras preparadas para ser procesadas por lo que, para que la aplicación funcione en tiempo real, el tiempo de filtrado de cada buffer de muestras ha de ser inferior a este tiempo.

Algoritmo 3.1 Implementación del filtro FIR usando el tipo de datos 16-bit integer

requiere: $^*\mathbf{X}$, $^*\mathbf{B}$
asegurar: $^*\mathbf{Y}$

```
1: int32x4_t acum, int32x2_t res; int16_t *px; //Auxiliary variables
2: for i=0 to xysize:  //Iterates through the input and outut data
3:   acum = vmovq_n_s32(0); px=x+i //Initialize auxiliary variables
4:   for j=0 to bsize: //Iterates vector B
5:     //Multiply and accumulate current inputs with vector B coefficents
6:     acum = vmlal_s16(acum, b[j], vld1_s16(py)); px += 4;
7:   res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
8:   res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
9:   for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
```

Una característica importante para desarrollar aplicaciones de audio es la manera de comunicar el ordenador con la tarjeta de audio que captura y reproduce las señales de audio. Dependiendo del sistema operativo, podemos encontrar diferentes paquetes software. Por un lado, tenemos el software *Advanced Linux Sound Architecture* (ALSA) [19], que forma parte del kernel de los sistemas operativos Linux y proporciona una interfaz de programación de aplicaciones (API) para controladores de tarjetas de sonido. Por otro lado, los usuarios de Windows utilizan *Audio Stream Input/output* (ASIO), que es un driver de tarjeta de sonido de ordenadores para audio digital que se comunica con la aplicación del usuario a partir de capas de software intermedias del Sistema Operativo Windows [20]. Ambos ALSA y ANSIO tienen características muy similares, sin embargo, la principal diferencia reside en el tamaño de muestras que manejan. El software ALSA está diseñado para capturar/reproducir muestras de tamaño 16-bit *integers*. En cambio el driver ASIO está diseñado para capturar/reproducir muestras de tamaño 32-bit *integers*. Cabe destacar que los ficheros *wav* o los CD's comerciales almacenan muestras de tamaño de 16-bit *integers*.

Por lo tanto, analizamos en este trabajo los resultados de tres estructuras de filtros: FIR, IIRI y PIIR, cuando estas tratan con muestras de 16-bit y 32-bit *integers*, y con muestras de 32-bit *floats*.

3.3.2. Implementación

En nuestra implementación utilizamos dos vectores principalmente para almacenar las muestras de entrada y las muestras de salida. El vector **X** es el encargado de facilitar las muestras de entrada, mientras que es en el vector **Y** donde se almacenan las muestras de salida una vez el filtro correspondiente ha sido aplicado. Además, usamos el vector **B**, el cual contiene los coeficientes del filtro que deben ser aplicados a las muestras de entrada. Cuando empleamos una estructura de filtro tipo IIRI, además del vector **B**, existe también un vector **A**, donde se almacenan los coeficientes del filtro a aplicar a las muestras de salida anteriores. En la implementación de la estructura de filtro PIIR hemos utilizados los vectores **A1**, **A2**, **B1** y **B2**, además de otros dos vectores **V1** y **V2** para almacenar cálculos intermedios.

Para obtener el máximo beneficio de las instrucciones SIMD, los datos de los vectores **X**, **Y**, **A**, **B**, **A1**, **A2**, **B1**, **B2**, **V1** and **V2** han sido almacenados en posiciones de memoria consecutivas para que las operaciones SIMD puedan acceder a 2 y 4 muestras concurrentemente. Reseñar que las implementaciones basadas en 16-bit *integers* emplean vectores del tipo `int16x4_t` para almacenar los coeficientes, mientras que para las implementaciones basadas en 32-bit *integers* y 32-bit *floats* emplean los tipos `int32x4_t` y `flo32x4_t`, respectivamente. Además, los vectores **A** y **B**, que almacenan los coeficientes del filtro, han sido reordenados con el fin de optimizar el rendimiento. Este reordenamiento permite el acceso en orden creciente (en cuanto a posiciones en memoria) de dichos datos por parte de las instrucciones de carga SIMD.

En lo referente a las versiones que trabajan con datos del tipo *integer*, estas utilizan un tipo de dato diferente para almacenar los cálculos intermedios, empleando los tipos de datos `int32x4_t` y `int64x2_t` para las versiones de 16-bit y 32-bit *integers*, respectivamente; esto es así para evitar desbordamiento en

Algoritmo 3.2 Implementación del filtro IIRI usando el tipo de datos 16-bit integer

requiere: $^*\mathbf{X}$, $^*\mathbf{B}$, $^*\mathbf{A}$
asegurar: $^*\mathbf{Y}$

```
 1: int32x4_t acum, int32x2_t res; int16_t *px, *py; //Auxiliary variables
 2: for i=0 to xysize:  //Iterates through the input and outut data
 3:   acum = vmovq_n_s32(0); py=(y-asize*4)+i+1 //Initialize auxiliary variables
 4:   for j=0 to asize: //Iterates vector A
 5:     //Multiply and accumulate previous outputs with vector A coefficents
 6:     acum = vmlal_s16(acum, a[j], vld1_s16(py)); py += 4;
 7:   res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
 8:   res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
 9:   for j=0 to 1: y[i] -= (int16_t)vget_lane_s32(res, j); //Store result
10:   acum = vmovq_n_s32(0); px=x+i //Initialize auxiliary variables
11:   for j=0 to bsize: //Iterates vector B
12:     //Multiply and accumulate current inputs with vector B coefficents
13:     acum = vmlal_s16(acum, b[j], vld1_s16(py)); px += 4;
14:   res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
15:   res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
16:   for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
```

Algoritmo 3.3 Implementación del filtro PIIR usando el tipo de datos 16-bit integer

requiere: $^*\mathbf{X}$, $^*\mathbf{B1}$, $^*\mathbf{A1}$, $^*\mathbf{B2}$, $^*\mathbf{A2}$
asegurar: $^*\mathbf{Y}$

```
 1: int32x4_t acum, int16x4 v0; int32x2_t res; //Auxiliary variables
 2: for i=0 to xysize:  //Iterates through the input and outut data
 3:   Y[i] = 0; acum = vmovq_n_s32(0); //Initialize auxiliary variables
 4:   for j=0 to absize:  //Iterates through the vectors of coefficients
 5:     v0 = vmov_n_s16(x[i]); //Load the same value (x[i]) in all positions
 6:     v0 = vmls_s16(v0, a1[j], v1[j]); //multiply a1 & v1 & substract to v0
 7:     v0 = vmls_s16(v0, a2[j], v2[j]); //multiply a2 & v2 & substract to v0
 8:     acum = vmlal_s16(acum, b1[j], v0); //multiply b1 & v0 & accum. to acum
 9:     acum = vmlal_s16(acum, b2[j], v1[j]); //multiply b2 & v1 & accum. to acum
10:     vst1_s16(&v2[j],v1[j]);  //Copy the value of v1[j] to v2[j]
11:     vst1_s16(&v1[j],v0);  //Copy the value of v0 to v1[j]
12:   end for
13:   res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
14:   res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
15:   for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
16:   y[i] += FIR*x[i]  //FIR is a constant value
```

los cálculos. Además, antes de realizar los cálculos con las muestras, los valores de los coeficientes se les aplica un desplazamiento *(shift)* de bits a la izquierda, para poder así tener una buena precisión en los cálculos [21]. Una vez realizados los cálculos, a las muestras de salida se le aplica el proceso inverso, es decir, se realiza un desplazamiento de bits a la derecha sobre ellas del mismo valor que antes.

En los algoritmos 3.1, 3.2 y 3.3 se pueden observar las implementaciones para las estructuras de filtros FIR, IIRI y PIIR, respectivamente. Además, en la Tabla 3.1, podemos observar las instrucciones NEON empleadas en la implementación de todas las estructuras de filtros y para todos los tipos de datos analizados en esta tesis, así como una breve descripción de su funcionalidad.

Tabla 3.1: Instrucciones NEON empleadas con una breve descripción.

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

3.4. Análisis de resultados

Como hemos comentado con anterioridad, en este análisis de resultados estudiamos tres estructuras diferentes de filtros: FIR, IIRI y PIIR, filtros que hemos expuesto al principio del capítulo, empleando como tamaños de coeficientes del filtro 52 y 256 valores. Para poder realizar una comparativa equitativa, elegimos los valores de M y N idénticos (M=N). Todos los códigos han sido compilados con el compilador GNU *GCC* empleando las opciones de compilación *-O3*, *-mfpu=neon* y *-march=armv7*. Para auto-vectorizar un código incluyendo rutinas NEON intrinsics, se emplea la opción *-ftree-vectorize* [22].

El primer resultado, plasmado en la Figura 3.4, muestra el tiempo de ejecución requerido por los tres tipos de datos utilizados: 16-bit y 32-bit *integers* y 32-bit *floats*. Nosotros comparamos la versión optimizada empleando instrucciones intrínsecas NEON con la versión que emplea operaciones comunes (no vectoriales). El resultado mostrado recalca el beneficio en coste computacional que obtenemos si utilizamos el tipo de datos 16-bit *iteger* comparado con el tipo de datos 32-bit *float*.

Para comparar el *speedUp* obtenido empleando los diferentes tipos de datos, se ha elegido la comparativa entre los tipos de datos *INT16* y *FLO32*, ya que se consideran los más interesantes de utilizar dependiendo de las necesidades de la aplicación. En la Figura 3.5, la parte superior del gráfico representa el *speedUp* alcanzado por el tipo de datos INT16, mientras que la parte inferior representa el *speedUp* obtenido para el tipo de datos *FLO32*. En el caso del filtro FIR, el *speedUp* obtenido es muy alto, obteniendose unos valores de 5 y 6,5 para un número de coeficientes de 52 y 256, respectivamente, cuando el tipo de datos utilizado es el INT16. En cambio, para el tipo de datos FLO32 y el mismo filtro FIR, los va-

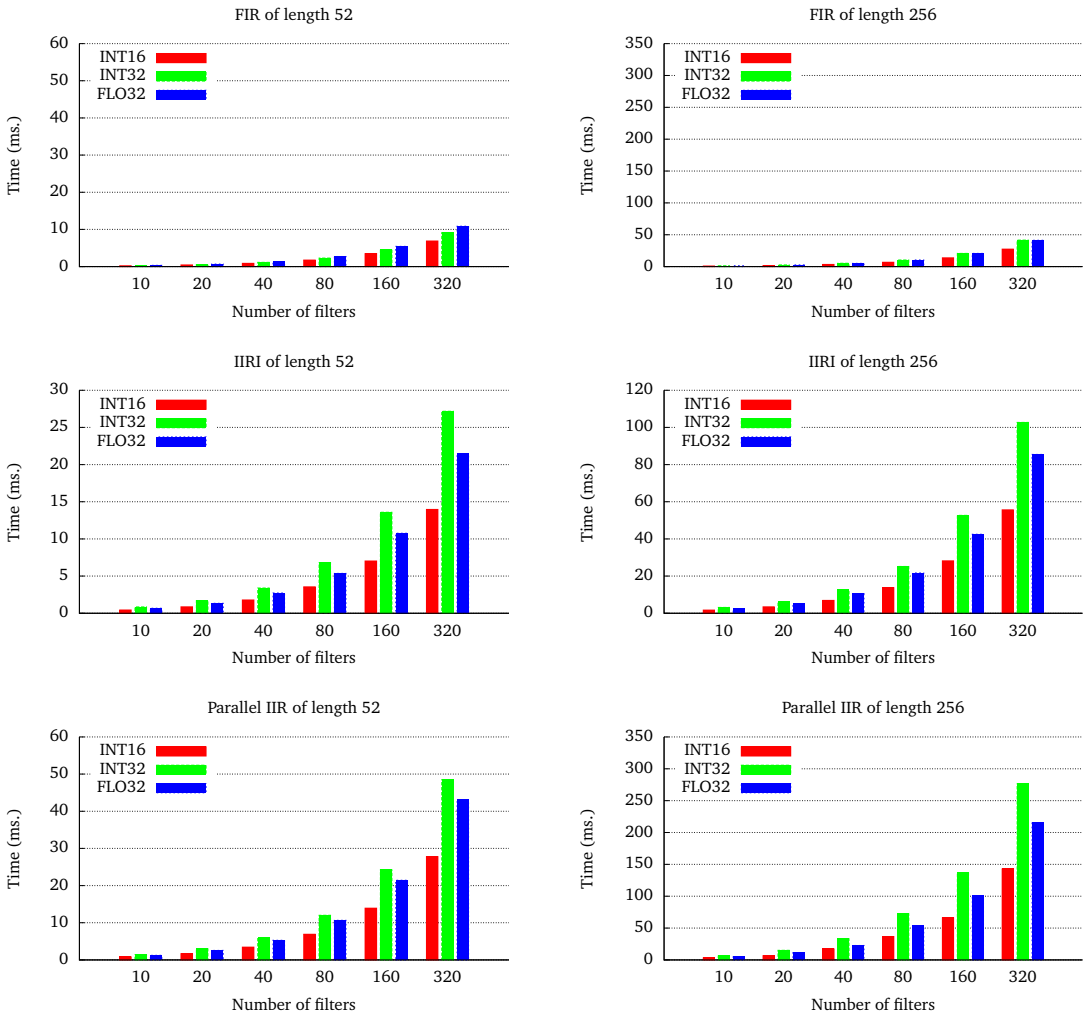

Figura 3.4: Rendimiento con respecto al tipo de datos.

Figura 3.5: Speedup alcanzado usando intrínsecas NEON.

lores de *speedUp* obtenidos son de 3 y 3,5 aproximadamente para los mismos tamaños de los vectores de coeficientes. Estos resultados nos dicen que obtenemos un gran beneficio en cuanto a tiempo de computo comparado con la versión auto-vectorizada por parte del compilador *gcc*; obteniendose un mejor beneficio cuando utilizamos el tipo de datos INT16 en vez del FLO32, aunque produciéndose un pérdida de precisión en los resultados del primero comparado con los del segundo (esta pérdida es dependiente del rango de los datos de entrada de la aplicación).

También se han implementado versiones paralelas de los códigos (empleando OpenMP) para estos tres tipos de datos y estructuras de filtro. La escalabilidad obtenida en muy razonable en todos los casos. Como se muestra en la Figura 3.6, los *speedUp* obtenidos para el caso de emplear dos núcleos computacionales (CPU), es de dos para ambas longitudes de los vectores de coeficientes. En cambio para

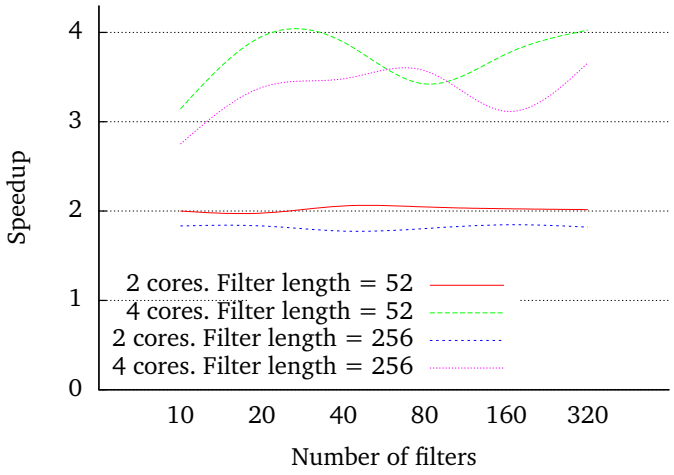

Figura 3.6: Speedup del filtro IIRI empleando 2 y 4 cores.

cuatro núcleos, el comportamiento es un poco más irregular obteniendose unos valores de *speedUp* entre tres y cuatro aunque entrando estos dentro de la normalidad. Con cuatro núcleos ya no son tan ideales como con dos núcleos, teniendo en cuenta que en ambos casos no se han contemplado dependencias entre los datos de los diferentes núcleos computacionales. Esta diferencia en el rendimiento debería estar relacionada con algún cuello de botella en el hardware del Jetson, presumiblemente en la memoria caché de nivel dos (L2).

Se define t_{buff} como el intervalo de tiempo que se tarda en generar dos *buffers* consecutivos de muestras de entrada a procesar. También se define t_{proc} como el tiempo que tarda nuestra aplicación en procesar las muestras del buffer. Por lo tanto, para que nuestra aplicación funcione en tiempo real, las muestras contenidas en el *buffer* actual han de ser procesadas antes de que el siguiente *buffer* esté listo para ser procesado, esto es $t_{\text{proc}} < t_{\text{buff}}$.

El objetivo de la Figura 3.7 es el de mostrar, para cada una de las estructuras de filtro y para los tipos de datos INT16 y FLO32 empleando en ambos casos 52 y 256 como tamaño de los coeficientes del filtro, cuál es el número máximo de filtros que nuestra aplicación sería capaz de procesar en tiempo real, es decir, siendo $t_{\text{proc}} < t_{\text{buff}}$.

En la Figura 3.7 se puede apreciar la evolución de t_{proc} en función del número de filtros procesados. Esta figura compara los tipos de datos INT16 y FLO32. Ahora vamos a comentar por encima que límites se pueden extraer de la gráfica para cada una de las estructuras de filtro estudiadas en este trabajo, que permitan a la aplicación que emplee estas estructuras de filtro trabajar en tiempo real. Para la estructura FIR los límites para los tipos de datos INT16 y FLO32 con 256 coeficientes son de 170 y 260 filtros, respectivamente. Para la estructura IIRI son de 80 y 125. Por último, para la estructura PIIR son de 50 y 25 filtros aplicados en tiempo real.

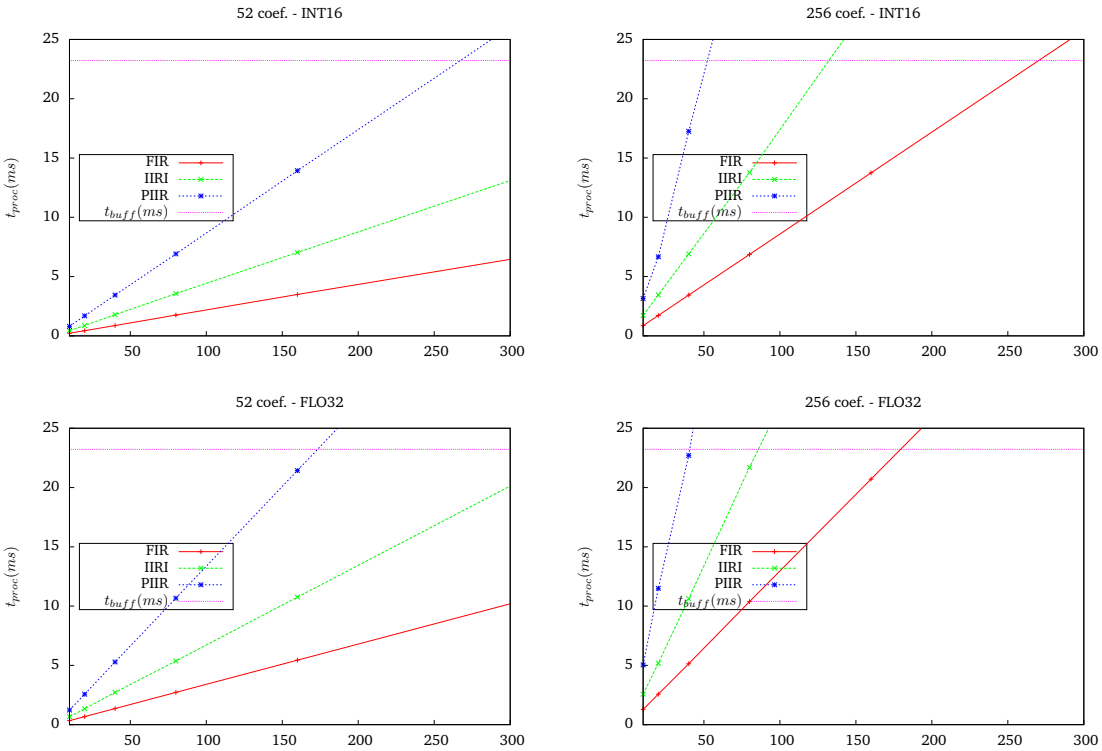

Figura 3.7: Evolución del tiempo de computo necesario para el cálculo en tiempo real.

Capítulo 4

Beamformer

ESTE capítulo aborda el desarrollo del algoritmo del *beamforming*, algoritmo capaz de, a partir de una señal recibida que ha sido alterada por ruido, reverberaciones y otras señales, limpiarla y separarla del ruido a fin de que ésta quede como originalmente era antes de haber sido transmitida y alterada por agentes externos. Para ello, la captura de la señal a procesar se realiza a partir de un array de micrófonos que capturan la señal desde diferentes puntos del espacio.

4.1. Modelo de señal

En el modelo presentado vamos a asumir un sistema compuesto por M altavoces que van a emitir M señales diferentes: $s_1(k), s_2(k), \ldots, s_M(k)$, y que van a ser capturadas conjuntamente por N micrófonos situados en otro espacio de la habitación diferente al de los altavoces y que capturarán las M señales como una señal única que, además, se encontrará alterada por ruido y reverberaciones de la habitación. El problema que trata de solucionar este algoritmo consiste en cómo separar cada una de las señales $s_1(k)$, $s_2(k)$,..., $s_M(k)$ a partir de las señales capturadas por los distintos micrófonos del sistema. En la Figura 4.1 se puede apreciar un ejemplo con dos altavoces y tres micrófonos. El enfoque adoptado en este documento hace uso de algoritmos de procesamiento de señal para el diseño del ancho de banda *beamformers* (filtros g_n en la Figura 4.1), una vez que todos los canales de respuesta de la habitación (h_{nm} en la Figura 4.1) son conocidos. Este sistema puede ser modelado como un sistema multicanal con dos entradas (altavoces) y tres salidas (micrófonos). La generalización a un sistema de Multiples entradas - Múltiples salidas (MIMO, Multiple Input - Multiple output) puede ser estudiado en [23].

De acuerdo con la Figura 4.1, la salida del micrófono nth viene dada por la EQ 4.1:

$$x_n(k) = \sum_{m=1}^{M} \sum_{j=1}^{L_h} h_{nm}(j) s_m(k-j) + v_n(k), \tag{4.1}$$

donde $n = 1, 2, \ldots, N$, siendo N el número de micrófonos del sistema, y $m = 1, 2, \ldots, M$, donde M es el número de señales de entrada (fuentes) o número de altavoces de la Figura 4.1. El parámetro L_h es la longitud del canal de respuesta acústico h_{nm} más largo de la habitación, mientras que $v_n(k)$ es el ruido de la señal. En aras de aportar una mayor claridad a la ecuación del problema, el término que aporta ruido, $v_n(k)$ en la EQ 4.1, no se considerará en el siguiente modelo de la señal EQ 4.2:

$$x_n(k) = \sum_{m=1}^{M} h_{nm}^T s_m(k), \tag{4.2}$$

siendo $s_m(k)$ el vector columna definido como:

$$s_m(k) = [s_m(k), s_m(k-1) \ldots s_m(k-L_h+1)]^T,$$

y h_{nm} es el $R^{L_h \times 1}$ vector del canal acústico del altavoz m al micrófono n.

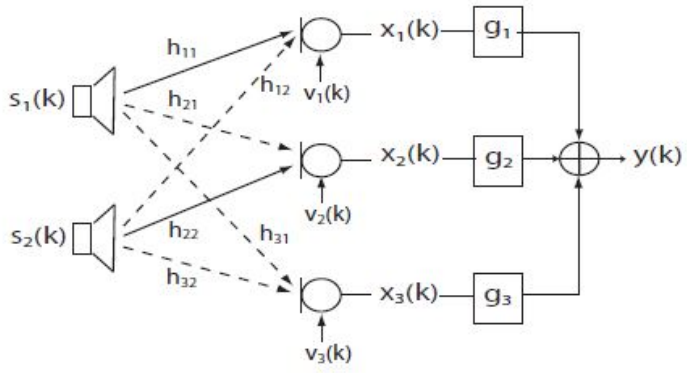

Figura 4.1: Modelo de la señal para M=2 altavoces (entradas) y N=3 micrófonos (salidas).

Considerando ahora el problema de recuperación de señales emitidas $s_m(k)$ a partir de las señales adquiridas por los micrófonos $x_n(k)$, el filtro beamforming g_n de la Figura 4.1 tiene que estar diseñado de forma que la señal de salida $y(k)$ constituya una buena estimación de la señal $s_m(k)$, es decir, que $y(k) = \hat{s}_m(k-\tau)$ nos proporcione un error mínimo. Dada una longitud máxima de L_g fija para cada uno de los filtros g_n, el ancho de banda de la señal de salida se puede expresar de forma similar a la EQ 4.2:

$$y(k) = \sum_{n=1}^{N} g_n^T x_n(k), \tag{4.3}$$

donde g_n es el vector $R^{L_g \times 1}$ conteniendo ordenadamente los filtros g_n de la Figura 4, y $x_n(k)$ es el vector columna definido como $x_n(k) = [x_n(k)x_n(k-1)\dots x_n(k-L_g+1)]^T$. Con el fin de calcular el vector completo $x_n(k)$ utilizado en la EQ 4.3 de forma matricial, la EQ 4.2 ha tenido que ser reescrita en forma compacta redefiniendo h_{nm} como matrices de Sylvester. Para más detalles consultar [23].

4.2. Algoritmos Beamforming

En [23], Benesty et al. presenta un excelente repaso de los principales algoritmos utilizados en procesamiento de señal. Debido a que ofrece un mejor rendimiento, nosotros nos centramos en este trabajo en un algoritmo basado en correlación de matrices, como es el LCMV (Linearly Constrained Minimum Variance).

El Algoritmo LCMV calcula los filtros beamforming como se puede apreciar en la EQ 4.4,

$$g^{LCMV} = \hat{R}_x^{-1} H_{:m} [H_{:m}^T \hat{R}_x^{-1} H_{:m}]^{-1} u_m, \tag{4.4}$$

donde g^{LCMV} está formado por la concatenación de los filtros g_n, esto es, $g^{LCMV} = [g_1^T \dots g_N^T]^T$, la matriz $H_{:m}$ es una partición de la matriz de impulso del canal que solo incluye las respuestas al impulso de las fuentes mth de los N micrófonos [23] utilizados en la matriz de Sylvester para las dimensiones $[NL_g \cdot Lg + Lh - 1]$. La matriz $\hat{R}_x$ es la matriz de correlación de las señales capturadas y 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, siendo ésta definida como $X \in \mathbb{R}^{[NL_g \cdot K]}$:

$$X = \frac{1}{\sqrt{k}} \begin{pmatrix} x_1(k) & x_1(k+1) & \dots & x_1(k+K-1) \\ x_2(k) & x_2(k+1) & \dots & x_2(k+K-1) \\ \vdots & \vdots & \dots & \vdots \\ x_N(k) & x_N(k+1) & \dots & x_N(k+K-1) \end{pmatrix}, \tag{4.5}$$

donde $K > NL_g$ 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 construímos directamente la matriz X^T en la representación *Column Major Order* para poder aplicar directamente estas rutinas eficientes.

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

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

Ahora nosotros definimos 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 4.4 queda de la siguiente forma:

$$g^{LCMV} = W[H_{:m}^T W]^{-1} u_m. \tag{4.6}$$

Se define la matriz Z como la solución al sistema lineal mostrado en la EQ 4.7:

$$R^T Z = H_{:m}, \tag{4.7}$$

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 4.8,

$$A b_m = u_m, \tag{4.8}$$

donde $A = H_{:m}^T W = H_{:m}^T R^{-1} Z = Z^T Z$. También aquí, la solución del sistema lineal de la EQ 4.8 es obtenida a partir de la factorización QR, en este caso, de la matriz Z. Ahora, $Z = Q' R'$ es la descomposición QR de la matriz Z, entonces, el vector b_m puede ser calculado resolviendo los dos siguientes sistemas lineales triangulares que se muestran en la EQ 4.9 y EQ 4.10.

$$R'^T y = u_m \,, \tag{4.9}$$
$$R' b_m = y \,. \tag{4.10}$$

Finalmente, es fácil deducir que el cálculo del filtro beamformer de la EQ 4.6, puede ser calculado empleando los últimos cálculos realizados, esto es, R, Z y b_m de la forma que se muestra en la EQ 4.11,

$$g^{LCMV} = R^{-1} Z b_m. \tag{4.11}$$

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

4.3. Configuración del sistema e implementación

La placa NVIDIA Jetson incorpora el chip Tegra K1 de NVIDIA, el cual, contiene a su vez una GPU basada en arquitectura Kepler. Por lo tanto, podemos emplear una CPU multi-core (4+1) y una GPU de 192 cores ensamblados en un único chip y que hacen uso de una memoria RAM común con 2 GB de capacidad. Al estar incorporada la GPU en el mismo chip y emplear la misma memoria RAM, los intercambios de datos entre CPU y GPU son a priori mucho más rápidos que en las configuraciones tradicionales donde la GPU se comunica con el host a través del bus PCIe. Además, se trata de un dispositivo de bajo consumo con un buen ratio GFlop/wattio, lo que permite desarrollar aplicaciones que después vayan a ser empleadas en dispositivos móviles.

4.3.1. Configuración del sistema

En este trabajo se han desarrollado diferentes implementaciones del algoritmo para el cálculo de los filtros beamformer que emplean la CPU quad-core Cortex-A15 a 2,32 GHz, la GPU Kepler de 192 cores a 0,85 GHz o ambas unidades computacionales (CPU y GPU). La dos unidades computacionales han sido configuradas al máximo de su potencial (modo performance) para obtener el menor tiempo computacional posible en el cálculo de los filtros beamformer. El fin que se persigue con las diferentes implementaciones realizadas para el cálculo de los filtros beamformer es poder calcularlos en tiempo real, por lo que se han empleado librerías de altas prestaciones como LAPACK, PLASMA y CUBLAS. En otras palabras, se han explorado todas las posibilidades basadas en librerías existens de altas prestaciones con el fin de potenciar el rendimiento de la aplicación y la eficiencia de los núcleos computacionales disponibles en el Jetson.

La configuración del sistema desarrollado en las pruebas es un sistema con dos altavoces situados en un lugar de la habitación que emiten dos señales diferentes de la misma duración (si no fuese así se alargaría la más corta rellenando el buffer con ceros), y tres micrófonos en otro lugar de la habitación que capturan las señales emitidas por los altavoces.

Para finalizar con esta sección, comentar que en este trabajo se analizan implementaciones que emplean para el cálculo de los filtros Beamformer: una sola CPU empleando la librería LAPACK, varias CPU's empleando la librería PLASMA y, por último, con una CPU y la GPU con LAPACK y CUBLAS conjuntamente.

4.3.2. Implementación

En las implementaciones realizadas tenemos el vector **Xmicro**, que es el vector que nos proporciona las señales capturadas por los micrófonos. A partir de los datos contenidos en el vector **Xmicro**, generamos la matriz **X** (EQ 4.5), la cual va a ser el punto de partida para calcular los filtros beamformer para cada señal que se quiera separar. Seguidamente tenemos las matrices **H**, **Z** y **A**, donde **H** es la matriz del canal. Existe una por cada altavoz (señal) que se emita en el sistema. La matriz **Z** es la matriz resultado de resolver el sistema de ecuaciones con múltiples vectores independientes (EQ 4.7), siendo **R** la QR de **matrixX**. La matriz **A** es el producto $Z^T Z = A$. Para finalizar, tenemos los vectores **u**, **b** y **g**, siendo **u** un vector que tiene ceros en todas sus componentes menos en la componente $L_g + 1$ que vale uno. El vector **b** es el vector resultado de resolver dos sistemas de ecuaciones lineales, EQ 4.9 y EQ 4.10. El vector **g**, por su parte, contiene los filtros beamformer y hay tantos vectores **g** como señales tengamos que separar. Estos vectores se calculan en mediante la EQ 4.11.

Para obtener el máximo rendimiento en el cálculo de los filtros beamformer, se ha aprovechado el hecho que diferencia el tipo de almacenamiento de los datos que hacen el lenguaje C y el lenguaje Fortran, que es el que se emplea en las librerías LAPACK y BLAS. Manejando adecuadamente el tipo de almacenamiento hemos evitamos tener que trasponer las matrices **X** y **Z** a la hora de calcular la QR de las mismas. Esto se consigue construyendo la matriz **X** directamente traspuesta. Por otro lado, también se

Algoritmo 4.1 Realiza el cociente de la raíz de k con cada uno de los valores de Xmicro a utilizar en la matriz X

requiere: ***Xmicro**, **Lx**, **MICROS**, **Ndatos**
asegurar: ***Xmicro**

```
1: float iraizk=1.0\sqrt(k);
2: float *pXmicro = Xmicro;
3: for(i=0; i<MICROS; i++){
4:   for(j=0; j<Ndatos; j++){
5:     pXmicro[j] *= iraizk;
6:   }
7:   pXmicro += Lx;
8: }
```

Algoritmo 4.2 Construir la traspuesta de la matriz X con los datos de Xmicro

requiere: ***X**, ***Xmicro**, **Lx**, **MICROS**, **Ndatos**, **Lg**
asegurar: ***X**

```
 1: float *pXmicro = Xmicro+Lg-1;
 2: float *pmatrixX = matrixX;
 3: for(i=0; i<MICROS; i++){
 4:   float *pXmicroaux = pXmicro;
 5:   for(r=0; r<Lg; r++){
 6:     for(j=0; j<K; j++){
 7:       *pmatrixX = *(pXmicroaux+j);
 8:       *pmatrixX++;
 9:     }
10:     pXmicroaux--;
11:   }
12: pXmicro += lx;
13: }
```

debe tener almacenada la matriz **H** por columnas, ya que **Z** se forma a partir de estas dos. En definitiva, hemos comenzado la optimización del algoritmo por la base: tener un acceso eficiente a memoria. A continuación desgranaremos paso a paso las distintas implementaciones realizadas para el cálculo de los filtros beamformer en los algoritmos siguientes.

En el Algoritmo 4.1 se calcula el cociente entre los valores de los datos capturados por los micrófonos y la raíz de la variable K. Se calcula la inversa de la raíz de K para así emplear multiplicaciones en vez de cocientes. Esto es debido a que las primeras tienen un coste computacional menor. Se calculan únicamente los $K + L_g$ primeros datos de cada micrófono en caso de no ser iterativo el algoritmo, y se le añade al valor anterior el producto del número de repeticiones del algoritmo por el valor de las nuevas muestras (solapamiento) que se procesan por iteración, es decir, $K + Lg + NFrames * overLap$. A este valor lo llamamos $Ndatos$. Este cálculo se realiza en una función independiente debido a que, a la hora de formar la matriz **X**, se repite varias veces el mismo valor, por lo que se gana en eficiencia al calcularlos una única vez. El valor de la variable Lx es el cociente entre el tamaño de **Xmicro** y el número de micrófonos $MICROS$.

Una vez ya tenemos los datos de la matriz **Xmicro** calculados correctamente, se procede a la construcción de la matriz **X** con dichos valores y, como hemos comentado con anterioridad, dicha construcción se realizará con la consideración de que la matriz debe ser traspuesta antes de emplearla. Esto se muestra en el Algoritmo 4.2.

El Algoritmo 4.3 muestra una función auxiliar que sirve para copiar de forma invertida los elementos de un vector en otro. Estos vectores invertidos son de un tamaño dado por un intervalo, por lo que se invierten los datos que componen cada intervalo por separado.

Para finalizar con el apartado de implementación, se muestran los algoritmos encargados del cálculo de los filtros beamformer, la descomposición QR de las matrices que lo requieran, y la modificación para

Algoritmo 4.3 Copiar un vector en otro trasponiendo los elementos dado un intervalo

requiere: ***dest**, ***source**, **m**, **n**
asegurar: ***dest**

```
1: int i,j,k,l;
2: for(i=0; i<m; i++){
3:   k=i*n;
4:   for(j=k,l=k+n-1; j<k+n; j++,l--)
5:     dest[j] = source[l];
6: }
```

hacer iterativo el cálculo de los filtros beamformer.

Para empezar, en el Algoritmo 4.4 se define la estructura de llamadas a funciones para el cálculo de los filtros Beamformer con el método LCMV. En este algoritmo se reserva la memoria de las matrices **R** y **A** como *Memoria Unificada* [9], que es un espacio de memoria común a la CPU y a la GPU; esto únicamente es así en las implementaciones que emplean ambos núcleos computacionales (CPU y GPU). El hecho de emplear la Memoria Unificada es para poder acceder indistintamente a las matrices declaradas de esta forma tanto para realizar operaciones con CPU como con GPU sobre esa misma área de memoria. De esta manera se evita la tediosa tarea de tener que realizar explícitamente las copias entre datos de CPU y datos de GPU.

Hemos definido la constante `type` para programar más fácilmente las versiones con los dos tipos de datos reales: `float` o `double`.

En este algoritmo únicamente se muestra el cálculo de los filtros Beamformer g_l dada un señal l, por lo que la porción de código a partir del comentario que indica que se va a proceder a calcular el filtro g_l debe ser ejecutado repetidamente con las diferentes matrices del canal $H_{:m}$ para calcular los filtros necesarios a aplicar a cada señal de las restantes, si es que se precisa. En nuestra implementación se repite dos veces, ya que tenemos dos altavoces emitiendo dos señales diferentes.

Seguidamente, en los algoritmos 4.5, 4.6 y 4.7 se calcula la descomposición QR de la matriz de tres maneras diferentes. En la primera se emplea la librería LAPACK. En la segunda se emplea PLASMA. En la tercera se ha realizado una implementación propia que utiliza ambas librerías donde, además, parte de los cálculos se realizan en la GPU mediante la librería CUBLAS.

Para la implementación propia, se subdivide la matriz original en bloques de nb columnas y se realizan las llamadas a las rutinas de LAPACK que emplea internamente la rutina `geqrf`, aplicando estas funciones a cada uno de esos bloques de la matriz original. Las rutinas que emplea internamente la rutina LAPACK `geqrf` [24] son: `geqrf`, `larft` y `larfb`. La rutina `geqrf` es la encargada de calcular la descomposición QR de la matriz que se le pasa como argumento; `larft` [25] es la rutina encargada de formar el factor triangular T a partir de los reflectores de Housholder calculados en la descomposición QR; y `larfb` [26] es la rutina que aplica los bloque de reflectores a la matriz que se le pasa como argumento. Esto permite realizar la descomposición QR de la matriz original a porciones, lo cual es más eficiente que calcular la descomposición QR de la totalidad de la matriz original en un solo paso.

En concreto, la rutina `larfb` de LAPACK ha sido reimplementada con llamadas a rutinas de CUBLAS para que esos cálculos los realice la GPU, de ahí que las matrices que se emplearían para cálculos en CPU y GPU han sido declaradas en la memoria unificada. El parámetro `NB` es una constante predefinida a un tamaño que determina el número de columnas que se emplean en la subdivisión de la matriz original.

Para finalizar el subapartado de implementación, se muestra la estructura algorítmica para el cálculo de los filtros beamformer g_l de forma iterativa. Lo cual permite que el algoritmo pueda ser procesado en tiempo real prefijando un retardo para que las primeras muestras a calcular sean capturadas por los micrófonos una vez los altavoces ya se encuentran emitiendo. En este algoritmo se supone que los datos capturados por **Xmicro** no se sobreescriben, ya que conocemos la duración de las señales que se van a emitir y hemos reservado el espacio pertinente para almacenarlos.

Algoritmo 4.4 Algoritmo para calcular los filtros beamformer

requiere: $^*\mathbf{X}$, $^*\mathbf{Hn}$, $\mathbf{Lg}$, $\mathbf{K}$, $\mathbf{MICROS}$
asegurar: $^*\mathbf{ygn}$, $^*\mathbf{ysn}$

```
//Declaración de variables
int M=K, N=MICROS*Lg, L=2*Lg, one=1;
type ap=1.0f, zr=0.0f;
type *R, *Z, *A, *b, *gn;
//Reserva de memoria
cudaMallocManaged(&matrixR, M*N*sizeof(type));
Z=(type*)malloc(N*L*sizeof(type));
cudaMallocManaged(&matrixA, N*L*sizeof(type));
b = (type *) calloc(L,sizeof(type)); vectorb[Lg] = 1.0;
gn = (type *)malloc(N*sizeof(type));
//Calculo QR matriz X
memcpy(R, X, M*N*sizeof(type));
calculateQR(M,N,R);
//Calculo del filtro gn para obtener sn
memcpy(Z,Hn,N*L*sizeof(type));
cblas_[d|s]trsm(CblasColMajor,CblasLeft,CblasUpper,CblasTrans,CblasNonUnit,N,L,ap,R,K,Z,N);
memcpy(A,Z,N*L*sizeof(type));
calculateQR(N, L, A);
clapack_[d|s]potrs(CblasColMajor,CblasLower,L,one,A,N,b,L);
cblas_[d|s]gemv(CblasColMajor,CblasNoTrans,N,L,ap,Z,N,b,one,zr,gn,one);
cblas_[d|s]trsv(CblasColMajor,CblasUpper,CblasTrans,CblasNonUnit,N,R,M,gn,one)
cpyvv_(ygn,gn,MICROS,lg);
for(i=0; i<MICROS; i++)
   cblas_[d|s]gemv(CblasColMajor,CblasNoTrans,M,lg,ap,&X[i*Lg*K],M,&gn[i*Lg],one,ap,ysn,one);
```

Algoritmo 4.5 Calcular QR diferentes librerías

requiere: $\mathbf{m}$, $\mathbf{n}$, $^*\mathbf{M}$
asegurar: $^*\mathbf{M}$

```
if(typeQR == LAPACK){
   type *tau = (type *)malloc(MIN(m,n)*sizeof(type));
   clapack_[d|s]geqrf(CblasColMajor,m,n,M,m,tau);
}else if(typeQR == PLASMA){
   PLASMA_desc *T;
   PLASMA_Alloc_Workspace_[d,s]geqrf(m, n, &T);
   PLASMA_[d|s]geqrf(m,n,M,m,T);
}else id(typeQR == USER){
   int nb = NB;
   [D|S]QRgpu(m, n, M, nb);
}
```

Un resumen esquemático de lo que realiza el algoritmo iterativo puede describirse con los siguientes pasos: copia de las $\mathbf{K} - \mathbf{overLap}$ columnas de datos finales de la matriz **X** actual a las $\mathbf{K} - \mathbf{overLap}$ primeras columnas de la misma matriz **X**, actualizar las $\mathbf{overLap}$ columnas finales de la matriz **X** con las nuevas muestras capturadas por los micrófonos (matriz **Xmicro**) y, por último, llamar a la función **LCMV** con la nueva matriz **X**. Estos pasos se repetirán tantas veces como secciones de la señal queramos procesar.

Algoritmo 4.6 Calcular QR empleando la GPU y la CPU

requiere: $\mathbf{m}$, $\mathbf{n}$, $^{*}\mathbf{M}$, $\mathbf{nb}$
asegurar: $^{*}\mathbf{M}$

```
//Definición de variables
char F='F', C = 'C', L = 'L', N = 'N';
int i, rep, lwork = n*292907, rows = m, cols = n;
type *T, *pM=M;
//Reserva de memoria
cudaMallocManaged(&T, nb*nb*sizeof(type));
//Calculamos el número de repeticiones en función de nb
if(n > nb) rep = n/nb;
else rep = 0;
//Bucle iterativo
for(i=0; i<rep; i++){
  cols -= nb;
  clapack_[d|s]geqrf(CblasColMajor,rows,nb,pM,m,tau);
  [d|s]larft_(&F,&C,&rows,&nb,pM,&m,tau,T,&nb);
  [D|S]larfbGPU(m,rows,cols,nb,pM,T,pM+nb);
  rows -= nb; pM += n*nb+nb;
}
clapack_[d|s]geqrf(CblasColMajor,rows,cols,pM,m,tau);
```

4.4. Análisis de resultados

Tal como se ha comentado con anterioridad, en este apartado se van a analizar diferentes implementaciones del Algoritmo **LCMV** estudiando la eficiencia en el cálculo de estas diferentes implementaciones, así como estudiando, a través de la Ley de Amdahl, el coste de cada una de las partes relevantes del código.

En las pruebas realizadas se ha experimentado con dos altavoces y tres micrófonos. Hay que notar que el número de micrófonos debe ser siempre superior al número de altavoces para que el algoritmo tenga un comportamiento adecuado. Todos los códigos han sido compilados con el compilador de NVIDIA `nvcc` empleando las opciones de compilación `-O3`, `-fopenmp`, `-lplasma`, `-lcoreblasqw`, `-lcoreblas`, `-lquark`, `-llapack`, `-lf77blas`, `-lcblas`, `-latlas`, `-lpthread`, `-lm`, `-lgfortran` y `-lcublas`. De manera resumida podemos decir que la opción `-fopenmp` es para emplear el estándar multihilo openMP, la opción `-lplasma` es para emplear la librería PLASMA que implementa las funciones de LAPACK pero empleando más de una CPU, la opción `-lquark` es para utilizar una librería encargada de crear tareas parecida a las tareas openMP [27], y la opción `-lgfortran` es para que el compilador compile el código FORTRAN con este compilador. El resto de opciones expuestas como, por ejemplo, `-llapack` es para enlazar con la librería determinada por su nombre.

Los primeros resultados a comentar son los expuestos en las tablas 4.1, 4.2, y 4.3. Estas tablas muestran el coste computacional para cada una de las partes importantes del código en las tres versiones consideradas: empleando LAPACK con una única CPU, empleando PLASMA haciendo uso de las cuatro CPU's disponibles en el Jetson, y empleando LAPACK y CUBLAS para utilizar conjuntamente una CPU y la GPU. Observando la información que muestran las tablas podemos decir que, la peor opción consiste en emplear únicamente la librería LAPACK, seguida de cerca por la implementación que hace uso de LAPACK y CUBLAS, y siendo la mejor opción la de utilizar la opción con PLASMA y cuatro CPU's.

Desgranado un poco más estos resultados se puede observar que, el mayor coste computacional está en el cálculo de la factoriación QR de la matriz X, teniendo un coste comprendido entre el 50 % y el 70 % del coste computacional total del algoritmo. El resto del coste computacional es prácticamente el coste del cálculo de los filtros beamformer g_n, asumiendo un coste entre un 12 % y un 19 % cada uno de los dos filtros que se han calculado. Dentro del coste del cálculo de los filtros g_n, gran parte del coste

Algoritmo 4.7 Calcula la función larfb de LAPACK empleando la GPU

requiere: **ld**, **m**, **n**, **k**, $^*\mathbf{V}$, $^*\mathbf{T}$, $^*\mathbf{C}$
asegurar: $*\mathbf{C}$

```
 1: //Constantes
 2: const type  ALPHA   =   1.0;
 3: const type  NALPHA  =  -1.0;
 4: //Declaración de variables
 5: int32_t i,j;
 6: cublasHandle_t handle;
 7: type *R, *W;
 8: //Reserva de memoria
 9: cudaMallocManaged(&R, n*k*sizeof(type));
10: cudaMallocManaged(&W, n*k*sizeof(type));
11: //Empiezan los cálculos
12: cublasCreate(&handle);
13: for(i=0; i<k; i++)
14:   cublas[D|S]copy(handle,n,C+(i*n),1,W+i,k);
15: cublas[D|S]trmm(handle, CUBLAS_SIDE_RIGHT, CUBLAS_FILL_MODE_LOWER, CUBLAS_OP_N, CUBLAS_DIAG_UNIT

16: ,n, k, &ALPHA, V, ld, W, n, R, n);
17: if(m > k)
18:   cublas[D|S]gemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, n, k, m-k, &ALPHA, C+(k+m), m, V+(k+m), m,

19:   &ALPHA, R, n);
20: cublas[D|S]trmm(handle, CUBLAS_SIDE_RIGHT, CUBLAS_FILL_MODE_UPPER, CUBLAS_OP_T, CUBLAS_DIAG_
21: NON_UNIT, n, k, &ALPHA, T, k, R, n, W, n);
22: if(m > k)
23:   cublas[D|S]gemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, m-k, n, k, &NALPHAF, V+(k+m), m, W, n,
24:   &ALPHA, C+(k+m), m);
25: cublas[D|S]trmm(handle, CUBLAS_SIDE_RIGHT, CUBLAS_FILL_MODE_LOWER, CUBLAS_OP_T, CUBLAS_DIAG_
26: UNIT, n, k, &ALPHA, V+(m), m, W, n, R, n);
27: cublasDestroy(handle);
28: for(i=0; i<k; i++){
29:   for(j=0; j<n; j++){
30:     C[i*n+j] -= R[i*k+j];
31:   }
32: }
```

Algoritmo 4.8 Copiar datos matriz X

requiere: $^*\mathbf{X}$, **K**, **overLap**, **MICROS**, **Lg**
asegurar: $^*\mathbf{X}$

```
1: type *pX  = X;
2: type *pX2 = &X[overLap];
3: for(j=0; j<MICROS*Lg;j++){
4: memcpy(pX, pX2, (K-overLap)*sizeof(type));
5: pX += k; pX2 += k;
6: }
```

está también en la factorización QR de la matriz A, por lo que esto nos lleva a centrar nuestros esfuerzos en conseguir optimizar el cálculo de esta factorización para ambas matrices.

La matriz **X** tiene cierta peculiaridad y es que se trata de una matriz con un número de filas bastante superior al número de columnas $m \gg n$, siendo m el número de filas y n el número de columnas. Esto nos ha llevado a plantear diferentes estrategias para poder optimizar el cálculo de esta factorización. Las dos estrategias adoptadas han sido:

Algoritmo 4.9 Construcción nueva matriz X

requiere: *X, K, overLap, MICROS, Lg, *Xmicro, Lx, iter
asegurar: *X

```
type *pXmicro = Xmicro+Lg-1+overLap*iter;
type *pX2     = &X[k-overLap];
type *pXmicroaux;
for(i=0; i<MICROS; i++){
  pXmicroaux = pXmicro;
  for(r=0; r<Lg; r++){
    for(j=0; j<overLap; j++){
      *pX = *(pXmicroaux+j);
      *pX++;
    }
    pXmicroaux--;
    pX += (K-overLap);
  }
  pXmicro += Lx;
}
```

Algoritmo 4.10 Llamar a LCMV con la nueva matriz X

requiere: *X, *H, Lg, K, *yg1, *yg2, *ys1, *ys2
asegurar: *yg1, *yg2, *ys1, *ys2

```
beamformerQRLCMV(X, H, Lg, K, yg1, yg2, ys1, ys2);
```

Tabla 4.1: ATLAS: Resultados Amdahl.

N=2 y M=3	Lg=319	Lg 319	Lg=699	Lg=699	Lg=1499	Lg=1499
Xmicro/raizk	0,0001	0 %	0,0003	0 %	0,0006	0 %
Matrix X	0,0374	1 %	0,1904	1 %	0,7923	0 %
QR(Matrix X)	3,2693	73 %	27,7862	75 %	242,1348	75 %
vector G1	0,5868	13 %	4,6272	12 %	40,1176	12 %
Vector G2	0,5822	13 %	4,6095	12 %	40,0313	12 %
Total	4,4758	100 %	37,2135	100 %	323,0765	100 %

Tabla 4.2: PLASMA: Resultados Amdahl.

N=2 y M=3	Lg=319	Lg 319	Lg=699	Lg=699	Lg=1499	Lg=1499
Xmicro/raizk	0,0001	0 %	0,0003	0 %	0,0006	0 %
Matrix X	0,0374	3 %	0,1777	1 %	0,7923	1 %
QR(Matrix X)	0,8436	60 %	7,2808	59 %	70,6306	61 %
vector G1	0,2667	19 %	2,4028	20 %	22,4592	19 %
Vector G2	0,2545	18 %	2,3808	19 %	22,1976	19 %
Total	1,4025	100 %	12,2404	100 %	116,0802	100 %

- La primera estrategia consiste en utilizar la librería PLASMA, la cual desarrolla una implementación de los núcleos computacionales matriciales basada en “teselas” (*tiles*) [28].
- La segunda estrategia está basada en una implementación propia de la factorización QR implementada en MAGMA [29]. Fue necesario implementarla ya que no se pudo instalar y configurar el MAGMA en el NVIDIA Jetson.

Como se puede apreciar en la Figura 4.2, el coste computacional del cálculo de la factorización QR

Tabla 4.3: CUBLAS: Resultados Amdahl.

N=2 y M=3	**Lg=319**	**Lg 319**	**Lg=699**	**Lg=699**	**Lg=1499**	**Lg=1499**
Xmicro/raizk	0,0001	0 %	0,0003	0 %	0,0005	0 %
Matrix X	0,0376	1 %	0,1757	1 %	0,7537	0 %
QR(Matrix X)	2,5209	69 %	16,5049	68 %	122,2906	67 %
vector G1	0,5496	15 %	3,7781	16 %	30,4592	17 %
Vector G2	0,5692	15 %	3,8522	16 %	30,1976	16 %
Total	3,6775	100 %	24,3112	100 %	183,7015	100 %

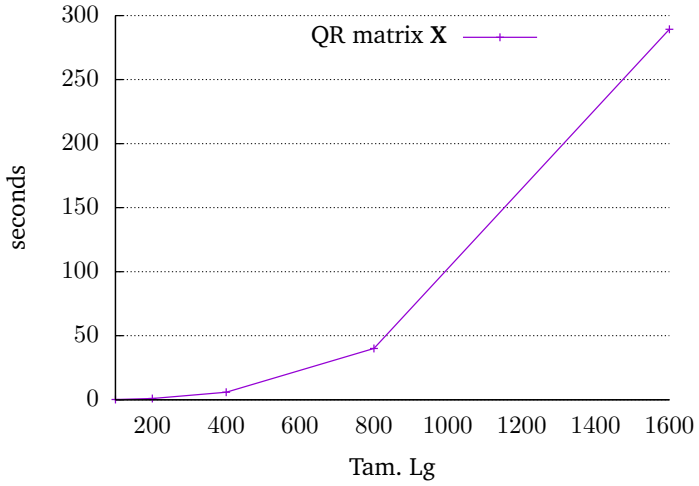

Figura 4.2: Evolución tiempo ejecución cálculo QR de la matriz X dado un incremento de Lg.

de las matrices no crece proporcionalmente al tamaño en datos de las mismas, sino que lo hace en mayor medida. Esto es debido a que el coste computacional de una factorización QR es del orden de $O(2n^2m)$ flops, siendo m el número de filas de la matriz y n el número de columnas. Debido a este comportamiento en el aumento de los datos, se emplean en el cálculo de la factorización QR de una matriz estrategias basadas en la división de la matriz original y el cálculo en CPU o GPU dependiendo de las características de las operaciones a aplicar.

Capítulo 5

Conclusiones y trabajo futuro

En esta tesis de máster se han tratado dos problemas del procesado de señales digitales acústicas. Ambos problemas, aunque diferentes, tienen en común su importancia en la búsqueda de un mejor ambiente sonoro, pero también el dispositivo sobre el que se ha trabajado.

Por un lado, se han implementado y estudiado tres estructuras de filtrado (FIR, IIR y PIIR) para señales digitales. Las operaciones de filtrado son básicas, es decir, operaciones aritméticas esenciales realizadas sobre vectores de mayor o menor dimensión. Para realizar estas operaciones de manera eficiente hemos utilizado las instrucciones intrínsecas NEON de los procesadores ARM. Los resultados comparativos entre la autovectorización por parte del compilador `gcc` y las implementaciones realizadas empleando instrucciones NEON muestran una gran mejora en la implementación utilizando explícitamente el uso de las instrucciones NEON frente a la autovectorización que proporciona el compilador. Esta mejora es importante, llegando a ser en varios casos superior a 4 veces el tiempo de la versión autovectorizada. Esto tiene que ver con el hecho de que la autovectorización proporcionada por el compilador `gcc` no ofrece una buena ganancia con respecto a la versión no vectorizada. Nuestra implementación permite su utilización en tiempo real. Por ejemplo, se pueden aplicar aproximadamente 125 filtros IIRI y 260 FIR para un número de coeficientes de filtro de 256 y con el tipo de datos `INT16` antes de que esté disponible el siguiente *frame* de datos de entrada a procesar. Además, se muestra en los códigos desarrollados que el tipo de datos `INT16` es más eficiente que el tipo de datos `FLO32`, esto es debido a que el coste computacional de una operación con números reales es más costosa que una operación con números enteros. A tenor de estos resultados, se podría concluir que es mejor emplear el tipo de datos `INT16`, siempre y cuando la precisión en los cálculos nos lo permita. Esto es así, no solo porque es más eficiente en cuanto a coste computacional que el tipo de datos `FLO32`, si no también porque es el tipo de datos genérico para formatos de audio digital como los CD's de música.

Por otra parte, se han desarrollado varias implementaciones del Algoritmo **LCMV** para el cálculo de filtros Beamformer. Este caso es diferente ya que las operaciones que involucra el algoritmo son de más alto nivel, es decir, operaciones de álgebra lineal numérica sobre matrices (descomposiciones matriciales, resolución de sistemas lineales, etc.). En este caso, lo apropiado era acudir a la utilización de librerías que resuelven este tipo de problemas de manera eficiente sobre arquitecturas parecidas basadas en multicore y manycore. Esto ha tenido una implicación natural en el trabajo consistente en la instalación, adaptación y comprobación de las librerías sobre la máquina objeto de estudio. La "inmadurez" del dispositivo y, especialmente, de su software, ha supuesto un trabajo de investigación añadido nada despreciable, pero ha permitido, al mismo tiempo, obtener un buen rendimiento de este novedoso dispositivo en lo que se refiere a la solución de este tipo de problemas. En particular, hemos comenzado por realizar un estudio del coste computacional de cada una de las partes de las que se compone el algoritmo. Hemos concluido que una gran parte del coste computacional del algoritmo viene dado por el cálculo de las factorizaciones QR matriciales previas a la resolución de los sistemas de ecuaciones lineales subsiguientes. En concreto, la factorización QR de la matriz X es la que mayor coste tiene, estando este coste comprendido entre un 60 % y un 70 % del coste total del algoritmo para un cálculo de dos filtros Beamformer (g_1 y g_2). En vista de estos resultados se han implementan diferentes estrategias para calcular eficientemente las factorizaciones QR, utilizando los distintos recursos computacionales del NVIDIA Jetson. Se concluye, a

raíz de los tiempos tomados, que lo más eficiente es emplear la librería PLASMA que dispondrá de los cuatro núcleos computacionales ARM para calcularla.

En relación al Capítulo 3 y como línea futura de investigación, se podría estudiar la conveniencia de emplear el tipo de datos *INT16* o *FLO32* en varias aplicaciones reales con diferentes requisitos de precisión y tiempo computacional, para poder concluir con una aplicación real que tipo de datos es más conveniente y mostrar resultados concretos para estas aplicaciones.

Con respecto al problema tratado en el Capítulo 4, cabe decir que una de las tareas pendientes de realizar en este apartado es el realizar un algoritmo que emplee todos los núcleos computacionales disponibles, es decir, las cuatro CPU's ARM y la GPU Kepler. Esto lo conseguiremos conjugando el calculo de las QR con PLASMA y el cálculo de la rutina larfb en GPU; dicha implementación no ha podido ser incluida en esta memoria por falta de tiempo, pero ya se esta trabajando en ella y en futuros trabajos se mostraran los resultados. Además, como futura línea de investigación, se podría proponer un algoritmo colaborativo entre diferentes Jetson que intercambien información empleando el estándar de intercambio de mensajes MPI, de tal modo que los Jetson colaboren entre si intercambiando parte de la información procesada para mejorar la calidad y/o el tiempo de procesado para la separación de la señal que se quiere.

Bibliografía

[1] ARM NEON, www.arm.com/, (accessed 2015 February 23).

[2] M. FLYNN, *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput **C-21** (1972) 948–960.

[3] INTEL SSE, https://software.intel.com/sites/landingpage/IntrinsicsGuide/, (accessed 2015 September 17).

[4] J. RÄMO, V. VÄLIMÄKI AND BALÁZS BANK, *High-Precision Parallel Graphic Equalizer*, IEEE Transactions on Audio, Speech and Language Processing **22** (2014) 1894–1904.

[5] J.C. RISSET, *Computer Music Experiments 1964*, Computer Music J. **22** (1985) 11–18.

[6] Y. HUANG, J. CHEN AND J. BENESTY, *Inmerse Audio Schemes*, IEEE Signal Processing Magazine **28** (2011) 20–32.

[7] J. LORENTE, G. PIÑERO, A.M. VIDAL, J.A. BELLOCH AND ALBERTO GONZÁLEZ, *19th European Signal Procesing Conference*, Parallel Implementations of Beamforming Design and Filtering for Microphone Array Applications (29 August - 02 september 2011).

[8] NVIDIA JETSON TK1, https://developer.nvidia.com/jetson-tk1/, (accessed 2015 February 10).

[9] CUDA, http://www.nvidia.es/object/cuda-parallel-computing-es.html, (accessed 2015 September 17).

[10] BLAS Library, http://www.netlib.org/blas/, (accessed 2015 February 15).

[11] LAPACK Library, http://www.netlib.org/lapack/, (accessed 2015 February 15).

[12] ATLAS Library, http://math-atlas.sourceforge.net/, (accessed 2015 February 15).

[13] PLASMA Library, http://icl.cs.utk.edu/plasma/, (accessed 2015 March 15).

[14] CUBLAS Library, http://docs.nvidia.com/cuda/cublas/, (accessed 2015 April 25).

[15] Nvidia Jetson Quick Start Guide, http://developer.download.nvidia.com/embedded/jetson/TK1/docs/2_GetStart/Jeston_TK1_User_Guide.pdf, (accessed 2015 September 04).

[16] A. V. OPPENHEIM, A. S. WILLSKY, AND S. HAMID *Signals and systems*, Processing series. Prentice Hall, 2nd edition, 1997.

[17] L. R. RABINER AND B. GOLD, *Theory and Application of Digital Signal Processing* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1975.

[18] ARM NEON intrinsics, gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/ARM-NEON-Intrinsics.html, (accessed 2015 July 12).

[19] Advanced Linux Sound Architecture (ALSA), www.alsa-project.org/, (accessed 2015 January 08).

[20] Steinberg Media Technologies GmbH, www.steinberg.net/, (accessed 2015 Aug. 14).

[21] S. B. HOLGERSSON, *Optimising IIR filters using ARM NEON*, Master Thesis of University of Danemark (2012).

[22] ARM NEON auto-vectorization, gcc.gnu.org/onlinedocs/gcc/ARM-Options.html, (accessed 2015 July 22).

[23] D. THEODOROPOULOS, G. KUZMANOV AND G. GAYDADJIEV, *Multi-core Platforms for Beamforming and Wave Field Synthesis* IEEE Transactions on Multimedia **99** (2010).

[24] Source DGEQRF.c, http://www.netlib.org/clapack/CLAPACK-3.1.1/SRC/dgeqrf.c, (accessed 2015 September 18).

[25] Source DLARFT.c, http://www.netlib.org/clapack/CLAPACK-3.1.1/SRC/dlarft.c, (accessed 2015 September 18).

[26] Source DLARFB.c, http://www.netlib.org/clapack/CLAPACK-3.1.1/SRC/dlarfb.c, (accessed 2015 September 18).

[27] QUARK Library, http://icl.cs.utk.edu/quark/, (accessed 2015 September 18).

[28] QR factoritation PLASMA, http://www.netlib.org/lapack/lawnspdf/lawn222.pdf, (accessed 2015 September 18).

[29] QR factoritation MAGMA, http://www.netlib.org/lapack/lawnspdf/lawn233.pdf, (accessed 2015 September 18).