Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Implementación en DSP Hexagon de la transformada discreta de Radon

Discrete Radon transform implementation on Hexagon DSP

Gabriel Alejandro Rodríguez De Abreu

La Laguna, 12 de marzo de 2021

D. **José Gil Marichal Hernández**, con N.I.F. 78.677.406-H profesor contratado doctor adscrito al área Teoría de la Señal y Comunicaciones, del Departamento de Ingeniería Industrial de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

”*Implementación en DSP Hexagon de la transformada discreta de Radon*”

ha sido realizada bajo su dirección por D. **Gabriel Alejandro Rodríguez De Abreu**, con N.I.F. 79.062.580-B.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 12 de Marzo de 2021

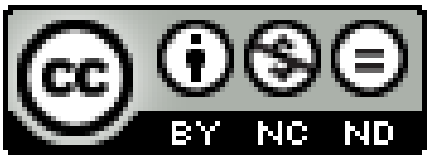
Agradecimientos

A Jose Marichal por haberme permitido participar en esta experiencia única; así como por su tiempo, ayuda y paciencia.

A todos en Wooptix por acogerme y aportarme su conocimiento y cooperación.

Finalmente a mi familia por apoyarme en el transcurso de mi vida académica.

Licencia



Resumen

La presencia de teléfonos móviles con una gran potencia es una realidad innegable, no obstante con este incremento de potencia vemos que las expectativas para estos dispositivos han crecido, por lo que algoritmos diversos para complacerlas surgen a diario, de creciente complejidad y que consumen cada vez más recursos. Cuando buscamos potencia de cómputo es usual usar como coprocesador especializado la unidad de procesamiento gráfico, GPU. Pero éstas están diseñadas con vistas a maximizar el rendimiento, sacrificando el consumo de energía.

En este trabajo de fin de grado estudiamos si los Procesadores Digitales de Señales, DPSs, con su potencial para hacer operaciones vectoriales pueden rivalizar en desempeño con las GPUs en caso de que el algoritmo exhiba potencial para la vectorización.

Con este fin vamos a utilizar los DSP de la gama HEXAGON, y la extensión vectorial de los mismos, Hexagon Vector eXtensions, HVX.

Estos DSPs los podemos encontrar en algunos procesadores de la familia Snapdragon, del fabricante Qualcomm. El cual prácticamente monopoliza el mercado de procesadores, System On Chips, SoCs *para móviles Android. En específico usaremos el kit de desarrollo hardware, HDK, del Snapdragon 845, que fue lanzado al mercado en 2019.*

El algoritmo al que aplicaremos este estudio es a la transformada discreta aproximada de Radon, que realiza todas las posibles integrales de línea en una imagen.

Palabras clave: DSP, HEXAGON, Snapdragon, HVX

Abstract

The presence of mobile phones with great computing power is an undeniable reality, however with this increase in power we see that the expectations for these devices have grown. Numerous algorithms arise daily to meet these expectations, with the cost of greater consumption and complexity.When we are looking for computing power, it is usual to use the graphics processing unit, GPU, as a specialized co-processor. But these are designed with a view to maximizing performance, sacrificing power consumption.

In this project we will study whether Digital Signal Processors, DSPs, with their potential to perform vector operations, can rival the performance of GPUs in case the algorithm exhibits potential for vectorization.

For this purpose we are going to use the DSPs of the HEXAGON family, in conjunction with the Hexagon Vector eXtension, HVX.

These DSPs can be found in some processors of the Snapdragon processors, made by Qualcomm. Which practically monopolizes the market of processors, System On Chips, SoCs *for Android mobiles. Specifically we will use the hardware development kit, HDK, of the Snapdragon 845, which was released in 2019.*

The algorithm that we will study is the approximate discrete Radon transform, which performs all possible line integrals on an image.

Keywords: DSP, HEXAGON, Radon transform, DRT, Snapdragon, HVX

Índice general

Índice de Figuras

Listings

Índice de Tablas

Capítulo 1

Introducción

1.1. Objetivos

En este apartado se expondrán los objetivos que se buscan lograr en este proyecto.

1.1.1. Implementación del algoritmo

El primer objetivo que se pretende conseguir en este proyecto es la implementación de la transformada aproximada, o multiescala, Discreta de Radon(1), en el DSP de cómputo de nuestro HDK. Se pretende ajustar el tiempo para alcanzar estándares cercanos al tiempo real pensando en que el algoritmo pueda aceptar la entrada de un vídeo de un dispositivo móvil en un futuro. El algoritmo será implementado de tal forma que podamos hacer uso de la extensión vectorial de Hexagon.

1.1.2. Firma en dispositivos de producción

El segundo objetivo de este proyecto es explorar las alternativas que tiene un desarrollador corriente para tener acceso a los recursos que dota Qualcomm para poder programar y desplegar una aplicación que haga uso del DSP.

1.2. Antecedentes y Estado actual del arte

1.2.1. Desarrollo en DSP

En la actualidad, los DSPs abarcan más que el procesamiento de señales convencionalmente ligado al multimedia. Factores que contribuyen a que esto sea así son, en líneas generales, que el coste de un DSP suele ser más reducido que el de otros tipos de procesadores y que su consumo energético tiende a ser menor también. Esto no quiere decir que los DSPs no puedan presentar una potencia de cómputo interesante, al nivel de poder rivalizar con las CPUs y GPUs en algunos tipos de operaciones. Esto implica

también una puerta abierta a atajar problemas de mayor complejidad al tener disponible una nueva unidad en la que procesar información, conjuntamente con el resto de unidades de procesamiento del SoC: las ya comentadas CPU y GPU, e incluso también la unidad de procesamiento ligada al sensor de imagen: ISP, *image signal processor*.

Un campo que ha tomado especial interés en el uso de DSP es el de procesamiento de imágenes médicas (2). Si bien el motivo de uso no tiene por qué compartirse con el nuestro, ya que en nuestro caso queremos aprovechar las operaciones vectoriales, mientras que en otros DSPs suele ser por el rendimiento en hacer sumas de multiplicaciones, si es apreciable que su utilidad no sea restringida a lo que normalmente se considera su trabajo.

1.2.2. Móviles con procesador Snapdragon

No son pocos los móviles que tienen un procesador Snapdragon 845 con un DSP de cómputo integrado. Suponía la gama alta de procesadores anunciada a finales de 2018 y comercializada durante 2019. Entre ellos destacan ciertos dispositivos como el Xiaomi Pocophone F1 (3) o el Google Pixel 3 (4). Si Qualcomm ofrece una opción para el desarrollo Hexagon en estos dispositivos supondría un incremento notable en sus prestaciones y haría que poseer un cDSP (Compute DSP) sea una característica preciada.

1.2.3. Desarrollo en Hexagon

El desarrollo en Hexagon parece poco explorado, la mayoría de aplicaciones que se ven desarrolladas y desplegadas se mantienen ocultas bajo acuerdos de confidencialidad cuando se firma con los fabricantes, por lo que la mayoría de ejemplos que vemos rondando son pertenecientes a Qualcomm, y de entre ellos la mayoría son demostraciones. Las ramas principales de desarrollo que aparecen recurrentemente cuando se habla de Hexagon son:

- Visión por computador.
- Machine Learning.
- Procesado de imágenes.

Ejemplos de librerías y productos públicos que hacen uso del DSP son:

- FastCV (Fast Computer Vision) (5)
- TensorFlow Lite (6) permite cargar kernels en el DSP mediante el uso de un delegado. Se puede usar en Hexagon 68X y 690.
- Aplicación de ejemplo para hacer histogramas de imágenes en el DSP. En este caso hablamos de una aplicación que está pensada para funcionar en el SDK, y no desplegada. Esta aplicación es un proyecto desarrollado por Bsquare(7).

Capítulo 2

Herramientas y recursos

2.1. Hexagon SDK

2.1.1. Overview

El SDK de Hexagon aporta al programador casi todas las herramientas que va a necesitar para programar en el DSP de Hexagon. Se puede obtener desde la página de Qualcomm en el apartado de Hexagon (8). Algunos recursos incluidos en este SDK son:

- Librerías. Se incluyen las diversas librerías de las que hacer uso programando en Hexagon. Van desde librería encargadas de la comunicación entre CPU y DSP, manejo de hilos, instrucciones intrínsecas de ensamblador para las unidades escalares y vectoriales y más.
- Scripts en pyhton2 se encuentran en el SDK para facilitar la creación, manejo y configuración de proyectos Hexagon. De entre los más útiles están `clone_project.py` y `testsig.py`. Cuando se usan estos scripts hay que tener en cuenta que pueden tener características obsoletas así que existe la posibilidad de que tengas que modificarlos levemente.
- IDE eclipse para Hexagon. No es obligatorio instalarlo cuando se descarga el SDK, dependerá de la preferencia personal.
- Ejemplos de códigos para Hexagon.
- Documentación completa de Hexagon. Parte de la documentación de Hexagon sólo se puede encontrar en el SDK (9).
- Herramientas para compilar código de Hexagon.
- NDK para android. Esto no es obligatorio instalarlo, pero simplifica el proceso de configuración del SDK.
- Otras herramientas útiles como `adsp_ps1`, una aplicación para poder ver los procesos que están usando fastRPC en el dispositivo o la utilidad para hacer las firmas entre otras.

2.1.2. Instalación

Para descargar el SDK de Qualcomm (8) primero tendremos que disponer de una cuenta de Qualcomm. Esto se hizo para reducir los posibles ataques que agentes maliciosos podrían lanzar en caso de tener acceso libre al SDK. Es por esto que puede que se tarde un rato en validar la cuenta. Una vez hayamos descargado y descomprimido el SDK sólo tendremos que seguir las instrucciones de instalación. Llegados a un punto nos preguntará para instalar Eclipse como IDE y el NDK de android **Figura** 2.2.

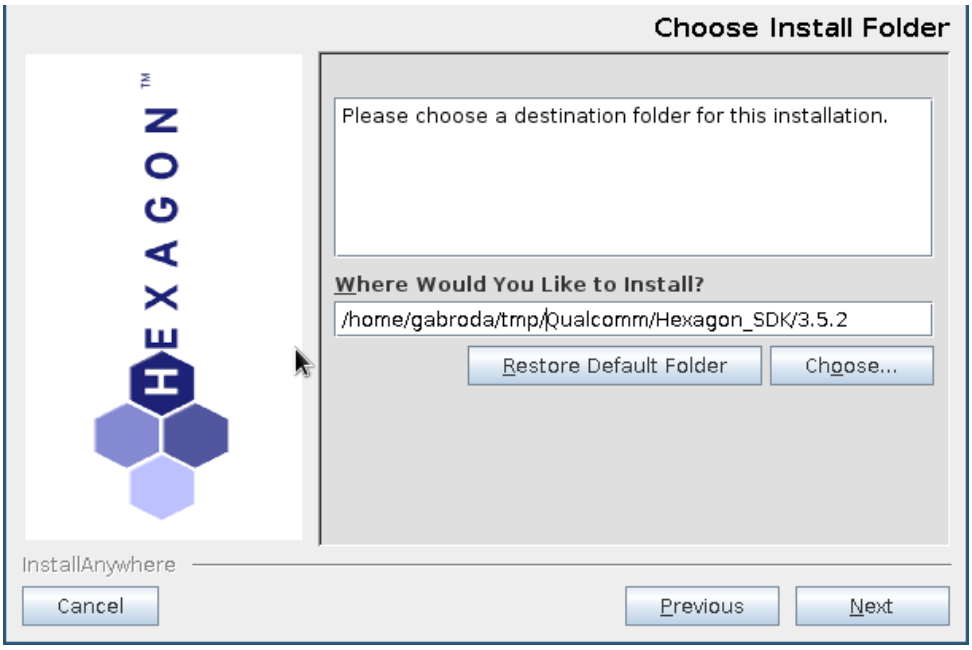


Figura 2.1: Elección de ruta

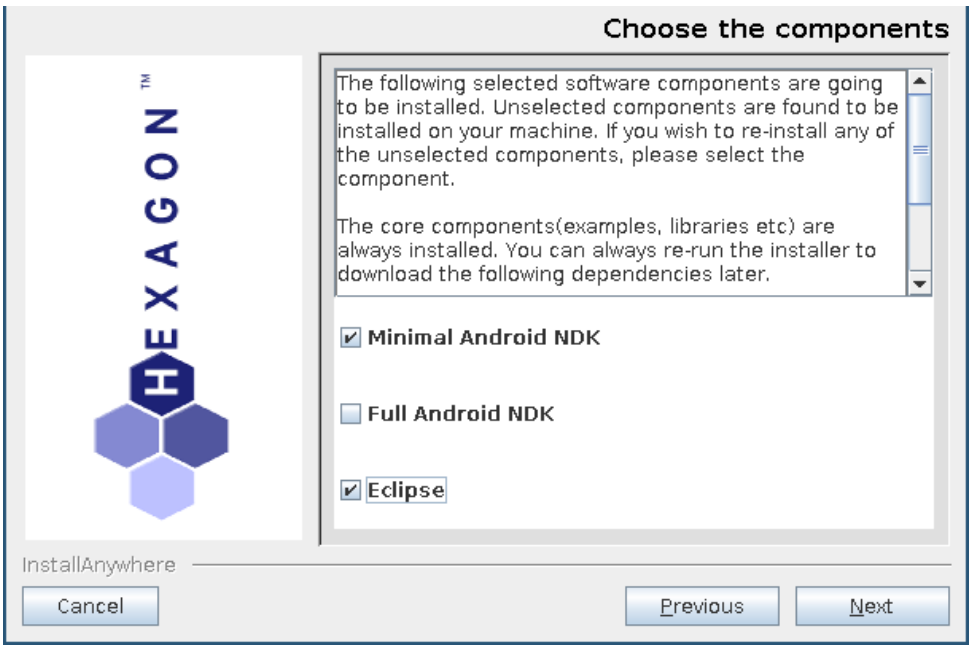


Figura 2.2: Instalar componentes

Si bien ninguno de estos 2 son obligatorios, es recomendable la instalación del NDK mínimo de android para reducir los problemas en las rutas de búsqueda al compilar.

Tras terminar esta instalación tendremos el SDK en la ubicación que seleccionamos en la instalación si no se produjeron problemas, no obstante hay que recordar exportar las variables de entorno cada vez que iniciamos una nueva sesión en una shell. Para ahorrarnos este problema podemos ejecutar el fichero de configuración `/{HEXAGON_SDK_ROOT}/setup_sdk_env.source` en nuestro `.bashrc`.

Con esto tendríamos ya el SDK instalado y estaríamos listos para empezar a programar, pero hay un paso más que realizar si queremos ejecutar código en el HDK, y es firmar el dispositivo. Para ello podemos utilizar el script que viene con el SDK en `/{HEXAGON_SDK_ROOT}/scripts/testsig.py`. Como se comentó anteriormente, los scripts proporcionados por el SDK están programados en python2, y algunos tienen partes obsoletas de código que pueden dar problemas. En el caso del script de firma va a ser necesario que introduzcas manualmente el número del dispositivo en lugar de obtenerlo con `adb devices` ya que parece no estar funcionando correctamente cuando se ejecuta en el script. También puedes arreglar el código, pero al ser un script que no vayas a usar continuamente podemos obviarlo. Una vez producida la firma de prueba sólo es cuestion de enviarla al DSP como se indica en el **listado 2.1**:

Listado 2.1: Cargar firma

```
$ adb push testsig-0xXXXX.so /vendor/lib/rfsa/adsp
```

2.1.3. Primeros pasos

En la documentación de Hexagon (10) podemos encontrar una guía que nos llevará paso a paso en lo referente a la ejecución del primer proyecto, de todas formas en este apartado se muestra cómo se compila, **listado 2.2**; carga y ejecuta un proyecto en el HDK, **listado 2.3**, tomando como referencia el código del algoritmo.

Listado 2.2: Compilación

```
$ make tree V=android_Debug
$ make tree V=hexagon_Debug_dynamic_toolv83_v65
```

Listado 2.3: Ejecución

```
$ adb push android_Debug/ship/PIDRT /vendor/bin
$ adb push android_Debug/ship/libPIDRT.so /vendor/lib
$ asb push \
   hexagon_Debug_dynamic_toolv83_v65/ship/libPIDRT_skel.so \
   /vendor/lib/rfsa/adsp

$ adb shell /vendor/bin/PIDRT 2 2 0
```

En caso de querer crear un nuevo proyecto lo más recomendado es hacer uso del script `/{HEXAGON_SDK_ROOT}/scripts/clone_project.py` para crear el proyecto a partir de otro, eligiendo normalmente el ejemplo que más se ajuste a los requerimientos de tu proyecto de entre todos los ejemplos del SDK. Una vez el proyecto esté clonado puedes ajustar sus dependencias modificando los ficheros `hexagon.min` y `android.min`.

2.2. Firmas

2.2.1. Proceso de firmado

Por motivos de seguridad, ya que la sección en la que se ejecuta el código de DSP podría ser vulnerable a ataques, Qualcomm protege el sistema haciendo obligatorio el uso de firmas para todos los `.so` que se quieran cargar en los dispositivos móviles o tablets del OEM. Las firmas son un mecanismo para añadir una firma criptográfica a los módulos dinámicos para que el loader pueda verificarlos cuando se ejecuten. Se emplean 2 tipos de firmas con este fin:

- Firmas de prueba: Estas firmas sólo nos permiten cargar módulos a dispositivos de prueba como nuestro HDK. Estan pensadas para que se pueda prototipar con los entornos de desarrollo. Este tipo de firmas no serán aceptadas en un dispositivo comercial, ya que los fabricantes de móviles, *Original Equipment Manufacturer, OEM*, bloquean esta posibilidad para evitar que los desarrolladores carguen módulos sin pasar primero por un contrato, certificando así que el módulo no será maligno.
- Firma de producción: Esta firma nos permite cargar módulos dinámicos a dispositivos de fabricantes. Para obtener este tipo de firma necesitamos que sea el OEM el que nos la aporte, por lo que la única forma que tiene un desarrollador, *independent software vendor, ISV*, de desplegar una aplicación firmada es mediante acuerdos con los fabricantes. La idea de Qualcomm es que los desarrolladores usen los kits de desarrollo para producir prototipos que mostrar a los fabricantes para conseguir acuerdos.

Según la documentación de Qualcomm en lo referente a las firmas, se introdujo un método de carga de objetos dinámicos compartidos sin firma al DSP mediante el uso de los procesos *Unsigned PD* a partir de la familia SM8150. Estos procesos tienen pocos derechos, pero tienen acceso a los DSP y mantienen ciertas funcionalidades. Las funcionalidades que mantienen los unsigned PD son:

- Creación y servicios de hilos.
- Creación y servicios de timers.
- Contextos HVX.
- Control de frecuencia del reloj.
- VTCM (memoria local HVX de alto rendimiento).
- Operaciones de caché.
- Mapeo y aplicación de memoria HLOS .

Los Unsigned PD ven limitada:

- Su capacidad de acceso a ciertos drivers como podrían ser UBWC/DMA o Camera Streamer
- El Tope de prioridad de los hilos que despliegan está limitado a 64.
- El máximo número de hilos que permite un Unsigned PD es de 128.

Capítulo 3

Hardware

3.1. Hexagon 685

El SoC Snapdragon 845 cuenta con un Hexagon 865 como DSP. El cual trabaja con una frecuencia de reloj que ronda los 940MHz, cuando opera nominalmente, pero puede alcanzar hasta 1190MHz como frecuencia tope. Si comparamos la frecuencia de reloj del DSP con la de los núcleos de CPU, Qualcomm Kryo 385 para el SD845, observamos que este último tiene más del doble de frecuencia que el DSP, llegando a alcanzar los 2.8GHz (11). Esto nos deja claro que el DSP debe competir explotando sus operaciones vectoriales y su pipeline[1] para lograr producir una mayor tasa de ejecución de tareas, *throughput*.

3.1.1. Unidad escalar

La unidad escalar del DSP Hexagon 685 posee 32 registros de propósito general que cuentan con una anchura de 32 bits. Para hacer operaciones que requieran de más anchura la arquitectura provee de la posibilidad de hacer operaciones usando pares de registros alineados para formar un ancho de palabra de 64 bits. Además de los registros de propósito general, cuenta con una serie de registros de control entre los que encontramos al *Program Counter, PC*, registros de estatus, registros predicados, etc. El DSP esta dotado de 4 unidades funcionales, 2 de ejecución y 2 de carga y almacenamiento. Esto le permite usar *Very Long Instruction Words, VLIW*, permitiendo paquetes de hasta 4 instrucciones por ciclo. Todo esto podemos verlo reflejado en la **Figura 3.1**

3.1.2. Carga de tareas

CPU y DSP se comunican mediante interrupciones de memoria compartida. Esta transacción puede tomar desde unos cientos de microsegundos hasta milisegundos dependiendo de la configuración. Es por esto que es preferible reducir el número de

[1] Pipelining: técnica para ejecutar simultáneamente varias tareas, a nivel de instrucciones, en un único procesador.

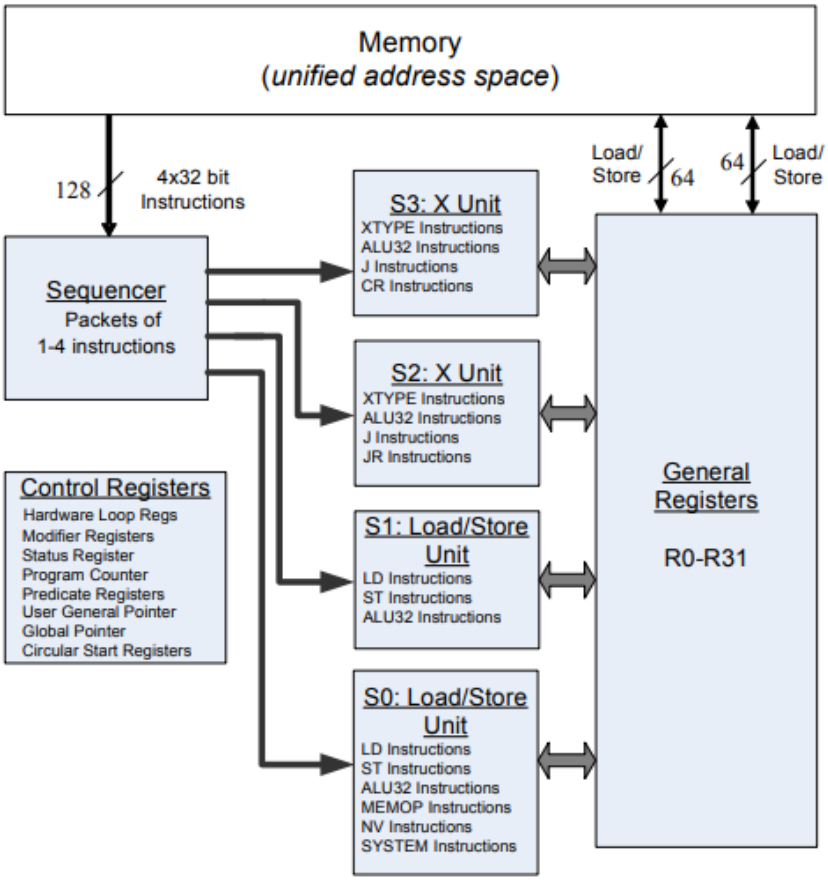


Figura 3.1: Arquitectura escalar de la máquina

invocaciones y hacer llamadas para completar tareas mayores, minimizando así los tiempo muertos en los que, de otra forma, estaríamos incurriendo.

El medio principal con el que se comunican la aplicación y el DSP es FastRPC o *Fast Remote Procedure Calls*. FastRPC se encargará entonces de supervisar el intercambio de información y organizará los parámetros que vayan a ser utilizados entre ambos procesadores según la información que les es suministrado a través los archivos IDL, *Interface Description Language*, ver la **Figura 3.2**.

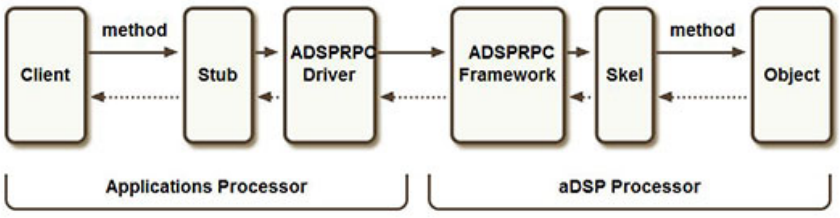


Figura 3.2: Procesos de comunicación con DSP

3.1.3. Hexagon Vector eXtension, HVX

HVX utiliza un coprocesador con habilidad de procesar vectores de 128 bytes, o, en familias anteriores a la del 685, de 64 bytes, accedidos a través de ficheros de registro HVX denominados contextos HVX. Se pueden acceder a los datos de la unidad escalar mediante el uso de una caché de nivel 2 compartida entre ambas unidades, ver **Figura 3.3**.

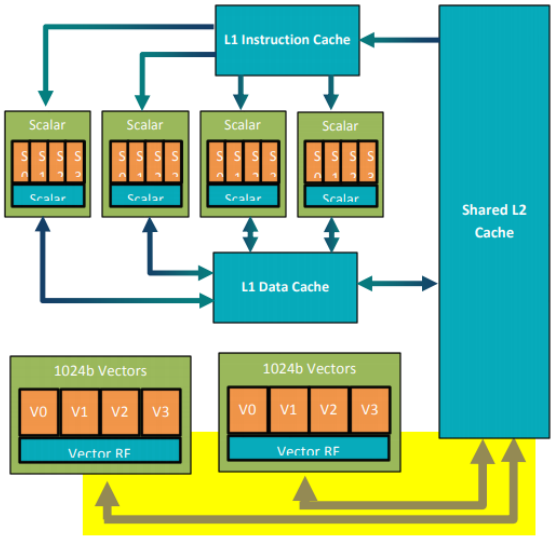


Figura 3.3: Diagrama de memoria y unidades funcionales

Como se puede observar en la **Figura 3.4** el Hexagon 685 posee 32 registros de vectores con una anchura de 1024 bits con operaciones para bytes, medias palabras y palabras enteras. Estos registros se pueden emparejar, existiendo también operaciones de parejas de vectores.

Aparte de los registros de vectores, posee 4 registros de predicados de vectores, cada uno con un ancho de 128 bits.

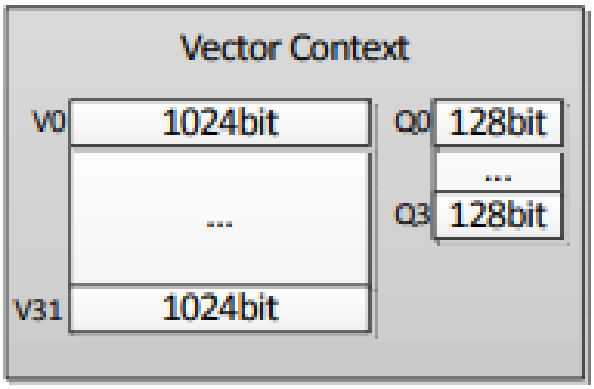


Figura 3.4: Contexto de Vectores

Capítulo 4

Desarrollo

4.1. Transformada Discreta de Radon

La transformada discreta de Radon multiescala, o aproximada, según el autor, es una transformada integral que fue propuesta en los años 90 (1; 12; 13) por diversos autores.

De esta transformada se ha propuesto una mejora en un congreso internacional (14), donde he participado como coautor, consistente en exponer más paralelismo, de grano fino y grueso.

La idea principal de la transformada discreta de Radon convencional es que podemos expresar las líneas discretas como unión de los segmentos que la componen. Por lo tanto, el cálculo sobre la línea es equivalente al cálculo de sus segmentos. Esto permite que nos acerquemos al problema con una aproximación divide y vencerás. Consistente en comenzar computando todas las posibles integrales (sumatorios) sobre rectas de longitud 2, sumando dos píxeles coalescentes de la entrada. Luego las rectas de longitud 4 sumando dos segmentos de longitud 2, etcétera.

En esta memoria se usará la modificación de la transformada presentada en el congreso ICTCE. Vamos a eliminar la limitación del cálculo por cuadrantes que tenía la transformada multiescala original, y con esta mejora vamos a poder aplicar la transformada a la imagen con únicamente dos pasadas. Una horizontal y otra vertical, en lugar de las 4 rotaciones para calcular los cuatro bloques de 45 grados por cuadrante, que requería originalmente esta transformada. El principio de funcionamiento de las pasadas horizontal y vertical no cambia, pero debemos rotar 90°la entrada, entre una pasada y otra.

Incorporo a continuación un resumen de la aportación al congreso donde se describe con mayor detalle la mejora sugerida para exponer mejor el paralelismo de la DRT (14).

4.1.1. DRT con paralelismo mejorado

La transformada discreta de Radon se diseñó originalmente para calcular simultáneamente todas las sumas de los datos situados en líneas discretas que tocan al menos un

punto de una imagen mientras se proyectan en una semicircunferencia alrededor de ella. Pero debido a la forma en que describía las líneas, en forma de intercepción-pendiente en lugar de trabajar con líneas generales de la forma $x \cdot \cos(\theta) + y \cdot \sin(\theta) = d$, el algoritmo sólo podía considerar simultáneamente fracciones de 45 grados: pendientes de 0 a 1 o -1; relacionando x con y, o y con x.

El algoritmo DRT convencional resolvía esas fracciones de 45 grados, llamadas cuadrantes, con una complejidad linealítmica, $O(N \log(N))$ para entradas de tamaño $N \times N$, con N potencia de dos. Pero que había que aplicar cuatro veces a versiones rotadas de la entrada para obtener los resultados de los 4 casos que surgen –líneas de la forma $y = x \cdot slope + d$, $y = x \cdot -slope + d$, $x = y \cdot slope + d$, y $y = x \cdot -slope + d$– para obtener un sinograma discreto completo que cubra los 180 grados de proyección.

Si bien se trata de un algoritmo muy valioso, la existencia de esta barrera que aísla el cálculo de cada uno de los 4 cuadrantes, es uno de los inconvenientes más destacados, según nuestra experiencia, para obtener el máximo rendimiento.

Por ello, proponemos explorar alternativas a un único algoritmo que aborde los cuatro cuadrantes de forma separada, buscando en su lugar un algoritmo que considere las líneas orientadas verticalmente, u horizontalmente, de forma simultánea; y luego expresar el paralelismo de forma que sea más fácil de exponer para que los lenguajes paralelos lo aprovechen a nivel de datos. Mientras tanto, deben conservarse dos características capitales de los DRT multiescala: no debe haber interpolación; y debe maximizarse la reutilización de las sumas parciales. En otras palabras, no queremos tratar con fracciones ni pesos; y queremos que el algoritmo siga siendo de complejidad lineal.

Idea básica de la DRT convencional

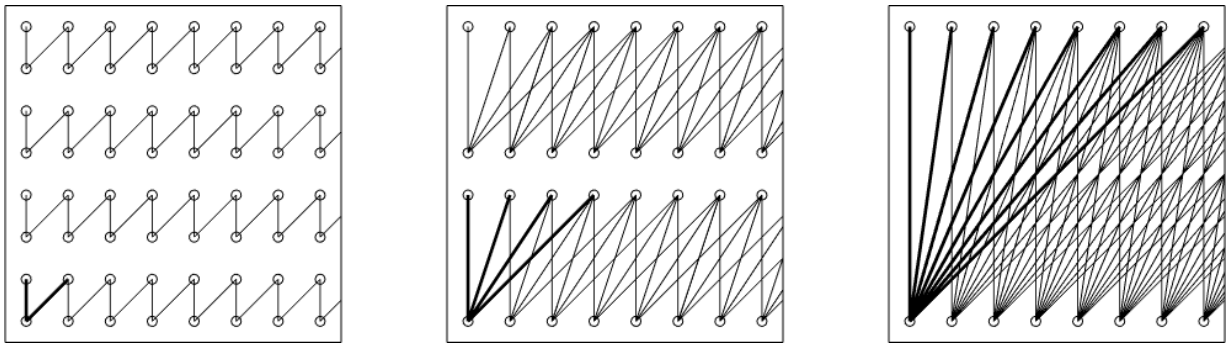


Figura 4.1: Representación de las líneas consideradas en varias escalas de la DRT convencional para el cuadrante ($x = y \cdot pendiente + d$).

Es posible expresar cualquier línea discreta como la unión de dos segmentos de línea de la mitad de su tamaño. Este es el principio básico de funcionamiento del algoritmo DRT multiescala. También se puede expresar como: habiendo calculado cada segmento de línea posible a una escala determinada, es posible calcular cada segmento de línea de la siguiente escala más gruesa simplemente reutilizando sumas parciales de la escala más fina. Mediante la recursividad es posible empezar a calcular cada segmento de línea de dos píxeles de longitud de una imagen, luego unirlos para obtener cada segmento de línea

de cuatro píxeles de longitud, y así sucesivamente. Salvo que... no se trata exactamente de cada segmento de línea.

Consideremos una imagen de tamaño $N \times N$, con $N = 2^n$, como la de la figura 4.1 para $N = 8$. Los segmentos de línea que se calculan a cualquier escala $m \in [0,\, n]$ son los que comienzan en los puntos $(k \cdot 2^m,\, d)\ \forall d \in [-N+1,\ ,N-1],\ k \in [0,\, 2^{n-m}]$ y a partir de ahí los puntos finales se situarán en $((k+1) \cdot 2^m,\, d+s),\ \forall s \in [0, 2^{m-1}]$.

El algoritmo de la DRT surge a partir de las definiciones de linea discreta, etapa de transformación parcial y ecuación de mapeo entre etapas que se pueden encontrar en Marichal-Hernandez *et. al.*(15). Especialmente a partir de la ecuación de mapeo entre etapas, cuyos detalles se pueden consultar en la bibliografía:

$$\tilde{f}^{\,m+1}(\overbrace{\underbrace{s_{n-m-1}}_{n\ bits},\overbrace{s_{n-m},\ldots,s_{n-1}}^{\boldsymbol{\sigma}:\,m\ bits}}^{\boldsymbol{s}:\,m+1\ bits}|\overbrace{v_{m+1},\ldots,v_{n-1}}^{\mathbf{v}:\,n-m-1\ bits}|\,d) =$$

$$\tilde{f}^{\,m}(\boldsymbol{\sigma}|\,0,\mathbf{v}|\,d) + \tilde{f}^{\,m}(\boldsymbol{\sigma}|\,1,\mathbf{v}|\,d + s_{n-m-1} + \lambda(\boldsymbol{\sigma})) \quad (4.1)$$

El algoritmo de un cuadrante de la DRT convencional se puede obtener rodeando esta fórmula con cuatro bucles anidados consistentes en la etapa m, el desplazamiento d y la descomposición de la dimensión horizontal en términos de franja v y pendientes σ dentro de la franja. En la DRT convencional no hay necesidad de definir una estructura de datos alternativa que contenga las etapas parciales de transformación, puesto que se pueden computar *in-place*, siendo la imagen de entrada, los datos transformados hasta la etapa 0, y el resultado la etapa n de transformación parcial.

Idea básica de la DRT paralela

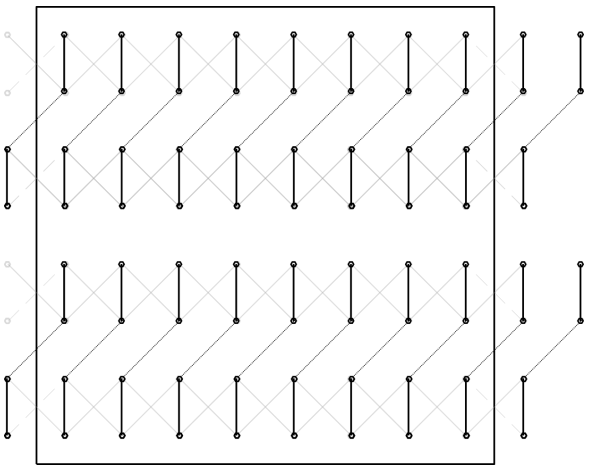


Figura 4.2: Independientemente del punto de partida, se utilizará el mismo esquema de cálculo para una pendiente determinada –ejemplificado para tamaño $N = 8$, etapa $m = 2$, pendiente $s = 1$–.

Para explotar el paralelismo de la DRT multiescala convencional nos centramos en los siguientes aspectos:

- ¿cuál es el bloque 'atómico' de cálculo a una escala determinada?
- ¿cuántas veces se repite?
- ¿se repite exactamente de la misma manera o presenta variaciones?

Las respuestas a estas preguntas definirán nuestra propuesta de algoritmo paralelo.

En la figura 4.2 se puede observar el patrón repetitivo de uno de esos cálculos atómicos de DRT para un determinado tamaño, escala y pendiente: tamaño $N = 8$, etapa $m = 2$ y pendiente $s = 1$. Para calcular una suma de longitud 4 píxeles, por ejemplo partiendo del píxel inferior izquierdo en coordenadas cartesianas (y, x) de valor $(0, 0)$ que llega al píxel tres posiciones verticales más arriba y una posición a la derecha, esto es $(3, 1)$, los datos a sumar son dos resultados de la etapa anterior que contienen los segmentos de dos píxeles de longitud que unen $(0, 0)$ con $(1, 0)$ y $(2, 1)$ con $(3, 1)$. Obsérvese que el píxel de llegada está a distancia $+(2^m - 1,\ slope)$ del inicial. Pero si consideramos cualquier otro segmento de longitud 4 que empiece en una posición arbitraria $(k \cdot 2, d)$ entonces los dos datos a sumar son los que van de $(k \cdot 2, d)$ a $(k \cdot 2 + 1, d)$ y $((k + 1) \cdot 2, d + 1)$ a $((k+1) \cdot 2+1, d+1)$. Este cálculo se repite exactamente igual en todo el dominio: partiendo de cualquier $y-$coordenada múltiplo de dos, y cualquier *d*esplazamiento a lo largo de la $x-$coordenada, dentro de los límites de la imagen.

La cuestión acerca de si existen variaciones a este patrón general queda pendiente puesto que requiere una respuesta más elaborada acerca de cómo cambian los tamaños, y por tanto los límites en lectura, de los datos al avanzar en las etapas de transformación.

4.1.2. Estructura de datos

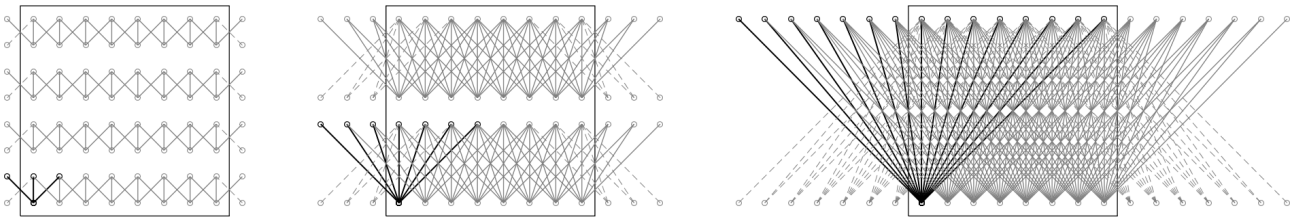


Figura 4.3: De izquierda a derecha, líneas de orientación vertical a una escala cada vez mayor

Nos interesa calcular la suma de píxeles que atraviesan la imagen de entrada empezando por cualquier píxel en una fila múltiplo de 2^m, y que llegan a $2^m - 1$ posiciones más arriba; y desde $2^m - 1$ posiciones a la izquierda hasta $2^m - 1$ posiciones a la derecha. Esto debe calcularse para las etapas que van de $m = 1$ a $m = n = log_2(N)$. Cuando $m = 0$ tenemos los datos de entrada (píxeles individuales: es decir, segmentos de línea de longitud 1) y cuando $m = n$ tenemos el resultado deseado (líneas que cruzan la entrada completa, atravesando $2^n = N$ píxeles). Y no sólo para los segmentos de línea orientados verticalmente, sino también para los orientados horizontalmente. El proceso se ilustra para $N = 8$ en la figura 4.3, sólo para las líneas orientadas verticalmente.

Nuestros datos en las etapas parciales serán una matriz tridimensional con dimensiones, comenzando por la más significativa a la izquierda: *(coordinada vertical, pendiente,*

desplazamiento horizontal). Los tamaños de esas dimensiones variarán con la etapa m. La dimensión de la coordenada vertical (implícitamente multiplicada por el factor 2^m) irá de 0 a $2^{n-m}-1$; la pendiente cubrirá de 0 a $2^{m+1}-2$, como índice de memoria sin signo, pero indicará pendientes reales que van de -2^m+1 a 2^m-1; y el desplazamiento horizontal irá de 0 a $N+2^{m+1}-3$, según se explicará más adelante.

Los autores originales propusieron duplicar la dimensión del desplazamiento rellenando la entrada con ceros por debajo de los datos reales, de modo que incluso una línea que comience desde N-1 posiciones por debajo de la coordenada de desplazamiento 0 seguirá siendo considerada porque ese desplazamiento *negativo* tocará la imagen para la pendiente máxima, N-1. Así que una gran cantidad de memoria se asignó y se introdujeron multitud de cálculos (ni más ni menos que duplicar una de las dimensiones del problema) para hacer frente a esos casos marginales.

Con la estructura propuesta para las etapas parciales podemos considerar exactamente los puntos de partida que alcanzan los datos originales para una escala dada, ni más ni menos. En la etapa m, la máxima pendiente absoluta será de 2^m-1, y por lo tanto ese es el número de desplazamientos iniciales adicionales que se deben adosar a ambos extremos de las filas de cálculo, como se ilustra en la figura 4.3 mediante los círculos fuera del rectángulo de entrada de los que surgen líneas discontinuas. Obsérvese que hay dos píxeles adicionales adosados a las filas de la etapa parcial $m=1$, uno a la izquierda y otro a la derecha; tres a cada lado para $m=2$ y siete a cada lado para $m=3$. Este hecho hace que el rango de la dimensión de desplazamiento horizontal sea 'extraño': de 0 a $N+2^{m+1}-3$, como índice de memoria sin signo, para considerar N posiciones centrales así como 2^m-1 antes y después de las posiciones centrales.

Si queremos alojar sólo dos búferes e intercambiarlos después de cada etapa parcial, éstos deben asignarse inicialmente a los tamaños de las etapas n y $n-1$, ya que la memoria crece con la etapa. El tamaño total de la etapa parcial m, siendo $(2^{n-m})\times(2^{m+1}-1)\times(N+2^{m+1}-2)\times \mathrm{sizeof}(\mathrm{atomicData})$. Normalmente, si este algoritmo se va a utilizar sólo para la detección de líneas es una buena idea para utilizar enteros sin signo de 8 bits como datos atómicos y, si fuera necesario, realizar un desplazamiento de bits a la derecha después de cada suma, a partir de la etapa 7, para evitar el desbordamiento de datos.

4.1.3. Algoritmo paralelo, sin adaptar aún a DSP

Las propuestas de cambio al algoritmo convencional de la DRT, que permite exponer mejor el paralelismo, pero que aún requerirá cambios para ser implementado en DSPs, se muestra, en Python, en **listing 4.1**. Es el código que computa las líneas verticales (y las horizontales por rotación de la entrada). Obsérvese que, en aras de la simplicidad, la memoria se asigna de forma independiente para cada etapa parcial, en lugar de intercambiar los búferes.

Listado 4.1: Código Python que computa la DRT paralela (vertical)

```
1 def sizePIDRT(n, m):
2     N = 2**n
3     M = 2**m
```

```
4       nSquares = (N // M)
5       slopeRange = 2 * M - 1
6       return (nSquares, slopeRange)
7
8
9   def PIDRT(f):
10      N = f.shape[0]
11      n = int(math.log2(N))
12      fm = []
13      for m in range(n + 1):  #memory allocation
14          nSquares, slopeRange = sizePIDRT(n, m)
15          fm.append(np.zeros((nSquares, slopeRange, (nSquares + 2) * 2**m - 2)))
16      for i in range(N):
17          for j in range(N):
18              fm[0][i][j][0] = f[i, j]
19      for m in range(n): # the loop will compute fm{m+1] from fm[m]}
20          M = 2**m
21          Mp1 = 2**(m + 1)  # M in stage m plus 1
22          nSquaresMp1, slopeRangeMp1 = sizePIDRT(n, m + 1)
23          for ySquareMp1 in range(nSquaresMp1):
24              for _slope in range(slopeRangeMp1): # slope as unsigned index
25              slope = _slope - Mp1 + 1  #slopes positives and negatives
26                  ab_s = abs(slope)
27                  s_sign = 1
28                  if slope < 0:
29                      s_sign = -1          # sign of slope
30                  s2 = ab_s // 2           # floor(half the absolute slope)
31                  rs = ab_s - 2 * s2       # remainder of absolute slope
32                  slopeM = M - 1 + s2 * s_sign # slopes of previous segments
33                  incIndB = s_sign * (s2 + rs) # displacement among segments
34                  for writeIdx in range(N + 2 * Mp1 - 2):
35                      readIdx = writeIdx + M - Mp1
36                      A = 0
37                      B = 0
38              # data at write index will sum segments A and B
39              # A starting from read index at stage m
40              # B starting from read index displaced half slope + rest
41              # inside segments A and B: half the absolute slope
42              # outside segments: as a relative movement of B, the rest
43                      if (readIdx >= 0 and readIdx < N + 2 * M - 2):
44                          A = fm[m][ySquareMp1 * 2, slopeM, readIdx]
45                      if (readIdx + incIndB >= 0
46                              and readIdx + incIndB < N + 2 * M - 2):
47                          B = fm[m][ySquareMp1 * 2 + 1, slopeM, readIdx +
                                incIndB]
48                      fm[m + 1][ySquareMp1, _slope, writeIdx] = A + B
49      return fm[n]
```

En cada etapa parcial hay tres bucles interiores, en las líneas 23 (dimensión vertical), 24 (pendiente sin signo) y 34 (desplazamiento horizontal). Esos bucles son propensos a la paralelización, ya que no imponen barreras a la ejecución del código... la única barrera de sincronización es el bucle exterior de la etapa parcial en la línea 19.

Para hacer versiones paralelas de este código una alternativa es convertir las operaciones de lectura, escritura y suma de las líneas 44, 47 y 48 en operaciones vectoriales o SIMD, entonces se podrían estudiar los efectos de la caché intercambiando el número de

valores calculados simultáneamente a lo largo de cualquiera de esas tres dimensiones/bucles.

4.2. Implementación en DSP

Para exponer la vectorización de la transformada de radón se realizaron una serie de primeras implementaciones en Matlab de la transformada. En el **Apéndice** A.1 tenemos el punto de partida, y en **Apéndice** A.2 la última iteración sobre la que basaremos el código en C.

4.2.1. Decisiones de diseño

Para poder implementar la transformada en el DSP aprovechando los vectores hubo que realizar una serie de cambios adicionales a los ya descritos de mejora del algoritmo. Estos cambios implican una reestructuración de los datos de tal forma que podamos almacenar la información en memoria de forma contigua, para así operar con los vectores. Si bien Hexagon aporta una solución para poder buscar y almacenar vectores disperos con las instrucciones de *Scatter* y *Gather*, no es ideal ya que son accesos a memoria costosos y por ello se desalienta recurrir a ellos de forma frecuente. También implicó trabajar con bloques de información en lugar de con índices sueltos. Para operar con los vectores fue necesario un enmascaramiento de los datos previo a la operación para evitar contaminar la estructura de datos.

Para aprovechar los vectores al máximo se ha operado con elementos de un byte. Ya que vamos a operar con imágenes en escala de grises con valores de rango 0 a 255 el operar a nivel de byte implica que tendremos overflow tarde o temprano. Para solucionar este overflow vamos a desplazar el resultado de la operación en lugar de sólo sumar para evitar este overflow. Es por ello que vemos que operamos con `Q6_V_vavg_VbVb` en lugar de `Q6_V_vadd_VbVb`.

Dada la naturaleza del algoritmo que se decidió implementar no resultó posible usar los vectores con la memoria alineada, es por ello que debemos llamar a instrucciones en ensamblador para poder cargar de la memoria no alineada con vmemu (vmemUnaligned).

4.2.2. Estructura de los datos en el DSP

La entrada a nuestro algoritmo será una imagen en escala de grises de tamaño $N \times N$, siendo N un tamaño potencia de dos. Nuestro algoritmo va a hacer uso de dos estructuras de datos de $(3N-2) \times (2N-1)$ bytes útiles para almacenar la etapa actual y guardar la siguiente. Hemos de añadir a esto un padding tanto por la izquierda como por la derecha para acomodar las líneas que salen de nuestra imagen. Este padding va tener un tamaño de $(2N-1) \times (N-1)$. Estas estructuras van a ser llamadas `fm` y `fmp1`, donde `fm` (fm por

analogía a f^m, datos de f transformados hasta la etapa m contiene los datos sobre los que se va a operar y en `fmp1` (f^{m+1}) se irán guardando los datos resultantes de la operación actual. Estas estructuras se intercambiarán a lo largo del programa para actualizar los datos, intercambiando lo que antes era array de lectura por escritura y viceversa, por lo que podremos proceder sobrescribiendo esos mismos arrays, con lo que la huella total en memoria se reduce a dos instancias del array correspondiente a una etapa.

Para la primera etapa poblaremos `fm` con los datos de la imagen, tomando el cuenta el offset debido al padding que hemos introducido. `fmp1` se está inicializando a cero, no obstante no es fundamental, ya que cuando vayamos a escribir en él vamos a sobrescribir los datos que contiene, y el uso de máscaras durante las operaciones va a evitar que se introduzcan valores extraños.

Por último vamos a utilizar un buffer con un tamaño de 256 bytes para cargar los vectores de memoria.

4.2.3. Algoritmo

Nuestro algoritmo debe ser llamado dos veces. En la primera pasada computará las líneas verticales de la imagen. Para la segunda llamada hay que rotar la imagen $90°$.

Como podemos ver consistirá de 4 bucles anidados, representando la etapa con `m`, el bloque a trabajar con `ySquareMp1`, las pendientes que aplicar con `unsignedSlope` y el desplazamiento por `i`. La etapa vectorizada es el desplazamiento, y por ello se incrementa según `SIMDsize`. También podríamos paralelizar los bloques de `ySquareMp1` en el futuro ya que no hay interacción entre los bloques y por lo tanto no requieren de sincronización.

En el bucle más interno, que vamos a recorrer con `i`, tenemos que enmascarar los vectores que extraigamos de memoria, ya que si no hiciéramos esto resultaría en una acumulación de basura no deseada en nuestras operaciones.

Para calcular estas máscaras se usarán vectores de predicado, pero primero tendremos que calcular sus límites. La primera iteración de este cálculo se muestra en el **listado 4.2**. Este código hace uso de muchos condicionales y operaciones de memoria para manipular los índices y vectores de predicados, y se puede ver fácilmente que va a afectar al rendimiento de nuestro programa. Además no presenta muchas vías de mejora. Por ello se indagó más profundamente en la documentación de la unidad escalar (16) y se llegó a una iteración del código con instrucciones C intrínsecas que no emplea condicionales ni operaciones de memoria de C **listado 4.3**. Este último código expone parte del posible código ensamblador que tendríamos que desarrollar, y se puede aprovechar este conocimiento para moverlo a ensamblador y explotar el potencial del VLIW en el futuro. Es importante diferenciar las instrucciones `Q6_Q_vsetq_R(Word32 Rt)` y `Q6_Q_vsetq2_R(Word32 Rt)` ya que mientras la primera creará una máscara de 0's cuando $Rt \bmod SIMDsize = 0$ la segunda creará una máscara de 1's. Como sabemos que el vector siempre tendrá algún dato y sabemos que $nEndX > 0 \ \& \ nEndX \leq 128$, utilizaremos `vsetq2` ya que si no nuestra máscara derecha estará a 0 cuando queremos aceptar a todo el vector. Así mismo con la premisa de $nStartX \geq 0 \ \& \ nStartX \leq |indLecturaX|$ tendremos que utilizar `vsetq1` para que la máscara se pueda poner a 0. En el resto de

casos ambas funciones se comportarán igual.

Listado 4.2: Código del cálculo de máscaras con condicionales

```
if(indLecturaA < 0){
   startA = -indLecturaA%SIMDsize;
}
// Create mask for the left side of first vector
memset((char*) &condA, 0xFF, startA);
// Create mask for the right side of the first vector
if(indLecturaA+SIMDsize > rangeIndexM)
        memset(((char*) &condA) + startA+ rangeIndexM%SIMDsize, 0xFF, SIMDsize
            - rangeIndexM%SIMDsize);


// Create mask for the left side of the second vector
if(indLecturaB < 0){
    startB = -indLecturaB%SIMDsize;
}
memset((char*) &condB, 0xFF, startB);
if(indLecturaB+SIMDsize > rangeIndexM)
        memset(((char*) &condB) + startB + (rangeIndexM%SIMDsize), 0xFF,
           SIMDsize - rangeIndexM%SIMDsize);

if(i+SIMDsize > rangeIndexMp1)
    nEsc = rangeIndexMp1-i;
        memset((char*) &condEsc, 0xFF, nEsc);
```

Listado 4.3: Código del cálculo de máscaras final

```
// value for left side part of the mask for vA and vB
nStartA = Q6_R_max_RR(0,-indLecturaA);
nStartB = Q6_R_max_RR(0,-indLecturaB);

// value for right side part of the mask for vA and vB
nEndA = Q6_R_min_RR(128, rangeIndexM + nStartA );
nEndA = Q6_R_max_RR(nStartA, nEndA);
nEndB = Q6_R_min_RR(128, rangeIndexM + nStartB );
nEndB = Q6_R_max_RR(nStartB, nEndB);
nEndEsc = Q6_R_min_RR(128, rangeIndexMp1-i);

leftMaskA  = Q6_Q_vsetq_R(nStartA);
leftMaskB  = Q6_Q_vsetq_R(nStartB);
rightMaskA = Q6_Q_vsetq2_R(nEndA);
rightMaskB = Q6_Q_vsetq2_R(nEndB);

condA = Q6_Q_xor_QQ(leftMaskA, rightMaskA);
condB = Q6_Q_xor_QQ(leftMaskB, rightMaskB);
condEsc = Q6_Q_vsetq2_R(nEndEsc);
```

Una vez tengamos las máscaras podemos aplicársela a los vectores traídos de memoria y operar con ellos tal y como se ve en **listado 4.4**

Listado 4.4: Operación

```
    // Fetch vector from memory using asm instruction
```

```
        get2v(  (int *) hvx_fm, (int *) &hvx_tmp[0], (offset + despA),
                        (int *) hvx_fm, (int *) &hvx_tmp[1], (offset + despB))
                            ;

        // Apply masks to vectors
        vA = Q6_V_vand_QV(condA, hvx_tmp[0]);
        vB = Q6_V_vand_QV(condB, hvx_tmp[1]);
        vEscUnmasked = Q6_Vb_vavg_VbVb_rnd(vA, vB);

        // Apply mask to result vector
        vEsc = Q6_V_vand_QV(condEsc, vEscUnmasked);

        // Save vector in memory
        setv((int *)&vEsc,(int *)hvx_fmp1, (offset + despEsc));
```

Los resultados de cada operación van a ser guardados contiguamente en memoria agrupados por la pendiente. Queremos que los resultados se agrupen por la pendiente para que cuando toque operar con el segmento se carguen todos los segmentos de misma pendiente en el vector, que es esencial para el funcionamiento correcto del algoritmo si queremos hacer aprovechar las operaciones SIMD. Esto se toma en cuenta cuando calculamos los desplazamientos para el acceso a memoria en **listado 4.5**. Para calcular el desplazamiento consideramos el desplazamiento vertical dado por el bloque `ySquareMp1`, considerado en `yDesp`, el inicio de la lectura en la fila representado por el índice de lectura, y finalmente el desplazamiento resultante de almacenar agrupadas las pendientes en `slopeDesp`. En el caso de la escritura tenemos que considerar los tamaños de la siguiente etapa.

Listado 4.5: Cálculo de desplazamientos

```
    // Calculate data location in memory
        despA = (yDespA + indLecturaA + slopeDesp);
        despB = (yDespB + indLecturaB + slopeDesp);
        despEsc = (yDespEsc + unsignedSlope*rangeIndexMp1 + i);
```

Como no hemos considerado un algoritmo que guarde los datos para la siguiente etapa alineados en memoria estamos forzados a acceder a memoria no alineada. Esto implica que tengamos que llamar a código ensamblador, ya que la operación de carga y almacenamiento en memoria no alineada `vmemu` no está disponible desde C **listado 4.2.3**.

```
#define vecIn1          R0
#define vecOut1         R1
#define desp1           R2

#define vecIn2          R3
#define vecOut2         R4
#define desp2           R5

#define newAddr1        R12
#define newAddr2        R13

  .text
  .p2align   2
  .p2align   4,,15
  .globl get2v
  .type get2v, function
```

```
get2v:
  {
        newAddr1 = ADD(vecIn1, desp1)
        newAddr2 = ADD(vecIn2, desp2)
  }

  v1   = VMEMU(newAddr1)
  v2     = VMEMU(newAddr2)
  VMEMU(vecOut1) = v1
  VMEMU(vecOut2) = v2
  JUMPR R31

    .p2align    2
    .p2align    4,,15
    .globl      setv
    .type       setv, function

setv:
    {
     newAddr1 = ADD(vecOut1, desp1)
     v0 = VMEMU(vecIn1)
    }
    {VMEMU(newAddr1) = v0}
    JUMPR R31
```

Al final de cada etapa debemos hacer un intercambio entre `fm` y `fmp1` para tener los datos en la siguiente etapa **Figura 4.6**. Es importante remarcar que debido a estos intercambios, al finalizar el algoritmo el resultado se encontrará en `fmp1` si $\log_2 N$ es impar, y en `fm` si es par.

Listado 4.6: Intercambio de estructuras

```
        hvx_tmp = hvx_fm;
        hvx_fm = hvx_fmp1;
        hvx_fmp1 = hvx_tmp;
        hvx_tmp = (HVX_Vector *) tmp;
```

4.2.4. Código en aplicación de CPU

Por el lado de la aplicación no hubo que desarrollar mucho código, ya que queremos que el cómputo se haga por el lado del DSP, pese a ello, necesitamos un código mínimo para inicializar los datos y las estructuras. Asimismo debemos reservar los recursos pertinentes para el correcto funcionamiento del programa.

Listado 4.7: Reserva de memoria del montón ION

```
    char * fm = NULL, * fmp1 = NULL, * tmp = NULL;
    int offset = (height-1)*(2*height-1);
    int sizeTot = (3*height - 2)*(2*height - 1);
    int fmSizes = sizeTot + 2*offset;
```

```
    if( 0 == ( fm = rpcmem_alloc(RPCMEM_DEFAULT_HEAP, RPCMEM_DEFAULT_FLAGS, (
        fmSizes+32)*sizeof(char)))){
         nErr++;
         printf("ERROR: Couldn't allocate fm\n");
         goto bail;
    }


    if( 0 == ( fmp1 = rpcmem_alloc(RPCMEM_DEFAULT_HEAP, RPCMEM_DEFAULT_FLAGS,
        (fmSizes+32)*sizeof(char)))){
         nErr++;
         printf("ERROR: Couldn't allocate fmp1\n");
         goto bail;
    }

    if(0 == (tmp = rpcmem_alloc(RPCMEM_DEFAULT_HEAP, RPCMEM_DEFAULT_FLAGS,
        256*sizeof(char)))){
             nErr++;
             printf("ERROR: Couldn't allocate tmp\n");
             goto bail;
    }
```

Si bien podemos utilizar el DSP con los parámetros por defecto, podemos hacer uso de utilidades para configurarlo a nuestras necesidades. Es esto lo que se hace en la función `initQ6` en el **listado 4.8**.

Listado 4.8: Configuración de parámetros del DSP

```
int initQ6() {
         int n_err = 0;
         dspCV_Attribute attr[] = {
                 {DSP_TOTAL_MCPS, 1000},
                 {DSP_MCPS_PER_THREAD, 500},
                 {PEAK_BUS_BANDWIDTH_MBPS, 12000},
                 {BUS_USAGE_PERCENT, 100}
         };

         n_err = dspCV_initQ6_with_attributes(attr, sizeof(attr)/sizeof(attr
             [0]));
         if(n_err) printf("dspCV_initQ6 exited with an error\n");

         return n_err;
}
```

Para poder elegir el dominio de DSP se usa la librería remote.h, que nos otorga la capacidad de abrir sesiones en tiempo de ejecución. En este caso nos interesa asegurarnos que utilizamos el DSP de cómputo, aunque también podríamos utilizar el de audio, ya que posee también el coprocesador de SIMD **listado 4.9**.

Listado 4.9: Obtención de la sesión remota

```
//DSP side code
int PIDRT_open(const char*uri, remote_handle64* handle) {
   void *tptr = NULL;
   tptr = (void *)malloc(1);
```

```
    *handle = (remote_handle64)tptr;
    assert(*handle);
    return 0;
}

//App side code
    char * uri = PIDRT_URI "&_dom=cdsp";
    int nErr = PIDRT_open(uri, &handle1);
    if (nErr) {
        printf("ERROR: Failed to open handle for CDSP domain\n");
        goto bail;
    }
```

Invocar el código en el DSP sólo requiere llamar a la función definida en el fichero `.idl` con la definición de la interfaz. En el caso que nos concierne se vería según el **listado 4.10**. Este caso específico requiere de un parámetro más de los listados explícitamente en la definición, ya que estamos heredando `remote_handle` con el propósito de poder seleccionar en tiempo de ejecución el cDSP.

Listado 4.10: Fichero IDL e invocación de la función

```
// Code on PIDRT.idl file
#include "AEEStdDef.idl"
#include "remote.idl"

interface PIDRT : remote_handle64 {
        long forwardPanelHorSIMD(
                inrout sequence<char> fm,
                        inrout sequence<char> fmp1,
                        inrout sequence<char> tmp,
                        in long width,
                        in long height
        );

};

// DSP call
    nErr = PIDRT_forwardPanelHorSIMD(
                        handle1,
                                        fm, fmSizes + 32,
                                        fmp1, fmSizes + 32,
                                        tmp, 256,
                                        width, height
    );
```

4.2.5. Resultados

Tras implementar el algoritmo y comprobar que funciona correctamente podemos ver los tiempos de ejecución de una pasada de la transformada en la **Tabla 4.1**, donde comparamos la ejecución entre el DSP y la CPU, podemos ver una mejoría interesante sobre el código ejecutado en el DSP, sentándose cómodamente con tiempos de ejecución 3 veces más rápidos que en el algoritmo local no optimizado. Esto teniendo en cuenta que la

frecuencia de reloj de los núcleos de la CPU duplica a la del DSP nos da testamento de la mejora por la ejecución de código en SIMD. Aunque el código en local podría ser mejorado inmensamente, recordemos que queremos demostrar la valía del DSP como alternativa ya que de esta forma podemos ahorrar energía del dispositivo, y también podemos dejar libre a la CPU y GPU para poder hacer otros cálculos que tal vez se aprovechen en mayor medida de sus características concretas.

Programa	$\mathbf{64 \times 64}$	$\mathbf{128 \times 128}$	$\mathbf{256 \times 256}$	$\mathbf{512 \times 512}$	$\mathbf{1024 \times 1024}$
HVX	792µs	1933µs	6874µs	28563µs	107105µs
Local	3833µs	7749µs	21281µs	85228µs	358064µs

Tabla 4.1: Comparación HVX vs Local no optimizado

Cabe destacar al observar los tiempos que todavía nos falta mejorar la vectorización, paralelización y codificación para alcanzar nuestro objetivo de ejecutar la transformada holgadamente bajo las restricciones del tiempo real. Aunque también como se ha querido dejar claro en el transcurso de esta memoria, y se comentará más extensivamente en las líneas futuras, queda claro que todavía hay potencial que aprovechar de los DSP. En el estado actual podríamos soportar sobre unos 17 ó 18 fps en una imagen de 512×512, cuando lo ideal sería estar por encima de 30fps. La importancia de conseguir estos tiempos es evidente, ya que proporcionaría una herramienta más para la Visión por computador.

Si ponemos estos números contra los obtenidos por la implementación con Halide (17) en (14) vemos que estamos cerca de lo esperado por Halide, aunque estos tiempos pueden tener un gran aumento en el futuro, cuando la planificación automática para Halide esté disponible en los DSP.

4.3. Carga en dispositivos de fabricante

Se buscó varias alternativas para hacer la carga de programas sin firmar a los dispositivos. En primer lugar se probó a ver si realmente el OEM bloqueaba las firmas de prueba, ya que, si bien no es la opción más cómoda, podría ser válida para poder desarrollar sin necesidad de un Kit de desarrollo, ya que el precio de estos es bastante elevado. Se confirmó lo descrito en la documentación de Qualcomm, viendo que es imposible cargar módulos con una firma de prueba en dispositivos de fabricante. Otra opción estudiada fue la del uso de los módulos sin firmar, ya que en el caso de nuestro algoritmo no tenemos problemas con las limitaciones de estos módulos.

Podemos preguntar al sistema si disponemos de esta capacidad. El código en **listado 4.11** nos muestra cómo.

Listado 4.11: Pregunta al sistema por módulos sin firma

```
#pragma weak remote_handle_control
```

```
if(remote_handle_control) {
        struct remote_dsp_capability dsp_cap = {CDSP_DOMAIN_ID,
            UNSIGNED_PD_SUPPORT,0};
        nErr = remote_handle_control(DSPRPC_GET_DSP_INFO, &dsp_cap, sizeof(
            struct remote_dsp_capability));
        printf("Result of capability query for UNSIGNED_PD_SUPPORT is %d", (
            int) dsp_cap.capability);
}
```

Para poder tener acceso a estos módulos tenemos que ejecutar el código del **listado 4.12** antes de hacer alguna llamada a fastRPC:

Listado 4.12: Código C para cargar módulos sin firma

```
#pragma weak remote_session_control;
// Call before any fastRPC call
if(remote_session_control){
    struct remote_rpc_control_unsigned_module data;
    data.enable = 1;
    data.domain = CDSP_DOMAIN_ID;
    remote_session_control(DSPRPC_CONTROL_UNSIGNED_MODULE, (void*) &data,
        sizeof(data));
}
```

Sin embargo, aunque esta característica parece haber existido en su momento para dispositivos Hexagon al menos para el SM8150, parece que la gama de SDM845, Snapdragon 821, SDM450, SDM675 no poseen esta funcionalidad véase en las **Figuras** 4.4, 4.5 y 4.6. Lo que nos lleva a concluir que han ido retirando esta capacidad de los nuevos procesadores. Se puede observar que para poder lanzar un módulo dinámico firmado

```
_PRELOAD=/vendor/lib/rfsa/adsp/libTestCapability.so ./TestCapability

---Starting TestCapability test
Result of capabity query for UNSIGNED_PD_SUPPORT is 0
108|sdm845:/vendor/bin #
```

Figura 4.4: Capacidad de carga de módulos sin firmar en SDM845

```
D_PRELOAD=./libTestCapability.so ./TestCapability

---Starting TestCapability test
Result of capabity query for UNSIGNED_PD_SUPPORT is 0
255|a70q:/data/local/tmp $
```

Figura 4.5: Capacidad de carga de módulos sin firmar en SDM675

se requiere crear el proceso `fastrpc_shell_unsigned_3`, el cual no está disponible si miramos el error en **Figura** 4.7. Si buscamos este proceso en el dispositivo tampoco lo encontraremos, como se puede ver también en la **Figura** 4.8.

Finalmente se tanteó la opción de usar un móvil con *AOSP (Android Open Source Project)* en un Google Pixel 3, ya que tal vez no tendría todas las restricciones que uno de fábrica. Se intentó firmar en el dispositivo con una firma de prueba. Este intento no dió resultado lamentablemente.

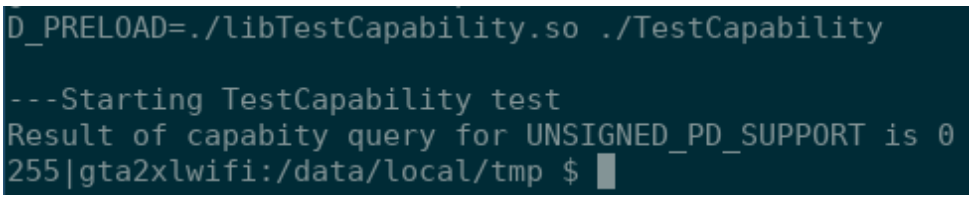


Figura 4.6: Capacidad de carga de módulos sin firmar en SDM450

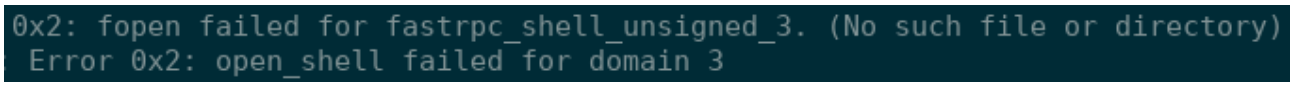


Figura 4.7: Error en creación de módulo sin firmar

Visto esto confirmamos que la idea de Qualcomm es que el SDK en conjunción con el HDK sirva para prototipar aplicaciones que presentar ante OEMs para obtener licencia para cargar el código (18).

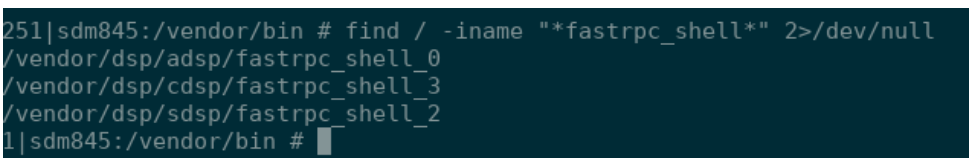


Figura 4.8: Procesos de comunicación con DSP

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

Este Trabajo de Fin de Grado ha supuesto una oportunidad única para aprender acerca de la transformada de Radon y para desarrollar sobre DSPs Hexagon. Una posibilidad que en condiciones normales se escaparía del alcance de un desarrollador por la barrera de entrada que supone el coste del material necesario, y lo específico del problema tratado.

El algoritmo desarrollado aprovechando las mejoras de HVX demuestra que el uso del DSP para aplicar la transformada de Radon a video no es una idea descabellada y que una futura iteración de este código podría llevar sus tiempos dentro del tiempo requerido, de 30 fps. Aunque traspasar esta transformada ha supuesto una reimaginación de la estructura de datos y del algoritmo queda demostrado en esta memoria el potencial del algoritmo a ser vectorizado.

En lo referente a la carga sin firma se ha comprobado que muchos dispositivos no poseen los requisitos necesarios para hacer uso de la carga de módulos sin firma. Por otro lado confirmamos la idea de que Qualcomm, a menos que cambie su modelo en el futuro, busca que para desplegar aplicaciones en DSP haya una estrecha coordinación entre fabricantes y desarrolladores. Esto relega a que el trabajo primordial de los ISV sea el de prototipar aplicaciones que mostrar al OEM. Esto tiene sus desventajas, ya que nada asegura que el desarrollo sea amortizado, pues el fruto del trabajo dependerá del contrato con el fabricante.

5.2. Líneas futuras

Como líneas futuras hay varias formas de conseguir mejoras en el proyecto:

- Estudiar el uso de la caché para mejorar las operaciones a memoria. Sabemos que Hexagon permite poblar previamente a caché con datos con los que se va a operar. Aprovechar la caché implica modificar la estructura de datos para conseguir que los datos se encuentren alineados cuando vayamos a acceder a ellos. Con el algoritmo

tal y como está esto supone un problema debido a que el tamaño de las estructuras de datos empleadas nos impiden reservar suficiente memoria contigua. Una posible solución a este problema podría ser compartimentar la ejecución de la transformada de tal forma que podamos usar una estructura menor. En caso de seguir esta vía se debería estudiar si la mejora que supone el uso de la caché compensa el incremento de invocaciones y la sobrecarga de operaciones que implicaría.

- Uso de ensamblador. Como se ha expuesto en el proyecto, los DSP de Hexagon poseen capacidad para VLIW y SIMD, puede haber una mejora significativa de las prestaciones si bajamos a ensamblador para codificar los bloques de operaciones. Ya podemos ver en el propio código intrinsic posibilidades para planificar las operaciones del bloque SIMD y se podrían exponer muchas más.
- Finalmente cabe la opción de explorar la paralelización en el DSP. Haciendo uso de los hilos de Hexagon conseguiríamos exprimir aún más rendimiento.

En lo referente a la carga sin firma no hay mucho que podamos hacer ya que dependemos de Qualcomm. En lo personal, probar algún móvil que sí tenga la capacidad de carga sin firma sería ideal para estudiar sus límites. Ya sea por medios permitidos, o logrando permisos de superusuario.

Capítulo 6

Summary and Conclusions

6.1. Sumary

This project has been a unique opportunity to learn about the Radon transform and to develop on Hexagon's DSP. This experience wouldn't be possible for me in any other circumstances due to the high entry barrier imposed by the price of the necessary material, and particularity of the studied problem.

The algorithm developed in this project while taking advantage of HVX enhancementes shows that applying the Radon transform to a video on a DSP is not a far-fetched idea, and future iterations on this code could take it to real time range. Although implementing this transform has meant a reimagining of the data structures and the algorithm, the vectorized algorithm potential has been shown in this report.

In regards of loading modules without signing, it has been found out that many devices do not meet the necessary requirements to make use of the Unsigned PD. On the other hand, we can confirm the idea that Qualcomm, unless it changes its views, intends developers and OEM to work in close coordination in order to deploy applications on comercial devices. This relegates ISV's to mainly prototype applications to show to OEM. This has its drawbacks, since nothing ensures you that the development can be redeemed because the decision to sign will be on the manufacturer's side.

6.2. Future lines

Some improvements that can be implemented into this project are:

- Cache usage could be studied to improve times. Hexagon allows us to pre-load the cache with data. Taking advantage of this will involve modifying the data structures to ensure that data is aligned when we try to access it. This is a problem to the algorithm as it is because the size of the data structures prevent us from allocating enough contiguous memory. A possible solution to this problem could arise from compartmentalizing the execution of the transform in such a way that we could use

smaller data structures. If we follow this path, it should be studied if cache usage compensates the increase of the number of invocations needed to complete the execution and the overhead of operations that it would imply.

- Switch portions of code to assembler. as it has been exposed in the project, Hexgon's DSP has VLIW and SIMD capabilities, and we can improve the performance significantly if we go down to assembler level and encode the operation blocks. We can already discern how some of these blocks will turn out from the intrinsic code in the SIMD block.
- One last option to improve the project could be exploring parallelization in the DSP. Making use of Hexagon threads could squeeze even more performance out of the algorithm.

There isn't much we can do when it comes to unsigned loads since it depends on Qualcomm. Trying a mobile with this capacity would be invigorating and would give the opportunity to study its capabilities and limitations. Either by allowed means, or by getting superuser access.

Capítulo 7

Presupuesto

7.1. Material

En este apartado podemos ver el costo de los materiales utilizados en este proyecto en la **Tabla** 7.2. Nótese que aunque en el proyecto final no se usa la cámara OpenQ, si se utilizó durante la toma de contacto con el SDK, y en caso de querer hacer un proyecto real es un elemento fundamental, es por eso que se añade al material.

Material	Precio (€)
Kit de desarrollo open-q-845-usom	835.15€
Cámara OpenQ 13MP	133,45€
Panel LCD OpenQ	125,90€
10 % Amortización ordenador de desarrollo	60€
Total en material	1154,5€

Tabla 7.1: Coste material

7.2. Personal

Para calcular el coste de personal se ha supuesto un sueldo aproximado de 30€/h.

Tarea	Horas	Total (€)
Instalación del SDK	1	30€
Preparación del HDK	1	30€
Primera toma de contacto con el SDK	25	750€
Estudio de documentación y bibliografía	30	900€
Pruebas de firma	6	180€
Desarrollo algoritmo	140	4200€
Preparación de la memoria	45	1350€
Total trabajado	208	7440€

Tabla 7.2: Coste personal

7.3. Coste total del proyecto

El precio total para reproducir este proyecto asciende a un total de 9110,17€ entre costes de material y de personal, a lo que se ha añadido un 6 % de beneficio.

Apéndice A

Códigos

A.1. PIDRT convencional en Matlab

```
%data = [[ 1,  2,  3,  4],
         %[ 5,  6,  7,  8],
         %[ 9, 10, 11, 12],
         %[13, 14, 15, 16]
        %];

data = [[ 0,  1],
        [ 2,  3]
       ];

    PIDRTforwardPANEL(data, max);

function [Rf, Rfparciales] = PIDRTforwardPANEL(f, maximumSlopes);
if nargin < 2
    maximumSlopes = true;
end;
Rf = [];
[Ny, Nx] = size(f);
if Nx ~= Ny
    disp('Tamanho f debe ser cuadrado');
    return;
end;

N = Nx;
nlog = log2(N);
n = floor(nlog);
if n ~= nlog
    disp('Tamanho f debe ser potencia de dos')
    return;
end;

Rfparciales = cell(1, n);

[sizeTot, ~, ~, ~, ~] = sizeRadonExteriorInvertiblenm(n, n);
fmp1 = zeros(1, sizeTot, 'uint32');
fm = zeros(1, sizeTot, 'uint32');
f=f';
fm(1:N*N) = uint16(f(:));
```

```
for m=0:n-1
    M = 2^m;
    Mp1 = 2^(m+1);
    [sizeTotM, sizeFilaM, sizeSquareM, sizeLateralM, nSquaresM] =
       sizeRadonExteriorInvertiblenm(n, m);
    [sizeTotMp1, sizeFilaMp1, sizeSquareMp1, sizeLateralMp1, nSquaresMp1] =
       sizeRadonExteriorInvertiblenm(n, m+1);
    if m==0
        for ySquareMp1 = 0:N/2-1
            %calculo del lateral izquierdo
            fmp1(indInOut2memIdx(ySquareMp1, -1, 0, N, 2, sizeFilaMp1,
               sizeLateralMp1) +1) = fm((ySquareMp1*2+1)*N +1);
            %calculo de la zona central
            for xSquareMp1=0:N/2-1
                if xSquareMp1 > 0
                    Cxm = fm((ySquareMp1*2+1)*N + (xSquareMp1-1)*2+1 +1);
                else
                    Cxm = 0;
                end;
                if xSquareMp1 < N/2-1
                    Dxp = fm((ySquareMp1*2+1)*N + (xSquareMp1+1)*2 +1);
                else
                    Dxp = 0;
                end;
                A = fm(ySquareMp1*2*N + xSquareMp1*2 +1);
                B = fm(ySquareMp1*2*N + xSquareMp1*2+1 +1);
                C = fm((ySquareMp1*2+1)*N + xSquareMp1*2+1 +1);
                D = fm((ySquareMp1*2+1)*N + xSquareMp1*2 +1);
                fmp1(indInOut2memIdx(ySquareMp1, xSquareMp1*2, xSquareMp1*2-1,
                    N, 2, sizeFilaMp1, sizeLateralMp1)+ [1:6]) =...
                    [A+Cxm, A+D, A+C, B+D, B+C, B+Dxp];
            end
            %calculo del lateral derecho
            fmp1(indInOut2memIdx(ySquareMp1, N, N-1, N, 2, sizeFilaMp1,
               sizeLateralMp1) +1) = fm((ySquareMp1*2+1)*N+N-1 +1);
        end
    else % etapas mayores que 0
        indEscritura=0;
    for ySquareMp1 = 0:sqrt(nSquaresMp1)-1
       for indIn = -Mp1+1:N-1+Mp1-1;
           for s = -Mp1+1:Mp1-1
               indOut = indIn+s;
               if (indIn < 0 && indOut < 0) || (indIn >= N && indOut >= N)
                   continue;
               end
               mods = abs(s);
               ss = sign(s);
               s2 = floor(mods/2);  % mitad de la pendiente
               rs = mods-2*s2;      % resto

               Ain = indIn; Aout = indIn+ss*s2;
               if (Ain < 0 && Aout < 0) || (Ain >= N && Aout >= N)
                 A = 0;
               else
                 A = fm(indInOut2memIdx(ySquareMp1*2, Ain, Aout, N, M,
                    sizeFilaM, sizeLateralM) +1);
```

```
                end
                Bin = Aout+rs*ss; Bout = indOut;
                if (Bin < 0 && Bout < 0) || (Bin >= N && Bout >= N)
                  B = 0;
                else
                  B = fm(indInOut2memIdx(ySquareMp1*2+1, Bin, Bout, N, M,
                      sizeFilaM, sizeLateralM) +1);
                end
                if m==n-1 && ~maximumSlopes && mods==Mp1-1
                    fmp1(indEscritura+1) = 0;
                else
                    fmp1(indEscritura+1) = A + B;
                end
                indEscritura = indEscritura+1;
            end
        end
    end
    fmp1(indEscritura+1:end)=0;
    end

    Rfparciales(m+1) = {fmp1};
    fm = fmp1;
end
Rf = fm;
end
```

A.2. PIDRT SIMD en Matlab

```
function Rf = PIDRTforwardPANELparallel1LoopSIMDmemContigua(f)
% function [Rf, Rfparciales] =
   PIDRTforwardPANELparallel1LoopSIMDsinCondicionales(f)

SIMDsizeMAX = 128;
Rf = [];
[Ny, Nx] = size(f);
if Nx ~= Ny
    disp('Tamanho f debe ser cuadrado');
    return;
end;

N = Nx;
nlog = log2(N);
n = floor(nlog);
if n ~= nlog
    disp('Tamanho f debe ser potencia de dos')
    return;
end;

Rfparciales = cell(1, n);

OFFSET = (N-1)*(2*N-1);

[sizeTot, ~, ~, ~, ~] = sizeRadonExteriorInvertiblenmParallelMemcontigua(n, n)
   ;
fmp1 = zeros(1, sizeTot+2*OFFSET, 'uint32');
fm = zeros(1, sizeTot+2*OFFSET, 'uint32');
```

```
disp(sizeTot + 2*OFFSET);
f=f';
fm(OFFSET+[1:N*N] ) = uint16(f(:));


for m=0:n-1
    M = 2^m;
    Mp1 = 2^(m+1);
    [~, sizeRowM, ~, rangeIndexM, ~] =
       sizeRadonExteriorInvertiblenmParallelMemcontigua(n, m);
    [~, sizeRowMp1, rangeSlopeMp1, rangeIndexMp1, rangeRowsMp1] =
       sizeRadonExteriorInvertiblenmParallelMemcontigua(n, m+1);
    for unsignedSlope = 0:rangeSlopeMp1-1;
        slope = unsignedSlope-Mp1+1;
        as = abs(slope);
        s2 = floor(as/2);
        rs = as-2*s2;
        ss = sign(slope);
        for initWriteIdx = 0:SIMDsizeMAX:rangeIndexMp1;  % bucle de indice, en
           SIMD
            SIMDstrides = [0:min(SIMDsizeMAX, rangeIndexMp1-initWriteIdx)-1];
                  % en las etapas avanzadas < SIMDmax
            readIdxA = (initWriteIdx + M - Mp1)+SIMDstrides;
            readIdxB = readIdxA + (s2 + rs) *ss;
            condA = uint32( (readIdxA >= 0) .* (readIdxA < rangeIndexM) );
            condB = uint32( (readIdxB >= 0) .* (readIdxB < rangeIndexM) );
            for ySquareMp1 = 0:rangeRowsMp1-1   %  b cle  de fila
                A = fm(OFFSET+ (ySquareMp1*2)*sizeRowM + readIdxA + (s2*ss+M
                   -1)*rangeIndexM +1) .* condA;
                B = fm(OFFSET+ (ySquareMp1*2+1)*sizeRowM + readIdxB + (s2*ss+M
                   -1)*rangeIndexM +1).* condB;
                fmp1(OFFSET+ ySquareMp1*sizeRowMp1 + unsignedSlope*
                   rangeIndexMp1 + initWriteIdx + SIMDstrides +1) = A+B;
            end
        end
    end
    Rfparciales(m+1) = {fmp1};   % para hacer trazas
    fm = fmp1;
end
Rf = fm(OFFSET+1:end-OFFSET);
end
```

A.3. PIDRT_test.c

```
/*
    ================================================================================

  Universidad De La Laguna.ULL.
  Perimetral Discrete Radon Transform.
================================================================================
    */

#include "PIDRT.h"
#include "PIDRT_test.h"
#include "dspCV.h"
#include "rpcmem.h"
```

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#if defined(__hexagon__)
#include "hexagon_sim_timer.h"
#include "hexagon_cache.h"
#endif

#include "AEEStdErr.h"

#include <math.h>
#include <sys/time.h>
#include <sys/types.h>

unsigned long long GetTime( void ){
        struct timeval tv;
//      struct timezone tz;
        gettimeofday(&tv, NULL);
        return tv.tv_sec * 1000000ULL + tv.tv_usec;
}

int lsign(int val){ return ( val == 0)? 0: (val < 0) ? -1 : 1;}

int initQ6() {
        int n_err = 0;
        dspCV_Attribute attr[] = {
                {DSP_TOTAL_MCPS, 1000},
                {DSP_MCPS_PER_THREAD, 500},
                {PEAK_BUS_BANDWIDTH_MBPS, 12000},
                {BUS_USAGE_PERCENT, 100}
        };

        n_err = dspCV_initQ6_with_attributes(attr, sizeof(attr)/sizeof(attr
            [0]));
        if(n_err) printf("dspCV_initQ6 exited with an error\n");

        return n_err;
}

int deinitQ6() {
        int n_err = 0;
        dspCV_Attribute attr[] = {
                {DSP_TOTAL_MCPS, 1000},
                {DSP_MCPS_PER_THREAD, 500},
                {PEAK_BUS_BANDWIDTH_MBPS, 12000},
                {BUS_USAGE_PERCENT, 100}
        };
        int size = sizeof(attr)/sizeof(attr[0]);
        printf("Size is: %d\n", size);
        n_err = dspCV_deinitQ6();
        if(n_err)
                printf("Error deinitializing Q6\n");
        return n_err;
}
```

```
int localSign(int val){ return ( val == 0)? 0: (val < 0) ? -1 : 1;}

void lsizeSIMD(int nN, int mM, int * sizeRow, int *  rangeSlope,int *
    rangeIndex, int * rangeRows){

    *rangeRows = nN/mM;
    *rangeIndex = nN + 2*mM - 2;
    *rangeSlope = 2*mM -1;
    *sizeRow = (*rangeIndex)*(*rangeSlope);
}

int localDRT(   char * fm, int fmLen,
                                char * fmp1, int fmp1Len,
                                int width, int height
                            ){


        int dump = 0;
        int nErr = 0;
        int offset = (height-1)*(2*height-1);
        int nIn = log2(width);
        int i = 0;
        int m = 0;
        int unsignedSlope = 0, slope = 0, as = 0, s2 = 0, rs = 0, ss = 0;
        int ySquareMp1 = 0;

        int slopeStride = 0;
        int yStrideA = 0, yStrideB = 0, yStrideEsc = 0;
        int xStrideEsc = 0;

        int a = 0, b = 0;


        int nN = width;
        int mM = 0, mMp1 = 0;

        int sizeRowM    = 0, rangeIndexM   = 0;
        int sizeRowMp1  = 0, rangeSlopeMp1 = 0, rangeIndexMp1 = 0,
            rangeRowsMp1 = 0;
        int indLecturaA = 0, indLecturaB   = 0;

        int despA = 0, despB = 0, despEsc = 0;
        char* tmp = NULL;

        if(width != height || (float) nIn != log2(width)){
                nErr++;
        }

        // This goes from 0 to log2(width)
        for(m = 0; m < nIn; ++m){
                mM =   pow(2,m);
                mMp1 = 2*mM;
                lsizeSIMD(nN, mM, &sizeRowM, &dump,  &rangeIndexM, &dump);
                lsizeSIMD(nN, mMp1, &sizeRowMp1, &rangeSlopeMp1,  &
                    rangeIndexMp1, &rangeRowsMp1);
                for(ySquareMp1 = 0; ySquareMp1 < rangeRowsMp1; ySquareMp1++){
                        yStrideA = 2*ySquareMp1*sizeRowM;
```

```
                        yStrideB = (2*ySquareMp1+1)*sizeRowM;
                        yStrideEsc = ySquareMp1*sizeRowMp1;
                        for( unsignedSlope = 0; unsignedSlope < rangeSlopeMp1;
                             unsignedSlope++){

                                // Calculate slopes
                                slope = unsignedSlope - mMp1 + 1;
                                as = abs(slope);
                                s2 = as/2;
                                rs = as - 2*s2;
                                ss = lsign(slope);

                                slopeStride = (s2*ss+mM-1)*rangeIndexM;
                                xStrideEsc = unsignedSlope * rangeIndexMp1;

                                for(i = 0; i < rangeIndexMp1; i++){


                                        indLecturaA = i + mM - mMp1;
                                        indLecturaB = indLecturaA + ((s2 + rs)
                                             *ss);

                                        // Calculate data location in memory
                                        despA = (yStrideA + indLecturaA +
                                            slopeStride);
                                        despB = (yStrideB + indLecturaB +
                                            slopeStride);
                                        despEsc = (yStrideEsc + unsignedSlope*
                                            rangeIndexMp1 + i);
                                        a = fm[offset + despA];
                                        b = fm[offset + despB];
                                        if(indLecturaA < 0 || indLecturaA >=
                                            rangeIndexM)
                                                a = 0;
                                        if(indLecturaB < 0 || indLecturaB >
                                            rangeIndexM)
                                                b = 0;

                                        fmp1[offset+despEsc] = a + b;

                                }
                        }
                }

                // Swap vectors
                tmp = fm;
                fm = fmp1;
                fmp1 = tmp;

        }

        return 0;
}

int PIDRT_test(int height, int width, int verbose)
{
        int nErr = 0;
```

```
if ( height != width) //La imagen no es cuadrada
  return -1;



int i  = 0;
char * fm = NULL, * fmp1 = NULL, * tmp = NULL;


int offset = (height-1)*(2*height-1);

int sizeTot = (3*height - 2)*(2*height - 1);
printf("SizeTot: %d\n", sizeTot);
unsigned long long start = 0, end = 0;

printf("- Initializing rpcmem\n");
if(initQ6()){
    nErr++;
            printf("Error initializing Q6/n");
    goto bail;
}

rpcmem_init();

int fmSizes = sizeTot + 2*offset;
printf("FMSizes: %d\n", fmSizes);
if( 0 == ( fm = rpcmem_alloc(RPCMEM_DEFAULT_HEAP, RPCMEM_DEFAULT_FLAGS, (
    fmSizes+32)*sizeof(char)))){
    nErr++;
    printf("ERROR: Couldn't allocate fm\n");
    goto bail;
}


if( 0 == ( fmp1 = rpcmem_alloc(RPCMEM_DEFAULT_HEAP, RPCMEM_DEFAULT_FLAGS,
    (fmSizes+32)*sizeof(char)))){
    nErr++;
    printf("ERROR: Couldn't allocate fmp1\n");
    goto bail;
}

if(0 == (tmp = rpcmem_alloc(RPCMEM_DEFAULT_HEAP, RPCMEM_DEFAULT_FLAGS,
    256*sizeof(char)))){
    nErr++;
    printf("ERROR: Couldn't allocate tmp\n");
    goto bail;
}


remote_handle64 handle1;
char *uri;

printf("- compute sum on CDSP domain\n");
uri = PIDRT_URI "&_dom=cdsp";
nErr = PIDRT_open(uri, &handle1);
if (nErr) {
    printf("ERROR: Failed to open handle for CDSP domain\n");
    goto bail;
```

```
    }
    printf("Handle obtained!\n");

    memset(fm, 0, fmSizes*sizeof(char));
    memset(fmp1, 0, fmSizes*sizeof(char));
    memset(tmp, 0, 256*sizeof(char));

    for(i = 0; i < height*width; i++)
        fm[i+offset] = i;

    printf("DSP computing starts here!\n");
    start = GetTime();
    for(i = 0; i < 30; i++)
        nErr = PIDRT_forwardPanelHorSIMD(       handle1,
                                                fm, fmSizes + 32,
                                                fmp1, fmSizes + 32,
                                                tmp, 192,
                                                width, height
                                            );
    end = GetTime();
    printf("DSP computing ended here! Time: %lld. nErr: %d\n", end - start,
       nErr);
    printf("DSP computing avg time is: %lld\n", (end - start)/30);
    printf("- Errcode, Time Elapsed: %d, %llu\n", nErr, (end - start));

    if(verbose == 1){
        FILE * fp;
        fp = fopen("/vendor/bin/results/result.txt", "w+");
        for(i = offset; i < sizeTot + offset; i++)
                fprintf(fp," %d ", (int)fmp1[i]);
        printf("\n");
                fclose(fp);
    }

bail:
    if(fm)      rpcmem_free(fm);
    if(fmp1)    rpcmem_free(fmp1);
    if(tmp)     rpcmem_free(tmp);
    deinitQ6();
    rpcmem_deinit();
    return nErr;
}
```

A.4. PIDRT_imp.c

```
/*
    ============================================================================

  Copyright (c) 2012-2014 Qualcomm Technologies, Inc.
  All rights reserved. Qualcomm Proprietary and Confidential.
/*============================================================================

                    PIDRT_imp.c
/*============================================================================

                Discrete Radon Transform
```

```
                Universidad de La Laguna
============================================================================
    */


#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "HAP_farf.h"
#include "PIDRT.h"

#include <sys/time.h>


#include "PIDRT_asm.h"

#include <hexagon_protos.h>
#include <hexagon_types.h>


int PIDRT_open(const char*uri, remote_handle64* handle) {
   void *tptr = NULL;
  /* can be any value or ignored, rpc layer does not care
   * also ok
   * *handle = 0;
   * *handle = 0xdeadc0de;
   */
   tptr = (void *)malloc(1);
   *handle = (remote_handle64)tptr;
   assert(*handle);
   return 0;
}

/**
 * param handle, the value returned by open*
     retval, 0 for success, should always succeed
 */
int PIDRT_close(remote_handle64 handle) {
   if (handle)
      free((void*)handle);
   return 0;
}

int sign(int val){ return ( val == 0)? 0: (val < 0) ? -1 : 1;}

void sizeSIMD(int nN, int mM, int * sizeRow, int *  rangeSlope,int *
    rangeIndex, int * rangeRows){

    *rangeRows = nN/mM;
    *rangeIndex = nN + 2*mM - 2;
    *rangeSlope = 2*mM -1;
    *sizeRow = (*rangeIndex)*(*rangeSlope);
}

#define DSIZE 1
#define SIMDsize 128
//#define SIMDsize 64
```

```
int PIDRT_forwardPanelHorSIMD(
                        remote_handle64 h,
            char * fm, int fmLen,
                        char * fmp1, int fmp1Len,
                        char * tmp, int tmpLen,
                        int width, int height
){


        int dump = 0;
        int nErr = 0;
        int offset = (height-1)*(2*height-1);
        int nIn = log2(width);
        int i = 0;
        int m = 0;
        int unsignedSlope = 0, slope = 0, as = 0, s2 = 0, rs = 0, ss = 0;
        int ySquareMp1 = 0;

        int slopeDesp = 0;
        int yDespA = 0, yDespB = 0, yDespEsc = 0;

        int nN = width;
        int mM = 0, mMp1 = 0;

        int sizeRowM    = 0, rangeIndexM   = 0;
        int sizeRowMp1  = 0, rangeSlopeMp1 = 0, rangeIndexMp1 = 0,
           rangeRowsMp1 = 0;
        int indLecturaA = 0, indLecturaB   = 0;

        int despA = 0, despB = 0, despEsc = 0;

        int nStartA = 0, nStartB = 0;
        int nEndA = 0, nEndB = 0, nEndEsc = 0;

        HVX_VectorPred condA, condB,/* aux,*/ condEsc;
        HVX_VectorPred rightMaskA, rightMaskB;
        HVX_VectorPred leftMaskA, leftMaskB;
        HVX_Vector vA, vB, vEscUnmasked;
        HVX_Vector vEsc;

        HVX_Vector * hvx_fm     = (HVX_Vector *)        fm;
        HVX_Vector * hvx_fmp1   = (HVX_Vector *)        fmp1;
        HVX_Vector * hvx_tmp    = (HVX_Vector *)        tmp;




        if(width != height || (float) nIn != log2(width)){
                nErr++;
                goto bail;
        }

        // This goes from 0 to log2(width)
        for(m = 0; m < nIn; ++m){
                mM = pow(2,m);
                mMp1 = 2*mM;
                sizeSIMD(nN, mM, &sizeRowM, &dump,  &rangeIndexM, &dump);
```

```
sizeSIMD(nN, mMp1, &sizeRowMp1, &rangeSlopeMp1,  &
    rangeIndexMp1, &rangeRowsMp1);
for(ySquareMp1 = 0; ySquareMp1 < rangeRowsMp1; ySquareMp1++){
        yDespA = 2*ySquareMp1*sizeRowM;
        yDespB = (2*ySquareMp1+1)*sizeRowM;
        yDespEsc = ySquareMp1*sizeRowMp1;
        for( unsignedSlope = 0; unsignedSlope < rangeSlopeMp1;
            unsignedSlope++){

                // Calculate slopes
                slope = unsignedSlope - mMp1 + 1;
                as = Q6_R_abs_R(slope);
                s2 = Q6_R_asr_RI(as, 1);
                rs = Q6_R_sub_RR(as, Q6_R_asl_RI(s2,1));
                ss = sign(slope);

                slopeDesp = (s2*ss+mM-1)*rangeIndexM;


                for(i = 0; i <= rangeIndexMp1; i+=SIMDsize){


                        indLecturaA = i + mM - mMp1;
                        indLecturaB = indLecturaA + ((s2 + rs)
                            *ss);

                        nStartA = Q6_R_max_RR(0,-indLecturaA);
                            // No need to control size>SIMD.
                           it loops
                        nStartB = Q6_R_max_RR(0,-indLecturaB);

                        // Calculate data location in memory
                        despA = (yDespA + indLecturaA +
                           slopeDesp);
                        despB = (yDespB + indLecturaB +
                           slopeDesp);
                        despEsc = (yDespEsc + unsignedSlope*
                           rangeIndexMp1 + i);
                        nEndA = Q6_R_min_RR(128, rangeIndexM +
                            nStartA );
                        nEndB = Q6_R_min_RR(128, rangeIndexM +
                            nStartB );
                        leftMaskA  = Q6_Q_vsetq_R(nStartA);
                        leftMaskB  = Q6_Q_vsetq_R(nStartB);
                        nEndEsc = Q6_R_min_RR(128,
                           rangeIndexMp1-i);
                        nEndA = Q6_R_max_RR(nStartA, nEndA);
                        nEndB = Q6_R_max_RR(nStartB, nEndB);

                        // Fetch vector from memory using asm
                           instruction
                        get2v(  (int *) hvx_fm, (int *) &
                           hvx_tmp[0], (offset + despA),
                                        (int *) hvx_fm, (int
                                           *) &hvx_tmp[1], (
                                           offset + despB));

                        rightMaskA = Q6_Q_vsetq2_R(nEndA);
```

```
                                        rightMaskB = Q6_Q_vsetq2_R(nEndB);
                                        condA = Q6_Q_xor_QQ(leftMaskA,
                                            rightMaskA);
                                        condB = Q6_Q_xor_QQ(leftMaskB,
                                            rightMaskB);
                                        condEsc = Q6_Q_vsetq2_R(nEndEsc);

                                        vA = Q6_V_vand_QV(condA, hvx_tmp[0]);
                                        vB = Q6_V_vand_QV(condB, hvx_tmp[1]);
                                        vEscUnmasked = Q6_Vb_vavg_VbVb_rnd(vA,
                                             vB);

                                        // Apply mask to result vector
                                        vEsc = Q6_V_vand_QV(condEsc,
                                            vEscUnmasked);

                                        // Save vector in memory
                                        setv((int *)&vEsc,(int *)hvx_fmp1, (
                                            offset + despEsc));

                                    }
                                }
                            }

                            // Swap vectors
                            hvx_tmp = hvx_fm;
                            hvx_fm = hvx_fmp1;
                            hvx_fmp1 = hvx_tmp;
                            hvx_tmp = (HVX_Vector *) tmp;

                    }

bail:

                    return nErr;
}
```

Bibliografía

[1] W. Götz and H. Druckmüller, "A fast digital Radon transform—An efficient means for evaluating the Hough transform," *Pattern Recognition*, vol. 29, no. 4, pp. 711–718, 1996.

[2] "Texas instruments defiende el uso de sus dsp en el procesado de imágenes médicas." https://www.ti.com/lit/wp/slyy019/slyy019.pdf?ts=1615535019772.

[3] "Xiaomi pocophone f1." https://www.gsmarena.com/xiaomi_pocophone_f1-9293.php. Accessed: 2021-03-12.

[4] "Google pixel." https://store.google.com/magazine/compare_pixel?toggler0=Pixel+3. Accessed: 2021-03-12.

[5] "Sdk fastcv hexagon." https://developer.qualcomm.com/software/fast-cv-sdk/sample-app.

[6] "Tensorflow lite hexagon delegate." https://www.tensorflow.org/lite/performance/hexagon_delegate.

[7] "Charla sobre el uso del sdk para procesamiento de imágenes." https://www.youtube.com/watch?v=FTyhnKdQdgg&t=1661s.

[8] "Página de descarga del sdk de hexagon." https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools.

[9] "Documentación hexagon del sdk." file:///{HEXAGON_SDK_ROOT}/docs/index.html.

[10] "Guía del proyecto calculadora en el sdk." file:///{HEXAGON_SDK_ROOT}/docs/calculator_android.html.

[11] "Página producto snapdragon 845." https://www.qualcomm.com/products/snapdragon-845-mobile-platform.

[12] M. L. Brady, "A fast discrete approximation algorithm for the Radon transform," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 107–119, 1998.

[13] A. Brandt and J. Dym, "Fast calculation of multiple line integrals," *SIAM Journal on Scientific Computing*, vol. 20, no. 4, pp. 1417–1429, 1999.

[14] O. Gómez-Cárdenes, R. Oliva-García, G. A. Rodríguez-Abreu, and J. G. Marichal-Hernández, "Exposing parallelism of discrete radon transform," in *Proceedings of the 3rd International Conference on Telecommunications and Communication*

Engineering, ICTCE '19, (New York, NY, USA), p. 136–140, Association for Computing Machinery, 2019.

[15] J. G. Marichal-Hernández, J. P. Lüke, F. Rosa, and J. M. Rodríguez-Ramos, "Fast approximate 4-D/3-D discrete Radon transform for lightfield refocusing," *Journal of Electronic Imaging*, vol. 21, no. 2, pp. 023026–1, 2012.

[16] "Manual de referencia para el programador de hexagon v65." https://developer.qualcomm.com/qfile/67413/80-n2040-39_a_hexagon_v65_programmers_reference_manual.pdf.

[17] "Halide language." https://halide-lang.org/.

[18] "Charla hexagon dsp en qualcomm developer network." https://www.youtube.com/watch?v=9rpp2DfokD0&t=2330s.