Trabajo de Fin de Grado

Aceleración de la NFFT en hardware mediante el uso de FPGA's

NFFT acceleration in hardware using FPGA's

Cristhian Javier García Conrado

La Laguna, 5 de septiembre de 2016

D. **Eduardo Magdaleno Castello**, con N.I.F. 43.824.397-J profesor Titular de Universidad adscrito al Departamento de Ingeniería Industrial de la Universidad de La Laguna, como tutor.

D. **Fernando Andrés Perez Nava**, con N.I.F. 42.091.420-V profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A (N)

Que la presente memoria titulada:

"Aceleración de la NFFT en hardware mediante el uso de FPGA's."

ha sido realizada bajo su dirección por D. **Cristhian Javier García Conrado**, con N.I.F. 45.939.280-T.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 5 de septiembre de 2016

Agradecimientos

A mi madre. Que a pesar de todas las dificultades que hemos tenido, ha logrado sacar ella sola a toda una familia adelante. Ella es y siempre será, la principal responsable de que yo haya llegado hasta aquí. Eres todo un orgullo para mi y un verdadero ejemplo de lucha y superación. Gracias por todo mamá!

A mi abuela. Que me ha criado y que tanto se ha sacrificado para que cada fin de semana yo pudiera ir a estudiar. Me has enseñado que a pesar de todo, siempre hay motivos por los que seguir luchando. Gracias Lela!

A mi hermano. Mi gran apoyo desde pequeños. Gracias por estar ahí siempre que lo he necesitado. Eres el mejor regalo que me han dado y un motivo más por el que esforzarme cada día. Gracias Mauri!

Gracias a mi tía, mi tío y mis primos. Son un gran pilar en mi vida y también es gracias a ellos por los que he podido alcanzar todas mis metas.

Gracias a mi amigo y compañero de carrera. Gracias a el, esta experiencia universitaria ha sido única y sin el me hubiera sido imposible haber llegado hasta aquí. Espero que sigamos cosechando éxitos juntos!

A ella. Que a pesar de llevar tan poco tiempo en mi vida, ya es parte fundamental de mi día a día. Gracias por mostrarme el lado positivo de las cosas cuando a veces yo no lo veo y por superar conmigo esos pequeños problemas diarios que a veces parecen tan grandes.

Y por último, y no menos importante, gracias a mis tutores de proyecto. Por estar ahí cuando los necesitaba y por ayudarme con cualquier duda que me pudiera surgir. Sin ellos, llevar a cabo este proyecto no hubiera sido posible. Muchas gracias!

Licencia

Resumen

El objetivo de este trabajo ha sido aprovechar todas las ventajas que ofrece el diseño hardware mediante el uso de dispositivos programables como las FPGAs y su integración conjunta a un ARM, en lo que se ha denominado como APSoC (All Programmable System-on-Chip), y aplicar dicha tecnología a un algoritmo de actualidad y cuyas exigencias de recursos fueran críticas.

En la actualidad, casi cualquier campo de investigación de áreas tan diversas de la ciencia como pueden ser la medicina, geología, astronomía o biología llevan a cabo procesos cuyos requerimientos son críticos y en ocasiones la complejidad de los algoritmos que se encargan de dichos procesos suele ralentizar bastante el cómputo de los mismos.

Un ejemplo de ello es la NFFT (Transformada rápida de Fourier no uniforme); un algoritmo frecuentemente utilizado en el análisis y procesamiento de señales, cuando éstas no siguen un patrón regular o una trayectoria uniforme. Así pues, decidimos aplicar técnicas de paralelismo y de aceleración hardware para optimizar una versión software de este algoritmo, solventando así la complejidad y el alto coste computacional del mismo. Esta optimización ha sido posible, gracias a las últimas innovaciones tecnológicas y de diseño provistas por los desarrolladores de FPGAs.

Palabras clave: FPGA, NFFT, Aceleración, Hardware, Software, FFT, Paralelismo, SDSoC, Convolución, Deconvolución, Zynq, ARM.

Abstract

The aim of this work has been to take full advantages of the hardware design using programmable devices such as FPGAs and their integration to an ARM, in what has been termed as APSoC (All Programmable System-on-Chip), and apply this technology into a current algorithm with a critical requirements of resources.

Today, almost any field of science research such as medicine, geology, astronomy and biology performs some processes whose requirements are critical and sometimes the complexity of the algorithms that handle these processes often slows computing them.

One example is the NFFT (Non-uniform Fast Fourier Transform); an algorithm that's frequently used in the analysis and signal processing, when those signals don't follow a regular pattern or a uniform path. So, we decided to apply techniques of parallelism and hardware acceleration to optimize a software version of this algorithm, thus solving the computational complexity and high cost thereof. This optimization has been possible thanks to the latest technological and design innovations, provided by the developers of FPGAs.

Keywords: FPGA, NFFT, Acceleration, Hardware, Software, FFT, Parallelism, SDSoC, Convolution, Deconvolution, Zynq, ARM.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1
Introducción

En este capítulo se incluye una introducción acerca del planteamiento del proyecto, así como de la estructura de este documento.

1.1 Objetivo y justificación del proyecto

El presente proyecto tiene como objeto principal aprovechar las ventajas que ofrece el diseño hardware mediante el uso de dispositivos programables como las FPGAs, junto con las últimas tendencias de fusionar en un mismo circuito integrado, lógica programable y una arquitectura ARM, y aplicar dicha tecnología puntera a un algoritmo de actualidad y cuyos requerimientos de recursos sean críticos.

En este caso hemos optado por acelerar un algoritmo comúnmente utilizado en casi todas las áreas de la ciencia moderna, como es la NFFT (*Non-uniform Fast Fourier Transform*). La NFFT tiene aplicaciones tan diversas como las imágenes de resonancia magnética, radares de apertura sintética o el análisis térmico y sísmico. La problemática principal del algoritmo es su alto coste computacional[1], sobretodo con tamaños de datos muy grandes; lo cual, junto a su complejidad hacen de la NFFT un objetivo perfecto para su implementación en hardware y poder así acelerar la ejecución del mismo además de poder optimizar cualquier posible cuello de botella que tenga el algoritmo.

El objetivo final que se ha planteado para ésta línea de trabajo es el de comparar una ejecución software de la NFFT con una ejecución hardware del mismo y finalmente mediante el uso de herramientas de diseño hardware junto con la utilización de técnicas de paralelismo, conseguir reducir el tiempo de CPU consumido al ejecutar el algoritmo.

Por último, destacar la motivación personal que supone realizar este proyecto ya que me ha dado la oportunidad de aprender a utilizar una herramienta tan versátil y potente como lo es el entorno de desarrollo SDSoC de Xilinx

además de aprender a utilizar a fondo la tecnología FPGA, cabe notar que el uso de esta tecnología es cada vez mayor en muchas áreas de investigación, sobre todo en aplicaciones que requieren un alto grado de paralelismo.

1.2 Fases del desarrollo

El desarrollo del proyecto se ha realizado en varias fases. En primer lugar hubo que afianzar los fundamentos matemáticos en los que se basa el algoritmo. Para ello fue necesario una primera codificación del mismo usando el software matemático Matlab, aprovechando las ventajas que ofrece esta herramienta para codificar de una forma más cercana a la notación matemática en la que se ha descrito el algoritmo en primera instancia. El siguiente paso fue llevar el código a lenguaje C, ya que este lenguaje nos permitía bajar un poco más en el nivel de abstracción necesario para empezar a plantearnos el código en hardware.

A continuación, comenzamos a trabajar con el entorno de desarrollo SDSoC de Xilinx. Esta herramienta es la que nos permite traducir nuestro código software en código hardware que pueda ser procesado por la FPGA. Como nunca antes habíamos trabajado con este entorno de desarrollo, primero hubo que familiarizarse con él, realizando prácticas de ejemplo y leyendo la documentación disponible para luego adaptar nuestro código y realizar todas las pruebas necesarias. Más adelante nos detendremos un poco más en explicar a fondo la herramienta así como también en detalles más específicos del algoritmo y su codificación. Por último, con la librería de *profiling* que ofrece el propio SDSoC llevamos a cabo una medición de ciclos de CPU consumidos tanto en la ejecución software como en la ejecución hardware para su posterior análisis.

1.3 Estructura de la memoria

Después del presente capítulo, el cual consta de una breve introducción al planteamiento general del proyecto, le seguirá una descripción detallada del marco teórico en el que situaremos el proyecto. En dicho apartado se explicará a fondo el algoritmo en cuestión, base matemática, jerarquía modular, requerimientos de recursos, así como también sus aplicaciones más relevantes en la sociedad actual.

A continuación se explicará detalladamente las herramientas y tecnologías utilizadas para el desarrollo del proyecto, así como también las técnicas usadas para el eficiente desarrollo hardware del código. Seguidamente describiremos el flujo de trabajo llevado a cabo, para posteriormente realizar un análisis de los resultados obtenidos. Finalmente, expondremos las

conclusiones extraídas de todo el desarrollo realizado junto con las posibles mejoras y desarrollos futuros.

Capítulo 2
Marco teórico

En este capítulo definiremos la base teórica que sustenta nuestro trabajo, además de la problemática presente al abordar el estudio.

2.1 Antecedentes

La FFT (*Fast Fourier Transform*) o Transformada rápida de Fourier es una de las herramientas más ampliamente utilizadas en el procesado digital de señales y el análisis numérico. Mientras que la DFT o Transformada discreta de Fourier, para una señal de tamaño N, requiere $O(N^2)$ operaciones; la FFT haciendo uso de la estrategia *"divide y vencerás"* consigue evaluar la misma señal con tan solo $O(N \log N)$ operaciones[2].

Tanto la DFT como la FFT, trabajan sobre señales muestreadas de forma periódica y regular. Sin embargo, ciertas aplicaciones pueden requerir, en su lugar, hacer uso de un conjunto de datos muestreados de forma no regular, ni en el tiempo, ni en el espacio. Para evaluar este tipo de muestras de una forma más precisa, es necesario utilizar una generalización de la transformada de Fourier como lo es la NFFT[3]. A continuación, vemos un ejemplo de las muestras no uniformes que recibe como entrada el algoritmo y su salida:

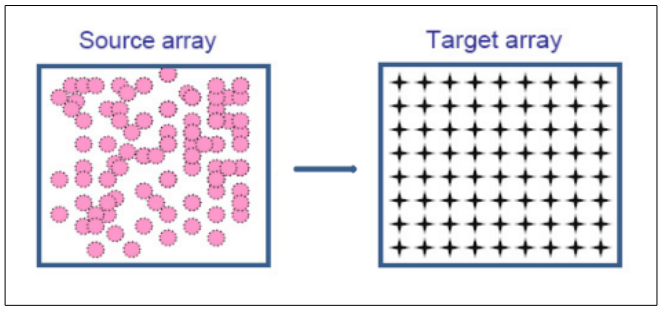

Figura 2.1 Muestras iniciales y finales de la NFFT[4].

Antes de profundizar en la NFFT es necesario entender primero el algoritmo de la NDFT (*Non-uniform Discrete Fourier Transform*), al que también llamaremos *"Algoritmo de Fuerza Bruta"*. Como veremos a continuación, este algoritmo es mucho menos eficiente que la NFFT, ya que requiere $O(NM)$ operaciones para $N \in 2\mathbb{N}$ frecuencias equidistantes y $M \in \mathbb{N}$ muestras no regulares. Para un conjunto $I_N = \{-\frac{N}{2}, \frac{N}{2} - 1\}$ de posibles frecuencias y un conjunto $\hat{f}_k \in \mathbb{C}, k \in I_N$ de coeficientes complejos de Fourier, evaluaremos el conjunto $x_j \in (\mathbb{R}^d / \mathbb{Z}^d) \sim [-\frac{1}{2}, \frac{1}{2}]^d, j = 0, \ldots, M-1$ de muestras irregulares para un número d de dimensiones. Por lo tanto, podemos definir formalmente la NDFT con la siguiente ecuación:

$$f(x) := \sum_{k \in I_N} \hat{f}_k \, \mathrm{e}^{-2\pi i k x}$$

Figura 2.2 Ecuación de la NDFT[5].

Como hemos mencionado con anterioridad, la NDFT no es tan eficiente como la NFFT, la cual propone un enfoque más apropiado para el cálculo rápido de la ecuación descrita anteriormente y cuyo coste computacional llega a ser de $O(M + N \log N)$. Dicho enfoque se basa en la utilización de la FFT junto con una función de Gauss φ como función ventana, localizada simultáneamente en el dominio tiempo/espacial y en el de frecuencia, la cual describiremos en la Figura 2.3. Básicamente, el esquema propuesto utiliza el teorema de la convolución; y a grandes rasgos podemos identificar los siguientes tres pasos:

1. Deconvolución del polinomio trigonométrico f con la función ventana en el dominio frecuencial.
2. Cálculo de la FFT sobre el resultado obtenido en el paso 1.
3. Convolución del resultado obtenido en el paso 2 con la función ventana en el dominio tiempo/espacial. Se evalúa la convolución en los puntos x_i .

Antes de definir formalmente la NFFT, debemos aclarar varios apartados. En primer lugar, y partiendo de la base matemática establecida para la NDFT anteriormente, es necesario establecer un factor de de sobremuestreo $\sigma > 1$, el cual viene a determinar el tamaño de la FFT que emplearemos en el algoritmo como $n = \sigma N$.

En segundo lugar definiremos el parámetro de truncamiento $m \in \mathbb{N}, m \ll n$, el cual nos permitirá truncar el error generado por la aproximación de la FFT. A continuación definiremos nuestra función ventana. Para este proyecto hemos usado la siguiente función Gaussiana como función ventana:

$$\varphi(x) = (\pi b)^{-1/2} e^{-\frac{(nx)^2}{b}} \quad \left(b := \frac{2\sigma}{2\sigma - 1}\frac{m}{\pi}\right) ,$$

$$\hat{\varphi}(k) = \frac{1}{n} e^{-b\left(\frac{\pi k}{n}\right)^2} ,$$

Figura 2.3 Función de Gauss como función ventana.

También tendremos en cuenta el conjunto $I_{n,m}(x) = \{l \in I_n : n \cdot x - m \leq l \leq n \cdot x + m\}$, ya que recoge los índices donde la función ventana se concentra principalmente.

Así pues, podemos definir formalmente la NFFT como:

Input: $d, M \in \mathbb{N}$, $\boldsymbol{N} \in 2\mathbb{N}^d$
$\boldsymbol{x}_j \in [-\frac{1}{2}, \frac{1}{2}]^d$, $j = 0, \ldots, M-1$, and $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$,

1: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute

$$\hat{g}_{\boldsymbol{k}} := \frac{\hat{f}_{\boldsymbol{k}}}{|I_{\boldsymbol{n}}|\, c_{\boldsymbol{k}}(\tilde{\varphi})}.$$

2: For $\boldsymbol{l} \in I_{\boldsymbol{n}}$ compute by d-variate FFT

$$g_{\boldsymbol{l}} := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{g}_{\boldsymbol{k}}\, e^{-2\pi i \boldsymbol{k}\left(\boldsymbol{n}^{-1} \odot \boldsymbol{l}\right)}.$$

3: For $j = 0, \ldots, M-1$ compute

$$f_j := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}}\, \tilde{\psi}\left(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l}\right).$$

Output: approximate values f_j, $j = 0, \ldots, M-1$.
Complexity: $\mathcal{O}(|\boldsymbol{N}| \log |\boldsymbol{N}| + M)$.

Figura 2.4 Algoritmo NFFT[6].

En la figura anterior podemos diferenciar los tres pasos en los que se divide el algoritmo de la NFFT. En nuestro proyecto cada uno de los pasos corresponderá con un módulo hardware, a saber, deconvolución, FFT y convolución.

De estos tres, el que supone el mayor reto es el módulo de la convolución, ya que es el paso que más recursos consume y por lo tanto el rendimiento de la NFFT vendrá determinado por dicho módulo.

Para finalizar este capítulo, queda resaltar la implicación directa que tendría acelerar la NFFT en mejorar el rendimiento de numerosas técnicas de procesado de señales y análisis numérico usadas en diversos ámbitos de la investigación científica y tecnológica. Por señalar algunos ejemplos, podemos encontrar técnicas de este tipo en campos como el procesamiento de imágenes 3D, diseño de filtros digitales, tomografía computarizada, biocomputación y algunas más que ya hemos citado anteriormente

Capítulo 3
Tecnologías

En este capítulo describiremos las tecnologías que fueron necesarias para el completo desarrollo del proyecto.

3.1 Lenguajes de programación

Una vez que hemos afianzado la base teórica de la NFFT pasamos a codificar el algoritmo. Para ello, hemos usado dos lenguajes de programación: el lenguaje matemático M y el lenguaje de programación C. A continuación describiremos sus características y su función dentro del proyecto.

3.1.1 Matlab y Octave

En primer lugar realizamos una primera codificación del algoritmo en lenguaje M, que es el lenguaje propio del que hace uso el software matemático Matlab. Debido a que Matlab es un software propietario tuvimos que buscar otras alternativas de software libre y en este caso optamos por la herramienta de GNU Octave[7].

Octave es el equivalente libre de Matlab, con el cual comparte muchas características tales como un intérprete del lenguaje para ejecutar órdenes de forma interactiva. Además, la sintaxis utilizada por Octave es casi idéntica a la que utiliza Matlab, por lo tanto nos permite también cargar archivos con funciones de Matlab[8].

Al ser un software matemático muy potente y con muy buena documentación en español, nos ha resultado de gran ayuda a la hora de entender en primera instancia la base matemática en la que se apoya el algoritmo. De esta forma obtuvimos una primera versión software de la NFFT además de lograr identificar las principales dificultades que nos podríamos encontrar al profundizar un poco más en el algoritmo.

3.1.2 C

La segunda codificación del algoritmo que hemos realizado fue hecha en lenguaje C. C es un lenguaje de programación ampliamente utilizado para el desarrollo de sistemas operativos así como para aplicaciones de escritorio. De la misma forma, es muy usado en aplicaciones científicas e industriales, además de experimentos informáticos, físicos, químicos, entre otros.

Sus principales ventajas son la facilidad para realizar programas modulares, sus librerías de código estándar, la portabilidad que ofrece, habiendo compiladores para casi todos los sistemas conocidos y la eficiencia propia del lenguaje puesto que es posible utilizar sus características de bajo nivel para realizar implementaciones óptimas[9]. Esta última fue la que nos promovió a realizar la segunda versión software de la NFFT en este lenguaje, ya que el nivel de abstracción que ofrece C nos facilitaría la tarea de plantear la estructura del diseño hardware del algoritmo.

3.2 FPGA

Una FPGA (Field Programmable Gate Array) es un dispositivo programable que contiene bloques de lógica, cuya interconexión y funcionalidad puede ser configurada para implementar funciones personalizadas en hardware, que puedan reproducir funciones tan sencillas como las que lleva a cabo una puerta lógica, hasta complejos sistemas en circuitos integrados. Para esto sólo es necesario desarrollar una tarea de cómputo digital en lenguaje HDL (Hardware Description Language) y compilarla en un archivo de configuración o *bitstream* que contenga la información de cómo deben conectarse los componentes de la FPGA.

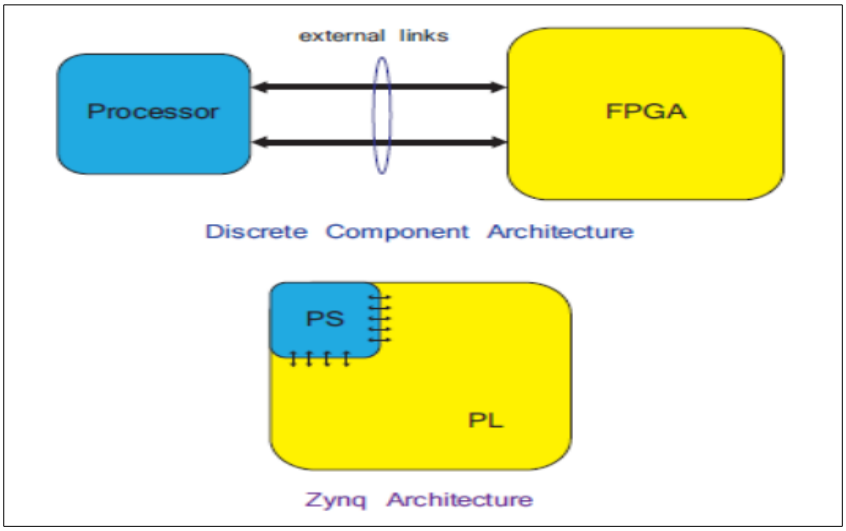

Figura 3.1 Arquitectura Zynq básica[10].

En la figura anterior, podemos apreciar la principal diferencia entre la arquitectura FPGA básica y la arquitectura Zynq que hemos usado para nuestro desarrollo. Como vemos, se ha integrado un ARM que trabaja conjuntamente con la lógica programable nativa de la FPGA. El uso de esta tecnología puntera hace que ahora nuestro flujo de diseño cambie, de forma que nos permite implementar una determinada función en hardware o en software, atendiendo a las necesidades del sistema. Más adelante explicaremos detalladamente cómo aplicamos este flujo de diseño en nuestro proyecto.

La adopción de esta tecnología continúa creciendo, mientras que las herramientas de alto nivel evolucionan para ofrecer a los ingenieros e investigadores con diferentes niveles de experiencia los beneficios de los dispositivos reprogramables. Anteriormente, los ingenieros más especializados en diseño hardware digital eran los únicos capaces de trabajar con la tecnología FPGA. Sin embargo, hoy en día, gracias al avance de las nuevas tecnologías, podemos convertir diagramas de bloques gráficos o código C a circuitos de hardware digital[11][12].

Las principales ventajas que nos ofrece esta tecnología se pueden resumir en las siguientes:

- Rendimiento: aprovechando el paralelismo del hardware, las FPGAs superan la potencia de cómputo de los procesadores digitales de señales, rompiendo el paradigma de ejecución secuencial y logrando realizar más operaciones en cada ciclo de reloj.
- Tiempo en llegar al mercado: la tecnología FPGA ofrece flexibilidad y capacidad para desarrollar rápidamente un prototipo para enfrentar así el reto que supone liberar un producto tardío al mercado.
- Precio: los costes de desarrollo y adquisición son mucho menores en comparación con los de otras tecnologías similares. Además la naturaleza reprogramable de la FPGA nos permite la reutilización del dispositivo tantas veces haga falta así como también nos evitará tener que invertir más recursos en nuevos dispositivos en caso de que los requerimientos iniciales del sistema cambiaran.
- Fiabilidad: los circuitos de una FPGA son una implementación segura de la ejecución de un programa gracias a que disponen de hardware preciso dedicado a cada tarea.
- Mantenimiento a largo plazo: los chips FPGA son actualizables por lo que no requieren el tiempo y el precio que implica rediseñar la

funcionalidad un sistema completo.

En nuestro proyecto hemos hecho uso de la tecnología FPGA para ejecutar la versión hardware de la NFFT que hemos obtenido durante el desarrollo. En este caso, hemos usado una Zedboard Zynq-7000 de Xilinx[13].

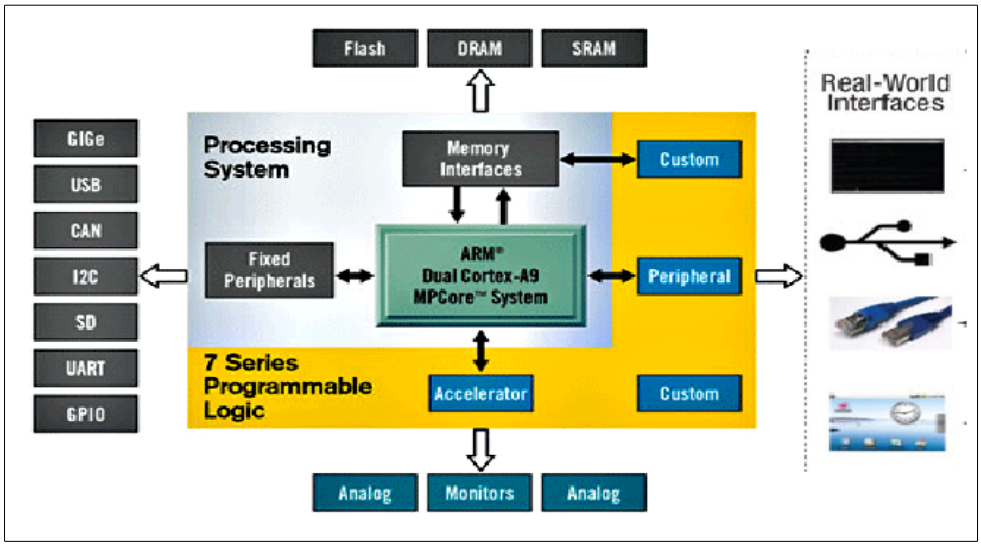

Figura 3.2 Arquitectura de la Zynq-7000.

3.3 SDSoC Development Environment

Ahora que ya hemos descrito las herramientas usadas para diseñar el software que vamos a usar en este proyecto y la tecnología usada para ejecutar nuestro hardware, pasaremos a explicar cómo generar el código hardware necesario para acelerar la NFFT. Para ello hemos usado el entorno de desarrollo SDSoC de Xilinx. Esta novedosa herramienta es la que nos permitirá acelerar nuestro código software y a partir de él generar un código hardware que pueda ser ejecutado por la FPGA.

Esta herramienta nos proporciona un entorno de desarrollo de aplicaciones C/C++ basado en Eclipse que incluye además su propio compilador optimizado. También encontramos en la herramienta un sistema de profiling con el que podremos calcular el rendimiento de nuestro algoritmo y encontrar las funciones del mismo que más recursos consuman. Además, el SDSoC también nos estimará de forma rápida qué función de nuestro código es la mejor candidata para acelerar en hardware. La herramienta también nos proporciona librerías que nos permitirán incluir paralelismo y técnicas de aceleración de software de forma eficiente. Además, gracias al software de diseño de Xilinx, Vivado, que viene incluida en el paquete de instalación del SDSoC podremos analizar el esquema hardware que ha sido generado, así

como la interconexión entre cada uno de los módulos del sistema[14].

Pero tal vez, de todas estas, la característica mas interesante que ofrece el SDSoC sea la posibilidad de generar código hardware para una parte específica de nuestro código software, de forma que podríamos pasar a hardware solo la parte más costosa o ineficiente de nuestro software. Así, el SDSoC se encarga de gestionar la comunicación entre nuestros módulos hardware y software, así como toda la transferencia de datos entre ambos; otorgándonos mayor flexibilidad a la hora de diseñar nuestro sistema.

Esta característica junto a la innovadora arquitectura Zynq, es lo que nos ha permitido particionar nuestro diseño en módulos hardware y módulos software, funcionando conjuntamente y dándonos la posibilidad de elegir qué partes ejecutar en el ARM y qué partes ejecutar usando la lógica programable, todo esto atendiendo a los requerimientos del sistema[15].

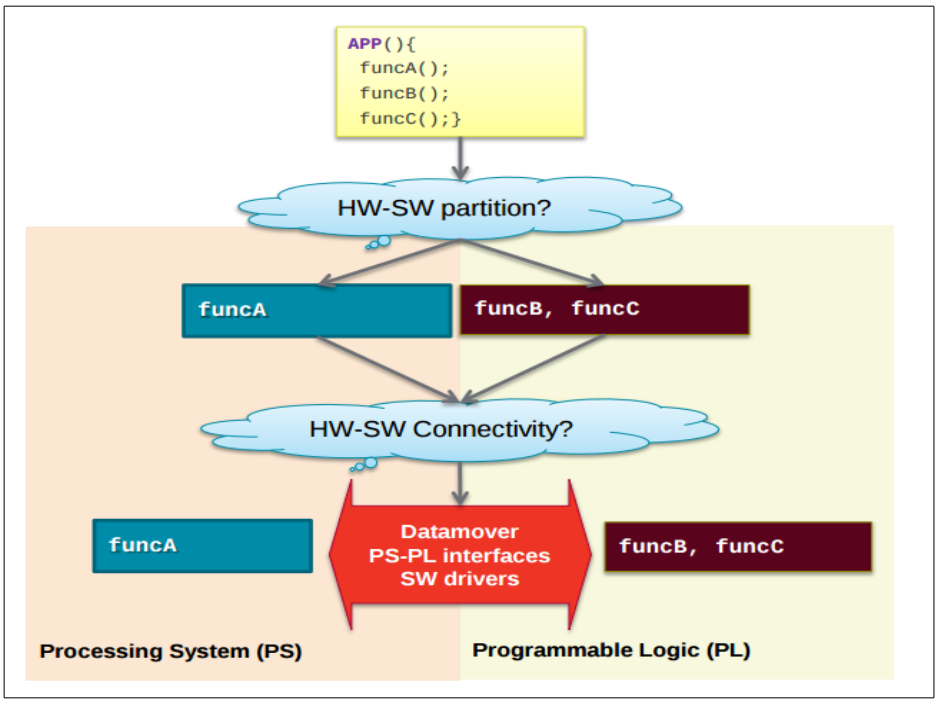

Figura 3.3 Flujo de diseño con el SDSoC y la Zynq.

Gracias a las múltiples ventajas que nos brindaba el SDSoC, nos hemos ahorrado horas de ardua codificación en lenguajes de descripción de hardware además de facilitarnos un entorno de desarrollo sencillo a la vez que potente. De esta forma hemos procesado nuestro código software de la NFFT en dicho entorno y generamos un sistema hardware, al cual más tarde le realizaríamos un análisis de rendimiento con la herramienta de profiling que también nos ofrece el SDSoC.

3.3.1 Pragmas

Como mencionamos anteriormente, el SDSoC nos ofrece la posibilidad de añadir técnicas de paralelismo a nuestro software de manera que podamos acelerar su ejecución de forma eficiente. En este caso, hablaremos de una de las herramientas que nos brindan más versatilidad a la hora de acelerar nuestro código, nos referimos a los pragmas.

Los pragmas son sentencias especiales que controlan la forma en la que el compilador se comportará para un fragmento de código determinado, es decir, son directivas del compilador. Estas directivas se añaden directamente en nuestro código C, afectando únicamente al código al que preceden. A continuación explicaremos los pragmas que hemos utilizado para mejorar el rendimiento de la NFFT.

- Loop Pipelining

Este pragma mejora el rendimiento de una función hardware empleando paralelismo entre las iteraciones de un bucle. En lenguajes secuenciales como C/C++, las operaciones llevadas a cabo dentro un bucle, se ejecutan de manera secuencial, por lo que la siguiente iteración siempre tendrá que esperar a que el último calculo en la iteración actual del bucle finalice. El *pipelining* nos permite implementar las operaciones de un bucle de manera concurrente, como podemos observar en la siguiente figura[16].

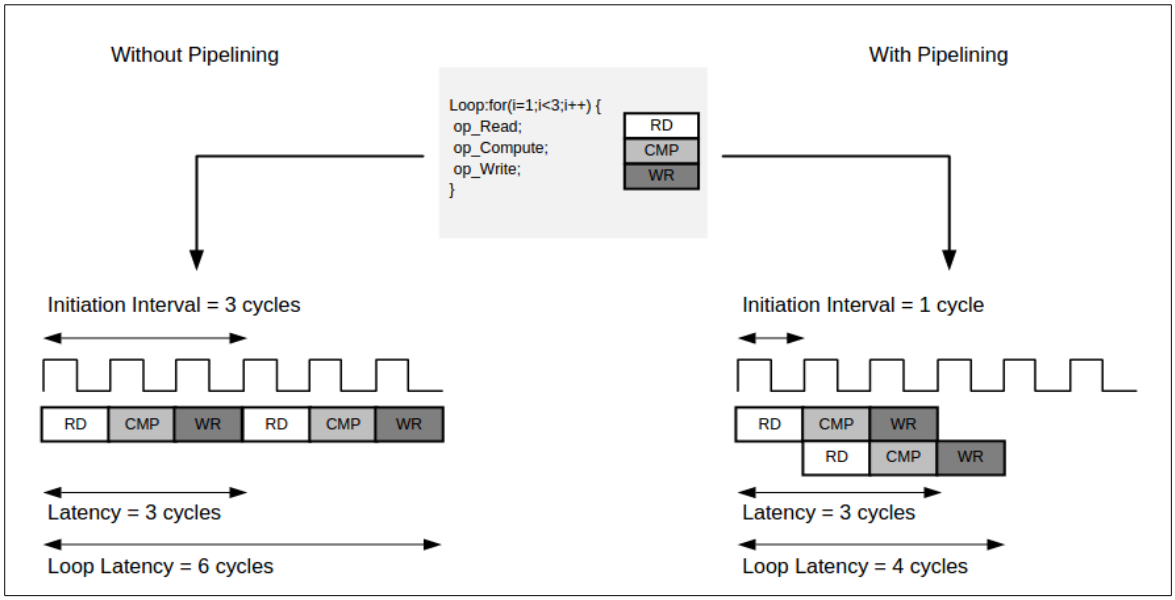

Figura 3.4 Loop pipelining.

Como se muestra en la figura anterior, sin usar paralelismo existen tres ciclos de reloj entre las dos operaciones de lectura y se requieren seis ciclos de reloj para que termine el bucle completo. Mientras tanto, usando paralelismo,

solo hay un ciclo de reloj entre las dos lecturas de datos y se requieren en total cuatro ciclos de reloj para completar el bucle. Lo que quiere decir que somos capaces de comenzar la siguiente iteración del bucle antes de que la actual termine.

Un factor importante a la hora de aplicar esta técnica de paralelismo es el *Intervalo de Iniciación;* el cual hace referencia al numero de ciclos de reloj que existen entre el inicio de dos iteraciones consecutivas del bucle. Esto viene a significar, que si en nuestro bucle existen varias operaciones que accedan al mismo espacio de memoria, las siguientes iteraciones tendrán que esperar irremediablemente a que estas finalicen. Podemos observar esto gráficamente en la siguiente figura.

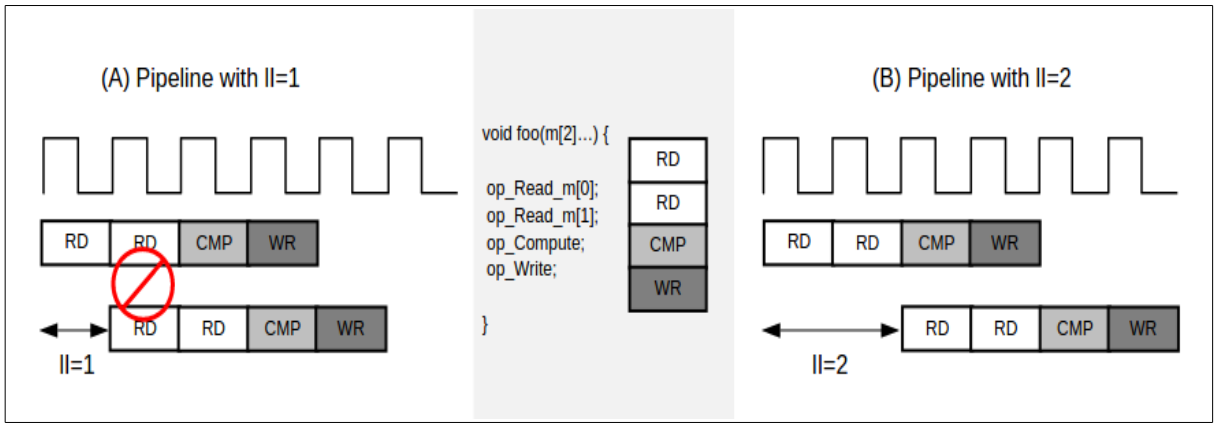

Figura 3.5 Initiation Interval (II).

Para emplear esta directiva en nuestro código, simplemente debemos añadir el siguiente pragma al comienzo del cuerpo del bucle: **#pragma HLS PIPELINE II=1.**

- Function Inlining

Esta directiva es similar a las usadas en varios lenguajes de programación para optimizar funciones software, de la misma forma el SDSoC la utiliza para acelerar funciones hardware. Su utilidad reside en sustituir la llamada a una función determinada de nuestro código por una copia del cuerpo de dicha función. Después de esto, la copia desaparecerá y la función ya no se mostrará en un nivel distinto de la jerarquía de nuestro código[17].

Esta directiva elimina cualquier posible jerarquía dentro de una función, permitiéndonos así optimizar cualquier operación que pueda estar dentro de dicha función. Por tanto, logramos de esta manera mejorar la latencia global del sistema o el intervalo de iniciación para un bucle determinado.

Para emplear esta directiva en nuestro código, simplemente debemos añadir el siguiente pragma al comienzo del cuerpo de la función que se desee optimizar: **#pragma HLS INLINE self.**

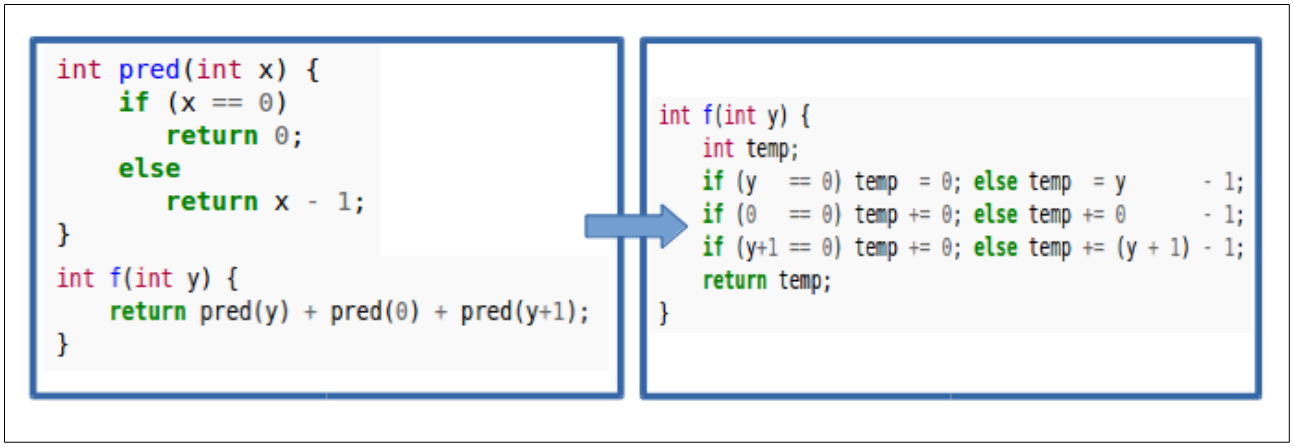

Figura 3.6 Ejemplo de Function Inlining.

- Data Access Pattern

Esta instrucción especifica el patrón de acceso a los datos dentro de una función hardware. El SDSoC diferencia dos tipos de patrones de acceso, el secuencial y el aleatorio. Si el patrón de acceso es secuencial, se genera una interfaz de streaming, mientras que si el patrón de acceso es aleatorio se genera una interfaz RAM[18].

Este pragma se debe especificar justo antes de la declaración de la función que se va a acelerar. La sintaxis de la directiva es: **#pragma SDS data access_pattern(ArrayName:pattern)**; donde **pattern** puede tomar los valores **SEQUENTIAL** o **RANDOM**.

- Loop Unrolling

Esta directiva introduce paralelismo entre las iteraciones de un bucle, creando múltiples copias del cuerpo del mismo e incrementando su índice acorde al número de copias creadas[16].

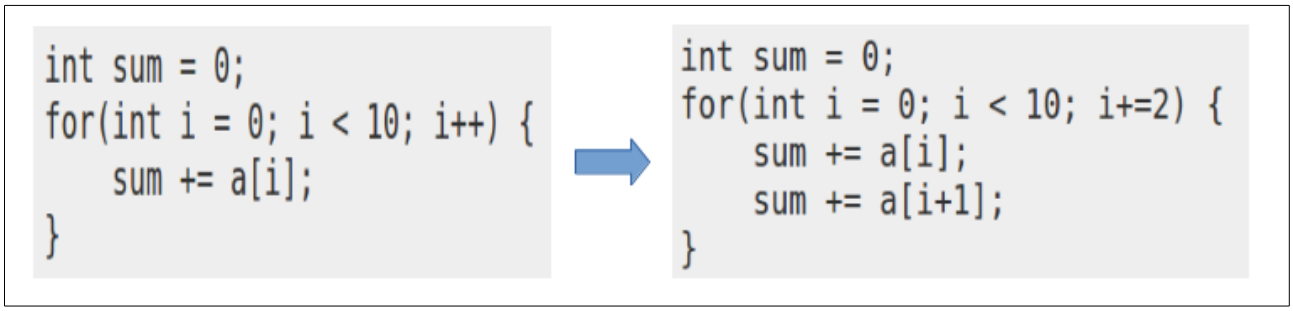

Figura 3.7 Loop Unroll con factor 2.

Como vemos en la figura anterior, se han creado dos copias del cuerpo del bucle, luego el SDSoC se encarga de calcular de forma automática el nuevo incremento del índice. De esta forma se generan más operaciones dentro de la

misma iteración, aumentando así el rendimiento del sistema.

Se denomina *desenrollado parcial,* si el factor que usamos es menor que el límite superior del bucle, mientras que denominamos *desenrollado total* cuando el factor es igual que su límite superior. En este último caso, es necesario conocer el límite del bucle en tiempo de compilación.

Para aplicar este pragma, simplemente debemos añadir al principio del bucle que queramos acelerar, la instrucción: **#pragma HLS unroll [factor=N]**. Si no le pasamos el parámetro **factor**, por defecto se realizará un *desenrollado total* del bucle.

3.4 Acceso a escritorio remoto

Durante el desarrollo del proyecto hemos hecho uso de una herramienta de conexión remota, con el que tuvimos acceso las 24 horas del día, los 7 días de la semana, tanto a los dispositivos como al entorno de trabajo. Esto fue posible gracias al servicio de conexión de red privada virtual VPN (*Virtual Private Network*) que ofrece el Servicio TIC de la ULL[19], y que nos permite acceder a nuestros recursos de forma segura desde Internet. Mediante este servicio se habilita la posibilidad de utilizar servicios y contenidos cuyo acceso está limitado por políticas de seguridad a la red interna de la universidad.

Para ello solo tuvimos que configurar el correspondiente cliente VPN y así disponer de una conexión a la red de la ULL desde cualquier punto de Internet. En nuestro caso, hemos usado el Cliente de escritorio remoto *Remmina*[20] para Ubuntu, y el cliente de VPN *Global Protect*[21] para Windows. Sobra decir que un requisito fundamental para poder utilizar esta tecnología es que los equipos estuvieran funcionando y los dispositivos y periféricos necesarios estuvieran debidamente conectados.

Una vez instalado los clientes de conexión remota, pasamos a configurarlos. En ambos casos, los pasos fueron los mismos. Primeramente, es necesario establecer la conexión VPN con la red de la universidad, para ello debemos especificar la dirección del servidor VPN de la ULL (**vpn.ull.es**) e introducir nuestras credenciales de acceso. Una vez establecida la conexión, tan sólo tendremos que introducir la dirección IP del equipo al que queramos acceder y la contraseña del mismo, si la tuviese, de esta forma el acceso a nuestros recursos estaría finalizado.

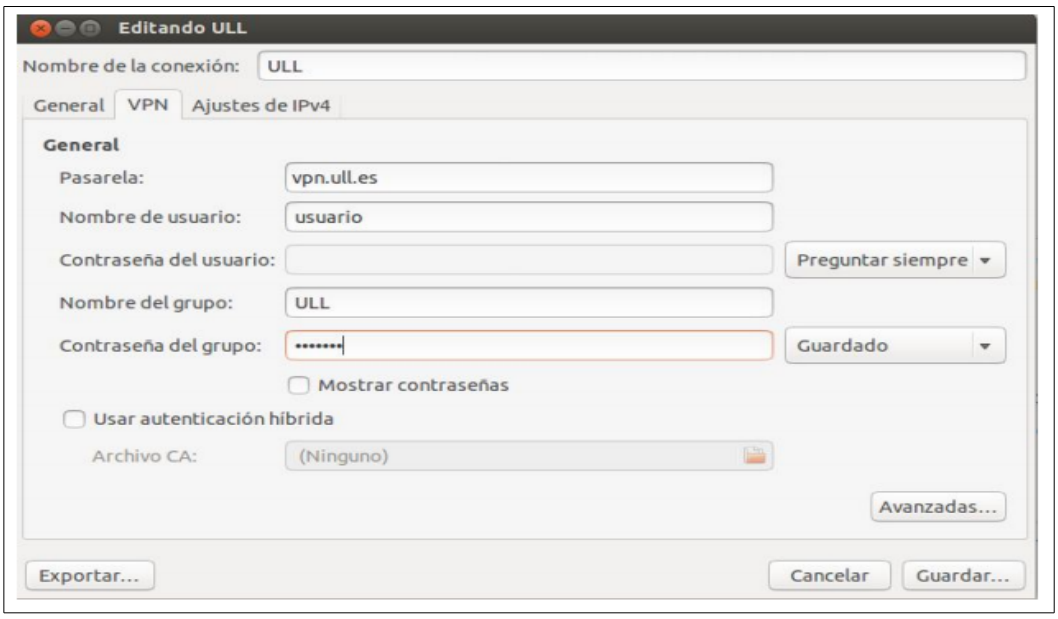

Figura 3.8 Configuración de cliente de escritorio remoto.

Gracias a esta tecnología fue posible dedicar más tiempo al proyecto y ampliar los horarios de trabajo, sobretodo cuando era necesario aumentar la carga de trabajo o no era posible acceder al laboratorio por algún motivo.

Capítulo 4
Proceso de desarrollo

En este capítulo explicaremos el flujo de trabajo llevado a cabo para la realización del proyecto, incluyendo también algunos detalles más específicos de nuestra implementación del algoritmo.

4.1 Versión software

Como ya comentamos al principio de este documento, el primer paso en el desarrollo del proyecto fue crear una versión software de la NFFT, que nos sirviera como base de la cual partir. Cabe recordar que en primer lugar, se llevó a cabo una primera codificación del algoritmo en lenguaje M, propio de Matlab, con la idea de afianzar la base matemática en la que se sustenta el algoritmo.

Sin embargo, resulta más interesante entrar en detalle en la segunda codificación llevada a cabo en lenguaje C, debido a que nos permite bajar un nivel más de abstracción y acercarnos al esquema hardware que planteamos en un principio. Esta versión del algoritmo es fundamental, ya que a partir de ella generaremos nuestro código hardware con el SDSoC, por lo que es importante asimilar por completo el flujo de datos del algoritmo así como intentar optimizar el mismo lo máximo posible.

Es importante destacar que en un principio hemos trabajado con tamaños de datos pequeños para asimilar conceptos, más adelante también comentaremos los distintos tamaños de datos que usamos para llevar a cabo las pruebas de rendimiento. Además, hemos decidido trabajar sólo con una dimensión de datos, es decir, vectores unidimensionales, dejando para desarrollos futuros la posibilidad de ampliar estos cálculos contemplando más dimensiones de datos.

En primer lugar, hemos implementado un versión software básica del algoritmo de fuerza bruta. A pesar de ser menos eficiente que la NFFT, nos ha sido de utilidad para comprobar los resultados que obteníamos al

desarrollar cada uno de los módulos de la NFFT. De esta forma, a medida que íbamos desarrollando un nuevo módulo del algoritmo comprobamos que la funcionalidad era correcta comparando los resultados con los obtenidos por la NDFT para el mismo conjunto de datos de entrada.

Una vez que teníamos una versión correcta de la NFFT, matemáticamente hablando, empezamos a optimizar y a estructurar el algoritmo. Una de las mejoras más básicas fue hacer que, tanto la entrada como la salida del algoritmo, vinieran en ficheros de texto que serían procesados por la NFFT.

Para el módulo de la FFT hemos usado un código externo[22], el cual está basado en el Lema de Danielson-Lanczos. Dicho lema propone aplicar la estrategia *"divide y vencerás"*, de forma que podamos dividir recursivamente en dos grupos el número total de puntos por analizar, hasta llegar al nivel en el que se requiera el cálculo de la transformada de dos muestras. Dicha estrategia implica que tendremos que usar muestras cuyos tamaños sean potencia de dos, para que el funcionamiento del algoritmo sea el correcto. Esto nos ofrece una ventaja considerable frente al algoritmo de la DFT, ya que, como hemos comentando anteriormente, la complejidad computacional de este modulo es de $O(N \log N)$.

Para el tratamiento de datos que realizamos dentro del algoritmo, hemos querido simplificar el tipo de datos complejos que se utiliza en los distintos módulos del algoritmo. Así pues, en lugar de utilizar el tipo de dato *complex,* definido en la librería estándar de C, hemos usado vectores de tipo *float* en los que cada número complejo se almacena secuencialmente y cuyas parte real e imaginaria son posiciones contiguas de dicho vector. De esta forma, a la hora de llevar nuestro código a hardware, será mucho más fácil de procesar por la FPGA.

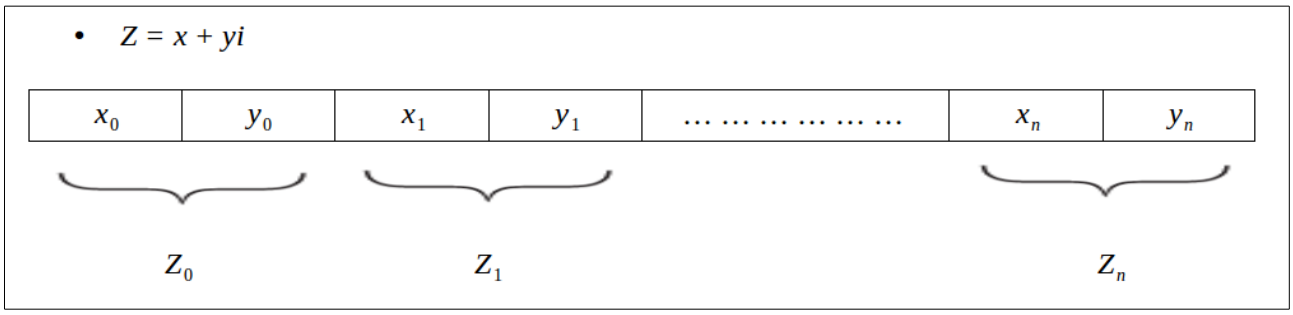

Figura 4.1 Estructura de datos para números complejos.

Por lo tanto, la estructura completa de nuestro software, incluyendo los tres módulos, la lectura del fichero con los coeficientes complejos de Fourier y las muestras irregulares, y la escritura del fichero con las muestras complejas

reordenadas de forma regular en el dominio tiempo/espacial, seguiría el siguiente esquema:

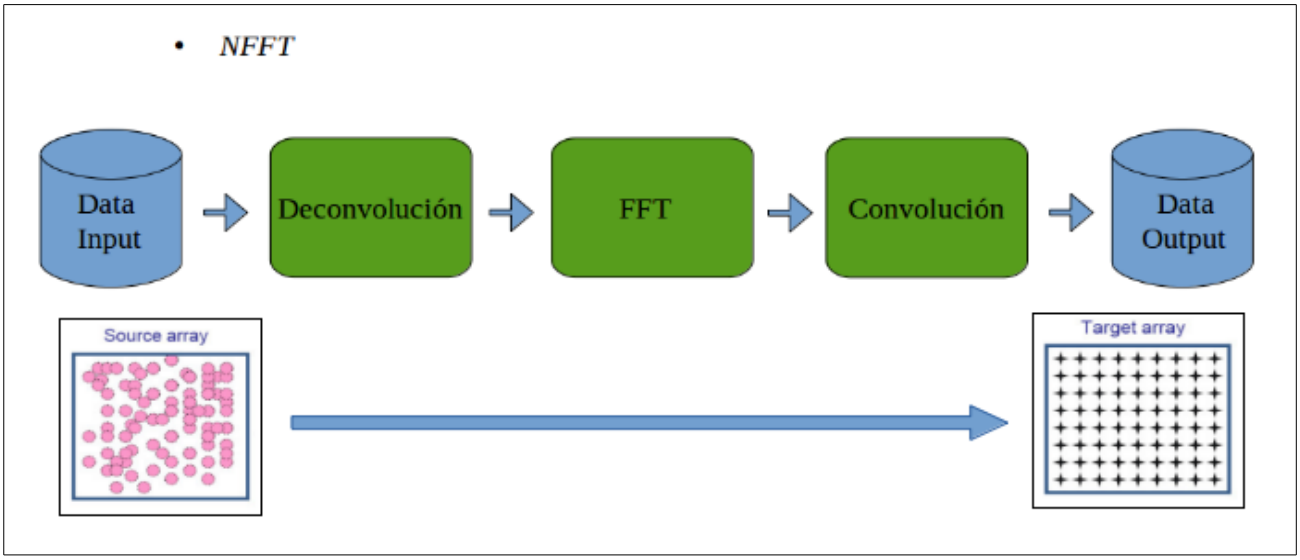

Figura 4.2 Estructura software de la NFFT.

4.2 Versión hardware

Una vez que obtuvimos una versión software eficiente y sólida, el siguiente paso fue empezar a trabajar con el entorno de desarrollo SDSoC. Esta herramienta ofrece múltiples ventajas, la primera que apreciamos es el entorno de edición y generación de código del que disponemos, el cual está basado en el IDE Eclipse. Por lo tanto, esto nos facilita la tarea de portar nuestro código C y seguir trabajando en base a él, en el entorno que nos ofrece el SDSoC.

La metodología que hemos seguido fue la de compilar cada módulo por separado y generar el código hardware para cada uno de ellos con el SDSoC, y junto con las características que nos brinda la arquitectura Zynq, ejecutar el resto de los módulos en hardware o software según convenga.

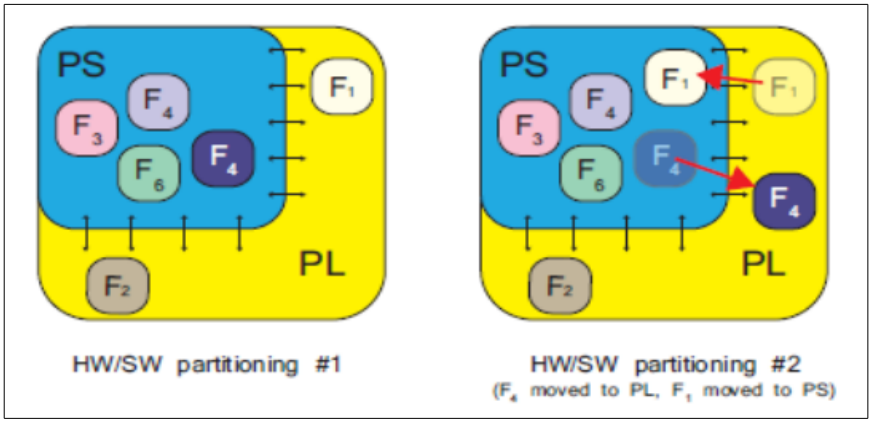

Figura 4.3 Ejemplo de flujo de diseño modular con la Zynq[10].

Para ello debemos seleccionar en la herramienta, qué función de nuestro código queremos acelerar. A partir de esto, el SDSoC nos dice cómo debemos modificar nuestro código de forma que se ajuste a los requerimientos que especifica la herramienta para generar un código hardware eficiente, ya sea especificar de forma explicita los límites de un bucle, simplificar el cuerpo de una función o eliminar cálculos redundantes de alguna variable. Además, este es el momento en el que también aplicamos las técnicas de paralelismo y optimización de software que explicamos antes. Si todo funciona correctamente, el SDSoC genera una imagen con un ejecutable de nuestro algoritmo hardware, la cual copiaremos en una tarjeta SD para ejecutar posteriormente en nuestra ZYNQ. Por último comprobaremos los resultados mediante una terminal conectada a uno de los puertos de la FPGA.

Entrando un poco más en detalle, la estructura del código de cada módulo en hardware comienza reservando la memoria para las variables que necesitaremos, luego se realiza la llamada a la función principal, en la que se crean *buffers* de memoria local para cada uno de los vectores de entrada, de esta forma se gestionan más eficientemente los accesos a memoria. A continuación se hace una llamada a una función *kernel*, que es la que contiene la funcionalidad básica del módulo. Finalmente, se devuelve el resultado final y se libera la memoria en uso.

Una vez que compilamos todos y cada uno de los módulos del sistema, procedemos a unirlos, volviendo a la estructura que seguía nuestra versión software y que podemos apreciar en la siguiente figura. Ahora los tres módulos del algoritmo estarán acelerados en hardware, mientras que la entrada y salida de datos, así como la inicialización y la liberación de la memoria, se mantendrán en software, encargándose el SDSoC de gestionar la comunicación entre el microprocesador y la lógica programable de nuestra Zynq.

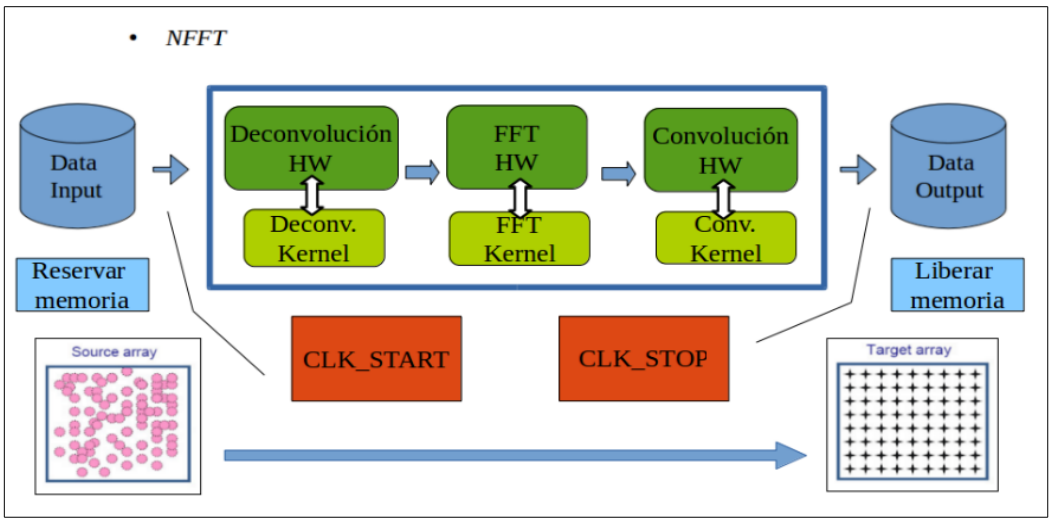

Figura 4.4 Estructura hardware de la NFFT.

4.3 Pruebas de rendimiento

Ahora que ya hemos obtenido una versión hardware a partir de nuestro algoritmo inicial en software es momento de analizar los tiempos de ejecución de ambas. Para ello hemos hecho uso de la librería de profiling *sds_lib* que ofrece el SDSoC[23].

Esta herramienta de *profiling* nos proporciona un método estático para medir el rendimiento nuestro código basado en el muestreo del contador de programa de la CPU. De esta forma, además de comprobar cuáles son las partes más ineficientes de nuestro código también podemos realizar una comparación entre nuestras dos versiones del algoritmo.

La librería *sds_lib* proporciona una notación simple basada en marcas de tiempo, con la que podremos medir el rendimiento del código calculando la diferencia entre cada una de ellas. Para ello haremos uso de las directivas *sds_clock_start* y *sds_clock_stop,* con las que capturaremos las marcas de tiempo al iniciar y finalizar la ejecución de la función respectivamente. Finalmente, con la instrucción *avg_cpu_cycles* obtendremos como resultado los ciclos de CPU consumidos por el algoritmo.

```
void measure_f_runtime()
{
    for (int i = 0; i < NUM_TESTS; i++) {
        sds_clock_start();
        f();
        sds_clock_stop();
    }
    printf("Average cpu cycles f(): %ld\n", avg_cpu_cycles());
}
```

Figura 4.5 Ejemplo de uso de la *sds_lib*.

Capítulo 5
Análisis de resultados

En este capítulo expondremos los resultados que hemos obtenido durante la fase de diseño. En primer lugar, mostraremos los resultados obtenidos al acelerar por separado cada uno de los módulos de la NFFT. Seguidamente, se mostrarán los resultados de forma global, una vez unidos todos los módulos del algoritmo. Las pruebas que hemos realizado han sido con vectores unidimensionales de con tamaño 32 y 1024.

5.1 Deconvolución

Tamaño de la muestra	Software			Hardware			% de Mejora
	Ciclos CPU	% ARM	% LP	Ciclos CPU	% ARM	% LP	
32 (2^5)	265328	24,6%	0%	14652	4,1%	19,7%	94,5%
1024 (2^{10})	669142	31,1%	0%	18064	7,4%	28,3%	97,3%

Tabla 5.1. Resultados de la deconvolución.

- Ciclos CPU: ciclos de CPU consumidos por la función.
- % ARM: porcentaje de memoria del ARM usada por la función.
- % LP: porcentaje de recursos de la lógica programable utilizados por la función.
- % de Mejora: porcentaje de mejora de la ejecución hardware en comparación con la software.

5.2 FFT

Tamaño de la muestra	Software			Hardware			% de Mejora
	Ciclos CPU	% ARM	% LP	Ciclos CPU	% ARM	% LP	
32 (2^5)	252298	25,2%	0%	7844	4,1%	23,4%	96,9%
1024 (2^{10})	3046906	31,9%	0%	7998	7,4%	35,7%	99,7%

Tabla 5.2. Resultados de la FFT.

5.3 Convolución

Tamaño de la muestra	Software			Hardware			% de Mejora
	Ciclos CPU	% ARM	% LP	Ciclos CPU	% ARM	% LP	
32 (2^5)	159932	61,3%	0%	143336	4,1%	54,5%	10,3%
1024 (2^{10})	5992656	89,9%	0%	5370804	7,4%	68,3%	10,4%

Tabla 5.3. Resultados de la convolución.

5.4 Análisis global

Tamaño de la muestra	Software			Hardware			% de Mejora
	Ciclos CPU	% ARM	% LP	Ciclos CPU	% ARM	% LP	
32 (2^5)	678094	69,9%	0%	166582	4,1%	61,5%	75,4%
1024 (2^{10})	9709418	93,6%	0%	5399846	7,4%	91,3%	44,4%

Tabla 5.4. Resultados de la NFFT.

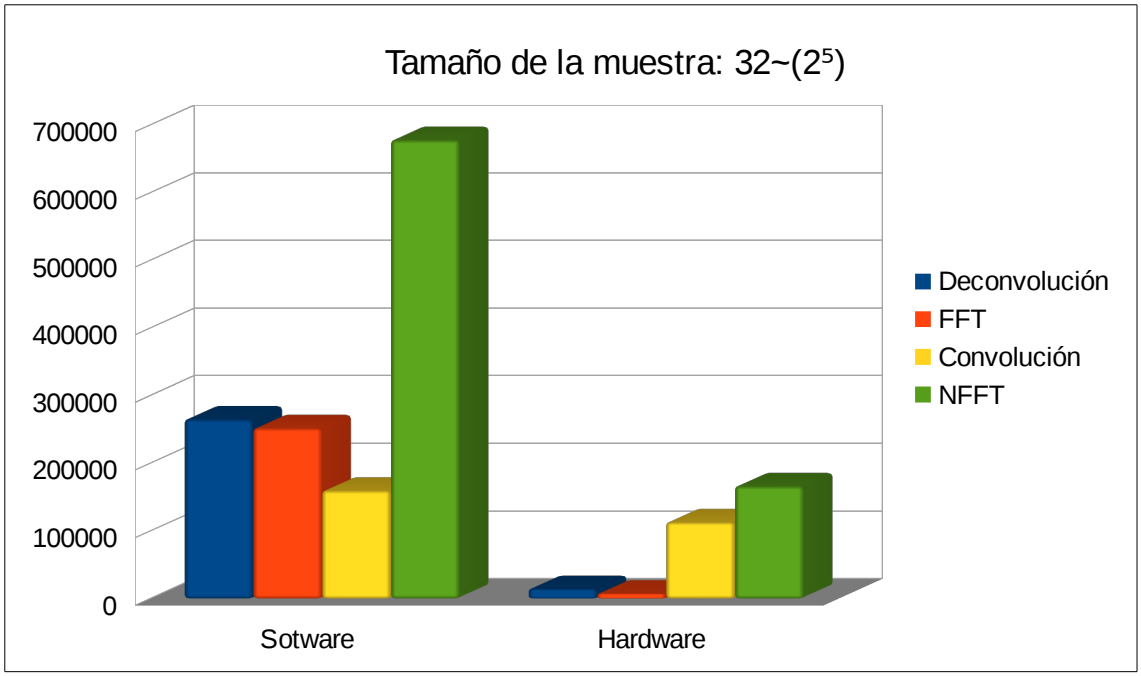

Figura 5.1 Gráfica para muestras pequeñas.

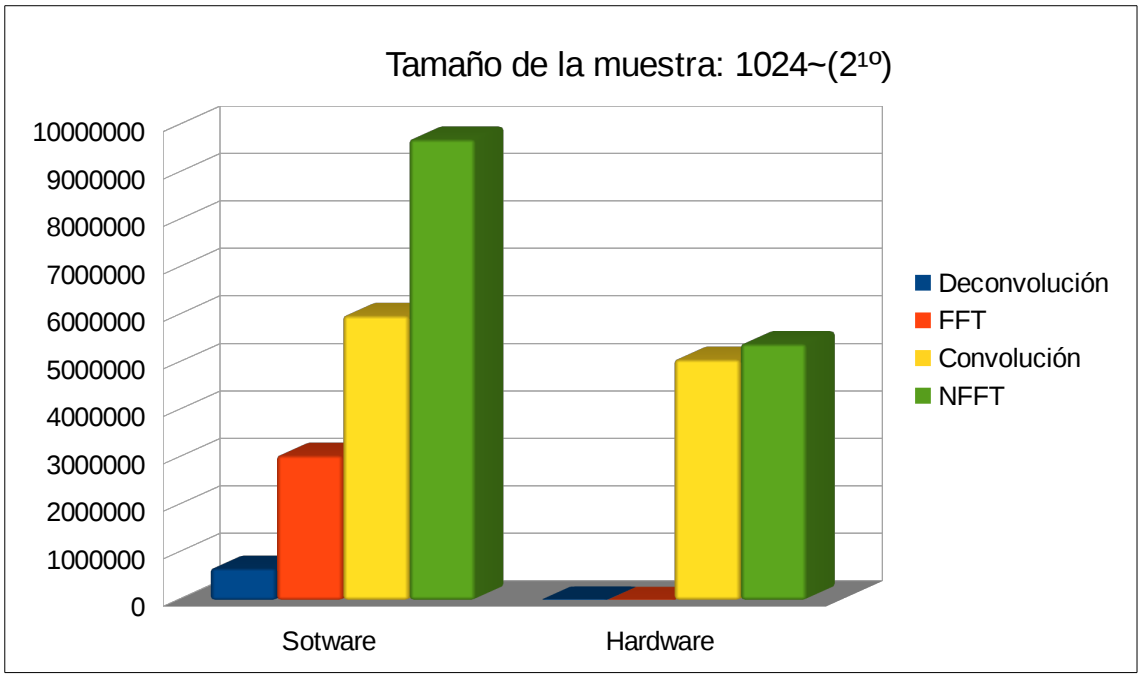

Figura 5.2 Gráfica para muestras grandes.

Como podemos observar, el tiempo de ejecución de la NFFT viene determinado por el tiempo consumido en el cómputo de la convolución, cuyo coste computacional es el mayor de los tres módulos del algoritmo. Por lo tanto, el principal reto que se nos presenta es el de acelerar dicho módulo y mejorar así la ejecución general de la NFFT.

Sin embargo, en líneas generales, podemos observar que implementando nuestro código en hardware y aplicando las técnicas de paralelismo antes descritas, se mejoran notablemente cada uno de los módulos, así como también el algoritmo completo, una vez que se han unido los tres módulos. Por lo tanto, queda patente la ventaja que nos ofrece la lógica programable a la hora de implementar algoritmos cuyo coste computacional y consumo de recursos sean elevados.

A continuación, expondremos una serie de conclusiones que hemos extraído una vez que hemos finalizado el proceso de desarrollo del presente proyecto.

Capítulo 6
Conclusiones y líneas futuras

A modo de conclusión, me gustaría destacar las numerosas ventajas, ya mencionadas anteriormente, que ofrece la utilización de la tecnología FPGA a la hora de desarrollar algoritmos con diseños mucho más eficientes. Creemos que es necesario seguir investigando acerca de este campo y sus múltiples aplicaciones en ingeniería e investigación científica de diversa índole. Asimismo, es importante mencionar también, la versatilidad y potencia que nos brinda una herramienta de alto nivel como el SDSoC, de la que se pueden beneficiar ingenieros e investigadores con diferentes niveles de experiencia en el diseño avanzado de circuitos digitales.

Como contrapartida, una dificultad con la que me he encontrado a lo largo del desarrollo, fueron los elevados tiempos de compilación del código hardware, esto es debido al bajo rendimiento de los ordenadores utilizados para el desarrollo del proyecto, lo cual ralentizaba en gran medida el flujo de trabajo.

Nuestro objeto de investigación ha sido el algoritmo de la NFFT; un algoritmo ampliamente utilizado en el análisis y procesamiento de señales, cuando éstas no siguen un patrón regular o una trayectoria uniforme. Con nuestro proyecto hemos querido profundizar un poco más en este algoritmo, aplicando técnicas de paralelismo y de aceleración en hardware, obteniendo resultados muy prometedores en comparación con las versiones software iniciales; de lo cual podemos sacar una lectura muy positiva del trabajo llevado a cabo.

Como posible desarrollo futuro, sería interesante seguir profundizando en el módulo de la convolución de señales, ya sea aplicando nuevas técnicas de paralelismo u optimizando aún más la estructura del código. Asimismo, otra posible mejora, es la posibilidad de contemplar más dimensiones de datos a la hora de realizar el cómputo de la NFFT, usando matrices y vectores multidimensionales como estructura de datos.

Por último, a nivel personal, el Trabajo de Fin de Grado me ha servido para aprender mucho más acerca del diseño hardware, el cual es un campo que me ha resultado muy interesante y que actualmente cuenta con un gran crecimiento y perspectivas de futuro. Y para terminar, he aprendido que pese a todos los obstáculos y fracasos que podamos tener, lo importante es levantarse y seguir adelante.

Capítulo 7
Summary and Conclusions

In conclusion, I would highlight the many advantages offered by the use of FPGA technology in algorithms development with more efficient designs; we believe that further research is needed about this field and its many applications in engineering and scientific research of various kinds. It is also important to mention also the versatility and power that gives us a high-level tool such as SDSoC, which can benefit engineers and researchers with different levels of experience in advanced digital circuit design.

In return, a difficulty which I have dealt with throughout the development were the high times of compiling the hardware code, this is due to the poor performance of the computers used for the project, which slowed greatly workflow.

Our object of research has been the NFFT algorithm; an algorithm that is widely used in the analysis and signals processing, when those signals don't follow a regular pattern or a uniform path. With our project we wanted to dig a little deeper in this algorithm using techniques of paralellism and hardware acceleration, obtaining very promising results compared with the initial software versions; from which we can draw a positive reading of the work carried out.

As a possible future development, would be interesting to further deepen in the convolution module, either by applying new techniques of parallelism or further optimizing the structure of the code. Likewise, another possible improvement is the chance to include more data dimensions to performing the computation of the NFFT, using matrices and multidimensional vectors as data structure.

Finally, on a personal level, this project has helped me to learn more about hardware design, which is a very interesting field and currently has a high growth and great future prospects. And finally, I learned that despite all obstacles and failures that we have, the most important thing is to get up and keep moving forward.

Capítulo 8
Presupuesto

En este capítulo detallaremos el presupuesto total que ha sido necesario para el desarrollo de este proyecto.

8.1 Personal

El desarrollo del proyecto ha sido llevado a cabo en su totalidad por Cristhian Javier García Conrado junto con la participación y ayuda del tutor y cotutores para la resolución de dudas y problemas que han ido surgiendo, así como para aportar sugerencias y posibles mejoras.

En total, el proyecto se ha desarrollado en 300h estimadas, cumpliendo los requisitos de la Guía Docente de la asignatura. Para cada una de las tareas programadas inicialmente se han empleado el siguiente número de horas:

Tarea	Nº de horas
Primera codificación software (Lenguaje M)	40
Segunda codificación software (Lenguaje C)	60
Versión hardware (Entorno de desarrollo SDSoC)	100
Aplicar técnicas de paralelismo (Pragmas)	80
Profiling y análisis de rendimiento	20
Total	**300**

Tabla 8.1. Estimación de horas por tarea realizada.

En este desglose no hemos tenido en cuenta el tiempo dedicado a la asistencia a seminarios, las reuniones con los tutores del proyecto, la redacción de documentos y otras tareas no relacionadas directamente con el proyecto.

8.2 Componentes

Los componentes de los que hemos hecho uso a lo largo del desarrollo del proyecto se listan a continuación.

Tarea	Nº de horas
Licencia Xilinx SDSoC Development Environment	885 €
FPGA Zedboard Zynq-7000 de Xilinx	425 €
Tarjeta de memoria SD	5 €
Total	**1310 €**

Tabla 8.2. Componentes usados en el proyecto.

Cabe destacar que todos estos componentes han sido aportados por los tutores o bien el alumno ya los poseía, por lo que no ha hecho falta adquirir nada nuevo.

8.3 Coste total

El presupuesto total aproximado que ha sido necesario para el desarrollo del presente Trabajo de Fin de Grado, sumando los cálculos de los apartados anteriores y tomando como referencia un coste de 10€/hora, nos da un total de:

Componente	Coste
Total en componentes	1310 €
Total en horas (300 * 10)	3000 €
Total	**4310 €**

Tabla 8.3. Presupuesto total.

Capítulo 9
Código software destacable

9.1 Deconvolución

```
/*****************************************************************************
* nuffts.c
*****************************************************************************
* AUTOR: Cristhian Javier García Conrado
* FECHA: 26/06/2016
* DESCRIPCION: Función que realiza la deconvolución de dos señales.
*              Recibe como entrada el vector complejo 'fk' y devuelve la
*              salida en el vector 'gk'.
*****************************************************************************/

void deconvolution(float fk[], float gk[])
{
      printf("\n* Calculando la DECONVOLUCION...\n");
      int cont = 0, k;

      for (k=-(N/2); k<(N/2); k++){
            gk[2*cont+1] = fk[2*cont+1] / exp(-b * pow(((PI * k) / n),2));
            gk[2*cont+2] = fk[2*cont+2] / exp(-b * pow(((PI * k) / n),2));
            cont++;
      }
}
```

9.2 FFT

```
/*************************************************************************
* nuffts.c
*************************************************************************
* AUTOR: Cristhian Javier García Conrado
* FECHA: 26/06/2016
* DESCRIPCION: Esta función realiza la FFT del vector 'gk' y devuelve el
*              resultado en el vector 'gl'. Se divide en:
*              - four1: Algoritmo de la FFT basado en el lema de
*                       Danielson-Lanczos.
*              - myfft: Algoritmo encargado de corregir el error de
*                       desplazamiento devuelto por la 'four1'.
*************************************************************************/
void myfft(float gk[], float gl[])
{
     printf("\n* Calculando la FFT...\n");
     int l, k = -(N/2), contj = 0, conti = 0;
     float gl_aux[(int)(2*n+1)];

     for (l=-(n/2); l<(n/2); l++){
          gl_aux[2*conti+1] = 0.00;
          gl_aux[2*conti+2] = 0.00;
          if ((l >= k) && (l < (N/2))){
               gl_aux[2*conti+1] = gk[2*contj+1] * pow(-1, k + n/2);
               gl_aux[2*conti+2] = gk[2*contj+2] * pow(-1, k + n/2);
               contj++;
               k++;
          }
          conti++;
     }
     four1(gl_aux, n, 1);
     for (l=0; l<n; l++){
       if (l == 0){
         gl[2*l+1] = gl_aux[2*l+1] * pow(-1, l) * pow(-1, n/2);
         gl[2*l+2] = gl_aux[2*l+2] * pow(-1, l) * pow(-1, n/2);
       }else{
         gl[2*l+1] = gl_aux[2*(int)(n-l)+1] * pow(-1, l) * pow(-1, n/2);
         gl[2*l+2] = gl_aux[2*(int)(n-l)+2] * pow(-1, l) * pow(-1, n/2);
       }
    }
}
```

```
void four1(float data[], int nn, int isign) {
    int xn, mmax, m, j, istep, i;
    float wtemp, wr, wpr, wpi, wi, theta;
    float tempr, tempi;

    xn = nn << 1;
    j = 1;
    for (i = 1; i < xn; i += 2) {
        if (j > i) {
            tempr = data[j];     data[j] = data[i];     data[i] = tempr;
            tempr = data[j+1]; data[j+1] = data[i+1]; data[i+1] = tempr;
        }
        m = xn >> 1;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    mmax = 2;
    while (xn > mmax) {
          istep = 2*mmax;
          theta = (2.0*PI)/(isign*mmax);
          wtemp = sin(0.5*theta);
          wpr = -2.0*wtemp*wtemp;
          wpi = sin(theta);
          wr = 1.0;
          wi = 0.0;
          for (m = 1; m < mmax; m += 2) {
               for (i = m; i <= xn; i += istep) {
                    j =i + mmax;
                    tempr = wr*data[j]   - wi*data[j+1];
                    tempi = wr*data[j+1] + wi*data[j];
                    data[j]   = data[i]   - tempr;
                    data[j+1] = data[i+1] - tempi;
                    data[i] += tempr;
                    data[i+1] += tempi;
               }
               wr = (wtemp = wr)*wpr - wi*wpi + wr;
               wi = wi*wpr + wtemp*wpi + wi;
          }
          mmax = istep;
    }
}
```

9.3 Convolución

```
/*****************************************************************************
* nuffts.c
*****************************************************************************
* AUTOR: Cristhian Javier García Conrado
* FECHA: 26/06/2016
* DESCRIPCION: Esta función realiza la convolución entre el vector 'gl'
*              y el vector de muestras 'x'. La salida final se almacena
*              en el vector 'sj'.
*****************************************************************************/

void convolution(float gl[], float sj[], float x[])
{
   printf("\n* Calculando la CONVOLUCION...\n");
   int j, l, index;
   float a;

   for (j=0; j<M; j++){
        sj[2*j+1] = 0.0;
        sj[2*j+2] = 0.0;

        for (l=ceil(n*x[j]-m); l<=floor(n*x[j]+m); l++){
            index = ((int)(l+n/2) % (int)n) + 1;
            a = (n * (x[j] - l/n)) * (n * (x[j] - l/n));
            sj[2*j+1] += gl[2*(index-1)+1] * (exp(-a/b) / sqrt(PI * b));
            sj[2*j+2] += gl[2*(index-1)+2] * (exp(-a/b) / sqrt(PI * b));
        }
    }
}
```

Capítulo 10
Código hardware destacable

10.1 Deconvolución

```
/*****************************************************************************
* deconv.c
*****************************************************************************
* AUTOR: Cristhian Javier García Conrado
* FECHA: 26/06/2016
* DESCRIPCION: Función que realiza la deconvolución de dos señales.
*              Recibe como entrada el vector complejo 'fk' y un vector
*              'op' que contiene las exponenciales necesarias para el
*              cálculo. Devuelve la salida en el vector 'gk'.
*****************************************************************************/
#pragma SDS data access_pattern(fk:SEQUENTIAL, op:SEQUENTIAL)
void deconvolution(float fk[SZ4_D], float gk[SZ4_D], float op[N_D]) {
     printf("\n* Calculando la DECONVOLUCION...\n");
     int i;
     float a_buf[SZ4_D];
     float b_buf[N_D];
     // Transfer 'fk' from multi-buffer into local RAM
     for(i=0; i<SZ4_D; i++) {
          #pragma HLS unroll
          #pragma HLS PIPELINE II=1
          a_buf[i] = fk[i]; }
     // Transfer 'op' from multi-buffer into local RAM
     for(i=0; i<N_D; i++) {
          #pragma HLS unroll
          #pragma HLS PIPELINE II=1
          b_buf[i] = op[i]; }
     // Deconvolution call
     deconv_kernel(a_buf, b_buf, gk);
}
```

```
void deconv_kernel(float a_buf[SZ4_D], float b_buf[N_D], float
gk[SZ4_D])
{
     #pragma HLS INLINE self
     int i;

     for (i=0; i<N_D; i++){
          gk[2*i+1] = a_buf[2*i+1] / b_buf[i];
          gk[2*i+2] = a_buf[2*i+2] / b_buf[i];
     }
}
```

10.2 FFT

```
/*****************************************************************************
* myfft.c
*****************************************************************************
* AUTOR: Cristhian Javier García Conrado
* FECHA: 26/06/2016
* DESCRIPCION: Esta función realiza la FFT del vector 'gk' y devuelve el
*              resultado en el vector 'gl'. Recibe también los vectores
*              'powa' y 'powb' que contienen cálculos de desplazamientos
*              previamente computados. Se divide en:
*              - four1: Algoritmo de la FFT basado en el lema de
*                       Danielson-Lanczos.
*              - desp1 / desp2: Algoritmos encargados de corregir el
*                       error de desplazamiento devuelto por la 'four1'.
*****************************************************************************/
#pragma SDS data access_pattern(powa:SEQUENTIAL, powb:SEQUENTIAL)
void myfft(float gk[SZ4_F], float powa[N_F], float powb[nn_F], float
gl[SZ1_F])
{
     printf("\n* Calculando la FFT...\n");
     int i;
     float a_buf[N_F];
     float b_buf[nn_F];
     float gl_aux_1[SZ1_F];
     float gl_aux_2[SZ1_F];
     // Transfer 'powa' from multi-buffer into local RAM
     for(i=0; i<N_F; i++) {
          #pragma HLS unroll
          #pragma HLS PIPELINE II=1
          a_buf[i] = powa[i];
     }
```

```
	// Transfer 'powb' from multi-buffer into local RAM
	for(i=0; i<n_F; i++) {
		#pragma HLS unroll
		#pragma HLS PIPELINE II=1
		b_buf[i] = powb[i];
	}
	// Desp1 call
	desp1_kernel(gk, a_buf, gl_aux_1);

	// FFT call
	four1_kernel(gl_aux_1, gl_aux_2);

	// Desp2 call
	desp2_kernel(gl_aux_2, b_buf, gl);
}

void desp1_kernel(float gk[SZ4_F], float powa[N_F], float gl[SZ1_F])
{
	#pragma HLS INLINE self
	int i, min = -(N_F/2), max = (N_F/2), conti = 0, contj = 0;

	for (i=-(n_F/2); i<(n_F/2); i++){
		if ((i >= min) && (i < max)){
			gl[2*conti+1] = gk[2*contj+1] * powa[contj];
			gl[2*conti+2] = gk[2*contj+2] * powa[contj];
			contj++;
		}
		conti++;
	}
}

void desp2_kernel(float gl_aux[SZ1_F], float powb[nn_F], float
gl[SZ1_F])
{
	#pragma HLS INLINE self
	int i;

	gl[1] = gl_aux[1] * powb[0];
	gl[2] = gl_aux[2] * powb[0];
	for (i=1; i<n_F; i++){
		gl[2*i+1] = gl_aux[2*(int)(n_F-i)+1] * powb[i];
		gl[2*i+2] = gl_aux[2*(int)(n_F-i)+2] * powb[i];
	}
}
```

```
#pragma SDS data access_pattern(gl_aux:SEQUENTIAL)
void four1_kernel(float data[SZ1_F], float gl_aux[SZ1_F]) {
     #pragma HLS INLINE self
     int xn, mmax, xm, j, istep, i;
     float wtemp, wr, wpr, wpi, wi, theta;
     float tempr, tempi;
     xn = nn_F << 1;
     j = 1;
     for (i = 1; i < xn; i += 2) {
        if (j > i) {
            tempr = data[j];     data[j] = data[i];     data[i] = tempr;
            tempr = data[j+1]; data[j+1] = data[i+1]; data[i+1] = tempr;
        }
        xm = xn >> 1;
        while (xm >= 2 && j > xm) {
            j -= xm;
            xm >>= 1; }
        j += xm;
     }
     mmax = 2;
     while (xn > mmax) {
          istep = 2*mmax;
          theta = TWOPI/mmax;
          wtemp = sin(0.5*theta);
          wpr = -2.0*wtemp*wtemp;
          wpi = sin(theta);
          wr = 1.0;
          wi = 0.0;
          for (xm = 1; xm < mmax; xm += 2) {
               for (i = xm; i <= xn; i += istep) {
                    j = i + mmax;
                    tempr = wr*data[j] - wi*data[j+1];
                    tempi = wr*data[j+1] + wi*data[j];
                    data[j] = data[i]   - tempr;
                    data[j+1] = data[i+1] - tempi;
                    data[i] += tempr;
                    data[i+1] += tempi; }
               wr = (wtemp = wr)*wpr - wi*wpi + wr;
               wi = wi*wpr + wtemp*wpi + wi; }
          mmax = istep; }
     for(i=0; i<SZ1_F; i++) {
          #pragma HLS PIPELINE II=1
          gl_aux[i] = data[i]; }
}
```

10.3 Convolución

```
/*****************************************************************************
* conv.c
*****************************************************************************
* AUTOR: Cristhian Javier García Conrado
* FECHA: 26/06/2016
* DESCRIPCION: Esta función realiza la convolución entre el vector 'gl'
*              y el vector de muestras 'xp'. La salida final se almacena
*              en el vector 'sj'. Además se usan los vectores 'ce' y
*              'fl', que contienen los valores precalculados de los
*              límites del bucle principal.
*****************************************************************************/
#pragma SDS data access_pattern(ce:SEQUENTIAL, fl:SEQUENTIAL,
xp:SEQUENTIAL)
void convolution(float gl[SZ1_C], float sj[SZ2_C], float ce[M_C], float
fl[M_C], float xp[SZ3_C]) {
     printf("\n* Calculando la CONVOLUCION...\n");
     int i;
     float a_buf[M_C];
     float b_buf[M_C];
     float c_buf[SZ3_C];

     // Transfer 'ce' from multi-buffer into local RAM
     for(i=0; i<M_C; i++) {
          #pragma HLS unroll
          #pragma HLS PIPELINE II=1
          a_buf[i] = ce[i];
     }
     // Transfer 'fl' from multi-buffer into local RAM
     for(i=0; i<M_C; i++) {
          #pragma HLS unroll
          #pragma HLS PIPELINE II=1
          b_buf[i] = fl[i];
     }
     // Transfer 'xp' from multi-buffer into local RAM
     for(i=0; i<SZ3_C; i++) {
          #pragma HLS unroll
          #pragma HLS PIPELINE II=1
          c_buf[i] = xp[i];
     }
     // Convolution call
     conv_kernel(gl, a_buf, b_buf, c_buf, sj);
}
```

```
void conv_kernel(float a_buf[SZ1_C], float b_buf[M_C], float c_buf[M_C],
float d_buf[SZ3_C], float sj[SZ2_C])
{
      #pragma HLS INLINE self
      int i, j, k = 0, index;

      for (i=0; i<M_C; i++){
            float re = 0, im = 0;
            for (j=b_buf[i]; j<=c_buf[i]; j++){
                  index = ((int)(j+n_C/2) % (int)n_C) + 1;
                  float product_re = a_buf[2*(index-1)+1] * d_buf[k];
                  float product_im = a_buf[2*(index-1)+2] * d_buf[k];
                  re += product_re;
                  im += product_im;
                  k++;
            }
            sj[2*i+1] = re;
            sj[2*i+2] = im;
      }
}
```

Bibliografía

[1] NFFT: Agorithm for irregular sampling. https://www.crewes.org/ForOurSponsors/ResearchReports/2009/CRR200923.pdf.

[2] Fast Fourier Transform. http://www.fing.edu.uy/if/cursos/fiscomp/extras/numrec/book/f12.pdf

[3] Nonequispaced fast fourier transform. https://www-user.tu-chemnitz.de/~potts/nfft/

[4] Algoritmo NuFFT. https://drive.google.com/file/d/0B9qdNdJBVeyFQzVjZXN3M21ndjQ/view?usp=sharing.

[5] The Nonequispaced FFT on graphics processing units. http://www.analysis.uni-osnabrueck.de/sukunis/paper/Pamm_KuKu12.pdf.

[6] NFFT 3.0 Tutorial. https://www-user.tu-chemnitz.de/~potts/nfft/guide/nfft3.pdf

[7] Matlab Documentation. http://es.mathworks.com/help/matlab/

[8] GNU Octave. https://www.gnu.org/software/octave/.

[9] The GNU C Reference Manual. https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

[10] The Zynq Book. http://www.zynqbook.com/

[11] Field Programmable Gate Array (FPGA): A Tool For Improving Parallel Computations. http://www.ijser.org/researchpaper%5CFIELD-PROGRAMMABLE-GATE-ARRAY-FPGA-A-TOOL-FOR-IMPROVING.pdf

[12] FPGAs accelerate time to market for industrial designs. http://www.design-reuse.com/articles/8190/fpgas-accelerate-time-to-market-for-industrial-designs.html

[13] Zedboard. http://zedboard.org/product/zedboard.

[14] SDSoC Development Environment Backgrounder. https://www.xilinx.com/support/documentation/backgrounders/sdsoc-development-environment-backgrounder.pdf

[15] SDSoC Technical Seminars. http://tinkerer.us/qualifications/SDSoC/SDSoC.pdf

[16] Loop Pipelining and Loop Unrolling. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html

[17] Function Inlining. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_function_inlining.html

[18] Copy and Shared Memory Semantics. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/system-optimization/concept_copy_shared_memory_semantics.html

[19] Servicio de VPN de la ULL. http://www.ull.es/stic/tag/vpn/.

[20] Remmina. http://www.remmina.org/wp/.

[21] GlobalProtect VPN. https://www.paloaltonetworks.com/products/secure-the-network/subscriptions/globalprotect.

[22] FFT code from the book Numerical Recipes in C. https://software.intel.com/sites/default/files/forum/392242/fft.c

[23] SDSoC Environment User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2_1/ug1027-intro-to-sdsoc.pdf

[24] SDSoC Optimization. https://forums.xilinx.com/t5/Xcell-Daily-Blog/Adam-Taylor-s-MicroZed-Chronicles-Part-89-SDSoC-Optimization/ba-p/639082

[25] El algoritmo de la Transformada Rápida de Fourier y su controvertido origen. http://www-elec.inaoep.mx/~jmram/cvjmr/El%20algoritmo%20de%20la%20FFT%20y%20su%20controvertido%201998.pdf